SILENUS - A FEDERATED SERVICE-ORIENTED APPROACH TO DISTRIBUTED FILE SYSTEMS

by

MAXIMILIAN BERGER, B.S.

A DISSERTATION
IN COMPUTER SCIENCE

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

DOCTOR OF PHILOSOPHY

Approved

Michael Sobolweski
Chairperson of the Committee

Noe Lopez-Benitez

Michael Shin

Per Anderson

Accepted

John Borelli
Dean of the Graduate School

December, 2006

ACKNOWLEDGEMENTS

I would like to thank my Advisor Dr. Michael Sobolweski. This thesis could not have been done without his permanent effort to keep me on focused on the goal. I would also like to thank Dr. Noe Lopez-Benitez, Dr. Michael Shin, and Dr. Per Anderson for serving in my comittee. They could provide me with a different viewpoint.

A very special thanks goes to Barbara Hartmann for being the greatest person on earth and for making our relationship work half way around the globe for three years.

I would also like to thank my best friend and roommate Nathan Larson. He was always there to support, no matter how crazy the idea.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

File storage in computer systems has to be reliable, fast, and available over the network. There are several approaches to distributed file systems, which suffer from common problems: They are either very difficult to set up and maintain (such as AFS) or have a single-point-of-failure (such as SMB, NFS).

The Federated Service Oriented Computing Environment (SORCER) provides a framework for dynamic network services. It promises support for providing reliable, autonomically deployed services.

Thus questions to be answered are:

- Can a dynamic distributed system such as SORCER provide the stability and reliability that is needed to provide a file system for metacomputing applications?
- Would the additional overhead lead to a severe impact in performance?
- Could users without computer science knowledge use such a system?

CHAPTER 1. INTRODUCTION

Storage of data has always been an issue in computer science. Saving your data to a hard drive is easy and convenient. Unfortunately, data saved to your hard drive is not safe. There are several potential problems:

A primary problem of data storage is data theft. Nowadays this has become one of the most important issues, but unfortunately, it is still overlooked by many developers. On most PCs, a person sitting directly at the computer can access any data. For this situation it is very unlikely that one or your competitors would walk into your office and turn on your computer. Nevertheless, think about how many people actually have keys to your office: your co-worker who may not like you, a housekeeper who is underpaid, and so on. Even if your data is stored on a server, any system administrator can usually access any stored data.

The second and most noticeable problem is that of computer failure. Computers are not, and will never be infallible. In fact, at any given time only 80% of all hosts on the network are working. Imagine having an important report on the server and not being able to work on it, because it is down. There are different possibilities for failure: planned maintenance, unplanned outages, network failure or server failure. Most of the times these failures are temporary, in which case they are just annoying, but sometimes these failures are permanent. In this case, one can only hope that you have a recent backup.

These are just two examples of the problems with today's file storage systems. Both of which can be solved using much energy and thought. A server could be put in a secure room with an alarm system where only one person has access. There could be multiple network connections, multiple servers, with fail over, a daily backup system, and so on. Nevertheless, solving these issues is very involving and requires a lot of maintenance. Smaller companies or even home users will not take the necessary precautions to protect their data.

Seeing this, there must be an easier way to manage data files. A simpler method must exist, that enables the average user to take advantage of the networked world, without buying expensive hardware or hiring an expert. This, however, calls for a new paradigm in networked computing.

Paradigms of computer networking have changed over time. When the first multi-user computers where introduced, they used the server-client paradigm. One large server would handle all the time-consuming tasks, and multiple, so-called "dumb terminals" did nothing but interaction with the user. Should the server fail, no users could work. The next big trend in the computer industry was the personal computer. Instead of being dependent on other hosts, now each user had their own personal computer. Failure in this case would result in this person's data could be lost, but no one else would be affected. Handling many of theses systems was a difficult task for administrators. They had to physically sit at the computer and disturb the user for each maintenance task. Therefore, people began networking their personal computers. They went back to the client-server paradigm for some items, such as storage space, and used their personal computers for other items, mostly computation. This is the current state in most computer networks around the world.

Another networking paradigm has emerged in the current decade. It is called peer-to-peer. In a peer-to-peer architecture, each client application is also a server and each server application is a client. These applications are known as servents [47]. The main idea is that instead of just consuming resources, like a client, or offering services, like a server, a host will do both. Peer-to-peer software is used mostly in file-sharing networks, such as bit-torrent. Instead of downloading a file from a single location, a user can now download a file from every other user that already has this file. This saves bandwidth and can vastly improve performance. Unfortunately, peer-to-peer networks have a bad reputation due to most of the content found in these networks if copyrighted material and should not be shared in the first place. However, peer-to-peer networks are a very recent technology and an active area of current research.

The fourth and most advanced network paradigm is the one of service-oriented computing. Peer-to-peer is already an advanced step, but why stop there? In service-oriented computing, a service provider exists on the network. It could provide computational power, storage space, or any other resource available on the provider. Moreover, most important: it's location does not only matter, it may even change. If a host becomes unavailable. The software for running a service does not need to be installed on a computer. Any computer joining the network can automatically pick up services, and provide them to all other computers. This provides a very dynamic and fail

proof network. SORCER, developed by Dr. Sobolewski at the Texas Tech University, provides a framework to support service oriented computing. There have been several thesis's researching the distribution of computational power, but so far none concerning the distribution of storage space.

The questions to be answered are: Can a dynamic approach, such as service-orientation, provide the reliability and stability required for a file system? And if so, how can this be done? There are currently no existing file systems that use a pure service-to-service approach.

To answer these questions, the SILENUS system was designed and built. SILENUS is a distributed file storage system that is secure, failsafe and easy to use. It uses a new approach: File storage should be a service where the user does not need to know where, when, or on which hosts the actual file data is stored. Files are automatically replicated and migrated. Data is encrypted and available only to authenticated users.

The SILENUS system introduces a new model for file storage. This model splits up the file system into several independent services. There are gateway services, to support existing applications. Data storage services store the information in the system. Management services keep an overall overview over the system and provide optimization. Each service can exist multiple times in the network, is independent, and federates whenever a request is made.

Problem Statement

To design a revolutionary new distributed file storage solution providing

- The file system core features as defined in the section called “File system core features”.
- The architectural qualities as defined in the section called “Architectural qualities for distributed systems”.
- The use cases defined in the section called “Use Case Roles”.

Which is done using a newly formed model, for this application, as described in Chapter 4, *Architecture and Design*.

Dissertation Outline

This dissertation is divided into six chapters. Chapter I introduces the problem of storing data. It then gives an overview over this dissertation.

Chapter II describes the background research and literature review necessary to understand the problems and the proposed solutions in this dissertation. It is written in two parts. The first part describes existing file storage solutions. It examines their advances and disadvantages, and how they relate to SILENUS. The second part looks into different qualities for distributed systems.

Chapter III describes the detailed objective of the SILENUS solution. The exact requirements are extracted from the knowledge about existing solutions and their shortcomings.

Chapter IV proposes a new model based on the concepts and technologies investigated in the background research. It describes an overall system architecture based on the service-oriented paradigm. It then describes the design of the individual components.

Chapter V describes a prototype based on the newly created model proposed in Chapter IV. It looks at a specific implementation of the proposed solution. It will describe details and algorithms that were necessary to solve problems that will appear during the implementation. It describes which test cases where used to validate the proposed solution. It also describes the operational validation of the prototype. It will show how the prototype was deployed and tested.

Chapter VI summarizes the dissertation. It provides an overview over the lessons learned. It compares the new solution with the existing ones. At the end, it will describe further work and research directions.

The appendix contains the technical reference for the prototype implementation. It shows the actual usage of the interfaces that where designed for the implementation.

CHAPTER 2. BACKGROUND AND LITERATURE REVIEW

The literature review and background includes two parts: In the fist part, existing distributed file storage systems are looked at and analyzed. Their content will provide a solid base for the current state of the art. In the second part, different techniques and approaches are investigated. This is needed to make a good decision on which approaches to choose for the actual design.

Existing model for remote file storage

To develop an architectural model for SILENUS it is necessary to look at existing models for distributed file systems. Coulouris describes a basic model for distributed file systems in his book [75].

Coulouris describes a basic model for a file service architecture consisting of three components: a flat-file service, a directory service, and a client module. In this model the flat-file service and the directory service export an interface to the client module. The client module maps the calls from the local operating system to calls to the file system.

The flat file service is concerned with operations on the contents of files. Files are identified by UFIDs, which are unique identifiers for all files in the file system. When a flat file service is asked to create a file, it creates an UFID.

The directory service provides mapping from textual file names to their UFID. Clients can obtain an UFID by asking the directory service for a given filename. The directory service is responsible for creating and browsing directories. Directories can hold references to other files and directories.

The client module runs on the client computer. It makes both the flat file service and the directory service available to the client computer under one interface. It provides an adapter for the operating system file functions to calls to the distributed file system. The client module is responsible for archiving performance through caching.

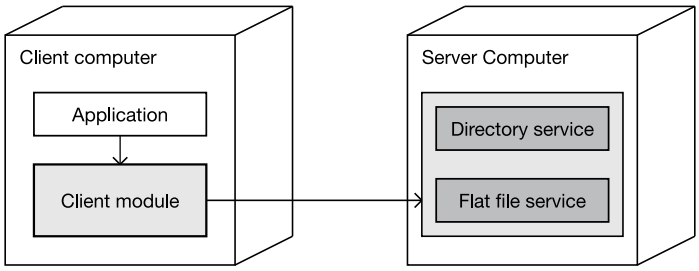


Figure 2.1. File service architecture according to Colouris

Model functionality

In Coulouris model, both the flat file service and the directory service provide operations for the client module.

The flat file service provides support for the read, write, create, delete, getAttributes, and setAttributes operations. Read and write are used to read and modify file content. Create is used to create new files, while delete is used to remove existing files. Get- and setAttributes are used to read and write file metadata.

The directory service provides the lookup, addName, unName, and getNames functions. Lookup is used to retrieve the UFID for a given filename. AddName and unName are used to add and remove files to and from directories. GetNames provides a way to list file contents.

Additional terms

Coulouris also defines two additional terms for distributed file systems: Hierarchic file system and file grouping. A hierarchic file system provides a tree-structure for files. File grouping defines a collection of files that should be seen as one unit.

The directory service already provides support for a hierarchic file system, since each directory can contain files and other directories. This allows access to a specific file by the use of a pathname: A multi-part filename that describes the path through the tree. The root node has to be represented with a distinguished, well-known name. Lookup can then be provided for files based on their pathname.

A file group is a collection of files on a server. Each server may host different file groups, and file groups may be migrated from server to server. Therefore, file groups have to be identified with a file group identifier. A file group identifier must be unique in the network.

Shortcomings of the traditional model

The Coulouris model has several shortcomings. The first shortcoming is in the distribution: Even though three components are identified, two of them have to reside on the same host. Some of the functionality is duplicated in the client module. Another shortcoming is the special importance of the file name attribute as opposed to other attributes. Directories may not have any attributes at all in this model.

Coulouris distinction of three modules is a step in the right direction, but the restriction that two of them have to reside on a server host and one of them has to reside on a client host seems rather artificial. Splitting up the service architecture in two modules provides several advantages. One of these advantages is scalability: If two components provide distinct services and do not rely on each other, they can be run on different server hosts. The Coulouris model does not use this advantage since it is based on the classical client-server model.

In Coulouris model, some of the functionality of the server modules has to be duplicated in the client module. Coulouris states that the client module has to provide caching support. This requires keeping a local directory service and a local flat file service. These local services are minimalized version of the ones available on the server, and are therefore integrated in the client module. If the directory service and flat file service would be more generalized, they could be re-used in the client module.

The attribute handling in Coulouris model is inconsistent. Coulouris defines a special filename attribute that is to be used with the lookup service. This disallows the use of the lookup service for other attributes, such as searching for a file by creation date. For such a search each file would have to be first identified in the directory service, and then its attributes retrieved from the flat file service. Directories may not have attributes at all. They exist only in the directory service, which has no provision for querying attributes.

Existing network file storage solutions

Before developing a new solution, one has to look at existing solutions. For once, they might provide very good hints on what is done and what is still missing, but for many people these existing solutions might already provide all the features needed.

This section will look at different existing network file storage solutions. Single computer solutions are skipped, as this dissertation is about distributed data storage. Different solutions will be looked at in the order of their complexity and the amount of functionality they provide.

When looking at these file systems three types of file systems have to be distinguished. The first type provides remote access to files, but these files exist in one place only. NFS and CIFS are examples of such file systems. The second type provides file replicas, providing better access and higher availability. AFS and Coda are examples of replicated file systems. The third type of solutions is data grid solutions. These provide full data management, mostly for high-performance computing applications. Globus GridFTP and the Avaki data grid are example solutions.

Non-replicated remote file systems

Non-replicated remote file systems are network file systems where the actual data exists in only one place on only one host. This usually means much less overhead, and simplicity. However, it also means less safety in the case of failures. If the host that contains the file is unavailable then the file will not be available.

Network File System (NFS)

NFS is the most widely used network file system in UNIX environment. It was originally developed by Sun but is now available on almost any UNIX or UNIX-like operating system. It is implemented as a set of remote procedure calls (RPC). It provides only host-based authentication and only suggests obeying use permissions. File locking was not possible until version 3 and still provides problematic between different operating systems. Newer implementations of NFS provide a little more security, but these are less used do to incompatibilities with other operating systems. NFS is not reliable in the case of network failures: The administrator can chose between "fail" after a certain timeout or "hang forever". Despite all shortcomings, NFS is a very fast network

file system with very little overhead. It works very efficient in local area networks (LAN). NFS mounts can be read-only cached for improved performance. Migration of data is impossible: Data is referenced by the server name and the location on the server. [1, 2]

Migration and replication of data has been added to version 4 of the NFS protocol. Unfortunately, this specification is still new, and so current implementations are limited. Many existing clients are now just having a fully working NFS v. 3 implementation. Even in NFS 4, a server must still be available to tell clients about the new location of their data. [5]

Common Internet File System (CIFS)

IBM developed CIFS under the name Server Message Block (SMB) protocol. If was then re-used by Microsoft as their network file system protocol and then later renamed to Common Internet File System (CIFS). It this system a user connects to a specific storage on a specific server. Then she can use the remote disk space like any local disk space. Locking and authorization are provided. The biggest drawback of CIFS is that it provides user-based authentication only. An administrator cannot mount a file system for all users and give them different permission. Despite other claims, CIFS is very secure: Since every user has to authenticate, there is no need to trust the client computer. User administration is needed on the server only. CIFS is the most commonly used file system protocol in the windows world. It even provides browsing for available shares. CIFS file systems can be easily migrated to different locations on the same server host but not across multiple hosts. [14, 13]

Despite of these drawbacks, single replica file systems are still the most common used. The main reason for that is their sheer simplicity. Any new remote file storage will have to compete with that. Even though solutions that are more sophisticated are available, most UNIX-like systems still use NFS, and most Windows-systems use CIFS.

Replicated file systems

Replicated file systems keep their data on more than one server. There will always be multiple copies of each file. The advantage is that now only one of the host has to be available. This helps to provide availability in the case of hardware and network failures. Multi-replica file systems are more sparsely used. They require a substantial amount of administration.

Andrew File System (AFS)

The Andrew File System was originally developed at the Carnegie Melon University (CMU). It was then continued by IBM, and eventually made open source. AFS was intended as a replacement for NFS on UNIX hosts. However, the AFS software is available for all common operating systems. AFS has a wide variety of features: The user does not need to know where the physical file is, only the address of an AFS master server. The master server and all data can be replicated. Replicas are usually read-only, but can be made the upgraded to the master copy in case of a permanent failure. The client software usually creates caches the data locally, giving better performance. AFS security is handled via Kerberos, which is a common standard for authentication. AFS data is not encrypted. The number of replicas of a file depends on what store the file is in.[48]

AFS is a very good distributed file system. Many larger organizations such as large companies and Universities use it. One major drawback used to be high license fees, which has disappeared since the software was made available as open source. The biggest problem with AFS is the time it takes to set up. Configuration is very complicated. It is easier if a Kerberos server is already in place, but it will still take a long time. This makes AFS unusable for the small work group or the home user.

Coda

Coda is also developed at the CMU. It is based on the code of AFS. Coda provides additional features: read-write replicas and hoarding. Coda even has conflict resolution: Should the network connection between two servers fail while two clients are writing on them it will automatically detect conflicts and provide both files. Coda also requires OS support, which is only available for a limited number of operating systems.

Coda is still in an experimental stage, and not recommended for production use. Code provides some kind of security; unfortunately, some of it has been cut out due to the encryption export restrictions of the USA.

Coda provides very interesting features: Especially the disconnected operation and automatic conflict resolution in code is very sophisticated. Unfortunately, the setup of Coda still requires a lot of manual administration.[15, 16]

Data grid solutions

Data grid solutions try to provide common data for computation intensive, distributed applications. They usually require specially written applications to function properly.

Globus file store

The file storage system in Globus was invented from a different viewpoint. While the others tried to supply a file system to all legacy applications, the Globus system tries to supply efficient file storage to new applications, which are specifically written for the Globus system. The Globus system is used to for distributed computing. Since this usually involves large data sets, the focus here was on performance. Files can be downloaded from multiple sources to prevent server overload.

The Globus file storage system has very good ideas. The main drawback is its incompatibility with legacy applications and that it never was meant to be a file system for legacy applications. [17, 18, 19, 49]

Avaki

Sybase Avaki Enterprise Information Integration (EII) provides a comprehensive grid data management solution. It stores data at different locations but provides one common interface to the user application. It combines data from different sources on different hosts and different locations in a unified view.

Avaki does not use replicas for redundancy. It provides cached replicas to support faster access. Administrators may even write to these replicas. The replication process is optimized for already existing fast and reliable network infrastructure in high performance computing labs.

Avaki originally started out at the University of Virginia under the name Legion. It was commercialized in 2000. Sybase bought Avaki in 2005. It was then integrated into their line of data oriented services.

The original Legion software was seen as a grid portal rather than a data management solution. It provides unification of different data sources and access through the same interface. The common interface makes Avaki interesting.

A major drawback of the Avaki software is its cost. Being commercial software, the initial costs are very high. The software requires an existing, reliable infrastructure. As such, it may be good for larger organizations but is unfit for the end user.

The Avaki software is unable to handle disconnected operations. Accessing data from its original source means that the original source must be available: the network must be working, the host must be up and the software must be running. All these assumptions can only be made in a very controlled environment that hardly exists outside of lab conditions. [20, 21, 22, 23, 24, 50]

Other existing file storage solutions

The solutions described here are some of the most commonly used distributed file systems. There are several other file systems that each try to solve a specific question. The Lustre File system is a file system developed for high-speed, robust file access in a cluster computing system [25, 51]. Google has developed a propriatary file system that is used for their internal data storage. It is geard toward their specific needs of storage and high-speed access [26]. These are just two examples of other specialized file systems. These file systems have in common that they are optimized for a specific need and are not intedend for general use.

File system core features

A set of distributed file system core features can be defined based on the analysis of the existing file storage solutions. Table 2.1, “File system core features on remote file storage solutions” shows these features and gives an overview of the existing remote file storage solutions:

Feature	NFS	CIFS	AFS	Coda	Globus	Avaki	SILENUS
Remote access	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Migratable on the same host	>= 4	Yes	Yes	Yes	Yes	Yes	Yes
Migratable onto another host	>= 4	No	Yes	Yes	Yes	Yes	Yes
Replicated	No	No	R / O	R / W	R / O	R / W	R / W
Self optimizing	No	No	No	No	Yes	Yes	Yes
Self managing	No	No	No	No	Yes	Yes	Yes
Easy install	Yes	Yes	No	No	No	No	Yes
Compatible with existing software	Yes	Yes	Yes	Yes	No	No	Yes

Table 2.1. File system core features on remote file storage solutions

Architectural qualities for distributed systems

When designing a distributed system, several architectural qualities have to be satisfied. First, these qualities have to be identified. Existing solutions have to be investigated. Then possible solutions will have to be proposed.

Transparencies

A good distributed system should provide network transparencies. These transparencies are defined by ISO, however most applications do a poor job of providing all of them. To make SILENUS easy to use, all of these transparencies should be provided: [12, 71]

- Location transparent: it shouldn't matter where the file is stored
- Access transparent: all elements in the file store should be accessible from classical, non-SORCER programs.
- Replication transparent: there should be no difference on what replication the user works

- Failure transparent: the system should still work even if a significant number of hosts are down.
- Read concurrency transparent: multiple users should be able to read the same file at the same time
- Write concurrency transparent: multiple users should be able to write to same file at the same time
- Migration transparent: the system or the user should be able to migrate the physical presence of a file without interrupting any work.

Confidentiality

One of the most important features of any distributed file storage solution is confidentiality. Confidentiality here means that only authorized people are allowed to view the files stored in the system. In a distributed system, this becomes even more important since files are stored on multiple systems. Even an administrator on one system should not necessarily be allowed to view all files stored on a particular device.

The term confidentiality is used in contrast to the usual term privacy. Privacy can have other meanings, where confidentiality is clearer in describing that only authorized people are able to view certain content.

Most of the existing file storage solutions check users' credentials. Once a user is authenticated, she has full access to all her data. Unfortunately, administrators can very often bypass the credential checks. Most systems allow administrators to impersonate any user on their system. While this is a good solution for single systems, where an administrator should have full rights, this can be a problem in a distributed system. Users may very often have administration rights on their personal work computer, but they should not be able to read data from other users on the same network.

Even if the user does not have administrative access, network ports are very often unsecured. In many cases, organizations provide network ports for guests, or students in the case of universities. These public ports can very often be used to listen into traffic on the network. A solution may be not to provide any public ports, but some of them might be outside of the organization: A user might want to access her data over the Internet, and there is no telling who could be listening.

Another security hole is direct access to the storage hardware. Even with no administrative rights, users can very often boot systems from an alternative medium and acquire administrative access. This can be prevented; however, there is currently no defense against someone physically taking a hard drive out of a computer. Making the hardware inaccessible is easily possible in large organizations. The servers would have to be put in a dedicated room with security cameras. Only highly trusted personnel would have a key. All the data will be stored in the server room; no data will be stored on the users' computers. Unfortunately, this solution is impossible for smaller organizations. It also makes redundancy almost impossible to acquire.

Encryption solves the problem of confidentiality: Instead of storing data in so-called plain format, the data is encrypted and then stored. To decrypt the data, a decryption key is needed. These keys are much smaller than the actual data. Current key sizes range from about 128 - 4096 bit. Storing a 4096-bit key takes up only 0.5 kilobytes of space and can safely encrypt several gigabytes of data. Sophisticated methods to secure encryption keys have been developed. Most common are pin-numbers, pass phrases and smart cards. [72]

There are two main types of encryption: symmetric and asymmetric encryption. Both have their advantages and disadvantages.

Symmetric encryption

In symmetric encryption, the encryption and decryption key are the same. The main disadvantage is that no data can be encrypted without the decryption key present. Therefore, no one can leave data in the system for other people to read unless that person has access to the same key. Symmetric encryption therefore requires a lot of trust between involved parties. The main advantage of symmetric encryption is its speed. Symmetric encryption with short key length can be done very fast. The most widely used symmetric encryption algorithms are DES, blowfish and AES. DES and AES where standardized by the U.S. government for use in commercial applications. [7, 8]

Asymmetric encryption

In asymmetric encryption, the encryption and decryption keys complement each other. Data can be encrypted with one key, and decrypted with the other. The main advantage here is that the encryption key can be made public: It is almost impossible to calculate the decryption key from the encryption key. This is by far more secure than symmetric encryption: The encryption key can be made public knowledge. Unfortunately, asymmetric encryption is by far slower than symmetric encryption and requires longer key length. The most widely used asymmetric algorithm is RSA. [27]

Encrypting decryption keys

Both symmetric and asymmetric encryption can be combined: In current applications, each individual data file is encrypted using symmetric encryption with a random encryption key. This encryption key is then encrypted using asymmetric encryption with the users' asymmetric key. The encrypted symmetric key is then attached to the data file. This method combines the speed of symmetric encryption with the security of asymmetric encryption. It also allows files to be available to a group: The symmetric data key is simple encrypted with multiple asymmetric keys.

This combination has the advantage that a different symmetric key can be generated for every stored item. The encryption keys do not repeat, so a smaller size key can be used. If the encryption on a file is broken, one that one file will be compromised. Smaller keys allow for greater speed and flexibility.

The second advantage is that a user can physically carry the secret asymmetric key. It could be saved on a disk, USB key, smart card, or some other small device. This allows the data to be encrypted on the users' computer. It will not be sent unencrypted through a public network. It will never be decrypted on the computer responsible for the actual storage. Thus, administrators and eavesdroppers will not be able to view any data they are not supposed to.

Existing cryptographic libraries

Instead of relying on a certain implementation, it is important to rely on a cryptographic library that has exchangeable algorithms. Cryptographic algorithms come and go. What is considered safe today may be considered flawed in the near future.

To cope with this, the algorithms themselves should be exchangeable. Cryptographic libraries provide support for multiple algorithms. The most common used library for the language C is gcrypt. There are several libraries for Java. Fortunately, Sun has developed a standard for Java cryptographic extensions (JCE). All cryptographic libraries based on JCE are exchangeable. [52, 9]

Global availability

In today's world, uses switch computers very frequently. A user may have a work computer and a home computer. However, the data should also be available at colleagues work computer, a friend's computer, or at a computer in an Internet café halfway around the world. Nevertheless, not only full computer systems, but also smaller devices such as cell phones and PDAs are now connecting to the Internet. A user's data should not only be restricted to the use of desktop computers, but should be available on any device anywhere.

In most cases, the users will not have the necessary administrative rights to install file system drivers. In some cases, like the home and work computer, this is no problem. However, installing software in an Internet café is usually not possible. Therefore, any file storage solution must be able to work with existing operating systems and applications.

Providing support to existing application is an important feature in remote file storage solutions. After all, it is very unpractical to store data and not being able to use it with existing software. Any new file storage solution should provide support for existing application by offering a support for as many operating systems as possible.

WebDAV

The Web Distributed Authoring and Versioning specification (WebDAV) provides a new standard for remote file storage. The name itself is ill chosen: WebDAV has nothing to do with the web, but rather with file storage over the Internet in general. It does not provide version information as the name suggests, but this is added by an extension called DeltaV.

So what does WebDAV specify? WebDAV extends the hypertext transfer protocol (HTTP) with file management function. The original HTTP specification provides support for authentication, uploading, and downloading of files. WebDAV provides additional functions for listing, moving, deleting, and locking files. This provides basic file management functionality. Two extensions to WebDAV provide support for versioning and more sophisticated access control lists (ACL). [3, 4, 6]

The WebDAV standard provides several option levels. Option level 1 provides basic functionality for upload, download and managing of files. Option level 2 provides support for file locking. The DeltaV and ACL extensions provide additional option levels. Each implementer may choose which option levels to implement in their product.

WebDAV support is built into most modern operating systems: Windows and Mac OS X provide native support for WebDAV. Any WebDAV storage can be mounted and used (almost) like a local file system. Both GNOME and KDE provide very good support for data stored in WebDAV. All of these have to be looked at in detail:

All Windows versions since Windows 98 support WebDAV. Microsoft calls it "Web Folders". A WebDAV folder can be mounted like any other file system by going to "My Network Places", selecting "Add Network Place" and then typing in the address in the **http://server/folder** format. The WebDAV folder then appears like any other network folder on the system. Unfortunately, files cannot be edited directly on the server; they have to be copied to a local directory, edited and then uploaded again. Fortunately, many software vendors implement WebDAV support directly into their applications. Among the most notably are the Microsoft Office products and the Adobe Creative Suite.

At the time of this writing Mac OS X has the best built-in WebDAV support of all major operating systems. A WebDAV folder can be mounted like any folder in the Finder under Go / Connect to server. Mac OS X has full read-write support. WebDAV folders can be used like any other local drive.

The only shortcoming of Mac OS X is that the Mac OS file systems store a file in two parts: The actual file, and a so called "resource stream". This resource stream contains additional information, such as the file icon. On non-HFS (the Mac OS native file system) file systems, these resource streams are emulated with files that start with dot-underscore (`._`). Ideally, a file system driver should know about that and emulate the appropriate information.

UNIX users that use the GNOME desktop are lucky: The standard file browser in GNOME is Nautilus, which supports WebDAV folders like any other folder. Simply type the address of a WebDAV folder in the address bar, and you can browse the files. Unfortunately, you cannot open files directly, so you have to do the same as on Windows: Copy the file to a local directory, edit it, and copy it back.

Cadaver is a very simple WebDAV client for all UNIX systems. Its interface is the same as the standard command-line FTP client found on all UNIX systems. This makes cadaver somehow tedious to use, but makes it highly portable. Use cadaver if you cannot use any of the other methods.

Davfs2 is the project of building WebDAV support as a file system into the Linux kernel. Unfortunately, at the time of this writing this project was still in beta stage. [53]

Web-based access to file storage

A web application framework provides the infrastructure necessary to run applications over the Internet. Traditional web servers have support for static web pages only. Web applications however require interactive content. Some solutions work on the client. Client-side Java, Java script and Active-X are the most common examples. These solutions, however, require special support and software installed on the users' computer. Other solutions run the application on the server. They provide a user interface by providing HTML pages and using HTML forms for interactivity. They may use client-side software, but do not require it. These solutions provide more security. Users do not need to run applications on their own host. Examples of such technology are Java Servlets and Java Server Pages.

Java Servlets and Java Server Pages (JSP) allow the provision of dynamic content on web pages. Traditional web pages are static and have to be manually updated on the server side. With server-side technology, such as Servlet and JSP code can be executed whenever a website is requested. This enables dynamic web applications such as web shops. While static web pages can be protected by authentication, the pages served if authenticated are always the same. Dynamic web pages can provide different content to different users. They may also add special request and response codes to the web page. [61, 62]

James Gosling first thought of Servlets in 1995. Later Pavani Diwanji picked up the concept and created Servlets that would eventually be part of the Java Web Server. James Davidson wrote the first Servlet specification. Java Server Pages were conceived by Anselm Baird-Smith, and later specified by Satish Dharmaraj in 1999. [63]

A Java Server Page is a shortcut version to a Servlet. Most Servlet just wanted to add a little dynamic content to an already existing web page instead of creating a completely new page. A JSP is a small part of Servlet code that is added in an otherwise valid HTML page. It is executed and its results are added right there into the page. It is usually a good compromise between just code (Servlet) and just content (HTML).

The big advantages of Java Servlets and Java Server Pages are the dynamic nature and the large existing software library. Java Servlets allow dynamic content to be created. They may go from as little as just one line of code to reprogramming the HTTP protocol and adding new network commands. There are several solutions for dynamic web applications. JSP and Servlets, however where not just invented for dynamic web applications, and can therefore fall back on a large library of existing software packages. In addition, since they are Java based they work on almost any web server platform.

As with all interpreted programming languages, there is a performance loss. This may not be so significant on a single-user system but on a web page with millions of hits every day, this is an issue. Fortunately, the Java interpreter provides extensive run-time optimization with its Hot-Spot engine. Nevertheless, Java Servlets will always use more memory and CPU than native applications would.

Disconnected Operation

Ideally, the Internet would be available everywhere on the world through a high-speed connection. Unfortunately, this is not the case yet. On the other hand, human expectations are more and more global. Data should be available everywhere whether connected to the network or not. Increasingly users want to use mobile devices, such as laptops. A distributed file storage system should have support for accessing files offline.

Even in places where the network is usually available, there are still many network outages. Wired networks at any organizations fail at some point in time. In this case, a distributed file system should not loose any data. It should still provide support for saving and accessing cached files.

The first step to provide support for disconnected operation is to except disconnection. Many existing systems assume that the network is reliable, as stated in the section called “Eight fallacies of distributed computing”. Instead, the exact opposite should be expected: Each host works independent, and uses data from other hosts if available. If not, it should carry on.

Each node will still have to collaborate with other nodes. They need to provide a synchronization mechanism. This synchronization mechanism should not depend on any global state, but rather detect the states of the nodes automatically. It should then try its best to synchronize the data in the two nodes.

Sometimes disconnection is predictable. In this case, a distributed file system should provide support for hoarding. A user may decide to work on certain files at home. She plugs her computer in at work, selects files for offline work. After a while, these files are made available on the users' computer for offline usage. Whenever the user connects back to the network, the files are synchronized with the rest of the file system.

Manageability

As soon as a system grows larger, or it has been used for a while, it becomes more difficult to manage. In the case of a file system, this means many files, from many users, on many hosts. Several problems arise here.

Managing many files is mostly the task of migrating and replicating them among multiple hosts. Files should be available on multiple hosts for safety. They should be available on different hosts to not overload a single host.

A large base of users is another manageability challenge. Each user should have access to different files in the file system, and only to these files. User access rights have to be managed. One single person cannot do this; there must be a way to delegate access rights to local administrators.

Hardware failure and adding hosts is a managing problem. When a host fails, all the files that where on this host will have to be moved to other hosts. To do so, they should have been backed up or replicated to another host beforehand. When a new host becomes available, files have to be moved to this host to utilize this new host.

One way to provide better manageability is to use federated services. In a service-oriented approach, each host provides services. Services are automatically discovered and used when they are available. These services can easily be moved from one host to another.

Some of these federated services are autonomic optimizer services. These services can make the decisions a human administrator would make. They can check the current available resources and make sure they are used according to the policies set by an administrator. Since federated services are loosely coupled, different optimizer services can be added and removed based on the needs of a particular system.

Scalability

Another problem arising from a larger file system use is that of scalability. A system should still perform well, no matter how many hosts, files, and users it serves.

Scalability can be achieved by distributing services across multiple hosts. If a service is available on only one host then this host will eventually be overloaded. By making it possible to have services available on as many hosts as needed, scalability can be provided by adding extra hardware.

A paradigm switch has to be made from client-server to federated services. Classical client-server solutions do not provide good scalability. They depend on a single server. As soon as the number of requests increases, so does the load on the server. Federated services, on the other hand, provide a way to load-balance the system. Instead of sending all requests through one server, the same functionality can be provided by many services. A requester can pick a service with a low load. Should all services be overloaded, an administrator can add extra hosts.

Reliability

A quality that is particular important for file systems is reliability. A file saved into a file system should stay there until deleted. Files should never disappear or get lost. Unfortunately, most existing file systems move the responsibility for reliability to the underlying hardware. Should the hardware fail, the files are lost.

Reliability can be achieved by replication. In the case of a distributed file system, this means replication among different hosts. Every file that should be stored reliable needs to be available on at least two hosts at any time. Should one host fail, there is still another copy available. There should be another backup copy of that made as soon as possible to provide reliability again.

Modifiability

Software systems are never stable. They evolve into newer systems. There are two main reasons a software system needs to evolve: bug fixes and new features.

Every software has bugs. Humans write software, and humans make mistakes. Even the best computer scientists make mistakes [54]. Therefore, no matter how well software is written and tested, it will always need to be updated to accompany new bug fixes.

After a while, users grow tired of an existing system and demand new features. Maybe a new device just came out, but the current computer system does not support it. Maybe different people who have a different focus and want different features now use the system. In these cases, the system needs to be updated to add new features to it.

Dynamic code loading helps to provide modifiability. When an update is available in classical systems, an administrator has to manually download and install this update. This works well on a single host, but is very hard to manage for multiple devices. It is even worse if there are multiple administrators, but a new version has to be rolled out immediately. With dynamic code downloading, the software checks for a new version and downloads it whenever it starts. Rolling out a new version is as easy as publishing a file on a server. All that is needed is for the modified parts of the software to be reloaded. This may also be triggered from the network. With dynamic code downloading, system-wide administrators can assure that all nodes have the latest version.

Platform independence

Existing computing devices use a wide variety of processors and operating systems. Supporting each of them with a custom solution is a major undertaking. An easier solution is using a virtual machine. An application would have to be written for that virtual machine. Only the virtual machine has to be ported to different platforms.

The programs are compiled into byte code. This byte code can be reused on any of these virtual machines. This makes code mobility possible. The most commonly used virtual machines are the Java virtual machine and .NET.

An example virtual machine specification is the Java virtual machine (Java VM). Originally specified by Sun it is now being developed through a community process. Byte code that is compiled for a certain version of the Java virtual machine will run on any JVM that complies with these specifications. Example Java VM implementations are provided by Sun, IBM, Apple, and several open-source development teams. [10, 55, 56, 57, 58]

Originally intended to run on home appliances, the Java VM is now available on all modern desktop and server operating systems. The Java environment provides both an object-oriented language and a runtime system. The language is similar C++, which used to be the most widely used programming language. The runtime system provides the same functionality across all platforms. Java is a true write once - run everywhere language. Even modern mobile devices, such as personal digital assistants (PDA) and cell phones now provide support for the Java platform. In the heterogeneous environment of the Internet, there is almost no way around a platform independent runtime system like Java. [59, 60]

Java also provides many built-in libraries. Unlike traditional programming languages, the Java standard requires a wide range of standard features. If a given Java runtime version is installed on a particular host, all standard libraries will have be included.

Service Orientation

Service oriented architectures provide most of the given architectural qualities for distributed systems. It is therefore necessary to investigate service orientation and understand how it functions.

Eight fallacies of distributed computing

To understand the motivation behind the service-oriented paradigm the common fallacies of network computing have to be investigated first. Peter Deutsch defined eight fallacies of network computing as follows: [66]

> Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause big trouble and painful learning experiences.
> 1. The network is reliable
> 2. Latency is zero
> 3. Bandwidth is infinite
> 4. The network is secure
> 5. Topology doesn't change
> 6. There is one administrator
> 7. Transport cost is zero
> 8. The network is homogeneous

In a service-oriented system, none of the assumptions is made. Instead, it is always assumed that these eight points are false.

Generations of Remote Procedure Calls

Two different levels of network communication exist: protocol based network communication and procedure based network communication. Procedure based network communication has evolved in the recent years.

Protocol based network communication concerns itself with direct input and output communication. Applications read and write raw data, usually through network sockets. The protocol has to be exactly specified. This is a very low level form of network communication and it is very error prone.

Instead of focusing on the language, it is more desirable to focus on invoking a method on the remote host. This is the idea of procedure based network communication. Procedure based network communication introduces a remote procedure call (RPC). This provides the programmer with a higher level network programming.

There are six generations of RPC specifications. The first generation is that of Sun RPC and others. It defined a protocol for support remote procedure calls that are language, architecture, and operating system independent.

The second generation of RPC, of which CORBA is an example, introduced support for objects. The original RPC specifications where written before the object-oriented concept was fully developed. Once object-orientation became more common, a new generation of RPC protocols was needed.

These RPC specifications made it possible to call existing code on remote hosts. The third generation of RPCs, such as Java RMI, introduced behavioral transfer. Instead of just calling a method on a remote system, actual behavior in the form of code could be send to another system for execution.

The next and fourth generation of RPC, introduced by Jini JERI, uses dynamic proxying. In previous generations, a precompiler would have to be used to generate network stubs and skeletons that wrapped the network calls for the user. With dynamic proxying, no preprocessing step is necessary. Any existing object can be exported and made remotely accessible.

The fifth generation of RPCs is the generation of web-services. Web-services use an XML-based protocol over HTTP for communication. This allows for services to be deployed using existing web-server installations.

The sixth and most current generation of RPC is the service oriented program, which is provided by the SORCER framework. Instead of communication with one specific server, a method invocation can be executed by any host that runs a matching service. A single invocation may even span multiple hosts. These hosts will federate together to provide the requested service.

Service Oriented Architecture

Instead of thinking of a service offered by a particular host, the paradigm shift should be towards services in the network — *the computer is the network*. In classical distributed applications, it is necessary to know exactly on which host a particular service is exposed. In most distributed file systems, for example, it is necessary to know the name of the host that stores a particular file. In a service-oriented environment, a service provider registers itself with a service registry. The service registry facilitates lookup of services. Once a service is found, a service requester binds to the service provider and then can invoke its services. Requesters do not need to know the exact location of a provider beforehand. Instead, they can find it dynamically. They discover a registry and then lookup a service. On the other hand, a provider can discover the registry and publish its own service, as depicted in Figure 2.2, “Discovery in Service-Oriented Architecture” and Figure 2.3, “Execution in Service-Oriented Architecture”.

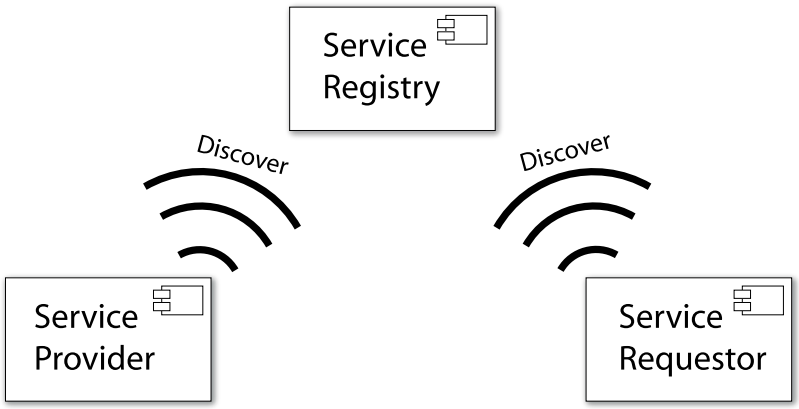


Figure 2.2. Discovery in Service-Oriented Architecture

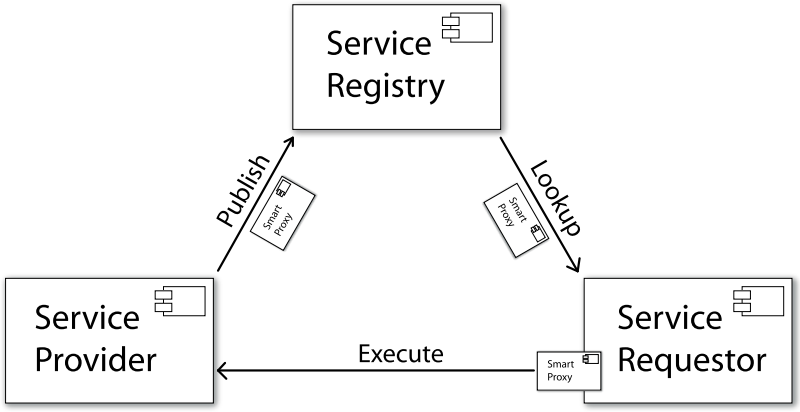


Figure 2.3. Execution in Service-Oriented Architecture

A service is identified by an interface (type) rather than its implementation, protocol, or name. If a service provider registers by name, the requesters have to know the name of the service beforehand. Registering services by interface has the advantage that the actual implementation can be replaced and upgraded independently from the requesters. Different implementations may offer different features internally, but externally have the same behavior. This independent type-based identification allows for flexible execution of service-oriented programs in an environment with replicated services.

A service-oriented program is composed of tasks, jobs, and service contexts. Figure 2.4, “Service oriented Tasks and Jobs” shows an example of service tasks and jobs. These concepts are defined differently than in classical grid computing. A service job is a structured collection of tasks and jobs. A task corresponds to an individual method to be executed by a service provider. A service context describes the data that tasks works on. This approach is different from classical grid computing, where a job corresponds to the individual method. In UNIX analogy, the individual tasks correspond to UNIX programs and commands. The context would be the input and output streams. A job corresponds to a shell script or a complex command line connecting the tasks together. Service-oriented programs can be created interactively and allow for a federated service environment. [32]

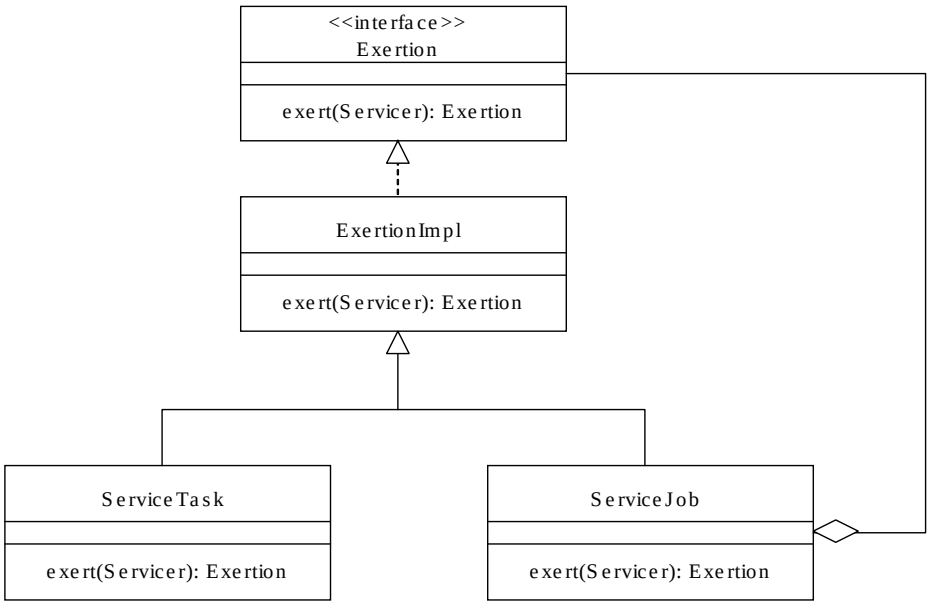


Figure 2.4. Service oriented Tasks and Jobs

In a federated service environment, not a single service makes up the system, but the cooperation of services. A service-oriented job may consist of tasks that require different types of services. Services can be broken down into small service methods instead of providing one huge all-embracing service. These smaller methods then can be distributed among different hosts to allow for reusability, scalability, reliability, and load balancing.

These grid concepts cannot just be applied to computational tasks. They can, and should be, applied to data as well. Once a file is submitted to the network, it should stay there. It should never disappear just because a few nodes or the network segment goes down. In addition, it should not matter what client node is used to request the file. With the SILENUS distributed file system in place, SORCER will also provide reliable and scalable file-based data services complementing the existing method services.

Jini Network Technology

The Jini network technology enables Java software to create dynamic networks that are adaptive to change. Jini uses a Service Oriented Architecture approach to network services. It is especially useful for scalability, evolvability and flexibility. Services can easily be replaced in runtime, started on multiple servers, or even migrated form one computer to another. [28, 74]

Jini technology was originally created by Sun. It was then contribute to the Jini Community in 1999. It is based on an open specification that can be developed through a community process. The reference implementation is provided still provided by Sun.

Jini provides almost everything necessary for service oriented computing, as described in the section called “Service Oriented Architecture”. Jini makes it easy to write services. Each service can register with a service registry. Service registries can be discovered by multicast announcements. Service requesters may use the service registry to find services and use them.

The dynamic nature of Jini is handled with leases. Each network service registering with another network service must obtain a lease. The lease must be renewed in given intervals or it will expire. This allows the detection of unreachable nodes, while putting the actual load on the requesting object, not the provider. Lease times may be adjusted depending on the stability of the network involved. A reliable network can work with higher lease times, while it is very desirable to have shorter leases in unreliable networks to detect disconnection quickly.

Jini also provides a standard to attach user interfaces to services. This ServiceUI standard allows the development of Jini service browsers. A Jini service browser will pick up all the registrars and display their services. If a service has an attached user

interface, the service browser can download and display that user interface to the user without having to install or configure any software locally. One example of such service browser software is the IncaX Service Browser. [11, incax]

Peer-to-peer networking

Another network technology widely used for modern distributed architectures is peer-to-peer networking. In peer-to-peer applications, each peer is equal. Peers communicate through an overlay network directly with each other. This eliminates the classical bottlenecks in client–server solutions.

Unfortunately, peer-to-peer has a bad reputation. It was first widely used by the application "Napster". Users were able to share music files with other users in a fairly fast and reliable way. In the peer-to-peer architecture, files are downloaded from other users rather than a central server. This makes peer-to-peer technology hard to control. It is therefore very often used for illegally distribution of files. Some companies even want to ban peer-to-peer technology because of that. However, peer-to-peer also has many legitimate uses. Most Linux distributions are now released through peer-to-peer technology to save server capacity and increase download speed. Common peer-to-peer applications today include Gnutella, KaZaa, eDonkey, BitTorrent, and JXTA.

JXTA (short for Juxtapose) is a set of protocols that allow any device on the network to communicate and collaborate. JXTA provides an overlay peer-to-peer network that clients can use to communicate with each other. The JXTA protocols are defined language independent. A reference implementation for Java exists and is very stable. [65]

Bill Joy and Mike Clary from Sun Microsystems started the JXTA project originally. The specifications and implementation where then made open-source and available on the JXTA web page.

JXTA focuses on peer-to-peer technology. Discovery in JXTA is made by the provider sending out service advertisements. These have to be sent out regularly for service requesters to find them. So-called rendezvous peers can cache these advertisements. Once a requester has found a service advertisement, it can use the JXTA overlay network to acquire a virtual channel between the requester and the provider. This channel can then be used to send messages back and forth.

JXTA is built for far distributed peers in an unstable network. A cached advertisement may provide a link to a service that has not been existent for a long time. A service must therefore be actually contacted before any assumption about its availability can be made.

When comparing JXTA and JINI the first distinction is the range of its application. JINI is designed for local area networks (LAN) and can be used over WANs with the use of special proxies. JXTA is designed for wide area networks (WAN) and all its network overlay is based on that. Fortunately, these two can be combined: Jini requests can be sent over the JXTA network. This provides the best of both worlds: Fast, optimized local access and reliable remote access via the JXTA network. [29]

SORCER

SORCER is a federated S2S framework that treats service providers as network objects with a well-defined semantics of service-object-oriented (SOO) programming based on the FIPER technology. [30, 31, 32]

Each SORCER provider offers services to other peers on the object-oriented overlay network. These services are exposed indirectly by methods in well-known public remote interfaces and considered as elementary (tasks) or compound (jobs) program instructions of SOO programming methodology [30]. A SORCER program can be created interactively [32] or programmatically (using SORCER APIs) and their execution can be monitored and debugged in the overlay network [33]. Service providers do not have mutual associations prior to the execution of a SOO program; they come together dynamically (federate) for all component tasks and jobs in the SOO program.

Each provider in the federation executes a task, or a job. A special SORCERS infrastructure services called jobber coordinates these jobs [30]. However, a job can be sent to any peer. A peer that is not a jobber is responsible to forward the job to an existing jobber in the SORCER grid and return results to the requester. Thus, any peer can handle any job or task. Once the job execution is complete, the federation dissolves and the providers disperse and seek other SOO programs to join. In addition, SORCER supports a traditional approach to grid computing - like in Condor [34] and Globus [35] style. Here, instead of SOO programs being executed by services providing business logic for

requested tasks, the business logic comes from the service requesters executable program that seeks compute resources on the network provided by grid services. These services in the SORCER grid are as follows: GridDispatcher and Jobber for traditional grid job submission, Caller and Tasker for task execution. [36]

To integrate applications and tools on a B2B grid with shared engineering data, the File Store Service (FSS) [37] was developed as a core service in SORCER. The value of FSS is enhanced when both web-based user agents and service providers can readily share the content in a seamless fashion. The FSS framework fits the SORCER philosophy of grid interactive SOO programming, where users create distributed programs using exclusively interactive user agents. However, FSS does not provide the S2S flexibility with separate specialized and collaborating service providers for file storage, replication, and meta information that are presented in this dissertation.

Eight truth of networked computing

Based on the fallacies given in the section called “Eight fallacies of distributed computing”, service-oriented architectures take into account the following eight truth of distributed networking:

1. The network can fail at any time
2. Network messages arrive in random order
3. The network is always too slow
4. Someone is always listening
5. Hosts get added and removed at any time
6. Every system has its own administrator
7. Moving data costs money
8. There will be any possible combination of OS / Architecture out there. They all want to be part of the network!

Security in existing file storage solutions

In this section the security concepts of different existing storage solutions are looked at. In particular, the granularity of security privileges is examined. Then the different authentication mechanisms are looked at. And last, some privacy features are discussed.

Privileges

Security privileges are usually defined through the underlying operating system. A UNIX server sharing files may only export the UNIX permissions. Therefore, the permissions are examined by their native operating system instead of the actual network file system.

UNIX (NFS, GlobusFTP)

The UNIX permission model is the oldest and simplest of the models examined here. Being simple is not necessarily a disadvantage: It is the easiest permission model to learn and to apply. This has kept this model in use for over 30 years.

The UNIX model defines permission bits for owner, group, and others. The actual permissions are for the first match and not the maximum permissions. A file owner can actually exclude themselves from permissions for read and write. The owner or the administrator may change file permissions.

The administrator in a UNIX system has full access rights to all files and directories. This may be seen as an advantage, as the administrator may need to be able to move files to a different location. It may also be seen as a disadvantage as the administrator may read every personal file. The UNIX system does not provide any confidentiality. Administrators must be trustworthy.

Newer UNIX systems such as Solaris define access control lists (ACLs) to extend the basic permission model. These ACLs allow a more fine-grained permission model. Instead of setting the permissions for one user or one group the permissions may now be set explicitly for any user and any group. ACLs allow very powerful, fine-grained sets of permissions, but are more difficult to manage. [67]

Access control lists are UNIX-implementation specific and are not compatible between different operating systems. As such, extensions for ACLs exist as extensions to the NFS 2 and 3, but they work only from and to Solaris systems. NFS version 4 defines a standard for ACLs, but this protocol is not widely supported yet. Most UNIX and UNIX-like systems still support only the basic UNIX permission model.

UNIX provides the three basic permissions read, write, and execute for files. These attributes can be set for the file owner, a specific group, or everyone else. Read and write are usually honored, while execute is more of a hint to the operating system. The

execute bit specifies that a file contains either binary code or an executable script. Two special permissions setuid and setguid can be set on executable files to allow changing the effective user and group. These special bits can lead to security problems and are ignored in many NFS implementations.

The same three permissions are also defined for directories, but their meaning is a little different than for files. Reading a directory means listing the files in this directory. Write permission for a directory allows adding and removing files and links from that directory, even if this entity does not have the specific rights for the file. The execution bit for directory maps to opening and executing files in this directory, and also listing directory contents. A directory with execute permissions can be traversed. Traversing a directory means being able to access subdirectories and use all rights on that given subdirectory. Since listing files is forbidden, the names of the subdirectories must be known. A special permission bit for directories is the sticky bit. In a directory with the sticky bit only the file owner and the administrator may delete files. This allows the creation of common directories, such as `/tmp` where everyone has read and write access on the directories, but not every user may delete every other users files.

Windows (CIFS)

On Windows the security depends on the underlying file system. The two most common file systems for Windows are FAT and NTFS.

The FAT (file allocation table) file system is older and a remainder from the DOS origins of the Windows operating system. The current version is the FAT32 system; older versions are sometimes called FAT12 and FAT16. The FAT system has absolutely no security features.

The NTFS (NT file system) was developed later for the more secure and robust versions of Windows based on Windows NT. As such, it supports a complete security model. Since all recent versions of Windows support NTFS, this is the model that will now be called the Windows model.

Windows uses ACLs for users and groups. Every ACL may define an allow or a deny bit. To calculate the actual permissions, the allow bits of all groups and for the user itself are combined using a logical or. The same happens for the deny bit. A user has a permission only if the allow bit is set and the deny bit is unset.

This model has received some criticism. If only allow bits are used, a user has the maximum possible permissions. A user that is part of one group may suddenly have more permissions than desirable. This may give a user to many rights. On the other hand, the deny bits cannot be overwritten. If a user is part of a group for which a specific resource is denied, even an allow bit for a specific user will not allow the user to access the resource. This may give a user not enough rights.

Every file and directory on the NTFS has exactly one owner. Only the owner may change the ACL and allow or deny rights to other users. Ownership of files cannot be given to other users.

An administrator may not set file permissions. This is very different from the UNIX administrator model. On Windows, an administrator may take ownership of files and directories. Once the administrator is owner, the permissions may be changed. Administrator may not transfer the ownership to other users. They may therefore not give the ownership back to the original user. An administrator is therefore unable to read the files of a user without the user being able to find out due to changed ownership.

The NTFS knows two sets of permissions: Basic permissions and special permissions. In most cases the basic permissions are sufficient. The special permission may be used if a more fine-grained permission model is desired.

The basic permissions for files are full control, modify, read and execute, read, and write. The permissions for directories are full control, modify, read and execute, list folder contents, read, and write. Some of these permissions include the lesser permissions: A user with full control will also have all the other permissions. The following tables are taken from [76]:

NTFS File Permission	Allowed Access
Read	This allows the user or group to read the file and view its attributes, ownership, and permissions set.
Write	This allows the user or group to overwrite the file, change its attributes, view its ownership, and view the permissions set.
Read & Execute	This allows the user or group to run and execute the application. In addition, the user can perform all duties allowed by the Read permission.
Modify	This allows the user or group to modify and delete a file including perform all of the actions permitted by the Read, Write, and Read and Execute NTFS file permissions.
Full Control	This allows the user or group to change the permission set on a file, take ownership of the file, and perform actions permitted by all of the other NTFS file permissions.

Table 2.2. NTFS basic file permissions

NTFS Folder Permission	Allowed Access
Read	This allows the user or group to view the files, folders, and sub folders of the parent folder. It also allows the viewing of folder ownership, permissions, and attributes of that folder.
Write	This allows the user or group to create new files and folders within the parent folder as well as view folder ownership and permissions and change the folder attributes.
List Folder Contents	This allows the user or group to view the files and sub folders contained within the folder.
Read & Execute	This allows the user or group to navigate through all files and sub folders including perform all actions allowed by the Read and List Folder Contents permissions.
Modify	This allows the user to delete the folder and perform all activities included in the Write and Read & Execute NTFS folder permissions.
Full Control	This allows the user or group to change permissions on the folder, take ownership of it, and perform all activities included in all other permissions.

Table 2.3. NTFS basic folder permissions

The special permissions allow a more fine-grained setting of possible permissions. For files these are: execute file, read data, read attributes, read extended attributes, write data, append data, write attributes, write extended attributes, delete, read permissions, change permissions, take ownership, and synchronize. And the special permissions for directories are: traverse folder, list folder, read attributes, read extended attributes, create files, create folders, write attributes, write extended attributes, delete sub folders and files, delete, read permissions, change permissions, take ownership, and synchronize. The following table is also from [76]:

Permission	Description
Traverse Folder / Execute File	This allows or denies a user to browse through a folder's sub folders and files where he would otherwise not have access. In addition, it allows or denies the user the ability to run programs within that folder.
List Folder / Read Data	This allows or denies the user to view sub folders and fill names in the parent folder. In addition, it allows or denies the user to view the data within the files in the parent folder or sub folders of that parent.
Read Attributes	This allows or denies a user to view the standard NTFS attributes of a file or folder.
Read Extended Attributes	This allows or denies the user to view the extended attributes of a file or folder, which can vary due to the fact that they are defined by the programs themselves.
Create Files / Write Data	This allows or denies the user the right to create new files in the parent folder. In addition, it allows or denies the user to modify or overwrite existing data in a file.
Create Folders / Append Data	This allows or denies the user to create new folders in the parent folder. In addition, it allows or denies the user the right to add data to the end of files. This does not include making changes to any existing data within a file.
Write Attributes	This allows or denies the ability to change the attributes of a files or folder, such as Read-Only and Hidden.
Write Extended Attributes	This allows or denies a user the ability to change the extended attributes of a file or folder. These attributes are defined by programs and may vary.

Table 2.4. NFS special access permissions

Permission	Description
Delete Sub folders and Files	This allows or denies the deleting of files and sub folder within the parent folder. It also true that if this permission is assigned files and sub folders can be deleted even if the Delete special access permission has not been granted.
Delete	This allows or denies the deleting of files and folders. If the user does not have this permission assigned but does have the Delete Sub folders and Files permission, she can still delete.
Read Permissions	This allows or denies the user the ability to read the standard NTFS permissions of a file or folder.
Change Permissions	This allows or denies the user the ability to change the standard NTFS permissions of a files or folder.
Take Ownership	This allows or denies a user the ability to take ownership of a file or folder. The owner of a file or folder can change the permissions on the files and folders she owns, regardless of any other permission that might be in place.
Synchronize	This allows or denies different threads to wait on the handle for the file or folder and synchronize with another thread that may signal it. This permission applies to only multi threaded, multiprocessing programs.

Table 2.5. NFS special access permissions (cont.)

AFS and Coda

Both AFS and Coda provide the same security model. Since Coda is the continued development of AFS it inherits the security model from AFS. The AFS security model has been very successful in organization wide deployment. In this section the AFS model is examined.

AFS uses access control lists with allow and deny bits. The mechanism is the same as on Windows: All allow bits are joined through or, then all deny bits are joined through or and subtracted from the allow bits. The remaining permission bits are the actual permissions.

In AFS, the file owner and the admin may change permissions. There is no need to take ownership of existing items. The disadvantage is that administrators may give themselves read permissions, read a file, and then remove the permissions to cover their trace. The advantage is that administrators have full access, which is needed for moving files and for backup purposes.

AFS does not provide file-based security but only directory-based security. The permissions in a directory are valid for all files in that directory. The rationale for this is manageability. ACLs can lead to very complicated security permissions. Sometimes users forget to set specific permission bits on their files. By reducing it to directory permissions there are much less items to handle. Users must be aware of this though, as moving a file from one directory to another directory will change its file permissions.

The security bits for directories provided by AFS are: lookup, insert, delete, administer, read, write, and lock. Lookup allows viewing anything in this directory and subdirectories. Insert allows creating new files. Delete is the permission for removing or moving files. An administer bit provides access to change the security settings. Read, write, and lock are file permissions that just apply to all files in this directory for reading, writing, and locking against concurrent use.

AFS also provides eight special bits that can be user defined. These bits are named A - H. There are no built-in provisions for these user bits other than retrieving and setting them. They may be used to add additional meta data to the directories. One use may be to emulate the archive, hidden, and system bits from the FAT file systems.

WebDAV with AC extensions

The WebDAV protocol as specified in the original RFC does not have any support for modifying or querying permissions. It was up to the WebDAV provider to enforce permission bits. There are standard provisions for authentication, however. There is also a standard response for permission denied.

There is an extension to the WebDAV protocol to provide access control. This extension was defined later than the original WebDAV protocol in [6]. There are currently few implementations of these extensions.

These extensions provide for querying and setting of permission bits. Enforcing these permissions is up to the server. This allows full backward compatibility to the WebDAV protocol without the access control extensions.

A server implementing the AC extension must allow the grant primitive. It may optionally allow a deny primitive. Whether allow or deny has a higher priority is unspecified. There is an optional deny-before-grant option that may be set to specify this behavior.

WebDAV AC does not specify any special rights for file owner or administrator. It is up to the server whether to grant automatic permissions.

The permission bits for files in WebDAV AC are: read, write, write-properties, write-content, unlock, read-acl, read-current-user-privilege-set, write-acl, and all. Each server may implement some or all of these permissions. The read, write, and all permissions are aggregate permissions. Having these permissions or being denied them counts for all sub permissions. The unlock permission may allow someone that is not the lock owner to break a file lock.

The WebDAV AC security bits for directories are: read, write, write-properties, unlock, read-acl, read-current-user-privilege-set, write-acl, bind, unbind, and all. Write is an aggregate permission for write-properties, bind, and unbind. Binding refers to the creation of files. Unbinding refers to the deletion of files. The all permission aggregates all permissions just as for files.

Security Table

The following two tables give an overview over the file and directory permission bits present in the discussed operating systems and protocols.

Permission	**UNIX**	**Windows**	**AFS**	**WebDAV + AC**
Read	X		X	X
Read data		X		
Read attributes		X		
Read ext. attributes		X		
Write	X		X	X
Write data		X		X
Append data		X		
Write attributes		X		X
Write ext. attributes		X		
Lock			X	
Unlock				X
Execute file	X	X		
SetGUID	X			
SetUID	X			
Delete		X		
Read permissions		X		X
Read own permissions				X
Change permissions		X		X
Take ownership		X		
Synchronize		X		

Table 2.6. File privileges in different file systems

Permission	**UNIX**	**Windows**	**AFS**	**WebDAV + AC**
Traverse	X	X		
List	X	X	X	X
Read attributes		X		
Read ext. attributes		X		
Write attributes		X		X
Write ext. attributes		X		
Create / Delete sub items	X			
Create sub items			X	X
Create files		X		
Create folders		X		
Delete items		X	X	X
Owner delete only	X			
Delete (this)		X		
Unlock				X
Read permissions		X		X
Read own permissions				X
Change permissions		X	X	X
Take ownership		X		
Synchronize		X		

Table 2.7. Directory privileges in different file systems

Authentication mechanisms

Each one of these existing systems uses one of the following three authentication mechanisms: client side authentication, server side authentication, or third party authentication. Each one of these systems has advantages and disadvantages that will be discussed here.

Client side authentication

By default NFS provides client side authentication only. When an NFS share is mounted on a client host, it is the client hosts responsibility to ensure proper access. This is very convenient in a multi user system: Multiple users may be logged on the same client host, but still have different access rights on the server system. The problem is that the client host has to be trustworthy. An illegitimate client can very easy ignore these security measures and give its user full access. The standard NFS protocol is therefore only safe in controlled environments.

Server side authentication

Most systems use server side authentication. Windows with server authentication and WebDAV are two examples. In these systems a user has to authenticate himself with a user name and password at the server where she wants to use the resources. The server will then verify the password and grant or deny access. This is secure if the server is secure and the connection is secure. In some implementations, the password is not send directly, but used to facilitate a challenge response mechanism. In these cases the server side authentication is safe. There are two problems with server side authentication: To create a connection, a user has to provide a password. This is possible in most cases, but it is impossible for an administrator to prepare connections for users. It also disables multiple users on the same host from reusing the same connection without introducing security holes. The second problem is that the password is transferred to the server. The password may be intercepted on the way or on the server. Users also tend to store their password on the client host to make it easier for them to access shared resources. These client hosts are usually not as well protected as servers, making them an ideal source for finding passwords.

Third party authentication

In a third party authentication mechanism the client authenticates with a third party. This third party can then voucher for the user being the claimed user. Coda and AFS use Kerberos as their trusted third party. Windows can use a Windows domain server as a trusted third party. Trusted third party authentication claims to be the most secure. The enable a user to log on once and then use different services. This minimizes password use and therefore makes it easier to persuade users not to store their password. Trusted third parties require a substantial investment in infrastructure. In most cases a special server has to be set up and managed. This system scales only as well as the third party scales.

Privacy mechanisms

In the discussed systems privacy is provided through the read permission. If a user has the read permission, the files may be read. If a user does not have the read permission bit, then access is denied. This works only as long as the systems are safe and administrators are completely trustworthy. In the case of NFS this has to apply for both the server and the client systems. In all other cases this has to apply to the server system only. In all the systems a user with physical access to the host has the possibility to access all data. In all of the discussed solutions a user with administrator rights has the possibility to read all the data on the system. The Windows security model is the only one of the discussed models where the user will even be able to find out that an administrator has read a file.

CHAPTER 3. REQUIREMENT ANALYSIS

The requirements for the system need to be identified, before the actual system is designed. It needs to be clear which requirements must be met. After all, there is no point in developing a system that solves the wrong problem.

Most of the requirements have already been identified. The features that are desirable in any file storage solution are described in the section called “File system core features”. Architectural qualities for distributed systems in general are described in the section called “Architectural qualities for distributed systems”. To identify the exact requirements system usage patterns have to be investigated. Based on these user roles have to be identified.

File Storage Scenarios

To identify the requirements different scenarios have to be looked at first. Who would benefit from an advanced file system? Who would be using this file system and why? Moreover, what are the things that are important to this particular user group? Of course, in the real world, there will not be a scenario exactly as described here, but rather a mixture. Nevertheless, theses examples will still help to find the actual uses.

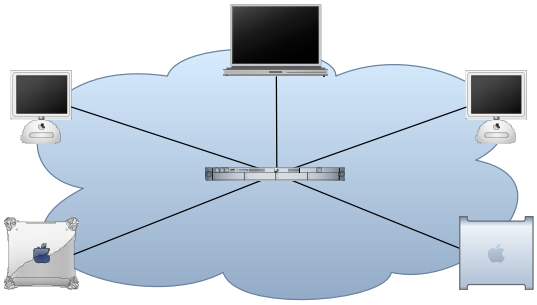
Small work group

I will start with the small work group because this is very easy to describe. Figure 3.1, “Small work group” shows an example. In most cases, there is one file server, one Internet-gateway (sometimes the same host) and a small number of client host (maybe five). Usually all client hosts are either personal computers or shared hosts. All data transfer is done via shared folders on the server.

This system lacks privacy. In most cases, there is a shared folder on the server. All users can read and store files there. There is nothing holding back one user from deleting the file of another user.

There is also usually no or a tedious backup system. All work has to stop should the server fail. In the case of an unrecoverable crash, all previous work could be lost.

Last, but not least, the client hosts are not used to their full potential. Most have extra hard drive space and, depending on the type of work, extra clock cycles. Some hosts are turned off at night, but others are just running idle, using electricity and providing nothing for other users.



A typical small work group example. This work group has one server and five clients (four PCs, one laptop)

Figure 3.1. Small work group

High-Performance Computing Lab

A high-performance computing lab is very similar to the small work group. The clients here are not idle, the distribution of CPU cycles is already taken care of. Nevertheless, many applications require a common data set. This is usually very large, and therefore not on every host, but on one single server. Multiple hosts (25, 50, 100, ...) are trying to get parts of the dataset at the same time. If not carefully planned, this performance leak can seriously reduce performance.

Large network

A large network is similar to the small network. However, suddenly there is more than one server. Some of these might provide backup for other servers. A very important feature here is that users want to be able to log into a different computer, maybe even in a completely different location and still want to be able to access their files. Files should be stored as close to the user as necessary, for performance, but should also be migratable to other hosts. Maybe two people from different location have to share common files. They should be able to share these files quickly.

Home user

The total opposite of the large network is the home user. The home user usually has very few computers: Maybe one desktop and a laptop, maybe two PCs for multiple people. Disk space is always low. Hosts usually have very different performance features, I might be asked to move to the other computer because my brother wants to play a game. In the case of the home user transparent file access to as much disk space as possible is very important.

Concurrent Engineers

Distributed, concurrent-engineering teams would greatly benefit from this system. They work at different physical locations, on different computer systems, with different computer architectures. However, common data such as design documents, schedules, engineering data, notes, etc. have to be shared. The support for versioning will allow the team to go back to older versions, if necessary, but most importantly to ensure that the current version is available to all team members instantly. Data will always be downloaded from one of the hosts available. If a file is already available on a host in the local network, this location will be preferred over a host at any remote location. This enables faster updates and ensures that slower WAN links are less used.

Student Computer Lab

A computer lab is a large array of computers. All computers should behave identically to the user, and offer the same file space. These lab systems usually use a central file storage server, which is a single point of failure. However, each lab host has a big hard drive nowadays, which is hardly used, if at all.

Astronomy

In a sky survey, the amount of data collected is very large. There must be some way to spread data files over multiple computers, or to make whole or partial files available to different users on different hosts. These files are usually associated with metadata. The metadata has to be kept in some kind of database to allow fast retrieval of the important data. [38, 69]

High-energy physics

When the Large Hadron Collider (LHC) study of subatomic particles and forces at CERN will launch in 2007, it will be one of the greatest data management challenges. More than a gigabyte of data will be generated every second. This data will have to be distributed among researchers around the world. With these large amounts of data, it is very important to prefer local replica to remote replica locations to minimize bandwidth usage. [39]

Host types on the network

Based on these usage scenarios, hosts participating in the network can be classified. Each host type has different properties.

Server

Server hosts are usually very reliable. They might have a RAID system, have fail-over power supplies, multiple network interfaces, and other reliability provisions. Server class computers are the easiest to use for administrators of distributed storage systems. Only one system has to work, only one system has to be backed up. Unfortunately, there also have to be client systems to make actual usage of the server.

Always up client

Administrators' favorite client hosts are the ones that are always up. These can easily be maintained remotely. They can be also be used to provide additional server features. However, not too many server features, since there is always a person wanting to work on the host.

Work time up client

Work time up clients are usually on 40 hours a week. A person turns her personal computer on whenever she enters the office, and turns it off whenever she leaves. Most leave the host running during lunchtime, but even that is uncertain. Usually these are personal hosts. The user is concerned about access speed to her personal files, and feels the host slow down if other people access data on the same host.

Laptop

A laptop is the most complicated system to support when it comes to distributed file systems. Usually laptops are moved around from one network to another, connecting and disconnecting it from servers all the time. Fortunately, laptop users are used to this, and therefore can be expected to specify which files they want to work on before they disconnect. Nevertheless, as soon as the laptop is connected to the Internet the laptop user wants to be able to access her files.

Mobile client

The last type of user is a special case of the laptop user, the so-called mobile client. When talking about mobile, I mean small devices like personal digital assistants (PDA) and cell phones. These devices usually connect temporary to the network with a very low bandwidth. Users do not expect to have access to all data, but they do want to have certain files available, usually calendar, address book, and notes.

Use Case Roles

Based on these usage scenarios different usage roles can be defined. These roles are regular file system users, administrators, optimizer services, service provisioners, and intergrid service providers.

File system users

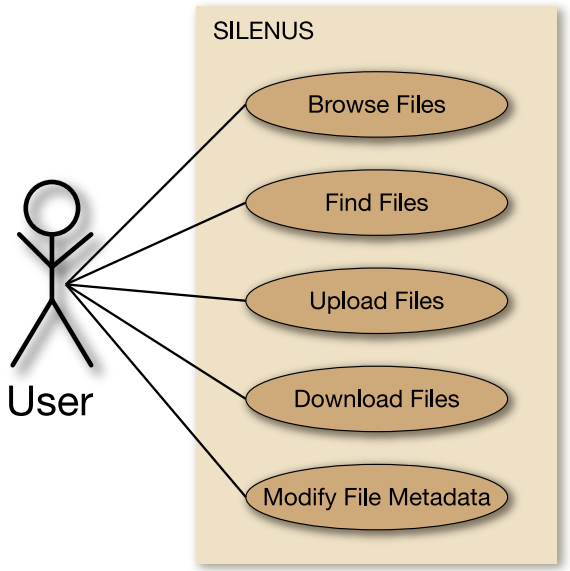


Figure 3.2. Typical user cases for a file storage system

Figure 3.2, “Typical user cases for a file storage system” shows the use cases that are identified for the regular user. These typical tasks can be executed on any existing file system.

Administrators

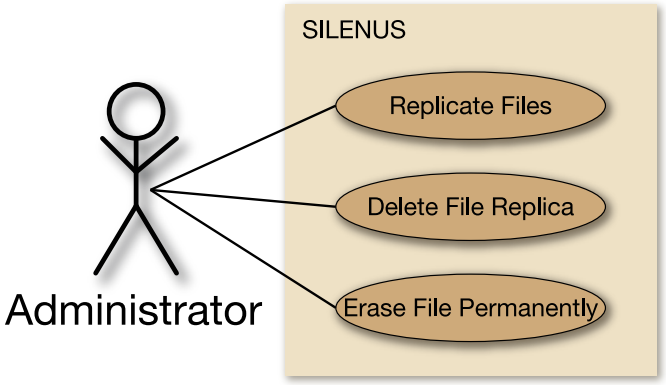


Figure 3.3. Administrator use cases for a replicated file system

Figure 3.3, “Administrator use cases for a replicated file system” shows the use cases for administrators. The administrator has the power to initiate all replication manually. If needed, administrators should be able to delete files completely.

Optimizer services

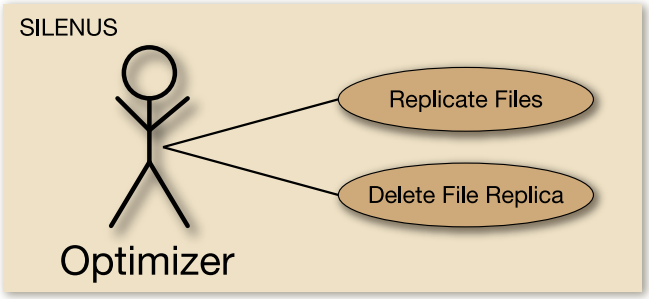


Figure 3.4. Optimizer use cases for a replicated file system

To provide manageability the system should provide internal optimizer services. Figure 3.4, “Optimizer use cases for a replicated file system” shows the use cases for these optimizer services. They have to be able to manage file replication by creating and deleting file replicas.

Service provisioners

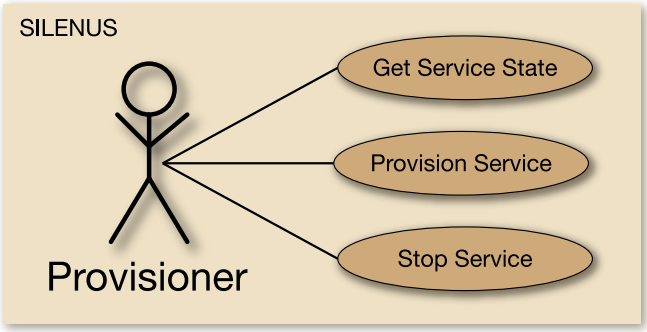


Figure 3.5. Provisioner user cases for a replicated file system

The service provisioner is another type of optimizer service. As Figure 3.5, “Provisioner user cases for a replicated file system” shows a provisioner has to be able to start (provision) and stop services in the network. To make the decision, which services to start or stop it needs to be able to query the current state of each service.

Intergrid service providers

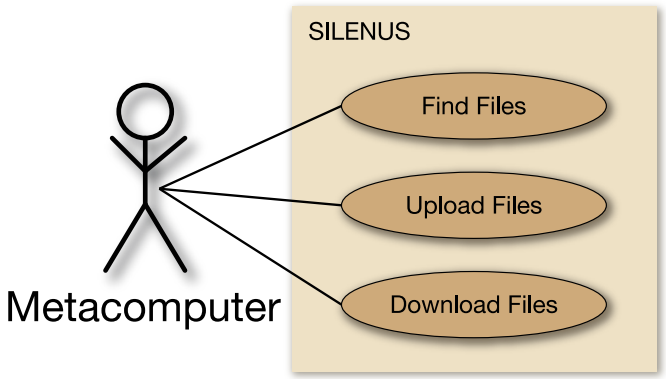


Figure 3.6. Use cases for the intergrid meta computer

Another type of usage role is an intergrid service provider. These provide computing services, providing a meta computer. Figure 3.6, “Use cases for the intergrid meta computer” gives an overview of the use cases required for meta computing. A service has to be able to find data files provided, download them, and upload them after it is done processing.

Use Case Design

Now that it is clear which use case is triggered by which user role each use case has to be described in more detail. A textual use case description has to be developed for every use case.

Use Case 1	Browse Files	
Goal in Context	User wants to view files in the system.	
Scope & Level	User action.	
Preconditions	User has accessed Human Interface; User is logged in.	
Success End Condition	User sees files.	
Failed End Condition	A file listing is not available.	
Primary, Secondary Actors	User, SILENUS system.	
Trigger	Human Interface starts.	
Description	Step	Action
	1	The system fetches a file list.
	2	The file system structure is displayed as a tree.
	3	The user expands directories.
	4	The file is found. User selectes the file.
Extensions	Step	Branching Action
	1a	The file system is unavailable.
	4a	The file is not available. The user has to find another way to access the file.
Sub Variations	Step	Branching Action
	2	Instead of a tree the system may display a list of files and directories in the root directory.

Table 3.1. Browse Files Use Case

Use Case 2	Find Files	
Goal in Context	User wants to find specific files in the system.	
Scope & Level	User action.	
Preconditions	User has accessed Human Interface; User is logged in.	
Success End Condition	User find the file.	
Failed End Condition	The file is not available.	
Primary, Secondary Actors	User, SILENUS system.	
Trigger	User calls Find Files action from menu.	
Description	Step	Action
	1	The user enters search criteria
	2	The system finds files based on matches in the file metadata
	3	The list of files is displayed to the user
Extensions	Step	Branching Action
	3a	The list of files is empty. The user may try again.
Sub Variations	Step	Branching Action
	1	Search criteria may be derived from the currently selected file, e.g.: Find all files with the same name.

Table 3.2. Find Files Use Case

Use Case 3	Upload Files	
Goal in Context	User wants to store files in the system.	
Scope & Level	User action.	
Preconditions	User has accessed Human Interface; User is logged in.	
Success End Condition	The file is stored in the system.	
Failed End Condition	The file if not available in the system.	
Primary, Secondary Actors	User, SILENUS system.	
Trigger	User calls Upload Files action from menu.	
Description	Step	Action
	1	The user selectes a directory as decribed in "Browse Files".
	2	The user chooses upload.
	3	The system displays a tree of the local files and directories.
	4	The user selects a local file or directory for upload.
	5	The selected files and directories are stored in SILENUS.
	6	The system now displays an updated list of files and directories.
Extensions	Step	Branching Action
	5a	No Byte Store service is available. The upload fails.
	5b	There is not enough space available. The upload fails.
Sub Variations	Step	Branching Action
	5	If the user agent is can start an active server process, it may passively upload the files (pull file upload). If the user agent is restricted, it has to upload the files through a push upload.

Table 3.3. Upload Files Use Case

Use Case 4	Download Files	
Goal in Context	User wants to retrieve files from the system.	
Scope & Level	User action.	
Preconditions	User has accessed Human Interface; User is logged in.	
Success End Condition	The file is retrieved to the local file system.	
Failed End Condition	The file is not available on the local system.	
Primary, Secondary Actors	User, SILENUS system.	
Trigger	User calls Download Files action from menu.	
Description	Step	Action
	1	The user selectes a file as described in "Browse Files".
	2	The user selects download.
	3	The system displays the local file system tree. The user selects a directory.
	4	The selected files are downloaded to the local file system.
Extensions	Step	Branching Action
	4a	The files may not be available. The user has to try again.
	4b	There may be insufficent space on the local file storage to download the file.

Table 3.4. Download Files Use Case

Use Case 5	Modify File Metadata	
Goal in Context	User wants to change files in the system.	
Scope & Level	User action.	
Preconditions	User has accessed Human Interface; User is logged in.	
Success End Condition	The file metadata is updated.	
Failed End Condition	The file metadata is unchanged.	
Primary, Secondary Actors	User, SILENUS system.	
Trigger	User selected action.	
Description	Step	Action
	1	The user selectes a file as described in "Browse Files".
	2	The file metadata is displayed in a table.
	3	The user may change the metadata.
	4	The metadata is updated on the metadata stores.
Extensions	Step	Branching Action
	3a	Some metadata may be read-only.
Sub Variations	Step	Branching Action
	3	Deleting a file is also changing its metadata.
	3	Renaming a file is a change in metadata.
	3	Moving a file is a change in metadata.

Table 3.5. Modify File Metadata

Use Case 6	Replicate Files	
Goal in Context	Make a file available on multiple byte stores.	
Scope & Level	Administrator action. Optimizer action.	
Preconditions	A file is uploaded and available at at least one byte store.	
Success End Condition	The file is available at another byte store.	
Failed End Condition	The file could not be replicated.	
Primary, Secondary Actors	Administrator, Optimizer, SILENUS system, byte store service.	
Trigger	A file is uploaded; or a files availability has dropped below a given level.	
Description	Step	Action
	1	The system identifies the byte stores that the file is stored on.
	2	The system identifies a target bytestore.
	3	It triggered copying between both byte stores.
	4	The metadata is updated to reflect the new situation.
Extensions	Step	Branching Action
	1a	No byte store may available. The file is unavailable and cannot be replicated.
	2a	No other byte stores may be available. The file cannot be copied.
	3a	The copying may fail. In this case another target byte store has to be selected.
	4a	The metadata may have changed in between. It must be ensured that there is no conflict.

Table 3.6. Replicate Files Use Case

Use Case 7	Delete File Replica	
Goal in Context	A file replica is removed from a byte store.	
Scope & Level	Administrator action. Optimizer action.	
Preconditions	A file is uploaded and available at multiple byte stores.	
Success End Condition	The file is deleted from a byte store.	
Failed End Condition	A file could not be deleted.	
Primary, Secondary Actors	Administrator, Optimizer, SILENUS system, byte store service.	
Trigger	A file exceeds its useful availability level.	
Description	Step	Action
	1	The system identifies the byte stores that the file is stored on.
	2	The system identifies a byte store to delete from.
	3	The metadata is updated to remove the byte store.
	4	The byte store is asked to delete the file.
Extensions	Step	Branching Action
	1a	No byte store may available. The file is unavailable and cannot be deleted.
	3a	The metadata may have changed in between. It must be ensured that there is no conflict.
	4a	The byte store may have become unavailable. In this case the delete request must be retried at a later time.

Table 3.7. Delete File Replica Use Case

Use Case 8	Erase File Permantly	
Goal in Context	A file is completely deleted from the file system.	
Scope & Level	Administrator action.	
Preconditions	Administrator has accessed Human Interface; Administrator is logged in.	
Success End Condition	The file is unavailable.	
Failed End Condition	The file is still available.	
Primary, Secondary Actors	Administrator, SILENUS system, byte store services.	
Trigger	Administrator triggered action.	
Description	Step	Action
	1	The administrator selects a file as shown in "Browse Files".
	2	The adminstrator chooses to delete a file permanently.
	3	The system retrieves a list of all byte stores containing the file.
	4	It updated the metadata to reflect the deletion.
	5	All byte stores are asked to delete the file.
Extensions	Step	Branching Action
	5a	Not all byte stores are available. In this case the delete request must be retried at a later time.

Table 3.8. Erase File Permanently Use Case

Use Case 9	Get Service State	
Goal in Context	A service provides information about its current state.	
Scope & Level	System information.	
Preconditions	A service is identified.	
Success End Condition	Service quality information is available.	
Failed End Condition	Service quality information is not available.	
Primary, Secondary Actors	SILENUS system; Optimizer service.	
Trigger	Automatically	
Description	Step	Action
	1	The system fetches a list of all services.
	2	The services provide information on their current state.

Table 3.9. Get Service State Use Case

Use Case 10	Provision Service	
Goal in Context	Make another service availale.	
Scope & Level	Provisioner action.	
Preconditions	Service states are collected.	
Success End Condition	A service is started on another host.	
Failed End Condition	The service could not be started.	
Primary, Secondary Actors	SILENUS system; Optimizer service.	
Trigger	Service state shows that a service is overloaded.	
Description	Step	Action
	1	The provisioner fetches the service state from all services.
	2	It checks if any service is overloaded.
	3	A new host to run a service is identified.
	4	The service is deployed at the target host.
	5	The service is started on the new host.
Extensions	Step	Branching Action
	2a	No services are overloaded. No new services need to be provisioned.
	3a	No hosts may be available. Try back at a later time.
	4a	Deployment may fail. Go back to step 3.
	5a	Starting the service may fail. Undeploy the service and go back to step 3.

Table 3.10. Provision Service Use Case

Use Case 11	Stop Service.	
Goal in Context	A running service is terminated.	
Scope & Level	Provisioner action.	
Preconditions	Service states are collected.	
Success End Condition	The service is stopped.	
Failed End Condition	The service is still running.	
Primary, Secondary Actors	SILENUS system; Optimizer service.	
Trigger	A service is underused.	
Description	Step	Action
	1	The provisioner fetches the service state from all services.
	2	The checks if a service is underused.
	3	A service that can be terminated is identified.
	4	It is ensured that all the data available on this service is still available on other services.
	5	The service is terminated.
Extensions	Step	Branching Action
	2a	The services may all be in use.
	4a	Some data may not be replicated on other services. In this case the service cannot be terminated.
Sub Variations	Step	Branching Action
	4	For byte stores it must be ensured that all files stored on this byte store are also available on other byte stores. For metadata stores it must be ensured that this service is synchronized with the other metadata stores.

Table 3.11. Stop Service Use Case

CHAPTER 4. ARCHITECTURE AND DESIGN

In this chapter a new model for a distributed file storage solution is introduced. This model is specified in terms of its system architecture, interfaces, and interaction among its components.

To specify a system, its architecture has to be defined first. The architecture is necessary to understanding and manage system complexity. Once the architecture is specified individual components can be designed.

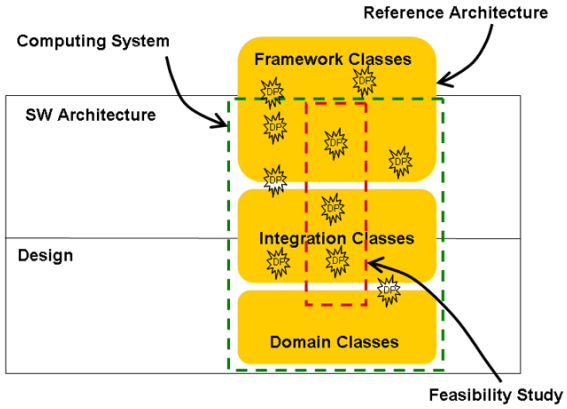


Figure 4.1. Class Model vs. Architecture and Design

A service-oriented approach is chosen to satisfy the given requirements. The system will be broken up into smaller components, which will be implemented as services. Each service has a specific responsibility. Since all services are dynamic in nature, there is no specific deployment to any particular host. Each host can host none, one, some or all of the services. These services will use the SORCER network to communicate with each other. [40]

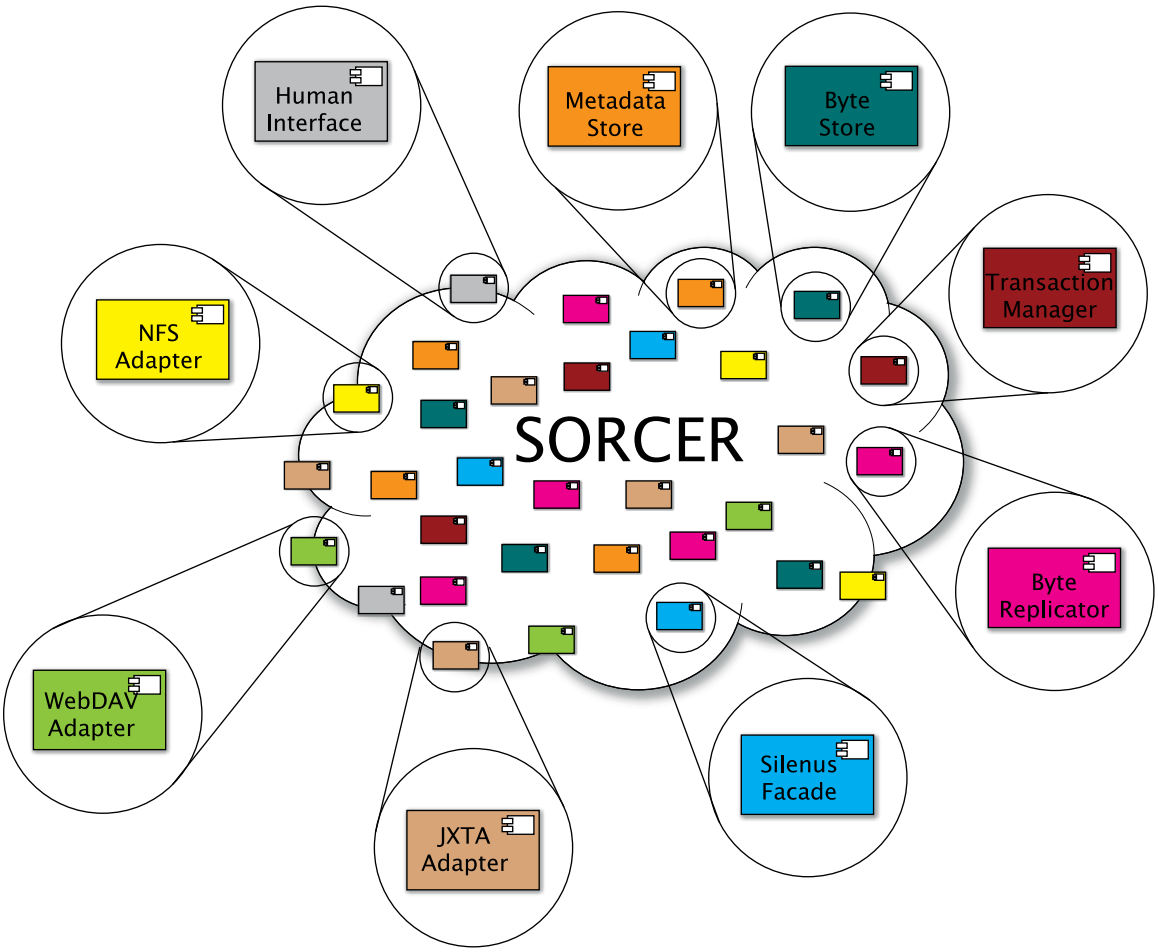


Figure 4.2. Silenus components communicating over the SORCER network

A model for a grid based environment

In a grid-based environment there is no clear notion of a client and a server computer. Every node in the grid is a client and a server at the same time. It is a server, since it offers services to other computers. It can offer computing services and data services. It is a client, since it requires services from other nodes. It requires computational results from other nodes through shared data.

A different architectural model, such as a peer-to-peer or a service-to-service is required. In a classical client-server application a large load is put on one server host. If multiple grid clients try to access the same server host at the same time, the server will be overloaded. In a peer-to-peer architecture, every host is a client and a server at the same

time. The load is now balanced between all hosts. This is a preferable model if all hosts are equal. The service-to-service architecture splits up the functionality of the system into smaller services. Each host may now run zero, one, or multiple service providers. A service requestor can use any service provided by any host in the network. This supports different host configurations: A set of hosts may run data service providers, another set of host may run computational service providers, and some hosts may run both.

I therefore propose the following changes to Coulouris model to be used as a grid model: Flat file service and directory services will become independent service providers. The client module will be stripped of all duplicated functionality.

Having the flat file service and the directory service as independent service on the network provides scalability. A service requester may choose any directory and flat file service that is available. Traversing directories does not use the flat file service, thus saving resources.

The client module will be stripped from its duplicated functionality. If local caching is desired, a flat file and directory service can be run on the local host.

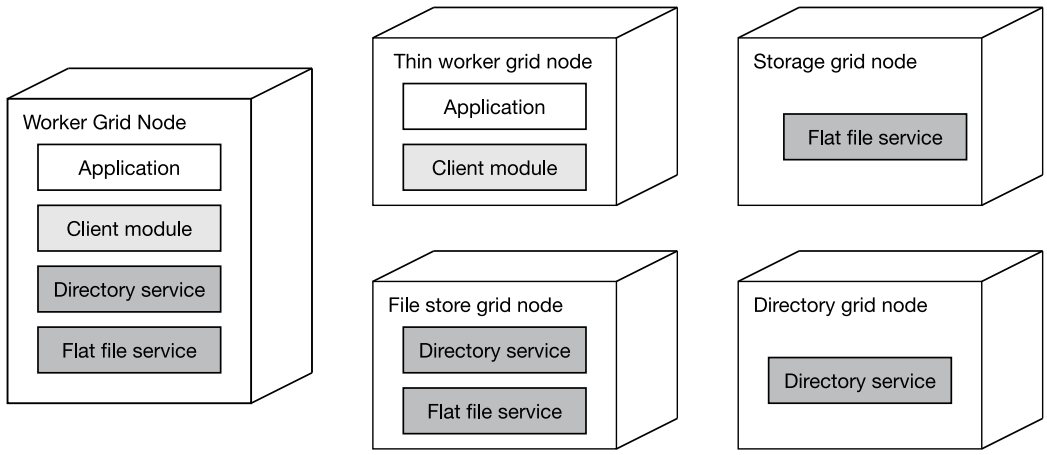


Figure 4.3. Grid model for data storage

SILENUS architectural model

The new SILENUS architectural model extends the traditional file storage model. SILENUS distinguishes between client modules, a directory service, and a flat file service. It introduces extra management services for coordination and for optimization.

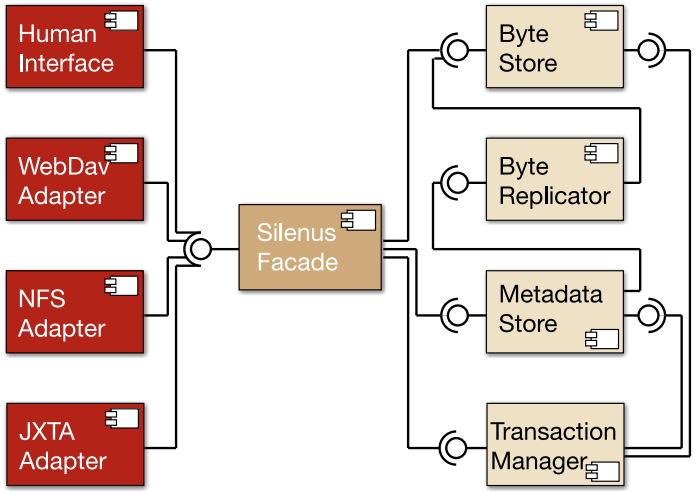


Figure 4.4. The SILENUS Components

The Human Interface, WebDAV Adapter, NFS Adapter and JXTA Adapter are client modules. Each one of them serves a particular type of client. The ones given here are just examples, adapters could be written for any other existing file storage solution. The human interface (ServiceUI) provides support for file storage and management through a proprietary user interface. It provides access to the extra features, which are not available through the other interfaces: Advanced features such as manual migration, number of replicas, log-file viewing, and others. The service interface should only be needed for these extra features and can be ignored by most users. The WebDAV Adapter provides support for operating systems that have support for WebDAV, such as Windows, Mac OS X, and newer UNIX systems. It provides support for existing applications. This gives current operating systems the possibility to use the file storage without having to install a client. The NFS adapter provides support for older UNIX systems. A JXTA adapter provides support for the JXTA content management interface. These adapters are just examples of mapping from SILENUS to existing systems, other adapters may exist as well. Unlike the Coulouris Model, these adapters do not have to provide support for advances features, such as caching, which makes them smaller and easier to adapt to other interfaces.

The SILENUS Facade and Transaction Manager introduce a new coordination service. To make the client modules even smaller, the coordination between the client modules and the providing services is sourced out to the SILENUS Facade. It provides a gateway to the SILENUS file storage. This component provides a facade to the underlying services. It takes care of transactional semantics between file and meta information storage. It provides one easy interface for the user. The facade provides support for forwarding requests to the appropriate services. It uses the Transaction Manager for ensuring consistency. The Transaction Manager is a JINI standard component for handling transactions in a distributed environment.

The Byte Store maps to the flat file store of the Coulouris model. It provides functionality for creating and retrieving file data. The Byte Store does not provide file attribute storage, which is different from the Coulouris model. It does, however, provide support for retrieving attributes that are derived from the file data. Such attributes include file size and checksums. These can be used to verify the file contents. The Byte Store provides fast access to the files stored on the provider's host. Files are usually stored encrypted, but can be unencrypted for performance reasons

The Metadata Store maps to the directory service in the Coulouris model. It provides functionality to create, list, and traverse directories. It also provides functionality to retrieve the file data location. Unlike the directory service, the Metadata Store is also responsible for file metadata. File metadata is all the information that is either included in the actual file data or that can be derived from the file data, such as file name, creation date, file type, type of encryption, and others. As a matter of fact, the file storage location, the file name, and even the directory a file is in are nothing different than just three file attributes. This allows all these attributes to be handled in a standard way. Multiple versions of one file may exist in the database for recovery purpose.

The Byte Replicator and other optimizer services provide a component that is not yet present in Coulouris model. They provide support for autonomic computing. In a classical data storage solution, an administrator has to manually move and distribute files among different servers. In SILENUS, this is done by optimizer services. These services will analyze the current network condition and make decisions on where to store files, where to keep replicas, and even when to startup and shutdown services. Each optimizer service is a separate component, allowing an administrator to chose exactly which kinds

and how many optimizer services to run on the network. One example of these services is the ByteReplicator service. It will make sure that uploaded files are replicated among different byte store nodes to provide redundancy. Optimizer services can request log information from the storage providers, and can automatically initiate replication and migration. It can detect usage patterns and make sure that the files are available to the user. It can alsodetect non-responding systems and automatically replicates all files that were stored on it. The Replicator also ensures that all storage servers have the latest version of the files.

After this overview over the services and their interactions, the individual services can now be looked at in more detail.

Components

Service user interface

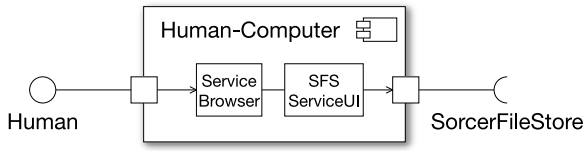


Figure 4.5. Component diagram for the user interface

To work with the file system, users need an interface. None of the compatibility interfaces can provide access to all of SILENUS capabilities. Therefore and additional user interface is provided.

The user interface is dynamically downloaded when needed. Unlike traditional systems that require installation on a client computer, SILENUS user interface is dynamic. The user needs to have a service browser installed. This service browser can detect services running in the network. It can then download and display these provided user interfaces. There is no actual configuration needed on the client computer.

WebDAV adapter

Figure 4.6. Component diagram for the WebDAV adapter

The WebDAV adapter provides the connection from existing applications and file systems to the SILENUS file storage system, as shown in Figure 4.7, “The WebDAV adapter”.

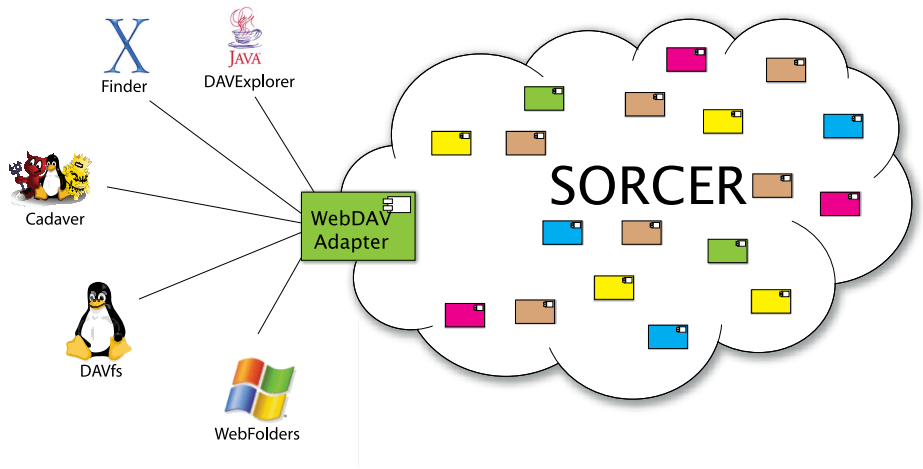


Figure 4.7. The WebDAV adapter

The WebDAV adapter uses Java Servlet technology to handle requests instead of rewriting complete new server software. WebDAV is based on HTTP, as explained in the section called “WebDAV”. Therefore, existing application servers that handle HTTP can be reused to provide a WebDAV server. One of these technologies is Java Servlets, as explained in the section called “Web-based access to file storage”. The Servlet standard provides functionality for handling HTTP requests with the HttpServlet interface. It is very easy to add the additional functionality required for WebDAV.

Incoming WebDAV requests will have to be mapped to the appropriate file store requests. Most requests are straightforward: GET and PUT will be implemented using the upload and download functions. PROPFIND uses the request node info, and PROPPATCH will set node info. LOCK requests can be ignored, but need to be handled internally to provide consistency.

The implementation and details of the WebDAV adapter is a pending master thesis topic for Fajin Wang.

NFS adapter

The NFS adapter provides a mapping from the NFS file system protocol to the SILENUS file storage. The NFS file system protocol is most widely available on UNIX client hosts. Although many newer UNIX systems provide support for WebDAV, not every older system has support for it. These systems, however, can access the NFS protocol. Providing support for NFS and WebDAV allows a broad number of clients to connect.

The NFS protocol is based on remote procedure calls (RPC). In an RPC the client sends a UDP packet to the server. This packet contains a program number, a procedure number, and data. The server will dispatch the appropriate program and procedure, and return a reply packet. The NFS protocol is specified in [1] and [2].

The functionality of the NFS protocol includes file system statistics, directory handling, attribute handling, and file read and write operations. The file system statistic functions provide a count of used and available block. The directory handling allows listing, creating, and deleting directories. File attributes can be read and set in with the common UNIX permissions. File read and write allows reading and writing of data blocks.

The mapping of most functions to the SILENUS system is straightforward. Directory handling and attribute handling can be directly mapped to reading and setting information in the metadata store.

NFS handles use 256 bits while SILENUS UUIDs have a variable bit length. They must therefore be mapped into the handle range of NFS. A special mapping table keeps a relation between NFS handle ids and SILENUS UUIDs. The mapping itself

is volatile. Should the mapping information be lost a "stale NFS handle" error will be created. NFS clients can recover from this error by rebrowsing the list of files and directories.

The file system statistics as such do not exist in the SILENUS system. NFS has support for used, available, and free blocks. At the moment these are faked using dummy values. A new service could be added that collects these statistics and provides a better estimate.

Mapping the read and write functions is more complex. NFS supports functionality for randomly reading and writing blocks. This is due to the fact that NFS calls are supposed to be stateless and idempotent: If the result or request for a read or write operation is lost this operation may just be sent again. SILENUS, however, supports file reading and writing through byte channels. The NFS adapter therefore has to keep the state of the clients current read or write function. Fortunately, most file reads and writes are not in random order but sequential. The NFS adapter keeps lists of open channels for the last read and write operations. If the same file is read or written again, the old byte sequence accessor is used to continue reading or writing the data. If no operation occurs in a given time, the channel is closed.

File store

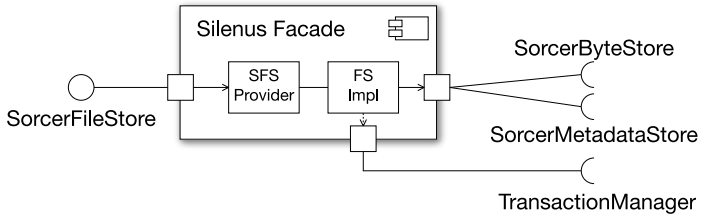


Figure 4.8. Component diagram for the SILENUS facade

The SORCER File Store interface provides a facade to the SILENUS network for clients that want to use the system. Since the metadata and actual file contents are stored in different services, there is need to coordinate between these two services. To make use of the file system easier this functionality is combined in the SILENUS facade with the File Store interface.

Most of the file store functionality is very straightforward and just consists of forwarding a request to the appropriate service. Actions like retrieving file metadata or setting file metadata can be directly forwarded to a metadata store. In this case, the extra step of going through the facade can be skipped: These functions are implemented as a smart proxy that will be downloaded to the client. The smart proxy can talk directly to the metadata store, thus reducing overhead.

File download has to be coordinated between two services: The file metadata has to be retrieved from the metadata store. This metadata contains information about the byte store that carries the file contents. A connection has to be made with that particular byte store.

File upload requires the use of transactional semantics. When a file is to be uploaded, two things have to be created: A new node in a metadata store, and the file data has to be uploaded to a byte store. To save time both requests can be started in parallel. However, it is very important that, should one of them fail, the other one be cancelled. Figure 4.9, “File upload transactional semantics” shows this transactional semantics.

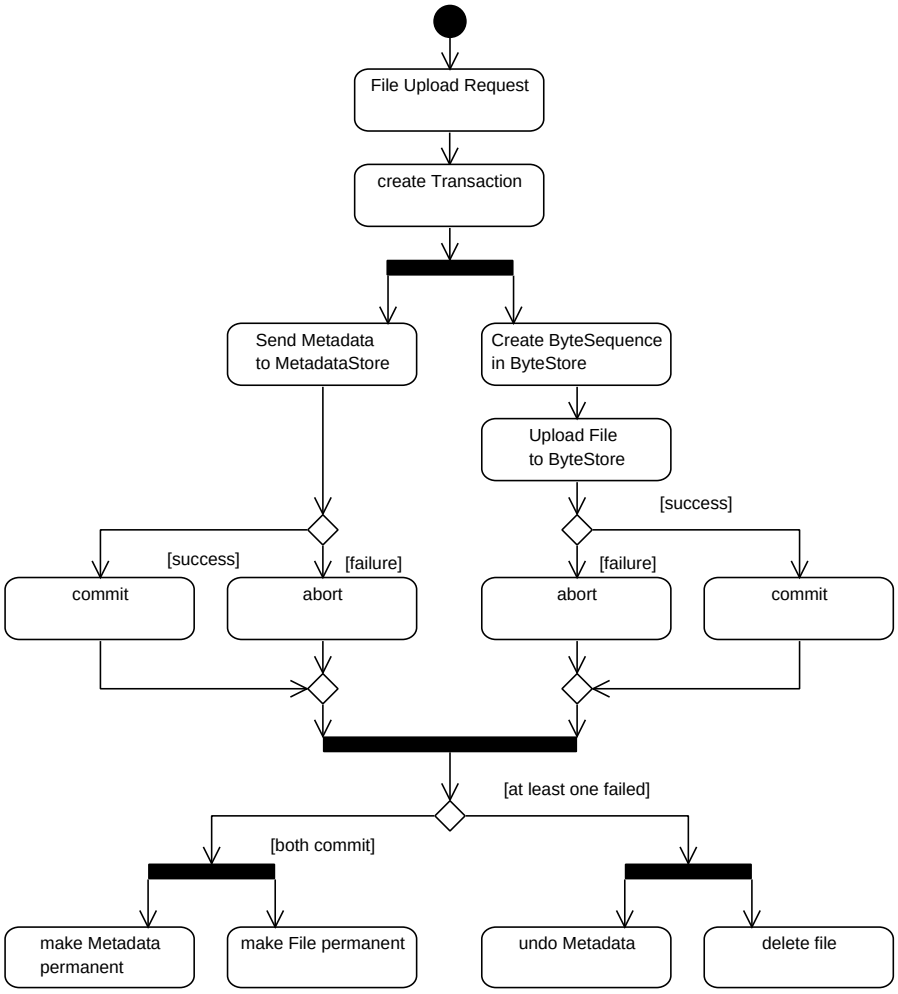


Figure 4.9. File upload transactional semantics

To support the transactions a separate transaction service is needed. Fortunately, Jini already provides a standard for the Transaction Manager interface. It also provides a reference implementation, called Mahalo, which implements this interface. The SILENUS facade can use either this or any other service that provides transactions to ensure that both operations succeed.

Metadata store

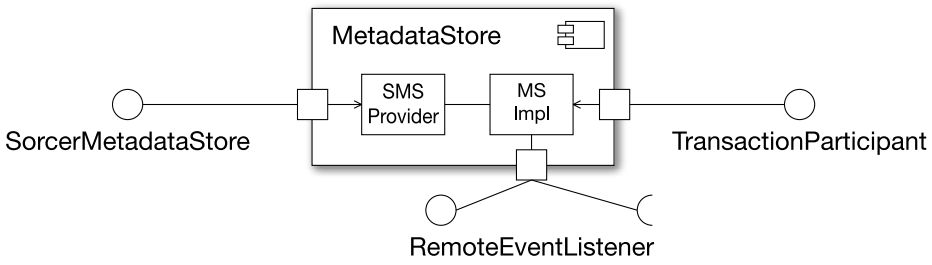


Figure 4.10. Component diagram for the metadata store

The metadata store provides attributes for the files stored in the file system. The analogy in a traditional storage system is the file system. The metadata information creates the well-known hierarchical structure. Files in the Metadata store are identified by UUIDs. The metadata provides mapping from and to file names.

The file metadata is stored in key-value pairs for each file. The key describes the kind of attribute (e.g. file name, creation date), where as the value describes the value of the attribute.

There are two types of file attributes. Basic attributes are of type string or are easily represented in string form. Extended attributes can be any Java object. This distinction is necessary when retrieving file attributes. Instead of having to choose a list of attributes, a client can choose to get either just the basic attributes or all attributes. This makes look-ups for basic attributes fast, but does not limit the attribute types.

The two attributes parent and mime type are used to create the well-known hierarchical file system structure. Every node except for the root directory has exactly one parent node. The mime type describes the type of the file. A special mime type is used for directories and links.

Metadata stores are synchronized while connected. All metadata stores contain the same information. Should a metadata store be disconnected while its information changes, it will be resynchronized when it is connected back to the other metadata stores.

The metadata store meta information is needed for metadata store synchronization. The metadata store needs to keep track of which file versions it has and when the last synchronization has occurred.

As in internal database, an embedded database is chosen. Using an embedded database makes installation much easier; it does not require the installation of external database software. The database access itself is done using the data access object pattern to extensibility and support for other databases if needed. A high-performance computing lab, for example, could set up commercial database software to increase performance.

Byte store

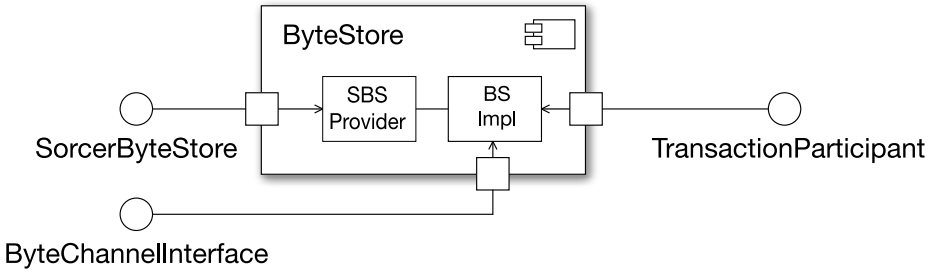


Figure 4.11. Component diagram for the byte store

The byte store service stores the actual file data. In the analogy of hardware, this would be the actual hard drive.

The ID of the byte store and an entry ID in the byte store identify files in a byte store uniquely. These ID numbers never change. This makes the file storage independent from file metadata such as the file name. The byte store services provide nothing but support for file storage. The advantage is that this service can be then optimized for performance. Adam Turner is currently working on his master thesis investigating potential performance optimization using a BitTorrent like file distribution.

Unlike the metadata stores, the byte stores are not synchronized. File data is much larger than file metadata. Would the file data be replicated on every node the storage capacity would be filled very quickly. It is the job of the optimizer services to provide file data replication.

Optimizer

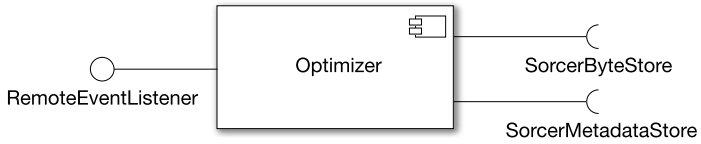


Figure 4.12. Component diagram for the optimizer

The optimizer services keep the network in good shape. There can be many different optimizer services. Each service could provide different optimizations.

One example service is the ByteReplicator optimizer service. This service is triggered when a new file or a new version of a file is uploaded to the file system. It will then look for another byte store that has enough storage space. It tells the other byte store to replicate the file. After the file is replicated, it will update the metadata stores to have the new location information. This ensures reliability by providing multiple copies. Not only new files can trigger replication. If a byte store service becomes unavailable, all files that where stored on that services are potential candidates for replication: They may now exist in the network only once, not providing reliability. In this case, the ByteReplicator has to trigger another replication.

Another type of optimizer services is an autonomic provisioner. When the file system becomes full, the provisioner may start more byte store services. When the file system is sparsely used, these byte store services may be shut down. When the metadata stores and the SILENUS facade get to many requests, the provisioner may start provision new services in the network. When the number of requests goes down, the provisioner may stop these services.

Component Use Cases

Based on the SILENUS architectural model we will examine three typical use cases in file storage systems. These cases are browsing for files, uploading a file, and downloading a file. For file upload there are two different use cases, one for push and one for pull operation. For downloading we will look at the caching and non-caching use cases.

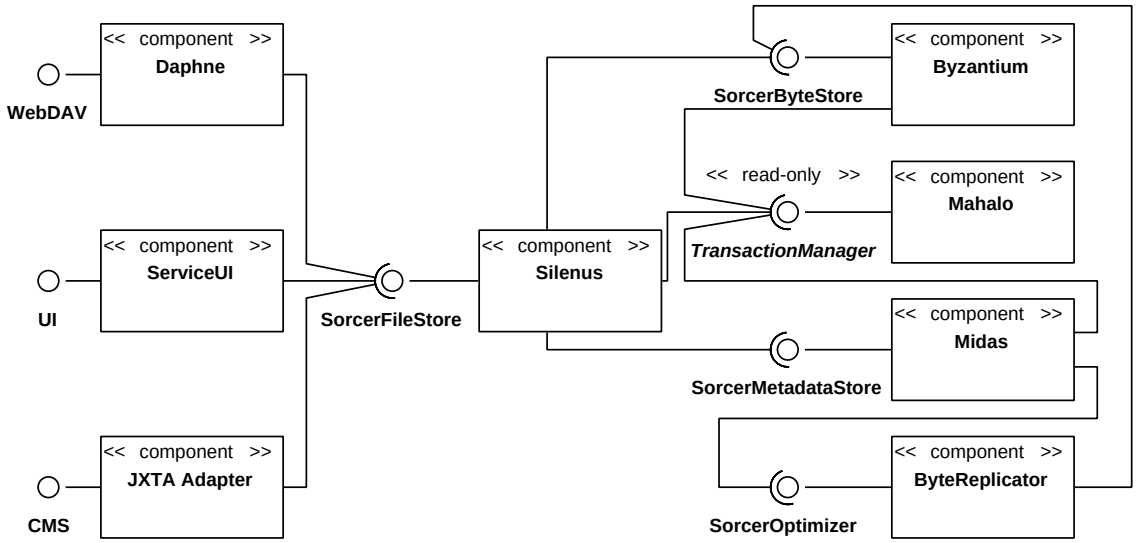


Figure 4.13. SILENUS architectural model overview

There is also a direct connection between the client adapters and the byte store that is not shown in the overview. This connection is used for the actual file data upload and download. There are two different cases: One for passive clients, and one for an active client that has its own service process.

In the case of a passive client adapter, the byte store offers it services to the client. All connections have to be initiated from the client. Files have to be uploaded with the push method.

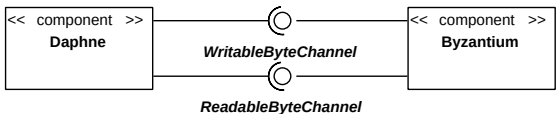


Figure 4.14. Direct connection with a passive client

In the case of an active client adapter a connection can be initiated from the byte store service. This method requires the client to run its own service process. If this is not possible, it can fail back to the passive method. An active client could be the user interface or another byte store.

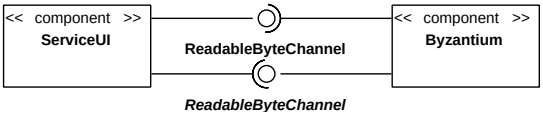


Figure 4.15. Direct connection with an active client

Given this model the use cases can now be described.

Browse files use case

The browse files use case is very straightforward. The request for browsing is forwarded to the facade, which will then forward it to a metadata store. The metadata store will return the attributes for a given nodes. The list of children is one of the attributes.

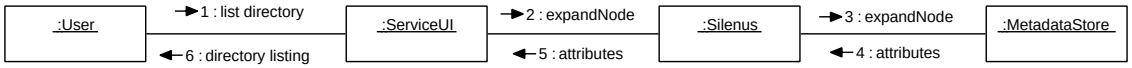


Figure 4.16. Browse Files

Push upload file use case

To facilitate the file upload, the file data has to be split up into file content and file metadata. It is the SILENUS facade's responsibility to coordinate this split up and to ensure that both actions succeed. The byte store returns a writable byte sequence. This byte sequence is passed back to the client, which can then use this sequence to upload the file.

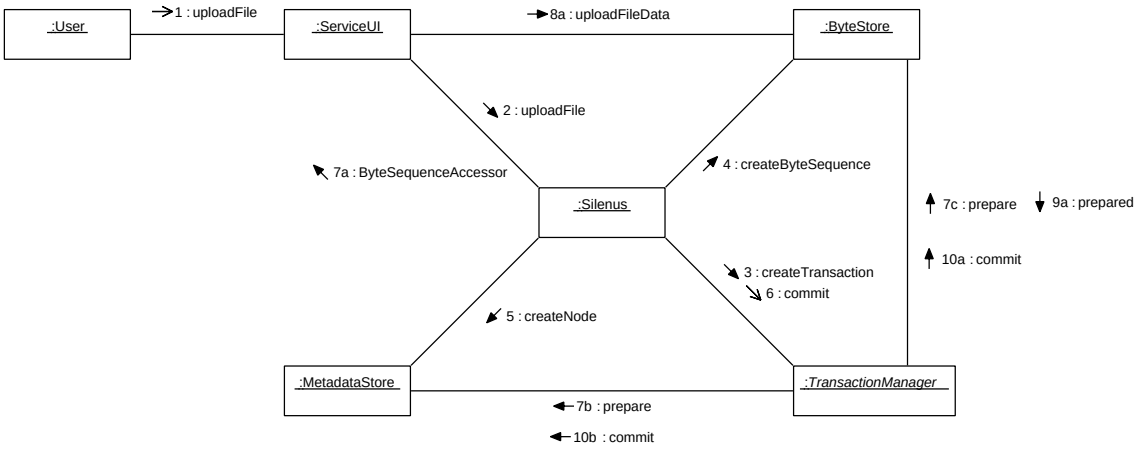


Figure 4.17. File upload with push

Pull upload file use case

The pull file upload is identical to the push file, but here the byte store is the active component pulling its data from the client. To allow a pull file upload the client module has to run its own service component. The byte store can then pull the byte data from the client. This moves the management of the actual file transfer to the byte store. As with the push file upload, the transaction manager is used to ensure that this operation completes successfully.

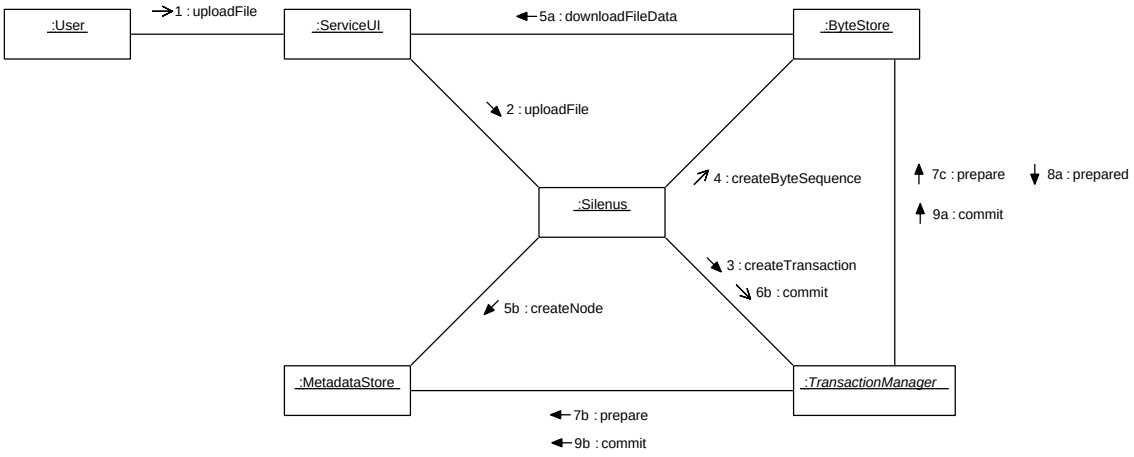


Figure 4.18. File upload with pull

Non-caching download file use case

To download a file, the SILENUS facade first asks a metadata store for the files metadata, which includes its location. The facade can then decide on a byte store. It will ask the byte store to return a byte sequence. A byte sequence is a smart proxy that contains the information on how to talk back to the byte store. It will return this byte sequence to the client module. The client module can then use this byte sequence to download the actual file contents.

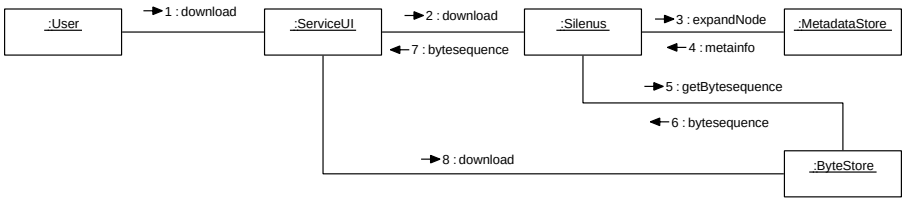


Figure 4.19. Downloading a file

Caching download file use case

To facilitate caching the SILENUS facade needs to know which byte store is considered "local" to the client host. This is usually a byte store on the same host, but may also just be a byte store in the local network. There may even be multiple local byte stores. If a file is available at a local location, this location is used and the interaction is the same as in the non-caching case. If the file is not available locally, the facade initiates a transfer between the remote byte store and the local byte store. It will return the handle to the local byte store to the client module. The client module can then download the content from the local byte store at the same time the local byte store downloads the content from the remote byte store.

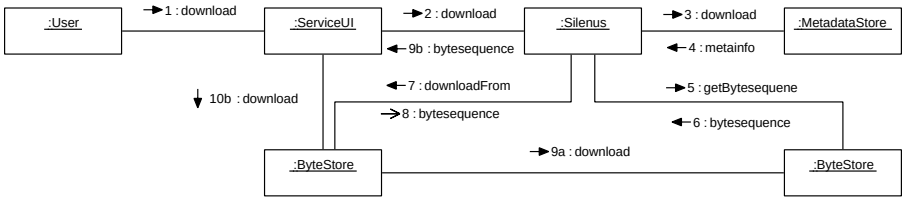


Figure 4.20. Downloading a file with caching

Use cases for Service-oriented programs

Service oriented programs can use the SILENSU file storage in a similar way the GUI and WebDAV client modules use the SILENUS file storage. When uploading or downloading a file, they will connect to the SILENUS facade to retrieve and store their data.

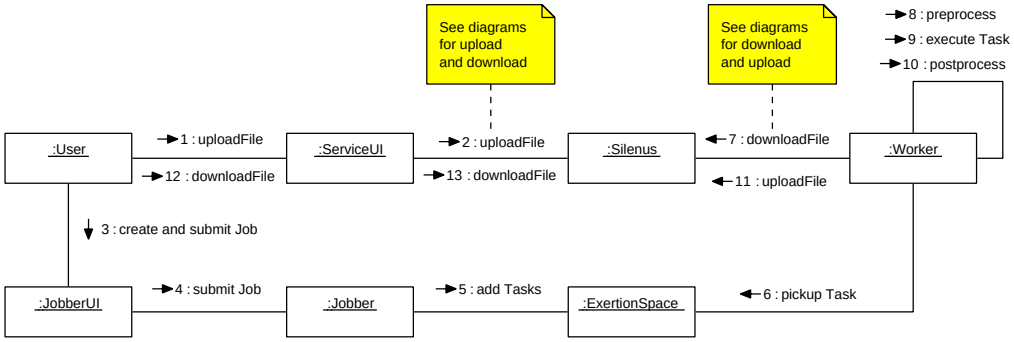


Figure 4.21. Use case for SO Task using file store

The retrieval and storage of file data is the exact same mechanism as from the other client modules. The interaction is the same, just with the ServiceUI replaced by the worker Task.

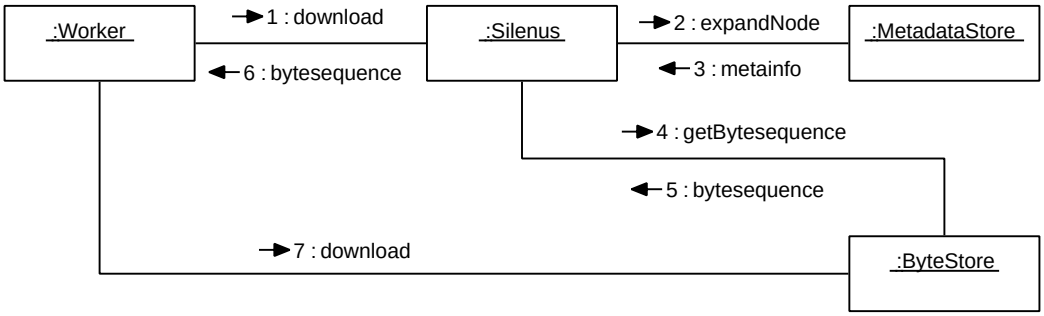


Figure 4.22. Worker service download case

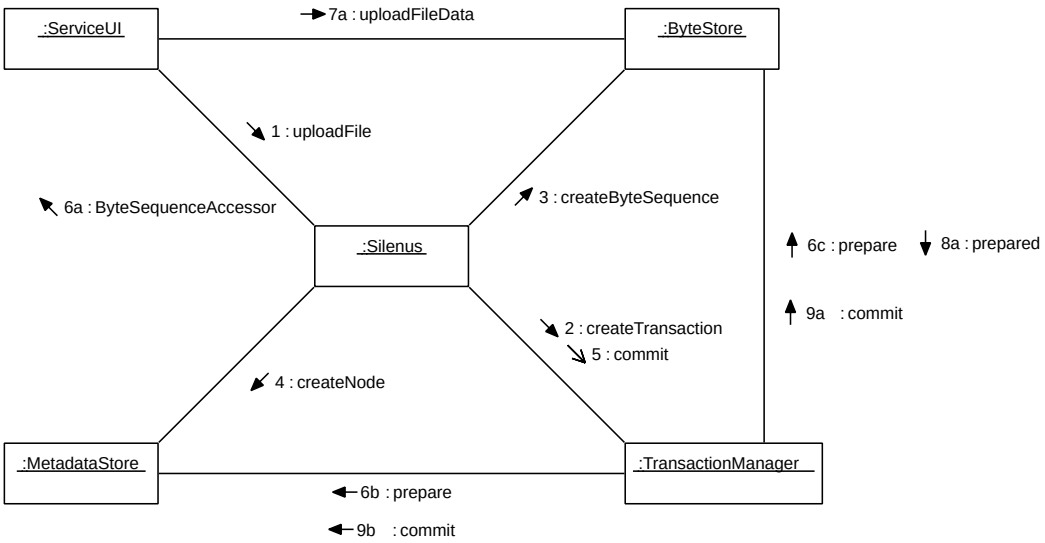


Figure 4.23. Worker service file upload case

If the processing for the job is done in multiple tasks, then each task has to upload and retrieve the file from the file storage system

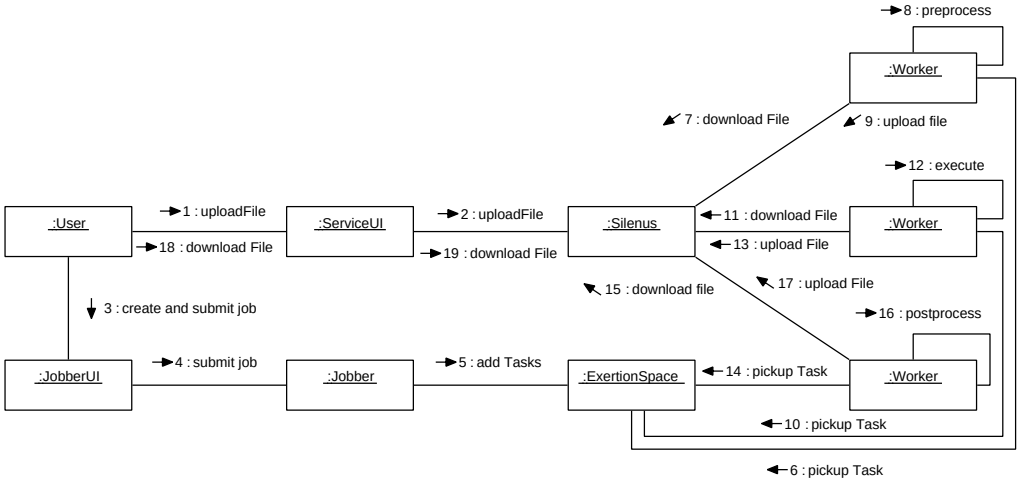


Figure 4.24. Use case for several tasks using SO file store

The Service context for the tasks will contain a link to the files used for the task. It will contain a link for the input file and a link for the output file for each task. These links are stored as SILENUS URIs.

1. The user uploads a file into SILENUS at /somePath/someData
2. The job is created. In its context the key "InputData" is set to "sorcer://FileStore/somePath/someData"
3. The preprocessor reads the field, downloads the data, and creates a new file "someData.preprocessed". In the context it sets the key "PreprocessedData" to "sorcer://FileStore/somePath/someData.preprocessed"
4. The worker service reads the field, downloads the preprocessed data, and creates a new file "someData.processed". It sets the key "ProcessedData" to "sorcer://FileStore/somePath/someData.processed".
5. The post process task reads the field, downloads the processed data. It then creates a new file "someData.postprocesssed". It sets the key "OutputData" to "sorcer://FileStore/somePath/someData.postprocessed".

Example 4.1. Sample usage of SILENUS URIs

Instead of passing SORCER URIs with filenames between the internal services, the services may pass the file store UUIDs between the services, thus saving the next service the lookup process. They would then use URIs like "sorcer://FileStore?uuid=1234-5678-90AB-CDEF". This can be done between the preprocessor task and the process task, and between the process task and the postprocessor task. It cannot be done for the input file name and the output file name where the result is returned to a human user, who will most likely prefer a readable URI.

File system attributes

We will look at several file system attributes and describe how the SILENUS model handles them. For each of these attributes advantages and disadvantages are identified.

Transparency

The ISO defines seven levels of transparencies in distributed applications. Each transparency is analyzed and looked at in the context of SILENUS.

The first transparency is location transparency: It should not matter where a file is actually stored, it should always be accessible. In SILENUS, a file is accessible as long as at least one byte store that has the file data and one metadata store are available. The

client module will provide access to the file just like local files, thus providing location transparency. The drawback of providing location transparency is slower file access. Finding the location of a file requires extra overhead. Retrieving the file content from a remote host is always slower than retrieving it from the local host. If the file content is stored at a remote location, there may a low bandwidth between the local host and the host storing the file data. Local caching can lessen this disadvantage.

The second transparency is access transparency. Files should be accessible through existing software. This transparency is provided through the client modules. Each client module adapts the file system to an existing environment. These adapters require extra overhead.

Replication transparency requires that it should not matter on which file replica a user works. This is provided by SILENUS update mechanism, which is explained in the section called “File Replication”.

Failure transparency states that the system should still work in the expected way in case of a failure. Failure transparency in SILENUS is acquired by file replication and by expecting disconnection. Both solutions lead to more overheads in the system. File replication requires more storage space. Expecting disconnection may lead to temporary inconsistencies.

Reading the same file from multiple nodes is called read concurrency transparency. SILENUS provides this through replication and non-exclusiveness of file reading. Multiple requestors may read the same file at the same time. This may create a bottleneck if a file is available on only one host. This is avoided through local caching, which makes a downloaded file immediately available to other hosts.

Write concurrency transparency in SILENUS is provided through its unique versioning mechanism. It is explained in the section called “Concurrent File Updates”.

Migration transparency requires that the actual file data can be moved from one to anther host without interrupting work. SILENUS provides standard mechanisms for adding and removing file replicas. The disadvantage is that clients may not immediately know about the adding or removal of a replica, thus either not taking advantage of a local copy, or trying to access a replica that is no longer available. In this case the failure handing mechanisms of SILENUS have to catch it.

Concurrent File Updates

SILENUS supports concurrent file updates through its versioning mechanism. If a file is updated, a new version of that file is created. The old version is not touched. If two clients try to update the same file at the same time a conflict occurs. Conflicts are solved through virtual duplication. This is explained in more detail in the section called “Conflict resolution through virtual duplication”.

File Replication

File replication in SILENUS is supported through the use of multiple byte store services. Whenever a file is requested it may be cached in a local byte store. Replicas of a file may be available in as many locations as needed. To automatically manage these file replicas two optimizer services are used.

The first one of these optimizer services is the Byte Replicator service. It ensures that at least two copies of a file are in the network at the same time. This service is triggered when a new file or a new version of a file is uploaded to the file system. It will then look for another byte store that has enough storage space. It tells the other byte store to replicate the file. After the file is replicated, it will update the metadata stores to have the new location information. This ensures reliability by providing multiple copies. Not only new files can trigger replication. If a byte store service becomes unavailable, all files that where stored on that services are potential candidates for replication: They may now exist in the network only once, not providing reliability. In this case, the Byte Replicator has to trigger another replication. If there are not enough byte stores available, one may be auto provisioned as explained in the section called “Optimizer”.

The second optimizer service, which is not shown in the architectural overview, is a replica deletion service. It will check the number of replicas of a file from time to time. If a file has a high number of replicas, it will free space by deleting some of the replicas. This ensures that more space is available if needed. It can use access data provided by the byte stores to decide which replicas to delete.

Providing multiple file replicas is mandatory for performance and reliability. Files that are replicated to the local host can be accessed much faster than files on remote hosts. To provide reliability a file has to exist in the network multiple times. It can then be downloaded from alternate sources, should one of the providers become unavailable.

On the downside it uses more storage space. Requiring that every file exists in the network twice uses up twice as much storage space. This requirement may have to be relaxed for large files on reliable hosts.

Operating system heterogeneity

To look at operating system heterogeneity two aspects have to be looked at: The potential heterogeneity for client systems and for the hosts running the services.

The use of small client adapters makes the file storage independent from the actual operating system and architecture used on the client. If the client system supports a standard protocol, such as WebDAV or NFS, it can be used. The use of small client adapters makes it easy to add another one should a new client system be developed.

The services may also run on various different host types. Supporting each of them with a custom solution is a major undertaking. A feasable solution is using a virtual machine. An application would have to be written for that virtual machine. Only the virtual machine has to be ported to different platforms. The programs are compiled into intermediate byte code language. This byte code can be reused on any of these virtual machines. This makes code mobility possible.

Using a virtual machine always has a performance impact. Running a virtual machine takes up processor time. Several solutions exist to prevent the performance impact, such as the Hotspot compiler in the Java Virtual Machine. Code that is repeatedly used is adaptively compiled to native machine code. This allows for improved performance. Recent evidence even suggests that the runtime optimization is better than the compile time optimization and Java program run faster than equivalent machine native programs [68].

Fault tolerance

In the SILENUS system, fault tolerance is provided by local replication and dynamic discovery. Each metadata store keeps a full copy of the file metadata. Should a system become disconnected, this local copy will be used. Each byte store on a local host caches the most recently accessed files as described in the section called “Caching download file use case”. These files will be used in the case of disconnection.

Services will be dynamically discovered whenever they are needed. In traditional file storage solutions addresses of servers are manually configured. If a server is unavailable, the request will fail. The dynamic nature of service-to-service makes this unnecessary. As long as there is at least one service of this type available in the network, this service can be discovered and used.

The dynamic discovery also provides for failover. Should a service not respond to a request in a certain time, the request can be sent to a different service. This service will then process the request.

Consistency

There are two types of consistency: File metadata consistency and file content consistency. File metadata consistency is provided with the mechanisms described in the section called “Concurrent File Updates”. File content consistency is provided through the use of versioning.

To ensure file contents are not corrupted derived file attributes such as file length and checksums are stored in the metadata store. A client module can then verify the downloaded file contents against these given attributes. A file can be considered corrupt if the file size or a checksum does not match the given value.

Efficiency

For a complete analysis of SILENUS efficiency please see the section called “Model Performance Analysis”.

Idempotency

Idempotence is the quality of something that has the same effect if used multiple times as it does if used only once. This is usually an issue with asynchronous network messaging without a reply. A message could be sent multiple times to increase the probability that at least one of the messages arrives. A server has to ensure that even though it receives the message multiple times it is only executed once.

The SILENUS system does not show this problem because every method call has a return value. This return value is either data returned from the call or an exception in case of any network failure. The return value can be examined, and the message resent

to a different server. Should any error occur during the method call, the transaction at the service will be aborted and the state of the service will be reset to the previous state through a rollback operation.

In the rare case that an error occurs during the return from the function call this behavior will still have idempotency issues. The service will behave as if the call was successful, while the requestor will see the call as failed and resend the message. This repeated call may be sucessfull or not: If the original request made a change that is unrepeatable, such as deleting a file, the second call will result in an error. If the original request was repeatable, such as setting attributes to a certain value, then the action will just be repeated. As of right now this issue is unsolved. A potential solution would be to use message sequence numbers to detect duplicate messages. This will have to be further investigated.

Security, Access Control, Authentication

The security concept for SILENUS is described in the section called “Security”.

Managing change

One of the challenges in any distributed file system is handling changes. Changes can occur on two levels: File content may be changed or file metadata may be changed.

Change in file metadata

Changes in file metadata occur every time the file content stays the same, but new information for the file is available. This does not only refer to the classical file metadata, such as file owner, or file name, but to all information stored in the metadata store. It includes information such as the directory a file is on or the location of the file contents.

Each change in file metadata triggers an event that needs to be sent to the other metadata stores. Since the metadata stores should contain the same information, it is necessary to synchronize them. An overview on how the metadata stores keep synchronized is given in the section called “Metadata store synchronization”.

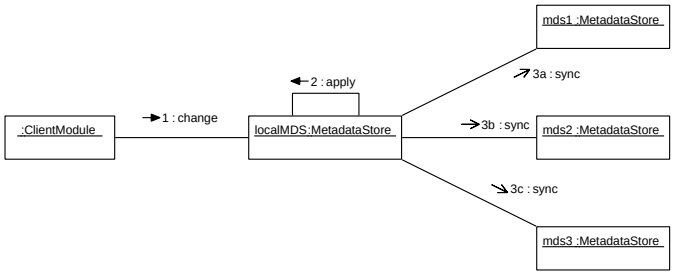


Figure 4.25. A metadata change

Change in file content

Changes in file content are handled through auto versioning. Every time a file is saved, a new version of that file is created. The old versions will stay intact. Instead of an actual change in file data, the change operation now becomes two operations: A change in file metadata, and the upload of new file content. The change in file metadata is handled with the same metadata synchronization process as for the regular metadata change. The same upload process as for a new file handles the change in file data.

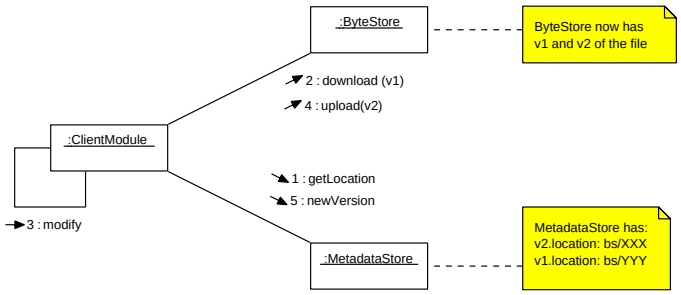


Figure 4.26. Change of file content

Partially modified files will have to be handled differently. If only a small part of a large file changes, it is inconvenient to create a full new file version. In this case, the change will now become three events: First, the existing version of the file on the byte store is pseudo-deleted. The location information will be removed from the metadata store, but the file content will stay in the byte store. This will ensure that no other processes accesses the file at the same time. After that, the file contents can be modified. When the modification is done, the new file version will be created pointing to the modified file.

Auto versioning may lead to many versions. If a file is saved regularly, auto versioning can create multiple and overly many backup copies. One example for this is the auto save feature in common text-processing applications: It will automatically save an open document every 5 minutes. In a 4-hour work session this amounts to 72 versions. In this case it is probably not worth keeping all versions. Traditional backup systems keep one version per day. An optimizer service needs to be added which supports automatic removal of old versions. One potential algorithm would check the time between versions. A version that is superseded by another version after a short interval is probably less worth keeping than a version that is replaced after a longer time span. Versions that are too old may also be deleted by this version garbage collector service. Another approach would be to use user-defined attributes, such as "frozen version" or "final version". These versions could be kept for a certain timespan, while "work versions" of the same file may safely be deleted. This topic is open for future research. Some possible approaches are shown in [44] and [45].

Metadata store synchronization

As shown in the section called “Change in file metadata”, the metadata stores are synchronized. When a metadata store receives a change request, it applies that request to its own store and sends out change information to all other available metadata stores. In an ideal environment with all metadata stores available and no events happening at the same time this would lead to consistency. Unfortunately these two assumptions are not true.

An algorithm has to be found that provides consistency across multiple metadata stores. First, consistency has to be defined. Then, an order of events has to be established to know which events are newer and may override older events. Once this is done, an algorithm can use that information to provide synchronization that leads to more consistency.

Consistency

Consistency needs to be defined before the term can be used. Three types of consistency are introduced: Global consistency, group consistency, and local consistency.

Global consistency requires that all metadata stores have the exact same informational state. If one of the metadata stores receives a change request and applies it, the system will become globally inconsistent. After all metadata stores are updated, the system will be global consistent again.

Group consistency requires all metadata stores that are currently reachable from one metadata store to have the same informational state. If one of the metadata stores receives a change request and applies it, the system will be group inconsistent. After it sends an update request to all available metadata stores, the system will be group consistent.

Local consistency applies to a single metadata store. If a change request is made and applied, the metadata store is already locally consistent. That change should be persistent in the metadata store until changed again on either the same metadata store or from another metadata store that has the same state.

Consistency requirements

After defining consistencies, it can be looked at what types of consistencies are desirable and possible.

Global consistency is the most desirable but impossible with the given requirements. The system should support disconnected operation. As long as at least one node is disconnected from the rest of the nodes, it is impossible for any update packets to reach that node. Therefore the disconnected nodes will never be able to have the same state until they are reconnected.

Group consistency is very desirable and achievable. As seen in the definition, any metadata store can send its update packets to all connected metadata stores. The updates can then be applied to these other metadata stores. There are two problems that can occur. The first one is a reconnected metadata stores. Theses will not have received all update packets that where sent before. They need to detect this time lapse and synchronize accordingly. If two conflicting changes where made, they need to be resolved. The second case happens if two conflicting changes are made to two metadata stores that are connected. Each metadata store will try to apply the change and then send update packets to the other stores. A conflict occurs which needs to be resolved. This second case is just a special version of the first case. It therefore does not need special handling.

Whenever an update packet from another metadata store is applied, it must be ensured that the local consistency on the metadata store is kept. Update packets from other metadata stores can destroy local consistency when they contain changes from an older state. It must therefore be ensured that a conflicting update packet is only applied if the remote metadata store knew about the current state of the local metadata store.

Measure of consistency

We can introduce a measure of consistency. We will assume that we have a number N of metadata stores. Each metadata store will know about a number E of metadata change events. For each individual metadata store we define its consistency as:

$$C_{mds} = \frac{E_{mds}}{E_{total}}$$

This measurement will always have a value in the range [0..1]. It will be 0 for a new metadata store. If the metadata store if completely synchronized (global consistent) then it will be 1.

To get a measure of consistency for the whole system we need to sum up the known events in every metadata store:

$$C_{total} = \frac{\sum E_{mds}}{N \times E_{total}}$$

This measurement will always have a value in the range]0..1] as long as $N > 0$ and $E > 0$.

For both measurements, a higher value, as close to 1 as possible is desirable. It is therefore mandatory optimize the system to reach a value as closest to 1 as possible. Global consistency is achieved when $C_{total} = 1$.

To reach global consistency, each node has to know about all events that have happened in the system. It is therfore necesarry to communicate the events to other nodes, and apply these events locally.

In the SILENUS model, group consistency is achieved by sending events to all connected metadata stores. Whenever an event occurs, it is sent to all metadata stores that can be reached. When a node is disconnected, it will not receive any events. It therefore has to be ensures that it receives the events onces it is connected with the other nodes again.

Order of events

In a distributed system, the state of the system depends on the time that this system was in this state. It is therefore required to have a notion of time. Time has do define an order of events. This order can then be used to recreate a change log or to decide which events to apply. First, existing algorithms for distributed order of events are investigated, and then a new algorithm is proposed.

To provide an order of events, a notion of time t, where t_e defines the time for an event e, must provide the following properties:

- t must be strictly monotonic increasing for every event.
- $t_{e1} < t_{e2}$ iff e1 happened before e2
- $t_{e1} > t_{e2}$ iff e1 happened after e2
- $t_{e1} = t_{e2}$ iff e1 and e2 are the same event on the same machine
- $t_{e1} \parallel t_{e2}$ iff e1 and e2 happen at the same time on different machines.

The first guess at providing an order of events is to use the real time clock. Every node would get the time from a global time server. It would then be easy to find out which events happened when and which events should override older events. Unfortunately, this would require all nodes to keep exact time and reconnect to the global time server often enough. The GPS system is an example of a distributed system that uses very exact time. Unfortunately, such an exact time is not available in real world applications. Synchronization with a global time service is impossible for disconnected hosts. The computer clock can give an approximation for the time, but it is not always exact. The local clock may be set to a faulty time on purpose. Also, real time does not provide a notion of events happening in parallel. [70, 41]

Instead of using a global absolute clock a logical clock is used. A logical clock is a monotonically incrementing software counter. It will start out at time zero. It is required that there is at least one clock tick between two events, so t is increased after every event. This time is strictly monotonic and allows comparison. If all events are time stamped, it becomes possible to reconstruct the order in which events occurred. This works very well for a single system, but shows limitations when applied to a distributed system. [42]

Timestamps work fine when all network messages arrive before new messages are sent of. Figure 4.27, “An event diagram using logical clocks” shows an example for this type of messaging. The individual elements on one process can be ordered. Every process can detect in which order events where generated.

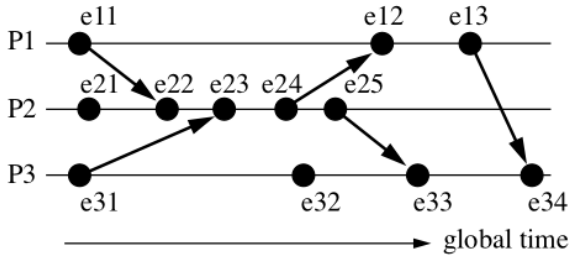


Figure 4.27. An event diagram using logical clocks

This system shows its limitations when network messages are lost. This may have happened due to one service being disconnected. Events that originated in different processes cannot be ordered. Figure 4.28, “An equivalent event diagram” shows an event diagram that is equivalent to the one in Figure 4.27, “An event diagram using logical clocks”. However, the events now happened at different times. Without global knowledge this is impossible to detect from within the processes.

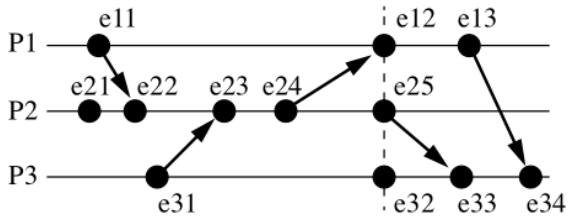


Figure 4.28. An equivalent event diagram

To order events in a distributed system, a time stamp on the local process is not enough. Every event needs to be time stamped with the global time at every system. This leads to vector clock timestamps. In vector time, every system keeps its own counter. A vector clock V contains the logical clock for every connected system. Figure 4.29, “Global vector time” shows an example of events tagged with vector time.

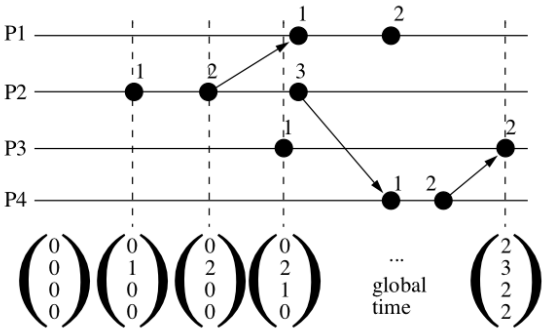


Figure 4.29. Global vector time

Unfortunately, a system with fully working vector clocks would need a reliable observer. In a truly distributed system this is impossible. Instead, the vector time is approximated with the best knowledge of a system. In a vector clock system, each node keeps the knowledge of its own logical clock and the logical clocks of all its peers. This clock vector is appended to all network messages. Other systems can then use this information to update their own vector clock and to check if the received message was current. Vector clocks can be used to provide total ordering of events in a system. Figure 4.30, “Vector time propagation” shows an example of such an ordered system. [43]

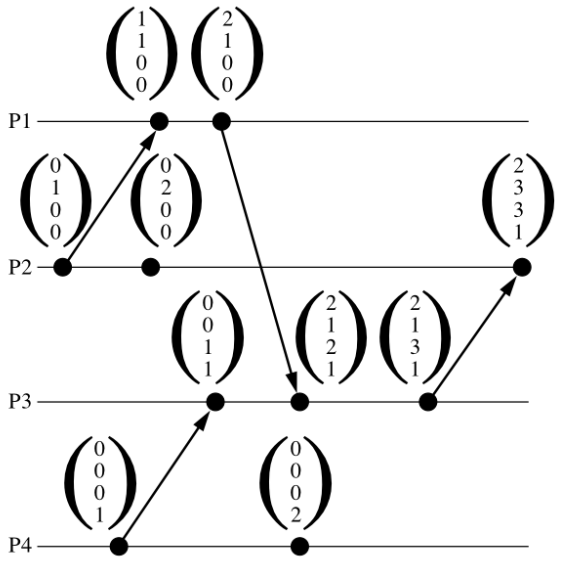


Figure 4.30. Vector time propagation

The vector clock algorithm is as follows:

- If an event occurs locally, increment the own clock.
- If an event is received, set each clock to the maximum of the clock received and the known clock. Increment your own clock by one.

Using this algorithm we can now compare vector clocks and define an absolute ordering of events. We will compare the received time vector V_r with the local time vector V_l.

- If all components of $V_r >= V_l$ (but $V_r != V_l$) then the received message is newer than the local state. Receive all events from the remote system and apply them.
- The case that all components of $V_r <= V_l$ (but $V_r != V_l$) is not possible. A metadata store will increment its own clock before sending out events, therefore at least the clock component V_r which corresponds to the sender must be greater than the component stored at the receiver side.
- If some components of $V_r > V_l$ and some components of $V_r < V_l$ then some events happened in parallel. In this case, the receiving metadata store needs to retrieve all events from the sending metadata store and merge the contents.

If a merge occurs there are two possible cases:

- None of the events concern the same files. In this case, the received events can be applied directly.
- If some of the events concern the same files, a conflict occurs. This conflict needs to be resolved.

The algorithms described in this section are commonly known and verified. Unfortunately, the traditional vector clock algorithm shows some problems, which will now be discussed.

Dual-Clock Time Vectors

The problem with this time vector algorithm is that it does not keep accurate track of the actual changes, but rather of the messages. According to the original algorithm the clock is incremented every time an event is received. If this time vector is then sent to a third party, this other host could not distinguish if the time was increased because a new event occurred or because another event was merged. Figure 4.31, “Vector clock problem” shows an example.

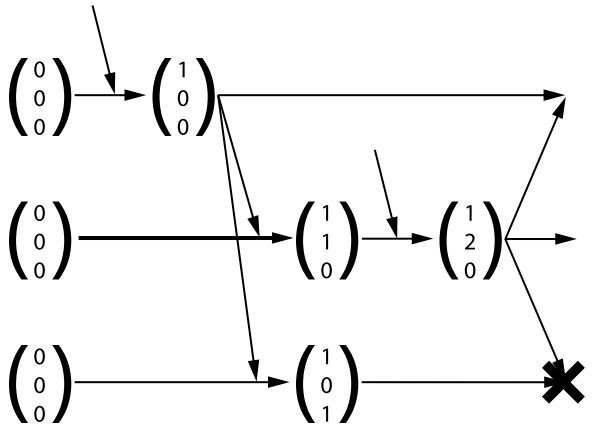


Figure 4.31. Vector clock problem

In this particular example, the first node receives an external event. It increments its own clock and propagates the event to all other nodes. These nodes apply the change and increment their own time vector. Another event occurs on the second node. It increments its time vector and notifies all other node. The first node will just apply the changes. The third node, however, will detect a conflict.

To solve the problem with single-time vector clocks a new dual time vector-clock system is introduced. A local timer counts only events that originated locally, whereas the global timer counts both local and external events. The time vector now contains both clocks for all nodes. The local component is used for time comparison, whereas the global component is used to recreate change data. When comparing these time vectors, only the local components of the time vector are compared. It will result in one of these four possibilities:

- If forall n: $Vl_r(n) = Vl_l(n)$ then $V_r = V_l$. Both nodes have the same information. Merge global components by setting them to the maximum of the current value and the received value.
- If forall n: $Vl_r(n) >= Vl_l(n)$ and $Vl_r != Vl_l$ then $V_r > V_l$. The received message is newer, all changes may be applied and the components updated by setting every component to the maximum of the current and received value and increasing the own global clock.
- If forall n: $Vl_r(n) <= Vl_l(n)$ and $Vl_r != Vl_l$ then $V_r < V_l$. The received message is older. Ignore the events but merge global components by setting them to the maximum of the current value and the received value
- If there exists an m, n: $Vl_r(n) > Vl_l(n)$ and $Vl_r(m) < Vl_l(m)$ then $V_l \parallel V_r$. Some events have happened in parallel. A potential conflict has occurred that must be resolved. After resolving the conflict, set every component to the maximum of the current and received value and increase the own global clock.

This algorithm shows a lower rate of false conflict detection. Figure 4.32, “Dual-clock time vectors with local and global counter” shows the same events as Figure 4.31, “Vector clock problem”. Since no actual conflict occurred, none is detected.

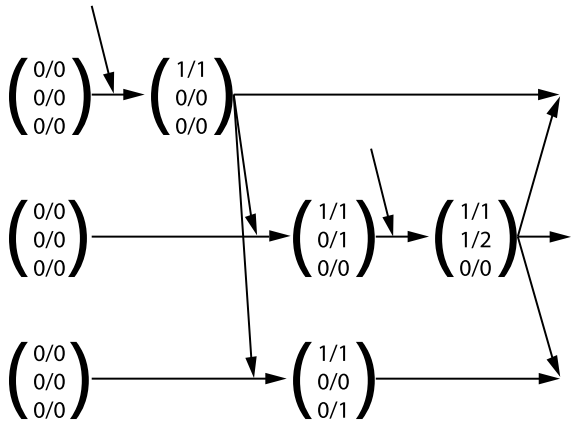


Figure 4.32. Dual-clock time vectors with local and global counter

Properties of Dual-Clock Time Vectors

Since the dual-clock time vectors are a new algorithm, we have to prove its properties. The requirements for time given in the section called “Order of events” were:

- t must be strictly monotonic increasing for every event.
- $t_{e1} < t_{e2}$ iff e1 happened before e2.
- $t_{e1} > t_{e2}$ iff e1 happened after e2.
- $t_{e1} = t_{e2}$ iff e1 and e2 are the same event on the same machine.
- $t_{e1} \parallel t_{e2}$ iff e1 and e2 happen at the same time on different machines.

Each of these required properties is now investigated. For each property, local events and remote events have to be investigated. The dual-clock time vector can without loss of generality be defined at:

$$V = \begin{pmatrix} l_1 / g_1 \\ \vdots \\ l_{self} / g_{self} \\ \vdots \\ l_n / g_n \end{pmatrix}$$

To prove that t is strictly monotonic increasing it must be shown that $V_{new} > V_{old}$. The algorithm states that in the case of a local event both the local and global clock of the current system have to be increased, therefore:

$$V_{new} = V_{old} + \begin{pmatrix} \vdots \\ 1/1 \\ \vdots \end{pmatrix} = \begin{pmatrix} l_1/g_1 \\ \vdots \\ l_{self}/g_{self} \\ \vdots \\ l_n/g_n \end{pmatrix} + \begin{pmatrix} \vdots \\ 1/1 \\ \vdots \end{pmatrix} = \begin{pmatrix} l_1/g_1 \\ \vdots \\ l_{self}+1/g_{self}+1 \\ \vdots \\ l_n/g_n \end{pmatrix}$$

We can immediately see that forall n: $Vl_{new} >= Vl_{old}$ and $V_{new} != V_{old}$. Therefore the requirement $V_{new} > V_{old}$ is satisfied.

The second case is the case of received remote events. There are four sub cases:

1. $V_r = V_l$. In this case, no event has happened; V_l does not have to increase.
2. $V_r < V_l$. In this case, the received event is older; V_l does not have to increase.
3. $V_r > V_l$. In this case, the received event is newer; V_l must increase.
4. $V_r \parallel V_l$. In this case, events have happened in parallel; V_l must increase.

In the sub cases 3 and 4 V_{new} is defined as:

$$V_{new} = \max(V_{old}, V_r) = \begin{pmatrix} \max(l_{old,1}, l_{r,1}) / \max(g_{old,1}, g_{r,1}) \\ \vdots \\ \max(l_{old,self}, l_{r,self}) / \max(g_{old,self}, g_{r,self}) \\ \vdots \\ \max(l_{old,r}, l_{r,r}) / \max(g_{old,r}, g_{r,r}) \\ \vdots \\ \max(l_{old,n}, l_{r,n}) / \max(g_{old,n}, g_{r,n}) \end{pmatrix}$$

The use of the max function satisfies the condition forall n: $Vl_{new} >= Vl_{old}$. $V_{new} != V_{old}$ follows directly from the sub case selection, would $V_{new} = V_{old}$ then sub case 1 would have been selected. This proves that the dual vector clock algorithm satisfies the requirement: t must be strictly monotonic increasing for every event.

The next property that must be proven is that $t_{e1} < t_{e2}$ iff e1 happened before e2. This property is a direct result from t being strictly monotonic increasing for every event.

The properties $t_{e1} > t_{e2}$ iff e1 happened after e2 and $t_{e1} = t_{e2}$ iff e1 and e2 are the same event on the same machine also follow directly from the property that t is strictly monotonic increasing.

The property $t_{e1} \parallel t_{e2}$ iff e1 and e2 happen at the same time on different machines can be proven as follows. Assuming two nodes N1 and N2 have the same vector V_{start} at some point:

$$V_{start} = \begin{pmatrix} l_1 / g_1 \\ \vdots \\ l_{N1} / g_{N1} \\ \vdots \\ l_{N2} / g_{N2} \\ \vdots \\ l_n / g_n \end{pmatrix}$$

After two events happened in parallel on both machines, the time vectors for nodes N1 and N2 will be:

$$V_{N1} = \begin{pmatrix} l_1 / g_1 \\ \vdots \\ l_{N1} + 1 / g_{N1} + 1 \\ \vdots \\ l_{N2} / g_{N2} \\ \vdots \\ l_n / g_n \end{pmatrix}; V_{N2} = \begin{pmatrix} l_1 / g_1 \\ \vdots \\ l_{N1} / g_{N1} \\ \vdots \\ l_{N2} + 1 / g_{N2} + 1 \\ \vdots \\ l_n / g_n \end{pmatrix}$$

When comparing the two time vectors, $V_{N1,l}(N1) > V_{N2,l}(N1)$ and $V_{N1,l}(N2) < V_{N2,l}(N2)$. Given the definition of parallelism these vectors are detected as $V_{N1} \parallel V_{N2}$.

The other direction is to provide that if $V_{N1} \parallel V_{N2}$ then there must be events on more than one host. This can be proven by looking at the algorithm: The only time the own local clock increases is if an event happened locally. Therefore, if more than one local clock has changed, there must be events that happened on more than one host.

The dual-clock time vector algorithm still supports all the properties that where required originally. It can therefore provide a reliable order of events for a synchronization mechanism.

Performance of Dual-Clock Time Vectors

When looking at performance for dual-clock time vectors, the different operations have to be looked at first. There are three operations that are needed: Comparing time vectors, increasing time vectors, and merging time vectors.

The easiest case is increasing a time vector. In the case of a local event, two entries in the vector are changed. In the case of an event from the outside, one component in the vector is changed. The required time is therefore O(1).

Comparing time vectors requires a comparison of every component. The required time is therefore O(n) where n is the numbers of participating nodes total in the system. In the case of the SILENUS system, this is the number of metadata stores.

Merging time vectors requires again a comparison and setting of every single component. The required time is also O(n).

When a host rejoins an existing network, it will discover all other hosts. It will have to compare its time vector with all hosts on the system. It will only have to merge the time vector with the first host it encounters as it will have the current time afterwards. It then hast to increase its own clock. The total time required for synchronization is therefore $O(n)*O(n) + O(n) + O(1) = O(n^2)$.

A SILENUS deployment with 1000 metadata stores could therefore need up to 10^6 operations for its synchronization. The operations required for synchronizing dual-clock time vectors are all simple integer operations. An integer operation uses very few clock cycles. Assuming about 100 clock cycles per operations, which is probably too high, it would take 10^8 clock cycles. Current computers running with processor speeds in the gigahertz range can run 10^9 clock cycles per second. Synchronization with 1000 nodes would therefore take less than 1/10 of a second on a modern computer.

Conflict avoidance

Even if there is a potential conflict, there is, and in most cases will not be, an actual conflict. The ordering of events using the dual vector clocks only gives the information that two or more events have happened at the same time.

These events must be related to create an actual conflict. In the case of SILENUS metadata, events are only related if they apply to the same node. Event applying to different nodes can therefore be applied without problems. Some event may not make sense together, such as deletion of a directory, and creation of a new file in the same directory, but they do not conflict.

The relation can even be specified more exactly on a field basis. If two different fields on the same node have changed, these can be merged without conflicts. If a file is renamed on one node, and modified on another, both changes are not in conflict with each other.

The last-changed-on metadata can never create a conflict. This data is updated every time any data of the node changes. As such, it would always lead to a conflict. However, this data is only used for conflict resolution. It is therefore not important what the actual value is.

Conflict resolution through virtual duplication

One possible conflict resolution mechanism is virtual duplication. Virtual duplication addresses the issue of local consistency and requires no direct user interaction.

An automatic conflict resolver will require no user interaction. If a file is modified in multiple places, the system should be able to provide a conflict handling strategy. This strategy should not require user interaction. In most environments it is impossible or impracticable to ask the user which conflicting option to choose. This should be done automatically.

One issue with automatic conflict management is that it can break local consistency. A change may be made to a local metadata store. Then this metadata store gets synchronized with another metadata store where a conflict occurs. The users on both metadata stores expect their action to take precedence over the conflicting action from the other user.

The Coda distributed file system introduced virtual duplication. It is used in the code file system to resolve conflicts between two versions of the same file with updated file content. In SILENUS this method is not applied to file content. It is applied to all changes in file metadata. Changing the file contents adds a new version and therefore triggers a change in file metadata. But other changes in file metadata are possible that may need to be resolved.

Virtual duplication provides a file under three different names: It will append a `.version` to the files depending on which store it was modified. It will also provide the file under its original name, as a soft link pointing to the version that was produced on this particular host. Figure 4.33, “Virtual duplication example” shows an example.

```
Store A contains: bla.txt (version 1)
Store B contains: bla.txt (version 2)
```

Stores get disconnected

File bla.txt is modified on store A

File bla.txt is modified on store B

```
Store A contains: bla.txt (version 2.A)
Store B contains: bla.txt (version 2.B)
```

Stores get reconnected

```
Store A shows: bla.A.txt (version 2.A)
               bla.B.txt (version 2.B)
               bla.txt -> bla.A.txt
Store B shows: bla.A.txt (version 2.A)
               bla.B.txt (version 2.B)
               bla.txt -> bla.B.txt
```

Figure 4.33. Virtual duplication example

Solving conflicts this way minimizes direct user interaction. Users can manually resolve the conflict without any special tools whenever they need to. Consistency on the same system is provided through soft links.

This way of resolving conflicts has the drawback that inconsistencies between different stores may now exist. These inconsistencies are only in the file name and not the file data. Conflicts will still have to be resolved manually.

The switchback problem

One hardship with independent synchronization is the switchback problem. Figure 4.34, “The switchback problem” gives a graphics illustration. The switchback problem occurs if two distinct stores that contain the same information. These stores synchronize and merge at the same time with two other stores containing another set of information. If they resolve the conflict differently then they will again create different versions, which will lead to a conflict.

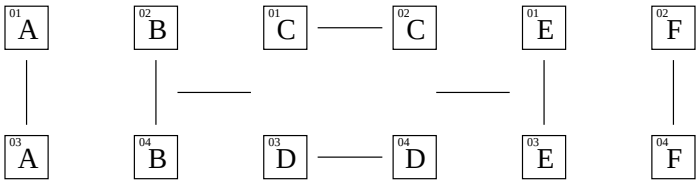


Figure 4.34. The switchback problem

This problem can be solved if both metadata stores resolve a conflict in the exact same manner and arrive at the exact same solution. This requires two things when using virtual duplication: The names must be exactly the same and the generated uids must be exactly the same. Figure 4.35, “A solution for the switchback problem” shows a graphical illustration.

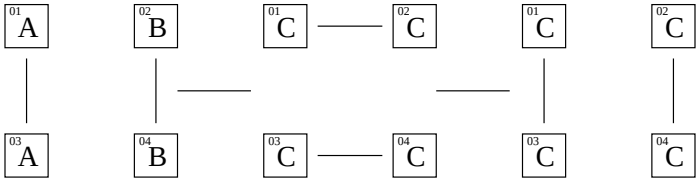


Figure 4.35. A solution for the switchback problem

The first problem is that the names of the files must be exactly the same. In the algorithm outlined above, the id of the synchronizing metadata store is used as an extra file name. This is insufficient, as it leads to different file names depending on the nodes involved in the synchronization. Instead, the id of the node that has last changed the metadata is used. This information was stored in the last-changed-on attribute.

Generating the same new ids is the second problem. To provide support for this, the original id extended with the last-changed-on information. The link will keep the original id. The two conflicting versions will get the original id with the last-changed-on information appended. This may happen again, since there may be another conflict in one of those files.

In this section, a complete solution to synchronize metadata stores was given. To provide proper synchronization, consistency among metadata stores was defined. A new algorithm for distributed time based on dual-clock time vectors was introduced. A conflict resolution algorithm based on virtual duplication was described. The switchback problem and a possible solution where shown. Using the methods described in here,

the metadata stores can synchronize with each other very efficiently. However, the algorithms described here also apply to other areas of distributed applications: The dual-clock vector time algorithm can be used for any order of events in a distributed system. The conflict resolution algorithm can be applied to any key-value based data.

Security

SILENUS has two needs for security: Authentication and Privacy. Users need to be securely identified and it must be made sure that only users with the right privileges can modify data. Data stored in the SILENUS system must be kept private. This is especially difficult since data is transmitted over an open network and may be stored on insecure nodes.

The classic security concept is authenticating with the node that provides the service. This approach works well in a traditional client - server environment. In a large distributed environment this would require the user credentials to be replicated among all service providers. This is an administrative challenge. It is also not very secure, as credentials may be intercepted or read by local administrators and users in the network.

A better approach uses tokens. Instead of authenticating with the service provider, a user will authenticate with one central server. The server will then issue a token to the user. This token can be used to authenticate with services providers which will verify the tokens authenticity with the central authentication server. This approach is used by Kerberos [46]. It works very well for smaller distributed applications, but does not scale beyond one organization. It also requires one particular node to be always available.

A problem with most existing security concepts is that they don't allow existing authentication and user databases to be re-used. Every system has its own user and password database. Most system can import users from other systems, but importing passwords is very often a problem. Passwords are usually stored in some encrypted format and cannot be exported. The current solution is to adapt applications to different credential providers.

Special credential mechanisms such as fingerprint scanners and smart cards are hardly ever supported. In few cases, some applications such as computer login are adapted for these devices. However, the keys stored on such a system could be used for all kinds of services.

What is needed is a scalable security system that makes use of existing credentials. It should support different administrative domains, but still provide one unique privacy and authentication mechanism.

Proposition

To solve this problem the following assumption is made: In large scale system it is more important to recognize returning users. It is not important which identity a user has as long as the user has the same identity when connecting again. Using this assumption a user could provide own credentials. As long as it is ensured that the credentials are safe the user can be uniquely identified.

Trusted third party model

Allowing the user to provide own credentials can lead to an explosion of user accounts. Therefore a user account has to be verifiable by a trusted source. This is commonly referred to as a trusted third-party model.

In a trusted third part model a user authenticates with an authentication service. The authentication service will then provide verification that a given user is always the same. The service provider can then verify that a user is the same. Figure 4.36, “Basic trusted third party model” gives an example.

Figure 4.36. Basic trusted third party model

The list of third parties should be small and change seldom. This information will have to be configured on every service provider. It should change as little as possible. Every change would require administration.

A trusted third party can be any service providing users. It could be an LDAP server, a windows domain server, a Kerberos server, or a trusted party signing public keys. It is only required that the server can verify users.

Decoupling the authentication service

The basic trusted third party model requires the service provider to be able to talk to the authentication service directly. This is undesirable, and very often not possible. It is also not defined how the credentials are passed to the service provider.

A standard for credentials needs to be defined. This standard should be common, and should allow verification without talking back to the original service. X500 is such a standard. It is based on public key cryptography.

The authentication provider will have two additional requirements: It will have to provide a public key which is signed by a trusted third party and it will have to be able to sign small network requests. The actual private key will never have to leave the authentication service. The service provider can then verify the public key with its own trust-store. It can verify all network requests with the public key. Figure 4.37, “Authentication with public-key cryptography and trust-store” gives an example. This model works very well with authentication services that provide public key cryptography such as smart cards.

Figure 4.37. Authentication with public-key cryptography and trust-store

To provide support for existing models that are not based on X500 an authentication adapter service is needed. This adapter service provides the required services and uses the existing authentication service as back-end. If will have to be run on a secure system.

It is important that the adapter service is able to create new keys for users that have not yet authenticated themselves with this services. If a new user authenticates herself, a new key must be created. This key must be automatically signed by the adapter. The signature from the adapter must be listed as a trusted third party in the truststores. Figure 4.38, “Authentication with public-key cryptography and trust-store” gives an example of the complete authentication process.

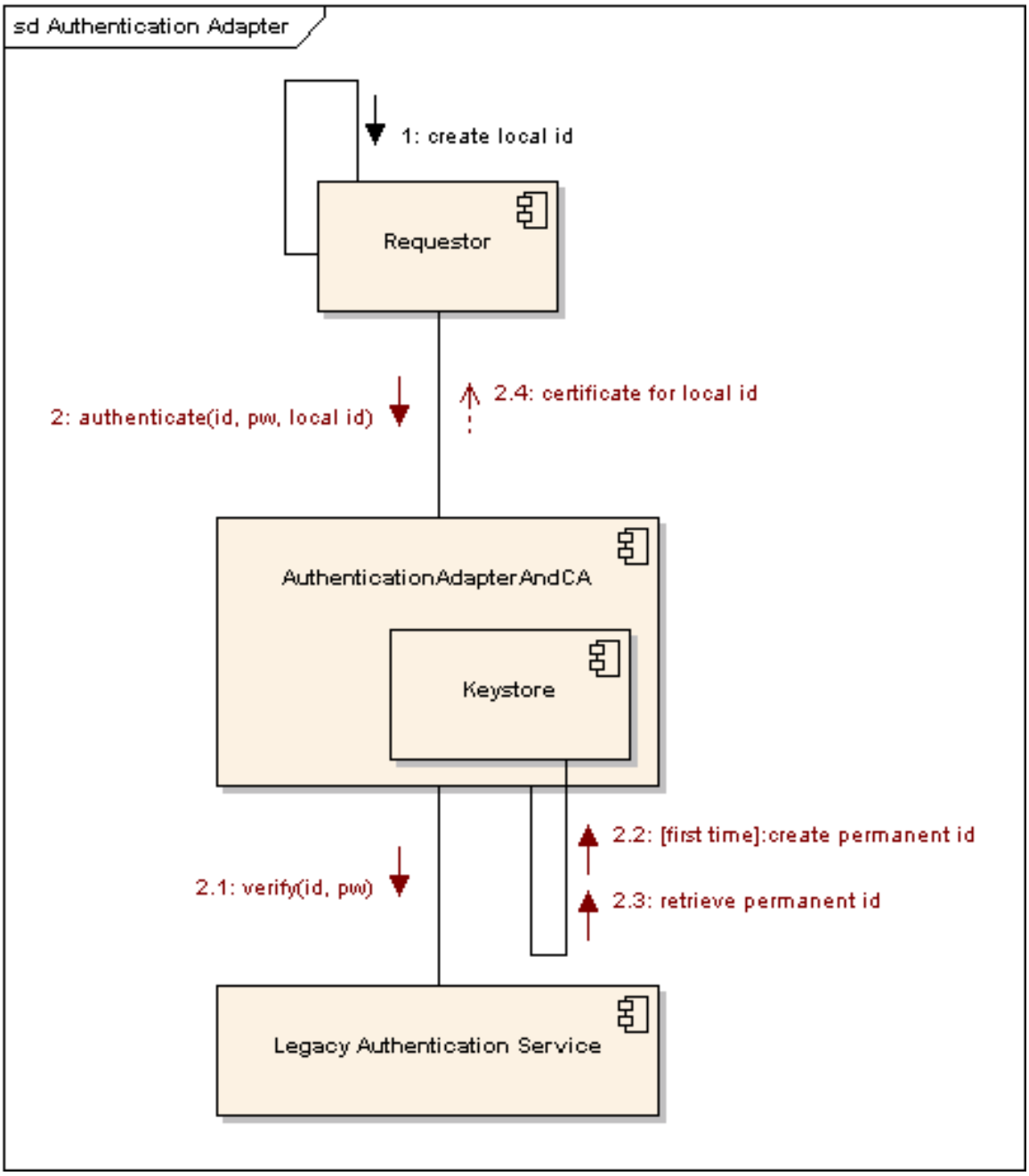


Figure 4.38. Authentication with public-key cryptography and trust-store

Privacy

A requirement for a data grid is providing privacy. In a data grid data is stored on different nodes in the network. Each one of theses nodes may have a different administrator or may be compromised. It is therefore important that a security system provides privacy. Data stored in the network by a user should only be readable by that user. Users listening on the network or local administrators should not be able to read all data.

To provide privacy the same security system should be reused. It is now possible to recognize returning users, so it should be possible to give access only to the user that has originally stored the data. In typical systems this happens using symmetric encryption. A user encrypts data with a secret key, and can decrypt the data using the same secret key.

It is not feasible to require the user to manually keep track of the encryption keys needed for their data. Therefore the encryption key itself is stored with the data. To ensure that it is not compromised, the key data is encrypted using public key cryptography.

Public key cryptography is already provided by all authentication providers for signing messages. This system can therefore easily be extended to provide support for encryption and decryption of data. Since the key data is small the workload on the authentication service is not significantly increased.

When the information is retrieved the encrypted symmetric key is retrieved along with the data. The service requester will forward this information to the authentication service which will then decrypt the key data. This decrypted key data can then be used to decrypt the actual file data.

Roles

The system described so far provides support for single users but not for groups of users and different roles. An extra step is needed to support user groups and roles. Belonging to a group or a role is the same in this context. If a user belongs to the group administrators she may assume the role of an administrator. To support user roles, a new service called "Role Manager Service" (RMS) is introduced.

Role Manager Service

The role manager service provides mapping from user to roles. Given a user handle, the role manager service will provide credentials for all roles this user belongs to. There may be multiple role manager services.

The role manager service acts like a service provider and an authentication service in one. A user will authenticate against an existing authentication service. Using the provided credentials, the user will authenticate again against the role manager service, which will in turn provide credentials for the roles this user is in. It will work similar to the existing authentication services. Figure 4.39, “Authentication via role manager service” gives an example of a role manager use.

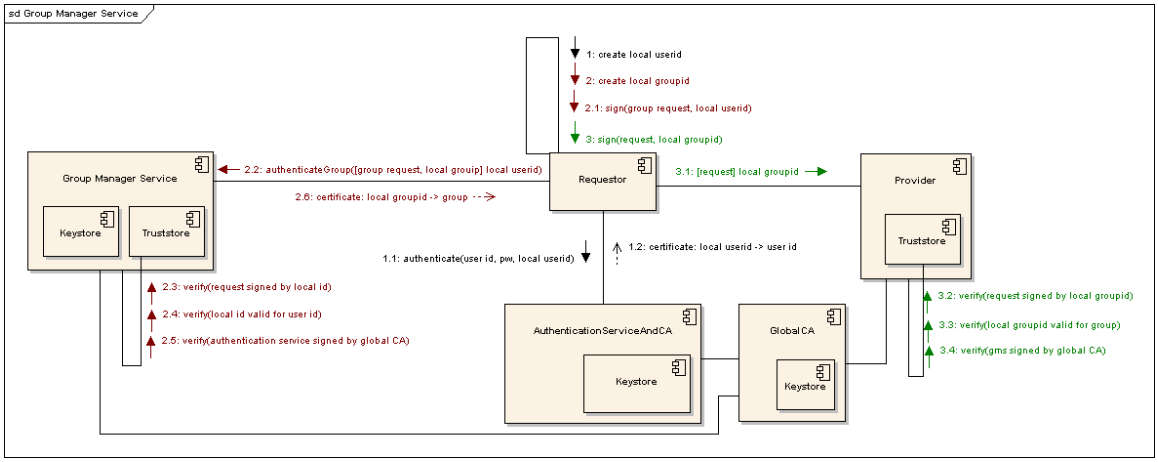


Figure 4.39. Authentication via role manager service

Using multiple role manager services provides scalable administration. A smaller part of a larger organization, such as a department at a university, may provide their own role manager service. The credentials from this role manager service can be used to authenticate access to local resources, such as file storage or lab access.

Splitting up security credentials into different authentication services and role manager services provides support for scalability. Users can have centrally managed accounts, but their privileges may be controlled by single parts of the organization. Management for these roles can be delegated to local administrators without giving them full access.

Nomadic RMS

The role manager service may be replicated to different nodes. Since the RMS has a copy of the secret keys for different roles, it may only be replicated to nodes that are trustworthy. In most cases, server computers in the local part of the organization are trustworthy, and sometimes client computers, if their users do not have local administrator rights. Replicating an RMS gives the usual benefits of replication, such as reliability and scalability.

To support disconnected operation, a subset of the roles existing on a particular RMS may be copied. A user may need to use her credentials while not connected to the network, to access data stored on a local nomadic system, such as a laptop. To provide access, a subset of the roles may be copied onto the users host. This is supported by the nomadic RMS. The nomadic RMS will replicate only the roles that contain a particular user. The user can then access these credentials locally, providing support for disconnected operation.

The local administrators for the hosts running a nomadic RMS must be trustworthy. A local administrator has full rights on the host and is able to extract and intercept all keys, undermining the security. Therefore only keys of roles the user is part in may be copied. The administrator of the main RMS may also specify which roles may be copied at all. If a user running a nomadic RMS is removed from a role all keys need to be changed and all existing data needs to be re-encrypted.

Model Performance Analysis

To analyze the performance of the SILENUS system, we have to first identify what to analyze and how. The performance is expressed in terms of time T needed to perform a certain operation. We define the following terms:

cm

The client module. May be the Service UI, the WebDAV adapter, or any other.

sf

The SILENUS facade module.

mds

The metadata store service involved.

bs

The byte store involved.

tm

The transaction manager.

lbs

The local byte store, for the caching use case.

T(op)

Time for an operation.

P(service)

The time this services needs to process the given operation.

BW(service1,service2)

The bandwidth between two services, given in data / time. If there are multiple links between both services this is the bandwidth of the narrowest link.

L(service1,service2)

The latencty between two services. This gives the time it takes to send a zero-sized packet from service1 to service2, without waiting for a return message. If there are multiple links between both services this is the total time to traverse all links.

size

The size of the file to be uploaded / downloaded.

We will also make the following assumptions:

- Network links are symmetrical. That is BW(s1,s2) == BW(s2,s1) and L(s1,s2) == L(s2,s1). This true on most network connections. DSL and cable Internet are notable exceptions.
- The size of method call and return messages is much smaller than the given bandwidth. In most cases, the method call and return messages will not contain a large data payload. The model is greatly simplified by not taken the bandwidth into account for these messages
- All services are available and already discovered. Discovering services and switching over to different services whenever a service becomes unavailable uses extra time. To simplify this model, this is not considered.

Given these definitions and assumptions several use cases can now be analyzed. To get an idea what these time values mean here are some network measures. These measurements where gathered experimentally and are to be understood as estimates only.

Type	Bandwidth	Latency
100 MBit LAN	11 MByte/s	0.08 ms
11 MBit WLAN	1.2 MByte/s	0.8 ms
Cable Modem	0.45 / 0.06 MByte/s	25 ms
Internet (USA -> Germany)	0.1 MByte / s	75 ms

Table 4.1. Examples of network types in use today

Browse files

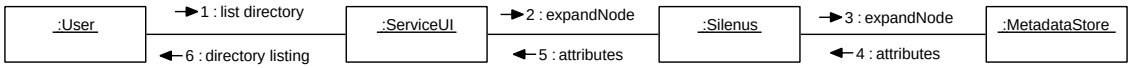


Figure 4.40. Browse files use case

The browse files use case is very straightforward. Adding the times for the sequence we get:

$$T_{browse} = P_{cm} + L_{cm,sf} + P_{sf} + L_{sf,mds} + P_{mds} + L_{mds,sf} + P_{sf} + L_{sf,cm} + P_{cm}$$

$$T_{browse} = 2P_{cm} + 2L_{cm,sf} + 2P_{sf} + 2L_{sf,mds} + P_{mds}$$

Equation 4.1. Browse file performance

There are two factors that could be dominant here. If we assume that the hosts are much faster than the network (L >> P) we can reduce this to:

$$T_{browse} = 2L_{cm,sf} + 2L_{sf,mds}$$

Equation 4.2. Browse file performance in slow network

Which would estimate to a time between 0.3 ms on a local network to 300 ms to an Internet network (assuming all 3 services are located on different continents, which is unlikely).

Upload files

To analyze the file upload speed we will have to look at both the pull and push file upload. In this case we are only interested in the speed for the actual user module, and not in anything that happens in the network afterwards. For the performance analysis we will stop as soon as the client module is done. The timing for both cases is therefore almost identical. There are extra messages in the push file upload use case.

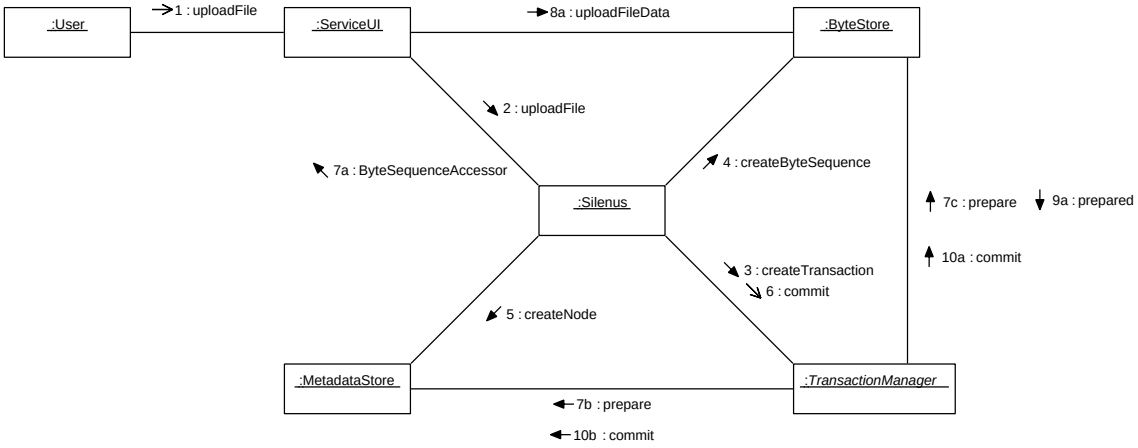


Figure 4.41. Push file upload use case

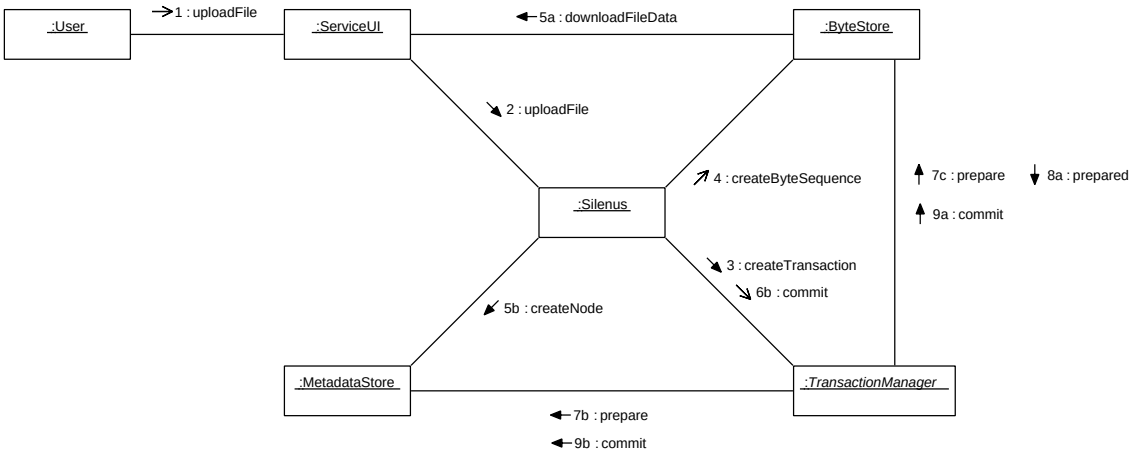


Figure 4.42. Pull file upload use case

$$
\begin{aligned}
T_{pull} = \quad & P_{cm} + L_{cm,sf} + P_{sf} + L_{sf,tm} \\
& + P_{tm} + L_{tm,sf} + P_{sf} + L_{sf,bs} \\
& + P_{bs} + \left(\frac{\text{filesize}}{BW_{bs,cm}} \right) + P_{cm} \\
T_{push} = \quad & T_{pull} + L_{bs,cm} + P_{sf} + L_{sf,mds} \\
& + P_{mds} + L_{mds,df} + P_{sf} + L_{sf,cm} + P_{bs}
\end{aligned}
$$

Equation 4.3. Upload performance

Assuming again that L >> P we set P = 0 and simplify:

$$
\begin{aligned}
T_{pull} &= L_{cm,sf} + 2L_{sf,tm} + L_{sf,bs} + \left(\frac{\text{filesize}}{BW_{bs,cm}} \right) \\
T_{push} &= T_{pull} + L_{bs,sf} + 2L_{sf,mds} + L_{sf,cm}
\end{aligned}
$$

Equation 4.4. Upload performance in slow network

The performance will depend greatly on the file size and the network used. Using the sample network values and some sample file sizes a speed estimate can be made.

File size	LAN	WLAN	Cable	Internet
0	0.32 ms	3.2 ms	100 ms	300 ms
1 kb	0.40 ms	4 ms	102 ms	310 ms
10 kb	1.2 ms	11.3 ms	120 ms	397 ms
1 mb	90.3 ms	836 ms	2322 ms	10300 ms
1 gb	93 s	853 s	2275 s	10240 s

Table 4.2. Estimated upload times for pull file upload

For small files (< 1kb) the network latency is the dominant factor. For larger files (>10kb) it is the bandwidth between the client module and the byte store service that determines the speed of the upload.

This analysis uses a single source for the file. Work is currently done to investigate download from multiple sources, which should greatly improve performance.

Download files

When downloading there are two cases to consider. The first one is direct downloading to the client module. The second one involves caching the downloaded file locally. Here, the second use case can again be expressed in terms of the first use case.

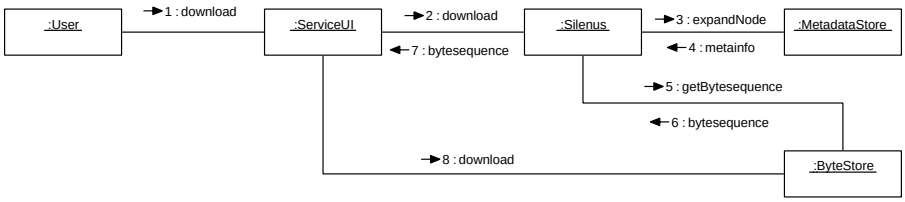


Figure 4.43. Download without caching use case

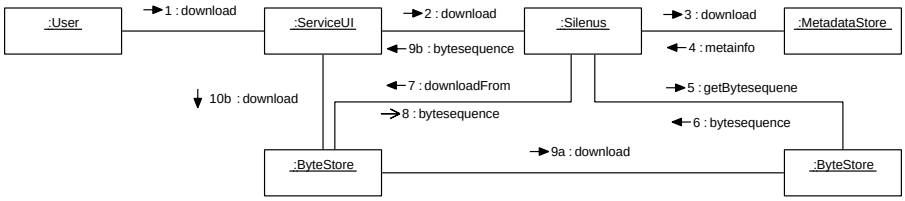


Figure 4.44. Download with caching use case

$$T_{nocache} = P_{cm} + L_{cm,sf} + P_{sf} + L_{sf,mds} + P_{mds} + L_{mds,sf} + P_{sf} + L_{sf,bs} + P_{bs} + L_{bs,sf} + P_{sf} + L_{sf,cm} + P_{cm} + \left(\frac{filesize}{BW_{cm,bs}} \right)$$

$$T_{cache} = T_{nocache} + L_{sf,lbs} + P_{lbs} + L_{lbs,sf} + \left(\frac{filesize}{BW_{cm,lsb}} \right)$$

Equation 4.5. Download performance

Assuming again that L >> P we set P = 0 and simplify:

$$T_{nocache} = 2L_{cm,sf} + 2L_{sf,mds} + 2L_{sf,bs} + \left(\frac{filesize}{BW_{cm,bs}} \right)$$

$$T_{cache} = T_{nocache} + 2L_{sf,lbs} + \left(\frac{filesize}{BW_{cm,lsb}} \right)$$

Equation 4.6. Download performance in slow network

For the example networks and some example file sizes this leads to the following results:

File size	LAN	WLAN	Cable	Internet
0	0.48 ms	4.8 ms	150 ms	450 ms
1 kb	0.56 ms	5.6 ms	152 ms	460 ms
10 kb	1.4 ms	12.9 ms	170 ms	547 ms
1 mb	90.5 ms	838 ms	2372 ms	10450 ms
1 gb	93 s	853 s	2275 s	10240 s

Table 4.3. Estimated download times without caching

Again, for small files the dominant factor is the latency. For large files the bandwidth is more decisive.

For downloading with caching it depends where the local byte store is located. If it is located on the same host as the client module the transfer will be fast, because it is local. But these numbers show that the local byte store does not have to be on the same host. If the original service is located on the Internet, a byte store service in the local network will not add a significant overhead.

Conceptual SILENUS Validation

There are two types of validation: Conceptual validity and operational validity. Some features, e.g. remote access and migration, are so inherit in the design that they can be validated if the given approach is followed. Other items, such as disconnected operation, have to be tested through experiments.

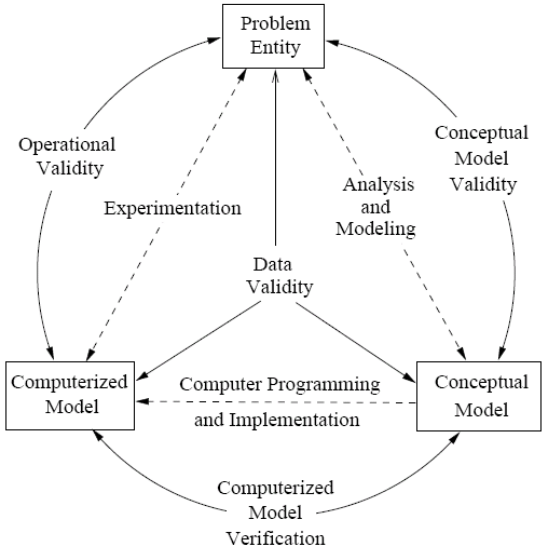


Figure 5.1. Sargent Circle

To verify the proposed architecture a conceptual model has to be designed. The conceptual model will describe a software system that solves the given problem. Conceptual model validity is defined as determining that the theories and assumptions underlying the conceptual model are correct and that the model representation of the problem entity is reasonable for the intended purpose of the model. Since the conceptual model is derived from the architecture it follows that it is valid if the architectural model is valid.

Class-level Design

To develop a conceptual system model an object-oriented approach is chosen. As such, the whole system is split into individual packages. Each package is then split into several classes.

Package Diagram

The most natural mapping is to create one package for every component in the SILENUS system. Figure 5.2, “Package overview for the SILENUS system” shows the top-level package diagram. The core package contains the interfaces. The util package contains common utilities. The silenus package contains the SILENUS facade and the human interface. The optimizer package contains optimizer services. The nfs package holds the implementation for the NFS adapter. The midas package contains the metadata store implementation. The compatibility package contains compatibility adapters. The byzantium package contains the byte store implementation.

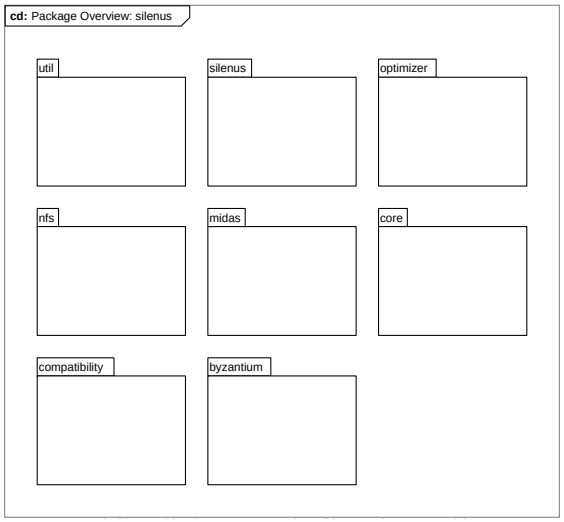


Figure 5.2. Package overview for the SILENUS system

Class Diagrams

Each package will consist of multiple classes. The class diagrams for all packages are shown here.

Figure 5.3, “SORCER interfaces in core package” shows the SORCER interfaces in the core package. These interface conform to the SORCER notion of an exertion, where an exertion defines a method name and a service context. The service context defines the data for the call.

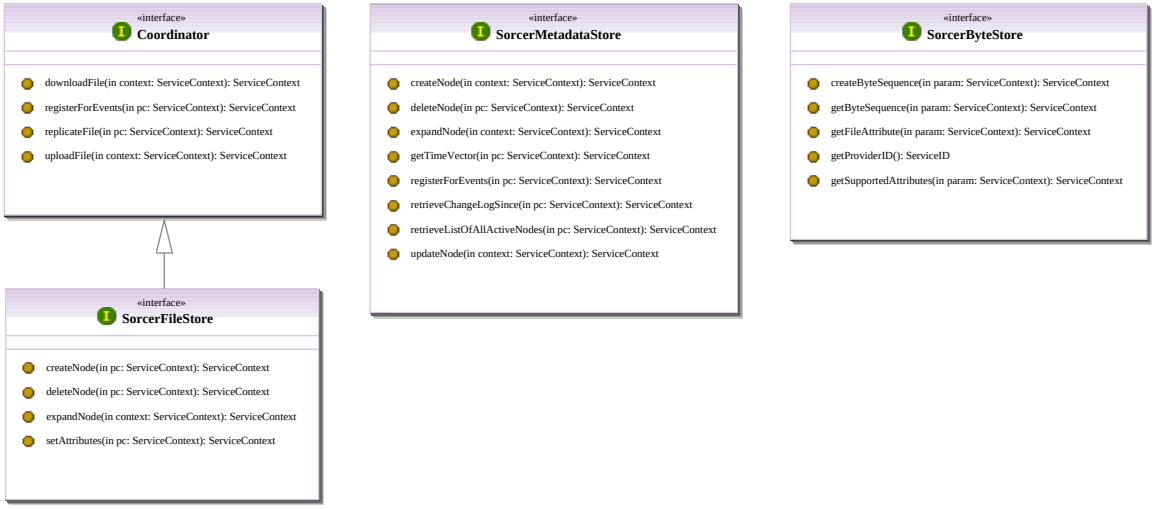


Figure 5.3. SORCER interfaces in core package

The SORCER interfaces are used when SILENUS is accessed through a service-oriented program. However, internally and externally am object oriented interface is provided. This interface follows the traditional object oriented approach. Figure 5.4, “Object-oriented interface to metadata store”, Figure 5.5, “Object-oriented interface to byte store”, and Figure 5.6, “Object-oriented interface to SILENUS facade” show the diagrams for the metadata store, the byte store, and the SILENUS facade.

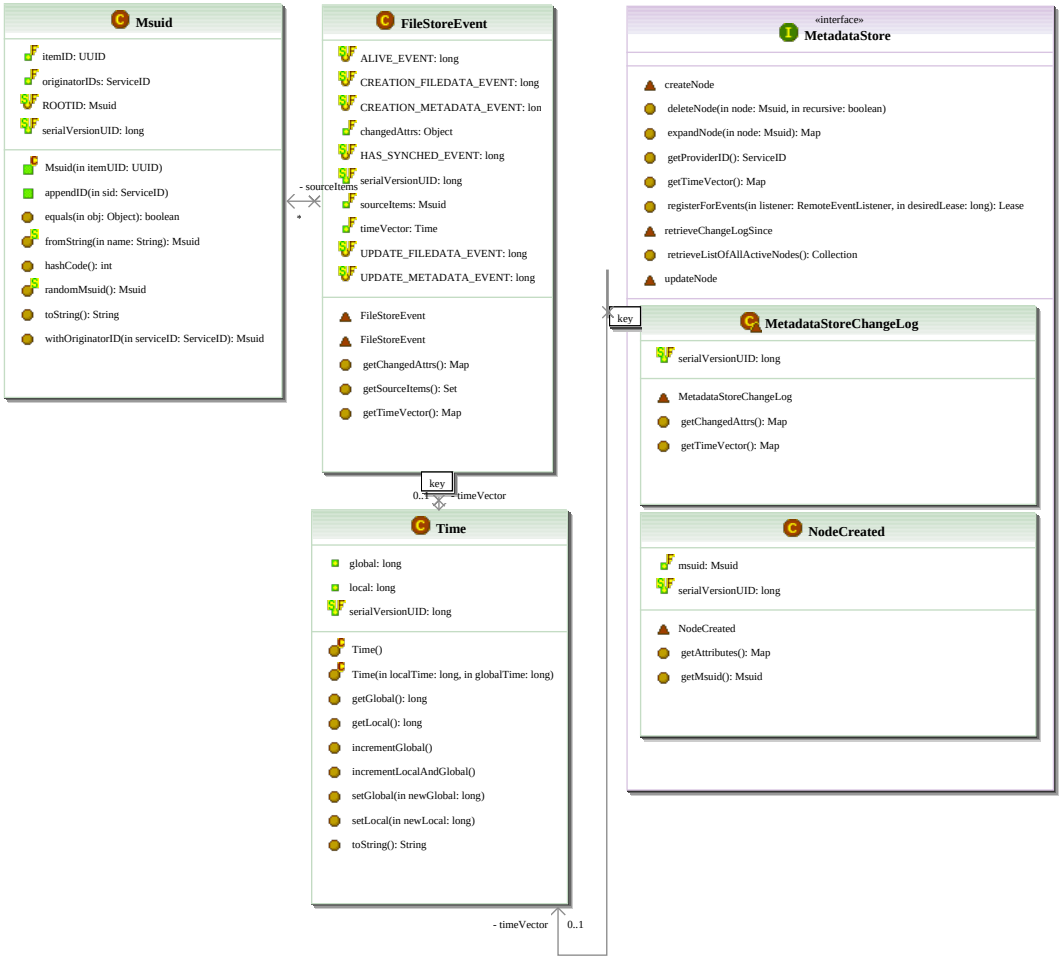


Figure 5.4. Object-oriented interface to metadata store

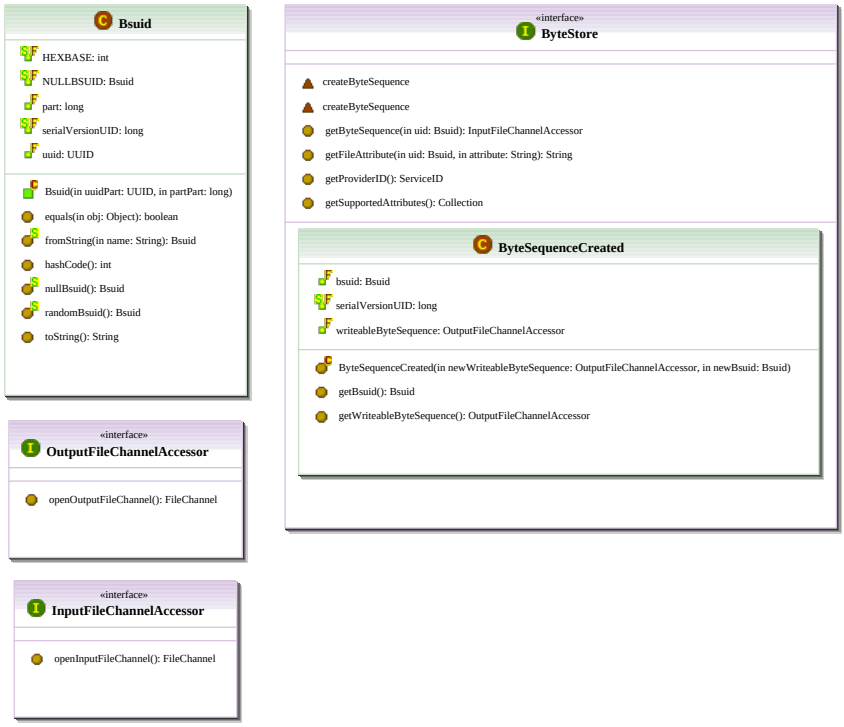


Figure 5.5. Object-oriented interface to byte store

Figure 5.6. Object-oriented interface to SILENUS facade

The detailed information on these interfaces can be found in Appendix A, *Reference*.

Technical Architecture

The actual implementation of these interfaces was done using the Java language and existing frameworks. We will describe the technical architecture and the deployment of the services during the validation.

The technical architecture describes which packages and frameworks have been used in developing the prototype. Figure 5.7, “SILENUS Technical Architecture” shows the technical architecture.

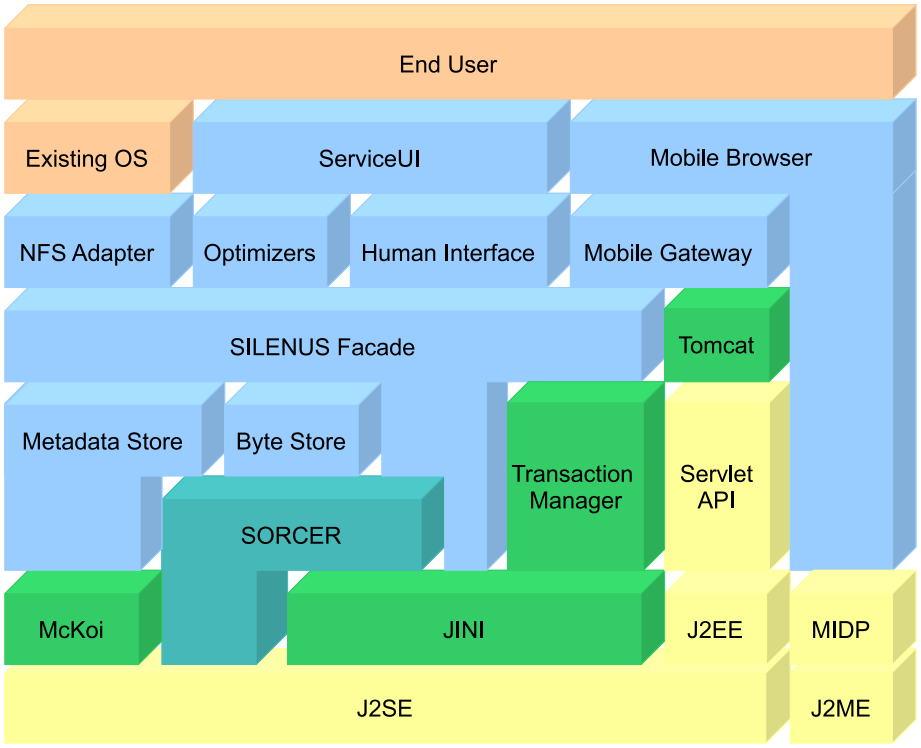


Figure 5.7. SILENUS Technical Architecture

Operational SILENUS Validation

To validate the proposed solution the design and its implementation have to be validated against the problem statement. The solution should provide all the requested core features. It should provide all the given architectural qualities. In addition, it should provide all the use cases for the given roles.

To validate through experiments a prototype was developed. The architectural model was then being tested against this prototype. The experimental setup is as following:

The implemented system and its components where deployed in the SORCER lab. The services where started on different machines using different architectures. The experiments that where conducted where: Validation of the use cases in a connected system, validation of the meta computer role using the SORCER proth application, and validation of disconnected use using a laptop. Each one of these experiments is now explained in more detail.

Deployment Diagram

To conduct the experiment the services where started on different hosts in the SORCER lab. Figure 5.8, “Deployment Diagram” shows the complete overview over the hosts involved in the experiment. Not all the services shown here where relevant for all experiments. The hosts Yew, Willow, Persimmon, Lime, and Hemp are servers in the lab. They where used to simulated a distributed server environment. The host Yucca simulated a desktop machine that was used to access the services. The host Cordelia is a laptop that was used to verify disconnected operation.

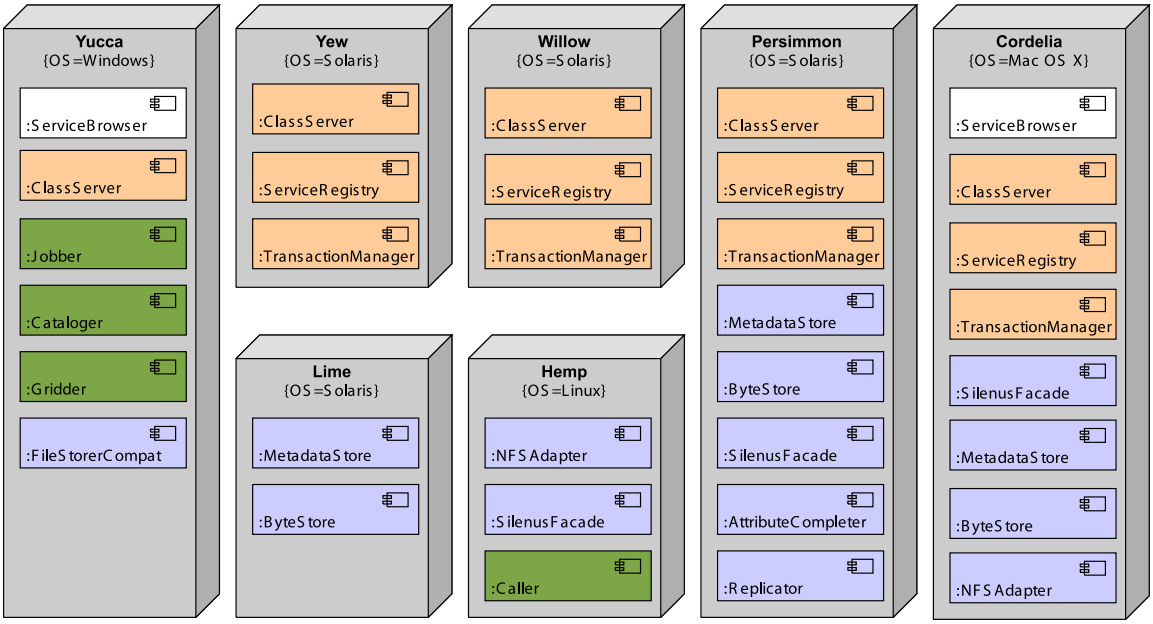


Figure 5.8. Deployment Diagram

Validation in a Connected System

After the services where deployed, the use cases where verified. The host Yucca was used to display the ServiceUI human interface user agent. The ServiceUI was used to verify the use cases. The host hemp provided and used the NFS adapter. This was used to verify the use cases using built in operating system support. Data integrity was verified using built-in functionality and using the actual saved data. The use cases where verified using the ServiceUI, the NFS adapter and the mobile client and gateway.

The use cases that where validated using the ServiceUI where: browse files, upload files, download files, modify file metadata, replicate files, provision service, and stop service manually.

Most of this functionality could be verified at the same time, as the use cases depend on each other. To download a file, it must be found first through file browsing. Once the file is found, it is displayed, for which it needs to be downloaded.

The browse files and the download file use cases proved to be successful. A service browser was invoked on the client host. The service browser picked up the SILENUS facade services running on the server hosts in the lab. Both facade services provided the ServiceUI user agent. This user agent showed the file directory structure

that was present in the SILENUS system. Figure 5.9, “Using the ServiceUI to browse files” shows a screenshot of the user agent displaying the file and directory structure. The selected file was downloaded to the client machine and displayed there.

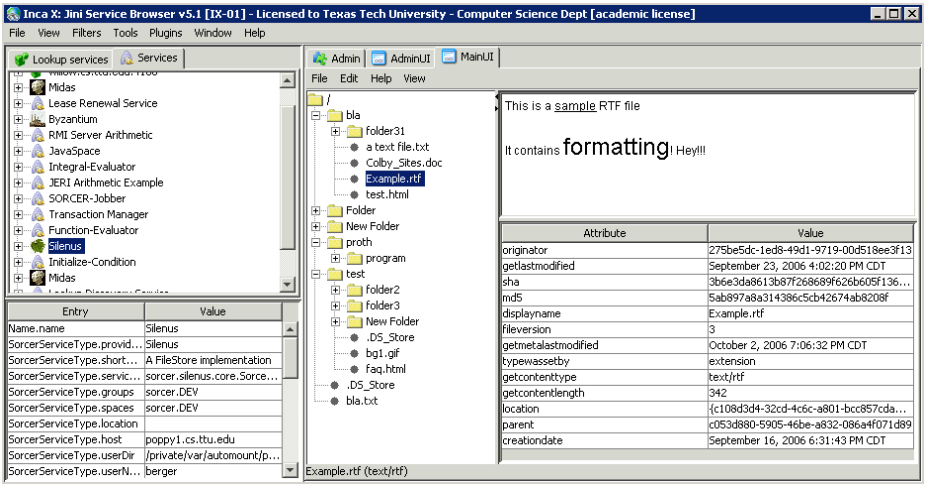


Figure 5.9. Using the ServiceUI to browse files

The same ServiceUI was also used to verify the other use cases. The files, which are displayed, have been uploaded prior using the same ServiceUI. A local file was selected, and then uploaded into the SILENUS system. The byte replicator service was running and immediately replicated the file as soon as it was uploaded. When the byte replicator was not running, the files where still uploaded to a random byte store.

Modifying file metadata was tested on several levels: The directory structure in the SILENUS system is pure metadata; so just creating a directory did already modify the metadata. When a file was uploaded, its metadata was filled with reasonable default values. When the attribute completer service was running, this metadata was completed with the checksum attributes for sha and md5. All attributes could be also be modified manually. This was tested by renaming files and modifying attributes such as file content type.

File replication was tested using the hoard functionality in the ServiceUI and the automated replication service. The hoard function checked if a byte store on the local machine is available, and if so, initiates replication to the local machine. This worked

well for the case where a local byte store exists, but showed to have too long timeout when no local byte store could be found. The byte replicator service worked very well: It was tested by terminating on of the byte stores on the network. The byte replicator then replicated the files to the third byte store available.

Service provisioning was tested using the RIO framework. The two optimizer services (byte replicator and attributes completer) where implemented as RIO service beans. As such, they could be successfully deployed on any host running a cybernode service. Provisioning the other services proved to be more difficult. The byte store and the metadata store need a local data directory. This must be available on the running machine. There was also a problem with RIO-assigned Service Ids and the way the SORCER framework handles Service Ids. These issues need further investigation, but the optimizer services served as a proof of concept.

Stopping a service manually was the easiest use case to test. There are two ways of stopping a service: Killing the actual service process, and terminating the service gracefully. When the service process is killed by force, it stays visible in the lookup service until its lease expires. This was sometimes confusing, as the service was still visible, but not responding. When a service was terminated gracefully through the destroy method, it properly deregistered.

Since the ServiceUI is the most specific client, it was used to verify most of the operations. The other interfaces had to provide less functionality due to protocol constraints in case of the NFS adapter and computational constraints in case of the mobile client.

The NFS adapter was used to validate the use cases: browse files, upload files, download files, and modify file metadata. The NFS adapter provides support for all operations present in the NFS version 2 protocol. As such, the directory was mounted on a UNIX machine. The directories could be browsed using the operating systems build in functionality. Files could be downloaded, uploaded, and edited directly using existing applications. Figure 5.10, “Standard UNIX ls application used for browsing” shows a screenshot of the standard UNIX ls application being used for browsing the files. Figure 5.11, “Standard UNIX cat application used for download” shows a screenshot of the cat application that is used to display a file in the system. The NFS adapter was also tested on a Mac OS X system where more graphical browsers and applications

where used. Mac OS X build in TextEdit seemed a very good candidate. Unfortunately, it creates and renamed several temporary files when saving, resulting in an administrative overhead and a significant performance impact.

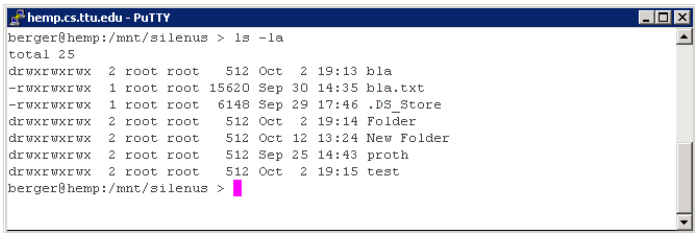


Figure 5.10. Standard UNIX ls application used for browsing

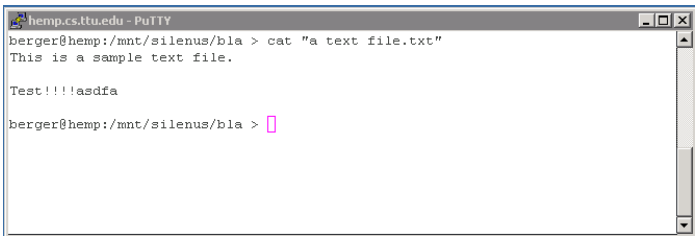


Figure 5.11. Standard UNIX cat application used for download

The mobile gateway and client where used to validate the use cases: browse files, and download files. The restraints here where imposed by the limited capability of a mobile phone device. The tests where done using a mobile phone emulator from the J2ME development toolkit. The current implementation provides read-only functionality, which is sufficient for testing purposes. Figure 5.12, “Mobile client used for browsing and displaying files from the file store” shows the mobile client being used to browse files in the file store. It also shows the mobile client being used to download and display files stored in the system.

Figure 5.12. Mobile client used for browsing and displaying files from the file store

These test have shown that the user role works fine, not matter which interface the user selects. The ServiceUI could provide the most functionality. Existing applications can be used through operating system adapters. The mobile client and gateway can provide the file system content anywhere.

Validation for the Metacomputer Role

To validate the metacomputer role the proth application was chosen. Proth is a grid application that searches for large prime numbers. It has been used in previous experiments to show other aspects of the SORCER framework. It uses the SORCER file store service to deploy an application across multiple hosts. It also uses the file store to transport input data to the service provider. After the calculation, the output data is written back into the file store.

The proth application was validated using a compatibility adapter for the existing SORCER file store service. The requested functionality was mapped from the old file store interface to the new SILENUS file store interface. No code changes where necessary to the existing proth application.

Proth is a calculation intensive grid application. The data transported through the file store is rather small. There where therefore no significant performance boosts or delays in using the SILENUS file store.

Validation for a Disconnected System

One of the main strength of the SILENUS model is that it expects and handles disconnection. To test disconnection, two approaches where chosen: Simulate disconnection by terminating services, and actual disconnection by unplugging a network cable.

When simulating disconnection by terminating services everything worked as expected: The services disappeared. The facade picked up another metadata store. When uploading a new file, the facade picked another byte store to store the data in. Changes could still be made to the system. When the original metadata store was started up again, it synchronized with the metadata store on the network and applied all changes. The byte replicator replicated all files that where available on only one byte store.

To simulate concurrent modification, two sets of services where run. First, both sets of services where run simultaneously. Some files where uploaded and directories created to test metadata propagation and replication. Then one set of services was shut down. The remaining services could still be used to browse and modify the file system. Two files where modified. Then this set of services was completely shut down and the other set of services was started. They still contained the old information, which could be browsed. Two files where modified: One was the same file that was modified before; the other file was another one that was not modified on the first set of services. Then the first set of services was brought up again. Both metadata stores immediately synchronized. The unrelated files just propagated their changes. The concurrently modified file was virtually duplicated, resulting in three files: Two with the actual file content, and a symbolic link to one of them.

Actual disconnection proved to be more difficult to validate. To test real disconnection, one set of services was run in the SORCER lab. Another set of services was run locally on a laptop. At first, the laptop was connected to the network. Browsing and downloading files worked just as expected. When the network cable was unplugged, two problems occurred: The network interface was shut down, and the services where still visible.

The first problem was in the network interface. Modern operating systems such as Windows XP or Mac OS X can detect if a network cable is plugged in or unplugged. They will automatically disable the network interface when a cable is unplugged. This will also delete all IP addresses and routes that where previously assigned to that interface. Since the services are registered using the external IP address, all services seemed to disappear. This problem could be solved by forcing the network interface to stay active. This is, however, not a very good solution. Unfortunately, the registration of the services is part of the Jini technology that SORCER is building upon. This issue will have to be investigated more in the future.

The second problem was that of services not disappearing, even though they are not connected. The reason for this lies in the way Jini manages disconnected resources: Using leases. Each service holds a lease on its registration. These leases must be renewed after a certain time to stay active. When a service is shut down properly, it deregisters itself from the registration service. When it is improper shutdown or disconnected, it will not disappear until the lease expires. The default timeout for registration leases was set to five minutes. It could therefore take up to five minutes for a service to disappear from the registration provider. For the validation, this issue was improved by reducing the lease timeout. In the future the SORCER framework will support a heartbeat mechanism. which will allow faster detection of disappearing services.

After restarting the network interface and waiting for the lease timeouts, the services disappeared as expected. The SILENUS system was still accessible on the laptop, as well as on the lab servers through a desktop system. Both worked as before. A file was modified concurrently to test the disconnected operation.

When plugging the network cable back it again took a while for the services to find each other. This is a similar problem that is inherited from the Jini framework. Jini registration services send out multicast packets every 30 seconds. Until one of

these packets is received, the services are unable to find each other on the network. It can therefore take a while until the metadata stores discover their reconnection. After they discovered each other, they exchanged the information and the file store was synchronized again. The concurrently modified file was virtually duplicated, as expected.

The system worked as expected during its disconnected operation. The detection of the disconnection and reconnection is very slow. However, when bringing a laptop back onto the network, it may be permissible to wait one minute before synchronization. The alternative is decreasing the lease time and decreasing the time between multicast announcements, flooding the network with more messages. In most cases, waiting for a short time to resynchronize is permissible, as long this happens very infrequently, such as once a day when the laptop is connected and disconnected.

Data Integrity

The data integrity was checked using two different methods. The first method was directly comparing the files in the byte store. The second one was using integrity checks implemented in the SILENUS administrative UI and in the byte store.

For the first integrity check, the files on the byte store where compared directly to their originals using the UNIX diff command. No files showed any difference, so the file transfer into the byte store worked without problems.

The other integrity check was build into the system. Each byte store has the capability to provide cryptographic checksums for all files stored in it. The supported algorithms are SHA and MD5. The expected values for these checksums are stored in the metadata store. When invoking the integrity check, each byte store is asked to compute the checksum, which is then compared. As expected, there where no differences in the expected and calculated checksum.

To test the checksum algorithms, a file was intentionally corrupted. When running the integrity check, the file reported different checksums and was detected as corrupted. This shows that the file integrity checking works as expected.

Validation of Architectural Qualities

The previous tests showed the use cases and disconnected operation. Another requirement was that the system should provide these architectural qualities: Network transparencies, confidentiality, global availability, disconnected operation, manageability, scalability, reliability, modifiability, and platform independence.

The network transparencies are inherited from the service-oriented design. It does not matter where the actual service is, it will always be available to a user on the current network. During tests, it showed that it may take time to discover the services, but the system always works as long as at least one of the services is available. It does not matter on which host.

Confidentiality was not tested. The security concept was designed, but not implemented. It could therefore not be tested. However, existing encryption algorithms have proven itself in the past.

Global availability was tested using the mobile browser, the ServiceUI, and a prototype of a WebDAV adapter. In all cases the file store and its contents showed to be available. The mobile browser was tested using an emulator. It connected to the system using a mobile gateway, which provided the actual files. The ServiceUI was run from different hosts, where only a service browser was installed and no component of the actual SILENUS system. The WebDAV adapter prototype was used to connect to the SILENUS system from a Windows and a Mac OS X host. In both cases the files where available for browsing, viewing, and modifying.

Disconnected operation was tested using simulated disconnection and actual network cable disconnection as explained in an earlier section.

Manageability was tested through testing the implemented optimizer services. When a service was terminated, the byte replicator picked up that there are not enough copies, and starts replicating files. This showed the concept of autonomic management services. It could further be improved by adding more optimizer services.

Scalability was not tested on the implementation; it is inherit in the design. In the design, services federate when a request is made. If the system is overloaded, new services of the same type can be added to provide more responsiveness. The only

services that need to communicate with multiple other services are the metadata store. The theoretical analysis suggests that the system scales well up to thousands of metadata store, but this is yet to be proven.

Reliability was tested through disconnecting and randomly terminating services. As long as there was still at least one of each service available, the system could not be brought down. It was always available to browse the files. When the right byte stores were terminated quickly enough the actual file content became unavailable. This could only be fixed by bringing at least one byte store with the file content back online. This will need to be improved with better optimizer services in the future.

Modifiability was constantly tested during development. Every time a modification was made, this update would have to be propagated to all hosts running the services. In all cases it was enough to just restart the services, and they did download the newest version of the code on startup.

Platform independence is provided by the choice of the Java platform. In the tests, the services where run on Windows, Solaris, Linux, and Mac OS X using the i386, sparc, and powerpc architectures. In all cases the services behaved exactly the same, on any tested architecture and platform combination.

This validates that all of the architectural qualities other than security that where requested are actually provided by the system.

Actual Performance

To measure the actual performance tests where conducted using the NFS adapter. These tests do not only measure the performance of SILENUS, but also the performance of the network device, the NFS adapter, and the NFS client application.

For local disk to disk a standard copy operation was used and timed. For Disk to SILENUS a file was uploaded into the SILENUS system. For SILENUS to disk a file was downloaded from SILENUS to the local hard disk.

This data was collected using the test layout that was shown in the deployment diagram earlier. The hosts are connected through a 1 GBit network. However, the hosts involved have a 100 MBit network interface.

What	0 kb	10 KB	1 MB	100 MB
Disk to disk	0.0 sec	0.0 sec	0.0 sec	0.7 sec
Disk to SILENUS	0.2 sec	1.6 sec	1.7 sec	22.8 sec
SILENUS to disk	0.0 sec	0.1 sec	0.2 sec	16.6 sec

Table 5.1. SILENUS performance over the NFS adapter

This shows that the performance of the SILENUS system is not so much dependent on the actual file size but rather on the number of requests. Creating an empty file is almost instant, but it still requires a metadata modification. Retrieving an empty file is instant, as there is no file content to retrieve. For small files, the time for creating the file is about 2 seconds, not really dependent on the file size. Retrieving a file is much faster: No transaction is needed and no modifications are done. For a large file, the actual network performance shows. The raw data given in the theoretical performance analysis suggested that a 100 MB file could be transferred in about 9.3 seconds. For file upload, the SILENUS system reaches 40% of the maximum network performance. For file download this increases to 56% of the maximal network performance. Given the overhead of locating the file, transferring it from a byte store to the NFS adapter, and through the NFS protocol to the local host these values are very satisfying.

This shows that the claim that the SILENUS model performance is just dependent on the network performance could not entirely be validated. For small files the time it takes to transfer the data over the network does not outweighs the management overhead. This is especially true in the creation of files, as several services are involved. However, once the file gets larger the management overhead diminished and the performance gets closer to the actual network performance. When the link is slower, such as over the Internet, there should be no performance impact.

CHAPTER 6. CONCLUSION

The questions that were asked in the introduction were: Can a dynamic approach, such as service-orientation, provide the reliability and stability required for a file system? And if so, how can this be done? These questions can be answered with: yes, it is possible to provide a reliable file system. This can be done using the SILENUS model introduced in this dissertation.

For this dissertation, a new model for a metacomputing file system has been introduced. The SILENUS model splits up the file storage into separate services for file content, metadata, and management. These services are not connected statically, but rather federate dynamically to provide a file system service.

Also, a new methodology for using this metacomputing file system was introduced. Users can access their files through a zero-install ServiceUI, through a mobile client, and with existing applications through the use of adapters. The facade services provide entry points and coordination services for the system. Services can be autonomically provisioned using the RIO framework. Optimizer services can take over management tasks and can be tailored to specific user needs.

A new algorithm for metadata store synchronization based on a dual-clock time vector system was devised. This algorithm is generic and can be used to synchronize any key-value based data in a distributed and disconnected system. It provides a reliable method of ensuring data consistency.

The data storage service byte store and metadata store have been designed and implemented. These services provide support for the storage of the data in the SILENUS system.

The management services facade, byte replicator, and attributes completer have been designed and implemented. These services provide management functionality and access point as specified by the model.

User agents have been invented and developed for desktop and mobile users. Adapters have been developed for existing operating systems. This gives the user a broad range of methods to access their stored files.

An initial security model for a distributed file storage system has been specified. However, this model has to be further developed, because full security is beyond the scope of this dissertation.

This dissertation could only provide a model and greater architecture. There are several aspects that can be improved. Some have already been outlined throughout the dissertation and others are described here. Most of these topics are already being actively investigated by other students.

The byte store to byte store and byte store to user transfer can be greatly improved. Adam Turner is currently working on distributing files in chunks across multiple byte stores in federation based on the bittorrent model. This would split up large files in smaller parts, allowing these parts to be downloaded from different locations. It would use multiple channels, thus increasing overall performance.

More confidentiality checks can be enforced. This dissertation could barely scratch the security topic. There are several more decisions to be made: How can permissions be set, modified, and revoked? How can these be enforced? How can a policy for a distributed file system be managed? Daniela Inclezan is currently investigating security in federated systems.

Adam Thomas-Murphy is looking into virtual files and directories. What happens if only a small part of a file is modified? Would the whole file be re-created? Or can small changes be made? What if a part in the middle of a file changes size?

The location of the files can also be optimized. Files that are available in the local network may not need to be replicated. It may make sense to keep at least one copy of each file at two different physical locations to provide resistance against catastrophes. Chris Hard is researching ways to optimize the locations of actual file storage.

The current implementation provides an NFS adapter for UNIX systems. In the original design, a WebDAV adapter for UNIX and Windows systems was suggested. This adapter is currently being completed by Fajin Wang.

The design also suggested a JXTA adapter for connection to the JXTA content management service (CMS). This would provide support for files over an existing wide-area peer-to-peer network. Some questions would have to be answered such as: How can SILENUS be mapped to JXTA advertisements? How would security be managed in such a widely distributed system?

The optimizer services designed and implemented here barely scratch the surface of what optimizers could do. One possible optimizer service would try to derive attributes from file content and add the information to the metadata store. It could read images, music files, or word documents. The additional information can then be used to quickly find matching files stored in the system.

Fully automated provisioning would be another exiting topic. Why does a service need to be started manually when it can be done automatically? Services can provide their current service state and utilization. Services can automatically be shut down when underused or automatically be started when overused. This is especially true for optimizer services, and the SILENUS facade, but can also be applied to metadata stores and byte stores.

One of the core features was easy installation. The existing system provides zero configuration: Services can discover themselves automatically. The configuration necessary on each machine is reduced to creating a network name. However, there should be a user-friendly way to install, start, and stop the services. Services should be started automatically when the machine boots up, on all major operating systems.

The system described here accumulates data: Metadata stores keep a lot of old information to re-create changelogs. Byte stores keep old files for undeletion. At some point, this old data has to be cleaned. Sophisticated algorithms have to be developed that can carry out this task.

Even with all these future research topics the framework model has proven itself. It can be used for file storage in a changing network environment. The future work can add value to the existing system.

My vision for the future of the SILENUS system would be: I create a document at work, and modify it, and at five I leave the office and drive home. The system automatically detects my behavior guessing that I want to continue working, and at that time automatically replicates the latest copy to my home machine. When I arrive the file is available locally, so that I can work and create new versions. If my Internet connection goes down I won't notice, and when it comes back up again the files are automatically replicated, giving me reliability and dependability.

BIBLIOGRAPHY

NORMATIVE DOCUMENTS

[1] Sun Microsystems. *RFC 1094*. “NFS: Network File System Protocol specification”. IETF. 1989. http://www.ietf.org/rfc/rfc1094.txt.

[2] B. Callaghan, B. Pawlowski, and P. Staubach. *RFC 1813*. “NFS Version 3 Protocol Specification”. IETF. Jun 1995. http://www.ietf.org/rfc/rfc1813.txt.

[3] Y. Goland, E. Whitehead, A. Faizi, S. Carter, and D. Jensen. *RFC 2518*. “HTTP Extensions for Distributed Authoring – WEBDAV”. IETF. 1999. http://www.ietf.org/rfc/rfc2518.txt.

[4] G. Clemm, J. Amsden, T. Ellison, C. Kaler, and J. Whitehead. *RFC 3253*. “Versioning Extensions to WebDAV (Web Distributed Authoring and Versioning)”. IETF. 2002. http://www.ietf.org/rfc/rfc3253.txt.

[5] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. *RFC 3530*. “Network File System (NFS) version 4 Protocol”. IETF. 2003. http://www.ietf.org/rfc/rfc3530.txt.

[6] G. Clemm, J. Reschke, E. Sedlar, and J. Whitehead. *RFC 3744*. “Web Distributed Authoring and Versioning (WebDAV) Access Control Protocol”. IETF. 2004. http://www.ietf.org/rfc/rfc3744.txt.

[7] “Data Encryption Standard (DES)”. *FIPS PUB*. 46. U.S. Department of commerce. National Institute of Standards and Technology. Jan 1977.

[8] “Announcing the Advanced Encryption Standard (AES)”. *FIPS PUB*. 197. National Institute of Standards and Technology. Nov 2001.

[9] *Java Cryptography Extension (JCE) for the Java 2 SDK, v 1.4*. http://java.sun.com/products/jce.

[10] Tim Lindholm and Frank Yellin. *The Java™ Virtual Machine Specification (2nd Edition)*. Apr 99. Addison-Wesley Professional. 0201432943.

[11] Bill Venners. *The ServiceUI API Specification, v. 1.1a*. Jun 2005. http://www.artima.com/jini/serviceui/Spec.html.

[12] *Open Distributed Processing*. Reference Model. 10746. ISO/IEC. 1995.

ARTICLES

[13] Paul Leach and Dan Perry. “CIFS: A Common Internet File System”. *Microsoft Interactive Developer magazine*. Nov 1996. http://www.microsoft.com/mind/1196/cifs.asp.

[14] Richard Sharpe. *Just what is SMB?*. Oct 2002. http://samba.org/cifs/docs/what-is-smb.html.

[15] M. Satyanarayanan. “Coda: a highly available file system for a distributed workstation environment”. *Workstation operating systems: proceedings of the Second Workshop on Workstation Operating Systems (WWOS-II), September 27--29, 1989, Pacific Grove, CA*. 114–116. http://ieeexplore.ieee.org/iel5/267/3322/00109279.pdf. IEEE Computer Society Press. 1989. 0-8186-2003-X. 0-8186-5003-6 (microfiche).

[16] M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. “Coda: A Highly Available File System for a Distributed Workstation Environment”. *IEEE Transactions on Computers*. 39 (4). 447-459. 1990.

[17] Ann L. Chervenak, Bill Ahcock, Carl Kesselman, Darcy Quesnel, Ian Foster, Joe Bester, John Bresnahan, Sam Meder, Steven Tuecke, and Veronika Nefedova. “Data Management and Transfer in High-Performance Computational Grid Environments”. *Parallel Computing Journal*. 28 (5). May 2002. 749-771. http://www.globus.org/research/papers/dataMgmt.pdf.

[18] Asad Samar, Bill Allcock, Brian Tierney and Heinz Stockinger, Ian Foster, and Koen Holtman. “File and Object Replication in Data Grids”. *Journal of Cluster Computing*. 5(3). 305-314. Sep 2002. http://www.globus.org/research/papers/FileRepCluster02.pdf.

[19] Gurmeet Singh, Shishir Bharathi, Ann Chervenak, Ewa Deelman, Carl Kesselman, Mary Manohar and Sonal Patil, and Laura Pearlman. “A Metadata Catalog Service for Data Intensive Applications”. *SC2003: Igniting Innovation. Phoenix, AZ, November 15--21, 2003*. ACM Press and IEEE Computer Society Press. 2003. 1-58113-695-1. http://www.globus.org/alliance/publications/papers/mcs_sc2003.pdf.

[20] Anand Natrajan, Marty A. Humphrey, and Andrew S. Grimshaw. “Grids: Harnessing Geographically-Separated Resources in a Multi- Organisational Context”. *High Performance Computing Systems*. Jun 2001.

[21] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. *Peer-to-Peer Computing*. *Internal HP report*. Mar 2002.

[22] Anand Natrajan, Anh Nguyen-Tuong, Marty A. Humphrey, and Andrew S. Grimshaw. *The Legion Grid Portal*. *Grid Computing Environments 2001, Concurrency and Computation: Practice and Experience*. 2001.

[23] Markus Lorch. *Symphony - A Java-based Composition and Manipulation Framework for Computational Grids*. Thesis Document, University of Applied Sciences in Albstadt-Sigmaringen. Jul 2002.

[24] Kandle Kulish, Jerry Perez, and Phil Smith. *Multivariate Minimization Using Grid Computing*. *Workshop on Grid Applications and Programming Tools*. Jun 2003. *Seattle, WA, USA.*

[25] Peter J. Braam. “File Systems for Clusters from a Protocol Perspective”. *Second Extreme Linux Topics Workshop*. Jun 1999. *Monterey CA.*

[26] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. “The Google File System”. *19th ACM Symposium on Operating Systems Principles*. 2003.

[27] R. Rivest, A. Shamir, and L. Adleman. “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”. *Communications of the ACM*. 21 (2). 120 - 126. 1978.

[28] Sun Microsystems. *Build a Compute Grid with Jini™ Technology*. Jini™ Technology White Paper. Dec 2004. http://www.jini.org/whitepapers/JINI_ComputeGrid_WP_FINAL.pdf.

[29] Carlos Queiroz, Bruno Souza, and Einar Saukas. *Beyond Web Services*. Combining Jini™ Network Technology and “Project JXTA” to Take Advantage of Edge Computing. *JavaOne, Sun's 2003 Worldwide Java Developer Conference*.

[30] Michael Sobolewski. “Federated P2P Services in CE Environments”. *Advances in Concurrent Engineering*. 13–22. A.A. Balkema Publishers. 2002. 90-5809-502-9.

[31] Michael Sobolewski. “FIPER: The Federated S2S Environment”. *JavaOne, Sun's 2002 Worldwide Java Developer Conference*. 2002. http://servlet.java.sun.com/javaone/sf2002/conf/sessions/display-2420.en.jsp.

[32] R. Kolonay and Michael Sobolewski. “Grid Interactive Service-oriented Programming Environment”. 97–102. *Concurrent Engineering: The Worldwide Engineering Grid*. Tsinghua Press and Springer Verlag. 2004. 7-302-08802-0.

[33] Sekhar Soorianarayanan and Michael Sobolewski. “Monitoring Federated Services in CE”. *Concurrent Engineering: The Worldwide Engineering Grid*. 89–95. Tsinghua Press and Springer Verlag. 2004. 7-302-08802-0.

[34] Douglas Thain, Todd Tannenbaum, and Miron Livny. “Condor and the Grid”. *Grid Computing: Making The Global Infrastructure a Reality*. John Wiley. Fran Berman. Anthony J.G. Hey. Geoffrey Fox. 2003. 0-470-85319-0.

[35] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. “Grid Services for Distributed System Integration”. *Computer*. 35. 6. 37–46. Jun 2002. 0018-9162. http://csdl.computer.org/dl/mags/co/2002/06/r6037.pdf.

[36] Vivek Khurana, Max Berger, and Michael Sobolewski. “A Federated Grid Environment with Replication Services”. *Next Generation Concurrent Engineering*. Omnipress. 2005. 0-9768246-0-4.

[37] Michael Sobolewski, Sekhar Soorianarayanan, and Ravi-Kiran Malladi Venkata. “Service-Oriented File Sharing”. *CIIT conference (communications,internet and information technology)*. 633–639. Nov 2003.

[38] Robert Lupton, F. Miller Maley, and Neal Young. “Data Collection for the Sloan Digital Sky Survey–-A Network-Flow Heuristic”. *Journal of Algorithms*. 27. 2. 339–356. May 1998.

[39] Eva Arderiu Ribera. “LHC Distributed Data Management”. *CHEP 98, Chicago*. Nov 1998. http://wwwinfo.cern.ch/asd/rd45/papers/proc_108.ps.

[40] Max Berger and Michael Sobolewski. “SILENUS - A federated service-oriented approach to distributed file systems”. *Next Generation Concurrent Engineering*. Omnipress. 2005. 0-9768246-0-4.

[41] Danny Dolev, Joe Halpern, and H. Raymond Strong. “On the possibility and impossibility of achieving clock synchronization”. *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*. 1984. 504-511. ACM Press. 0-89791-133-4.

[42] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. *Communications of the ACM*. 21. 7. 558-565. Jul 78.

[43] Friedemann Mattern. “Virtual time and global clocks in distributed systems”. *Workshop on Parallel and Distributed Algorithms*. 215-226. 1989.

[44] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. “Deciding when to forget in the Elephant file system”. *Symposium on Operating Systems Principles*. 110-123. 1999. http://www.stanford.edu/class/cs240/readings/p110-santry.pdf.

[45] Zachary N. J. Peterson, Randal Burns, Joe Herring, Adam Stubblefield, and Aviel D. Rubin. “Secure Deletion for a Versioning File System”. *Proceedings of File and Storage Technology (FAST)*. USENIX. 2005.

[46] J. G. Steiner, B. Clifford Neuman, and J. I. Schiller. “Kerberos: An Authentication Service for Open Network Systems”. *Proceedings of the Winter 1988 Usenix Conference*. 191–201. Feb 1988.

ONLINE RESOURCES

[47] *Gnutella Protocol Development*. http://www.the-gdf.org/.

[48] *OpenAFS*. http://www.openafs.org.

[49] *Globus Alliance*. http://www.globus.org.

[50] *Sybase Avaki EII*. http://www.sybase.com/products/developmentintegration/avakieii/distributedarchitecture.

[51] *Lustre*. http://www.lustre.org.

[52] *Libgcrypt*. http://www.gnupg.org.

[53] Sung Kim. *WEB-DAV Linux File System(davfs2)*. http://dav.sourceforge.net/.

[54] *Knuth reward check*. http://en.wikipedia.org/wiki/Knuth_reward_check.

[55] Sun Microsystems. *Java Technology*. http://java.sun.com/.

[56] IBM. *Java technology*. http://www-128.ibm.com/developerworks/java.

[57] Apple Computer. *Java for Mac OS X*. http://www.apple.com/macosx/features/java/.

[58] *Kaffe.org*. http://www.kaffe.org/.

[59] *Java Technology*. http://www.java.com.

[60] *Java 2 Platform, Micro Edition (J2ME)*. http://java.sun.com/j2me.

[61] *JavaServer Pages Technology*. http://java.sun.com/products/jsp/.

[62] *Java Servlet Technology*. http://java.sun.com/products/servlet/.

[63] Jim Driscoll. *Jim Driscoll's Blog*. Servlet History. http://weblogs.java.net/blog/driscoll/archive/2005/12/servlet_history_1.html.

[64] Phil Bishop. *IncaX*. http://www.incax.com.

[65] *JXTA*. http://www.jxta.org/.

[66] Peter Deutsch. *The Eight Fallacies of Distributed Computing*. http://today.java.net/jag/Fallacies.html.

[67] Sun Microsystems, Inc.. *System Administration Guide: Security Services*. Using UNIX Permissions to Protect Files. http://docs.sun.com/app/docs/doc/816-4557/6maosrje8?q=ACL&a=view.

[68] Keith Lea. *The Java is Faster than C++ and C++ Sucks Unbiased Benchmark*. http://kano.net/javabench/.

[69] *Sloan Digital Sky Survey*. http://www.sdss.org/.

[70] Peter H. Dana. *Global Positioning System Overview*. http://www.colorado.edu/geography/gcraft/notes/gps/gps_f.html.

BOOKS

[71] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall. Jan 2002. 0130888931.

[72] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition*. Wiley. Oct 1995. 0471117099.

[73] Charlie Kaufman, Padia Perlman, and Mike Spencer. *Network Security: PRIVATE Commincation in a PUBLIC World*. Prentice Hall. 2002. 0-13-046019-2.

[74] Jan Newmarch. *A Programmer's Guide to Jini Technology*. Apress. Nov 2000. 1893115801.

[75] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison Wesley. May 2005. 0321263545.

[76] Will Willis, David Watts, and Tillman Strahan. *Windows 2000 System Administration Handbook*. Addison-Wesley Professional. Dec 2000. 0130270105.

Package sorcer.silenus.core

This package defines the core interfaces that are needed to use the SILENUS file system.

Class Bsuid

Class to handle UUIDs for objects stored in a byte store.

Synopsis

```
package sorcer.silenus.core;

public class Bsuid implements Serializable {

  // Public Static Methods
  public static Bsuid fromString(java.lang.String name);
  public static Bsuid nullBsuid();
  public static Bsuid randomBsuid();

  // Public Methods
  public boolean equals(java.lang.Object obj);
  public int hashCode();
  public String toString();
}
```

Methods inherited from java.lang.Object: `clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`

Version

$Revision: 1.2 $ $Date: 2006/09/02 19:26:36 $

Since

Nov 22, 2005

Inheritance Path. java.lang.Object-> sorcer.silenus.core.Bsuid

equals(Object)

Synopsis: public boolean **equals**(java.lang.Object *obj*);

fromString(String)

Synopsis: public static Bsuid **fromString**(java.lang.String *name*);

Parameters

name

the string to parse

return

a Bsuid object, if possible

Tries to create a Bsuid from a given String.

hashCode()

Synopsis: public int **hashCode**();

nullBsuid()

Synopsis: public static Bsuid **nullBsuid**();

Parameters

return

the null Bsuid.

Returns the Null Bsuid. The Null Bsuid represents no object.

randomBsuid()

Synopsis: public static Bsuid **randomBsuid**();

Parameters

return

a valid Bsuid.

Creates a random Bsuid.

toString()

```
Synopsis: public String toString();
```

Parameters

return

a String representation

Creates a String representation for this Bsuid.

This representation can be parsed with `fromString(java.lang.String)`

Interface ByteStore

Java interface to a ByteStore.

Synopsis

```
package sorcer.silenus.core;

public interface ByteStore {

  // Public Methods
  public ByteStore.ByteSequenceCreated createByteSequence(sorcer.silenus.co\
re.Bsuid wantedUID,

                                                           net.jini.core.tra\
nsaction.server.ServerTransaction transaction,
                                                           java.util.Map exp\
ectedMetadata)
      throws java.rmi.RemoteException;
```

```
  public ByteStore.ByteSequenceCreated createByteSequence(sorcer.silenus.co\
re.Bsuid wantedUID,

                                                           net.jini.core.tra\
nsaction.server.ServerTransaction transaction,
                                                           java.util.Map exp\
ectedMetadata,
                                                           sorcer.silenus.co\
re.InputFileChannelAccessor fileData)
      throws java.rmi.RemoteException;
  public InputFileChannelAccessor getByteSequence(sorcer.silenus.core.Bsuid\
 uid)

      throws java.io.IOException;
  public String getFileAttribute(sorcer.silenus.core.Bsuid uid,
                                 java.lang.String attribute)
      throws java.io.IOException;
  public ServiceID getProviderID() throws java.rmi.RemoteException;
  public Collection getSupportedAttributes() throws
    java.rmi.RemoteException;
}
```

Version

$Revision: 1.2 $ $Date: 2006/09/02 19:26:36 $

Since

Nov 15, 2005

See Also

`sorcer.silenus.core.SorcerByteStore`

Inheritance Path. sorcer.silenus.core.ByteStore

createByteSequence(Bsuid, ServerTransaction, Map)

```
Synopsis: public ByteStore.ByteSequenceCreated createByteSequence(sorcer.si\
lenus.core.Bsuid wantedUID,

                                                                 net.jini.\
core.transaction.server.ServerTransaction transaction,
                                                                 java.util\
```

```
.Map expectedMetadata)
              throws java.rmi.RemoteException;
```

Parameters

`wantedUID`

hold the requested Bsuid. May be null. If it is given, the bs will try to create an object with this uuid, but may always refuse and create a different one.

`transaction`

the transaction this upload is under. May be null. If used, the BS will report if the upload succeds. If the transaction fails the newly created file will be deleted.

`expectedMetadata`

provides some expected properties for the byte sequence. May be null. If the properties of the uploaded file do not match the ones given here then the byte sequence will be rejected. Please check `sorcer.silenus.core.FileStoreConstants` for possible values. The bytestore will very likely support things like ATTR_SIZE and ATTR_SHA1.

return

there parameters for this byte sequence.

Exceptions

`RemoteException`

If a remote IO error occurs.

See Also

`sorcer.silenus.core.FileStoreConstants`,
`sorcer.silenus.core.ByteStore.ByteSequenceCreated`

Creates a new Byte Sequence on the byte Store.

createByteSequence(Bsuid, ServerTransaction, Map, InputFileChannelAccessor)

```
Synopsis: public ByteStore.ByteSequenceCreated createByteSequence(sorcer.si\
lenus.core.Bsuid wantedUID,
```

```
                                                                    net.jini.\
core.transaction.server.ServerTransaction transaction,
                                                                    java.util\
.Map expectedMetadata,
                                                                    sorcer.si\
lenus.core.InputFileChannelAccessor fileData)
              throws java.rmi.RemoteException;
```

Parameters

`wantedUID`

a requested uid. May be null.

`transaction`

a transaction object. May be null.

`expectedMetadata`

expected metadata. If given, the byte sequence must match this data or it will be rejected.

`fileData`

a readable byte sequence to initialize this byte sequence to. May be null.

return

The Bsuid of the new byte sequence. The writeableByteSequence is filled in if the fileData was null.

Exceptions

`RemoteException`

If a remote IO error occurs.

See Also

`createByteSequence(sorcer.silenus.core.Bsuid, net.jini.core.transaction.server.ServerTransaction, java.util.Map)`

creates a bytesequence and fills it with the data given.

getByteSequence(Bsuid)

```
Synopsis: public InputFileChannelAccessor getByteSequence(sorcer.silenus.co\
```

```
re.Bsuid uid)

              throws java.io.IOException;
```

Parameters

`uid`

the requested byte sequence.

return

a reable byte sequence that can be used to access this file.

Exceptions

`IOException`

if any IO errors occur.

Retrieve an accessor to the byte sequence matching the given Bsuid.

getFileAttribute(Bsuid, String)

```
Synopsis: public String getFileAttribute(sorcer.silenus.core.Bsuid uid,
                                         java.lang.String attribute)
              throws java.io.IOException;
```

Parameters

`uid`

the Bsuid of the byte sequence.

`attribute`

the requested attribute.

return

value for this attribute or null.

Exceptions

`IOException`

if any IO errors occur.

Retrieves an intrisic attribute for a stored byte sequence.

getProviderID()

```
Synopsis: public ServiceID getProviderID() throws
            java.rmi.RemoteException;
```

Parameters

return

the ServiceID of this provider

Exceptions

`RemoteException`

If a remote IO error occurs.

Standard method to retrieve the UID of the provider.

getSupportedAttributes()

```
Synopsis: public Collection getSupportedAttributes() throws
            java.rmi.RemoteException;
```

Parameters

return

a Collection of attribute names.

Exceptions

`RemoteException`

If a remote IO error occurs.

Retrieves a list of intrinsic attributes suported on this byte store.

Class ByteStore.ByteSequenceCreated

Data class for created byteSequences.

Synopsis

```
package sorcer.silenus.core.ByteStore;

public static class ByteStore.ByteSequenceCreated implements Serializable {

  // Public Constructors
  public ByteStore.ByteSequenceCreated(sorcer.silenus.core.OutputFileChanne\
lAccessor newWriteableByteSequence,

                                       sorcer.silenus.core.Bsuid newBsuid);

  // Public Methods
  public Bsuid getBsuid();
  public OutputFileChannelAccessor getWriteableByteSequence();
}
```

Methods inherited from java.lang.Object: `clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`

Inheritance Path. java.lang.Object-> sorcer.silenus.core.ByteStore.ByteSequenceCreated

ByteStore.ByteSequenceCreated(OutputFileChannelAccessor, Bsuid)

```
Synopsis: public ByteStore.ByteSequenceCreated(sorcer.silenus.core.OutputFi\
leChannelAccessor newWriteableByteSequence,

                                               sorcer.silenus.core.Bsuid ne\
wBsuid);
```

Parameters

`newWriteableByteSequence`

the writeable byte sequence.

`newBsuid`

Bsuid of that sequence.

Creates a new ByteSequenceCreated object.

getBsuid()

```
Synopsis: public Bsuid getBsuid();
```

Parameters

return

Returns the bsuid.

Accessor method for property bsuid.

getWriteableByteSequence()

```
Synopsis: public OutputFileChannelAccessor getWriteableByteSequence();
```

Parameters

return

Returns the writeableByteSequence.

Acessor method for property writeableByteSequence.

Interface Coordinator

Java interface to the coordinator part of the SILENUS facade.

These operations are actually to be executed on the facade service itself rather than on the client

Synopsis

```
package sorcer.silenus.core;

public interface Coordinator implements Remote {

  // Public Methods
  public ServiceContext downloadFile(sorcer.base.ServiceContext context)
      throws java.rmi.RemoteException,
```

```
            ServiceUnavailableException;
  public ServiceContext registerForEvents(sorcer.base.ServiceContext pc)
      throws java.rmi.RemoteException,
             net.jini.core.lease.LeaseDeniedException;
  public ServiceContext replicateFile(sorcer.base.ServiceContext pc)
      throws java.rmi.RemoteException;
  public ServiceContext uploadFile(sorcer.base.ServiceContext context)
      throws java.rmi.RemoteException,
             ServiceUnavailableException;
}
```

Inheritance Path. sorcer.silenus.core.Coordinator

downloadFile(ServiceContext)

```
Synopsis: public ServiceContext downloadFile(sorcer.base.ServiceContext con\
text)

              throws java.rmi.RemoteException,
                     ServiceUnavailableException;
```

Parameters

`context`

the parameters as a ServiceContext

return

the result as a ServiceContext

Exceptions

`RemoteException`

if a remote io error occurs.

`ServiceUnavailableException`

if not all required services are available.

See Also

`downloadFile(sorcer.silenus.core.Msuid)`

download a file.

registerForEvents(ServiceContext)

```
Synopsis: public ServiceContext registerForEvents(sorcer.base.ServiceContex\
t pc)
              throws java.rmi.RemoteException,
                     net.jini.core.lease.LeaseDeniedException;
```

Parameters

`pc`

the parameters as a ServiceContext

return

the result as a ServiceContext

Exceptions

`RemoteException`

if a remote io error occurs.

`LeaseDeniedException`

if the lease requested cannot be granted.

See Also

`registerForEvents(net.jini.core.event.RemoteEventListener, long)`

register for file store events.

replicateFile(ServiceContext)

```
Synopsis: public ServiceContext replicateFile(sorcer.base.ServiceContext pc\
)
              throws java.rmi.RemoteException;
```

Parameters

`pc`

the parameters as a ServiceContext

return

the result as a ServiceContext

Exceptions

`RemoteException`

if a remote io error occurs.

See Also

`replicateFile(sorcer.silenus.core.Msuid, net.jini.core.lookup.ServiceID)`

Initiate file replication.

uploadFile(ServiceContext)

```
Synopsis: public ServiceContext uploadFile(sorcer.base.ServiceContext conte\
xt)

              throws java.rmi.RemoteException,
                     ServiceUnavailableException;
```

Parameters

`context`

the parameters as a ServiceContext

return

the result as a ServiceContext

Exceptions

`RemoteException`

if a remote io error occurs.

`ServiceUnavailableException`

if not all required services are available.

See Also

`uploadFile(sorcer.silenus.core.Msuid, java.util.Map)`,
`uploadFile(sorcer.silenus.core.Msuid, java.util.Map, sorcer.silenus.core.InputFileChannelAccessor)`

uploads a file.

Interface FileStore

Java interface to a FileStore.

Synopsis

```
package sorcer.silenus.core;

public interface FileStore {

  // Public Methods
  public MetadataStore.NodeCreated createNode(java.util.Map metadata)
      throws ServiceUnavailableException,
             java.rmi.RemoteException;
  public void deleteNode(sorcer.silenus.core.Msuid node,
                         boolean recursive)
      throws ServiceUnavailableException, java.io.IOException;
  public InputFileChannelAccessor downloadFile(sorcer.silenus.core.Msuid no\
de)

      throws ServiceUnavailableException,
             java.rmi.RemoteException;
  public Map expandNode(sorcer.silenus.core.Msuid node)
      throws ServiceUnavailableException,
             java.rmi.RemoteException;
  public Lease registerForEvents(net.jini.core.event.RemoteEventListener li\
stener,

                                 long desiredLease)
      throws net.jini.core.lease.LeaseDeniedException,
             java.rmi.RemoteException;
```

```
  public boolean replicateFile(sorcer.silenus.core.Msuid msuid,
                               net.jini.core.lookup.ServiceID byteStore)
      throws java.rmi.RemoteException;
  public Map setAttributes(sorcer.silenus.core.Msuid uuid,
                           java.util.Map newAttributes)
      throws ServiceUnavailableException,
             java.rmi.RemoteException;
  public OutputFileChannelAccessor uploadFile(sorcer.silenus.core.Msuid uid\
,

                                              java.util.Map metadata)
      throws ServiceUnavailableException,
             java.rmi.RemoteException;
  public void uploadFile(sorcer.silenus.core.Msuid uid,
                         java.util.Map metadata,
                         sorcer.silenus.core.InputFileChannelAccessor fileD\
ata)
      throws ServiceUnavailableException,
             java.rmi.RemoteException;
}
```

Version

$Revision: 1.4 $ $Date: 2006/10/12 01:41:34 $

See Also

`sorcer.silenus.core.SorcerFileStore`

Since

Nov 15, 2005

Inheritance Path. sorcer.silenus.core.FileStore

createNode(Map)

```
Synopsis: public MetadataStore.NodeCreated createNode(java.util.Map metadat\
a)

              throws ServiceUnavailableException,
                     java.rmi.RemoteException;
```

Parameters

`metadata`

the desired node metadata.

return

a NodeCreated object containing the actual metadata that was set and the msuid of the new node.

Exceptions

`ServiceUnavailableException`

not all services required to process this request are available.

`RemoteException`

If a remote IO error occurs.

Creates a new node with the given metadata.

A new node id will be automatically created.

deleteNode(Msuid, boolean)

```
Synopsis: public void deleteNode(sorcer.silenus.core.Msuid node,
                                 boolean recursive)
              throws ServiceUnavailableException,
                     java.io.IOException;
```

Parameters

`node`

the node to delete.

`recursive`

whether to delete all child nodes.

If set to true, all child nodes and their child nodes, etc. will be deleted.

If set to false, the node must not have any child nodes or an IOException will occur.

Exceptions

`ServiceUnavailableException`

not all services required to process this request are available.

`IOException`

If an IO error occurs.

See Also

`deleteNode(sorcer.silenus.core.Msuid, boolean)`

Deletes a node from this metadata store.

Deleting a node essentially sets most attributes to null.

downloadFile(Msuid)

```
Synopsis: public InputFileChannelAccessor downloadFile(sorcer.silenus.core.\
Msuid node)

              throws ServiceUnavailableException,
                     java.rmi.RemoteException;
```

Parameters

`node`

the id of the node to download.

return

a ReadableByteSequence with access to the file content.

Exceptions

`ServiceUnavailableException`

not all services required to process this request are available.

`RemoteException`

If a remote IO error occurs.

Tries to download the file with the given Msuid.

This function evaluates the location attribute, finds a suitable ByteStore and tries to retrieve the file contents from there.

expandNode(Msuid)

```
Synopsis: public Map expandNode(sorcer.silenus.core.Msuid node)
              throws ServiceUnavailableException,
                     java.rmi.RemoteException;
```

Parameters

`node`

the node to get information about

return

the node metadata.

Exceptions

`ServiceUnavailableException`

not all services required to process this request are available.

`RemoteException`

If a remote IO error occurs.

Gets the attributes for a given node.

This is the basic function to retrieve information stored in the SILENUS file system.

registerForEvents(RemoteEventListener, long)

```
Synopsis: public Lease registerForEvents(net.jini.core.event.RemoteEventLis\
tener listener,

                                         long desiredLease)
              throws net.jini.core.lease.LeaseDeniedException,
                     java.rmi.RemoteException;
```

Parameters

`listener`

the listener to register. Should be a remote proxy.

`desiredLease`

desired length of the lease.

return

a Lease object that can be used to renew the lease.

Exceptions

`LeaseDeniedException`

the lease request was denied.

`RemoteException`

If a remote IO error occurs.

See Also

`sorcer.silenus.core.FileStoreEvent`

Register a listener for FileStoreEvents.

This function is used to register listeners on events. In the case of a change, a `sorcer.silenus.core.FileStoreEvent` is sent.

replicateFile(Msuid, ServiceID)

```
Synopsis: public boolean replicateFile(sorcer.silenus.core.Msuid msuid,
                                       net.jini.core.lookup.ServiceID byteS\
tore)
              throws java.rmi.RemoteException;
```

Parameters

`msuid`

Uuid of the file to replicate

`byteStore`

Id of the bytestore to target. May be null.

return

true if the file was sucessfully replicated.

Exceptions

`RemoteException`

If a remote IO error occurs.

Initiate replication of a given file to a given bytestore.

setAttributes(Msuid, Map)

```
Synopsis: public Map setAttributes(sorcer.silenus.core.Msuid uuid,
                                   java.util.Map newAttributes)
              throws ServiceUnavailableException,
                     java.rmi.RemoteException;
```

Parameters

`uuid`

the id of the node to change.

`newAttributes`

set of new attributes. Use null to delete an attribute

return

the complete set of attributes after the change.

Exceptions

`ServiceUnavailableException`

not all services required to process this request are available.

`RemoteException`

If a remote IO error occurs.

Sets the attributes for a given Msuid to new values.

This function will only modify the attributes given as parameters. To delete an attribute, set it to null.

uploadFile(Msuid, Map)

```
Synopsis: public OutputFileChannelAccessor uploadFile(sorcer.silenus.core.M\
suid uid,

                                                      java.util.Map metadat\
a)
              throws ServiceUnavailableException,
                     java.rmi.RemoteException;
```

Parameters

`uid`

the Msuid of the node to update or null for new nodes.

`metadata`

the desired node metadata.

return

a WriteableByteSequence to fill in the file contents.

Exceptions

`ServiceUnavailableException`

not all services required to process this request are available.

`RemoteException`

If a remote IO error occurs.

See Also

`sorcer.silenus.core.OutputFileChannelAccessor, uploadFile(sorcer.silenus.core.Msuid, java.util.Map, sorcer.silenus.core.InputFileChannelAccessor)`

Allows uploading of file data for new and existing content nodes through push file upload.

If the given uid is null, a new node id will be created.

If the given uid exisits, a new content version for this particular file will be created.

A bytestore will be contacted and the location attribute filled in.

This method provides push file upload. It is the users responsibility to open the WriteableByteSequence, add data, and close it again.

uploadFile(Msuid, Map, InputFileChannelAccessor)

```
Synopsis: public void uploadFile(sorcer.silenus.core.Msuid uid,
                                 java.util.Map metadata,
                                 sorcer.silenus.core.InputFileChannelAccess\
or fileData)
              throws ServiceUnavailableException,
                     java.rmi.RemoteException;
```

Parameters

`uid`

the Msuid of the node to update or null for new nodes.

`metadata`

the desired node metadata.

`fileData`

a ReadableByteSequence with access to the file contents.

Exceptions

`ServiceUnavailableException`

not all services required to process this request are available.

`RemoteException`

If a remote IO error occurs.

See Also

`sorcer.silenus.core.InputFileChannelAccessor`,
`uploadFile(sorcer.silenus.core.Msuid, java.util.Map)`

Allows uploading of file data for new and existing content nodes through pull file upload.

If the given uid is null, a new node id will be created.

If the given uid exisits, a new content version for this particular file will be created.

A bytestore will be contacted and the location attribute filled in.

This method provides pull file upload. It will automatically contact the file content holder and retrieve the file from there.

Interface FileStoreConstants

Constants for MetadataStore

This interface provides constants for MetadataStores. It is implemented as an interface for easy inclusion (implement this interface to use the constants).

Synopsis

```
package sorcer.silenus.core;

public interface FileStoreConstants {

  // Public Static Fields
  public final static String ATTR_CHILDREN;
  public final static String ATTR_CONTENTLASTMODIFIED;
  public final static String ATTR_CREATIONDATE;
  public final static String ATTR_FILEVERSION;
  public final static String ATTR_LOCATION;
  public final static String ATTR_MAX_COPIES;
  public final static String ATTR_MD5;
  public final static String ATTR_METADATALASTMODIFIED;
  public final static String ATTR_MIN_COPIES;
  public final static String ATTR_NAME;
  public final static String ATTR_OPT_COPIES;
  public final static String ATTR_ORIGNIATOR;
  public final static String ATTR_PARENT;
  public final static String ATTR_SHA;
  public final static String ATTR_SIZE;
  public final static String ATTR_TARGET;
  public final static String ATTR_TYPE;
  public final static String ATTR_TYPE_WAS_SET_BY;
  public final static String EV_CONTEXT_DURATION;
  public final static String EV_CONTEXT_LEASE;
  public final static String EV_CONTEXT_LISTENER;
  public final static String FS_CONTEXT_ATTRIBUTELIST;
  public final static String FS_CONTEXT_ATTRIBUTENAME;
  public final static String FS_CONTEXT_ATTRIBUTES;
  public final static String FS_CONTEXT_ATTRIBUTEVALUE;
  public final static String FS_CONTEXT_CONTENT;
  public final static String FS_CONTEXT_OLD_ATTRIBUTES;
```

```
  public final static String FS_CONTEXT_RECURSIVE;
  public final static String FS_CONTEXT_SERVICEID;
  public final static String FS_CONTEXT_SUCCESS;
  public final static String FS_CONTEXT_TRANSACTION;
  public final static String FS_CONTEXT_UUID;
  public final static Map MAP_ATTR_DIGEST;
  public final static String MDS_CONTEXT_CHANGELOG;
  public final static String MDS_CONTEXT_MSUIDS;
  public final static String MDS_CONTEXT_TIMEVECTOR;
  public final static String MIMETYPE_DIRECTORY;
  public final static String MIMETYPE_LINK;
  public final static String TYPE_SET_CONTENT;
  public final static String TYPE_SET_EXT;
  public final static String TYPE_SET_OLDCONTENT;
  public final static String TYPE_SET_USER;
}
```

Version

$Revision: 1.5 $ $Date: 2006/10/02 23:07:22 $

See Also

sorcer.silenus.metadatastore.MetadataStore,

sorcer.silenus.metadatastore.SORCERMetadataStore

Inheritance Path. sorcer.silenus.core.FileStoreConstants

ATTR_CHILDREN

Synopsis: `public final static java.lang.String` **`ATTR_CHILDREN`**

Attribute for children.

This attribute is read-only.

Datatype: Collection<Msuid>.

ATTR_CONTENTLASTMODIFIED

Synopsis: `public final static java.lang.String` **`ATTR_CONTENTLASTMODIFIED`**

attribute name for Last modified date.

Datatype: String

ATTR_CREATIONDATE

Synopsis: `public final static java.lang.String` **`ATTR_CREATIONDATE`**

attribute name for creation date.
Datatype: String

ATTR_FILEVERSION

Synopsis: `public final static java.lang.String` **`ATTR_FILEVERSION`**

Attribute name for file version.
Datatype: String

ATTR_LOCATION

Synopsis: `public final static java.lang.String` **`ATTR_LOCATION`**

attribute name for location.
Datatype: Map<ServiceID, Bsuid>

ATTR_MAX_COPIES

Synopsis: `public final static java.lang.String` **`ATTR_MAX_COPIES`**

attribute for maximum number of available copies.
Datatype: long

ATTR_MD5

Synopsis: `public final static java.lang.String` **`ATTR_MD5`**

attribute name for MD5 checksum.

Datatype: String

ATTR_METADATALASTMODIFIED

Synopsis: public final static java.lang.String **ATTR_METADATALASTMODIFIED**

attribute name for Last modified date.
Datatype: String

ATTR_MIN_COPIES

Synopsis: public final static java.lang.String **ATTR_MIN_COPIES**

attribute for minimum number of available copies.
Datatype: long

ATTR_NAME

Synopsis: public final static java.lang.String **ATTR_NAME**

attribute name for filename.
Datatype: String

ATTR_OPT_COPIES

Synopsis: public final static java.lang.String **ATTR_OPT_COPIES**

attribute for optimal number of available copies.
Datatype: long

ATTR_ORIGNIATOR

Synopsis: public final static java.lang.String **ATTR_ORIGNIATOR**

attribute for originating metadata store.

Datatype: String

The originating store is the metadata store the file withh this uuid was last changed on.

ATTR_PARENT

Synopsis: `public final static java.lang.String` **`ATTR_PARENT`**

attribute name for parent node.

Datatype: String

ATTR_SHA

Synopsis: `public final static java.lang.String` **`ATTR_SHA`**

attribute name for SHA1 checksum.

Datatype: String

ATTR_SIZE

Synopsis: `public final static java.lang.String` **`ATTR_SIZE`**

attribute name for file size.

Datatype: String

ATTR_TARGET

Synopsis: `public final static java.lang.String` **`ATTR_TARGET`**

target for SILENUS links.

Datatype: Msuid

This value stores the target of soft links in the SILENUS file system. `ATTR_TYPE` should be set to `MIMETYPE_LINK`.

ATTR_TYPE

Synopsis: public final static java.lang.String **ATTR_TYPE**

attribute name for mime type.
Datatype: String

ATTR_TYPE_WAS_SET_BY

Synopsis: public final static java.lang.String **ATTR_TYPE_WAS_SET_BY**

See Also

TYPE_SET_CONTENT, TYPE_SET_EXT, TYPE_SET_USER

who has set the mime type?
Datatype: String

EV_CONTEXT_DURATION

Synopsis: public final static java.lang.String **EV_CONTEXT_DURATION**

Context path to a lease duration (long).

EV_CONTEXT_LEASE

Synopsis: public final static java.lang.String **EV_CONTEXT_LEASE**

Context path to a *net.jini.core.lease.Lease*.

EV_CONTEXT_LISTENER

Synopsis: public final static java.lang.String **EV_CONTEXT_LISTENER**

Context path to a *net.jini.core.event.RemoteEventListener*.

FS_CONTEXT_ATTRIBUTELIST

Synopsis: `public final static java.lang.String` **`FS_CONTEXT_ATTRIBUTELIST`**

Context path to a single file attribute.

FS_CONTEXT_ATTRIBUTENAME

Synopsis: `public final static java.lang.String` **`FS_CONTEXT_ATTRIBUTENAME`**

Context path to a single file attribute name.

FS_CONTEXT_ATTRIBUTES

Synopsis: `public final static java.lang.String` **`FS_CONTEXT_ATTRIBUTES`**

Context path to file attributes.

FS_CONTEXT_ATTRIBUTEVALUE

Synopsis: `public final static java.lang.String` **`FS_CONTEXT_ATTRIBUTEVALUE`**

Context path to a single file attribute name.

FS_CONTEXT_CONTENT

Synopsis: `public final static java.lang.String` **`FS_CONTEXT_CONTENT`**

Contenxt path to file contents. Can be either a `sorcer.silenus.core.InputFileChannelAccessor` or a `sorcer.silenus.core.OutputFileChannelAccessor`.

FS_CONTEXT_OLD_ATTRIBUTES

Synopsis: `public final static java.lang.String` **`FS_CONTEXT_OLD_ATTRIBUTES`**

Context path to old file attributes.

FS_CONTEXT_RECURSIVE

Synopsis: `public final static java.lang.String` **`FS_CONTEXT_RECURSIVE`**

Context path to the recursive attribute of type *java.lang.Boolean*.

FS_CONTEXT_SERVICEID

Synopsis: `public final static java.lang.String` **`FS_CONTEXT_SERVICEID`**

Context path to a *ServiceID*.

FS_CONTEXT_SUCCESS

Synopsis: `public final static java.lang.String` **`FS_CONTEXT_SUCCESS`**

Context path to a bool.

FS_CONTEXT_TRANSACTION

Synopsis: `public final static java.lang.String` **`FS_CONTEXT_TRANSACTION`**

Context path to a *net.jini.core.transaction.server.ServerTransaction* object.

FS_CONTEXT_UUID

Synopsis: `public final static java.lang.String` **`FS_CONTEXT_UUID`**

Context path to a file `sorcer.silenus.core.Msuid`.

MAP_ATTR_DIGEST

Synopsis: public final static java.util.Map **MAP_ATTR_DIGEST**

mapping from attribute names for message digests to names common in java security providers.

MDS_CONTEXT_CHANGELOG

Synopsis: public final static java.lang.String **MDS_CONTEXT_CHANGELOG**

Context path to a file store change log.

MDS_CONTEXT_MSUIDS

Synopsis: public final static java.lang.String **MDS_CONTEXT_MSUIDS**

Context path to a list of Msuids.

MDS_CONTEXT_TIMEVECTOR

Synopsis: public final static java.lang.String **MDS_CONTEXT_TIMEVECTOR**

Context path to a time vector.

MIMETYPE_DIRECTORY

Synopsis: public final static java.lang.String **MIMETYPE_DIRECTORY**

See Also

ATTR_TARGET

special mime type for directories. You can use this to check if a node is a directory. Please note: Links to directories will also have children.

MIMETYPE_LINK

Synopsis: public final static java.lang.String **MIMETYPE_LINK**

special mime type for links. A link can point to a directory or a file.

TYPE_SET_CONTENT

Synopsis: public final static java.lang.String **TYPE_SET_CONTENT**

type was set from file content.

TYPE_SET_EXT

Synopsis: public final static java.lang.String **TYPE_SET_EXT**

type was set from file extension.

TYPE_SET_OLDCONTENT

Synopsis: public final static java.lang.String **TYPE_SET_OLDCONTENT**

type was set from older fileversion content.

TYPE_SET_USER

Synopsis: public final static java.lang.String **TYPE_SET_USER**

type was set by the user.

Class FileStoreEvent

This class represents events within the SILENUS file store system.

A file store event is sent everytime something changes to all interested parties. Every file store event is designed so that is may be missed without implications.

Synopsis

```
package sorcer.silenus.core;
```

```
public class FileStoreEvent extends RemoteEvent  {

  // Public Static Fields
  public final static long ALIVE_EVENT;
  public final static long CREATION_FILEDATA_EVENT;
  public final static long CREATION_METADATA_EVENT;
  public final static long HAS_SYNCHED_EVENT;
  public final static long UPDATE_FILEDATA_EVENT;
  public final static long UPDATE_METADATA_EVENT;

  // Public Constructors
  public FileStoreEvent(net.jini.core.lookup.ServiceID sourceService,
                        long seqNum, java.util.Map timeVec);
  public FileStoreEvent(net.jini.core.lookup.ServiceID sourceService,
                        long eventID, long seqNum,
                        java.util.Set changedSourceItems,
                        java.util.Map timeVec,
                        java.util.Map attrs);

  // Public Methods
  public Map getChangedAttrs();
  public Set getSourceItems();
  public Map getTimeVector();
}
```

Methods inherited from net.jini.core.event.RemoteEvent: `getID`, `getRegistrationObject`, `getSequenceNumber`

Methods inherited from java.util.EventObject: `getSource`, `toString`

Methods inherited from java.lang.Object: `clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `wait`

Version

$Revision: 1.2 $ $Date: 2006/09/02 19:26:36 $

Inheritance Path. java.lang.Object-> java.util.EventObject-> net.jini.core.event.RemoteEvent-> sorcer.silenus.core.FileStoreEvent

FileStoreEvent(ServiceID, long, long, Set, Map, Map)

Synopsis: public **FileStoreEvent**(net.jini.core.lookup.ServiceID *sourceServic\
e*,

long *eventID*, long *seqNum*,
java.util.Set *changedSourceItems*,
java.util.Map *timeVec*,
java.util.Map *attrs*);

Parameters

`sourceService`
the serviceID of the service where the event occured.

`eventID`
type of the event. Please use the constants defined in this class.

`seqNum`
the sequence number of the event. Should increase with every event. This is ignored in the SILENUS core components.

`changedSourceItems`
a set of Uuids of the file store items that have changed. Use this and the sourceService to acquire additional information.

`timeVec`
the timestamp for the change event.

`attrs`
the attributes that have changed during this event. Only defined if there is exactly one source item.

Creates a new file store event.

This event can then be sent to all SILENUS listeners.

FileStoreEvent(ServiceID, long, Map)

Synopsis: public **FileStoreEvent**(net.jini.core.lookup.ServiceID *sourceServic\
e*,

long *seqNum*,
java.util.Map *timeVec*);

Parameters

`sourceService`
the service id of the service that is alive

`seqNum`
a sequence number. Should increase with every event. This is ignored in the SILENUS core components.

`timeVec`
the timestamp at the originating service.

Generates a new `ALIVE_EVENT`.

ALIVE_EVENT

Synopsis: public final static long **ALIVE_EVENT**

Nothing has changed. This is just to imform that the source node is alive.

CREATION_FILEDATA_EVENT

Synopsis: public final static long **CREATION_FILEDATA_EVENT**

Actual byte data for a new file has been stored.

CREATION_METADATA_EVENT

Synopsis: public final static long **CREATION_METADATA_EVENT**

Metadata for a new file has been created.

HAS_SYNCHED_EVENT

Synopsis: public final static long **HAS_SYNCHED_EVENT**

To inform all listeners that we have synched with s/o else. In this case, sourceItem will not be set, since there were multiple changes.

UPDATE_FILEDATA_EVENT

Synopsis: public final static long **UPDATE_FILEDATA_EVENT**

File content for an exisiting file has been changed.

UPDATE_METADATA_EVENT

Synopsis: public final static long **UPDATE_METADATA_EVENT**

Metadata for an existing file has changed.

getChangedAttrs()

Synopsis: public Map **getChangedAttrs**();

Parameters

return

a map of attributes or null

Returns the attributes that have changed for sourceItem.

This is only defined if there is exactly one source item.

getSourceItems()

Synopsis: public Set **getSourceItems**();

Parameters

return

a set of Uuids of the changed items or null.

Returns the sourceItems. The sourceItem is a set of Uuids of the files which have changed.

getTimeVector()

```
Synopsis: public Map getTimeVector();
```

Parameters

return

the time vector.

Returns the timeVector. The timeVector contains the timestamp of the event.

Interface InputFileChannelAccessor

A readable byte sequence accessor. This class is serializable. It contains all information necesary to open a readable byte channel.

Synopsis

```
package sorcer.silenus.core;

public interface InputFileChannelAccessor implements Serializable {

  // Public Methods
  public FileChannel openInputFileChannel() throws java.io.IOException;
}
```

Version

$Revision: 1.1 $ $Date: 2006/09/02 19:26:36 $

Since

Nov 15, 2005

Inheritance Path. sorcer.silenus.core.InputFileChannelAccessor

openInputFileChannel()

```
Synopsis: public FileChannel openInputFileChannel() throws
            java.io.IOException;
```

Parameters

return

a newly created and opened readable byte channel.

Exceptions

`IOException`

if an IO error occurs.

create the readable byte channel that is stored in this sequence.

Interface MetadataStore

Java interface to a MetadataStore.

Synopsis

```
package sorcer.silenus.core;

public interface MetadataStore {

  // Public Methods
  public MetadataStore.NodeCreated createNode(java.util.Map attributes,
                                               net.jini.core.transaction.ser\
ver.ServerTransaction transaction)
      throws java.rmi.RemoteException;
  public void deleteNode(sorcer.silenus.core.Msuid node,
                         boolean recursive) throws java.io.IOException;
  public Map expandNode(sorcer.silenus.core.Msuid node)
```

```
      throws java.rmi.RemoteException;
  public ServiceID getProviderID() throws java.rmi.RemoteException;
  public Map getTimeVector() throws java.rmi.RemoteException;
  public Lease registerForEvents(net.jini.core.event.RemoteEventListener li\
stener,

                                 long desiredLease)
      throws net.jini.core.lease.LeaseDeniedException,
             java.rmi.RemoteException;
  public MetadataStore.MetadataStoreChangeLog retrieveChangeLogSince(java.u\
til.Map timeVector)

      throws java.rmi.RemoteException;
  public Collection retrieveListOfAllActiveNodes() throws
    java.rmi.RemoteException;
  public Map updateNode(sorcer.silenus.core.Msuid node,
                        java.util.Map newAttributes,
                        java.util.Map oldAttributes,
                        net.jini.core.transaction.server.ServerTransaction \
transaction)
      throws java.rmi.RemoteException;
}
```

Version

$Revision: 1.3 $ $Date: 2006/09/28 00:54:28 $

See Also

`sorcer.silenus.core.SorcerMetadataStore`

Since

Nov 15, 2005

Inheritance Path. sorcer.silenus.core.MetadataStore

createNode(Map, ServerTransaction)

```
Synopsis: public MetadataStore.NodeCreated createNode(java.util.Map attribu\
tes,

                                                       net.jini.core.transac\
tion.server.ServerTransaction transaction)
```

```
            throws java.rmi.RemoteException;
```

Parameters

`attributes`

the attributes desired.

`transaction`

a ServerTransaction if needed.

return

a NodeCreated object containing the actual attributes that where set and the Msuid of the new object.

Exceptions

`RemoteException`

If a remote IO error occurs.

creates a node.

deleteNode(Msuid, boolean)

```
Synopsis: public void deleteNode(sorcer.silenus.core.Msuid node,
                                 boolean recursive)
            throws java.io.IOException;
```

Parameters

`node`

the node to delete.

`recursive`

whether to delete all child nodes.

If set to true, all child nodes and their child nodes, etc. will be deleted.

If set to false, the node must not have any child nodes or an IOException will occur.

Exceptions

`IOException`

If an IO error occurs.

See Also

`deleteNode(sorcer.silenus.core.Msuid, boolean)`

Deletes a node from the file store.

This operations is forwared to a metadata store.

expandNode(Msuid)

```
Synopsis: public Map expandNode(sorcer.silenus.core.Msuid node)
              throws java.rmi.RemoteException;
```

Parameters

`node`

the node id to expand

return

a Map<String,Object> with key-value pairs

Exceptions

`RemoteException`

If a remote IO error occurs.

See Also

`sorcer.silenus.core.FileStoreConstants`

Returns metainformation for the given node.

This is the main function to acuire metainformation for a given node. It returns key-value pairs with the information. Most values will be of type string, but other object types are possible. Please see the list of `sorcer.silenus.core.FileStoreConstants`.

Most notable attributes are:

- FileName: ATTR_NAME
- FileType: ATTR_TYPE
- List of children: ATTR_CHILDREN

getProviderID()

```
Synopsis: public ServiceID getProviderID() throws
            java.rmi.RemoteException;
```

Parameters

return

the ID of this provider

Exceptions

RemoteException

If a remote IO error occurs.

Standard method to receive the ServiceID of this provider.

getTimeVector()

```
Synopsis: public Map getTimeVector() throws java.rmi.RemoteException;
```

Parameters

return

the time vector.

Exceptions

RemoteException

If a remote IO error occurs.

Asks a metadata store for its current time vector.

The time vector can be used to check if a metadata store is in synch.

registerForEvents(RemoteEventListener, long)

```
Synopsis: public Lease registerForEvents(net.jini.core.event.RemoteEventLis\
tener listener,

                                         long desiredLease)
              throws net.jini.core.lease.LeaseDeniedException,
                     java.rmi.RemoteException;
```

Parameters

`listener`

the client listener.

`desiredLease`

length of the desired lease.

return

a Lease object that can be used to renew the lease.

Exceptions

`LeaseDeniedException`

if the lease cannot be granted.

`RemoteException`

If a remote IO error occurs.

See Also

`sorcer.silenus.core.FileStoreEvent`

Registers a client to receive remote events from this metadatastore.

A listener registered with a metadatastore will receive messages of type `sorcer.silenus.core.FileStoreEvent`

retrieveChangeLogSince(Map)

```
Synopsis: public MetadataStore.MetadataStoreChangeLog retrieveChangeLogSinc\
e(java.util.Map timeVector)
```

```
            throws java.rmi.RemoteException;
```

Parameters

`timeVector`

the time vector of the caller.

return

a `sorcer.silenus.core.MetadataStore.MetadataStoreChangeLog` object with the information.

Exceptions

`RemoteException`

If a remote IO error occurs.

Retrieves all the changes that have happend since the given time vector.

retrieveListOfAllActiveNodes()

```
Synopsis: public Collection retrieveListOfAllActiveNodes() throws
            java.rmi.RemoteException;
```

Parameters

return

a list of Msuid

Exceptions

`RemoteException`

If a remote IO error occurs.

Retrieves a list of all items stored in this metadata store that are still active.

An active item is an item that has a parent (is not deleted).

updateNode(Msuid, Map, Map, ServerTransaction)

```
Synopsis: public Map updateNode(sorcer.silenus.core.Msuid node,
```

```
                                java.util.Map newAttributes,
                                java.util.Map oldAttributes,
                                net.jini.core.transaction.server.ServerTran\
saction transaction)
              throws java.rmi.RemoteException;
```

Parameters

`newAttributes`

the new attributes to set.

`node`

the node to update.

`oldAttributes`

old attributes that must be still be set for the command to execute or null.

`transaction`

a ServerTransaction if needed or null if not.

return

the current attributes of the node given.

Exceptions

`RemoteException`

If a remote IO error occurs.

Updates a node with the given attributes.

This only updates the values given. You must set a value explicitly to null to delete it.

Class MetadataStore.MetadataStoreChangeLog

A class representing a metadata store change log.

It contains a list of changed items and their changes. It also contains the current time vector.

Synopsis

```
package sorcer.silenus.core.MetadataStore;
```

```
public static class MetadataStore.MetadataStoreChangeLog implements Seriali\
zable {

  // Public Constructors
  public MetadataStore.MetadataStoreChangeLog(java.util.Map theTimeVector,
                                              java.util.Map theChangedAttrs\
);

  // Public Methods
  public Map getChangedAttrs();
  public Map getTimeVector();
}
```

Methods inherited from java.lang.Object: `clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`

Inheritance Path. java.lang.Object-> sorcer.silenus.core.MetadataStore.MetadataStoreChangeLog

MetadataStore.MetadataStoreChangeLog(Map, Map)

```
Synopsis: public MetadataStore.MetadataStoreChangeLog(java.util.Map theTime\
Vector,

                                                      java.util.Map theChan\
gedAttrs);
```

Parameters

`theTimeVector`

the time vector;

`theChangedAttrs`

the changed attributes.

Creates a new changelog with the new time vector (the later time) and the given changes.

getChangedAttrs()

```
Synopsis: public Map getChangedAttrs();
```

Parameters

return

the list of changed items.

Contains a list of changed items and the attributes that have changed.

getTimeVector()

```
Synopsis: public Map getTimeVector();
```

Parameters

return

the current time vector.

Contains the current time vector.

Class MetadataStore.NodeCreated

Data class for created mds objects.

Synopsis

```
package sorcer.silenus.core.MetadataStore;

public static class MetadataStore.NodeCreated implements Serializable {

  // Public Constructors
  public MetadataStore.NodeCreated(sorcer.silenus.core.Msuid uid,
                                   java.util.Map attrs);

  // Public Methods
  public Map getAttributes();
  public Msuid getMsuid();
```

```
}
```

Methods inherited from java.lang.Object: `clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`

Inheritance Path. java.lang.Object-> sorcer.silenus.core.MetadataStore.NodeCreated

MetadataStore.NodeCreated(Msuid, Map)

```
Synopsis: public MetadataStore.NodeCreated(sorcer.silenus.core.Msuid uid,
                                           java.util.Map attrs);
```

Parameters

`attrs`
Attributes of the new node.

`uid`
msuid of the new node.

Creates a new NodeCreated object.

getAttributes()

```
Synopsis: public Map getAttributes();
```

Parameters

return
Returns the attributes.

Getter method for property attributes.

getMsuid()

```
Synopsis: public Msuid getMsuid();
```

Parameters

return

Returns the msuid.

Getter method for property msuid.

Class Msuid

Class to represent objects store in the metadata store.

This class provides UIDs that can be extended by adding ServiceIDs. This is necessary for synchronization.

Synopsis

```
package sorcer.silenus.core;

public class Msuid implements Serializable {

  // Public Static Fields
  public final static Msuid ROOTID;

  // Public Static Methods
  public static Msuid fromString(java.lang.String name);
  public static Msuid randomMsuid();

  // Public Methods
  public boolean equals(java.lang.Object obj);
  public int hashCode();
  public String toString();
  public Msuid withOriginatorID(net.jini.core.lookup.ServiceID serviceID);
}
```

Methods inherited from java.lang.Object: `clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`

Version

$Revision: 1.1 $ $Date: 2006/09/02 19:26:36 $

Inheritance Path. java.lang.Object-> sorcer.silenus.core.Msuid

ROOTID

Synopsis: public final static sorcer.silenus.core.Msuid **ROOTID**

Id for the object at the root node.

equals(Object)

Synopsis: public boolean **equals**(java.lang.Object *obj*);

fromString(String)

Synopsis: public static Msuid **fromString**(java.lang.String *name*);

Parameters

name

the string to parse.

return

a Msuid object, if possible, or null if not.

Tries to create a Msuid from a given String.

hashCode()

Synopsis: public int **hashCode**();

randomMsuid()

Synopsis: public static Msuid **randomMsuid**();

Parameters

return

a valid Msuid.

Creates a random Msuid.

toString()

Synopsis: `public String` **`toString`**`();`

Parameters

return

a String representation

Creates a String representation for this Msuid.

This representation can be parsed with `fromString(java.lang.String)`

withOriginatorID(ServiceID)

Synopsis: `public Msuid` **`withOriginatorID`**`(net.jini.core.lookup.ServiceID` *`serv\ iceID`*`);`

Parameters

`serviceID`

the new originatorID

return

the new Msuid

returns a new instance representing a UUID with the same itemID but with a different originatorID.

Interface OutputFileChannelAccessor

A writable byte sequence accessor. This class is serializable. It contains all information necesary to open a writable byte channel.

Synopsis

```
package sorcer.silenus.core;

public interface OutputFileChannelAccessor implements Serializable {

  // Public Methods
  public FileChannel openOutputFileChannel() throws java.io.IOException;
}
```

Version

$Revision: 1.1 $ $Date: 2006/09/02 19:26:36 $

Since

Nov 15, 2005

Inheritance Path. sorcer.silenus.core.OutputFileChannelAccessor

openOutputFileChannel()

```
Synopsis: public FileChannel openOutputFileChannel() throws
            java.io.IOException;
```

Parameters

return

an open, writeable byte channel that can be used to write data.

Exceptions

`IOException`

if an io error occurs

opens up a WriteableByteChannel that can be used to write data.

Interface RemoteSilenusAccessor

Interface to a service providing access to other SILENUS services.

Synopsis

```
package sorcer.silenus.core;

public interface RemoteSilenusAccessor implements Remote {

  // Public Methods
  public MetadataStore getMetadataStore(net.jini.core.lookup.ServiceID oldI\
D)

      throws ServiceUnavailableException,
             java.rmi.RemoteException;
}
```

Inheritance Path. sorcer.silenus.core.RemoteSilenusAccessor

getMetadataStore(ServiceID)

```
Synopsis: public MetadataStore getMetadataStore(net.jini.core.lookup.Servic\
eID oldID)

              throws ServiceUnavailableException,
                     java.rmi.RemoteException;
```

Parameters

`oldID`

invalidate this metadatastore if given. May be null.

return

A proxy to a metadatastore.

Exceptions

`ServiceUnavailableException`

if no metadata store could be found

`RemoteException`

if a remote io error occurs.

Retrieves a proxy to a running metadatastore.

Exception ServiceUnavailableException

Exception class that states that a needed service is currently unavailable.

Synopsis

```
package sorcer.silenus.core;

public class ServiceUnavailableException extends Exception  {

  // Public Constructors
  public ServiceUnavailableException(java.lang.String whichService);
}
```

Methods inherited from java.lang.Throwable: `fillInStackTrace`, `getCause`, `getLocalizedMessage`, `getMessage`, `getStackTrace`, `initCause`, `printStackTrace`, `setStackTrace`, `toString`

Methods inherited from java.lang.Object: `clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `wait`

Version

$Revision: 1.2 $ $Date: 2006/09/02 19:26:36 $

Inheritance Path. java.lang.Object-> java.lang.Throwable-> java.lang.Exception-> sorcer.silenus.core.ServiceUnavailableException

ServiceUnavailableException(String)

```
Synopsis: public ServiceUnavailableException(java.lang.String whichService)\
;
```

Parameters

`whichService`

the name of the service that is unavailable.

Creates a new ServiceUnavailableException.

Interface SorcerByteStore

SORCER interface to a ByteStore.

Synopsis

```
package sorcer.silenus.core;

public interface SorcerByteStore implements Remote {

  // Public Methods
  public ServiceContext createByteSequence(sorcer.base.ServiceContext param\
)

      throws java.rmi.RemoteException;
  public ServiceContext getByteSequence(sorcer.base.ServiceContext param)
      throws java.io.IOException;
  public ServiceContext getFileAttribute(sorcer.base.ServiceContext param)
      throws java.io.IOException;
  public ServiceID getProviderID() throws java.rmi.RemoteException;
  public ServiceContext getSupportedAttributes(sorcer.base.ServiceContext p\
aram)

      throws java.rmi.RemoteException;
}
```

Version

$Revision: 1.2 $ $Date: 2006/09/02 19:26:36 $

Since

Nov 15, 2005

See Also

`sorcer.silenus.core.ByteStore`

Inheritance Path. sorcer.silenus.core.SorcerByteStore

createByteSequence(ServiceContext)

```
Synopsis: public ServiceContext createByteSequence(sorcer.base.ServiceConte\
xt param)
```

```
            throws java.rmi.RemoteException;
```

Parameters

`param`

the parameters as a ServiceContext

return

the result as a ServiceContext

Exceptions

`RemoteException`

if a remote io error occurs.

See Also

`createByteSequence(sorcer.silenus.core.Bsuid, net.jini.core.transaction.server.ServerTransaction, java.util.Map)`, `createByteSequence(sorcer.silenus.core.Bsuid, net.jini.core.transaction.server.ServerTransaction, java.util.Map, sorcer.silenus.core.InputFileChannelAccessor)`

create a new byte sequence on this store.

getByteSequence(ServiceContext)

Synopsis: `public ServiceContext` **`getByteSequence`**`(sorcer.base.ServiceContext \ param)`

```
            throws java.io.IOException;
```

Parameters

`param`

the parameters as a ServiceContext

return

the result as a ServiceContext

Exceptions

`IOException`

if a io error occurs.

See Also

`getByteSequence(sorcer.silenus.core.Bsuid)`

retrieve an accessor to a stored byte sequence.

getFileAttribute(ServiceContext)

```
Synopsis: public ServiceContext getFileAttribute(sorcer.base.ServiceContext\
 param)

              throws java.io.IOException;
```

Parameters

`param`

the parameters as a ServiceContext

return

the result as a ServiceContext

Exceptions

`IOException`

if a io error occurs.

See Also

`getFileAttribute(sorcer.silenus.core.Bsuid, java.lang.String)`

Retrieves an intrisic attribute for a stored byte sequence.

getProviderID()

Synopsis: `public ServiceID` **`getProviderID`**`() throws`
`java.rmi.RemoteException;`

Parameters

return

the providers ID.

Exceptions

`RemoteException`

if a remote io error occurs.

See Also

`getProviderID()`

get the ID of this provider.

getSupportedAttributes(ServiceContext)

Synopsis: `public ServiceContext` **`getSupportedAttributes`**`(sorcer.base.ServiceC\`
`ontext` *`param`*`)`

`throws java.rmi.RemoteException;`

Parameters

`param`

the parameters as a ServiceContext

return

the result as a ServiceContext

Exceptions

`RemoteException`

if a remote io error occurs.

See Also

`getSupportedAttributes()`

Retrieves a list of intrinsic attributes suported on this byte store.

Interface SorcerFileStore

SORCER interface to a FileStore.

Synopsis

```
package sorcer.silenus.core;

public interface SorcerFileStore implements Remote,Coordinator {

  // Public Methods
  public ServiceContext createNode(sorcer.base.ServiceContext pc)
      throws java.rmi.RemoteException,
             ServiceUnavailableException;
  public ServiceContext deleteNode(sorcer.base.ServiceContext pc)
      throws ServiceUnavailableException,
             java.io.IOException;
  public ServiceContext expandNode(sorcer.base.ServiceContext context)
      throws java.rmi.RemoteException,
             ServiceUnavailableException;
  public ServiceContext setAttributes(sorcer.base.ServiceContext pc)
      throws java.rmi.RemoteException,
             ServiceUnavailableException;
}
```

Version

$Revision: 1.3 $ $Date: 2006/10/02 05:44:31 $

See Also

`sorcer.silenus.core.FileStore`

Since

Nov 15, 2005

Inheritance Path. sorcer.silenus.core.SorcerFileStore

createNode(ServiceContext)

```
Synopsis: public ServiceContext createNode(sorcer.base.ServiceContext pc)
              throws java.rmi.RemoteException,
                     ServiceUnavailableException;
```

Parameters

`pc`

the parameters as a ServiceContext

return

the result as a ServiceContext

Exceptions

`RemoteException`

if a remote io error occurs.

`ServiceUnavailableException`

if not all required services are available.

See Also

`createNode(java.util.Map)`

create a new node.

deleteNode(ServiceContext)

```
Synopsis: public ServiceContext deleteNode(sorcer.base.ServiceContext pc)
              throws ServiceUnavailableException,
                     java.io.IOException;
```

Parameters

`pc`

the parameters as a ServiceContext

return

the result as a ServiceContext

Exceptions

`ServiceUnavailableException`

if not all required services are available.

`IOException`

if a io error occurs.

See Also

`deleteNode(sorcer.silenus.core.Msuid, boolean)`

delete a node.

expandNode(ServiceContext)

```
Synopsis: public ServiceContext expandNode(sorcer.base.ServiceContext conte\
xt)

              throws java.rmi.RemoteException,
                     ServiceUnavailableException;
```

Parameters

`context`

the parameters as a ServiceContext

return

the result as a ServiceContext

Exceptions

`RemoteException`

if a remote io error occurs.

`ServiceUnavailableException`

if not all required services are available.

See Also

`expandNode(sorcer.silenus.core.Msuid)`

expand a node.

setAttributes(ServiceContext)

```
Synopsis: public ServiceContext setAttributes(sorcer.base.ServiceContext pc\
)

              throws java.rmi.RemoteException,
                     ServiceUnavailableException;
```

Parameters

`pc`

the parameters as a ServiceContext

return

the result as a ServiceContext

Exceptions

`RemoteException`

if a remote io error occurs.

`ServiceUnavailableException`

if not all required services are available.

See Also

`setAttributes(sorcer.silenus.core.Msuid, java.util.Map)`

Set attributes for a node.

Interface SorcerMetadataStore

SORCER interface to a MetadataStore.

Synopsis

```
package sorcer.silenus.core;

public interface SorcerMetadataStore implements Remote {

  // Public Methods
  public ServiceContext createNode(sorcer.base.ServiceContext context)
      throws java.rmi.RemoteException;
  public ServiceContext deleteNode(sorcer.base.ServiceContext pc)
      throws java.io.IOException;
  public ServiceContext expandNode(sorcer.base.ServiceContext context)
      throws java.rmi.RemoteException;
  public ServiceContext getTimeVector(sorcer.base.ServiceContext pc)
      throws java.rmi.RemoteException;
  public ServiceContext registerForEvents(sorcer.base.ServiceContext pc)
```

```
      throws java.rmi.RemoteException,
             net.jini.core.lease.LeaseDeniedException;
  public ServiceContext retrieveChangeLogSince(sorcer.base.ServiceContext p\
c)

      throws java.rmi.RemoteException;
  public ServiceContext retrieveListOfAllActiveNodes(sorcer.base.ServiceCon\
text pc)

      throws java.rmi.RemoteException;
  public ServiceContext updateNode(sorcer.base.ServiceContext context)
      throws java.rmi.RemoteException;
}
```

Version

$Revision: 1.3 $ $Date: 2006/10/12 01:29:51 $

Since

Nov 15, 2005

Inheritance Path. sorcer.silenus.core.SorcerMetadataStore

createNode(ServiceContext)

```
Synopsis: public ServiceContext createNode(sorcer.base.ServiceContext conte\
xt)

              throws java.rmi.RemoteException;
```

Parameters

`context`

the parameters as a ServiceContext

return

the result as a ServiceContext

Exceptions

`RemoteException`

if a remote io error occurs.

See Also

`createNode(java.util.Map, net.jini.core.transaction.server.ServerTransaction)`

create a node.

deleteNode(ServiceContext)

```
Synopsis: public ServiceContext deleteNode(sorcer.base.ServiceContext pc)
              throws java.io.IOException;
```

Parameters

`pc`

the parameters as a ServiceContext

return

the result as a ServiceContext

Exceptions

`IOException`

if a io error occurs.

See Also

`deleteNode(sorcer.silenus.core.Msuid, boolean)`

delete a node.

expandNode(ServiceContext)

```
Synopsis: public ServiceContext expandNode(sorcer.base.ServiceContext conte\
xt)
              throws java.rmi.RemoteException;
```

Parameters

`context`

the parameters as a ServiceContext

return

the result as a ServiceContext

Exceptions

`RemoteException`

if a remote io error occurs.

See Also

`expandNode(sorcer.silenus.core.Msuid)`

expand a node.

getTimeVector(ServiceContext)

```
Synopsis: public ServiceContext getTimeVector(sorcer.base.ServiceContext pc\
)

              throws java.rmi.RemoteException;
```

Parameters

`pc`

the parameters as a ServiceContext

return

the result as a ServiceContext

Exceptions

`RemoteException`

if a remote io error occurs.

See Also

`getTimeVector()`

retrieve the current time vector.

registerForEvents(ServiceContext)

```
Synopsis: public ServiceContext registerForEvents(sorcer.base.ServiceContex\
```

```
t pc)

              throws java.rmi.RemoteException,
                     net.jini.core.lease.LeaseDeniedException;
```

Parameters

pc

the parameters as a ServiceContext

return

the result as a ServiceContext

Exceptions

RemoteException

if a remote io error occurs.

LeaseDeniedException

if the Lease requested cannot be granted.

register for MDS events.

retrieveChangeLogSince(ServiceContext)

```
Synopsis: public ServiceContext retrieveChangeLogSince(sorcer.base.ServiceC\
ontext pc)

              throws java.rmi.RemoteException;
```

Parameters

pc

the parameters as a ServiceContext

return

the result as a ServiceContext

Exceptions

RemoteException

if a remote io error occurs.

See Also

`retrieveChangeLogSince(java.util.Map)`

retrieve a change log.

retrieveListOfAllActiveNodes(ServiceContext)

```
Synopsis: public ServiceContext retrieveListOfAllActiveNodes(sorcer.base.Se\
rviceContext pc)

              throws java.rmi.RemoteException;
```

Parameters

`pc`

the incoming context. Not used.

return

a collection of Msuids

Exceptions

`RemoteException`

If a remote IO error occurs. *

See Also

`retrieveListOfAllActiveNodes()`

Retrieves a list of all items stored in this metadata store that are still active.

An active item is an item that has a parent (is not deleted).

updateNode(ServiceContext)

```
Synopsis: public ServiceContext updateNode(sorcer.base.ServiceContext conte\
xt)

              throws java.rmi.RemoteException;
```

Parameters

`context`

the parameters as a ServiceContext

return

the result as a ServiceContext

Exceptions

`RemoteException`

if a remote io error occurs.

See Also

`updateNode(sorcer.silenus.core.Msuid, java.util.Map, java.util.Map, net.jini.core.transaction.server.ServerTransaction)`

modify a node.

Class Time

Describes a single logical time element consisting of local and global time.

Synopsis

```
package sorcer.silenus.core;

public class Time implements Serializable {

  // Public Constructors
  public Time();
  public Time(long localTime, long globalTime);

  // Public Methods
  public long getGlobal();
  public long getLocal();
  public void incrementGlobal();
  public void incrementLocalAndGlobal();
  public void setGlobal(long newGlobal);
```

```
  public void setLocal(long newLocal);
  public String toString();
}
```

Methods inherited from java.lang.Object: `clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`

Version

$Revision: 1.2 $ $Date: 2006/09/02 19:26:36 $

Inheritance Path. java.lang.Object-> sorcer.silenus.core.Time

Time()

Synopsis: `public Time();`

Creates a new logical time with local and global values of 0.

Time(long, long)

Synopsis: `public Time(long localTime, long globalTime);`

Parameters

`localTime`

local time.

`globalTime`

global time.

Creates a new logical time with the given local and global values.

getGlobal()

Synopsis: `public long getGlobal();`

Parameters

return

the global time component.

Returns the global time component.

getLocal()

```
Synopsis: public long getLocal();
```

Parameters

return

the local time component.

Returns the local time component.

incrementGlobal()

```
Synopsis: public void incrementGlobal();
```

Increments just the global time component.

incrementLocalAndGlobal()

```
Synopsis: public void incrementLocalAndGlobal();
```

Increments both the local and global time component.

setGlobal(long)

```
Synopsis: public void setGlobal(long newGlobal);
```

Parameters

`newGlobal`

The global time component to set.

Sets the global time component.

setLocal(long)

```
Synopsis: public void setLocal(long newLocal);
```

Parameters

`newLocal`

The local time component to set.

sets the local time component.

toString()

```
Synopsis: public String toString();
```

Constant field values

Package sorcer.silenus.core.*

ALIVE_EVENT	0
ATTR_CHILDREN	children
ATTR_CONTENTLASTMODIFIED	getlastmodified
ATTR_CREATIONDATE	creationdate
ATTR_FILEVERSION	fileversion
ATTR_LOCATION	location
ATTR_MAX_COPIES	maxcopies

ATTR_MD5	md5
ATTR_METADATALASTMODIFIED	getmetalastmodified
ATTR_MIN_COPIES	mincopies
ATTR_NAME	displayname
ATTR_OPT_COPIES	optcopies
ATTR_ORIGNIATOR	originator
ATTR_PARENT	parent
ATTR_SHA	sha
ATTR_SIZE	getcontentlength
ATTR_TARGET	target
ATTR_TYPE	getcontenttype
ATTR_TYPE_WAS_SET_BY	typewassetby
CREATION_FILEDATA_EVENT	101
CREATION_METADATA_EVENT	100
EV_CONTEXT_DURATION	Event/Duration
EV_CONTEXT_LEASE	Event/Lease
EV_CONTEXT_LISTENER	Event/Listener
FS_CONTEXT_ATTRIBUTELIST	ByteStore/AttributeList
FS_CONTEXT_ATTRIBUTENAME	File/AttributeName
FS_CONTEXT_ATTRIBUTES	File/Attributes
FS_CONTEXT_ATTRIBUTEVALUE	File/AttributeValue
FS_CONTEXT_CONTENT	File/Content
FS_CONTEXT_OLD_ATTRIBUTES	File/OldAttributes
FS_CONTEXT_RECURSIVE	File/Recursive
FS_CONTEXT_SERVICEID	File/ServiceID

FS_CONTEXT_SUCCESS	File/Success
FS_CONTEXT_TRANSACTION	File/Transaction
FS_CONTEXT_UUID	File/UUID
HAS_SYNCHED_EVENT	1
MDS_CONTEXT_CHANGELOG	Metadata/ChangeLog
MDS_CONTEXT_MSUIDS	Metadata/Msuids
MDS_CONTEXT_TIMEVECTOR	Metadata/TimeVector
MIMETYPE_DIRECTORY	silenus/dir
MIMETYPE_LINK	silenus/link
TYPE_SET_CONTENT	content
TYPE_SET_EXT	extension
TYPE_SET_OLDCONTENT	oldcontent
TYPE_SET_USER	user
UPDATE_FILEDATA_EVENT	201
UPDATE_METADATA_EVENT	200

COLOPHON

This thesis was edited using XMLmind XML Editor Standard Edition 3.4.0, Vi IMproved 6.2, and Aquamacs Emacs 0.9.9d. It was written using DocBook XML 4.4. It was translated to xsl-fo using a customized version of the docbook-xsl stylesheets, snapshot from 10/8/06. The translation was done using the xsltproc from libxml. The typesetting was done using fop, snapshot from Oct 06. Automation was provided by ant 1.6.5. Formulas where edited with NeoOffice 2.0 Aqua Beta 3. They where converted to SVG using JEuclid, snapshot from Oct 06. UML diagrams where created using Poseidon For UML CE 4.2.1. Additional graphics where created using Adobe Illustrator CS2 and NeoOffice Draw.