TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

DANIEL RAJALA

REDESIGNING SESSION ESTABLISHMENT IN A COMMAND AND CONTROL SYSTEM

Master’s thesis

Examiner: Professor Timo D. Hämäläinen
Examiner and subject approved at the meeting of the Faculty Council of the Faculty of Computing and Electrical Engineering 17. October 2016

ABSTRACT

DANIEL RAJALA: Redesigning session establishment in a command and control system
Tampere University of Technology
Master of Science Thesis, 48 pages
September 2016
Master's Degree Programme in Electrical Engineering
Major: Embedded Systems
Examiner: Professor Timo D. Hämäläinen

Keywords: smart card, authentication, session, Java Web Start, certificate

The aim of this thesis was to redesign the session establishment mechanism of a large command and control system. In this context session establishment refers to starting an instance of the command and control system's client application while authenticating its user to the command and control system's application server. User authentication is performed using a smart card containing the user's certificate.

The session establishment solution to be replaced was based on Java Web Start technology and a browser. A redesign of this solution was undertaken because it suffered from problems such as poor user experience, poor maintainability and complexity. Additionally, it made testing of the started application difficult and introduced a problem in which the application failed to open secure network connections using certificates stored in smart cards.

The architecture of the command and control system was explored to understand how the previous session establishment solution worked. The roles of smart cards, certificates and SSL-connections in user authentication were also identified. After gathering requirements, a new session establishment solution consisting of an authentication service, authentication client and application launcher was designed and implemented. Compared to the previous solution, it was found to achieve its targets by providing better maintainability, user experience and reliability.

TIIVISTELMÄ

DANIEL RAJALA: Istunnon avauksen uudelleen suunnittelu hajautetussa johtamisjärjestelmässä
Tampereen teknillinen yliopisto
Diplomityö, 47 sivua
Syyskuu 2016
Sähkötekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Sulautetut järjestelmät
Tarkastaja: professori Timo D. Hämäläinen

Avainsanat: toimikortti, todennus, istunto, Java Web Start, varmenne

Tämän työn tarkoitus oli suunnitella uudelleen hajautetun johtamisjärjestelmän istunnonavausmekanismit. Tässä yhteydessä istunnon avauksella tarkoitetaan johtamisjärjestelmän asiakassovelluksen käynnistämistä samalla, kun sen käyttäjä todennetaan sovelluspalvelimelle käyttäjävarmenteen sisältävää toimikorttia hyödyntäen.

Korvattava istunnonavausratkaisu perustui Java Web Start-teknologiaan ja selaimeen. Korvaustyö tehtiin, sillä tämä ratkaisu kärsi useista ongelmista, kuten huonosta käyttökokemuksesta, vaikeasta ylläpidettävyydestä ja monimutkaisuudesta. Lisäksi se teki sitä kautta käynnistettyjen sovellusten testaamisesta vaikeaa ja toi mukanaan ongelman, jossa tietyissä tilanteissa sovellus ei saanut avattua salattuja verkkoyhteyksiä käyttäen toimikorttia.

Johtamisjärjestelmän arkkitehtuuria tutkittiin edellisen istunnonavaus ratkaisun ymmärtämiseksi. Toimikorttien, varmenteiden ja SSL-yhteyksien roolit käyttäjän autentikoinnissa myös tunnistettin. Vattimusten keräämisen jäkeen suunniteltiin ja toteutettiin uusi istunnonavaus ratkaisu, joka koostui todennuspalvelusta, todennuspalvelun asiakaskomponentista sekä sovelluksesta, jolla voi käynnistää johtamisjärjestelmän eri sovelluksia. Verrattuna entiseen, uusi istunnonavauksen toteutus oli luotettavampi, ja se nähtiin tarjoavan paremman käyttökokemuksen sekä olevan helpommin ylläpidettävä.

FOREWORDS

This thesis was written at Insta DefSec, where I work as a software developer trainee.

I would like to thank my examiner, Professor Timo D. Hämäläinen for his eagerness to help and cheery disposition, and my thesis supervisor, Sc. D. Panu Hämäläinen, for his many incisive comments and suggestions. I would also like to thank my manager, Teemu Salmivesi, for suggesting the thesis topic and taking an interest in my work. Finally, I would like to thank my girlfriend Hanna-Kaisa Kemppainen for her moral support.

Tampere, 11.09.2016

Daniel Rajala

INDEX

ABBREVIATIONS AND SYMBOLS

AJP	Apache JServ Protocol, a binary protocol used to proxy requests from a web server to an application server.
API	Application Programming Interface, a collection of function definitions which define how a software system can be used by other applications.
CI	Continuous Integration, a software production practice in which developer work is frequently integrated into a shared codebase, which is automatically built and tested when changes are made.
HTTP	Hypertext Transfer Protocol, an application protocol used by the World Wide Web for communication between clients and web servers.
HTTPS	Hypertext Transfer Protocol Secure, the HTTP protocol when used over an SSL-secured connection.
IDE	Integrated Development Environment, a tool for developing software consisting of a source code editor usually coupled with integrated tools such as build tools and debuggers.
JKS	Java Key Store, a format used by the Java development platform for storing key material such as certificates.
JNLP	Java Network Launch Protocol, a protocol used to define an application which can be downloaded and run using the Java Web Start platform.
JVM	Java Virtual Machine, a virtual machine fulling the Java Virtual Machine Specification, used to run Java applications.
JWS	Java Web Start, a technology for downloading and running Java applications in a secure sandbox environment.
MVC	Model-View-Controller, a software architecture pattern in which user interface software components are divided into three parts: a model for storing data, a view for displaying data and a controller for modifying data.
PKCS	Public Key Cryptography Standards, a set of public-key cryptography standards.
PKCS#11	A PKCS standard defining an interface for using cryptographic hardware tokens such as smart cards
PKCS#12	A PKCS standard defining an archive file format for storing cryptographic objects such as private and public keys.
SSL	Secure Sockets Layer, a protocol for establishing cryptographically secure network connections using public key infrastructure.
TLS	Transport Layer Security, the successor of SSL.
URL	Universal Resource Locator, a reference to a resource on a computer network consisting mainly of an address and the protocol used to access the resource.
VM	Virtual Machine, an emulation of a computer system that can be used to run compatible software while abstracting away the platform on which the virtual machine is run.
XML	eXtensible Markup Language, a widely-used human-readable markup language.

1. INTRODUCTION

1.1 Background

Insta DefSec is a leading Finnish defence and security technology company with expertise in network-based command, control and communication systems as well as information networking and security [1]. Among their products is a distributed command and control system with client-server architecture within a single site of operations.

The product uses Java Web Start (JWS) technology to download and cryptographically verify the client application dependencies, authenticate the user through mutual authentication using a smart card and start the client application with the appropriate parameters and a secure channel of communications with the server. This process is referred to as session establishment in this thesis.

The JWS-based method for session establishment is complicated, involving a web browser, a smart card reader with its associated software, and start up scripts. Though functional, this approach has some serious disadvantages. Among these are a poor user experience and difficulty in debugging the application. The session establishment has been found difficult to maintain and develop further, since so much of the session establishment process occurs using the web browser outside the control of the application code. A persistent problem also arose where the application could intermittently lose connection to the smart card reader due to problems relating to the web browser and the smart card reader driver software, meaning that the client application had to be restarted.

For a long time, it was difficult to implement a system for reading the smart card from within the application code, because the Java 7 platform for 64-bit Windows did not include a vital module for communicating with smart card readers [2]. With the release of the Java 8 platform, the required module was included, and implementing custom code for accessing smart cards became significantly easier. This thesis is the result of the decision to redesign the product's session establishment.

1.2 Objectives

The objective of this master's thesis was to design and implement a replacement for JWS as a means of session establishment in the distributed client-server command and control system. The purpose of redesigning session establishment was to improve the potential for future development of the product and to enhance the security, maintainability and user experience of the product. This work involves investigating the role that JWS plays in the product, gathering requirements, considering alternative solutions to fulfill the requirements and implementing the replacement solution. Finally, the thesis seeks to evaluate the suitability of the implemented solution and to suggest future improvements to its session establishment.

The thesis aims to take into account different perspectives on the implemented replacement. These are the perspectives of fellow developers, who may have to work with the system at the level of source code or in testing, the operational maintenance staff, who maintain the application, and the end users of the finished product.

1.3 Structure of the Thesis

To provide the necessary context, chapter 2 seeks to give an overview of the product and the environment in which it is run. This comprises a presentation on how the product is used, its architecture as it relates to this thesis, the current session establishment solution and the various environments in which it is used. This is followed by exploration of the requirements for the new session establishment solution in chapter 3. Next, chapter 4 details the implementation of the new session establishment solution. The implementation is then evaluated for suitability from the perspectives of various stakeholders and suggestions for improvement are made in chapter 5. Finally, conclusions are drawn in chapter 6.

2. COMMAND AND CONTROL SYSTEM

2.1 Command and Control System

The product related to this thesis is a large, distributed, multi-site command and control system. Based on feeds from various external information sources, it displays and distributes a real-time model of real-world situations between command and control sites. Operators of the system can interact with the real-time model and use it for command and control activities, with user actions propagated through all sites connected in the distribution network. The system emphasizes high availability of data, real-time awareness and robustness.

The product is accessed by users through a graphical user interface client application that is run on a local computer and which communicates with an application server through remote method invocations for most actions related to the real-time model. There are a handful of different client applications. These applications are used for different purposes, but they share a common client application framework and are all related to viewing a model of real-world conditions.

The users of the product can have different privileges when interacting with it, depending on their roles. For example, a trainee may have fewer privileges than a regular operator may.

The scope of this thesis is related to the command and control system at the level of a single command and control site, usually consisting of a single application server and any number of client applications.

2.2 Architecture

The product is written in Java and uses the Java 7 version of the platform with very little use of external native components. At the very highest level, the product uses a server-client architecture, as presented in Figure 1.

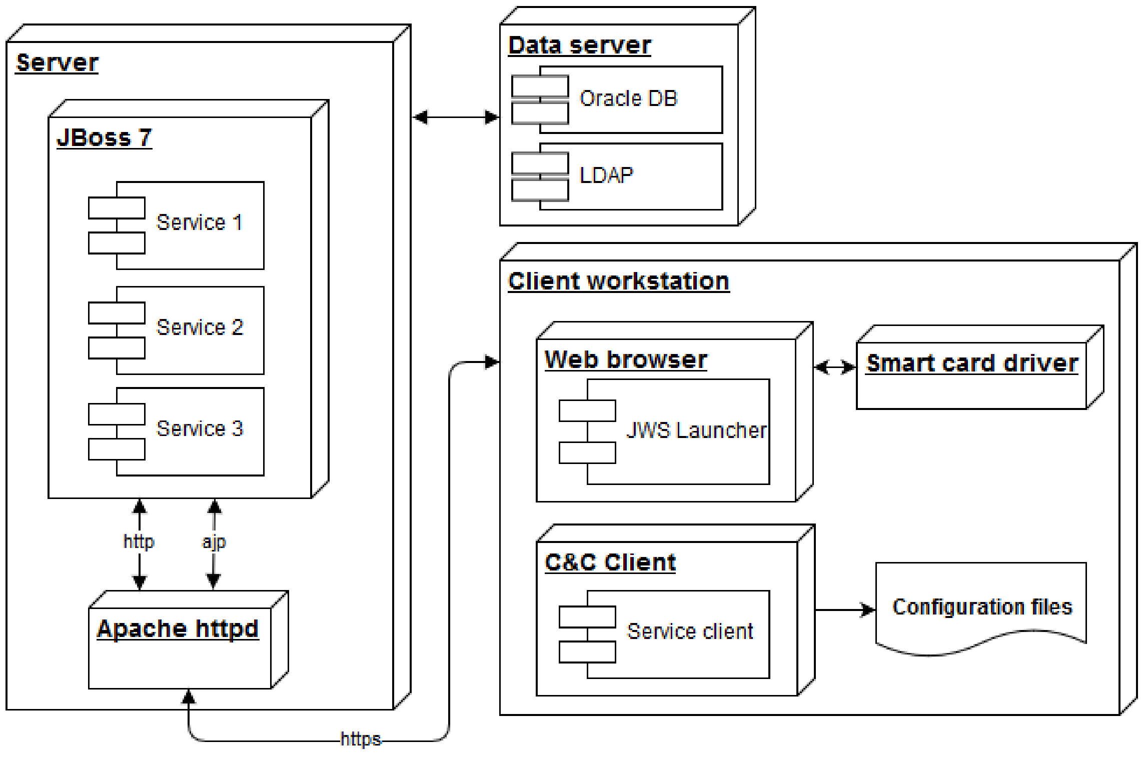

Figure 1. *Deployment diagram of an instance of the command and control system.*

Application Server

The server runs an instance of the JBoss application server. The JBoss application server hosts all of the products's server-side business logic, which includes a set of web services. Each web service is accessed through its own standard web application using Java servlets. Java servlets are small Java programs for handling HTTP requests in a web server [3]. The web container is configured in such a manner that all the web applications are accessed through the same port, with the requested web application being specified by an URL pattern. The web applications can be accessed through either the HTTP or Apache JServ Protocol (AJP) protocols. The AJP protocol is a binary protocol used to proxy requests from web servers (such as the Apache httpd web server) to application servers (such as JBoss) [4].

The web services hosted by the JBoss application server are not accessed directly by the client. Instead, they are accessed through the Apache httpd web server which acts as a reverse proxy. The purpose of the proxy is to encrypt traffic between the client and the server using a well-established and widely used SSL implementation. The proxy also serves as a centralized point for configuring access control and connection parameters.

Client Workstation

The client workstation has a smart card reader and driver software for it. The user's smart card contains a certificate which is accessed by the web browser for use in SSL client authentication.

The web browser is used to access a web page served by the application server that contains links for launching the client application. This is used by JWS, and it is discussed in more detail in chapter 2.2.3.

The client application is run on the client workstation as a Java application. The client application uses a set of configuration files that are installed manually and accessed through environment variables passed to the client application.

The data server is accessed by the application server for user details. An instance of an LDAP server provides user details such as name, role, organizational details and the user's certificate used in user authentication. An instance of an Oracle database is used to store information on which application permissions belong to which roles. The server uses this information together to authenticate users and authorize access to various services.

2.2.1 Security

SSL

Secure Sockets Layer (SSL) is a technology for establishing cryptographically secure connections using certificates. TLS (Transport Layer Security) the successor of SSL with the same purpose. For historical reasons, 'SSL' is often used to refer to either TLS or SSL [5]. In this thesis 'SSL' is used when referring to TLS to be consistent with the naming of various Java platform classes and methods that perform TLS related operations.

SSL encrypts communications using symmetric cryptography. In symmetric cryptography, the communicating parties possess a shared secret, namely a key used to encrypt and decrypt communications [6]. The shared key is created upon the establishment of the connection in a process called the SSL handshake. The SSL handshake uses public-key cryptography to identify the parties of the connection and to generate the shared secret [5].

Public-key cryptography is a form of cryptography based on pairs of keys called the public and private keys. As its name suggests, only its owner knows the private key and it is never distributed to others, whilst the public key can be distributed freely to anyone. This form of cryptography works in such a way that data encrypted using a public key can only be decrypted using the corresponding private key and vice versa. Using these principles, it is possible to send messages encrypted with the recipient's public key that only the intended recipient can decrypt using their private key. This technique can also be used to prove a party's identity. [7]

SSL connections use certificates to implement the asymmetric cryptography. A certificate consists of a public key, details such as the name of the certificate, information about the certificate owner’s organization and various other pieces of information such as used cryptographic algorithms. A certificate also contains the name of the certificate issuer, which is the entity that created the certificate. The issuer is represented by its own certificate, and each certificate contains a cryptographic signature which can be used to verify that it is indeed issued by the specified issuer. [7, pp. 215-216]

The linking of certificates to each other via the certificate issuer forms a certificate chain, which ultimately leads to a root certificate [7, p. 217]. The advantage of the certificate chain is that instead of having to keep track of each trusted certificate individually, an SSL agent can store a handful of trusted root certificates and trust certificates issued by them.

During a SSL handshake using mutual authentication, the following certificate-related steps are performed:

1. The server presents the client its certificate chain
2. The client verifies the server certificate chain
3. The client presents the server its certificate chain
4. The server verifies the client certificate chain

[5]

The product makes use of SSL for connections between the client application and application server.

Smart Cards

Smart cards are typically plastic cards with an integrated microprocessor with a broad range of uses, including personal identification [8, p. 893]. The product uses smart cards to authenticate the user of the client application to the application server.

The smart card contains the user certificate and the certificate’s private key. Certificates stored on the smart card are freely readable and require no PIN number to access. The private key, however, is designed at the hardware level to be inaccessible by the machine reading the smart card. Any operations involving the private key, for example encrypting or decrypting data with it, are performed by the smart card hardware itself, and require user authorization using a PIN number. [9]

Smart cards are read using dedicated smart card reader hardware. The smart card reader hardware has its own drivers, which in turn are used by applications to interact with the smart card. The smart card reader software used in the product provides an API library implementing the PKCS#11 standard for interacting with cryptographic tokens such as smart cards [10]. Figure 2 shows a high level overview of how a client application accesses smart cards using the PKCS#11 API.

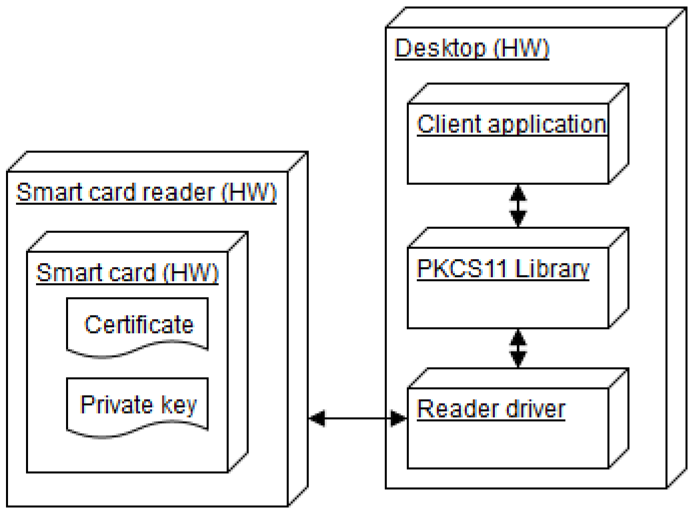

Figure 2. *Hardware and software used in interfacing with a smart card.*

The 64-bit Windows version of the Java 8 platform includes a module for interacting with PKCS#11 libraries. This module abstracts away most of the details of PKCS#11, providing a convenient interface for accessing the smart card as a key store.

Confidentiality

The communications between the client application and the application server are sensitive, meaning that their confidentiality must be ensured. This is achieved using secure SSL connections between the client application and the application server.

The SSL connections are made to an Apache httpd server that forwards incoming requests to the application server's web container that does not use SSL. The Apache server is configured to use SSL, but in principle the user client could attempt to make connections straight to the application server's unsecured web container thus circumventing the confidentiality provided by SSL. This is prevented by a firewall configuration that prevents connections to the application server's web container from the outside.

In the development environment, where the client application and application server are run on the same machine, confidentiality is not necessary. In this case the application server can be connected to directly without the Apache httpd server providing SSL.

User Authentication

Users of the product must be authenticated. Upon starting the client application, a mutually authenticated SSL connection is formed to the application server. The client certificate used in the connection is read from a smart card, requiring that the user prove ownership of the card by entering the correct PIN number. The application server uses the client certificate to resolve the user's details using the LDAP and database servers, after which the user details are used to form a session that is then stored in a disk-persistent cache. A session token is created that can later be used to access the cached session information.

The session token is provided to the client in the response to a successful mutually authenticated request. When establishing new SSL connections to the application server, the client may forgo mutual authentication by attaching the session token to the header of each request, as the application server can use it to fetch previously authenticated session details from the cache. Since the session token is used to identify a user's session, it must be protected by making sure it is never sent over unsecured connections or otherwise leaked.

User Authorization

The product allows for control over which features are available to a user. For example, it may be necessary to restrict what kind of information a user can view or manipulate. This is implemented using a permission system, consisting of users, roles and permissions. Roles consist of a set of permissions and the users who belong to the role. For example, a training role with few permissions can be created for someone learning to use the product.

The application server is configured to use the Spring Security framework for web security. The framework is based on 'security filters', which are a form of middleware between receiving the request and servicing it in the application logic [11]. This is illustrated in Figure 3.

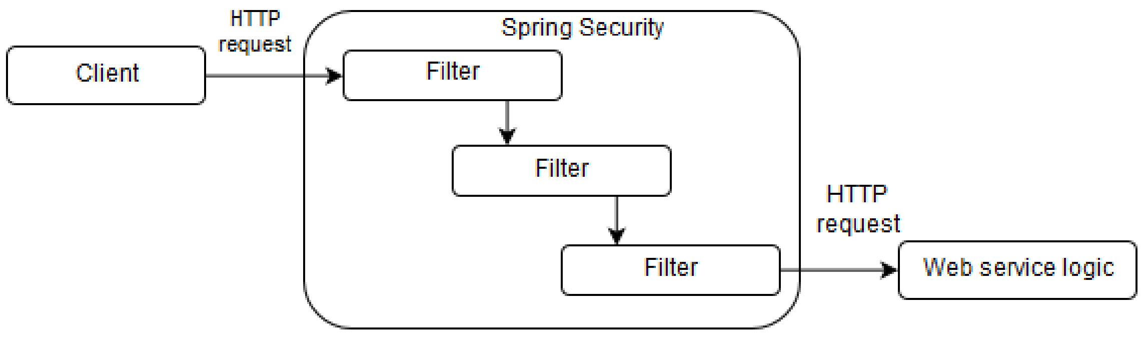

Figure 3. *Requests pass through Spring Security filters before being handled by the web service.*

The purpose of filters is to perform tasks necessary before the request is processed by business logic, for example ensuring user authentication and authorization. Figure 4 summarizes how the product performs authentication and authorization.

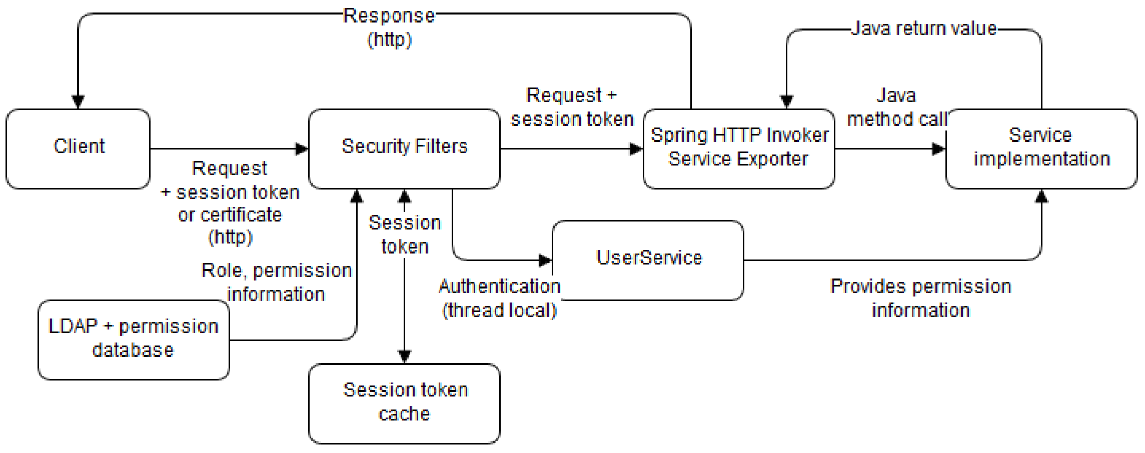

Figure 4. *Authentication and authentication in the command and control system.*

The Spring security filters combined with the product's own customizations extract the user certificate from the request and determine the user's roles and permissions. This information is fetched from LDAP and database servers. If a certificate is not provided, but a valid session token is presented, the session token is used to fetch the session details that were cached when the session token was created. Once the session information has been found, a thread local security context containing details about the user such as their username and their permissions is created, which services can use to make authorization decisions.

Services can use a separate user service, which provides methods for checking if the user is authorized to perform an action. The user service accesses the thread local Authentication object to find the user permissions. In case a request is not authorized, it fails, and the Spring framework's security filters return the appropriate HTTP status code to the caller.

2.2.2 Remote Method Invocation

The product uses Spring framework's HTTP invoker technology for remote method invocation. Spring's HTTP invoker technology is used to expose Java objects over HTTP using Java's native serialization features. It has the advantage of requiring little configuration and providing Java web services as regular method invocations on plain Java objects. [12]

Client

The client-side implementation of Spring's HTTP invoker comprises a *HttpInvokerProxyFactoryBean* class which is responsible for creating a service proxy as seen in Figure 5. This class is configured by setting the URL of the server-side service and the interface to be implemented by the proxy it creates.

The proxy object is generated dynamically using reflection and it uses Java serialization to serialize the method calls along with the method parameters when calling the remote service. It uses the same serialization mechanisms to deserialize the results of the remote method call returned from the service. To the user, the proxy object appears as a regular Java object that implements the given interface.

Since the method calls are made over HTTP, the Spring HTTP invoker framework allows developers to control the way the HTTP request is executed by creating a custom class implementing the *HttpInvokerRequestExecutor* interface, as seen in Figure 5. This can be used to add custom headers to the request, or change the way the connection is established. For example, the product uses a customized implementation of this interface to attach previously discussed session tokens to the requests' header.

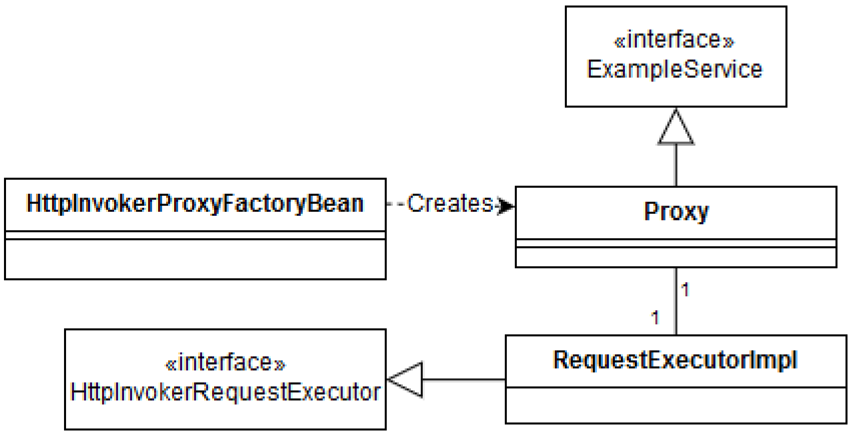
«interface» ExampleService
HttpInvokerProxyFactoryBean
Creates
Proxy
1
1
«interface» HttpInvokerRequestExecutor
RequestExecutorImpl

Figure 5. *Implementation of a client-side Spring Remoting proxy.*

Server

The server-side implementation of Spring's HTTP invoker is based on the *HttpInvokerServiceExporter* class. This class implements the Java servlet framework's *RequestHandler* interface, meaning that it can be configured to handle requests to a given URL, in this case the remote service URL. The relationships between these classes are show in Figure 6. The service exporter deserializes the incoming requests into Java method invocations, which are then applied to the provided service implementation object. The return values or exceptions provided by the service implementation are then serialized and sent to the client as an HTTP response. Only requests which make it through the previously discussed security filters are serviced by these request handlers.

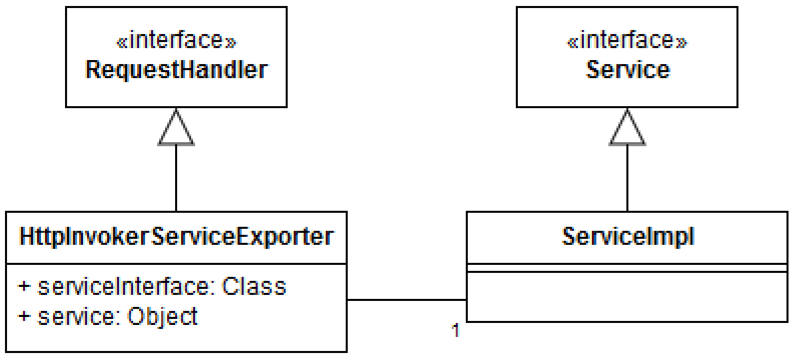

Figure 6. *Server-side implementation of a Spring Remoting web service.*

2.2.3 Java Web Start

The product uses the JWS technology for session establishment. JWS is a technology created in 2001 by Sun Microsystems, the developers of the Java platform at the time. Its purpose is to allow the easy deployment of Java applications from the web browser. Among the advantages of the technology are the automatic downloading of the latest version of the application, ease of use, and a secure sandbox for running the downloaded application. [13]

JWS applications are defined using the Java Network Launching Protocol (JNLP). JNLP uses the XML format to specify aspects of the application such as command line arguments and application dependencies and their locations. [14] The web browser associates JNLP content with the JWS executable, meaning that clicking on a JNLP link will start the given application. An example JNLP file is presented in Program listing 1.

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp
      spec="1.0+"
      codebase="https://example.net/ExampleApplication/webstart">
      <information>
            <title>Example Application</title>
            <vendor>Example Company</vendor>
            <icon href="icon.png" kind="default"/>
      </information>
      <offline-allowed>false</offline-allowed>
      <security>
            <all-permissions/>
      </security>
      <resources>
        <j2se version="1.6+" />
        <jar href="Example.jar" />
    </resources>
      <application-desc main-class="net.company.example.ExampleApp">
            <argument>--arg1</argument>
            <argument>--arg2</argument>
      </application-desc>
</jnlp>
```

Program listing 1. *An example JNLP file defining a simple JWS application.*

The example JNLP is a simple definition of an application which can be started using the JWS framework. Among the most important things specified are the required resources, the URL from which they can be downloaded, the permissions the application requires and the program arguments.

By default, JWS applications run in a sandbox with restricted privileges, for example without access to the file system. However, if the application and its dependencies are signed using a trusted certificate, these privileges can be granted to the application. [15]

2.2.4 Session Establishment Process

The product's application server hosts a web page presenting the different application that can be started. The web browser is configured to use mutual authentication using a certificate found on the user's smart card, so when the user starts an application, a mutually authenticated request to the web start service is made. If the mutual authentication is successful, the product's security architecture generates a session token identifying the user as discussed previously. The web start service generates a JNLP description of the requested application, and inserts the session token into the JNLP file as a command line parameter for the client application that is being started. The JNLP file is automatically executed by the JWS framework, which downloads all the dependencies specified in the JNLP from the application server. Once the dependencies are downloaded and JWS verifies their integrity, the application is started.

The session token created when fetching the JNLP description is used to authenticate and authorize all future remote method invocations. This is done by a custom implementation of the *HttpInvokerRequestExecutor* interface described in chapter 2.2.2, which attaches the session token to the headers of all remote method invocation requests.

From the above it can be seen that JWS' roles in session establishment are the following:

- Authentication of the user and fetching a session token
- Gathering application dependencies (Java libraries)
- Verifying the integrity of the downloaded dependencies

To fulfill these roles, the product's codebase contains much code and configuration related to JWS. Among the components related to JWS are:

- A build plugin which generates a list of client application dependencies along with their versions for JWS
- A web application serving the web page containing the links for the client application JNLP file
- A web application for dynamically generating the JNLP file for the client application so that it starts with the right session token
- A custom Java Servlet for downloading JWS dependencies
- Build configurations for signing all the dependencies of the JWS client application, along with the signing certificates
- Classes that generate and inject a session token into the JNLP file of the client application
- Classes for parsing JWS-related command line arguments in the client application
- Apache httpd reverse proxy configurations for the JWS web application

The amount of code and configuration indicate the complexity of the JWS-based session establishment.

2.3 Environment

2.3.1 Development Environment

The development environment is the environment in which the developers work on the product, for example by developing new features. It is in itself a secure environment with no requirements for user authentication, permissions or encrypted communications (unless a developer is working on them). The main concerns for developers when using the system is ease and speed of use, because development often entails restarting the application very often. For this reason, the development environment is simplified, with the application server and desktop client running on the same development machine without an Apache reverse proxy, and security features such as user authentication and authorization disabled.

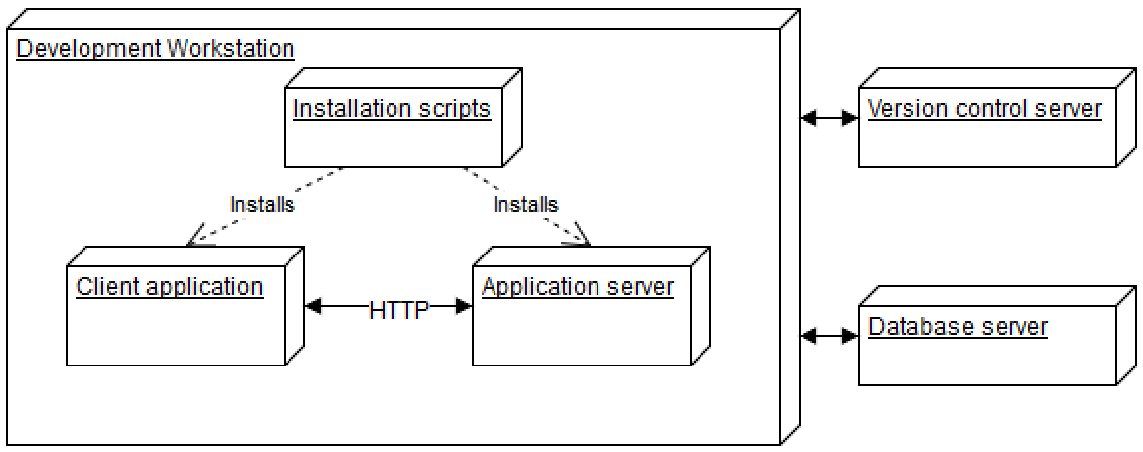

Figure 7. *The product and its installation in the development environment.*

As seen in Figure 7, installing a new version of the product in the development environment consists of simply running a script included with version control. The application's configuration is also included in version control, with environment variables pointing to the configuration files. This is considerably simpler than in the other environments, which can require extensive customization.

JWS is not used to start the client application in the development environment. Instead, the application client is started directly in the IDE for speed of use and easy debugging.

2.3.2 Production Environment

The production environment is where the product is put to its intended use by the end users. Only finished, stable versions of the product get installed in the production environment by the customer. Security, robustness and usability are very important in the production environment.

The infrastructure of the production environment is shown in Figure 8.

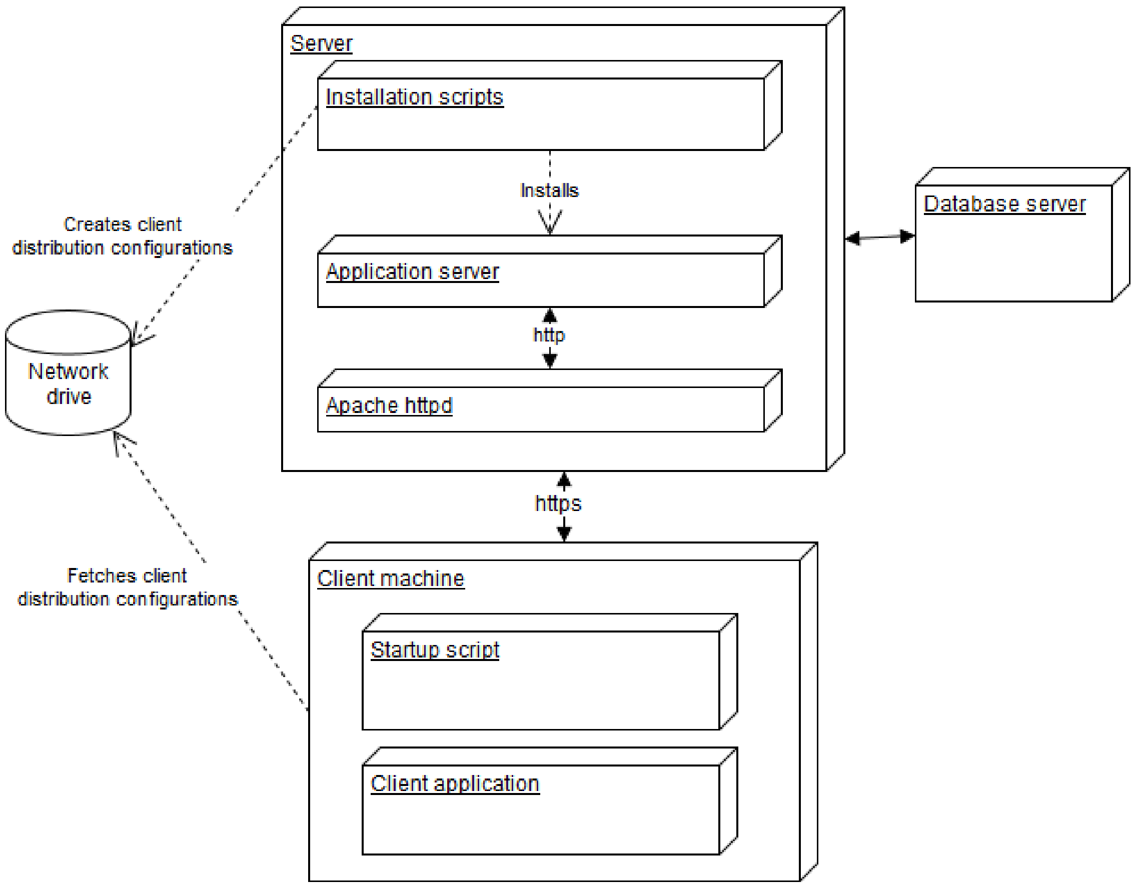

Figure 8. *Infrastructure of the product in the production environment.*

The production environment consists of separate machines for running the client application, the application server and the database server. It also has a network drive which can be accessed by both the application server and client machine, used by the client machine to fetch configurations.

New versions of the product are provided to the customer on a disk, that an administrator installs using a command line script. During installation, the user can set the values for a broad range of parameters, that are then injected into the various configuration files. The installation script creates a compressed package of configuration files for the desktop client, which it then places on a network drive for the client machine to access. To install a new version of the product on the client machine, the user fetches the configuration from the network drive and extracts into a defined location, which the client application can access through an environment variable.

The product has a requirement for supporting several ‘operational environments’. An operational environment is an independent application server configured for a given purpose, with its own data sources and configurations. The user is able to start the client application such that it uses an operating environment of their choosing. The main purpose of operating environments is to separate the use into different domains, such as training and operational use. Instead of connecting to a live command and control environment in which actions can have real-world consequences, a trainee can connect to an operating environment specifically created for training.

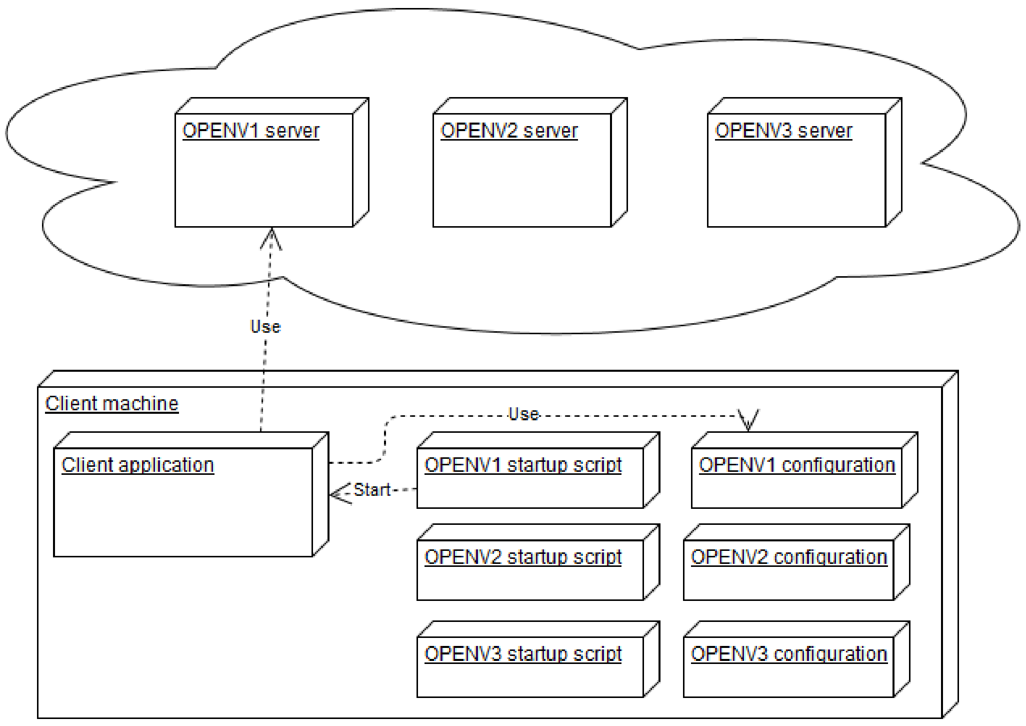

Figure 9. *Operating environments consisting of client application configurations and application servers.*

As shown in Figure 9, operating environments are implemented by having several application servers running on their own virtual machines. The user chooses the operating environment to use by running an operating environment specific client application startup script, which sets environment variables such as the host name and port numbers to point to the given application server.

Each operating environment has its own set of client application configurations that are independent of one another. This means that a client application started in OPENV1 uses only configuration files for OPENV1. The configuration files of the operating environments are stored in their own directories in a common distribution directory. Each operating environment configuration directory contains a file containing the name of the operating environment to which it belongs.

The operating environment is chosen by starting the application with the appropriate startup script. Each operating environment has its own batch script that sets a series of environment variables and copies files used by the client application to the appropriate locations before starting the web browser. For example, the script sets an environment variable with the IP address of the operating environment's application server and copies custom application settings to the client application's application data directory. The scripts are generated when the application server is installed.

To run the product in the production environment, the application server has to be started. Once it is running, a startup script is executed on the client machine. The operating environment is chosen by starting the application with the appropriate startup script. Each operating environment has its own startup script that sets a series of environment variables and copies files used by the client application to the appropriate location. For example, the script sets an environment variable with the IP address of the operating environment's application server and copies custom application settings from a network drive to the client application's application data directory on the local disk. Finally, the script starts the web browser and directs it to a startup page hosted by the application server serving links to each client application's JNLP file.

2.3.3 Test Environment

The test environment is where incremental versions of the product are tested. The test environment attempts to match the production environment closely so that problems likely to occur in the production environment are more likely to be detected. This environment is used mainly by full-time testers, who do testing of a broader scope and over a longer time than developers, though developers often use test environment to test features or defects they are working on. To support testing and diagnosis of defects, the test environment has various tools for interacting with the product.

As can be seen from Figure 10, the test environment features a very similar infrastructure to the previously discussed production environment.

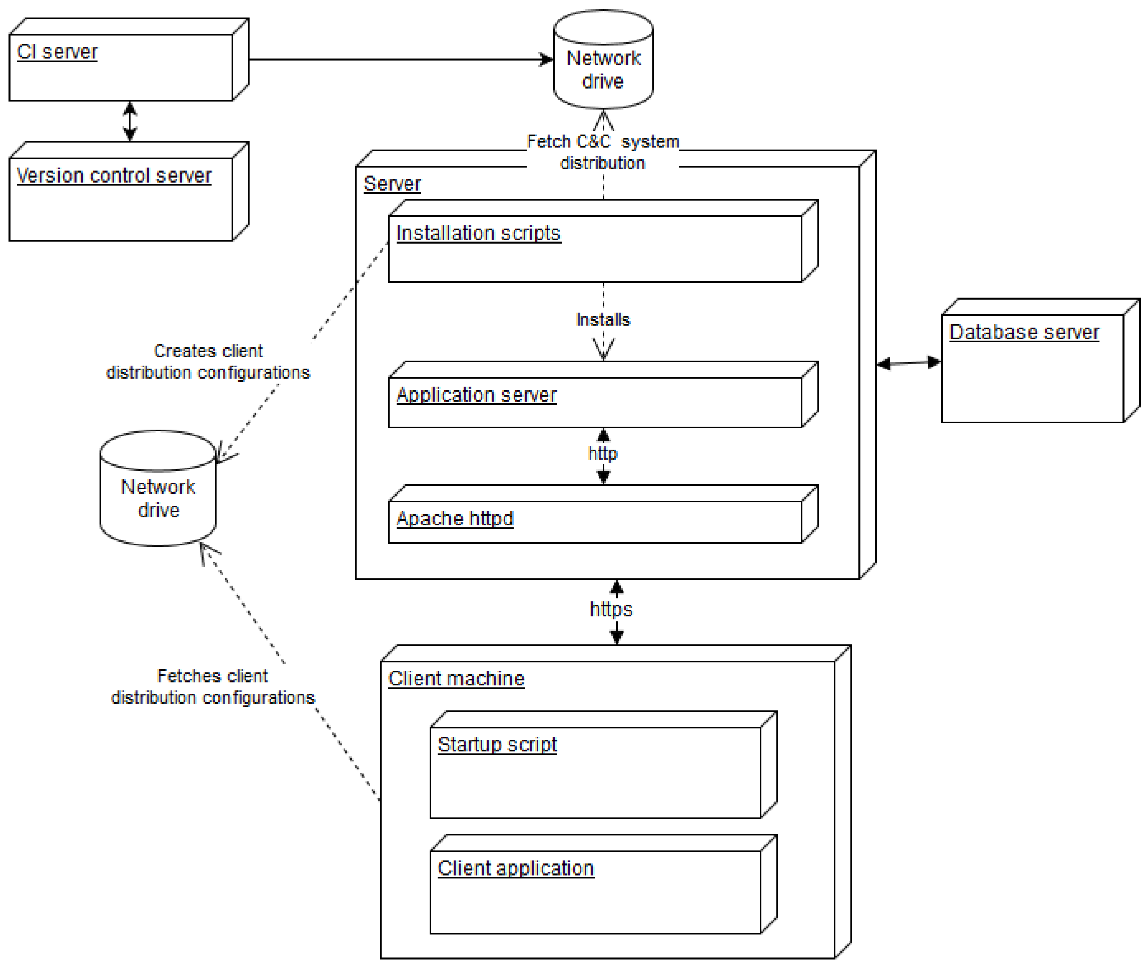

Figure 10. *Infrastructure of the product in the test environment.*

The main difference in infrastructure between the production and test environment is the inclusion of a continuous integration (CI) server. The CI server can be commanded to build a version of the product from a given branch in version control. The CI server produces a compressed package of the given version which is then placed on a network drive. The installation of the package is very similar to that of the production environment, with the same kind of installation scripts installing the application server and exporting client application configurations to a network drive.

Starting client applications is done similarly to in the production environment, except that smart cards are not used. Instead, file-based certificates are used by the browser when fetching the client application JNLP files. This is done for the sake of ease of use and to ensure that the product works the same for different testers on the same machine. The test environment additionally supports different operating environments in the same way as the production environment.

3. REQUIREMENTS FOR SESSION ESTABLISHMENT

The purpose of this chapter is to detail the high-level requirements of a system to replace JWS. This is done by first identifying the broad components of the replacement system. Next, more detailed requirements related to each component are introduced.

3.1 Need to Redesign Session Establishment

There are several reasons that led to the decision to redesign session establishment in the product. Using the JWS technology to establish sessions suffers from many disadvantages from the perspectives of both the users and the developers.

The primary goal for the redesign of session establishment is an improved user experience. The current user experience in which the user has to run one of many startup scripts for the given operational environment is considered messy, and the browser interface for starting applications is disliked. Instead, something like a Java application providing a simple, centralized way of starting client applications in the right operating environment would be valuable to end users.

Developers benefit from a session establishment redesign in many ways. An immediate concern is the difficulty of debugging and testing the client application. Since the client application is run in its own JWS container instead of being run as a regular Java application, it is almost impossible to do debugging in the test environment. Debugging in the test environment is often crucial, since there are notable differences between the development and test environments. For example, using the Java VisualVM tool to monitor memory and processor use is nearly impossible. Even small changes to the client application code require a great deal of time and effort. This is because the client application dependencies are fetched from the application server and are signed, meaning that a small change to the client application code requires an entire re-installation of the application server, a process that can take an hour. It would be preferable to be able to change individual client application libraries without requiring a reinstallation.

Redesigning session establishment would also help solve a long-standing problem that users have noticed with the JWS-based session establishment, which causes client applications to become inoperable. Although the exact cause of the problem is uncertain, it relates to how the client application attempts to re-establish connections to the server after the smart card used in mutual authentication is removed for a while. JWS installs custom instances of classes for handling SSL connections which make attempts to read from the smart card. If there is a problem reading the smart card, remote method invocations fail. By redesigning the session establishment mechanism for greater control over reading smart cards, this problem can be eliminated.

A more general concern with using JWS is that it leaves the developers with little control. The future development of the application is tied to the JWS platform and the decisions made by its maintainers. The JWS platform perceived by the organization as buggy, poorly maintained, and little used by the industry, meaning that developers have to deal with bugs that they have no way to fix. A session establishment solution produced in-house would provide developers with greater flexibility and control over the future direction of how the client application is managed and started.

3.2 Starting the Client Application

One of the principal tasks of JWS is to allow the user to start client applications. The product achieves this using a web project that hosts links to the correct JNLP files used to launch the application. Additionally, the application must be started such that it uses the correct operating environment.

This means that the web project hosting the JNLP links must be removed and a new way of starting client applications in each of their modes must be created. The new way of starting client applications must also be capable of the same logic as the client application start scripts that are used to select the correct operating environment.

It is required that the new way of starting client applications take the form of a 'launcher' application with a clear user interface. The launcher application should display all the applications that can be started and provide means of starting them. Additionally, the launcher must be able to able to support web links, so that when the user clicks on the link in the launcher application, the web browser is opened to the given web page.

To replace the scripts needed to start the client application in the right operating environment, the new user interface must support selecting the desired operating environment. Ideally, it should be capable of detecting the available operating environments on its own instead of having them configured.

The launcher application should be configurable, so that no changes in code are necessary to modify which applications can be started. The way in which applications are started should also be configurable, with the possibility to edit the command line arguments of the application to be started.

The only technical limitation is that the launcher application must be implemented in the Java programming language. This is to keep the launcher application in line with the rest of the product.

3.3 Dependency Management and Installation

The JWS technology fetches the client application dependencies from the application server by downloading them over HTTP upon startup. The dependencies are resolved at compile time by a build plugin and injected into a JNLP template.

To create a new session establishment solution, an alternative way of resolving the client application dependencies must be created. Mechanisms must also be implemented to deliver the dependencies to the desktop machine, which involves refactoring the product's installation process.

There are no explicit requirements as how this should be achieved, but using a network drive to provide the dependencies is preferred, as it is simple solution when using existing network drives. Using an existing network drive would also make it easy to integrate dependency management with the installation process.

3.4 User Authentication

The user is authenticated to the application server when they access the JNLP file of the application over a mutually authenticated SSL connection. This creates a session token which is dynamically injected into the JNLP and passed to the client application as a command line parameter. This means that once JWS is removed, a new way of authenticating the user and fetching a session token must be developed.

It is required that the authentication process still use the user's smart card for authentication. It is furthermore required that a component be developed which can use the smart card reader for authentication from within Java code, a task which was previously done by the browser. This smart card reading functionality must be based on the PKCS#11 standard for interacting with cryptographic hardware tokens. If possible, the smart card reading component should be an independent component, reusable for possible future smart card reading needs in the company's other projects.

Since typically users do not use their smart cards in the test environment, the user authentication mechanism should be able to support file-based certificates for authentication. For convenience, the implementation should revert to smart card authentication if file-based authentication is not possible. Authentication and SSL connections are very rarely used in the development environment, meaning that the implementation should be able to forego authentication entirely in the development environment. This should be done in such a way that authentication cannot be disabled by the user in the production environments, in which authentication is required.

An additional requirement is that the client application, once started by an authenticated user, should be able to continue operating in case the smart card is removed from the reader. It should also continue operating seamlessly once the smart card is replaced in the reader.

3.5 Dependency Integrity

The JWS framework requires that application dependencies be cryptographically signed using a trusted certificate to ensure their integrity. The effects on security from removing dependency signing were considered, and it was not deemed necessary to sign dependencies in the system that replaces JWS. This is because the product's installation process is tightly controlled by administrators in a secure environment in a closed network, with no access from the outside.

4. DESIGN AND IMPLEMENTATION

4.1 Launcher

4.1.1 Starting Java Applications

To build a launcher application for starting the products's client applications, it is useful to understand how the Java virtual machine is used to start Java applications. The most important aspects in starting a Java application are the classpath, the main class, the Virtual Machine arguments, program arguments and environment variables.

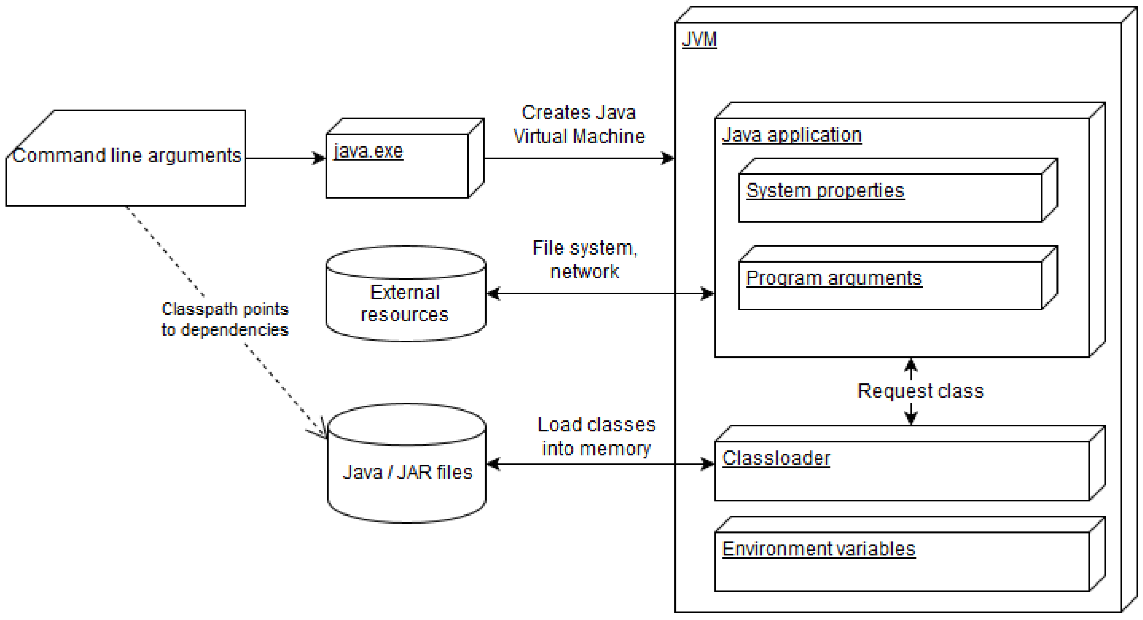

Figure 11. *Starting a Java application in a Java Virtual Machine.*

Classpath

As can be seen in Figure 11, the Java Virtual Machine (JVM) contains a component called a classloader, the task of which is to load compiled Java class files into memory so that they can be executed. The classloader looks for classes from the classpath. The classpath refers to the file system path where the application dependencies reside. If a class used by the application is not found on the classpath, the application cannot run. The classpath can consist of more than one file path.

There are three kinds of classpath for the JVM. These are the bootstrap classpath, the extension classpath and the user classpath. The bootstrap classpath is the classpath to the default Java runtime libraries and it can be used to override classes in the underlying

Java platform. The extension classpath is used to point to classes that are used to extend the Java platform using the Java extension mechanism. Neither the bootstrap nor the extension classpath are used in the command and control application. The most consequential classpath type is the user classpath, which is used to locate the classes of the application to be started. [16]

The classpath is provided to the JVM as a command line parameter. The classpath can consist of any number of file system paths, meaning that the classes used by the application do not have to be placed in the same file system location. The classloader also searches for compiled Java classes in located inside .jar and .zip compressed archives placed on the classpath.

Main Class

The main class is the class containing a method that serves as the starting point of the application execution. When starting a Java application, the name of the main class is provided as a command line argument. The JVM searches for a class of this name, and starts executing its main method if found.

The main method is a method with the following method signature.

```
public static void main(String[] args)
```

The *args* parameter is an array of command line arguments provided to the Java program.

Virtual Machine Arguments

The Virtual Machine arguments are command line arguments that affect the JVM in which the application is run. They are typically low-level, affecting things such as the heap size, garbage collection, etc. JVM arguments can also be used to set system properties for the Java application to be started. System properties are key-value pairs similar to environment variables that are accessible to the Java application. This makes system properties a convenient way to provide certain parameters to the Java application.

Program Arguments

Program arguments are command line arguments provided to the main method of the main class of the application that is being started. As the name suggests, they are designed solely for the Java application and not the JVM. The program arguments can take any form.

Example

Below is an example of a typical command line command for starting a Java application.

```
java –Xmx1024M –Djavax.net.debug=ssl –cp /lib/network.jar;/lib/gui.jar
net.company.product.Application –-host=https://company.net/application/
```

The command starts a JVM running the main method of the class `net.company.product.Application`. The JVM arguments specify a maximum heap size of 1024 MB and a system property called `javax.net.debug` with the value `ssl`. The classpath points to two .jar files containing the required classes. The program is provided with a command line argument `--host=https://company.net/application/`, which the program will need to parse itself.

Environment Variables

The JVM in which the Java application is run inherits environment variables like any other process. Since the command and control application client is run on a Microsoft Windows operating system, this means that the JVM child process inherits its parent process' environment variables [17]. Similarly, if a new Java process is created by the Java application, the new process will inherit the environment variables of the Java process that spawned it.

The Java application can access the environment variables in a similar manner to system properties, but changing the environment variables through this mechanism does not affect the underlying environment variables of the operating system process in which the Java virtual machine is run. It is however possible to create new Java processes from within Java code with custom environment variables.

4.1.2 Launcher Configuration

Chapter 2.3.2 details the concept of an 'operating environments' within a single client desktop machine. An operating environment consists of its own application server and client application configuration. Since each operating environment may need to define its own list of applications to be started with a custom command line parameters and environment variables, it is natural to include an independent launcher configuration with each operating environment instead of creating a monolithic configuration common to all operating environments.

Instead of creating a launcher application for each operating environment, it is required that a single launcher application be able to support all operating environments. This means that the launcher will have to be able to find the launcher configurations included with each operating environment. This requires that the launcher application have its own configuration containing information about where the operating environments are installed and how their launcher configurations can be found, as in Figure 12.

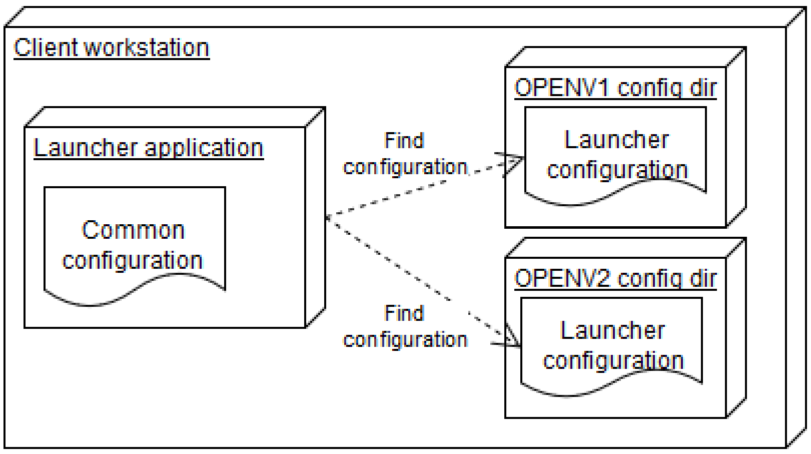

Figure 12. *A common configuration is used by the launcher application to find operating environment specific launcher configurations.*

The details of the two types of configuration (common configuration and the operating environment specific launcher configuration) are discussed separately below.

Given the complexity of the types of configuration items described below, an obvious technology for implementing the configurations is XML. The advantages of using XML are the ability to model complex data, human readability and ease of editing.

Common configuration

The 'common configuration' is used by the launcher application to find operating environment specific configurations. Operating environments are installed as their own directories under a common distribution directory, each of which includes a file containing the name of the operating environment. The operating environment specific launcher configurations are placed in these configuration directories. Given these facts, the common configuration should contain the following configuration items:

- Path to the directory containing the operating environment configuration directories
- Path to the launcher configuration file from the operating environment's directory
- Name of the file identifying the operating environment

These configuration items are sufficient for the launcher application logic to find all the operating environments and their launcher configurations. An example of this configuration is given in Program listing 2.

```
<?xml version=”1.0” encoding=”UTF-8” ?>
<launcher-common-config>
  <distribution-dir-path>C:/Applications/</distribution-dir-path>
  <openv-filename>operating-environment.txt</openv-filename>
  <launch-config-path>config/launcher.xml</launch-config-path>
</launcher-common-config>
```

Program listing 2. *An example common launcher configuration.*

Operating Environment Specific Configuration

The launcher application uses the operating environment specific launcher configuration to find which applications can be launched and which web links can be opened. Since the launcher application replaces the startup scripts that copy files from the client configuration directory into the local application data folder as described in chapter 2.3.2, the launcher must be capable of this file copying. Given the requirements of the launcher application, the operating environment specific configuration must contain the following configuration items:

- The name of the operating environment
- A list of applications that can be started, their names, descriptions and the command line arguments to start them
- For each application, the filenames to which the stated application’s logs are directed
- A list of environment variables used by the command and control client application
- A list of web links that can be opened in a web browser along with their names and descriptions
- A list of files to copy when an application is started, including the source and destination paths

The following is an example of an operating environment specific launcher configuration with a single application and a single link.

```
<?xml version="1.0" encoding="UTF-8" ?>
<launcher>
  <copy-files>
    <copy-file>
      <overwrite>false</overwrite>
      <source>C:/Applications/example/config.xml</overwrite>
      <source>%LOCALAPPADATA%/App/config.xml</overwrite>
    </copy-file>
  </copy-files>
  <java-launch-configs>
    <java-launch-config>
      <name>Example application</name>
      <description>Example description</description>
      <java-params>
        <java-executable>java.exe</java-executable>
        <vm-args>
          <arg>--Djavax.net.ssl.trustStore=C:/certs/trustStore.jks</arg>
          <arg>--Xmx1024M</arg>
        </vm-args>
        <program-args>
          <arg>--host=example.company.net</arg>
          <arg>--port=80</arg>
          <arg>--mode=normal</arg>
        </program-args>
        <classpaths>
          <classpath>C:/Applications/example/lib/*</classpath>
          <classpath>C:/3rdPartyComponent/lib/3rdpartyComponent.jar</classpath>
        </classpaths>
        <main-class>net.company.example.Application</main-class>
      </java-params>
      <envvars>
        <envvar>
          <name>CONFIG_DIR</name>
          <value>C:/Application/config</value>
        </envvar>
      </envvars>
      <output>
        <merge-stderr>false</merge-stderr>
        <stderr-path>application_error.log</stderr-path>
        <stdout-path>application_stdout.log</stdout-path>
      </output>
    </java-launch-config>
  </java-launch-configs>

  <links>
    <link>
      <name>App homepage</name>
      <description>Example app home page</description>
      <url>https://example.company.com/appliation</url>
    </link>
  </links>
</launcher>
```

Program listing 3. *An example operating environment specific configuration.*

4.1.3 User Interface

The user interface should clearly present the different applications that can be started and links that can be opened for a given operating environment. The solution is to show a list of labelled buttons for each application and link. The user interface should also allow the user to select which operating environment to use, such that changing the operating environment changes the available buttons. This is most easily implemented using a combo box. A mockup of a user interface satisfying this functionality is shown in Figure 13.

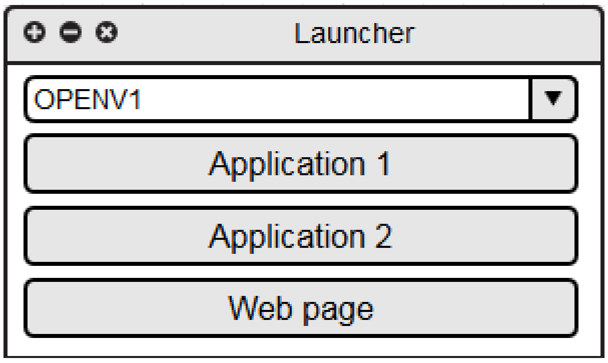

Figure 13. *A mockup user interface for the launcher application with the ability to open web links and start Java applications in the selected operating environment.*

It should be noted that the mockup is intended to represent the basic functionality of the application and the actions that the user can perform on it. As such, it does not reflect the launcher application in its final polished form.

4.1.4 Implementation

As a user interface application with the user interacts with information contained in configuration files, the high-level architecture for the launcher application is the Model-View-Controller (MVC) architecture. In this architecture, the model represents some kind of data, the view represents the user interface that displays the data and passes user input to the controller, and the controller implements the business logic that modifies the data. The advantages of using the MVC architecture are the clear separation of concerns that make the system easier to modify and maintain [18]. The ability to change the user interface is particularly important in this case, because the final appearance of the user interface is undecided and will be changed by other developers later.

Designing the application top-down using the MVC architecture led to the interfaces shown in Figure 14.

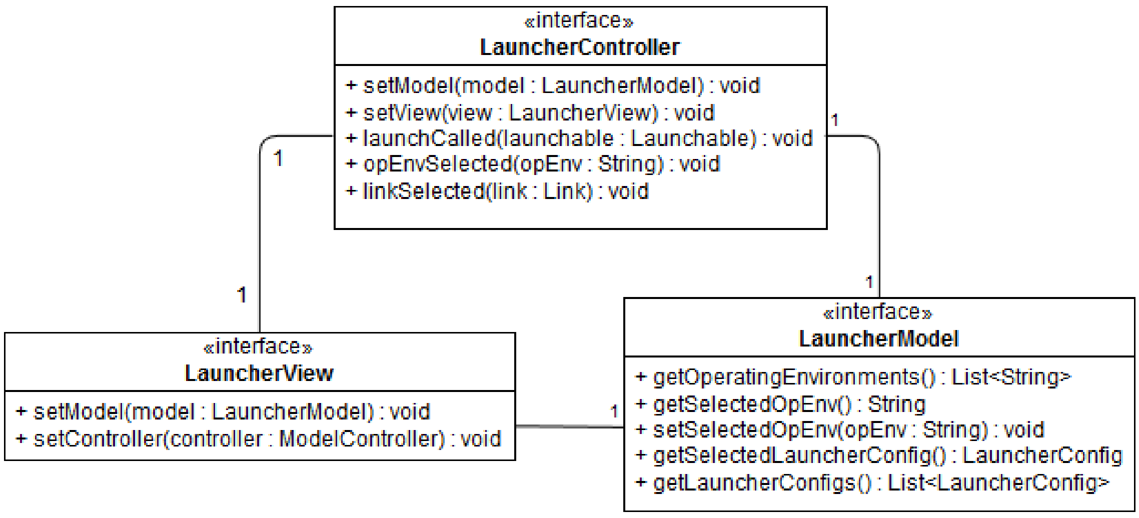
«interface»
LauncherController
+ setModel(model : LauncherModel) : void
+ setView(view : LauncherView) : void
+ launchCalled(launchable : Launchable) : void
+ opEnvSelected(opEnv : String) : void
+ linkSelected(link : Link) : void
1
1
1
1
1
1
«interface»
LauncherView
+ setModel(model : LauncherModel) : void
+ setController(controller : ModelController) : void
«interface»
LauncherModel
+ getOperatingEnvironments() : List<String>
+ getSelectedOpEnv() : String
+ setSelectedOpEnv(opEnv : String) : void
+ getSelectedLauncherConfig() : LauncherConfig
+ getLauncherConfigs() : List<LauncherConfig>

Figure 14. *The model, view and controller components of the launcher application's MVC architecture.*

When the model is set for the view, it displays the applications, links and operating environments contained in the model. It informs the controller when the operating environment is changed, when an application is selected to be launched or when a link is clicked on. It fetches the applications and links from an instance of *LauncherConfig*, a class which will be introduced later.

The controller is informed by the view of which links are opened and which applications are launched. The controller then decides how to launch an application or open a link. The controller is also informed when the user changes the operating environment. In this case the controller edits the model by changing the currently selected operating environment. Once the model is changed, it calls *setModel* on the view, updating the displayed links and applications to those defined in the new operating environment.

The three new interfaces introduced in this MVC model are *Launchable*, *Link* and *LauncherConfig*, show in Figure 15.

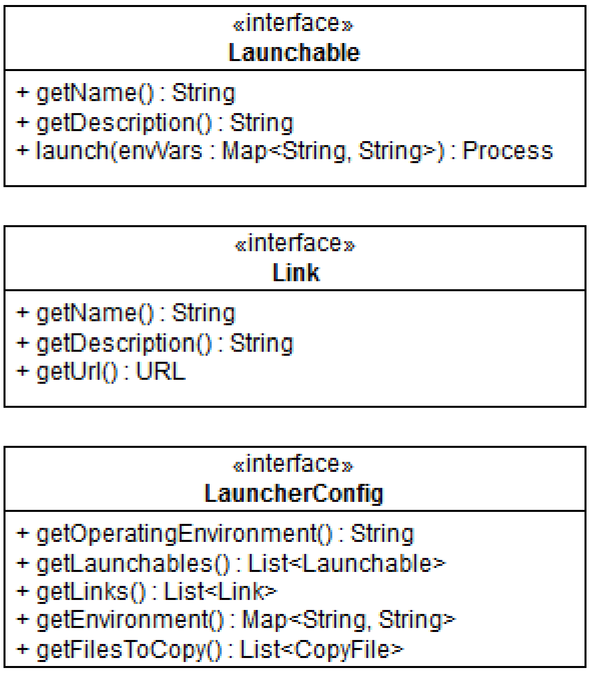
«interface»
Launchable
+ getName() : String
+ getDescription() : String
+ launch(envVars : Map<String, String>) : Process
«interface»
Link
+ getName() : String
+ getDescription() : String
+ getUrl() : URL
«interface»
LauncherConfig
+ getOperatingEnvironment() : String
+ getLaunchables() : List<Launchable>
+ getLinks() : List<Link>
+ getEnvironment() : Map<String, String>
+ getFilesToCopy() : List<CopyFile>

Figure 15. *Interfaces used in the launcher defining launchable applications.*

The *Launchable* interface encapsulates an application that can be started as its own process by calling the *launch* method. The *launch* method takes a map of environment variables that are inserted into the launched application's process. This is necessary because there are environment variables that are specific for a given operating environment and inheriting environment variables from the launcher application's process is not sufficient. The launch method returns a handle on the started process, which can be used e.g. to check if the launched application is still running. The *getName* method is used for displaying the application's name in the user interface and the *getDescription* method can be used e.g. for tooltips when hovering the mouse over the application's button in the user interface.

The *Link* interface defines a link that when clicked on opens the web browser to a given URL. The *getName* and *getDescription* methods serve the same purpose as they do in the *Launchable* interface. The *getUrl* method is used to fetch the URL of the link.

The *LauncherConfig* interface encapsulates the launcher configuration for a given operating environment. It has methods for fetching the name of the operating environment, the list of applications that can be started, the list of links that can be opened, the environment variables for the operating environment and the details on which files need to be copied before the applications can be started. The implementation of the *LauncherConfig* interface parses an XML operating environment specific launcher configuration to find these details. As such, there are as many different *LauncherConfig* instances as there are operating environments.

These interfaces form the skeleton of the launcher application, although few important utility interfaces are needed. These interfaces are shown in Figure 16.

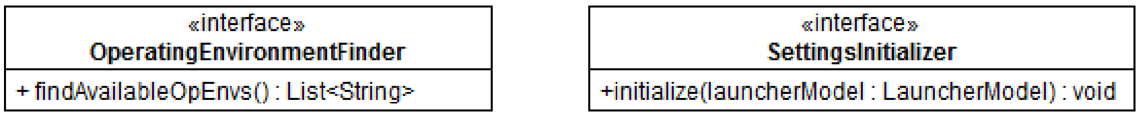
«interface»
OperatingEnvironmentFinder
+ findAvailableOpEnvs() : List<String>
«interface»
SettingsInitializer
+initialize(launcherModel : LauncherModel) : void

Figure 16. *Interfaces used to find opearating environments and to initialize the launcher settings.*

The *OperatingEnvironmentFinder* interface is used to find the available operating environments. The default implementation uses the previously discussed common launcher configuration to search for operating environments based on the configured file paths. This interface is used by the implementation of *LauncherModel*.

The *SettingsInitializer* interface is used to perform the logic of the startup scripts which the launcher replaces. This includes copying the necessary files. *SettingsInitializer* is used when *LauncherModel* is initialized upon starting the launcher application.

Together, the interfaces described above form the high-level architecture for the launcher application shown in Figure 17.

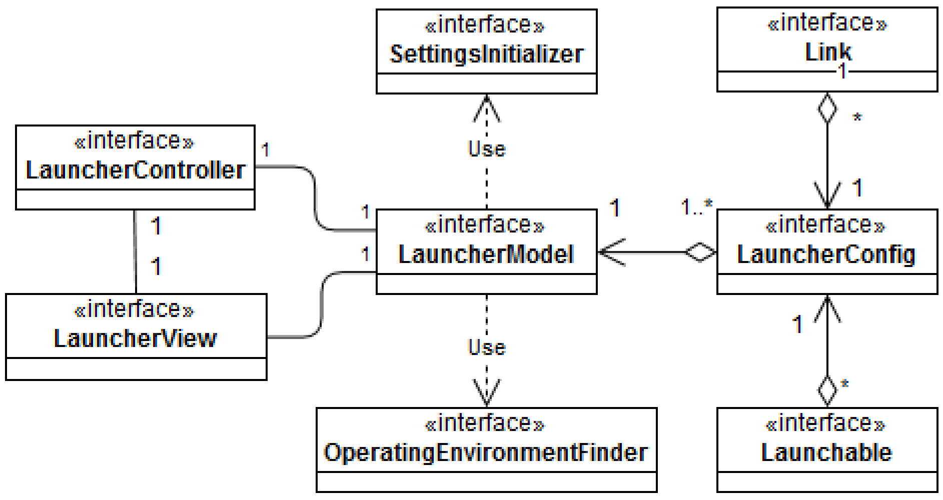

Figure 17. *High level architecture of the launcher application.*

Once the interfaces were decided upon, implementing them was straightforward. There were two considerable choices of technologies. The first choice was the user interface technology, for which Java's Swing user interface framework was chosen. The second technology choice was Apache XMLBeans for parsing XML configuration files. XMLBeans is a technology which generates code based on an XML schema, resulting native Java types which bind to XML data [19]. This makes using XML configuration files very convenient.

4.1.5 Installation

Once the launcher application was created, it had to be made available to the test and production environments. The launcher application is not used in the development environment, so it was not taken into account.

Since the launcher application was designed to be used as a standalone Java application, it was decided that it should be a runnable Java .jar archive. The runnable Java archive is like a regular Java archive except that it contains additional metadata about how the archive should be run and all the external class dependencies that are needed to run the application. Using a runnable Java archive confers the advantage of ease of use, since in the Windows operating system they can be run simply by double clicking on them like a regular executable.

The command line application needs to read the common configuration file in order work. Since the configuration needed to be user editable, it could not be packed inside the runnable Java archive. This created the problem of how to tell the launcher application the path to the common configuration without having to pass it as a command line parameter. Pointing to the common configuration using an environment variable was considered. However, this was rejected because of administrative work required to create such an environment variable. Eventually it was decided that the configuration be distributed alongside the runnable Java archive. The launcher application was programmed to find its own file path and access the common configuration relative to this path. This resulted in the directory structure show in Figure 18.

```
Launcher
|-- common-config
|   `-- common-config.xml
`-- launcher.jar
```

Figure 18. *Directory structure of the launcher installation directory.*

It was decided that the launcher application be distributed as a compressed .zip archive because of the simplicity of installation by extraction anywhere on the client machine.

4.2 User Authentication

4.2.1 Approach

In principle, it is not necessary to keep the product’s security architecture, in which remote method invocations are authenticated using a session token fetched after successful SSL mutual authentication. For example, it could be possible to mutually authenticate all connections to the application server independently. However, it was considered easier to implement the new authentication system on top of the existing security architecture using session tokens to authenticate remote method invocations. In addition, the reusing session tokens results in better performance than if the smart card were used in establishing every SSL connection. Using the session token also reduces the risk of failed remote method invocations due to previously mentioned problems reading the client certificate from a smart card.

It was decided that authentication system should work similarly to the previous implementation, with an initial mutually authenticated SSL connection used to authenticate the user and fetch a session token to be used in future remote method invocations. This leads to the idea of an independent authentication web service from which the client application can fetch session tokens upon successful user authentication using a mutually authenticated SSL connection. The authentication service would require an authentication client component in the product's client application. This approach results in the structure presented in Figure 19.

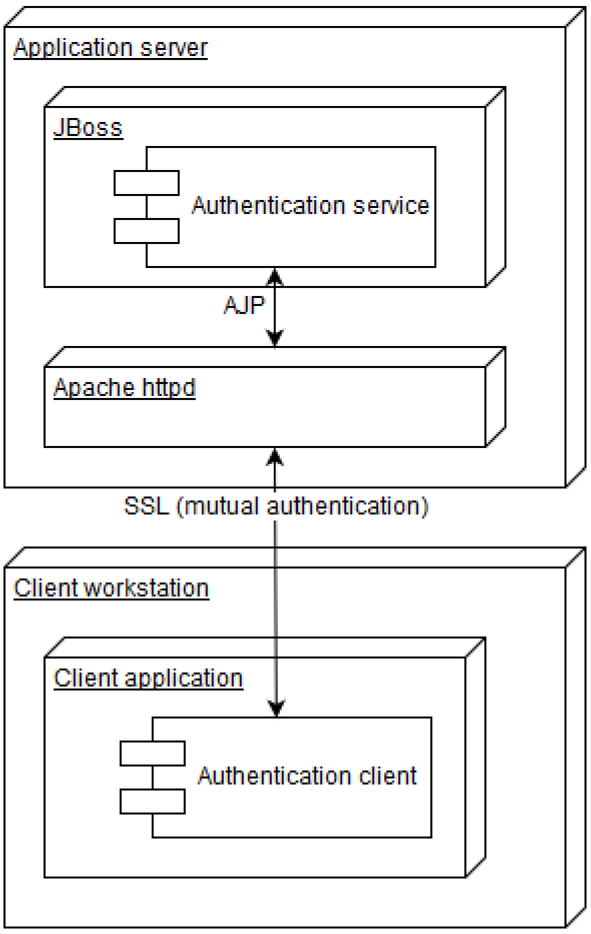

Figure 19. *Communication between the authentication client and authentication service.*

Since the product requires user authentication through smart cards, the authentication client implementation requires components with the ability to interact with the user smart card. To make use in the test environment simpler, it should also be able to support file-based certificates in place of the smart card.

4.2.2 Authentication Service

The functioning of the authentication service is similar to what came before. The user makes a mutually authenticated HTTPS request to the authentication service. The product's security architecture resolves user details based on the certificate presented in the request and creates a session token, which the authentication service returns as a response to the client. This is illustrated in Figure 20.

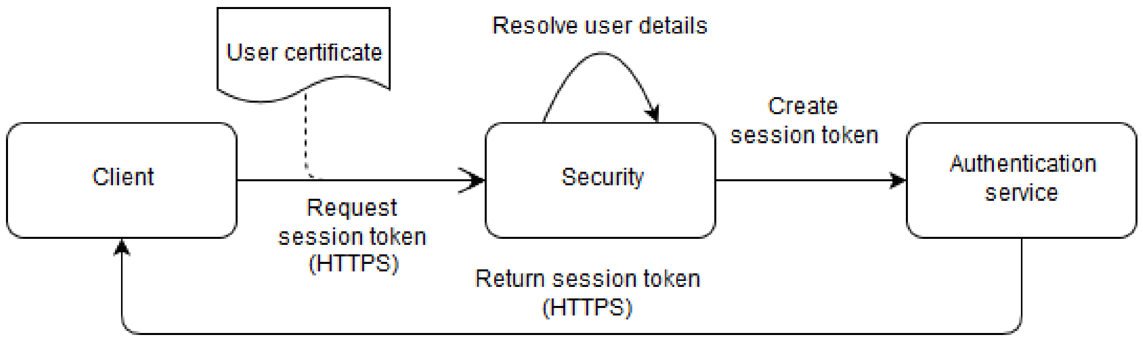

Figure 20. *Fetching of a session token from the authentication service.*

Like all other web services in the product, the authentication service is implemented using the remote method invocation architecture described in chapter 2.2.2.

Since the application server's security filter middleware performs user authentication, the authentication service itself is very simple, leading to the authentication service interface shown in Figure 21.

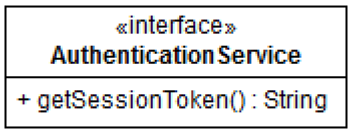

Figure 21. *The authentication service interface.*

The interface has a single method for fetching the session token, which only succeeds if the user making the request can authenticate themselves with their certificate used in the SSL handshake.

The authentication service implementation has to obtain the session token created by the security filters. Since the scope of the session token is the request, there are two primary ways for the service implementation to obtain the session token. The most obvious way is to use the session token stored as thread local variable of the thread servicing the web request. However, it was found that this thread local variable is only stored whilst the security filters handle the request, and removed once execution progressed into the web service business logic. The second way is to obtain the session token from the request object, as the session token is added to the request as a header by the security filters.

The Spring HTTP Invoker remote method invocation technology does not let the class implementing the web service see the HTTP request that is being serviced. To let the authentication service access the session token contained in the request header, a custom request handler was developed, which lets the service implementation see the request for accessing the token. The solution is presented in Figure 22.

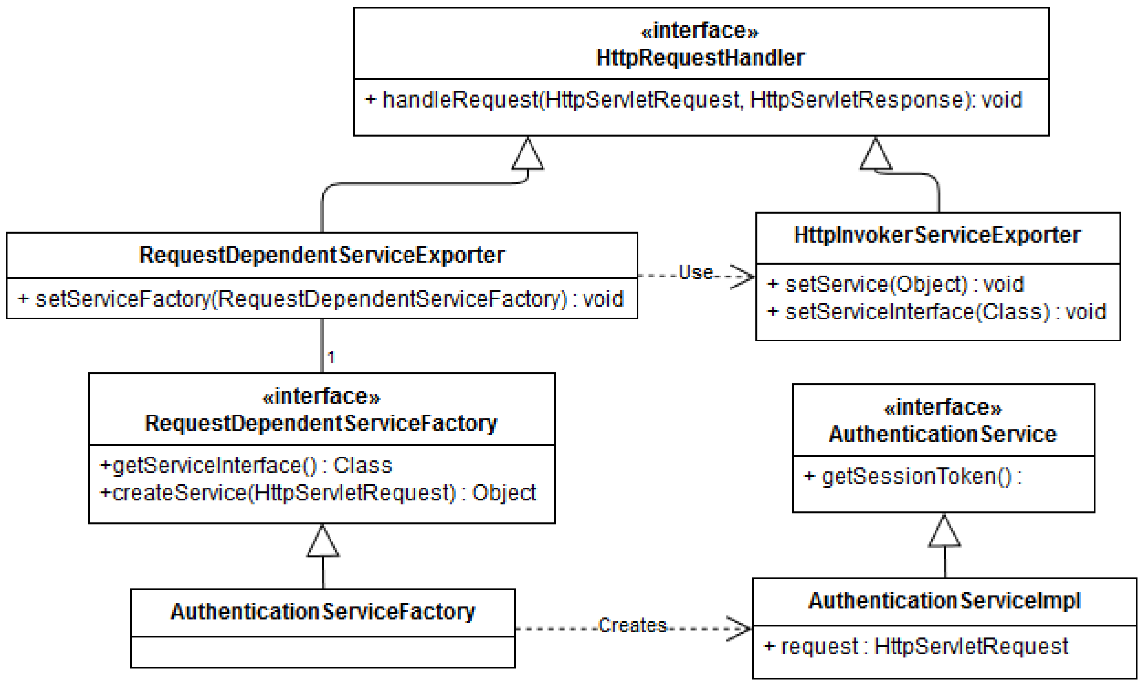
«interface»
HttpRequestHandler
+ handleRequest(HttpServletRequest, HttpServletResponse): void
RequestDependentServiceExporter
+ setServiceFactory(RequestDependentServiceFactory) : void
Use
HttpInvokerServiceExporter
+ setService(Object) : void
+ setServiceInterface(Class) : void
1
«interface»
RequestDependentServiceFactory
+getServiceInterface() : Class
+createService(HttpServletRequest) : Object
«interface»
AuthenticationService
+ getSessionToken() :
AuthenticationServiceFactory
Creates
AuthenticationServiceImpl
+ request : HttpServletRequest

Figure 22. *Server-side implementation of the authentication service.*

As detailed in chapter 2.2.2, an instance of Spring HTTP Invoker framework's *HttpInvokerServiceExporter*, which implements the *HttpRequestHandler* interface, handles the HTTP request by calling methods on the service implementation provided to it. Since this does not allow the service implementation to see the request, a new implementation of the *HttpRequestHandler* interface was made: *RequestDependentServiceExporter*. This class uses an instance of *RequestDependentServiceFactory* to create the service implementation. The service implementation can then access the request object using the method parameter of the *createService* method. Once the service implementation is instantiated, it is provided to the *HttpInvokerServiceExporter* instance, which finishes handling the request by invoking methods on the newly created service instance. The outline of this process is shown in Figure 23.

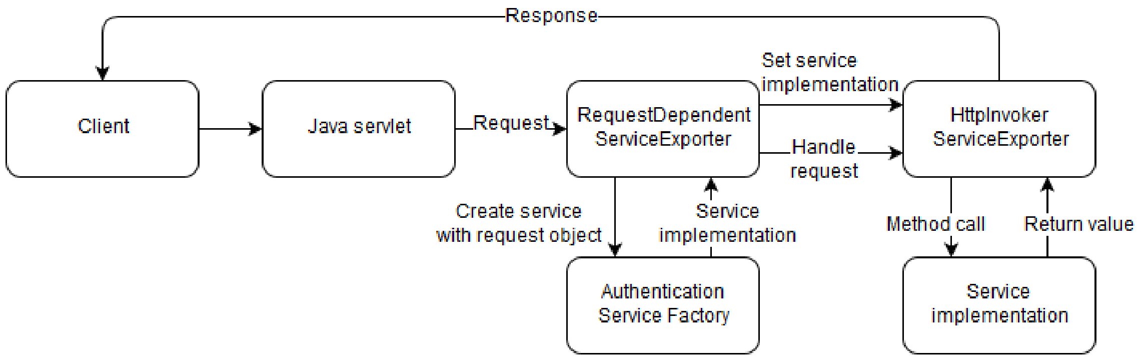
Response
Client
Java servlet
Request
RequestDependent ServiceExporter
Set service implementation
Handle request
HttpInvoker ServiceExporter
Create service with request object
Service implementation
Method call
Return value
Authentication Service Factory
Service implementation

Figure 23. *Handling of a client request to the authentication service.*

Apart from the changes described above, the authentication service implementation uses the same architecture for web services as described in chapter 2.2.1.

The authentication service requires mutual authentication to be accessed. Since the product uses an Apache httpd server as a reverse proxy for SSL connections, a new virtual host was created for handling the authentication service requests. The virtual host was configured in such a way that all connections to the authentication service require client authentication. At the same time the virtual host for other services was configured to make mutual authentication optional, allowing connections using the session token to forego mutual authentication. The creation of a new virtual host exclusively for the authentication service means that the authentication service is accessible through a different port than other services.

4.2.3 Authentication Client

The authentication client is a component of the product's client application that uses the authentication web service to fetch a session token to be used for further remote method invocations. It has to be able to perform mutual authentication using certificates on a smart card to identify the user. It must also support file-based certificates in lieu of smart cards in the test environment. This means that the following things have to be designed and implemented as part of the authentication web service client:

- A typical Spring HTTP Invoker client proxy
- Classes for establishing mutually authenticated SSL connections using a configuration
- A configuration for specifying what kind of key material to use (smart card/file)

Proxy

As explained in chapter 2.2.2, the Spring *HttpInvokerProxyFactoryBean* class creates a proxy implementing the *AuthenticationService* interface, such that method calls to the proxy are invoked on the server implementation of the authentication service. This is shown in Figure 24.

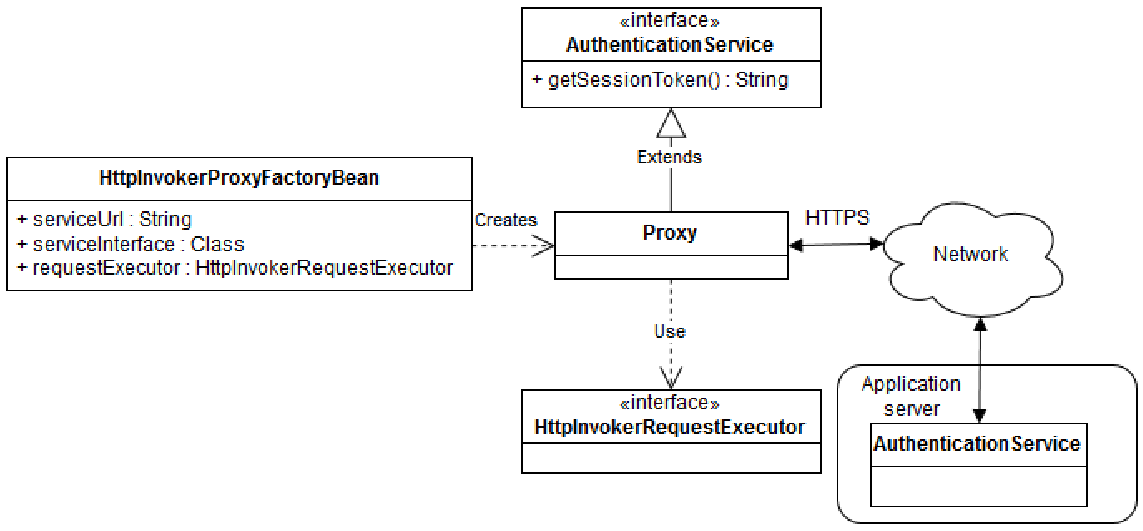
«interface»
AuthenticationService
+ getSessionToken() : String
Extends
HttpInvokerProxyFactoryBean
+ serviceUrl : String
+ serviceInterface : Class
+ requestExecutor : HttpInvokerRequestExecutor
Creates
Proxy
HTTPS
Network
Use
«interface»
HttpInvokerRequestExecutor
Application server
AuthenticationService

Figure 24. *Creation of a proxy capable of using the authentication service.*

The requests to the application server are executed by an instance of the *HttpInvokerRequestExecutor* interface described in chapter 2.2.2. An implementation of this interface must be created which uses mutual authentication in SSL connections using the configured key material.

HTTPS Connections in Java

Typically, Java applications use an instance of the *HttpsURLConnection* class to create HTTPS connections. This class in turn uses an instance of the *SSLSocketFactory* class to create SSL sockets used in the HTTPS connection. *SSLSocketFactory* instances rely on instances of the *TrustManager* and *KeyManager* classes to handle the trust stores and key stores respectively.

Trust stores and key stores are containers of key material such as certificates, public keys and private keys. The trust store contains trusted certificates and it is used by the Java SSL implementation to check if the server certificate is trusted. The key store contains the client certificate sent to the server during an SSL connection and the user's private key used for cryptographic operations. Key stores and trust stores are password protected, which means that modifying them or accessing sensitive information within them such as private keys requires a valid password.

Both key stores and trust stores are implemented using the *KeyStore* class. *KeyStore* instances can be configured to use different sources for their key material using the Java Security Provider framework. For example, a smart card can be specified as a *KeyStore* using the PKCS#11 provider included with the Java platform for interfacing with cryptographic hardware. More typically, the key material is provided using file-based key stores. The most common file-based key store type is the Java Key Store (JKS). JKS is a file format for storing key material. PKCS#12 archives can also be used as key stores using the PKCS#12 Provider. PKCS#12 is a standard for storing several cryptographic objects in a single archive, for example used in packaging a public key and private key.

Figure 25 demonstrates how the classes discussed above relate to each other.

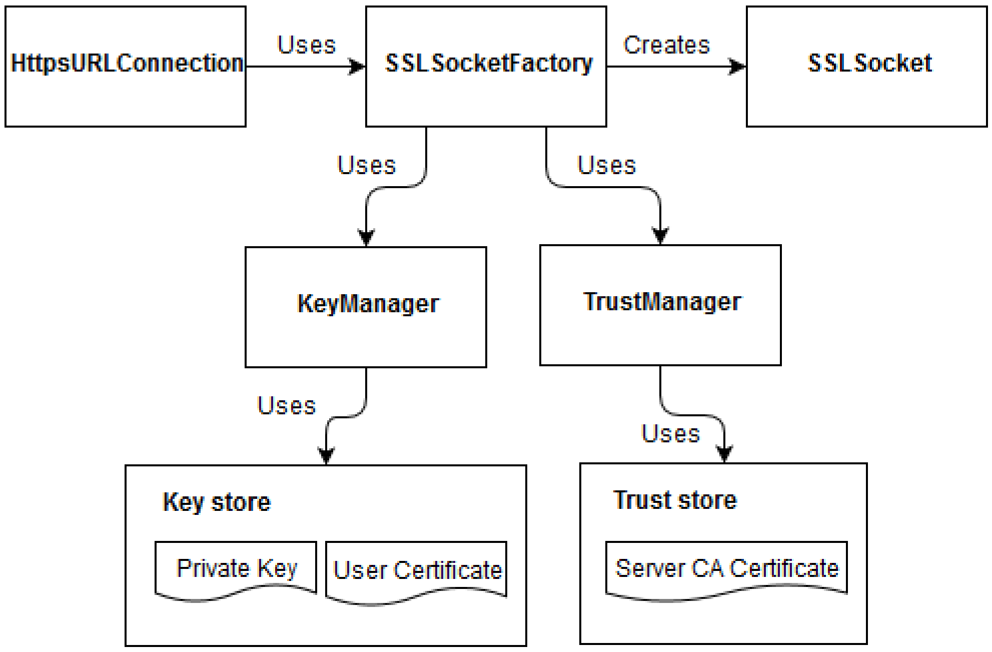

Figure 25. *The principal components used when forming an HTTPS connection from Java code and their relations.*

To be able to create custom connections that can use a smart card or file as the key store, it is necessary to create a set of classes that, based on a given configuration, create the necessary key stores, trust stores, key managers, trust managers, and SSL socket factories.

Mutual Authentication

First, a high-level design was made for the way mutually authenticated connections could be established from the authentication service proxy to the application server. The outline of the design is show in Figure 26.

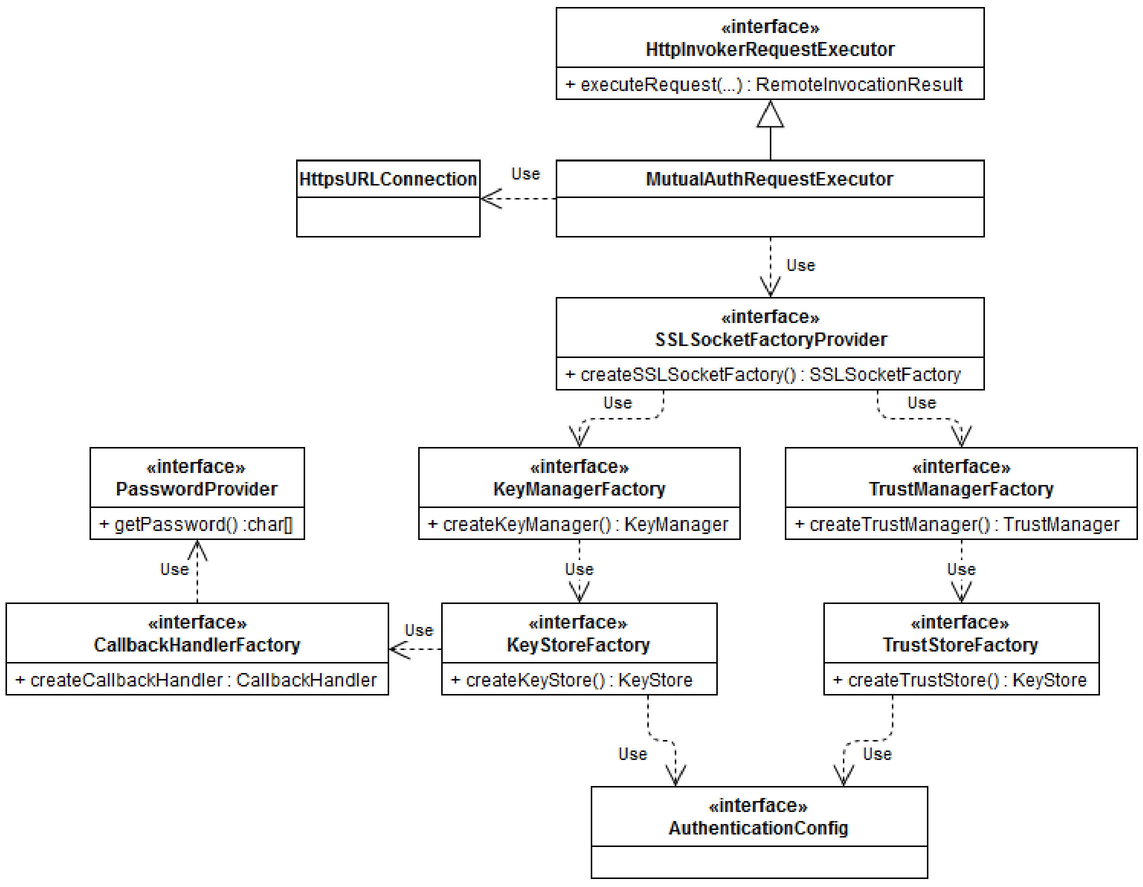

Figure 26. *High-level design for making mutually authentication connections using a custom authentication configuration.*

In this design, when a request is made to the server, the *MutualAuthRequestExecutor* class uses an instance of the previously discussed *HttpsURLConnection* class, for which it sets an *SSLSocketFactory* that it obtains from the *SSLSocketFactoryProvider*. The *SSLSocketFactoryProvider* creates *SSLSocketFactory* instances that use key managers and trust managers created by the *KeyManagerFactory* and *TrustManagerFactory* classes, which in turn obtain their key store and trust store from instances of *KeyStoreFactory* and *TrustStoreFactory*. These last two factories use an authentication configuration to decide what key store and trust store to use. This way, either the smart card or a file can be configured as the key store for the connection.

The trust store can also be configured in a similar manner. The *KeyStoreFactory* uses *an instance of CallbackHandlerFactory* to create callback handlers for providing key store passwords. When accessing a private key in a key store, a password has to be provided. The *KeyStore* implementation invokes registered *CallbackHandlers* requesting the password, which is fetched from an instance of *PasswordProvider*, which can e.g. request a password from the user via a graphical user interface.

In the development environment, the application server's security features are disabled and SSL is not used, as connections are established over HTTP directly to the application server. To support this kind of use in the development environment, the MutualAuthRequestExecutor class only attempts to perform mutual authentication when the used protocol is HTTPS. In case of HTTP the request is made over HTTP, with no certificate sent.

For the most part, the implementation of the high-level design described above was straightforward. However, it was that when deciding which user certificate to send to the server, the default Java implementation of the *KeyManager* interface would choose the first certificate in a *KeyStore*, leading to situations in which the wrong certificate was used from a smart card containing more than one certificate. To ensure that the correct certificate is always sent to the server, a custom *KeyManager* was created. This new mechanism for choosing the correct certificate is presented in Figure 27:

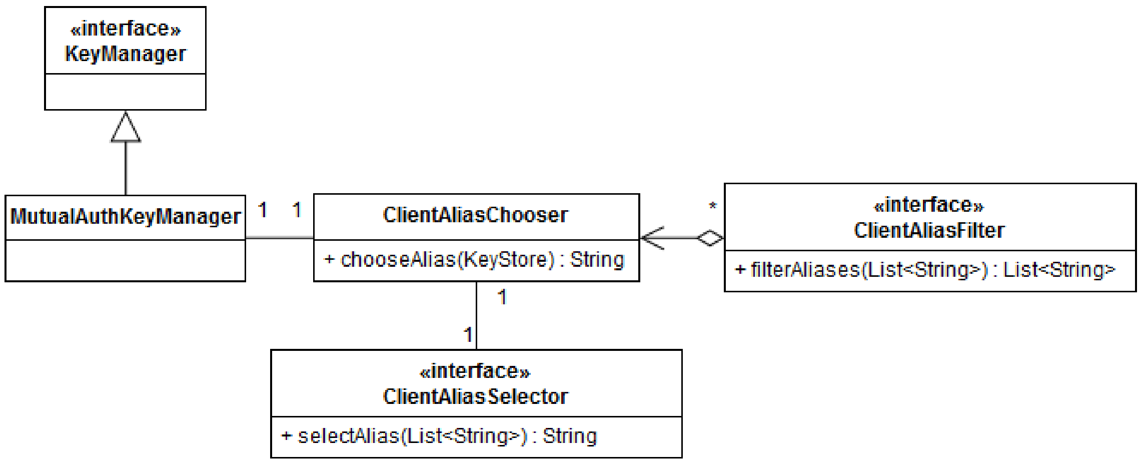
«interface» KeyManager
MutualAuthKeyManager
1
1
ClientAliasChooser
+ chooseAlias(KeyStore) : String
*
«interface» ClientAliasFilter
+ filterAliases(List<String>) : List<String>
1
1
«interface» ClientAliasSelector
+ selectAlias(List<String>) : String

Figure 27. *Design of the mechanism for choosing client certificates to present in SSL handshakes.*

Each certificate in the key store is represented by an alias, which is a string. First the *ClientAliasChooser* filters the aliases such that only certificates with e.g. appropriate names or algorithms remain. The remaining aliases are presented to an instance of *ClientAliasSelector*, which is implemented by a GUI window from which the user can select the certificate to be used in the connection.

Support was added for PKCS#11 (smart card), PKCS#12 and JKS key stores. During development, an additional requirement emerged for automatically choosing one of these key store types based on what key stores are available. For example, this key store choosing mechanism could be configured to use PKCS#11 if available and default to using JKS otherwise. This mechanism was implemented as a virtual 'AUTO' key store type combined with a configurable list of preferred key stores in order of priority.

Authentication Configuration

Program listing 4 shows an example configuration of the mutual authentication classes.

```
useKeyStoreType = AUTO
useTrustStoreType = JKS
pkcs11.libraryPath = C:/Programs/DigiSign/Cryptoki.dll
pkcs12.keyStorePath = C:/CRT/User.p12
pkcs12.trustStorePath = C:/CRT/Server.p12
jks.keyStorePath = C:/CRT/User.jks
jks.trustStorePath = C:/CRT/Server.jks
auto.keyStorePriorities = JKS, PKCS12, PKCS11
alias.allowedClientAlises = Authentication Certificate, User Certificate
```

Program listing 4. *Example authentication configuration for specifying key stores and trust stores to be used in mutually authenticated connections.*

The configuration file allows the user to choose which kind of key store and trust store to use. Using the given key store or trust store requires that the corresponding trust store or key store file path is specified. The `auto.keyStorePriorities` parameter decides the priorities of the different key stores when the AUTO keystore type is used to automatically choose the key store type based on the available key stores. The `alias.allowedClientAlises` parameter is used to decide which certificate aliases from the key store can be used in mutual authentication.

Use of Authentication in the Product

The authentication mechanism described in this chapter is used in the product's client only once per client application instance. Whereas before the session token was provided to the client application as a command line parameter, the new implementation fetches a session token from the authentication service immediately upon start up and sets it as a system property. From then on, other remote method invocations use the session token to refer to the active session. If fetching the session token fails, the client application is shut down.

4.3 Dependency Management

Apart from user authentication and starting the client application, the final function of JWS in the product is to provide the client application with all the dependencies needed to run it. Since the client application already requires a configuration package to be installed manually in the test and production environments, the most natural solution is to include the client dependencies with this package. This way, it is easy to add these dependencies to the class path of the launcher application.

The existing client application build configuration gathered all configuration files from different projects and produced a .zip configuration archive. This build process was modified to resolve all the client application dependencies, build the dependencies and add them into the mentioned .zip package as shown in Figure 28.

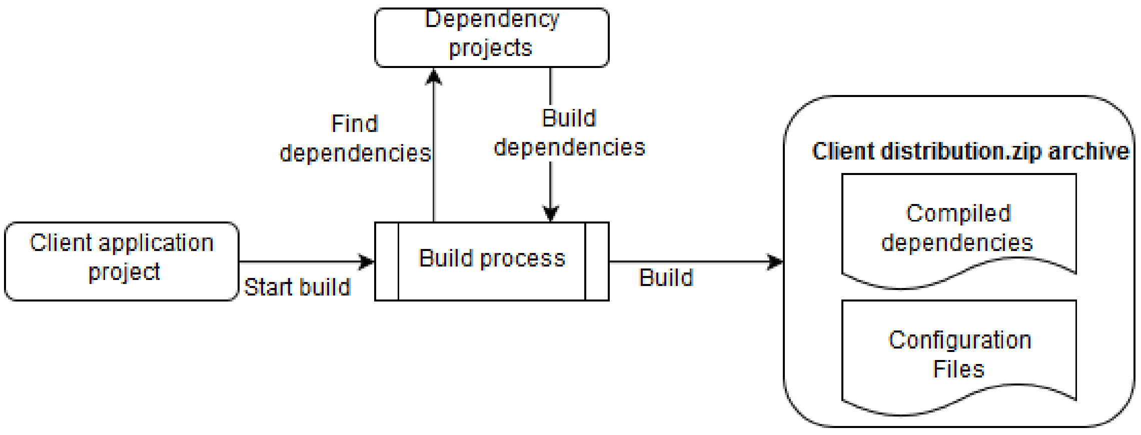

Figure 28. *Production of a distribution archive containing compiled Java dependencies and configuration files.*

This solution has the advantage of simplicity, as it only required changing the build configuration. It also simplifies testing in the test environment, since when testing changes developers can recompile and replace a single dependency as needed instead of having to create and install an entirely new version of the product. This was not possible using JWS, as all the libraries were signed and the application would not start if a dependency was changed without updating its cryptographic signature.

5. EVALUATION

5.1 Impact

The command and control system is used in a few dozen physical sites, each with several users. Because of the nature of the command and control system, its lifespan in operational use is likely to be measured in decades. The new session establishment is very visible, since the launcher application is the user's first point of contact with the product. Given the number of users, the long lifetime of the application, and the visibility of the end result, the implemented session establishment can have a large impact on the end users.

5.1.1 User Experience

From the perspective of the users in the production environment, starting the client application with the new session establishment is simple, with a pleasant and easy to use graphical user interface for starting client application in the right operating environment. This is a large improvement over the JWS-based session establishment in which there were command line scripts for each operating environment and in which the applications were started using a web browser.

In the test environment the new session establishment provides the same user experience improvements as in the production environment. Additionally, it makes debugging and testing much easier. Whereas before it was almost impossible to use profiling tools on applications started through JWS, applications can now be profiled with ease. Similarly test automation has become easier with the new session establishment solution, since it was difficult for the used automation testing framework to access the internals of a client application running in the JWS sandbox.

Whereas before making any change to the client application in the test environment typically required the re-installation of the command and control distribution, the new session establishment solution allows the user to simply recompile and replace the required library in the client application's class path, which rarely takes more than a minute. Since rebuilding and reinstalling the command and control system typically takes at least 30 minutes, this can translate into substantial time savings when debugging the client application in the test environment.

A comparison was made between the startup times of the client application using the old and new session establishment. Startup time was measured from the moment the user pressed the button to start the application to the moment that the client application GUI became visible. To reduce the effect of human reaction time, the test was configured not to ask the user for a key store password during mutual authentication, but use a hard-coded password instead. In both cases, GUI visibility also indicated successful session establishment via acquisition of the session token. The startup time using the new session establishment mechanism was 3.0 seconds on average. The JWS-based session establishment startup time depended on whether the application was started for the first time or not. For the first start up, JWS had to download all the client application dependencies from the application server, leading to a long average startup time of 21.3 seconds. For subsequent startups, JWS averaged 6.6 seconds. Based on the measurements it can be said that the new session establishment mechanism is approximately twice as fast as its predecessor.

The new session establishment mechanism also reduced the build time of the command and control system from an average of 22 minutes to an average of 17 minutes, a reduction of approximately 20%. This is explained by the client application dependencies not needing to be cryptographically signed as is the case with JWS. This makes developers waste less time waiting for a build and reduces the burden on the CI servers.

The new session establishment solution was developed so as to not interfere with the way the product is used in the development environment. For example, no authentication of the user is performed if a plain HTTP connection is used, as is the case in the development environment. As a result, the new system has no effect on the typical use of the product in the development environment. This is as intended, because session establishment was very rarely used in the development environment to begin with.

From the perspective of system administrators in the test and development environments there were some tradeoffs in the complexity of administration. Whilst server certificate authority (CA) certificates no longer need to be installed into the web browser, they have to be installed into a key store for the new implementation to work. Adding a launcher application and its associated configuration adds some complexity, but since the startup scripts used in the JWS-based session establishment solution can be removed, the overall complexity was lowered in this area. Finally, since the authentication service is accessed through a different port than the other web services, firewall configurations may need to be changed, adding slightly to the administration work.

It can be concluded that the new session establishment solution broadly achieves the user experience goals that it set out to achieve.

5.1.2 Reliability

The JWS-based session establishment solution suffered from a problem with reading the smart card after it is removed and replaced after a few minutes, rendering the client application inoperable. This problem has been solved in the new session establishment solution, since only the authentication service ever performs mutual authentication using the smart card, and it does this just once when the session is established. This means that once the application is started, the smart card can be removed and replaced freely, and the application still remains operational.

By eliminating this long-standing problem in the product, its reliability has been improved appreciably. Previously the client application was rendered inoperable in approximately 30% of the cases in the test environment, and 100% of cases in the customer's environment. Now that the problem has been solved, the customer can use the client application reliability without having to restart it each time they remove their smart card.

5.1.3 Security

The new session establishment mechanism has improved the security of the command and control system considerably. This is because the JWS-based session establishment required the user to open a JNLP file using the browser, which then downloads the file onto the local disk so to be opened by the JWS executable. As discussed in chapter 2.2.4, the session token is injected into the JNLP file to be passed to the client application as command line argument. This means that anyone with access to the local disk can open the JNLP file and see the session token in plain text.

By replacing JWS in session establishment, the session token is never leaked outside of the application in plaintext. Instead, every transmission of the session token is performed over a cryptographically secure SSL connection.

5.1.4 Maintenance and Further Development

One of the concerns with the JWS-based session establishment mechanism was the extent to which it was beyond the control of the development team. This is because it relied on a command line scripts, a web browser communicating independently with a smart card reader and JWS, none of which are directly accessible in the product's code. By using Java code to communicate with the smart card reader in the client application, and by creating a Java-based launcher application, much of this process has been brought under control of the developers. In case of any future problems or development needs, it is easy to work on the new session establishment, whereas it was almost impossible using the JWS-based session establishment mechanism.

Although it was imagined that a smart card reading component could be developed into its own product to be used within the company, it was quickly discovered that the Java platform abstracts away so much of the card interaction that there was no need for this. Despite this, the classes used in establishing mutually authenticated connections were general enough that some of them were used in another version of the product to better control how connections to the application server were made.

5.2 Potential Future Improvements

While implementing the new session establishment solution, some opportunities for further improvement were noted. Although they were related to session establishment, they were considered to be outside of the scope of this thesis. These potential improvements are presented here.

The launcher application's user interface (not the logic) created in connection with this thesis was very plain. Since the launcher application provides the customer's first glimpse of the product, it was decided that it be re-implemented to be more aesthetically pleasing using the modern Java FX user interface framework.

Another improvement likely to be implemented in the future is the filtering of available operating environments in the launcher application. Currently, the launcher application displays all the operating environments installed on the machine. However, the operating environments typically have restrictions on which users are allowed to access them. It could be possible for the launcher application to find out which operating environments the user is allowed to access and only present those environments to the user.

The previous session establishment solution would ensure the integrity of the client application dependencies by cryptographically signing them and refusing to run if they were tampered. Since this is feature of the replaced JWS technology, no integrity checks are performed. The security implications were considered, but it was decided that presently there is no need to protect the dependencies in this way, because the environments in which the product is used are considered secure enough, having no access to outside networks. This might change as the customer's security requirements evolve, meaning that similar integrity checks may be performed by the launcher in the future.

6. CONCLUSIONS

The purpose of this thesis was to redesign session establishment in a distributed command and control system. The work was undertaken because of a variety of disadvantages in using the previous JWS-based session establishment solution. Among these were poor end user experience, poor support for testing and debugging, poor support for continued development, unreliability and a dependence on an increasingly superannuated technology.

The problem was approached by elucidating the product's architecture, identifying how the previous session establishment worked, and gathering requirements for the new session establishment solution. It was decided that the new session establishment solution should handle all smart card interactions itself using Java code, support file-based certificates and allow the user to start applications from a Java launcher application.

Finally, the new session establishment solution was designed and implemented, resulting in a well-functioning system relying mainly on a web service for authenticating users based on their certificates from either a file or a smart card, and a launcher application for starting command and control client applications.

The implemented session establishment solution was evaluated for how well it achieved its stated goals and how it impacted end users. In comparison with the previous solution, it was deemed to provide greater opportunity for further development, improvements in usability, testability and maintainability, making it much better suited for the product's needs.

REFERENCES

[1] "Insta DefSec Website," 9 5 2016. [Online]. Available: http://www.insta.fi/defsec/en/. [Accessed 12 09 2016]

[2] Oracle, "Java SE 7 Documentation," [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/security/p11guide.html. [Accessed 12 09 2016].

[3] Oracle, "Servlet API Documentation," [Online]. Available: http://docs.oracle.com/javaee/7/api/javax/servlet/Servlet.html. [Accessed 12 09 2016].

[4] Apache Software Foundation, "AJP Protocol Reference," [Online]. Available: https://tomcat.apache.org/connectors-doc/ajp/ajpv13a.html. [Accessed 12 09 2016].

[5] C. Heinrich, "Secure Socket Layer (SSL)," in *Encyclopedia of Cryptography and Security*, Springer US, 2005, pp. 548-551.

[6] J. Davies, Implementing SSL / TLS Using Cryptography and PKI (1), Wiley, 2011.

[7] R. Oppliger, SSL: Theory and Practice, Artech House Inc, 2009.

[8] W. Rankl and W. Effing, Smart Card Handbook (4), Wiley, 2010.

[9] M. Keith and M. Konstantinos, Smart Cards, Tokens, Security and Applications, Springer, 2008.

[10] RSA Laboratories, "PKCS #11: Cryptographic Token Interface Standard," [Online]. Available: http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-11-cryptographic-token-interface-standard.htm. [Accessed 13 09 2016].

[11] Pivotal Software, "Spring Security Filter Chain," [Online]. Available: http://docs.spring.io/spring-security/site/docs/3.0.x/reference/security-filter-chain.html. [Accessed 12 09 2016].

[12] Pivotal Software, "Remoting and web services using Spring," [Online]. Available:

http://docs.spring.io/autorepo/docs/spring/3.2.x/spring-framework-reference/html/remoting.html. [Accessed 12 09 2016].

[13] Oracle, "Java Web Start Technology Documentation," [Online]. Available: http://docs.oracle.com/javase/8/docs/technotes/guides/javaws/developersguide/overview.html#jws. [Accessed 12 09 2016].

[14] Oracle, "JNLP File Syntax," [Online]. Available: http://docs.oracle.com/javase/8/docs/technotes/guides/javaws/developersguide/syntax.html#intro. [Accessed 12 09 2016].

[15] Oracle, "JWS Security and Code Signing," [Online]. Available: http://docs.oracle.com/javase/8/docs/technotes/guides/javaws/developersguide/development.html#security. [Accessed 12 09 2016].

[16] Oracle, "Finding Classes," [Online]. Available: http://docs.oracle.com/javase/8/docs/technotes/tools/findingclasses.html. [Accessed 12 09 2016].

[17] Microsoft, "Windows Environment Variables," [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682653(v=vs.85).aspx. [Accessed 12 09 2016].