An Object-Oriented Implementation of an Authentication Protocol

Master's Thesis

Sanna Suoranta

Helsinki University of Technology	Teknillinen korkeakoulu
Department of Computer Science and Engineering	Tietotekniikan osasto
Telecommunications Software and Multimedia Laboratory	Tietoliikenneohjelmistojen ja multimedian laboratorio
Espoo 1998	

HELSINKI UNIVERSITY OF TECHNOLOGY	ABSTRACT OF MASTER'S THESIS

Author:	Sanna Suoranta		
Name of thesis:			
An Object-Oriented Implementation of an Authentication Protocol			
Date:	November 30, 1998	**Pages:** 8 + 61	
Department:	Department of Computer Science and Engineering	**Chair:** Tik-110	
Supervisor:	Professor Arto Karila		
Instructor:	Pekka Nikander, Tech. Lic.		

Design Patterns and protocol frameworks are tools of object-oriented software development. They facilitate protocol design by offering reusable components. In this work, I used Jacob framework that is designed for developing cryptographic protocols.

There are many algorithms and protocols for key management and authentication. When two parties want to have secure communication path, they have to agree on how the connection will be established. The Internet Security Association and Key Management Protocol (ISAKMP) offers solutions for exchanging keys and authentication information. The ISAKMP itself is a framework of authentication and key management. It does not enforce using some specific algorithm or protocol.

In this Thesis, the Jacob framework is presented. The Conduits+ framework, that the Jacob is based on, is also presented. This Thesis present the Conduits+ graph of ISAKMP protocol and describes how a part of this protocol is implemented with help of Jacob framework. My prototype consists of Cookie handling and Base Exchange of the ISAKMP protocol.

Keywords: ISAKMP, Conduits+, Jacob, framework, authentication

Language: English

TEKNILLINEN KORKEAKOULU DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Sanna Suoranta		
Työn nimi:	Todennusprotokollan oliosuuntautunut toteutus		
Päivämäärä:	30. marraskuuta 1998	**Sivuja:**	8 + 61
Osasto:	Tietotekniikan osasto	**Professuuri:**	Tik-110
Työn valvoja:	Professori Arto Karila		
Työn ohjaaja:	TkL Pekka Nikander		

Suunnitteluhahmot ja protokollasovelluskehykset ovat oliosuuntautuneen ohjelmistokehityksen työkaluja. Ne helpottavat protokollien suunnittelua tarjoamalla uudelleen käytettäviä komponentteja. Käytän tässä työssä Jacob-sovelluskehystä, joka on suunniteltu kryptografisten protokollien kehittämiseen.

Avaintenhallintaan ja autentikointiin on monia algoritmeja ja protokollia. Kun kaksi osapuolta haluavat turvallisen yhteyden, heidän täytyy sopia kuinka yhteys muodostetaan. Internet Security Association and Key Management Protocol (ISAKMP) tarjoaa ratkaisun avainmateriaalin ja autentikaatioinformaation vaihtoon. ISAKMP-protokolla on itsekin autentikoinnin ja avaintenhallinnan sovelluskehys. Se ei pakota käyttämään jotain tiettyä algoritmiä tai protokollaa.

Tässä diplomityössä on esitelty Jacob-sovelluskehys. Conduits+ sovelluskehys, johon Jacob perustuu, on myös esitelty. Tässä diplomityössä esitellään Conduits+ kaavio ISAKMP-protokollasta ja kuvataan kuinka osa tästä protokollasta on toteutettu Jacob-sovelluskehyksen avulla. Prototyyppini sisältää ISAKMP-protokollan Keksien käsittelyn ja perusviestinvaihdon.

Avainsanat:	ISAKMP, Conduits+, Jacob, sovelluskehys, todennus
Kieli:	englanti

Acknowledgements

This thesis was written as a part of the Telecommunications Software Security Architecture (TeSSA) research project at Telecommunications Software and Multimedia Laboratory at Helsinki University of Technology. I thank all the employee of this project for supporting me during this work.

I want to thank my professor, Arto Karila. He encourage me to get this Thesis done in scheduled time and he read this work through in a week-end.

My special thanks to my instructor, Pekka Nikander. He patiently had the strength to direct me in my design and implementation and to correct my grammatical errors.

Finally, I thank my family and especially my fiancé Janne for support and patience. I am sorry that I had so little time for you all during this work.

Otaniemi 1998-11-30

Sanna Suoranta

Contents

List of Figures

Chapter 1

Introduction

Recently, the use of Design Patterns has been widely accepted in computer science, and software frameworks are gaining more acceptance. Both of these are tools of object-oriented software development. Design Patterns are reusable components that solve some common problem. Frameworks are extendible programs that are developed for some specific purpose. For example, the Conduits+ framework [11] is built for communication protocols and Jacob [18] is built for cryptographic communication protocols. Frameworks are usually black box frameworks where a programmer does not need to know about how the pieces work together. Object-Oriented frameworks are usually heavily grounded on Design Patterns.

The Internet Security Association and Key Management Protocol (ISAKMP) is a protocol framework that is used for negotiation of secure communication. ISAKMP can be used for authentication and key exchange with help of several algorithms [14]. ISAKMP itself does not enforce using some specific protocol or algorithm.

With the Jacob framework the IP, UDP, ICMP and ARP protocols have been implemented. These protocols are much simpler than the ISAKMP protocol, which is a framework itself. The goal of this work is to study how to construct an object-oriented design and implementation of the ISAKMP protocol. In this work, a part of ISAKMP is implemented with the help of the Jacob framework.

In the following, when describing or facilitating understanding of cryptographic protocols, the following principals are used:

Alice	First participant of a protocol
Bob	Second participant of a protocol
Eve	Eavesdropper
Mallory	Malicious active attacker

The rest of this Thesis is organized as follows. The second chapter define some key terms of distributed system security and authentication protocols. The third chapter presents the ISAKMP authentication and key management protocol. The fourth chapter describes object-oriented protocol design and tools used. The fifth chapter presents how to construct the ISAKMP architecture in object-oriented terms. The sixth chapter introduces the O-O implementation of the ISAKMP. Seventh chapter presents results and experiences, and finally the eighth chapter presents conclusions and summarizes the work.

Chapter 2

Background

From the beginning of computer history until about 1985, computers were large and expensive. Organizations had only few computers and computers operated independently. In the beginning of 80's, powerful microprocessors were developed and computers become common. The second development was the invention of high-speed local area networks (LANs). Networking computers together has found useful. With them large amount of computers can be connected together and information can be transferred very fast. A system where a large number of computers are connected through a high-speed network is called a distributed system.

Why build distributed systems when information is more secure in centralized ones and the network can cause problems? Distributed system can be more advantageous, effective and computing power can be added in small increments. In a distributed system there may be many users and users can share devices. Furthermore, information can be easily shared and users can communicate, for example, by electronic mail [23].

2.1 Distributed System Security

A distributed system consists of computers which communicate through a network. The Internet is made up of such systems. In a distributed system, there are a number of different computers running different operating system and programs. Also the network itself consists of many kinds of equipments manufactured by different vendors.

When a message is send through a network, the user does not know which path it travels along to its destination or whether it gets through at all.

In transmission, there may happen, for example, a routing error. Internet protocols like TCP get quite well over from such errors. The message can also be caught by a malicious party who intercepts or changes the message. To prevent malicious acts in a distributed system, users can use cryptographic protocols.

If a computer is not connected to any networks and it is in a well controlled room, it can be easily made physically secure. On the other hand, such a computer is not very usable. Today, most computers are connected to some network. In a distributed system, there are much of information that is confidential and services that only some users have right to access. Thus, we need a secure way to communicate in a distributed systems and to use services and information in the system.

According to Douglas Comer, most Internet security problems can be divided into three sets. The first set focuses on the problems of authentication and integrity; the second set focuses on the problem of privacy or confidentiality, while the third set focuses on the problem of availability by controlling access [4]. Internet Engineering Task Force (IETF) has also defined six security services [8]. These services are Integrity, Confidentiality, Authentication, Non-Repudiation, Access Control and Availability. Authorization is not in these security services but nowadays, it has been found very important.

In the next sections, there are some definitions and examples for these security terms that are used when discussing distributed systems security.

2.1.1 Integrity

In order to a system work right, information stored in it should be correct. In hospitals, for example, the patient's medical history has to be correct when a doctor chooses a medicine for the patient who is, e.g., allergic to something.

In computer security, integrity means that only those people who have the right to change information can do that. The information must not change during storage or transmission. If the representation of some information has been changed even by a mistake, the integrity of this piece of information has been compromised.

2.1.2 Confidentiality

When people talk about computer security, confidentiality is usually the first thing that comes to mind. According to Edward Amoroso, the majority of research and development in computer security has specifically focused on disclosure threats [1]. Disclosure is a threat that involves someone accessing information that should not be seen. Respectively, confidentiality means that only legitimate users have access to information.

In a company, for example, only the president and members of the board have access to all information concerning future decisions; chiefs of departments may have some information concerning decisions that affect their department. The chiefs must not tell their subordinates about the secrets.

Usually information must be protected from both disclosure and integrity threats.

2.1.3 Availability

Working outside an office is becoming more popular. Many companies also have several offices around the world. Both information and services may be distributed. On the other hand, services of a company may be centralized in one or a few locations.

System availability means that information and computer resources are available when a legitimate user needs them. The threat of denial of service occurs when access to some computer resource is blocked. The blocking may be permanent or long enough that the use of the resource is no longer useful.

Disclosure and integrity have been considered to be the primary threats. Less attention has been paid to availability and denial of service attacks.

2.1.4 Authentication

In order to maintain confidentiality and integrity in distributed systems the user, i. e. Alice, has to proof that she has right to access the system. Traditionally, Alice has to log in to the system by correctly typing a username and a password. In more modern systems, smart cards or finger print recognition devices may be used. Thus, user identification is based on something that an user knows (a password), something that the user has (a smart card) or something that the user is (a fingerprint).

In order to allow access for legitime users to a computer system, they have to identify themselves. Strong identification, or authentication, means that the system can be sure of a user's identification.

In his Licentiate's thesis, Pekka Nikander makes a distinction between two different types of authentication: message origin authentication and peer authentication [17]. Message origin authentication means that we can be sure of the identity of the originator of a message. The message must also be integral if we want to be sure that the message is received as it was sent. Peer authentication is a way of identifying a peer at the other end of a secure connection. It ensures that the person is who he claims to be and we can be sure of it.

When Bob gets a message from Alice, it should be possible for him to ascertain the origin of the message. The message can be considered to be an electronic document that contains some information. Nobody should be able to send a message that looks like having been sent by someone else.

2.1.5 Access Control

Access control is a mechanism to control access to a computer system and information. There are two main types of access control: discretionary access control (DAC) and mandatory access control (MAC). A discretionary access control allows users to affect how their files and computer resources are protected. Of course, the security policy limits users power to allow access.

A mandatory access control does not allow users to influence to access control. The system administration makes decisions according to the security policy. Typically, both discretionary and mandatory access control may be used together.

2.1.6 Non-repudiation

A contract is usually accepted by signing it. Every party gets its own piece of the contract. If the contents become disputable, nobody can deny it because everybody has signed the contract and everybody has a similar copy (this assumes, of course, that the authenticity of the signatures can be verified).

In a distributed system, non-repudiation means that the sender shall not be able to deny later that she has sent a message or that the receiver can not deny that he has received the message. For example, in electronic commerce, a client should not be able to deny that he has ordered a product.

In the computer world, non-repudiation is carried out with signatures conceptually similar to ones in the real world.

2.1.7 Authorization

When two parties conclude a contract, both of them have to signed it. The signers have to know that other party is authorized to sign contracts. If neither party has authorization, the contract is invalid.

Authorization in a computer system means that someone, a user or a program, gets rights to do something. Today users usually have to identify themselves to get access rights. This is often unnecessary, because the computer system does not need to know the identity of the user, it only needs to know if the user has a right to access to information or to perform an operation.

If your grandmother is too sick to go to a bank to take care of her business, she can write a letter that authorizes you to act on her behalf, i. e., a proxy.

A proxy is a token that allows user to operate with the rights of the principal that granted the proxy [16]. There are proxies also in computer world. For example, an user can have a certificate that allows he to assign the right to use some service.

2.2 Cryptography

When Alice and Bob want to communicate securely through an unsecured network, they have to use cryptography to protect their communications. A cryptosystem consists of an algorithm that is used to secure communications and the keys that are used for encryption. All plaintexts and ciphertexts belong also to the cryptosystem.

First Alice and Bob choose an algorithm for protecting communication, then they agree on a key. Alice uses the chosen key to encrypt the message and sends the ciphertext to Bob. Then Bob can decrypt the ciphertext and he gets Alice's original message. Alice can use two different types of methods to encrypt a message: she can use symmetric cryptography or public key cryptography.

A cryptographic algorithm is a mathematical function that is used for encryption and decryption. A synonym for cryptographic algorithm is a cipher [22].

2.2.1 Symmetric Ciphers

In symmetric ciphers, the encryption key can be calculated from the decryption key, and vice versa. In most symmetric ciphers the encryption key and the decryption key are the same. These algorithms are also called secret-key algorithms.

When symmetric ciphers are used, Alice and Bob have to agree on a key before they can communicate securely. The key must be kept secret as long as the information which is encrypted with the key needs to remain secret.

Symmetric ciphers can be divided into two categories: stream ciphers and block ciphers. A stream cipher takes one bit from the plaintext at a time and encrypts it. A block cipher takes a group of bits.

Symmetric cryptosystems have some problems: the keys must be distributed in secret to the communicating peers. Everyone who has a specific key can decrypt messages that are encrypted with that key and encrypt false messages to fool the other parties. When a group of people wants to exchange messages secretly from each other, each pair has to use their own secret key. For example, 10 users need 45 different keys to talk with each other [22].

2.2.2 Public Key Ciphers

Public key ciphers have different keys for encryption and decryption. The decryption key cannot be calculated from the encryption key in any reasonable amount of time. The encryption key is called the public key and the decryption key is called the private key. Accordingly, the public key can be made public. If Alice and Bob use a public key cipher, and Alice wants to send a message to Bob, she takes Bob's public key (maybe from Bob's home page or from his visiting card), encrypts the message and sends it to Bob. Only Bob has the corresponding private key, and therefore only he can decrypt the message.

2.2.3 One-way Hash Functions

A hash function is a function that takes a variable length input and converts it to a fixed length output. The input of hash function is called a pre-image and the output is called a hash value or digest.

A one-way hash function is a hash function that works in one direction. That is, a hash value is easy to compute from a pre-image, but finding a pre-image that hashes to certain value is hard. With a good one-way hash function it is also hard to generate two pre-images that have the same hash value. Such a hash function is said to be collision free.

Hash functions can be used, for example, as a message fingerprint. If Alice wants to know that Bob has some secret file (a piece of information), she can ask him to send a hash of that file. If the hash value is correct, Alice can be almost certain that Bob does have the file. When a hash function is used without a key, everyone having a copy of the pre-image can verify the hash.

2.2.4 Message Authentication Codes (MAC)

A message authentication code (MAC) is an authenticator. It is calculated by a one-way hash function from the data to be authenticated together with a secret key. The idea of a MAC is exactly the same as that of basic hash functions, but only a key holder can generate and verify the hash value. MACs can be based on a hash function or a block encryption algorithm.

MACs can be used to authenticate files between users. With the help of a MAC a user can notice if his files have been changed by a virus or some attacker.

2.2.5 Digital Signatures

Signatures are important proofs of volition of an individual. We sign our sales tickets when we buy things with a credit card. Presidents of companies and heads of states sign contracts. According to Bruce Schneier, signatures are used because the signature is authentic and unforgeable, it is not reusable, the signed document is unalterable, and the signature cannot be repudiated [22]. In reality, none of these are completely true.

In the computer world signatures have even more problems. Computer files are easy to copy, a piece of information can easily moved from one document to another document, and files are easy to modify.

Digital signatures can be made with help of a trusted third party or with public key cryptography.

There are two kinds of public key algorithms that can be used for digital signatures. Some of them can be used both encrypting and signing of an documents; others are separate algorithms for digital signatures, and they cannot be used for encryption. If Alice wants to sign a document with a public key algorithm, she encrypts it with her private key. Then Alice can send the signed document to Bob who can verify the signature by decrypting the document with Alice's public key.

Chapter 3

Authentication and Key Management

3.1 Introduction

Secure communication between two parties, Alice and Bob, can be taught of as a box that can be locked. Alice puts the message in the box and locks it with her key. Then she gives the box to an office boy that transports the box to Bob. Bob opens the box with his key. Alice and Bob have got their keys from some bureaucrat that manages keys.

In computer system, the procedure is similar to the procedure above. Alice encrypts the message with her key or with Bob's public key. Then she sends the encrypted message to Bob through network. Bob decrypts message with the key (shared key or his own private key). The keys can be generated by Alice and Bob or by some trusted third party.

3.2 Authentication and Authentication Protocols

Protocols provide rules for communication between computers. Protocols define message formats and specify how to handle arriving messages and errors. Standardized protocols are vendor independent. A security protocol performs a security service (or more than one service) for networked communications.

In his Licentiate's thesis, Pekka Nikander defines an authentication protocol to be a cryptographic protocol whose purpose is to create enough of evidence that one or more of the communicating parties may reasonably believe in something [17]. One typical goal of authentication protocols is that a client can give a proof to a server that it has right to access to some service.

Digital signatures are often used in cryptographic protocols. If a message is signed digitally, the receiver can be sure of its authenticity and integrity. Symmetric cryptography also provide authenticity, but the receiver can not convince someone else that the message was not sent by itself.

3.2.1 Examples of Authentication

In computer systems, users are usually authenticated. When users are authenticated, the system knows what they are allowed to do. Traditionally, the identity of user is authenticated with passwords, which is a weak method of authentication. Some methods are biometrics. For example, a fingerprint is good proof that user is who he claims to be. Authentication can also be based on a smart card.

Some computer systems do not need to know the identity of a user but they need to know that the user has right to access the system. A smart card can be also used in this case. The card may contain information of authority instead identity. The second method for authentication of authority is use of authorization certificates. Certificates combine information to global identifier (a name or other information like a key) and they can be used as a proof of authority.

Messages can also be authenticated. When an employee gets an electronic mail from his superior, how the employee can know that the message is authentic? The superior can sign the message and then the employee know that the message is authentic. Other solution to the authentication of message origin is the Message Authentication Code (MAC). Using of public key cryptography allows also third parties to become convinced of message origin.

3.2.2 The Idea of Authentication Protocols

Authentication protocols are developed for authenticate other peer's identity or authority or origin of a message. The next section gives an example of authenticating communication peer.

Some protocols combine authentication and key exchange. These protocols solve a general problem: How Alice and Bob can exchange a secret key and at the same time be sure that they are communicating together and not with some other person.

3.2.3 An Example of Authentication Protocol

SKID2 is an identification protocol that is based on a symmetric algorithm [22]. Bob can use SKID2 to prove his identity to Alice. In SKID protocol, Alice and Bob share a secret key K.

(1) Alice chooses a random number, R_A. She sends it to Bob.

(2) Bob also chooses a random number, R_B. He Sends Alice: $R_B, H_K(R_A, R_B, B)$ where H_K is a MAC and B is Bob's name.

(3) Alice computes $H_K(R_A, R_B, B)$ and compares it whit what she received from Bob. If both are same, Alice knows that she is communicating with Bob.

With SKID3 also Bob can be sure that he is communicating with Alice.

(1) Alice sends Bob: $H_K(R_B, A)$ where A is Alice's name.

(2) Bob computes $H_K(R_B, B)$ and compares it whit what he received from Alice. If the results are identical, Bob knows that he is communicating with Alice.

These protocols are not secure against man-in-the-middle attacks. This attack is presented in section 3.3.2.

3.3 Key Management

3.3.1 Key Generation

In a cryptosystem, use of a strong mechanism for key generation is very important because the security of an algorithm rests on the key. If the key generation algorithm is weak, Eve the Eavesdropper needs not to cryptoanalyze the encryption algorithm, but she can solve the key by breaking the key generation.

A key generation algorithm should uniformly generate all keys of the keyspace. If only some keys are used (for example, only ASCII keys are allowed), the keyspace is smaller and easier to break.

When the users can choose their own keys, they generally choose poor keys that are easy to remember. Keys like a user's name are vulnerable to dictionary attacks. A smart attacker uses a dictionary of common keys to solve the key. Good keys are random bit sequences when every key is equally likely. Some encryption algorithms have specific keys that are less secure than other keys. For example, DES has 16 weak keys [22]. These weak keys should be avoided.

3.3.2 Transferring Keys

In a symmetric cryptosystem, all communicating parties need to have the same key. The key must be given securely to all parties. For example, the key can be divided into parts that are send to all parties via separate channels, that is, one part could be sent via telephone, the second part via mail and the third part via telefax.

If a group of N persons want to use symmetric cryptography and communicate together securely from each other, they need n(n-1)/2 keys. In a large network, the use of a Key Distribution Center makes exchanges much efficient; initially only n keys are needed.

Key management is easier in public key cryptosystems. Each user has a public key that can be published, for example, at the user's home page. Everyone can use a public key for encrypting a message but only the correct receiver can decrypt the message. However, public key cryptosystems are vulnerable to man-in-the-middle attacks.

In the man-in-the-middle attack some malicious person is in the middle of communication. The communicating parties suppose that they are communicating securely with each other. For example, when Alice sends Bob her public key, Mallory intercepts this transmission and sends Bob his own public key. Then Bob sends to Alice his public key, but Mallory intercepts also this transmission and sends Alice his own public key. Now Alice and Bob believe that they have each other public keys. Alice encrypts message with "Bob's" key and sends it to Bob. Mallory intercepts the message, decrypts it, encrypts it again with Bob's key and sends it to Bob. The man-in-the-middle attack works when Alice and Bob has no way to verify other party. Certificates provide on solution to the man-in-the-middle problem. They are discussed in section 3.3.5.

3.3.3 Lifetime of Keys

Bruce Schneier gives in his book [22] several reasons why an encryption key should not be used for an indefinite period:

- The longer a key is used, the greater the chance that it will be compromised.
- The longer a key is used, the greater the loss if the key is compromised.
- The longer a key is used, the greater the temptation for someone to spend the effort necessary to break it.
- It is generally easier to do cryptoanalysis with more ciphertext encrypted with the same key.

A local security policy should determine the lifetime of keys. Different keys may have different lifetimes. The lifetime depends on purpose of use. For example, private keys that are used for digital signatures have usually years of lifetime while session keys should be changed at least every day.

3.3.4 Storing Keys

Some systems do not store keys at all. Alice must remember her own key and she has to enter it to the system every time she wants to encrypt or decrypt some information.

Keys can be hard to remember. These kinds of keys can be encrypted with other keys and stored in a hard drive. Also a tamperproof smart card is a good for storing keys. Users need not to remember keys and the storage itself is safe. A physical item like a smart card is perceived important and users take care of the key better.

3.3.5 Certificates

How Bob can know that the public key that he has received from Alice, belongs to Alice? Certificates are one solution to this; they can bind key to user.

A digital certificate is a signed record that states some information about the entity the certificate was issued to [19]. For example, the entity may be a public key or a name and the information may be the entity's name or what the entity is allowed to do.

There are two types of certificates: identity certificates or authorization certificates. The identity certificate tells that the issuer of the certificate knows or beliefs that the holder of the certified key has a certain name, which is given in that certificate. X.509 certificates are example of identity certificates [12]. An authorization certificate tells that some key has authority to do something. This kind of certificates are rather new. The IETF is standardizing SPKI certificates that can also be used as authority certificate [5].

3.4 Internet Security Association and Key Management Protocol

The Internet Engineering Task Force (IETF) is in the process of standardizing the Internet Security Association and Key Management Protocol (ISAKMP). It is a protocol that combines the security concepts of authentication, key management, and security associations to establish the required security for government, commercial, and private communications on the Internet. ISAKMP defines procedures and packet formats to establish, negotiate, modify and delete Security Associations (SA) [14].

3.4.1 Terminology of the ISAKMP Protocol

A *Security Association* (SA) defines a set of services and mechanisms that are needed to protect communication. Each Security Association has its own unique identifier that is a pair of (security policy, SPI) where the SPI is a Security Parameter Index.

ISAKMP is a *framework*. It does not enforce one to use a specific cryptographic algorithm, key generation technique or security mechanism.

In the ISAKMP, Security Associations are negotiated in two *phases*. In the first phase, two ISAKMP servers agree on how the forthcoming negotiation communication is to be protected, and a generic ISAKMP SA is established. In the second phase, negotiated security associations for some other security protocols are established. These security associations can be used to protect message/data exchanges of different security protocols.

In the ISAKMP, a *Cookie* is a hash that is calculated from the IP Source and Destination Addresses, the UDP Source and Destination Ports, date

`0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1`

Initiator Cookie				
Responder Cookie				
Next Payload	MjVer	MnVer	Exchange Type	Flags
Message ID				
Length				

Figure 3.1: ISAKMP Header Structure

and time, and a locally generated secret random number. The Cookie helps users to protect the communication from outsiders and from denial of service attacks.

A *Domain of Interpretation* (DOI) defines payload formats, exchange types and conventions for naming security-relevant information such as security policies or cryptographic algorithms and modes.

ISAKMP defines payloads which are used to transfer standardized information. ISAKMP has also number of exchange types. These exchanges define a set of security services. ISAKMP allows definition of new kinds of exchanges. The next two sections present ISAKMP payloads and exchange types more precisely.

3.4.2 ISAKMP Payloads

An ISAKMP message is modular. It consists of an ISAKMP Header and a number of payloads. In this section, the most important payloads are introduced. These payloads are needed for Security Association negotiation.

ISAKMP Header Format

Every ISAKMP message begins with an ISAKMP Header, which is not actually a payload. This header contains information to maintain state, process payloads, and possibly prevent denial of service or replay attacks. The structure of the ISAKMP Header is depicted in figure 3.1.

The Initiator and Responder Cookies in the ISAKMP Header are used to identify the ISAKMP SA to which the message belongs. The Next Payload field tells the type of the first payload in the message. Major and Minor

Versions indicate the used version of the ISAKMP protocol. The Exchange Type field tells the type of used exchange. Flags are options for the ISAKMP exchange (for example, if the encryption bit is set, all payloads are encrypted). Message ID is a unique message identifier. It is used during Phase 2 negotiations; in Phase 1 negotiations it must set to 0. Length is total length of the message (header + payloads) in octets.

General Payload Header

`0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1`

Next Payload	RESERVED	Payload Length

Each ISAKMP payload begins with a General Payload Header. The first field of it tells which payload is next in the message. The second field is reserved for future use. Payload Length is length of the payload in octets including the generic header.

Security Association Payload

The ISAKMP messages for the negotiation and establishment of Security Associations is built from the Security Association, Proposal and Transform Payloads. After one SA Payload, there comes at least one Proposal Payload, which is followed by at least one Transform Payload. In the Next Payload field of the SA Payload is the value of the next actual payload, not the type for Proposal or Transform Payloads.

With the Security Association Payload two parties can agree on security attributes, a Domain of Interpretation and a Situation under which the negotiation is taking place.

`0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1`

<table>
<tr><td>Next Payload</td><td>RESERVED</td><td colspan="2">Payload Length</td></tr>
<tr><td colspan="4">Domain of Interpretation (DOI)</td></tr>
<tr><td colspan="4">Situation (variable)</td></tr>
</table>

The DOI value is zero during Phase 1 negotiation when a generic ISAKMP SA is established. Internet Assigned Numbers Authority (IANA) has reserved all other DOI values for future use, except value 1 which is assigned to the IPSec DOI [20]. The Situation field is DOI-specific. It is used to make policy decisions.

The Proposal Payload contains a Security Parameter Index (SPI) which specifies a protocol that is used for negotiation of services and mechanisms. One Security Association payload may contain several Proposal payloads. These proposals are arranged in the initiator's preference order.

Every Proposal Payload has a proposal number. If two payloads have the same number, they belong together. For example, there might be a message whose Security Association payload contains two Proposal Payload with same proposal number. The first Proposal Payload insists of using IPSec ESP [3] and the second Proposal Payload insists of using IPSec AH [2]. If there are no other proposals, communication must secured with both ESP and AH.

`0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1`

Next Payload	RESERVED	Payload Length	
Proposal #	Protocol-Id	SPI Size	# of Transforms
SPI (variable)			

The Next Payload field of Proposal payload has only two possible values: "0" means that there are no more Proposal Payloads in this message, and "2" means that there is at least one additional Proposal Payload after this one. The entire length of the payload, including all Transform Payloads associated with this Proposal Payload, is calculated to the Payload Length field.

Similarly, a Proposal Payload may contain one or several Transform Payloads. The Transform Payload contains DOI-specific information about security association attributes. The Next Payload field must be "3" or "0". If there are additional Transform Payloads in the proposal, this field is 3, otherwise it is 0. Each Transform Payload within the Proposal Payload has a unique Transform number. Transform-Id depends on the protocol that is negotiated.

`0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1`

Next Payload	RESERVED	Payload Length	
Transform #	Transform-Id	RESERVED2	
SA Attributes (variable)			

The last field of a Transform Payload contains SA Attributes. These Data Attributes have also specific format. The first two octets contain a unique identifier for each type of attribute (Attribute Type field). The most significant bit indicates Attribute Format. The Attributes have two formats: the Type/Length/Value (TLV) format and the Type/Value (TV) format. The second field of Data Attributes contains the length in octets of the Attribute Value. If the Attribute Format is Type/Value, the Attribute Value is only two octets and the Attribute Length field is not present. An Attribute Value field is the last field of Data Attributes. The actual value depends on the DOI used.

Key Exchange Payload

The ISAKMP framework does not enforce of using some specific key exchange technique. For example, Oakley, Diffie-Hellman and PGP can be used in ISAKMP. Oakley defines a method to establish an authenticated key exchange [9]. Oakley has four modes that can be used for key exchange. Oakley uses the Diffie-Hellman key exchange protocol. Diffie-Hellman was the first invented public key algorithm. It is based on difficulty of calculating discrete logarithms [22]. With Diffie-Hellman, keying material can be exchanged in an unsecured network. PGP uses IDEA for data encryption, RSA for key management and digital signatures and MD5 for one-way hashing.

0 1 2 3 4 5 6 7	8 9 0 1 2 3 4 5	6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
Next Payload	RESERVED	Payload Length
Key Exchange Data (variable)		

Data required to generate a session key is in the last field of a Key Exchange Payload. This data is specified by the DOI and the Key Exchange algorithm.

Identification Payload

An Identification Payload carries DOI specific data that is used for identification information exchange.

0 1 2 3 4 5 6 7	8 9 0 1 2 3 4 5	6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
Next Payload	RESERVED	Payload Length
ID Type	DOI Specific ID Data	
Identification Data (variable length)		

The ID Type field specifies the type of used a Identification. The last two fields are DOI specific identity information.

Hash Payload

The Hash Payload looks like the Key Exchange Payload. In the last field there is a hash value of some part of the ISAKMP message and/or state. This information can be used to verify the integrity of the data or for authentication.

Nonce Payload

The Nonce Payload looks like the Key Exchange Payload, too. The Nonce Data field contains random data generated by the transmitting entity. Nonce is used to guarantee liveness during an exchange and to protect against replay attacks.

Other Payloads

Other ISAKMP payloads are Certificate Payload, Certificate Request Payload, Signature Payload, Notification Payload, Delete Payload and Vendor ID Payload. The first two are used for transmitting certificates and informing to the communication peer which certification authorities are accepted. Signature Payload contains data generated by a digital signature function. Notification Payload is used to transmit informational data to the peer. This information can be for example an error type. A Delete Payload informs the receiver that the sender has removed its security association from his database and the SA is no longer valid. A Vendor ID Payload contains a constant that is used for identifying and recognizing the vendor of the implementation.

3.4.3 ISAKMP Exchanges

ISAKMP defines five default exchange types: Base Exchange, Identity Protection Exchange, Authentication Only Exchange, Aggressive Exchange and Informational Exchange. In all these exchanges, the ISAKMP SA is negotiated first. Different exchanges provide different protection for the exchanged information. These exchanges are described in more detail in the next sections.

Base Exchange

In the Base Exchange basic key and authentication information is transmitted. It does not protect the identity of communicating parties. The next table shows the messages exchanged in the Base Exchange. The first column of table is sequence number of message, the second is sender of the message, the third is receiver of the message and the last field indicates which payloads are included in the message.

			Message
1	Alice	Bob	Header, Security Association, Nonce
2	Bob	Alice	Header, Security Association, Nonce
3	Alice	Bob	Header, Key Data, Identity of Alice, Authentication
4	Bob	Alice	Header, Key Data, Identity of Bob, Authentication

In the first message, the initiator (Alice) proposes how the negotiation should be protected. These propositions are in Security Association, Proposal and Transform payloads. The Nonce guarantees liveness, and it protects the exchange against replay attacks. In the second message, the responder (Bob) chooses a suitable protection method. The Nonce has the same task as above.

In the third and fourth messages, the initiator and responder exchange key material and identification information.

Identity Protection Exchange

The Identity Protection Exchange is the only ISAKMP exchange that protects identity and authentication information of parties.

			Message
1	Alice	Bob	Header, Security Association
2	Bob	Alice	Header, Security Association
3	Alice	Bob	Header, Key Data, Nonce
4	Bob	Alice	Header, Key Data, Nonce
5	Alice	Bob	Header(e), Identity of Alice, Authentication
6	Bob	Alice	Header(e), Identity of Bob, Authentication

In the first two messages, parties negotiate a tentative Security Association. Then the parties exchange keying material and nonces to guarantee liveness. In the last two messages, everything after the header is encrypted with the key negotiated by earlier messages. These messages contain identity and authentication information.

Authentication Only Exchange

Authentication Only Exchange does not exchange keying material, and none of the transmitted information is encrypted. The encryption may be done elsewhere.

			Message
1	Alice	Bob	Header, SA, Nonce
2	Bob	Alice	Header, SA, Nonce, Identity of Bob, Authentication
3	Alice	Bob	Header, Identity of Alice, Authentication

In the first message, Alice proposes a method to protect traffic. The message contains also a Nonce to guarantee liveness. In the second message, Bob accepts a suitable Security Association. This message also contains identity and authentication information of Bob. Alice responses in the third message and tells her own identity and authentication information.

Aggressive Exchange

Aggressive Exchange combines all information exchanged in the Base Exchange in three messages.

			Message
1	Alice	Bob	Header, SA, Key, Nonce, Identity of Alice
2	Bob	Alice	Header, SA, Key, Nonce, Identity of Bob, Authentication
3	Alice	Bob	Header(e), Authentication

The first message of the Aggressive Exchange contains same information than that of the Base Exchange. Keying material and the identification information of the initiator is also included in the first message. In the second message, Bob accepts one of the security associations proposed by Alice. This message contains also keying material, a nonce as well as identification and authentication information. In the third message, Alice sends encrypted information to complete authentication on her behalf.

Informational Exchange

An Informational Exchange message can be send during every other exchange. The Informational Exchange contains either notification information or a Delete Payload. This exchange has only one message which is send by either the initiator (Alice) or the responder (Bob) to inform the other party about deletion of a security association or some error occurred during an exchange.

			Message
(1)	Alice	Bob	Header(e), Error notification or Deletion

An Informational Exchange message can be transmitted during both Phases of ISAKMP. The message has no protection if it send during Phase 1 negotiation before keying material has been exchanged. After exchange of keying material or ISAKMP SA establishment, the transmission of the Informational Exchange message is protected by the keying material or the ISAKMP SA.

3.5 Discussion

The ISAKMP provides a standardized way to change information between parties that want authenticate the each other and exchange a key with the other peer. The idea of the ISAKMP is to provide general methods for exchanging security related information. Because ISAKMP is a framework, adding new properties to it is easy. For example, not all of the ISAKMP payloads are used in current exchanges. The developer can add own exchanges to ISAKMP. For example, certificates can be exchanged and verified with ISAKMP when a suitable exchange is developed.

The ISAKMP draft is clearly written with a functional programming language in mind. The fifth section of it defines ISAKMP payload processing very precisely. According that section, the ISAKMP is easy to implemented with some functional programming language. An object-oriented language allows more flexible ways to a programmer, but leads some problems. The programmer should decide where all payload processing must be implemented.

In the next chapter, some object-oriented design tools are introduced. These tools are Design Patterns and object-oriented frameworks.

Chapter 4

Object-Oriented Protocol Design

4.1 Introduction

Object-oriented programming is based on a different way of thinking than functional programming. A programming problem must be divided into classes. Ideally, these classes are easily reusable; especially, when they are well documented and proven to be usable in many applications. These kinds of collections of object classes may become known as design patterns or widely accepted models for future use.

Communication protocols have usually lot of similar functionality. Frameworks allow program developers to concentrate on problems that are specific for the protocol being implemented. A framework is developed to solve problems of some specific area. For example, Jacob framework is for developing cryptographic communication protocols.

There are two kinds of frameworks: black-box and white-box frameworks. The black-box frameworks are easier to use but harder to design than white-box frameworks. Users that use a white-box frameworks need more knowledge about the implementation of the components that they reuse. The components of white-box framework are reused usually by inheritance. Users of a black-box framework reuse its components mostly by composing instances. The Conduits+ framework is an example of a black-box framework [11].

The following sections present design patterns and frameworks, and give examples of them. The last section of this chapter describes development process and design tools.

4.1.1 Design Patterns

In 1977, Christopher Alexander wrote a book titled “A Pattern Language” about design patterns that are used in building architecture. In book, he describes a structured set of patterns that are based on experiences and knowledge of architects. In the beginning of 1990's, the developing of design patterns in computer science begun. The most influenced book was written by the “Gang of Four”: Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (also known as GoF) [7]. In this seminal work, the four authors described 23 generic design patterns.

A Design pattern is documentation that describes a general solution to a common problem. Usually, a pattern has four elements:

- The **pattern name** is few words that describes the pattern. The name allows programmers to discuss their solutions more easily.
- The **problem** describes when the pattern should be used. It explains the problem and its context.
- The **solution** is a template that describes the elements of design. It does not explain concrete design or implementation.
- The **consequences** are the results of using the pattern. The consequences concerns usually space and time trade-offs.

Design patterns can be classified by two criteria. The first criterion is purpose: the pattern can have either creational, structural or behavioral purpose. Creational patterns are used to create objects. Structural patterns handle structures of classes or objects. Behavioral patterns describe how classes or objects interact. The second criteria is scope, which can be a class or an object. Class patterns describe relationships between classes and their subclasses. This relationship is static (fixed when the classes are compiled). Object patterns handle object relationships, and are more dynamic in nature.

How a programmer can use design patterns? First he has to choose one that fits to his problem by reading pattern documentation. Then the user should carefully read the chosen pattern through and look at the sample code to get an example of using this pattern. After this, the user has to make such a class model from the pattern that fits to his own problem. The last phase is implementation of the solution.

4.1.2 Examples of Design Patterns

The next sections present three design patterns that are useful in implementation of communication protocols. All of these pattern are described in the book of GoF [7] and they have different purpose; however, all of them handle objects.

The Singleton pattern is a creational pattern that ensures that a class has only one instance. The Composite pattern is a structural pattern. The Composite pattern has two kinds of classes: leafs and composites. The composites can consist other composites and leafs. The Composite pattern represents part-whole hierarchies and lets clients threat individual object and compositions of objects uniformly. The Visitor pattern lets the programmer to define a new operation without changing the classes of elements on which it operates. It is a behavioral pattern.

All of these three design patterns are used in my ISAKMP implementation. Two of then can be found in the conduit framework. There are also other design pattern that are used, but they are not presented in here.

Singleton

The Singleton pattern is useful when there must be exactly one instance of a class and the clients should be able to access it. It is also handy when the sole instance has extensible subclasses, and the clients should be able to use a subclass without modifying their code.

The Singleton defines a static Instance operation that lets clients access the unique instance. This pattern may also be responsible for creating the unique instance.

Using the Singleton pattern has several benefits. The Singleton have control over how and when clients access it. By using the Singleton pattern, it is easy to change the implementation to allow more than one instances of a class.

The Singleton pattern is used in Conduits+ framework to make sure that every state class in a protocol has exactly one instance. The state is also a pattern itself (State pattern).

Composite

The Composite pattern can be used, for example, to present graphical objects like pictures. A picture can consist of some text, a couple of lines and a rectangle. The lines can be grouped and they become an another picture. The original picture now consists text, rectangle and picture that consists the lines. The text and the rectangle are leafs of a composite structure and the picture from lines that is a composite itself.

The Composite pattern is useful when the user wants clients to be able to ignore the difference between compositions of objects and individual objects. This pattern represents part-whole hierarchies of objects.

Use of the Composite pattern make the client simpler because they can handle all objects similarly. The Composite pattern makes adding new kinds of components easier.

The Composite pattern is used in my ISAKMP implementation for representing ISAKMP payloads in a message. In the Conduits+ framework, the Mux is a kind of an example of the Composite pattern.

Visitor

The Visitor pattern is more complex than the two above presented ones. It is useful when, in a system, there are several classes that all implement a similar operation for different kinds of data. The Visitor pattern is worth using when an object structure contains many classes with different interfaces and the user wants to make operations that depend on the concrete classes. The Visitor pattern can collect all operations in one class. The class model of the Visitor pattern is presented in figure 4.1.

New operations are easy to add to an object structure with the Visitor pattern. On the other hand, changing the object structure classes requires redefining the interface to all visitors. Thus, the object structure should not change often. The Visitor pattern gathers related operations in one class.

The Visitor pattern is used in the Conduits+ and Jacob frameworks to carry messages. A Visitor carries Messages around in the conduit graph (MessageTransporter) and other kinds of visitor can be used to change the conduit graph (Installer).

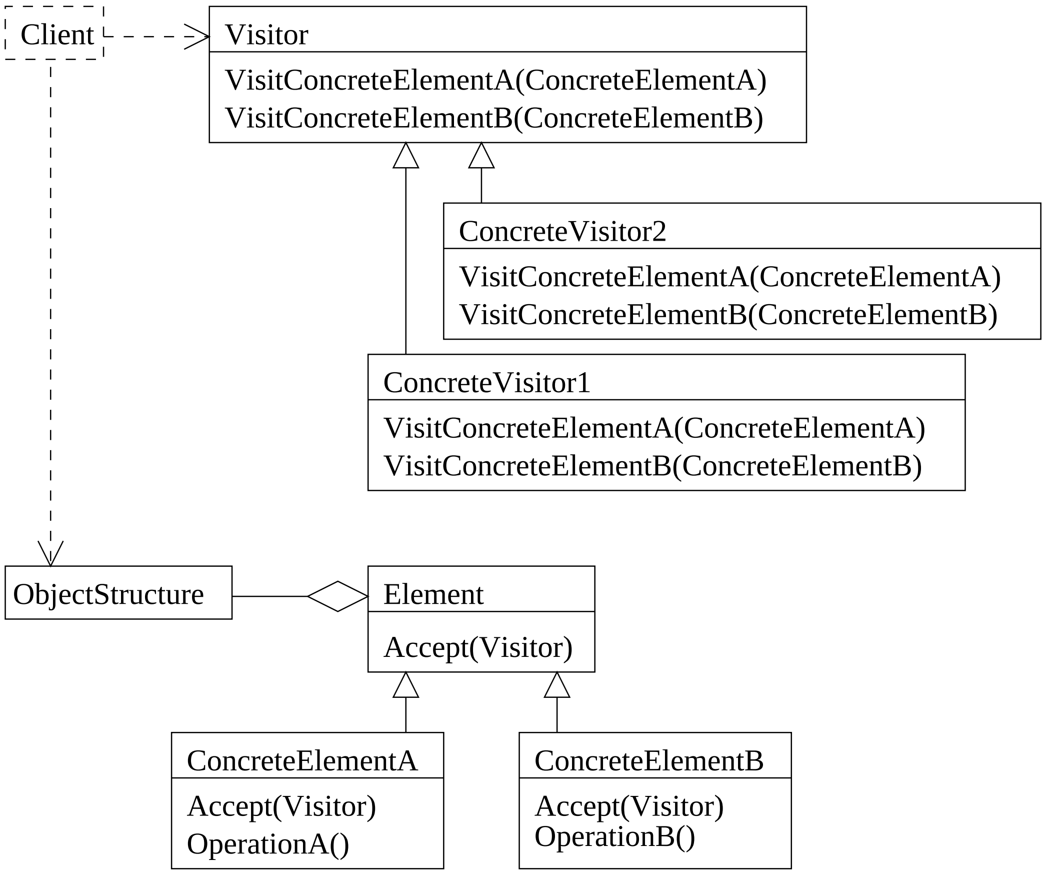

Figure 4.1: Structure of the Visitor pattern

4.2 Protocol Frameworks Examples

This section describes the Conduits+ framework, it inheritors, Jacob and JVOPS, and an another protocol framework. The Conduits+ framework was presented in the Object-Oriented Programming System, Languages and Applications Conference in 1995. The two Conduits+ based framework are Jacob and JVOPS. The first of these two is developed for cryptographic protocols. JVOPS is a Virtual Operating System that implements Conduits+ framework with Java programming language. Java ACE is an object-oriented framework that is not based on Conduits+ framework. All of these frameworks use Design Patterns. Next sections presents these frameworks.

4.2.1 Conduits+

In their paper, Herman Hüni, Ralph Johnson and Robert Engel introduced the Conduits+ framework for developing network protocol software [11]. With this framework the common functionality of protocols can be separated from protocol specific functionality. Other benefits of using Conduits+ framework include the following.

- The protocol specific software parts become simpler and easier to maintain and evolve since the framework provides the infrastructure operations.
- The result is a framework for network protocol software, in other words, a generalized software architecture for communication protocols. Documenting this framework helps communicate the underlying model and how it works.

The Conduits+ framework contains information chunks and four kinds of conduits. Conduits are software components which usually have two sides: sideA and sideB. The information chunks travel through the conduit from one side to the opposite side.

The Conduits

Figure 4.2 shows the four kinds of conduits. The following paragraphs explain how these conduits work.

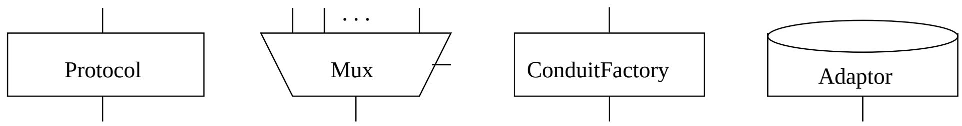

Figure 4.2: Four kinds of conduits

A *Protocol conduit* implements a finite state machine that describes a communication protocol. It remembers the current state of the communication. The Protocol may have counters, timers and storages that are needed for handling information chunks. This conduit has two sides which both are connected to exactly one neighbor conduit.

A *Mux* can have several conduits connected to its sideB. The Mux multiplexes information chunks that arrives from some sideB to the conduit that is connected to its sideA. An information chunk that arrives to the Mux from sideA is demultiplexed to one sideB based on the session key of the information chunk. New conduits can be added to and removed from Mux's sideB dynamically.

When the Mux receives an information chunk from sideA and this chunk has no suitable conduit in some Mux's sideB, the Mux routes the chunk to its default sideB conduit. This default conduit is usually a *ConduitFactory*. Then the conduit factory creates a required session protocol to sideB of the Mux. Now the Mux can handle this information chunk.

An *Adapter conduit* is an interface to some software or hardware that is outside of the conduit framework. The Adapter is connected from its sideA to the Protocol.

The conduits can be connected to any other kind of conduit. This allows a programmer to construct complicated graphs that implement communication protocols.

Design Patterns used in the Conduits+ Framework

The Conduits+ framework uses design patterns for making the framework more reusable and easily extensible. In Conduits+, there are two level of design patterns usage. Some conduits are examples of design patterns: the Mux is a Composite and the Adapter is an example of the Adapter pattern. Design patterns are also used to provide a more flexible solution. In this section, design pattern usage for this purpose is presented.

Conduits+ framework uses Visitor and Singleton patterns. These patterns were described in more detail already in section 4.1.2.

A Mux has no subclasses and thus it works always in the same way. Each Mux has an Accessor which computes a key from an information chunk. In this way, Mux can multiplex information that has different formats. The Strategy pattern has this kind of behavior. It separates the algorithm, or strategy, from the object that uses the algorithm. The Accessor abstracts out the differences of the objects.

Communication protocols are usually realized by a finite state machine. The Protocol conduits are implemented using the State pattern. When a protocol change its state, it replaces its old state-object with a new one. Thus Protocols delegate their behavior to the state objects.

The Command pattern encapsulates a request in common object that can be manipulated in some way. The Command pattern is used when a conduit informs it neighbor that it will be destroyed.

A ConduitFactory adds new Protocols to a Mux. Like Muxes, the ConduitFactory has no subclasses. Each ConduitFactory has prototype conduit which it copies when it needs to create a new one. This is an example of the Prototype pattern.

4.2.2 Jacob

The Java Conduits framework (Jacob) is based on Hüni's, Johnson's and Engel's Conduits+ framework. The goal of Jacob is to allow application-specific secure protocols to be built from components. The protocols are constructed from conduits and they can be combined into a complete protocol stack [18].

Each conduit in Jacob is also a Java Bean. The component is constructed from conduit itself, sides A and B, and usually also from some other objects, depending on the conduit. The sides are inner classes of the Conduit class. When conduits are connected, the internal side object is passed to the neighbor conduit and a message arrives to conduit from a known side. This side identifies the message source. The Java Bean properties allow, for example, setting the initial state to a Session conduit.

In Jacob, a message consist three objects: a message carrier (Message), a message body (Contents) and a message interpreter (Messenger). The message carrier is a Bean event. The Beans can broadcast and receive events notifications. Messages travel through conduit graph using the Java event delivery mechanism. The internal side of the neighbor conduit is an event listener that receives messages from the previous conduit.

Differences between Conduits+ and Jacob

In Jacob, the Conduits+ Protocol conduit is divided to two different conduits: a *Session* and a *Protocol*. The Session implements a basic functional unit and it has basically same function as the protocol conduit of the Conduits+ framework. It implements a finite state machine of a protocol. The Session has several State objects that use the State design pattern like the Protocol in the Conduits+ framework.

The Protocol conduit of the Jacob framework is a kind of meta-conduit that can contain several other conduits. The Protocol has two sides. These sides are usually used for exchanging interprotocol control messages. The actual data is typically delivered between conduits that are located inside some protocol. The Protocol conduit makes the structure of conduit graphs simpler. A protocol can be encapsulate in a Protocol conduit. The Protocol can then be used just like other components to built more complex graphs. Figure 4.3 presents an example of conduits in the Jacob framework and how a protocol can be constructed from these conduits.

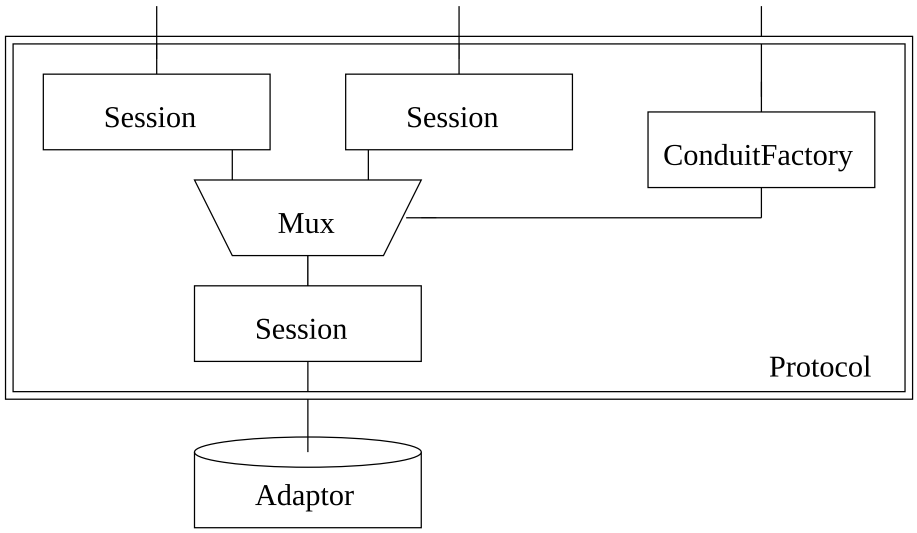

Figure 4.3: An example of a protocol in Jacob framework

While the Conduits+ framework has implemented with the C++ programming language, the Jacob framework is based on the Java programming language. Java has several benefits compared to C++. Java inner classes, Beans properties and Bean events make protocol developing easier. In Java, a framework can be secure and open at the same time. Java allows a programmer to create trusted protocols and combine them with untrusted ones. This is important for cryptographic protocols [18].

Design Patterns in Jacob Framework

Because Jacob is based on the Conduits+ framework, the same design patterns are used in both. The Jacob framework brings a new conduit to the framework. This Protocol conduit is an example of the Composite pattern which was presented in section 4.1.2. From an other point of view, the Protocol can be seen as an example of the Proxy pattern. The Proxy pattern provides a surrogate for another object to control access to it, and the Protocol sides do exactly this. The Proxy can also be responsible for creating and deleting the real subject.

4.2.3 Java Virtual Operating System (JVOPS)

Java Virtual Operating System (JVOPS) has been developed (and is still being developing) by Necsom ltd. JVOPS is a framework that is heavily based on the Conduits+ framework. JVOPS is an environment that allows creation of applications in distributed reliable computing system. JVOPS is developed for telecommunication application programming [15].

There are several other Virtual Operating Systems: CVOPS, OVOPS and OVOPS++. JVOPS and OVOPS++ are based on the Conduits+ framework. OVOPS++ and OVOPS are programmed with the C++ programming language, CVOPS with C programming language and JVOPS with Java programming language.

Compared to Jacob framework, the JVOPS does not bring anything new to Conduits+ framework, while Jacob introduces a number of modifications (these were discussed in section 4.2.2).

4.2.4 ADAPTIVE Communication Environment (ACE)

The ADAPTIVE Communication Environment (ACE) is an object-oriented network programming framework. The ACE is developed for high-performance and real-time communication services and applications. It simplifies the development of network application and services that utilize interprocess communication, event demultiplexing, explicit dynamic linking, and concurrency [21].

Originally, ACE was programmed with the C++ programming language, but nowadays, there is also a Java version of the framework. Java ACE has same functionality than the original version of ACE. The Java ACE has fewer components than the original version because the Java Virtual Machine (JVM) handles many of the portability issues, and Java does not support certain operating system features [13].

ACE and Java ACE are not based on the Conduits+ framework, but it has lot of same kind functionality because they are also communication frameworks.

4.3 Development Process

Many times at the corridor and computer classes at university, I have heard complaining about design: why we have to design, why we cannot just get start programming? Design is an important part of development. If a group of programmers develops a large program, a proper design is necessary or, otherwise, the components would not co-operate. For example, UML class diagrams quickly give a clear picture of a system also to an outsider that has not been a participant of the programming team.

Software development is not an easy task and there is no single process for it. Software development has several factors: what kind of software, how many developers, etc. However, some elements are common in most software development.

Martin Fowler describes four phases in a development process: interception, elaboration, construction and transition [6]. During the interception phase, the project is sold to its customers and its scope is decided. In elaboration, more and detailed information is collected about requirements. The developers decide with the customers what and how they are building and what technology should be used in the development. Risks of the project is also scan in elaboration phase. The result of elaboration is a plan for the construction phase. The construction phase is iterative. Each iteration consists analysis, coding, testing and integration. Also more detailed planning can be made within each iteration. The last phase is transition, which includes beta testing, performance tuning and user training.

In the construction phase, all UML techniques are useful. Some of these techniques are presented in the next section.

4.3.1 Unified Modeling Language (UML)

The Unified Modeling Language (UML) version 1.0 was standardized by OMG (Object Management Group). A new version UML is in the middle of a standardization process. It is a modeling language, not a method. The modeling language is a notation that a method uses to explain design. The UML does not have a notation for processes, which is an important part of a method [6].

The UML defines several diagrams for describing a programming problem.

- The **use case diagram** describes requirements of a system.

- The **class diagram** are most important and regular of the UML diagrams. They are very expressive. There is always a danger to put too much information to one diagram. The class diagrams are presented in more detailed in the next section.
- The **interaction diagram** shows how a group of objects collaborate. There are two kind of interaction diagrams: sequence diagrams and collaboration diagrams. The sequence diagram shows how the methods are called. The collaboration diagram shows static connections between objects and the method calls.
- The **package diagram** shows packages and their dependencies. A package can consist of several classes. The package diagram facilitates comprehending of large systems.
- The **state diagram** shows what happen when an event reaches an object and how the state of the object changes.
- The **activity diagram** is appropriate to describing parallel activities.
- The **deployment diagram** describes physical relationships between software and hardware components.

Class diagram

The class diagram is the most used of UML's diagrams. As an example, figure 4.4 shows a class diagram of the Cookies and CookieStorage of my ISAKMP implementation.

A class diagram describes objects and their static relationships in a system. There are two main types of relationships: associations and subtypes. Associations represent relationships between instances of classes. For example, the CookieStorage stores several Cookies in my ISAKMP implementation. On the other hand, the CookieStorage is a subtype of a HashingCollection.

A class diagram usually also describes attributes and operations of a class. For example, the Cookie class has three attributes: `initiatorCookie`, `responderCookie` and `next`. Operations describe actual functions that a class can carry out. For example, the CookieStorage can find a cookie with the `findCookie` operation.

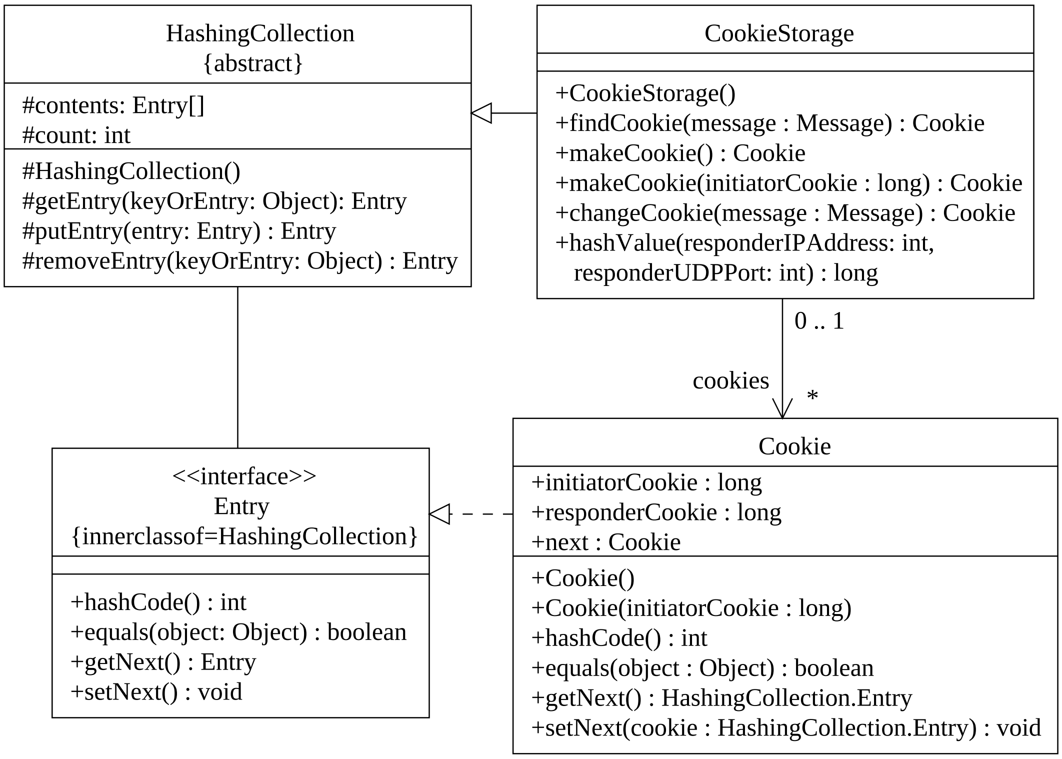

Figure 4.4: An example of class diagram: Cookies in the ISAKMP Protocol

4.3.2 Case: Cookie Handling in ISAKMP

A Conduits+ graph of a protocol looks very simple. A protocol can be constructed from conduits, but understanding how these conduits work together is not very easy. For an unexperienced programmer, like I am, a picture can explain how things should be done. Besides, just to start programming is hard but a good design helps understanding the problem and how it should be implemented.

In my developing process, there were two rounds of three phases that actually consists of only the construction phase of development. First, I made a draft for a conduit model of ISAKMP. Based on this conduit model, I made the first class diagrams that contained mainly the conduit classes and their subclasses. The messages and their handling were not almost at all modeled in these first diagrams. Then I made a crude sequence diagram about handling cookies in an ISAKMP message exchange. A part of this diagram is shown in figure 4.5.

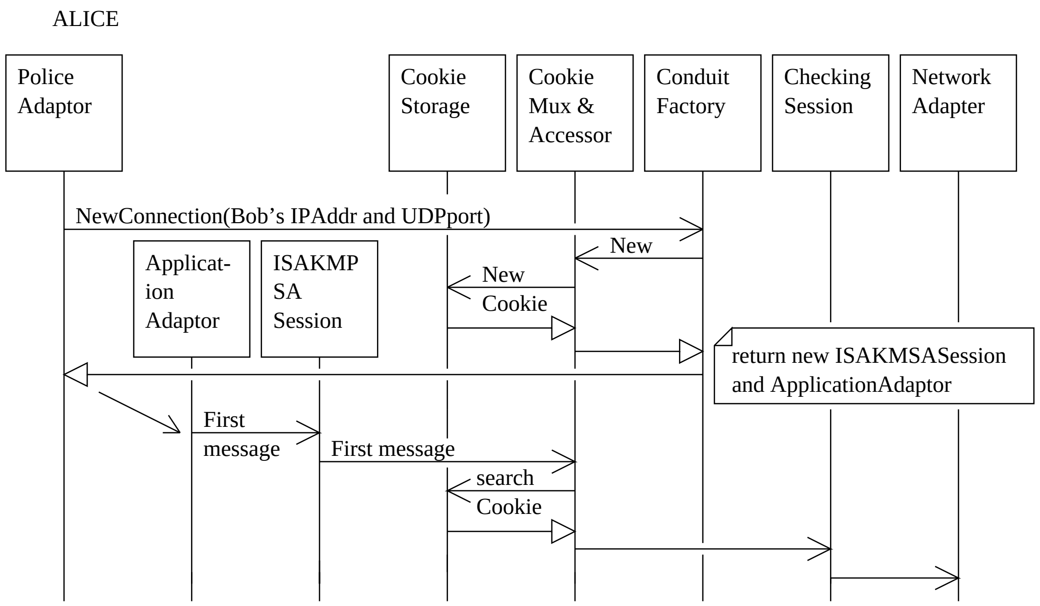

Figure 4.5: Sequence diagram of first message sending

After this first development round I had a clear picture what I was doing.

In the second round, these three phases were performed again, and some modifications and specifications were made, especially for message handling, to the conduit model and the class diagrams.

For an inexperienced programmer, the use of a case tool may make programming easier. In my work I used the ClassDesigner of Cayenne Software. In this program, there is a code generator for C++. The code generator makes all classes and declarations from class model. The programmer gets a skeleton from all classes and only the substance must be implemented.

In Telecommunication Software and Multimedia Laboratory, a Java code generator was added to the ClassDesigner [10]. I used this code generator to get out class skeletons. Then I implemented the contents to classes. Every time I wanted to change, for example, a name or a type of a variable I had to change it first into my model and then generate the code. This way the documentation of my implementation is in good condition and up to date all the time.

4.4 Summary

In this chapter the design tools were presented. These tools were used in the implementation part of this work. The next chapter describes in more details the Conduits+ model and class diagrams of my ISAKMP implementation. In addition, the chapter after that presents the implementation.

Chapter 5

ISAKMP Architecture in O-O Terms

The construction phase of the ISAKMP framework was planned to be realized in three phases. In the first phase, a basic conduits+ model of the ISAKMP protocol was built. The second phase added message encoding and decoding between binary and object formats and the Base Exchange was completed. The last phase, all other exchanges are planned be implemented using experience gained in phase two. The actual realization of phase three is beyond the scope of this thesis.

The design results of these phases are Conduits+ graphs, class diagrams and sequence diagrams. The implementation of ISAKMP itself is also made in three phases. First, the handling of cookies and the basic ISAKMP has been planned and implemented. The second phase consists has a plan of the Base Exchange and its implementation. The last phase is planned to consists of implementation of all other exchanges. The following sections describe the architecture of the ISAKMP.

5.1 Base Functionality and Cookie Handling

In this section, the Conduits+ graph of the ISAKMP is presented. The ISAKMP is presented as a Protocol conduit that is connected to two Adaptor conduits: the PolicyAdaptor and the NetworkAdaptor. The PolicyAdaptor is used by the Security Policy Daemon (SPD) that decides what connections are allowed and what not and what kind of connections can be accepted. The NetworkAdaptor presents the network. Figure 5.1 presents the basic

Conduits+ graph of the ISAKMP. The dashed line presents conduits that are dynamically added to the graph. Later in this section, UML models of the ISAKMP are also introduced.

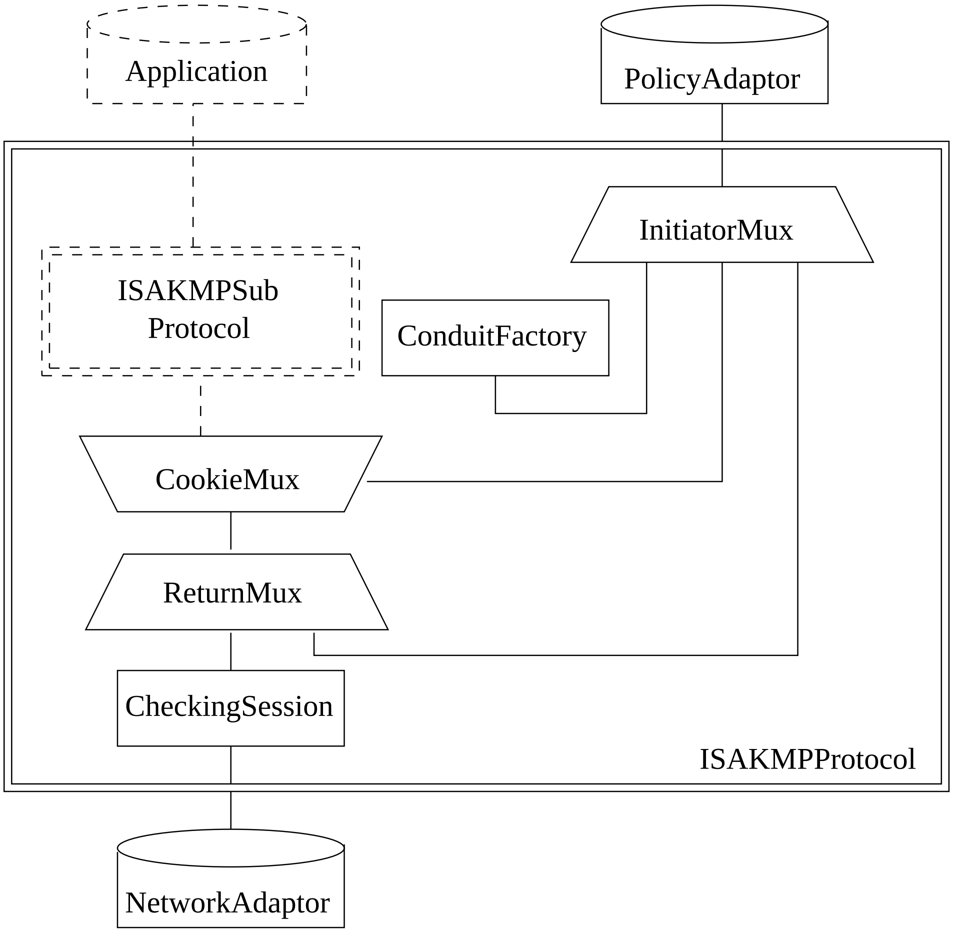

Figure 5.1: A Conduits+ graph of the ISAKMP

5.1.1 The Conduits+ graph of the ISAKMP

When Alice wants a connection to Bob, she clicks a button in her application. The application asks from the Security Policy Daemon (the PolicyAdaptor) if the connection is allowed and under which conditions. The PolicyAdaptor sends an install message to the ConduitFactory. The responder information and exchange type are given in this message. The responder information is need for Cookie generation.

Each Cookie consists of two parts: an initiatorCookie and a responderCookie. They both are hashes from initiator's and responder's IP addresses and UDP ports, random data and current time and date. The ConduitFactory ask from CookieMux for new key, which also is the Cookie. The key is generated in the CookieStorage, which is not a conduit but still an essential part of the ISAKMP. In the first message, the Cookie has only the initiatorCookie part set. After that, the ConduitFactory attaches an ISAKMPSubProtocol to a newly created side of the CookieMux. The type of the ISAKMPSubProtocol depends on the exchange type. The ISAKMPSubProtocol is also connected to an ApplicationAdaptor.

Now Alice (or her application) can start the actual ISAKMP message exchange with Bob. The goal is to establish an ISAKMP Security Association and an application Security Association. In first phase of ISAKMP, the ISAKMP SA is established for protecting further negotiation. Alice's ISAKMP-SubProtocol forms the first message that consists of an ISAKMP Security Association Payload. This payload has information about what kind of connections are allowed according Alice's Security Policy. The message travels through the SubProtocol and arrives to the CookieMux. The CookieMux adds the Cookie to the message. The Cookie is asked from the CookieStorage. The CheckingSession calculates the length of message and adds it and the Major and Minor Versions to the message. Then the message continues its trip through the network (NetworkAdaptor) to Bob.

Bob's ISAKMP protocol gets a message from the NetworkAdaptor. First, the length and version numbers are checked. If there is some mistake, a notification message is sent to Alice. The message travels through the Conduits+ graph to the CookieMux. The CookieMux asks from the CookieStorage if the Cookie is known. Each connection has the own Cookie. In this case, the received message is the first message from Alice and its Cookie is not stored in the CookieStorage. Therefore, the message is sent to the PolicyAdaptor. If the connection is allowed, similar with Alice's ISAKMP, an install message is sent to the ConduitFactory. The ConduitFactory asks for a side from the CookieMux. The CookieMux asks the Cookie from the CookieStorage. Now the CookieStorage notices that this is a new connection. The Cookie consists of only the initiatorCookie generated by Alice. The CookieStorage generates a responderCookie and adds it to the Cookie. The Cookie is then saved in the CookieStorage and a new ISAKMPSubProtocol is created. Alice's message is send to Bob's ApplicationAdaptor through the new ISAKMPSubProtocol.

Bob's communication application alarm that new connection request has arrived. Bob answers to the message. The ISAKMPSubProtocol chooses a suitable protection from Alice's suggestions. Then the message travels to the CookieMux that adds to it the Cookie from the CookieStorage. The message is send to Alice.

The new message comes in to Alice's ISAKMP. The length and versions are checked. The CookieMux asks form the CookieStorage if the connection is known. The CookieStorage notices that the Cookie has changed, but using the initiatorCookie field it knows that there is a corresponding ISAKMP-SubProtocol for this message. The CookieStorage changes the Cookie in the storage and saves it. Then the CookieStorage tells to the CookieMux to which connection (side) the message should be send. The CookieMux sent the message to the ISAKMPSubProtocol that handles it. Now the Cookie is stabile and messages travel through the graph without passing through the PolicyAdaptor anymore.

5.1.2 Messages

Messages in the Jacob framework consists of three parts: Message contents, Messenger and Out-of-Band. The message contents may be any kind of data, for example, a binary stream or some hierarchy of objects. The messenger knows how to handle message contents. The Out-of-Band consists of information that may be needed elsewhere in the Conduits+ graph. Usually in Jacob, each Session conduit demands two Messengers: one for messages that travel from sideA to sideB and one for the other direction. The Session, or more precisely some of its states, changes the Messenger for the next session that is reached when message travels through the graph. In other words, the Session must know where the message is going and what kind of messenger it needs there.

Figure 5.2 presents the Messengers of ISAKMP. The Messenger class, the BufferMessenger and InstallMessenger are parts of the conduits model. The Messenger is an interface. The BufferMessenger allows programmers to handle binary format messages as arrays of integers, octets, bytes etc. The InstallMessenger is also an interface. It is used when new sides must be added to a Mux. The ConduitFactory gets necessary information from the InstallMessenger.

When Alice wants to create a connection with Bob or Bob gets the first message from the network, a message is send to the PolicyAdaptor. If Alice is sending the first message, the PolicyAdaptor sends a message with an

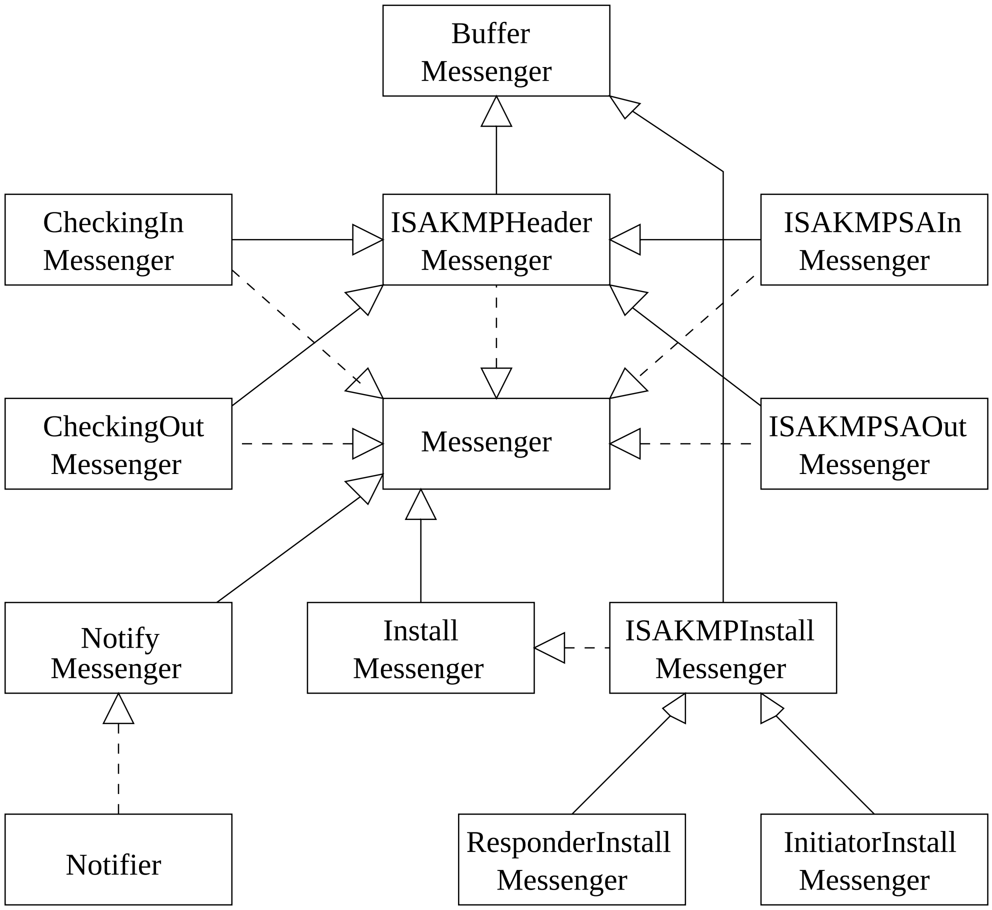

Figure 5.2: UML model of ISAKMP Messengers

InitiatorInstallMessenger to the ConduitFactory. The ConduitFactory asks from the CookieMux the side to which a new ISAKMPSubProtocol should be connected. The CookieAccessor in the CookieMux asks for a new key, that is a Cookie, from the CookieStorage. Then the ConduitFactory informs the PolicyAdaptor with a Notifier that a new ISAKMPSubProtocol is connected to the CookieMux. The other side of the ISAKMPSubProtocol is connected to an ApplicationAdaptor.

When Bob's PolicyAdaptor orders the ConduitFactory to make a new ISAKMPSubProtocol for the new connection with Alice, it sends a ResponderInstallMessenger to the ConduitFactory. The ConduitFactory asks for a new side from the CookieMux. The CookieMux asks for a key (Cookie) from the CookieAccessor. The CookieAccessor notices that the message has only the initiatorCookie and the CookieStorage has to add the responderCookie in to the Cookie. When a new side is added in Cookie Mux and a new ISAKMP-SubProtocol is attached to this side, the ConduitFactory sends a Notifier to the PolicyAdaptor. This Notifier informs the PolicyAdaptor that the new ISAKMPSubProtocol is connected to the CookieMux and a new ApplicationAdaptor, and Alice's message can be send to the ISAKMPSubProtocol.

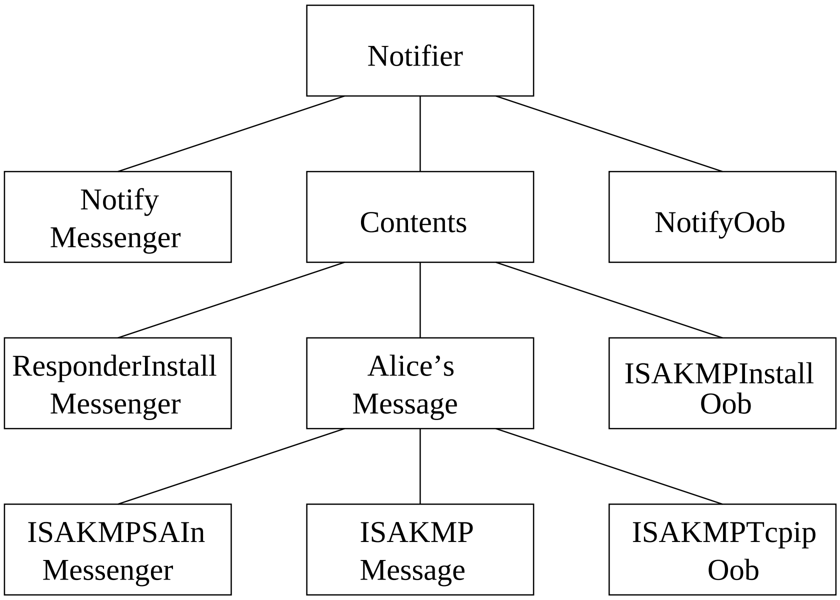

Figure 5.3: The notify message send by the ConduitFactory

Bob's PolicyAdaptor gets a message from the ConduitFactory that a new ISAKMPSubProtocol is connected. Figure 5.3 shows the structure of this message. The PolicyAdaptor must then forward Alice's original message to the CookieMux that can now demultiplex it to the correct ISAKMP-SubProtocol. Alice's message has traveled through the ConduitFactory to the CookieStorage and back. When the PolicyAdaptor sends a message to the ConduitFactory, Alice's message was the contents of that message. The ISAKMPInstallOob consists information for the ConduitFactory: there are the sides to which the new conduit should be connected and the key for the Mux's side. In this case, the key is null and the CookieMux has to asks the key from its Accessor. After a new ISAKMPSubProtocol is connected, the ConduitFactory sends a Notifier to the PolicyAdaptor. The contents of this Notifier is the message that is send to the ConduitFactory, i.e., the ResponderInstallMessage. The PolicyAdaptor takes Alice's original message out of this structure and sends it through the InitiatorMux, the ReturnMux and the CookieMux to the ISAKMPSubProtocol (see figure 5.1).

5.2 Base Exchange

Figure 5.4 presents the Conduits+ graph of one type of ISAKMPSubProtocol: the BaseExchangeSubProtocol. This figure contains also the class diagram of ISAKMPSubProtocols.

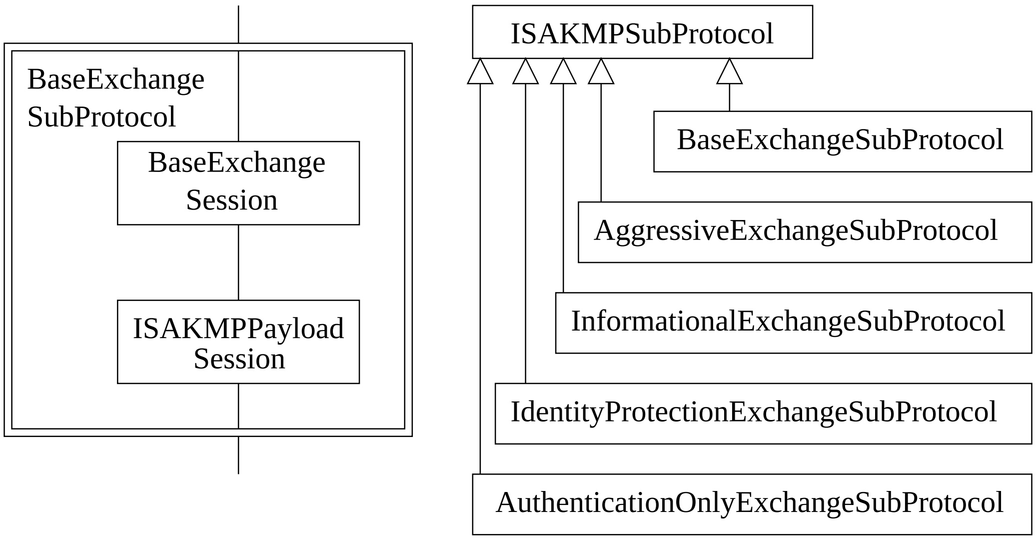

Figure 5.4: The class diagram of ISAKMPSubProtocols and the Conduits+ graph of BaseExchangeSubProtocol

The Conduits+ graph of the BaseExchangeSubProtocol consists of only two Session conduits: BaseExchangeSession and ISAKMPPayloadSession. The ISAKMPPayloadSession divides the binary contents of messages to payload objects. Until here, the message is in binary format and messengers handle it with help of the BufferMessenger of Jacob framework. The ISAKMPPayloadSession does not create any object representing for the ISAKMP Header. The necessary information from the header is moved to an Out-of-Band object. The Initiator and Responder Cookies, the Exchange Type, the Length and the Versions are not needed in the BaseExchangeSession.

The BaseExchangeSession implements the finite state machine of the Base Exchange. This finite state machine knows which message should send and what to do to with received messages. Figure 5.5 presents the finite state machine of the BaseExchangeSession. First, the state machine moves to the IDLE state right after the ISAKMP protocol is activated. Then it is ready for sending or receiving the first message of the Base Exchange. When a message is send, there might come two kinds of response. The peer can send a notify message to inform that something was not correct in the sent message. After the notify message, the state machine moves back to the IDLE state. If the received message is not a notify message, it should be a response. Then the state machine checks that the message is correct and send next message to peer according to the Base Exchange specification. If everything was not right, the state machine sends a notify message to the peer and moves back to the IDLE state.

There is an exchange type for notify messages. This Informational Exchange requires no answer and it can be send during any other exchange. If the Informational Exchange carries a Notification Payload it is routed to the exchange which sent the original message. For example, if Alice send a message that has an old Nonce and Bob notice this, Bob should send a notify message to Alice. Then Alice's BaseExchangeSession gets a message that contains notification that the nonce was invalid. The state machine of the BaseExchangeSession moves to the idle state, and Alice has to start from the beginning. The Informational Exchange can also carry a Deletion Payload. If this is the case, the message is routed to the InformationalExchangeSubProtocol.

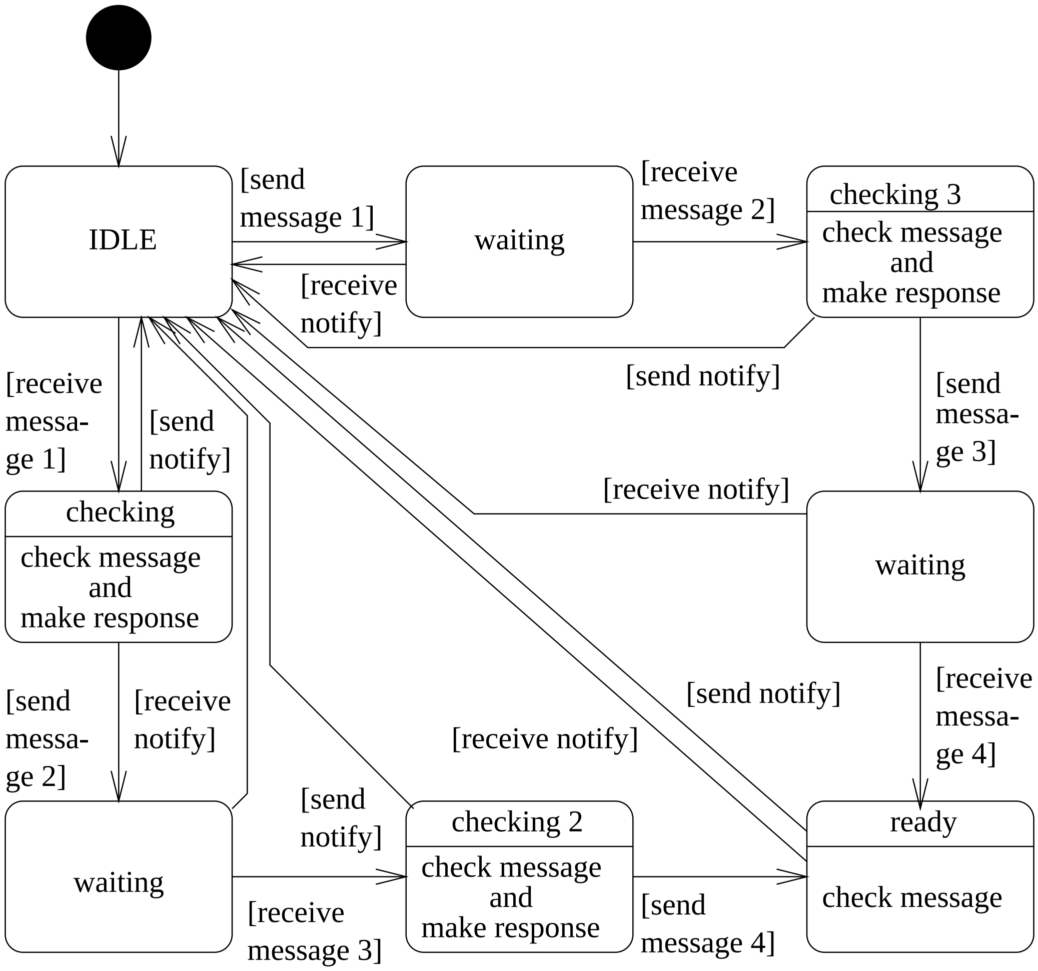

Figure 5.5: Finite state machine of Base Exchange

Chapter 6

Implementation

In this work, the ISAKMP protocol was planned to be implemented in three phases. First, the basic functionality was implemented. This implementation consists of Cookie handling and creation of new sub protocols for different exchanges. In the second phase, one of ISAKMPExchangeSubProtocols was implemented. In the future, the rest of these sub protocols are implemented. That part of the development is beyond the scope of this thesis.

6.1 Base Functionality of ISAKMP

In figure 5.1, the Conduits+ graph of ISAKMP was presented. Actually, this figure also shows most of the classes in ISAKMP. Figure 6.1 shows the class diagram of the conduits in the ISAKMP. There are three Adaptors in my implementation. The NetworkAdaptor presents the underlying network that consists of protocols like UDP and IP and the hardware. The ApplicationAdaptor is an application that wants a secure connection with some service or other application through a network. The PolicyAdaptor is the Security Policy Daemon that know with who and what kinds of connections are allowed. The CheckingSession checks from incoming messages that the length of ISAKMP message corresponds to the value in length field of the ISAKMP Header. It also checks that the version of incoming message is same than the version of this implementation. The CheckingSession adds the length and version numbers to the outgoing message.

The ISAKMPSASession is a temporary class in the first phase of my implementation. The ISAKMP is a protocol that encapsulates all the other conduits except adaptors. It also encapsulates subprotocols for connections. The subprotocol can be either BaseExchangeSubProtocol, IdentityProtectionExchangeSubProtocol, AggressiveExchangeSubProtocol, AuthenticationOnly-ExchangeSubProtocol or InformationalExchangeSubProtocol. All of these are subclasses of the ISAKMPSubProtocol.

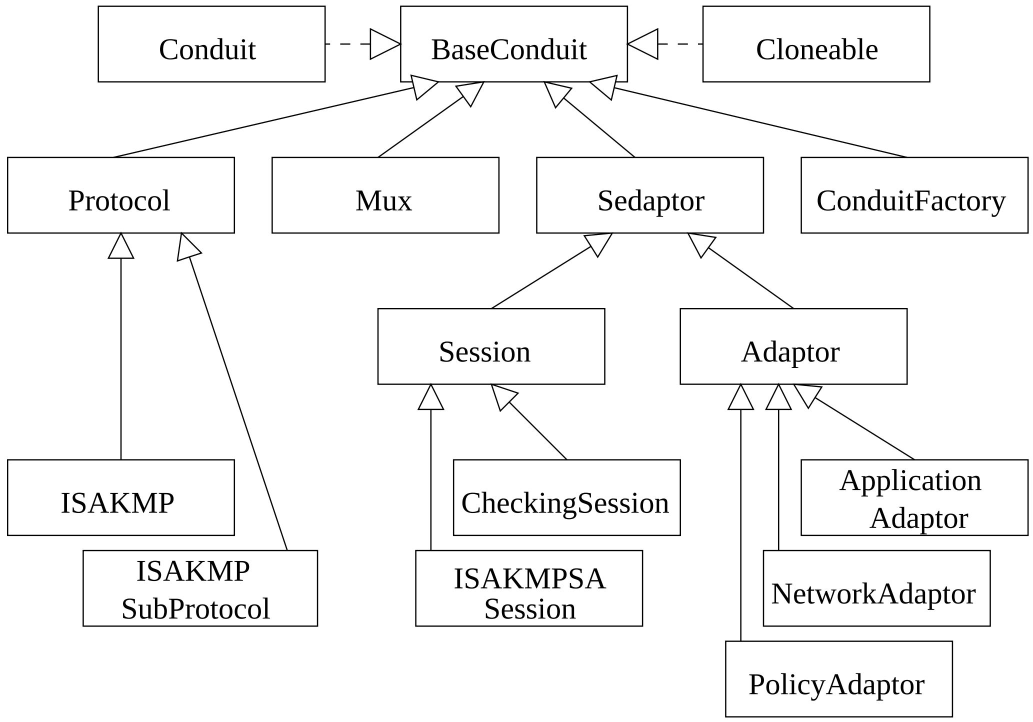

Figure 6.1: UML model of Conduits+ classes in ISAKMP

Each Session has its own State class that implements the finite state machine of the Session. For example, the CheckingSession has a CheckingState class that contains all functionality of the Session.

All Muxes are of the same kind. The Accessor makes the difference in their behavior. The Accessor calculates the keys where the demultiplexing is based on. This key is usually some part of a message, but it could also be a constant. For example, in this ISAKMP implementation, the ReturnAccessor always returns "1" when the key is asked and the Mux always demultiples the message to the CheckingSession (figure 5.1). For simplicity, I call the Mux by their Accessor's name. For example, the CookieMux means the Mux whose Accessor is the CookieAccessor.

How the Conduits+ graph can add dynamically new Protocols or Sessions to a Mux? The ConduitFactory can do this but how it knows what kind of Protocol or Session should be connected to the Mux? Normally a Conduit-Factory has a prototype of a Protocol or a Session. It clones the prototype and connect that new Protocol or Session to new side of the Mux. However, the prototype can also be supplied in the InstallMessage that informs the ConduitFactory that new Session is required.

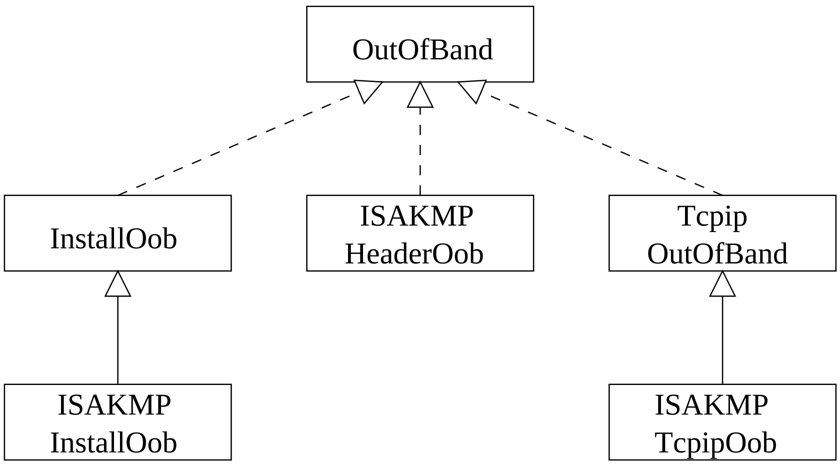

Figure 6.2: Out of Band classes

Rest of the classes in this phase are related to Messages and Cookies. The UML model of the Messengers is presented in figure 5.2. The Message consists three parts from which the Messenger is one. Other two are the contents of the Message and the Out-of-Band. The Out-of-Band classes are presented in figure 6.2. The ISAKMPHeaderOob contains a variable for all fields in the ISAKMP Header and methods for setting and getting these variables. The ISAKMPSubProtocol (and in the first phase the ISAKMPSASession) uses this class to store information from the header after the message is changed from a bit stream to objects. A message received from the network contains an ISAKMPTcpipOob. This Out-of-Band has information about IP address numbers and UDP port numbers that are needed in the Cookie generation. The ISAKMPInstallOob informs the ConduitFactory what kind of ISAKMPSubProtocol should connect to the CookieMux.

The CookieStorage stores Cookies that are used in communication, and it also makes these Cookies. The CookieStorage extends a HashingCollection class. The Cookies are Entries of this HashingCollection. The HashingCollection works like a hat that contains pieces of papers. When a programmer wants to find a certain paper, he asks the hat if it has that piece of paper. If that paper is in the hat, it gives the paper to the programmer.

6.2 Adding a New Protocol to the CookieMux

Alice can have connections with several people. Each connection has its own Security Association that protects it. When Alice wants to create a new connection with Bob, a new ISAKMPSubProtocol is attached to the CookieMux by the ConduitFactory. How this can be done with Jacob, was not clear because Jacob is currently not documented very well.

Figure 6.3 shows how the ConduitFactory attaches a new ISAKMPSubProtocol to the CookieMux. First, the `facture` method of the ConduitFactory is called. It asks for prototype from the messenger. In my first implementation, the initiatorInstallMessenger returned null and the ConduitFactory used the prototype conduit that had been given to it when ISAKMP was built. In the actual implementation, the initiatorInstallMessenger returns an ISAKMP-SubProtocol type. The type of the ISAKMPSubProtocol is gotten from the Exchange Type field of the ISAKMP Header. After getting prototype, the ConduitFactory asks for endpoints from the messenger. These endpoints are the sides where to connect the cloned prototype. Then the ConduitFactory clones the prototype.

After creating the clone, the ConduitFactory asks for a new sideB from the CookieMux. The CookieMux needs a key for the new side. It asks that key from the CookieAccessor. The CookieAccessor notices that a new Cookie is needed. It asks the responder's IP address and UDP port from messenger and gives them to the CookieStorage. The CookieStorage makes the new Cookie. Based on this Cookie, the CookieMux makes a new side and gives it to the ConduitFactory.

Figure 6.3 does not show how the sides are referred. Each side has an `attach` method, that connects a side to that side. The ConduitFactory calls the `attach` method of the side that it has got from the CookieMux. This side is connected to the side of the cloned prototype conduit. Now the new ISAKMPSubProtocol is connected to the CookieMux.

The ConduitFactory sends a notification to the PolicyAdaptor indicating that the request has been completed. The structure of this message was shown in figure 5.3.

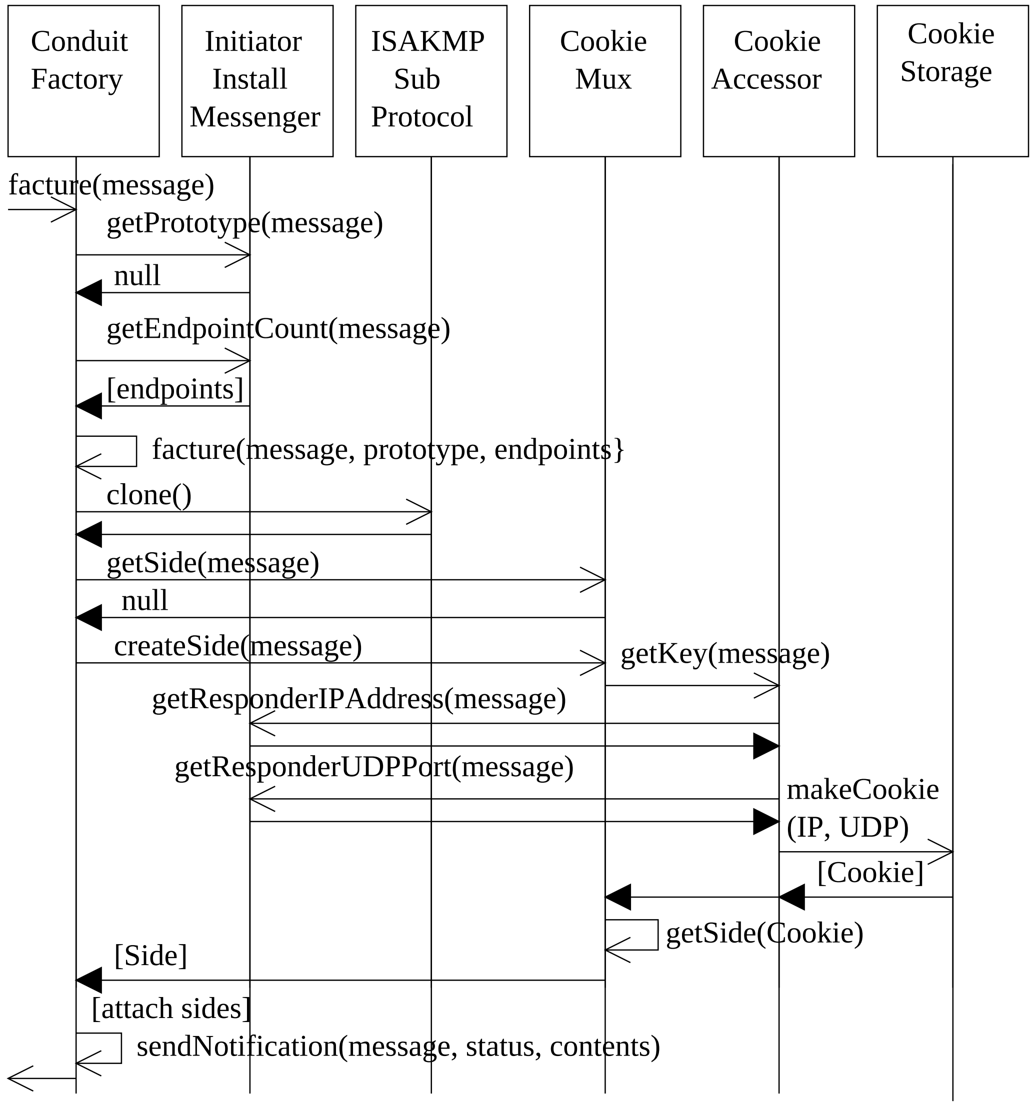

Figure 6.3: Creation of a new Session

6.3 Base Exchange

The BaseExchangeSubProtocol is simple. It has only two Sessions: BaseExchangeSession and ISAKMPPayloadSession. Both of these have a corresponding State object. Figure 6.4 presents the UML model of the classes of the BaseExchangeSubProtocol.

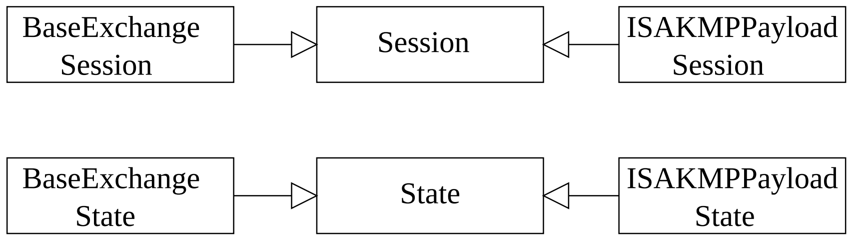

Figure 6.4: UML model of Base Exchange

The ISAKMPPayloadSession uses the ISAKMPSAInMessenger and the ISAKMPSAOutMessenger to handle the message. This Session changes the messages to object form. Each payload has its own class, and the message is formed from these classes. The BaseExchangeSession has own Messengers for incoming and outgoing messages. The BaseExchangeSession implements the finite state machine that was presented in figure 5.5.

Chapter 7

Evaluation

7.1 Suitability of the Jacob framework

The Jacob framework was designed for developing cryptographic protocols. The ISAKMP protocol is an authentication protocol which is one type of cryptographic protocols. The idea of frameworks is that a developer of a protocol does not need to know how everything works. In a Conduits+ type of framework, the developer does not need to know all about the framework to develop a working implementation. The Jacob framework is currently poorly documented which make the understanding of it difficult. Especially, the message handling and the using of the ConduitFactory was hard for me. A sequence diagram could make understanding of these easier. Such a diagram should show how the ConduitFactory makes a new session or protocol to the Mux and how the messages travel through a Conduits+ graph.

7.2 Suggestions for Improvement

The Jacob framework does not currently work similarly with the Conduits+ specification. When a message arrives to a Mux that has no suitable side for it, the message is send to Mux's default conduit that is usually a Conduit-Factory. The ConduitFactory creates a new side to the Mux and connects the required Session or Protocol to it. After this, the ConduitFactory of the Jacob framework sends the original message back to Muxes sideA. According the Conduits+ framework specification, after the ConduitFactory has establish a new protocol to some sideB of the Mux, the original message is send to this new protocol. The way this version of Jacob work makes the conduits graphs more complex. For example, the correct Conduits+ graph of the ISAKMP is presented in figure 7.1. The InitiatorMux and the ReturnMux are unnecessary if the Jacob framework works correctly.

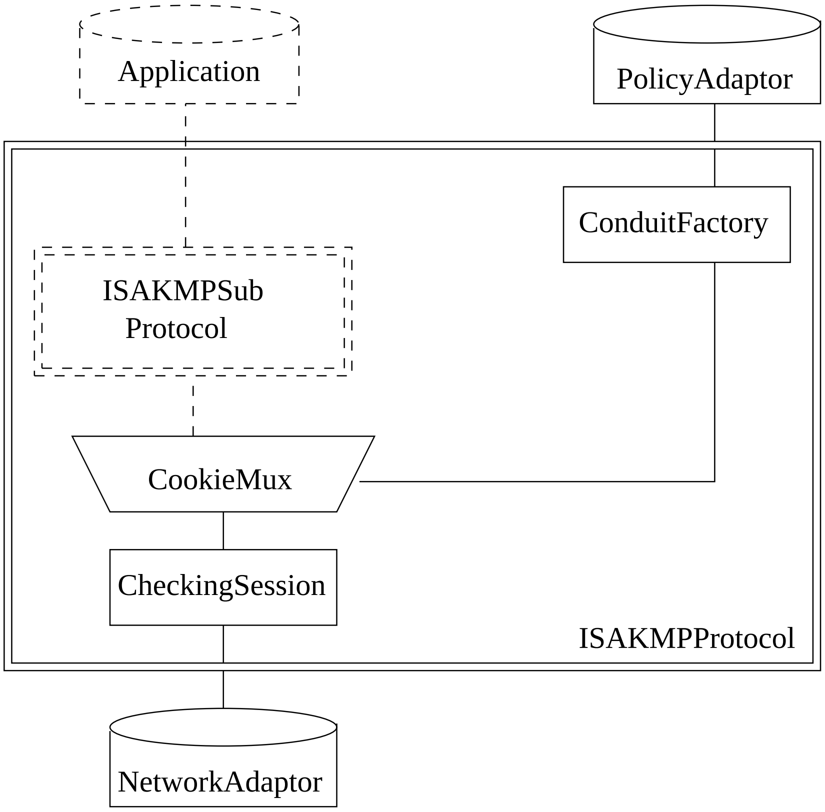

Figure 7.1: Correct Conduits+ model of ISAKMP

7.3 Future Work

The most demanding task in the future is to integrate this ISAKMP implementation to the other prototypes produced in the Telecommunication Software Security Architecture (TeSSA) project. In this project, the goal is to develop secure ways to handling information and authorization in open networks.

This future work is planned to be implemented in three phases. In first phase, the current ISAKMP prototype will be completed by implementing all of the ISAKMP Exchanges. Then the IKE key exchange protocol will be implemented [9]. In the last phase, an exchange for certificates and their validity checking will be developed and implemented. In the other subprojects of TeSSA, a prototype of SPKI certificates and another prototype for their storage in the Domain Name System (DNS) are being developed. The ISAKMP prototype will be integrated with these in the last phase of development.

Chapter 8

Conclusions

The goal of my work was to implement the Cookie handling and the Base Exchange of the ISAKMP protocol with help of the Jacob framework. In the future, other exchanges will be implemented.

The idea of frameworks in general is good. To separate common functionality of protocol from protocol specific functionality is a practical solution. A framework makes protocol implementation more effective once the programmer knows the framework. Understanding of a framework depends on how good the documentation of the framework is. For example, a programmer must read source codes of the Jacob framework if he/she wants understand how Jacob works. The Jacob framework consists of two parts: conduits and messages that travels through the conduits. The messages and their functions are harder to understand than the conduits and their connections.

The Jacob framework is suitable for developing complex cryptographic protocols like the ISAKMP protocol. Jacob makes adding new exchanges to ISAKMP easy. For example, the existing source code needs no changes when other ISAKMP exchanges will be added to it.

The current version of Jacob has an error. The ConduitFactory and the Mux do not work correctly. This error makes the Conduits+ graph of ISAKMP more complex and understanding how a message travels through the graph even more difficult.

In this work, I used Cayenne Software's ClassDesigner to design class diagrams and a code generator developed at Helsinki University of Technology. The code generator saves me from "fear of a white paper" and helps me to start programming. On the other hand, code generating forces me to keep the class diagrams up to date. Every time the class diagram is changed the code must be regenerated. After generation, some part of the code may not be in their own places. The code generator moved the code to “unused code” section of a class.

In the future, this ISAKMP implementation will be integrated with other prototypes of Telecommunication Software Security Architecture (TeSSA) project. In our security architecture, authorization is carried out with SPKI certificates. The certificates are storaged in Domain Name System (DNS). The ISAKMP protocol will be used for exchanging certificates and their validation messages.

Bibliography

[1] E. Amoroso. Fundamentals of Computer Security Technology. Prentice-Hall, 1994

[2] R. Atkinson. IP Authentication Header. RFC 1826. 1995

[3] R. Atkinson. IP Encapsulating Security Payload (ESP). RFC 1827. 1995

[4] Douglas E. Comer. Internetworking with TCP/IP Volume I: Principles, Protocols, and Architecture, Prentice Hall, 1995

[5] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, T. Ylönen. Simple Public Key Certificate. Internet Draft, work in progress, March 1998. URL: http://www.ietf.org/internet-drafts/draft-ietf-spki-cert-structure-05.txt

[6] M. Fowler. UML Distilled Applying the Standard Object Modeling Language. Addison-Wesley, 1997

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995

[8] N. Haller, R. Atkinson. On Internet Authentication. RFC 1704, 1994

[9] D. Harkins, D. Carrel. The Internet Key Exchange, Internet Draft, work in progress, July 1998, URL: http://www.ietf.org/internet-drafts/draft-ietf-ipsec-isakmp-oakley-8.txt

[10] E. Heimonen, P. Nikander. TCM Java Code Generator for Cayennesoft's ClassDesigner. URL: http://www.tcm.hut.fi/Research/TeSSA/Jacob/codegen.html, 1998

[11] H. Hüni, R. Johnson, R. Engel, A Framework for Network Protocol Software. Object Oriented Programming System, Languages and Applications Conference Proceedings (OOPSLA'95), ACM Press 1995

[12] ITU-T Recommendation X.509, Information Technology - Open Systems Interconnection - The Directory: Authentication Framework. 1993

[13] P. Jain. The ADAPTIVE Communication Environment (ACE). An Object-Oriented Network Programming Toolkit in Java. URL: http://siesta.cs.wustl.edu/ pjain/java/ace/ACE.html

[14] D. Maughan, M. Schertler, M. Schneider, J. Turner. Internet Security Association and Key Management Protocol (ISAKMP). Internet Draft, work in progress, July 1998, URL: http://www.ietf.org/internet-drafts/draft-ietf-ipsec-isakmp-10.txt

[15] Necsom ltd. JVOPS & Conduits+ Framework. Slide Show. 1998

[16] B. C. Neuman. Proxy-Based Authorization and Accounting for Distributed Systems. Proceedings of the 13th International Conference on Distrubuted Computing System, 1993

[17] P. Nikander. Modelling of Cryptographic Protocols - A Concurrency Perspective. Licenciate's Thesis, Helsinki University of Technology, 1997.

[18] P. Nikander, A. Karila, A Java Beans Component Architecture for Cryptographic Protocols

[19] J. Partanen, P. Nikander. Adding SPKI Certificates to JDK 1.2. Proceedings of the Third Nordic Workshop on Secure IT System, 1998

[20] Derrel Piper, The Internet IP Security Domain of Interpretation for ISAKMP, Internet Draft, work in progress, URL: http://www.ietf.org/internet-drafts/draft-ietf-ipsec-ipsec-doi-10.txt

[21] D. C. Schmidt. The ADAPTIVE Communication Environment (ACE). An Object-Oriented Network Programming Toolkit in C++. URL: http://www.cs.wustl.edu/ schmidt/ACE.html. 1998

[22] B. Schneier. Applied Cryptography. Second edition, Wiley, 1996

[23] Andrew S. Tanenbaum: Modern Operating Systems, Prentice-Hall, 1992