Imperial College London

Department of Computing

A Secure Session-Based Distributed Programming Language

by Nuno Alves

Supervisor:
Dr. Nobuko Yoshida

Second Marker:
Prof. Susan Eisenbach

Submitted in partial fulfilment of the requirements for the MSc Degree in Advanced Computing of Imperial College London

September 2009

Abstract

Since type-checking is not enough for a fully secure language, there is a need for a security protocol embedded in that same programming language. Thereby, something like peer authentication and key negotiation for an encrypted communication could be implemented but nowadays there are more secure and stronger protocols.

Session Java (SJ) Framework, which is a new framework for session-based distributed programming, could then be modified so it includes a security protocol in order to increase its security with low overhead. In order to maximise the security of SJ, a modified TLS connection is established on each session with authentication based on the Secure Remote Password (SRP) protocol. The most delicate part of the security lies on the delegation, since there could be several ways to achieve security, without certainty of which one is better. Thus, this topic is carefully studied and implemented.

Finally, all modified delegation protocols are modelled into π-Calculus and analysed in order to prove their correctness properties.

Acknowledgements

Firstly, I would like to thank my supervisor Nobuko Yoshida for all her support and guidance throughout the project. Without her precious comments, this project would not have been the same. I would also like to thank Pierre-Malo Deniélou and Raymond Hu for their help in some technical parts of the project.

I appreciate all the support from my friends and I would specially like to mention Fábio Neves for his suggestion on a specific part of the project and Daniel Pina, Chris Hamann and Margarida Sorribas for their support and motivation over the last months. Special thanks to Ana Guerreiro for her revision and suggestions on the report.

And last but not the least, I would like to thank my parents for all their support in everything. Without them none of this would have been possible.

Contents

1 Introduction 1

2 Session Java 3
- 2.1 Basic Functionalities of SJ . . . 3
  - 2.1.1 Session Delegation . . . 3
  - 2.1.2 noalias type . . . 3
  - 2.1.3 SJ Framework Layers . . . 4
  - 2.1.4 Benchmarking . . . 5
- 2.2 Comparison with other parallel and distributed languages . . . 5
  - 2.2.1 Sing# and Crypto F# . . . 5
  - 2.2.2 XMem, X10, Chapel and Fortress . . . 5
  - 2.2.3 Summary . . . 5

3 Cryptography and Cryptographic Protocols 7
- 3.1 Symmetric Cryptography . . . 7
  - 3.1.1 Data Encryption Standard . . . 7
  - 3.1.2 Advanced Encryption Standard . . . 8
- 3.2 Public-Key Cryptography . . . 8
  - 3.2.1 RSA . . . 9
- 3.3 Cryptographic Protocols . . . 9
  - 3.3.1 Diffie-Hellman . . . 9
  - 3.3.2 Kerberos . . . 10
  - 3.3.3 Secure Remote Password . . . 10
  - 3.3.4 Secure Socket Layer/Transport Layer Security . . . 11
  - 3.3.5 HTTPS . . . 13

4 Design of the Secure SJ 15
- 4.1 Design choices . . . 15
- 4.2 Delegation Protocols . . . 16
  - 4.2.1 Resending Protocol . . . 17
  - 4.2.2 Forwarding Protocol . . . 19
  - 4.2.3 Redirection Protocol . . . 21
- 4.3 Attacks' Protection . . . 21

5 Comparison with other approaches 23
- 5.1 A Cryptographic Protocol for Multiparty Sessions . . . 23
- 5.2 Security Policy Assertion Language . . . 23
- 5.3 Binder . . . 24
- 5.4 Delegation Logic . . . 25
- 5.5 Grid Delegation Protocol . . . 25
- 5.6 Constrained Delegation . . . 26

6 Implementation of Secure SJ **27**
6.1 Transport Layer Security - Secure Remote Password . . . 27
6.2 Delegation Cases . . . 30

7 Large Protocols in Secure SJ **33**
7.1 Online Shopping . . . 33
7.1.1 Protocol details . . . 34
7.1.2 The Network . . . 34
7.1.3 Execution of the program . . . 35
7.1.4 User's Manual . . . 35
7.2 Primes . . . 36
7.2.1 Protocols . . . 37
7.2.2 Execution of the program . . . 38

8 π-Calculus Delegation Protocols and their Correctness **39**
8.1 Syntax and Reduction . . . 39
8.1.1 Process Syntax . . . 39
8.1.2 Session Type Syntax . . . 40
8.1.3 Reduction Rules . . . 40
8.1.4 Typing rules . . . 40
8.2 Resending Protocol . . . 41
8.2.1 Case 1 . . . 41
8.2.2 Case 2 . . . 43
8.2.3 Case 3 . . . 44
8.2.4 Case 4 . . . 46
8.3 Forwarding Protocol . . . 50
8.3.1 Forwarding and Buffering functions . . . 50
8.3.2 Case 1 . . . 51
8.3.3 Case 2 . . . 53
8.3.4 Case 3 . . . 54
8.3.5 Case 4 . . . 56
8.4 Protocols Correctness . . . 60
8.4.1 Formalised Properties . . . 60
8.4.2 Proving Protocols Correctness . . . 62

9 Evaluation **69**
9.1 Implementation . . . 69
9.1.1 Delegation Case 1 . . . 69
9.1.2 Delegation Case 2 . . . 70
9.1.3 Delegation Case 3 . . . 71
9.1.4 Delegation Case 4 . . . 72
9.1.5 Running Test classes . . . 73
9.2 Protocols Correctness . . . 74
9.3 Session Java . . . 74

10 Conclusion and Further Work **75**

A Delegation Implementation **81**
A.1 Session sending . . . 81
A.2 Session receiving . . . 82

B Delegation Tests **85**
B.1 Case 1 . . . 85
B.1.1 A . . . 85
B.1.2 B . . . 85
B.1.3 C . . . 86
B.2 Case 2 . . . 86
B.2.1 A . . . 86
B.2.2 B . . . 87
B.2.3 C . . . 87
B.3 Case 3 . . . 88
B.3.1 A . . . 88
B.3.2 B . . . 88
B.3.3 C . . . 89
B.3.4 D . . . 90
B.4 Case 4 . . . 90
B.4.1 A . . . 90
B.4.2 B . . . 90
B.4.3 C . . . 91
B.4.4 D . . . 92

C π-Calculus **93**
C.1 Reduction rules . . . 93
C.2 Typing rules . . . 94

Chapter 1

Introduction

Nowadays communication is the base of most of the software development. However, since the majority of the API provides no structures, and object oriented programming does not provide enough support for high-level abstraction of distributed communications, we need to use some lower-level abstractions such as socket programming. Unfortunately, this type of communication is not very secure seeing that the conversation structure is both unspecified and untyped. Therefore, one should explore session types to provide a more structured and, consequently, a more secure communication type. A session is defined as a conversation made over a private channel without interference and a session type as a specification of the structure of the messages from the same conversation. Sessions provide a high-level programming abstraction for communications, grouping several interactions into a structured unit of conversation which provides a safe way of communicating.

Session Java (SJ) [32] was created in order to overcome the previously mentioned issues, using sessions as a means of communication. The main properties of this new language are the transport independency and the session delegation. It also uses a zero-copy passing of memory blocks approach, which prevents memory leaks and racing conditions. However, most of this work will be focused on the session delegations and their secure implementation.

In order to have Hardware/OS independent implementations, such as language-level virtual machines, a new language and a runtime framework based on session types were created. The implementation of this new SJ Framework [31] starts by type-checking the code, then creating a transport-independent intermediate form in Java, which can be executed by any machine with any transport protocol. In addition, different transports can also be added with slight modifications only. The current implementation already includes a set of well-known transports and its implementation is an extension of SJ that also uses Java as a core language.

SJ Framework is implemented upon three layers: application layer, session layer and abstract transport layer. With this design method there is a clean decoupling of session services and transport implementations with just a small degradation in performance. This framework also shows us some gains in portability, productivity, safety, expressiveness and a major type-directed optimisation.

The one thing that could be improved on this language would be the implementation of some security protocol since type-checking is not enough for a fully secure language, namely in the new version of SJ which already contains transport modules for HTTPS and SSL. This issue is the main focus of the project as the goal is for SJ to be as secure as possible. Accordingly, we will implement a Secure SJ which will have this problem fixed in the best possible way.

In order to achieve this security, we will start by researching the existing security protocols and choose one (or combine several) to embed in SJ so as to add the desirable properties. We will then specify the design of Secure SJ with specific incidence on the delegation protocols. Later on, we will compare this new version of SJ with several other approaches and will discuss the implementation issues in overall and specifically in the delegation protocols. Afterwards, we will test and create several examples in order to show SJ's security properties and will then formalise all delegation protocols and prove their correctness using π-calculus.

Report Structure

The following chapter is based on the report we did previously [13] on comparing this language with other parallel and distributed languages. The subsequent chapters are new work regarding the improvement of the security of the same programming language. Chapter 3 covers some background on the most important security protocols and chapter 4 traces the design of the new secured programming language. Chapter 5 compares Secure SJ with current related work and chapter 6 discusses all the implementation details. Several examples programmed in Secure SJ and all testing cases of delegation are provided in chapter 7. Then, we formally prove the correctness of the delegation protocols in chapter 8 and conclude the project with some further work notes in chapter 9.

Chapter 2

Session Java

2.1 Basic Functionalities of SJ

Session programming consists of two steps: specifying the protocols using session types and implementing these protocols using session operations. Session operations are performed by a set of session sockets created in order to implement those protocols, whereas session server sockets only accept connections if the type of the server is compatible with the requesting client. Then, every message exchanged between the server and the client is statically type checked so all messages respect the type of the session, which makes a more secure way of communicating.

The background for theoretical session types can be found at [23, 30, 33, 36, 45].

2.1.1 Session Delegation

Session delegation is an important feature of session-based programming which is transparent for the passive party. The mobility of the session is achieved by the session delegation at the same time as safety is preserved. It is also a type of communication that continues even if the peers are interchanged a large amount of times during the communication.

The easiest delegation method is for the sender to redirect all messages in both directions, between the receiver and the passive-party. The best part of this approach is that the passive-party is not required to do anything. However, the sender needs to be connected even after the communication has been delegated, which means that a failure of the sender implies in the failure of the session. An alternative is to close the original connection and replace it by a connection between the new peers. This obviously releases the sender from being connected all the time but it requires additional operations. Therefore, the first method is good for reliable hosts but not for dynamic network environments, whereas the second one gives an efficient solution for those environments. More details about session delegation can be found in chapters 4 and 8.

2.1.2 noalias type

SJ Framework includes a new type called *noalias*, with which the reference owns completely and exclusively the object it points to, using a zero-copy reference approach. This way the aliasing problem, such as memory leak and race conditions, is removed. In order for everything to work consistently a noalias variable can only be assigned by a noalias expression; and assigning a noalias variable to an existing noalias variable returns null. Furthermore, SJ session type variables are automatically considered both noalias and final and the appearance of the noalias type on the new version of SJ is one of the most important changes on this language (apart from being transport independent as explained later).

2.1.3 SJ Framework Layers

As in the previous SJ, programs are declared by the specification of session types and its implementation. The top layer (Application Layer) provides an abstract type-safe communication programming. It also offers flexibility by the use of session primitives and uses TCP sockets for communicating. The safety is ensured with the type-checking and it is currently possible to communicate without the runtime having constantly to check compatibility from every message received.

After the compiler checks everything is well-typed, SJ code is converted to Java. However, specific types such as *noalias* have no direct translation to Java and are encoded with the usage of new fields in the classes, being those instructions executed by calling the methods defined in the SJ Session Layer.

The SJ Runtime (SJR) is formed by both SJ Session Layer and SJ Abstract Transport Layer. SJ Session Layer exports several high-level session services to SJ Application Layer and imports the ones from the Abstract Transport, achieving with this design independency from concrete transports and becoming easier to extend or modify the code.

In the Application Layer a session is abstracted by using a session socket, which is a data structure for programmers. On the other hand, Session Layer links these abstract endpoints with the abstract peer. In SJ Session Layer, after the connection is established each peer sends its session types and checks their duality. Then, if they are duals the communication proceeds; if not, an exception is thrown.

This current version of SJ interprets all errors as exceptions resulting in a session termination, and this could be improved in future versions of SJ to give more reliable and long-lasting sessions. On the other hand, control messages for coordination are exchanged in the same channel as the regular session messages, thus preventing the sending of additional messages and reducing the overhead. The Abstract Transport Layer is composed by the ATI (Abstract Transport Interface) and the SJTM (SJ Transport Manager). The first one is a small subset of TCP sockets and uses important methods like as read, write and disconnect. The latter one gives us a set of session services such as interactions with other transports. This layer was created to use a transport-independent communication and gives support to the above SJ Session Layer. By the usage of the ATI we can use the "write once, run anywhere" paradigm and decouple high level from low-level layers. This way, the framework can be extended on a specific layer without affecting the other layers.

Finally, there were other transports implemented apart from the TCP, such as shared memory, HTTP/S and UDP, showing that the ATI is easily used for the creation of new transports.

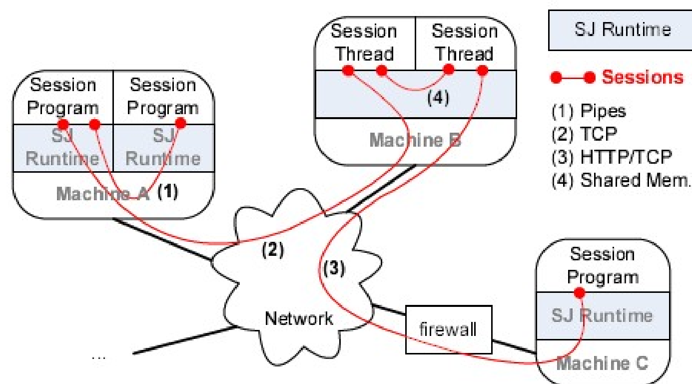


Figure 2.1: Sessions using different transports [31]

2.1.4 Benchmarking

Benchmark results show that sockets with the forwarding protocol have a low overhead compared to standard Java sockets which decreases in longer sessions. On the other hand, sockets with the resending protocol are slower then forwarding sockets but are still faster than RMI for longer sessions.

Other benchmarking and implementation details are explained in [7, 31, 32].

2.2 Comparison with other parallel and distributed languages

In our previous report, we compared SJ with several parallel and distributed languages such as Sing#, Crypto F#, MPI, XMem, X10, Fortress and Chapel. Briefly, we attained the following conclusions:

2.2.1 Sing# and Crypto F#

Regarding the similarities between Crypto F# [17] and SJ, both of them are based on sessions and SJ followed the idea of decoupling the implementation into an abstract layer to communicate with more than one type of platforms. The same way Crypto F# creates a ML file that is common to all peers, SJ converts the code into an Abstract Layer with the help of the ATI that makes its implementation platform free. On the other hand, the differences with SJ is that in Crypto F# it is not intended to create a transport-independent language and it is more focused on creating a secure system with cryptographic and integrity protocols. However, these security issues can be implemented on SJ and are the main purpose of this project.

The main similarity between Sing# [28] and SJ is the safe zero-copy passing of memory blocks. This eliminates aliasing problems such as memory leaks and racing conditions. In Singularity this is achieved with the channel contracts whereas in SJ this is achieved with the introduction of the *noalias* type. It is also similar since both implementations statically check the code preventing certain errors and communication incompatibility. On the contrary, the differences with SJ are that in Sing# an exchangeable heap is used, which all processes access for sharing data amongst themselves. In SJ, there is no common area and if a peer wants to transfer a shared part of memory, that is only possible with the noalias type since it does not copy the message and only one instance of that object exists. Additionally, Sing# is not aimed to be transport-independent and focus more on communicating in a shared memory environment and in a non-distributed way.

2.2.2 XMem, X10, Chapel and Fortress

Both SJ Framework and XMem [41] support type-safety and shared memory communication (the latter one due to the ATI). However, XMem uses multiple MREs (Managed Runtime Environments) on a single machine to gain efficiency comparing to other communication types and targets to obtain the same efficiency in those multiple MREs than in threads. Still, XMem does not ensure transparency between its communications whilst SJ does.

X10 [19], Fortress [4] and Chapel [3] focus on a partitioned memory accessible for all processes in order to communicate in something like a remote DMA which is more efficient than point-to-point communications. Furthermore, these languages are trying to reduce the complexity of programming due to several constructs for synchronization and coordination. Nevertheless, none of these languages have tried to decompose their architecture into abstract modules in order to create a transport-independent communication [42].

2.2.3 Summary

To summarise these comparisons, the introduction of the noalias type is compared to the channel contracts of Sing# and is implemented in the Abstract Layer as part of the shared memory transport. X10, Chapel and Fortress also influenced shared memory transport in the pursuit of one-sided communication and Crypto F# has influenced on HTTP/S connections. Other transports

can be easily implemented and can be used depending on the where the peers are relatively to each other. Table 2.1 provides a summary of all these parallel languages. To conclude, SJ programs are guaranteed to be free from communication errors [16], unlike for example MPI.

	SJ	Sing#	Crypto F#	XMem	X10	Fortress	Chapel
One-sided communication					✓	✓	✓
Point-to-point communication	✓	✓					
Transparency	✓			✓			
Session-based	✓		✓				
Message-based	✓	✓			✓		
Shared-memory	✓			✓			
Type-check	✓	✓	✓	✓			
Zero-copy of memory blocks	✓	✓					
Transport-independency	✓						
Security Protocol			✓				

Table 2.1: Comparison of SJ with the studied parallel and distributed languages

Chapter 3

Cryptography and Cryptographic Protocols

In this chapter we aim to find or combine several protocols in order to achieve message authentication, integrity and confidentially in the messages shared amongst the several peers connected through SJ. We will focus on the attacker being the network that can eavesdrop and modify every message, amongst other possible threats.

The following sections are divided between Symmetric and Public-Key cryptography, followed by a description of several important Cryptographic Protocols.

3.1 Symmetric Cryptography

3.1.1 Data Encryption Standard

Data Encryption Standard (DES) is the most important symmetric cipher algorithm. There are many others such as AES, although they are not as important as DES.

The structure of DES is much more complex than public-key ciphers such as RSA and more information about it can be found on [21]. The algorithm uses a 56-bit key to encrypt 64-bit blocks of data by a series of steps, and the decryption can be done by performing the same steps with the same key applied in reverse order.

However, it was found that DES could be broken by brute-force attacks (1998) and the new version Triple DES was released in 1999 which basically repeats DES three time on the plaintext using either two (EDE2) or three keys (EDE3).

Strengths and vulnerabilities

Using a 56-bit key there are 2^{56} possible keys which was practically impossible to break when DES was originally developed. For example, if only half the key space needed to be searched and one DES encryption took one microsecond, a single machine would take more than a thousand years to break the cipher. However, with today's technology it is easily breakable and that was proved in 1998 by Electronic Frontier Foundation (EFF).

Equally difficult is to recognise the plaintext. For example, if we do not have any previous knowledge of the language of the plaintext, it will be extremely difficult to distinguish the real plaintext from a bogus one. This task is quite difficult to automatise but EFF covered this issue and created other techniques useful in other contents.

There are sixteen keys that are advised not to be used [34] as they have strange properties. Four of them are the *weak keys* which are formed by all zeroes or all ones, being these keys their own inverse, so that encrypting with one is the same as decrypting with the other. The other twelve keys are the *semi-weak keys* in which each of those is the inverse of one of the others.

Timing Attacks

This type of attack lies on measuring the time to perform decryptions on various ciphertexts, exploiting the fact that encryption takes different times on different inputs. This does not give us the key but decreases the search space dramatically. However, this type of attack is more used on public-key cryptography and is unlikely that it actually works on DES or AES which the authors confirm, although they suggest some things left to explore.

Cryptanalysis

With the increasing of keys in DES it became extremely difficult to use brute-force attacks and cryptanalysis attacks started to increase. There are two types of cryptanalysis: differential and linear.

Differential cryptanalysis can decrypt DES in a matter of seconds for 8 rounds and with "only" 2^{47} encryptions for 16 rounds instead of 2^{55} which is just of theoretical interest. Although this type of cryptanalysis is powerful, it does not perform well on DES because its authors already knew about this kind of attack whilst creating the algorithm.

Linear Cryptanalysis tries to find linear approximations to describe the transformations happened in DES. This way the key can be found in 2^{43} attempts for 16 rounds which is not a big improvement compared to the differential one but is still a little better.

3.1.2 Advanced Encryption Standard

The Advanced Encryption Standard (AES) [2] was originally developed with the intention of substituting DES. It uses 128-bit data block and keys of 128 (10 rounds), 192 (12 rounds) or 256 bits (14 rounds). It is also very fast and parallelisable and is resistant to known attacks. AES consists of byte substitution, permutation, arithmetic operations and a XOR operation with a key, instead of the Feistel structure like DES.

Strengths and vulnerabilities

Although AES is much safer than DES and resistant to lots of attacks, there were still some possible attacks. Since this encryption can be expressed in equations, it is "possibly" faster to solve those equations instead of a brute-force attack (XSL attack). Other more plausible attack is based on cache side channel attacks [10, 11]. Cache attacks had already happened in the past, but the innovation is to perform these attacks in a remote way. This kind of attacks exploit the misses and hits on the cache while encrypting/decrypting some message and also the variations caused by the cache behaviour. The main purpose of these attacks is to decrease the search space of the possible keys.

Fortunately, these attacks are not that common and AES is one of the best symmetric-key cryptographic algorithms. However, AES will still have to wait until it becomes the most famous one, as Triple DES is the current approved algorithm for the U.S. government.

3.2 Public-Key Cryptography

Public-Key Cryptography can be used for authentication and confidentiality and is based on mathematical functions instead of permutations and substitutions that symmetric encryption uses. There is one myth regarding this kind of cryptography being safer than symmetric cryptography but the truth is that it only depends on the key size and the computational work for breaking the cipher.

The main applications of public-key cryptography are obviously the encryption/decryption of messages, a way of using *Digital signatures*, and probably the most important is for key exchange. Key exchange is the way of two parties to negotiate a session key with which they will continue to communicate. On the other hand, symmetric key encryption is used for high-volume of messages because it is up to one thousand times faster than the asymmetric one.

3.2.1 RSA

RSA was developed in 1977 and is based on being easy to multiply two prime numbers but computationally infeasible to factorise them back. The RSA is a block cipher in which the messages are numbers and are split into blocks less than the multiplication of those primes. This algorithm is based on number theory and is explained in detail in several books like [40]. However, if one accepts the results given by the detailed algorithm, a full understanding of whole the number theory is not required. The most important part is that it is based on two prime numbers that are kept private and that with some mathematical functions two keys are generated. Then one is kept secret and the other one is made public. Moreover, it is infeasible to calculate the original prime numbers from the public key.

Strengths and vulnerabilities

There are four possible attacks on RSA: brute force, mathematical attacks, timing attacks and chosen ciphertext attacks.

Regarding brute force attacks, the defence is based on using large keys as it is in all other cryptosystems. Nevertheless, since RSA calculations are complex, the larger the key, the slower the encryption/decryption process.

As for mathematical attacks, the primary attack is to factorise n into prime factors. With these numbers, it is easy to determine both the keys. Fortunately, it is computationally hard to factorise large primes and once again, everything depends on the length of the key. However, this problem is solved in polynomial time in a quantum computer. There is already a quantum algorithm [37] for factorising prime numbers with a high probability of correctness: the famous Peter Shor's algorithm [39]. Therefore, there is an important need for improving quantum cryptography so that when quantum computers are a reality, the world does not fall apart with major security leaks.

Timing attacks are the same as for DES and other cryptosystems and the chosen ciphertext attacks are based on an attacker choosing a number of ciphertexts and the corresponding plaintexts. With this, the attacker can "study" several blocks of each message and exploit the properties of RSA by some kind of cryptanalysis.

3.3 Cryptographic Protocols

3.3.1 Diffie-Hellman

Diffie-Hellman was the first public-key algorithm published and is still widely used amongst several commercial products. The goal of this algorithm was for two parties to exchange a session key in a secure way for future communications and its success depends on discrete logarithms.

Although there are improvements of this algorithm, it is the basis for some more complex key exchange protocols.

Strengths and vulnerabilities

The strength of this algorithm relies on the difficulty of solving discrete logarithms. If an eavesdropper gets the messages, she would need to solve the Diffie-Hellman problem [25], which is computationally hard. However, an efficient discrete logarithm algorithm would reduce the complexity of discovering the exponents and, hence, solve the Diffie-Hellman problem. Once again, the length of the chosen prime numbers is a key factor for the success of this algorithm. Furthermore, one should be aware of the risk of using a random numbers generator, because there is no algorithm for complete randomness and therefore it makes it easier to predict those numbers.

The original Diffie-Hellman algorithm does not grant authentication and is open to man-in-the-middle attacks. Therefore, a method for authenticating the parties such as digital signatures or public-key certificates is essential for a secure protocol.

3.3.2 Kerberos

Kerberos provides mutual authentication for a distributed environment and protects against eavesdropping and replay messages by using a protocol similar to Needham-Schroeder. It requires a trusted third-party and uses symmetric-key cryptography. The current version V5.0 is specified at [18] and operates like Figure 3.1.

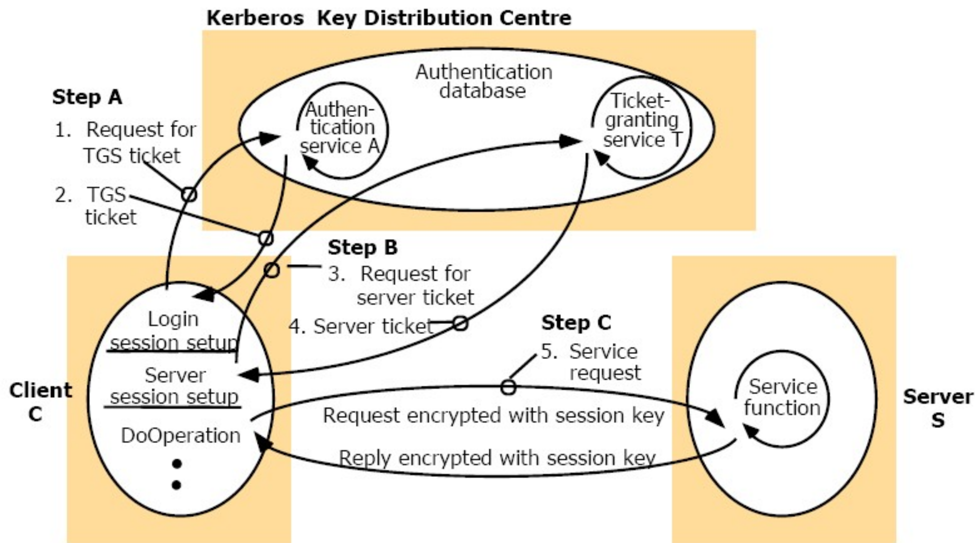


Figure 3.1: Kerberos Protocol [26]

The protocol starts by getting authentication in the system, followed by obtaining a ticket to access the service, and concluding by the access of the wanted service. Kerberos is widely used amongst major companies like Sun, Apple, Microsoft and Google.

Strengths and vulnerabilities

Kerberos has some leaks and vulnerabilities like dictionary attacks that were explored and explained in detail in [44]. In order to overcome this issue, a secure password authentication protocol like SRP or EKE should be used together with the preauthentication phase of Kerberos.

3.3.3 Secure Remote Password

The Secure Remote Password (SRP) protocol is an authentication method that allows the exchange of passwords over an unprotected network without revealing them to an eavesdropper. This protocol is resistant to dictionary attacks and uses perfect secrecy, which protects past passwords against future attacks. Moreover, no third party is used, being only the main two parties involved in the authentication protocol. Even if an attacker gets access to the password database, she cannot compromise the security of the system as passwords are stored in a form that is not plaintext equivalent.

To set up a password P with Steve (server), Carol (client) has to pick a salt s and compute the hash $x = H(s,P)$ and $v = g^x$. Then Steve stores v and s and x is discarded.

The authentication protocol is as in Figure 3.2 and has already been proven secure in [43].

	Carol		Steve
1.		$\xrightarrow{C}$	(lookup s, v)
2.	$x = H(s, P)$	$\xleftarrow{s}$	
3.	$A = g^a$	$\xrightarrow{A}$	
4.		$\xleftarrow{B,u}$	$B = v + g^b$
5.	$S = (B - g^x)^{a+ux}$		$S = (Av^u)^b$
6.	$K = H(S)$		$K = H(S)$
7.	$M_1 = H(A, B, K)$	$\xrightarrow{M_1}$	(verify M_1)
8.	(verify M_2)	$\xleftarrow{M_2}$	$M_2 = H(A, M_1, K)$

Figure 3.2: Secure Remote Password Protocol [43]

Strengths and vulnerabilities

The strength of this algorithm is that it is not vulnerable against dictionary attacks and even if the attacker captures the host's password or the session key it cannot perform authentication nor mount a dictionary attack on the password. All this security is based, once again, on the difficulty of solving the Diffie-Hellman Problem. However, in order to get the most out of this protocol, some large primes must be chosen. In addition, this protocol has a good performance compared to other protocols like A-EKE and B-SPEKE, which makes it not only secure and simple, but also speed efficient.

3.3.4 Secure Socket Layer/Transport Layer Security

The Secure Socket Layer (SSL) offers security between TCP and the applications that use it and also provides confidentiality (symmetric encryption) and message integrity (MACs). Later on, the successor Transport Layer Security [24] (TLS) was published and defined as a standard protocol. This protocol has several mechanisms to allow users to determine which services and security mechanisms they want to use.

SSL is composed by two layers of protocols as shown in Figure 3.3: the bottom layer is the SSL Record Protocol which provides the basic security to the above layers, and the above layers are the Alert Protocol, the Handshake Protocol and the Change Cipher Spec Protocol, which provide the transfer of SSL messages and the session management.

SSL handshake protocol	SSL cipher change protocol	SSL alert protocol	Application Protocol (eg. HTTP)
SSL Record Protocol			
TCP			
IP			

Figure 3.3: SSL Protocol Stack [5]

SSL Connection and Session

The SSL Connection and the SSL Session are two important notions and can be defined as:

- *Connection:* a peer-to-peer link with the suitable type of service. Note that each connection is associated with a session.
- *Session:* an association between a client and a server that characterises certain parameters like which algorithms are used. A session is produced by the Handshake Protocol and is utilised to avoid the negotiation of new security parameters for each new connection.

During the Handshake Protocol, the encryption methods are negotiated as well as the various states associated with each session. A session state has several parameters like the identifier, certificate, cipher specification, a secret, etc. On the other hand, a connection state has parameters such as a random number, client and server MAC secret and key, initialisation vectors and sequence numbers.

SSL Record Protocol

The SSL Record Protocol provides confidentiality and message integrity for SSL connections. The main purpose of this protocol is to fragment the data of an application message, encapsulate it with certain headers and create a record, which is encrypted and then sent using the TCP protocol. Finally, the receiver decrypts, verifies and reassembles the data, and delivers it to higher-level users.

Change Cipher Specification, Alert and Handshake Protocols

The Change Cipher Spec Protocol is the simplest SSL protocol and consists of a single message containing the value 1. The purpose of this message is to copy the pending session state into the current state, updating the cipher suite to be used on that connection.

The Alert Protocol is used to transmit alerts to other peers. Each message of this protocol is composed by two bytes: the first one indicating whether is a warning or a fatal error, and the second one containing the error code.

The Handshake Protocol (Figure 3.4) is the most complex one and allows the two peers to authenticate each other and negotiate the encryption, the MAC algorithm and the cryptographic keys. This protocol has four different phases:

- **Phase 1 - Establish Security Capabilities:** including protocol version, session ID, cipher suite, compression method and initial random numbers. The client has a list of cryptographic algorithms in order of preference for which the server selects one of those. The following key exchange methods are available on the latest version of TLS: RSA, Fixed Diffie-Hellman, Ephemeral Diffie-Hellman and Anonymous Diffie-Hellman. Regarding Cipher Algorithms, we have that RC2, RC4, DES, 3DES, DES40 and IDEA are supported, and that the MAC algorithm can be either the MD5 or SHA-1.
- **Phase 2 - Server Authentication and Key Exchange:** the server may send the certificate and key exchange and may request the client certificate. Thereafter, it obligatory sends a signal of the end of the hello message phase.
- **Phase 3 - Client Authentication and Key Exchange:** the client should verify if the certificate received is valid and, if requested, the client should send its own certificate. Afterwards, it should send its key exchange message and may send a certificate verification.
- **Phase 4 - Finish:** the last phase consists of changing the cipher suite and finishing the handshake protocol.

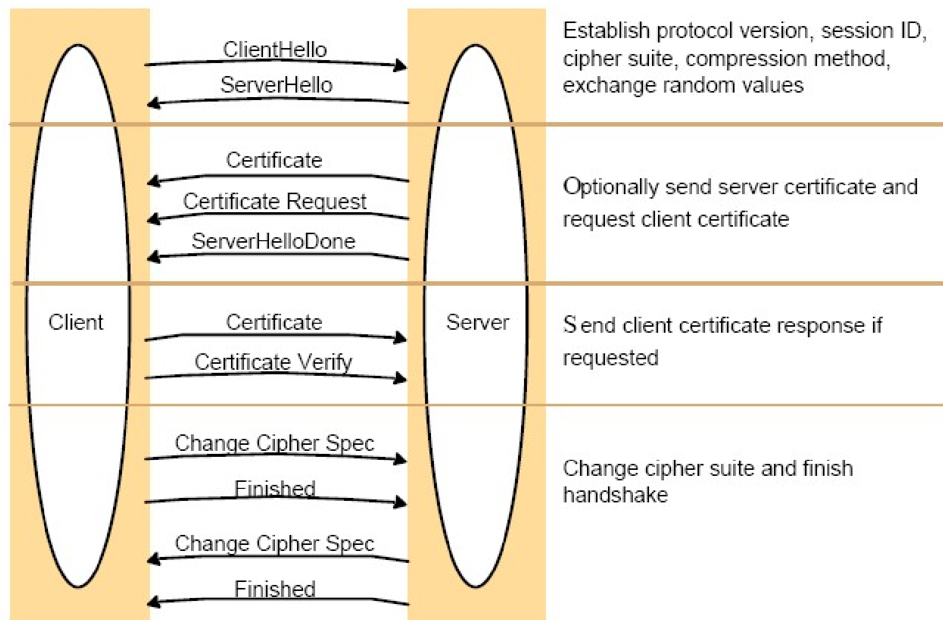


Figure 3.4: SSL Handshake Protocol [26]

3.3.5 HTTPS

The Hypertext Transfer Protocol Secure (HTTPS) [29] is a combination of the HTTP with a network security protocol. HTTPS used to use the SSL protocol but nowadays uses its successor TLS and to call forth this secure protocol, one should use https:// in the URI (which uses port 443 by default) instead of the typical http://. By the use of the HTTPS, we can achieve protection of eavesdropping and man-in-the-middle attacks.

HTTP is executed at the application layer (TCP/IP) and the security protocol executes at a lower layer, encrypting and decrypting an HTTP message. Therefore, the main difference is calling SSL functions instead of using TCP calls, being necessary to add some calls to setup the security configuration. This difference can be seen of Figure 3.5.

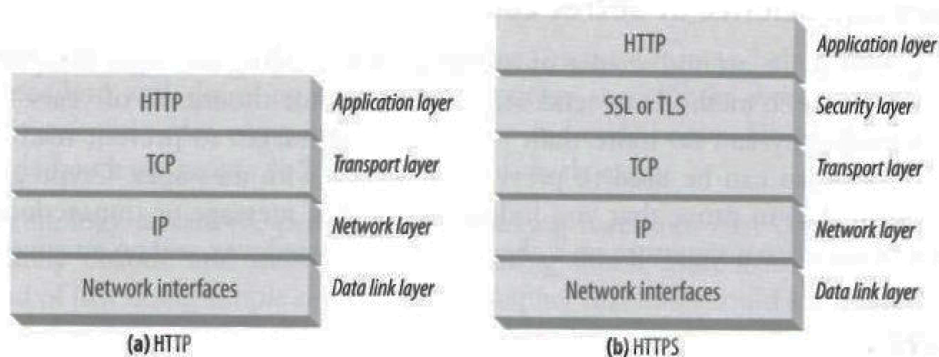


Figure 3.5: HTTP vs HTTPS [29]

Strengths and vulnerabilities

The HTTPS is vulnerable when applied on publicly-available static content or any other source of indexation. This vulnerability allows the attacker to have access to the plaintext as well as to the ciphertext. In order to get the best out of the HTTPS, one should follow the idea of [27] or in case of wanting to initialise a TLS over an existing TCP connection, allowing for all traffic to share the same port, [38] provides a good information.

Chapter 4

Design of the Secure SJ

4.1 Design choices

In order to improve the security of SJ, we wanted authentication amongst the peers and we did not want external parties to have access to the exchanged information between the peers. We also did not want for the intruder to modify the messages, so message integrity was also a key factor. To sum up, the properties that we wanted to achieve were: authentication, message integrity and confidentiality.

	Diffie-Hellman	Kerberos	SRP	SSL/TLS
Confidentiality	✓	✓	✓	✓
Integrity				✓
Authentication		✓	✓	✓
Trusted Third Party		✓		
Eavesdropping protection		✓	✓	✓
Man-in-the middle vulnerable	✓			
Dictionary attacks vulnerable		✓		

Table 4.1: Comparison of various security protocols

After carefully analysing these protocols (Table 4.1) and bearing in mind that we aimed for the above properties using only two peers and not with third-parties, we opted to adapt the SSL/TLS protocol. Our major goal was for SJ to be as secure as possible in order to be adopted by several institutions without fear of compromising their data.

Since with the TLS protocol, most of the key exchange protocols require certificates or external authorities, it was needed to adapt it [20] with an authentication method that allowed the exchange of passwords over unencrypted channels without revealing them to an eavesdropper. The best method for this was the SRP, which was mentioned in the previous chapter.

The developers of SJ had already implemented HTTPS as a transport using SSL sockets, but it was the one implemented by Sun that did not have SRP as a key exchange protocol. Furthermore, there were some bugs in the original implementation which we intended to fix. Hence, we planed to implement the SRP Sockets, followed by our own SSL sockets adapted to use the SRP protocol. Then, it was just needed to modify the HTTPS transport in order to use our SSL sockets instead of the Java ones. Figure 4.1 shows the structure of the implementations we intended to do.

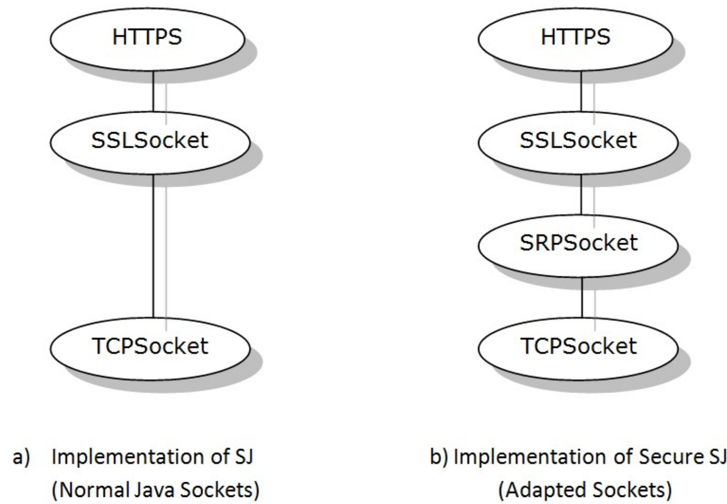


Figure 4.1: Socket Layers of Secure SJ

4.2 Delegation Protocols

One of the most important features of SJ is the session delegation, which achieves the mobility of the session at the same time as safety is preserved. In this type of communication the sender and the receiver are initially unknown to each other until a third party delegates the session for them, being this transparent to the passive-party.

There are two reconnection-based delegation protocols: the resending and the forwarding protocols. Whilst the former one resends all lost messages after reconnection, the latter one resends all lost messages before reconnection. There is also a redirection-based delegation protocol where the sender redirects all messages in both directions, between the receiver and the passive-party. Although the passive-party is not required to do anything, the sender needs to be connected even after the communication has been delegated, which is not so good in the way that if there is a failure in the sender, the session will also fail. Nevertheless, this protocol is good for reliable hosts but not for dynamic network environments, whereas the reconnection-based one gives an efficient solution for those environments.

Communication security

In order to secure communications and bearing in mind that the security properties we wanted to achieve were (as mentioned before) *message authentication*, *integrity* and *confidentiality*, SSL sockets were used to create a secure channel, achieving all these properties and preventing eavesdropping, message tampering and forgery from malicious parties.

Delegation was based on the trust that one party has on another. For example, if Alice trusts Bob and Bob trusts Carol and wants to delegate its session to her, then Alice would automatically trust Carol. Upon the delegation, Bob would create new credentials that would be sent to both Alice and Carol, since he is the only one both of them trust (being not so "trustworthy" if either Alice or Carol are the ones creating the new credentials).

Please note that in all of the defined protocols, in case of a Failure in credential checking, the protocol will stop and **no further operations are executed**.

4.2.1 Resending Protocol

Case 1

In the Resending protocol, Alice will have to send all the lost messages to Carol after reconnection. The modifications in this case of this protocol from the original one [32] can be seen on Figure 4.2 and are the creation of the credential by Bob on step 1 and its sending on step 2 to Carol and on step 5 to Alice. Carol will then store the credential and allow the connection from Alice. Then, on step 9 Alice sends the credential to Carol in order to establish a new connection, which is checked by Carol and if the credential matches the previous one, a successful connection is established, otherwise there is a failure and the socket is closed.

1. B: creates CredA
2. B → C: $SD^{B}_{A}(C)$::CredA
3. C: open server socket on free port p_c, accept on p_c
4. C → B: p_c
5. B → A: $DS^{B}_{A}(C) = \langle ST^{B}_{A}, IP_c, p_c, CredA \rangle$
6. A → B: ACK_{AB}
7. A: close s | 7'. B: close s
8. A: connect to $IP_c : p_c$
9. A → C: CredA | 9''. C: CredA checking
9a. - pass: successfully connected | 9a''. - pass: connection established
9b. - fail: login error, close s | 9b''. - fail: login error, close p_c
10. A → C: $LM(ST^{A}_{B} - ST^{B}_{A})$

Figure 4.2: Resending Protocol: Case 1 (adapted from [32])

Case 2

In case 2, Alice has already finished her part of the session and is waiting for the acknowledgment. The modifications in comparison with case 1 are that Alice does not send an ACK to Bob and instead sends a FIN (embedded in the close operation) to let Bob know that she has already finished her side of the session.

As in case 1, Bob generates a new credential in message 1 and sends it to Carol in message 2 and to Alice in message 5. The overhead is minimum, since no new messages are necessary for the sending of the credential. This case can be seen on Figure 4.3.

1. B: creates CredA
2. B → C: $SD^{B}_{A}(C)$::CredA
3. C: open server socket on free port p_c, accept on p_c
4. C → B: p_c
5. B → A: $DS^{B}_{A}(C) = \langle ST^{B}_{A}, IP_c, p_c, CredA \rangle$
6. A: close s | 6'. B: close s
7. A: connect to $IP_c : p_c$
8. A → C: CredA | 8''. C: CredA checking
8a. - pass: successfully connected | 8a''. - pass: connection established
8b. - fail: login error, close s | 8b''. - fail: login error, close p_c
9. A → C: $LM(ST^{A}_{B} - ST^{B}_{A})$

Figure 4.3: Resending Protocol: Case 2

Case 3

In this case Alice is also trying to delegate another session (with Dave) to Bob. This case works as listed in Figure 4.4. The differences from case 1 are that Bob will receive the delegation signal from Alice in message 2' but will ignore it because he was already delegating another session. Alice will then resend the delegation signal after sending the lost messages to Carol in message 9, which will trigger a new delegation between Alice and Dave.

Regarding security, the overhead is just creating and sending the credentials in message 2 and 2' like in the previous cases, the sending and checking of those credentials in message 8 and 8" and also in the delegation signal in message 9.

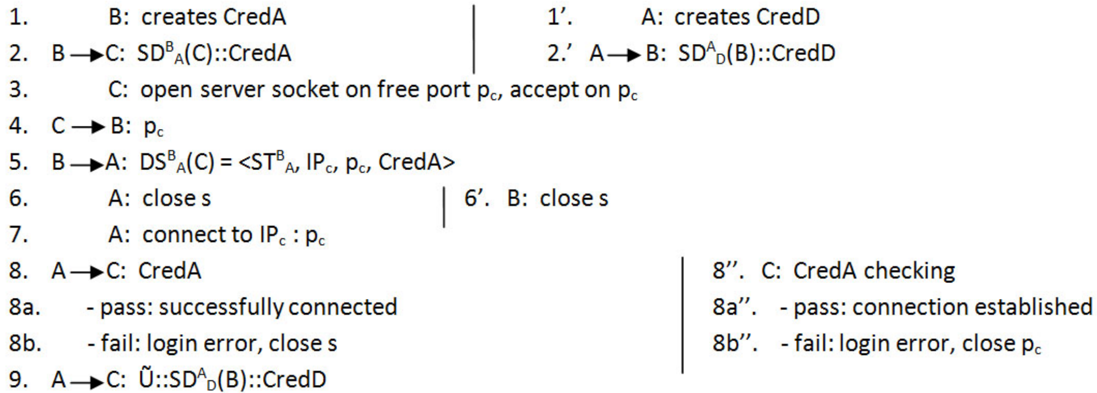


Figure 4.4: Resending Protocol: Case 3

Case 4

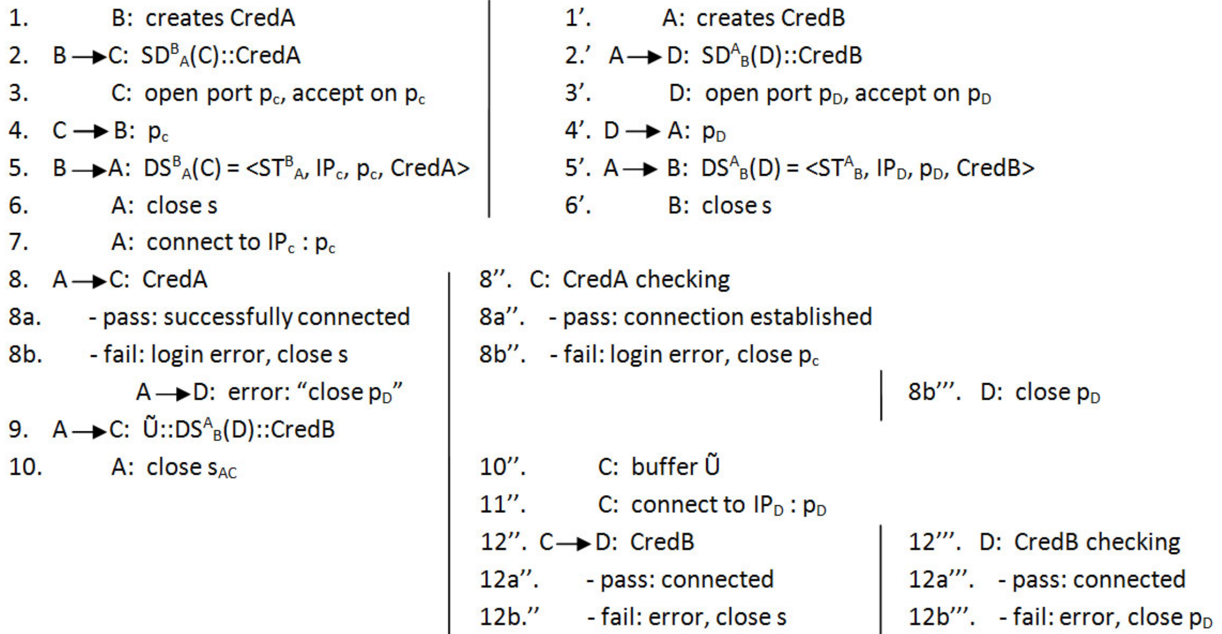


Figure 4.5: Resending Protocol: Case 4

The most difficult case – double delegation – can be seen in Figure 4.5. Comparing with case 1, it does not have the ACK in message 6 and the steps 1 to 5 are repeated in parallel between Alice and Dave. Then Alice establishes an intermediate connection between herself and Carol to send the lost messages along with the Delegation Signal between herself and Dave. At the same time, Carol buffers the lost messages, followed by connecting to Dave. If the intermediate connection

fails, the socket between Alice and Carol is closed and Alice notifies Dave to close his port as well, since the delegation is no longer taking place.

To secure this protocol, the credentials are sent together with the Delegation Signals like in the previous cases. One particular point in this case is that since Bob delegates his session to Carol, it is her who connects to Dave with the credentials that were supposed to be from Bob. This does not create any security problem since Bob has to believe in Carol in order to delegate his session to her in the first place.

4.2.2 Forwarding Protocol

Case 1

In this protocol, Alice will not have to resend the lost messages, which will be done by Bob instead. As in the previous protocol, the modifications in this case of this protocol are the creation of the credential on step 1, the sending of that credential on step 2 to Carol and on step 5 to Alice. This can be seen on Figure 4.6. Thereafter, Carol will also store the new credential, allowing a future connection from Alice. Whilst connecting, Alice will send the credential to Carol and if it passes the checking, the final connection is established, otherwise a failure message is presented.

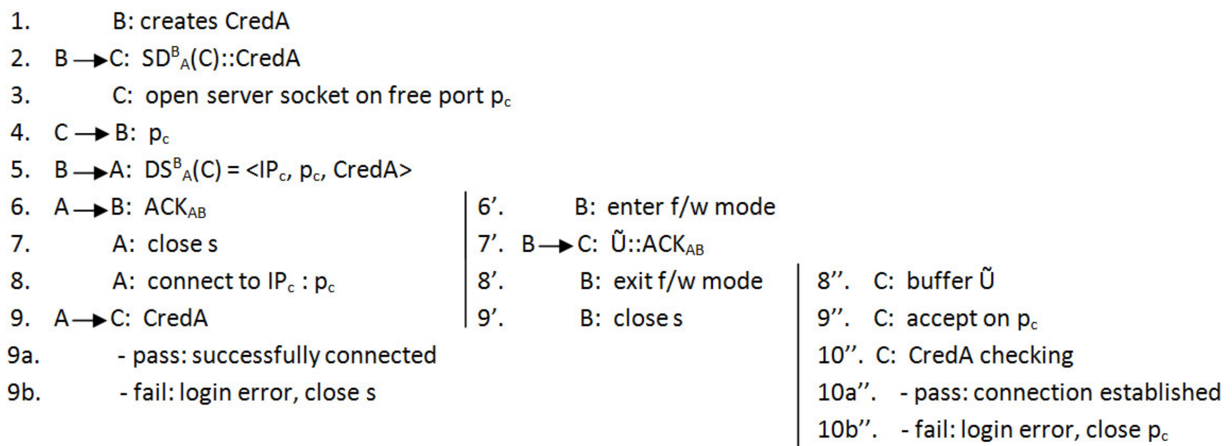


Figure 4.6: Forwarding Protocol: Case 1 (adapted from [32])

Case 2

In case 2 (Figure 4.7) of this protocol and as in the resending protocol, the ACK message on step 6 disappears, giving place to a FIN signal (inside the close operation). Message 8 also disappears since in the forwarding protocol it is Bob who forwards the lost messages and not Alice, not needing to connect to Carol like in the Resending protocol. Then in message 7' Bob sends a FIN signal after the lost messages to let Carol know that Alice has finished her part of the session, which Carol buffers in step 8"after the lost messages. This protocol is concluded by the closure of the port c that Carol has opened for the connection with Alice.

This case is secure as Bob sends the credentials to both Carol and Alice. However, these credentials are not needed since Alice will not connect to Carol in the end, being just the existing connection between Bob and Carol used.

1.	B: creates CredA
2.	B → C: $SD^B_A(C)$::CredA
3.	C: open server socket on free port p_c
4.	C → B: p_c
5.	B → A: $DS^B_A(C)$ = <IP_c, p_c, CredA>
6.	A: close s

6’.	B: enter f/w mode		
7’.	B → C: Ũ::FIN		
8’.	B: exit f/w mode	8”.	C: buffer Ũ::FIN
9’.	B: close s	9”.	C: close p_c

Figure 4.7: Forwarding Protocol: Case 2

Case 3

In this case Bob receives a delegation signal from Alice in message 2’, which will then forward to Carol after the resending of the lost messages (step 7’), replacing the previous ACK. Carol will then buffer that delegation signal (step 8”) which will trigger the new delegation with Dave. Finally Alice connects to Carol by sending the credentials and as before, only if the credentials are accepted, the new connection takes place, otherwise an exception is thrown.

Once again, this protocol is secure by the credentials sent by Bob in the first place, not increasing the overhead since no additional messages are sent. This case is shown in Figure 4.8.

1.	B: creates CredA	1’.	A: creates CredD
2.	B → C: $SD^B_A(C)$::CredA	2.’	A → B: $SD^A_D(B)$::CredD
3.	C: open server socket on free port p_c		
4.	C → B: p_c		
5.	B → A: $DS^B_A(C)$ = <IP_c, p_c, CredA>		
6.	A: close s		
7.	A: connect to IP_c : p_c		
8.	A → C: CredA		
8a.	- pass: successfully connected		
8b.	- fail: login error, close s		

6’.	B: enter f/w mode		
7’.	B → C: Ũ::$SD^A_D(B)$::CredD		
8’.	B: exit f/w mode	8”.	C: buffer Ũ::$SD^A_D(B)$::CredD
9’.	B: close s	9”.	C: accept on p_c
		10”.	C: CredA checking
		10a”.	- pass: connection established
		10b”.	- fail: login error, close p_c

Figure 4.8: Forwarding Protocol: Case 3

Case 4

Last but not the least, the case 4 of the forwarding protocol, shown in Figure 4.9, is basically another delegation requested in parallel (steps 1’ to 5’). Then Bob informs Carol that there is a simultaneous delegation after the lost messages (step 7’) and Alice informs Dave about the same. Then, in message 9” Carol closes her opened port and connects to Dave instead, concluding the double delegation. In the Forwarding case, there is no need for an intermediate connection, since everything is done with the existing connections. Therefore, errors may only occur in the connection between Carol and Dave, which as before will close the sockets and result in an exception.

Regarding security, the credentials are sent in messages 2 and 2’ like in the previous cases; and also inside the delegation signals in messages 7 and 7’, allowing the creation of the final connection.

In this case only credential B is used, since we chose Carol to connect to Dave and not the other way around.

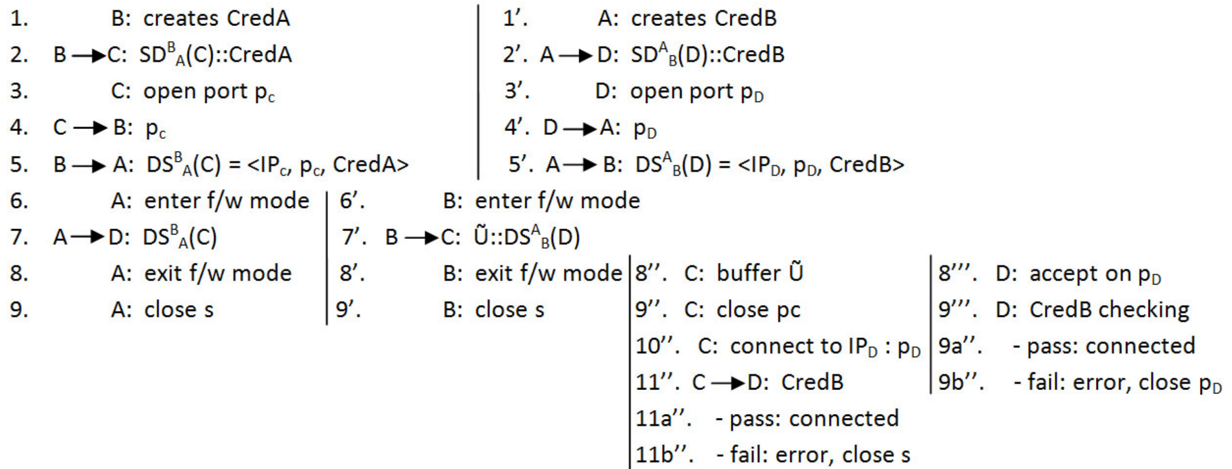


Figure 4.9: Forwarding Protocol: Case 4

Note that both the Resending and the Forwarding protocols have the weakness of Bob being an unreliable peer, compromising the entire communication. However, we assume Bob as a reliable party, since we are already connected to him (if we did not trust Bob, we would not be connected to him in the first place).

4.2.3 Redirection Protocol

The Redirection Protocol is the simplest protocol which only redirects the messages without additional operations. To implement security on this protocol, there are also no additional operations to perform, since the communications are already happening in a secure channel. This has the same weakness as before, in which if Bob is compromised, all peers will be jeopardised. However, we follow the same assumption as before.

4.3 Attacks' Protection

This design prevents the most common forms of attack, leaving the other not-so-common attacks for future work.

Regarding the fundamental threats: masquerade, bypassing control, authorisation violation, Trojan horse and trapdoor, it protects against the first three, leaving the latest two unprotected. However, these two threats cannot be protected via cryptography but only with efficient security services that protect the terminals of all peers, since this is where the vulnerabilities are.

A list of some of the threats is presented and briefly discussed:

Interception

Using SSL will prevent the attacker from gaining access to the data, since it provides *confidentiality*. However this does not protect the transmitted data against traffic flow analysis, although the attacker cannot decipher the data. This property is useful when sending private information such as credit card numbers or other kind of messages that we do not want an intruder to discover.

Fabrication

Using the design above, there is a protection against the insertion of "counterfeit" messages, since it assures that the message is from the proper source and protects against third-party masquerade (*authentication*). This feature is also very important to verify the other party is who she claims to be and to ensure all communication transactions are genuine.

There are some known vulnerabilities on SSL regarding an attacker injecting certificates to the client forcing her to believe in the certificate and establish a secure connection. However, this problem is not possible with our design, since to establish a connection, the client should know the password that she shares with the server. In case the client does not know the password, the session is not established, solving this vulnerability.

Modification

With the protocol defined previously, an attacker aiming to modify a message will be blocked, since the secure channel guarantees the message is received as sent (*integrity*). Integrity breaches can be quite serious, since an intruder could modify the data sent, such as a credit card numbers or any other important information.

Denial of Service

Unfortunately, SSL does not prevent a Denial of Service attack, since cryptography cannot prevent this kind of attacks (one can attack either the client or the server, forcing her to disconnect, creating a DoS).

Data Storage Issues

SSL guarantees that the information sent during the communication is protected against a series of attacks. However, once the information reaches the destination, we do not know how the data is stored on the computer, which may not be in a secure way. Nevertheless, we assume that information is well-stored and we can use Secure SJ for important applications such as banking, etc.

Chapter 5

Comparison with other approaches

5.1 A Cryptographic Protocol for Multiparty Sessions

The best way to protect multiparty sessions (as described in [17]) is to use directly the compiler and automated tools for generating efficient and secure cryptographic protocols, instead of leaving that job to the programmer. Obviously, the programmer remains in total control of the session, such as which messages to send or receive, sessions to join, etc. However, with this design, there are fewer chances of the programmer leaving any security issues unsolved, since implementing a secure system can be difficult and easily error-prone.

Moreover, all cryptographic protocols are hidden from the programmer to prevent reasoning about the session behaviour. Then, for all session specifications, the compiler creates type annotations that are typechecked alongside with the authentic executable code.

Every session is converted into a graph to easily describe their features and design choices. Thereafter, the compiler extracts the necessary information from the graph, by exploring all possible execution paths and extracting the control flow for each role, applying cryptographic protection to each message exchanged.

All these protocols focus more on integrity rather than secrecy, although secrecy is also implicit by typechecking. If we assume that the code is reliable to provide secrecy for values assigned to session variables and that they do not leak to an opponent, than secrecy is preserved whilst typechecking.

Comparison

In Secure SJ, the security will be preserved by the secure channels (SSL) in a more high-level way, rather than encrypting every message and typecheck it against others generated automatically by the compiler. Nevertheless, since there is a secure channel, all messages will also be protected against integrity and secrecy issues, but all the overhead would be on the creation of the channel, instead of in every message exchanged.

One difference is that the programmer would have to explicitly state that he wants to use the secure sockets, whereas in the Cryptographic Protocol mentioned previously, everything would be kept away from the programmer and done automatically. However, the major difference is that SJ connections involve only two peers and this cryptographic protocol has multiparty sessions, which means that in SJ, security issues are much easier to solve. One other thing is that in the cryptographic protocol there are no session delegations, which is one of the most important topics of this project.

5.2 Security Policy Assertion Language

The Security Policy Assertion Language (SecPAL) [15] is an authorisation language for expressing policy expressiveness and execution efficiency. Several distributed systems interact with entities

they do not know, so they need the authorisation decisions automated according to some policies that are pretty close to natural language.

This language is based on logics and is composed by delegation, negation and constraints, as the three major features.

Since we are mostly interested about delegation, in this language it is possible to bound in depth the number of re-delegations allowed, i.e. we can explicitly say the total number of nested delegations that one can perform. Moreover, it is also possible to limit delegation by width, meaning that we do not care about the number of delegations, but are only interested that all delegators satisfy a certain property, like an IP range, email address, etc.

Comparison

In the current design of Secure SJ we do not care about the number of delegations and allow to delegate all sessions to everyone based only on trust. However, in future approaches, we might want to reconsider by limiting delegations in a similar approach.

5.3 Binder

Binder [22] is an extension of the logic-programming language Datalog and its programs can be more expressive than standard security languages. There are many data structures to store security statements such as Access Control Lists (ACLs) and X.509 certificates. However, they all need to comply with some formal schemas. Nevertheless, if we want an open system that allows multiple actions without previous knowledge of what those actions might be, then a more expressive language is needed. That is when Binder takes place since it is a programming language more expressive than most security languages.

One advantage of using Binder is that it allows separate databases to securely interoperate, whereas Datalog programs just allow a single database, being relatively hard to encode all policies into a single program. In order to distribute into several programs, each component has its own *context* - used to decide whether or not to allow certain local authorisations. These contexts interact with each other through signed *certificates*, by exporting or importing statements into them. Another key property of this programming language is that all queries are decidable in polynomial time.

Delegation and trust are not primitives of the language, although they can be easily performed via some policies related with the importation of certificates. Furthermore, Binder is *monotonic*, which means that if a single statement is derivable, it will still be derivable if more statements are added. Hence, if a proof is generated in a small environment, it is still valid on a bigger environment where we have much more statements stored. Due to monotonicity, certificates cannot be automatically revoked and there are three possible ways to extend Binder in order to do so. We can either add expiry timestamps on the statements, add a freshness keyword on the syntax of the language or add a Boolean flag regarding the revocation of a certain certificate. In case a statement is not fresh anymore, we can ask for the validity of the proof by regenerating it again, or we can choose to ignore that statement.

Binder was compared with other security languages and no other language has all of its properties. Binder was also the pioneer of the authorisation security languages and influenced this research stream, namely other languages like SecPAL (5.2).

Comparison

Binder is very similar to SecPAL for both of them are an extension of Datalog. However SecPAL is an improvement comparing to Binder, because as it was influenced by Binder, certain flaws were removed. Comparing with SJ, neither of them is similar to the current version of Secure SJ but in a future version we can include expiry timestamps in the certificates to revoke them after a certain period of time. This may be useful to ensure the freshness property, but for the time being we

assume sessions do not last too long and we can use the make (Cred) method to ensure its freshness (cf Chapter 8).

5.4 Delegation Logic

Delegation Logic (DL) [35] is a logic-based authorisation language for large distributed systems.

When requesting some information, its authorisation can be divided into authentication and access control. The problem with authorisation lies on deciding whether the credentials used on trust managements prove that a request fulfils the policy involved with it. Sometimes just the identity of the peer is enough, but in more complex systems it is not. Hence, it is necessary to provide an easy way to specify the authorisation policies, which are understandable by everyone while remaining expressive.

DL distinguishes from other common trust-management engines in offering "credentials proving that a request complies with a policy" which is abstracted from implementation details. Moreover, it is extensible to negation and non-monotonicity and its programs start with Datalog proofs. In addition, it allows complex principals and has explicit support for delegation features such as limiting the depth.

There was an old version of DL but this one was improved in several aspects like the fact that the transformation into OLP (Ordinary Logic Programs) is computationally tractable in polynomial time and only needs to be done once.

Whenever an authoriser gets a query supported by some credentials, the authoriser creates a DL program that has all his policies and all received credentials. Then, DL creates a unique minimal model for that program and answers the query based on that minimal model. Overall, DL starts by converting the D1LP (Delegation Logic Program version 1) program into an OLP program, which led for a direct implementation based on compiling D1LP into OLP.

Regarding specific keywords and statements, one important statement of DL is the "*threshold*" with which we can specify a structure with several principals and a minimum value of those principals for a certain atom to be true. Furthermore, DL can control the depth of the re-delegation which can be infinite in some cases. There is also a statement called "*speaks for*" similar to the delegation, whose difference is that a speaks for statement does not consume delegation depth. The main reason for having these kinds of statements is to handle delegation to principals that cannot make any statements directly like in the X.509 certificates and in the SPKI/SDSI.

In order to test this logic-based programming language, several examples were implemented and studied, which showed some positive results.

Comparison

DL is also a logic-based language like both SecPAL and Binder, but it is currently completely different from Secure SJ. Apart from that, DL is similar to SecPAL in the way that we can specify the depth of the re-delegations. This shows that several systems already adopt this characteristic, which should be considered for future version of Secure SJ.

5.5 Grid Delegation Protocol

Grid Delegation Protocol (GrDP) [12] is a delegation protocol based on WS-Trust specification [8], which is uniform and independent from the security mechanisms and can also be used by several different grid environments and applications. Ws-Trust is a web service specification that provides some extensions to the secure mechanisms of WS-Security, such as a request/response protocol for creating, renewing and validating security tokens and to establish trust relationships between peers in a secure message exchange.

The current version of GrDP only has a profile for using it with X.509 proxy certificate delegation but others like GSI proxy delegation and Kerberos delegation are being currently developed. The profile is used for describing how to use the WS-Trust for a specific GrDP implementation.

The desired properties of the GrDPs are: transparency, generality, interoperability, modularity and flexibility; and all of those are met within the WS-Trust specification. Thus, it was shown that this specification was good for achieving the independency and uniformity of the security mechanisms.

In order to start a delegation, the delegatee issues the request and sends it to the delegator, followed by the delegator verifying if the request is valid, performing the operations requested and sending the credential back to the delegatee. The numbers of messages can be simplified if the delegator already knows what are the operations to perform (Pull Model), which is what happens in Secure SJ.

Finally, in GrDP it is also possible to revoke credentials by their lifetime and in case of the credentials being revocable, it is also possible to opt for being renewable or not.

Comparison

Comparing with SJ, both of them can have the delegator creating the credentials straight away because they already know their format (Pull model in GrDP). Concerning the comparison with the previous mentioned systems, GrDP is similar to Binder in that both of them have lifetime and revocation on the certificates.

5.6 Constrained Delegation

It is possible that we want to delegate some privilege over some resources but not the right to access them. This is not possible in current delegation languages, so there was a need to separate these two issues. In order to achieve that, Constrained Delegation [14] proposed a model that uses constraints to restrict the delegation by the use of regular expressions. With these regular expressions we can limit certain aspects of the delegation such as its depth, group, timing or any other constraint related to security context or external data.

However, this approach distinguishes from the previous ones in that credential revocation is not allowed (neither in SJ) and it uses a central server for verifying every delegation.

Comparison

This paper has a different approach from the one needed since it uses a central authorisation server. Nevertheless, it was still worth mentioning because of the use of regular expressions to constraint delegations: this is something completely new and worth researching for future versions of SJ.

Concerning the comparisons with SecPAL and DL, all of them can limit the depth of re-delegations but this paper can impose some more constraints due to the regular expressions as mentioned before. Regarding Binder and GrDP, both of them can revoke certificates whereas in this approach, that is not possible.

Chapter 6

Implementation of Secure SJ

6.1 Transport Layer Security - Secure Remote Password

In order to integrate the TLS-SRP (Transport Layer Security - Secure Remote Password) protocols in SJ we tried to use already implemented TLS-SRP applications, since the aim of the project is to secure SJ in the best possible way and not to discover or implement new security protocols. However, there are not many implementations of this, so we aimed our efforts in trying to find a good SSL/TLS implementation so we could manually add the SRP cipher suite like RFC5054 [20] suggests.

SUN's SSL sockets

SUN's sockets do not have SRP integrated and to add it is quite complicated since SUN's code is very complex and the implementation is spread in several classes/packages. We would need several weeks to fully understand the code and modify it in the best way. Therefore, we continued searching for another approach that could be simpler.

OpenSSL

OpenSSL is the largest project implementing SSL/TLS sockets with an excellent cryptographic library. In addition, there are already some patches to add the SRP cipher suite to this project. The only problem is that it is implemented in C and we need it to run in Java. For that, we found a wrapper in C++ to serve like a "bridge" between C and Java. The major problem is that it uses libraries for MAC OS and not for Unix systems and that those libraries are completely obsolete (since the wrapper was created more than 10 years ago). Moreover, this wrapper is only for the client and we needed to implement another one for the whole server, which would be extremely complicated to debug since there was code in Java, C++ and C.

Mozilla NSS and JSS

Mozilla NSS is written in C, so again there is the problem to link it with Java. To solve that, Mozilla created JSS but it does not solve everything, since it uses Java Native Interface (JNI) which is extremely complex to debug: the slightest modification in the code may lead to several errors. JNI just throws a *"Native Exception"* without telling what part of the C code threw the error, similar to the famous *"Segmentation Fault"*.

There are also several reasons why we should not rely everything on JNI such as losing the portability that Java offers, the fact that it does not provide automatic garbage collection (we would need manual memory handling in C), among others.

GnuTLS

Same reasons as above: code written in C, hence the need to use JNI to export the methods to Java. After this, we realised that we should not even look for more C code and find everything

done in Java, or to write everything by ourselves.

Bouncy Castle

Bouncy Castle is a project comprising a series of Java cryptographic APIs. It was the best implementation of TLS we have found and it is not so hard to use. Unfortunately, only the client side is implemented, so we would have to implement the server side to match that implementation. That was the main reason we did not stick to this API but we will consider implementing the server as future work.

Currently, one of the project members is implementing the SRP to add it to the client side, so that the new version will also have the SRP, lacking only the server implementation. Therefore, this might be a good approach to follow in the future.

Jessie

The only complete Java implementation of the SSL sockets with SRP integrated. Unfortunately, the poor and incomplete documentation is a major drawback of this implementation. We spent a whole week trying to debug the errors occurred but they seem to vary on different machines, making the task even harder to solve. The documentation does not mention several important features such as the structure of the password files nor how to use them. We tried using the same as those from GnuTLS but with no success.

In addition, Jessie only has implementations of the older versions of the SSL and has implemented TLS 1.0, whereas the current version is 1.2. Hence, the modifications on this promising implementation would be quite substantial, making us giving up on this approach (along with the poor documentation).

Functional Implementation

As mentioned before, TLS is formed of three basic phases: negotiation of the algorithms supported, key exchange and authentication, and symmetric cipher encryption/message authentication.

Since we could not add the SRP in the list of cipher suites and modify the handshake, we were not able to use SRP as a key exchange protocol. Therefore, we used the SRP as a login before any of the three TLS phases. This way, everytime a new TLS connection is established, it first uses the SRP protocol for mutual authentication and if succeeded, connects both parties, otherwise an exception is thrown.

With this approach, the SSL Sockets used are those from SUN implementation and not some modified ones. We had just to extend those classes and implement the SRP protocol on the *accept* method, so everytime a new connection is established, the SRP is activated before the handshake. This may not be what we had originally planned, but it also protects the connection in the same way as the before. The major difference is that it uses other key exchange protocols to establish the session key, instead of using just the SRP for that purpose.

More technically, we created new classes extending the SSL ones from SUN in order to add the SRP:

```java
public class MySSLSocket extends SSLSocket
public class MySSLServerSocket extends SSLServerSocket implements Serializable
public class MySSLSocketFactory extends SSLSocketFactory implements Serializable
public class MySSLServerSocketFactory extends SSLServerSocketFactory implements
    Serializable
```

Then we modified the accept method in class *MySSLServerSocket* to perform the SRP after the acceptance of the connection. This method may give several exceptions if the authentication fails with the other peer or if the username already exists in the database. If an exception is raised, a

null value is returned instead of the actual socket. In addition, we added an additional message in the end of the protocol to inform the other party if this one had had an authentication failure.

```
public SSLSocket accept() throws IOException {
  SSLSocket m = (SSLSocket)s.accept();
  try{
    ... // Performs SRP

    // Receives response from client
    String resp = bufferedreader.readLine();

    if(serv.verify(Util.fromhex(resp))) {
      bufferedwriter.write("OK" + '\n');
    }
    else {
      bufferedwriter.write("ERROR" + '\n');
      throw new InvalidPasswordException("Authentication Failed [SRP]: Wrong password");
    }
    return new MySSLSocket(m);

  } catch(Exception e) {
    e.printStackTrace();
    return null;
  }
}
```

Dual to this is the method *performSRP* in class *MySSLSocketFactory* that is executed after the *createSocket* method. After performing the SRP, if it receives anything other than an OK from the other party, it throws an exception and returns *null*. As mentioned before this is not the best way of implementing the TLS-SRP as this protocol is just used for authentication and the session key resulting from it is discharged. A future version of this implementation will have the TLS handshake modified in order to use the session key resulting from the SRP. However, this implementation provides the same level of security, although it adds an overhead of an additional session key.

```
private SSLSocket performSRP(SSLSocket s) {
  try {
    ... // Performs SRP

    // Receives confirmation of everything
    str = bufferedreader.readLine();

    if (!str.equals("OK"))
      throw new InvalidPasswordException("Authentication Failed [SRP]: Wrong password");

    return new MySSLSocket(s);
  } catch(InvalidPasswordException e) {
    System.out.println("[Client] Authentication Failed in SRP");
    return null;
  }
}
```

In order to use the SSL from SUN, we have to configure it to use the protocols and the certificates we want. In our case, we opted by the X509 certificates. Then a Key manager is loaded from a file on the hard drive called *serverKeystore*, which password is *password*. Obviously this is not the most secure password but this was just created for testing purposes. In future versions it could be improved and it would not even be necessary if the real TLS-SRP was implemented, since no certificates would be needed. The code from the server side is:

```java
KeyManagerFactory mgrFact = KeyManagerFactory.getInstance("SunX509");
KeyStore serverStore = KeyStore.getInstance("JKS");

// Loads keystore
serverStore.load(new FileInputStream(Constants.DEFAULT_KEYSTORE), "password".
    toCharArray());
mgrFact.init(serverStore, "password".toCharArray());

// create a context and initialises it
SSLContext sslContext = SSLContext.getInstance("TLS");
sslContext.init(mgrFact.getKeyManagers(), null, null);
ssf = (SSLServerSocketFactory)sslContext.getServerSocketFactory();
```

Then for the client side the process is similar, although no Key managers are needed but only Trust managers. Note that in both the client and the server the Socket Factory is created by the SSL Context with the chosen specifications.

```java
// set up a trust manager so we can recognize the server
TrustManagerFactory trustFact = TrustManagerFactory.getInstance("SunX509");
KeyStore          trustStore = KeyStore.getInstance("JKS");

trustStore.load(new FileInputStream(Constants.DEFAULT_KEYSTORE), "password".toCharArray
    ());
trustFact.init(trustStore);

// create a context and set up a socket factory
SSLContext sslContext = SSLContext.getInstance("TLS");
sslContext.init(null, trustFact.getTrustManagers(), null);
ssf = (SSLSocketFactory)sslContext.getSocketFactory();
```

6.2 Delegation Cases

As mentioned in chapter 4, it is up to the session-sender to create the credential to authenticate the passive-party to the session-receiver. Since most of the times all parties are running in the same computer, they all have the same password file; and since it is not allowed for a user to have different passwords, the session-sender creates a new username and password for only that session. This way, the security level is higher because the credentials are only used once and then discharged — similar to the one-time pad. In order to do this, the session-sender generates a random username of eight chars, a password with the same length and a salt of size five. However there is always the chance that the generated username and password can be the same after some time, since there is no perfect random generator (being this possibility ignored for obvious reasons).

```java
private SJCredentials generateUserPassword() {
  String user = generateRandom(8);
  String pwd = generateRandom(8);
  String salt = generateRandom(5);
  String index = generateIndex();
  return new SJCredentials(user,pwd,index,salt.getBytes());
}
```

The weakness on this approach is that the password is sent in plaintext to both parties and further investigation should be done in order to prevent this from happening. Nevertheless, some security is preserved as the channels in which the password is sent are SSL and should protect the password from being intercepted by a malicious party.

Getting into more detailed, the session sender starts by sending the new generated Credential to the session receiver after the delegation signal:

```
if (s.getConnection().getTransportName().equals(Constants.HTTPS)) {
  cred = generateUserPassword();
  ser.writeObject(cred);
}
```

On the other hand, the receiving party receives that credential and stores it in its password file. This may throw exceptions while receiving the credential or if it cannot read the password file whilst storing the new information. As we can see, after storing the credential in the password file, the variable containing the credential is discharged to avoid memory leakages.

```
if (s.getConnection().getTransportName().equals(Constants.HTTPS)) {
  try {
    SJCredentials cred = (SJCredentials) receive();

    PasswordFile pwf = new PasswordFile(Constants.DEFAULT_PASS);
    pwf.add(cred.getUser(), cred.getPwd(), cred.getSalt(), cred.getIndex());

    cred = null; // Delete credentials from memory to prevent leaks
  } catch (ClassNotFoundException e) {
    throw new SJIOException("[SJSessionProtocolsImpl] Unable to receive credentials: "
        + e);
  } catch (IOException e) {
    throw new SJIOException("[SJSessionProtocolsImpl] Unable to read password file: " +
         e);
  }
}
```

These credentials are only sent in the case that the transport being used is the HTTPS. While sending back the credential to the passive party, it sends null values for other protocols and the credentials in the secure one. This may leave space for sending useful information on the other transports in future versions of SJ. Moreover, by extending the Delegation Signal, there is no overhead as no additional messages are being sent.

```
if (s.getConnection().getTransportName().equals(Constants.HTTPS)) {
  // B -> A: <IPc, Pc, CredA>
  foo.writeControlSignal(new SJDelegationSignal(hostName, port, cred.getUser(), cred.
      getPwd()));
  cred = null; // Deletes credential from memory to prevent memory leaks
}
else // B -> A: <IPc, Pc>
  foo.writeControlSignal(new SJDelegationSignal(hostName, port, null, null));
```

In response to this, the passive party then connects to the session receiver using this line:

```
SJRuntime.reconnectSocket(s, ds.getHostName(), ds.getPort(), ds.getUserName(), ds.
    getPwd());
```

Note that the last two arguments can be *null* in case the protocol is not secure.

Finally, in case 4 of the delegation (simultaneous delegation) a new connection is opened between party C and D (cf chapter 4) using the new credentials. Once again the last two arguments can be null in case the protocol is not HTTPS.

```
if (simdel && origReq) {
  SJDelegationSignal ds = (SJDelegationSignal) bar.get(bar.size() - 1).getContent();

  conn = sjtm.openConnection(ds.getHostName(), ds.getPort(), params, ds.getUserName(),
      ds.getPwd());
}
```

Delegation has been implemented in the Forwarding protocol for all four delegation cases in SJ [32]. The Resending protocol is done in a similar way and is already formalised in detail in chapter 4, being unnecessary to implement it for the scope of this project, since its implementation is similar to the Forwarding protocol.

Chapter 7

Large Protocols in Secure SJ

The full source code of these large protocols here explained can be found in [1].

7.1 Online Shopping

The program provided is an online shopping company where a customer buys certain products from a shop, followed by the shop delegating that purchase to the Payments department.

The Shop starts by sending a list of all available products along with their prices. Then, the customer starts a loop of choosing the products he wants to buy and selects them by sending the product with the respective quantity. The shop replies with the current total cost of the basket. Then the customer can either Exit the program without buying anything (Exit branch) or proceed to checkout (Checkout branch). If he does the latter, the Shop sends the total cost of the basket to Payments and delegates the session with the Customer to that department. The customer then sends his credit card information and receives the date when the products were dispatched. These last two operations are taken with the Payments department, being completely transparent to the Customer who thinks he is still communicating with the Shop. A more detailed (and visual) explanation is on Figure 7.1.

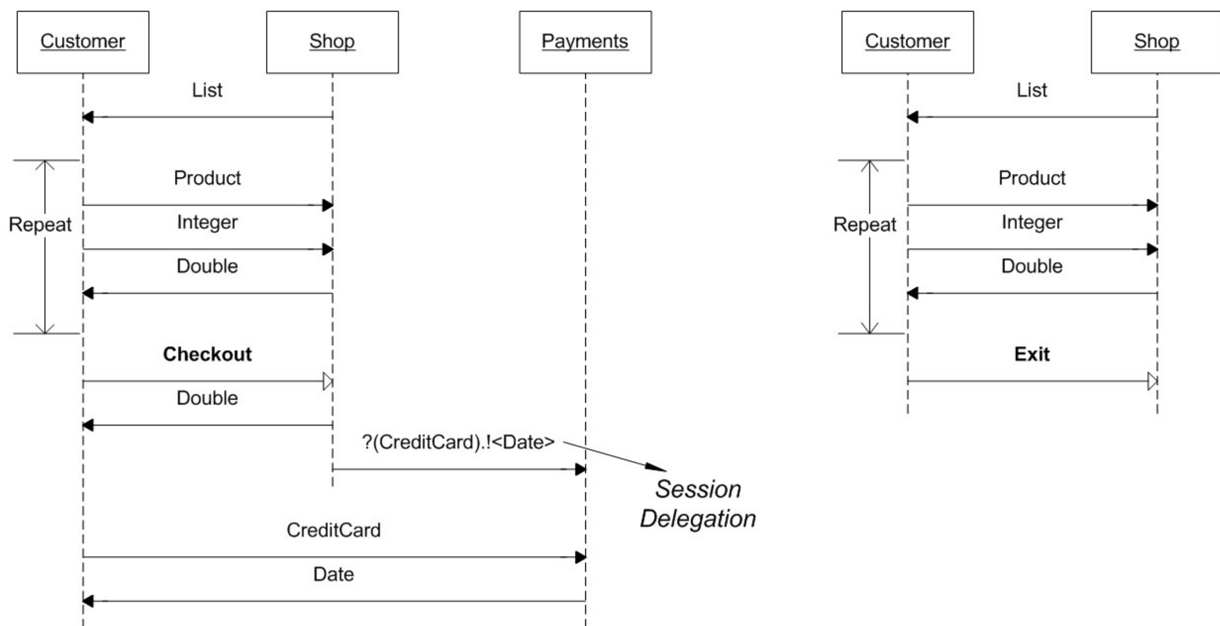


Figure 7.1: Online Shopping diagram

7.1.1 Protocol details

The detailed protocols specification is in Figure 7.2.

```
final noalias protocol options {
   !{
         PURCHASE: ![!<Product>.!<Integer>.?(Double)]*,
         BASKET: ,
         OTHER:
   }
}

final noalias protocol customerShop {
   cbegin.
      ?(List).
      ![@(options)]*.
      !{
            CHECKOUT: ?(Double).!<CreditCard>.?(Date),
            EXIT:
      }
}
```

Customer

```
final noalias protocol options {
   ?{
         PURCHASE: ?[?(Product).?(Integer).!<Double>]*,
         BASKET: ,
         OTHER:
   }
}

final noalias protocol shopCustomer {
   sbegin.
      !<List>.
      ?[@(options)]*.
      ?{
            CHECKOUT: !<Double>.?(CreditCard).!<Date>,
            EXIT:
      }
}
```

Shop

```
final noalias protocol shopPayments {
   cbegin.
      !<    //Session Delegation
         ?(CreditCard).
         !<Date>
      >
}
```

Shop

```
final noalias protocol paymentsCustomer {
   ?(CreditCard).
   !<Date>
}

final noalias protocol paymentsShop {
   sbegin.
      ?(@(paymentsCustomer))
}
```

Payments

Figure 7.2: All communication protocols of the Online Shopping

As the names suggest the protocol customerShop and shopCustomer interact with each other, leaving the customer unaware of a future delegation. As in every protocol in SJ, in order for these protocols to interact they need to be dual, otherwise an exception is thrown.

The customerShop protocol describes the interaction between the Customer and the Shop which starts by receiving a *List* of all available products, followed by a loop of all possible options (Purchase, Basket and Other). The last two branches do not perform any action regarding session operations (only internal operations), whereas in the Purchase branch a new loop is initiated. This loop sends a *Product* and an *Integer* referring to the product and the respective quantity and receives a *Double* with the current total cost of the basket. When both the loops are finished, two new branches are available: we can either Checkout or Exit the program. The latter one just terminates the program whereas the former one receives a *Double* with the final cost of the basket and sends the *credit card* information, terminating by receiving a *Date* concerning the dispatch date of the whole basket. As mentioned before, these last two operations are delegated to Payments department without the customer being aware of it.

We can see the delegation in protocol shopPayments and the reception in protocol paymentsShop. The instructions for the delegation are basically the sending of a series of sends and receives, which the other party receives and executes them with the passive party. Finally, the protocol paymentsCustomer is about the instructions that were delegated to the Payments department. More details about how the protocols work can be found in [13, 32].

7.1.2 The Network

However, in order for the customer to connect to the shop, it has to pass the Network. The Network works as a router, in the way that it forwards all the messages from the customer to the shop and vice-versa.

As we can see, running this program with TCP or any connection other than HTTPS, the Network is able to intercept important information like credit cards:

CREDIT CARD INFORMATION FOUND!!!!
sr ?shopping.CreditCardgUh??4? ?L ?namet ?Ljava/lang/String;L ?numberq ?L secCodeq
?xpt **Mike Yoon**t ?**8798 2839 2934 2940**t ?**293**y

With this valuable information (emphasised in bold) the Network could, among other things, change the information (jeopardising message integrity), replay it later for purchasing goods for own interest, or just block this information, resulting in the cancelation of the purchase.

Another attack could be the non-repudiation of the purchase, since there is no authentication method that guarantees that the customer is who he claims to be.

Fortunately, all the attacks shown are prevented with the usage of HTTPS with the TLS-SRP. The SRP provides protection against the non-repudiation attacks, whereas all other attacks are protected with the TLS protocol.

7.1.3 Execution of the program

In order to run this program, the user needs to run the following programs in this specific order:

– **Payments**, with arguments: *d d <portP>*

– **Shop**, with arguments: *d d <portS> <hostP> <portP>*

– **Network**, with arguments: *<portN> <hostS> <portS>*

– **Customer**, with arguments: *d d <hostN> <portN>*

Using the arguments '*d d*', it uses the default TCP connection as transport and we can see the previous error message on the Network. In order to use a secure connection, one should replace '*d d*' by '*s s*' and the error message no longer appears since all the information is encrypted and not passed in plaintext.

This is a simple program where the Network only tracks credit card informations, but we can see the importance of secure connections in real applications.

7.1.4 User's Manual

The program starts with a menu for choosing to either purchase new products, view the basket or quit the program. This menu is shown below:

```
Welcome to Online Shopping
--------------------------
Choose one of the following options:
1 - Purchase products
2 - View current basket
3 - Exit
```

If we choose the first option, we can start buying new products. After entering the second menu, we can choose which product we want to buy, followed by the quantity we want. We then receive a confirmation that the product was indeed added to the basket and the current price of the basket.

```
Choose one of the products available:
1 - Laptop
2 - TV
3 - DVD Player
4 - Mobile Phone
5 - Exit to main menu
```

When we are satisfied with the basket, we can proceed to checkout. In order to do so, we should return to the main menu and choose option 2. If the basket is empty we get a message like:

```
--------------
Current Basket
--------------
EMPTY!
Press Enter to return to main menu
```

and we cannot do anything else apart from returning to the main menu. Otherwise we can choose to checkout and conclude the program:

```
--------------
Current Basket
--------------
1 Laptop
2 TV
Proceed to checkout? (y/n)
```

Upon checking out, we receive a confirmation of the final value of the basket along with the dispatch date. We can see here one of the most important features of delegation by noticing that the delegation to the Payments department was completely transparent to the Customer. The customer always thinks he is interacting with the Shop and with no other party.

```
--------------
Current Basket
--------------
1 Laptop
2 TV
Proceed to checkout? (y/n)
y
Total cost: 759.77
Dispatch date: Sun Aug 30 01:21:49 BST 2009
Thank you for shopping with us!
```

7.2 Primes

This program was implemented to solve one typical parallelisation problem: calculating all prime numbers up until one given number. The algorithm implemented was the Sieve of Eratosthenes [6] and the program is composed by three parties: the client, the service and the workers. The more workers we have, the fastest the program calculates the prime numbers. However, there is a limit from where it will not make a difference increasing the workers and sometimes it may even decrease the performance. Therefore the user must study the number of workers before running this program.

The Client starts by sending the limit number to calculate the primes and receives a list of all prime number below the specified input. On the other hand, most of the work lies on the Service, which starts by creating a list of all numbers from two up to the requested number. Then, using multiple threads, it sends even parts of that list to all workers and the current number being iterated. The workers then delete all multiples of the number being iterated from their list and return the new list to the Service, which continues to perform the same operations whilst the number being iterated is less than the square root of the requested number. Each thread in the Service accesses a shared memory variable with the current list of possible primes and every thread modifies that variable in order to achieve the final result. Moreover, there is a main loop which controls every

thread and returns the final result to the Client.

The list of operations can be better visualised in Figure 7.3.

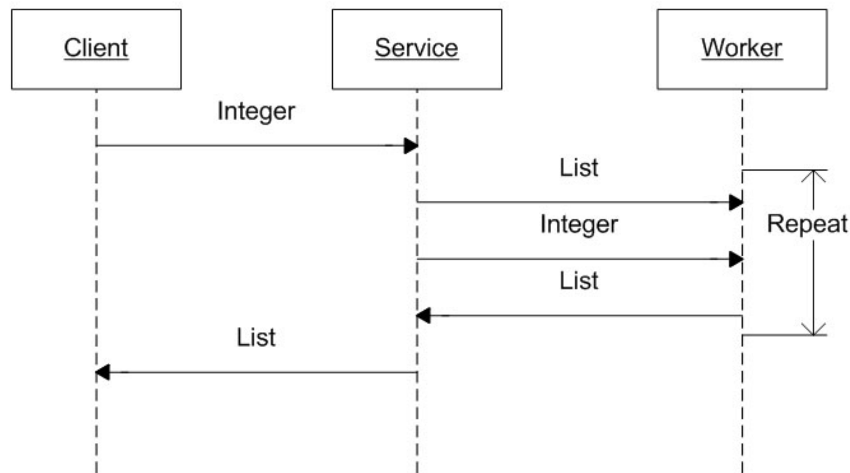


Figure 7.3: Prime Numbers diagram

7.2.1 Protocols

The protocols for this program are relatively simple since there is no delegation and most of the work is performed in the Service, like mentioned before, and are listed in Figure 7.4.

Regarding the interaction between the Client and the Service, the former just sends the requested *Integer* number whereas the latter replies with the *List* of the results. Concerning the communication between the Service and the Workers, there is a loop where the Service sends the chunk of the *List* for the worker to perform the calculations, along with the current prime being iterated. The Worker then replies with a *List* of all number except the multiples of the prime sent, which continues being iterated until every prime number is found and the program concludes.

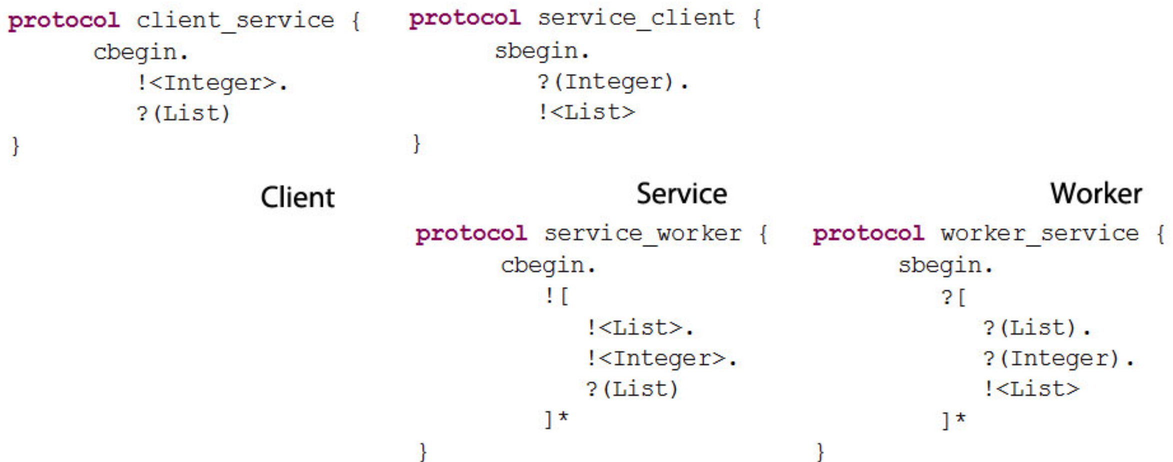


Figure 7.4: Prime Numbers Protocols

7.2.2 Execution of the program

In order to run this program, the user needs to run the following programs in this specific order:

– **Worker1**, with arguments: $d\ d < portW_1 >$

....

– **Workern**, with arguments: $d\ d < portW_n >$

– **Service**, with arguments: $d\ d <portS> < hostW_1 > < portW_1 > \ldots < hostW_n > < portW_n >$

– **Client**, with arguments: $d\ d <hostN> <portN>$

We can choose any amount of workers to share the workload of calculating the prime numbers but we then need to specify all workers whilst running the Service.

Like in the Online Shopping, we can use '*d d*' for using normal TCP connection or '*s s*' for a secure connection. In the current version of this program, this works normally. However, the way it was implemented, running in a secure mode should be avoided, since it asked the username and password all times it connected to the worker. Since arrays of sessions are not yet implemented on the current version of SJ, this program created a new session everytime it needed to send information to a specific worker. Therefore, when this feature is implemented on SJ, this program should be modified in order to store all sessions in an array and then just using the already initialised sessions to send the information.

Temporarily, this issue was solved with the shared memory approach using multiple threads. Every thread now accesses the same list of possible primes and operates on this list before sending the work to the Workers and gathering the final result. This approach is also good: the only possible disadvantage is the synchronization between the shared variable, which can be solved with the array of sessions as mentioned before.

Chapter 8

π-Calculus Delegation Protocols and their Correctness

This chapter is composed by the modelling of all delegation cases of both protocols into π-calculus. We start by giving the syntax of this instance of the π-calculus which is an extension of [30, 33, 36, 45], followed by the specification of the reduction rules. Secondly, we model all protocols into π-calculus and discuss their correctness. In order to do that, the three main properties of SJ were extended to include a security property. It is just left to formalise those properties and proof their correctness, which is done in the end of this chapter.

8.1 Syntax and Reduction

8.1.1 Process Syntax

P, Q	$::=$	$u\,(x\colon S).P$	Session Accept
	$\mid$	$\overline{u}\,(x\colon S).P$	Session Request
	$\mid$	$k\,!\langle e\rangle;\ P$	Output
	$\mid$	$k\,?(x);\ P$	Input
	$\mid$	$k \triangleleft l;\ P$	Select
	$\mid$	$k \triangleright \{l_1\colon P_1, ..., l_n\colon P_n\}$	Branch
	$\mid$	$\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$	Conditional
	$\mid$	$P \mid Q$	Parallel Composition
	$\mid$	$(\boldsymbol{\nu}\, a\colon \langle S\rangle)\ P$	Shared Channel Restriction
	$\mid$	$(\boldsymbol{\nu}\, s)\ P$	Session Channel Restriction
	$\mid$	$\texttt{def}\ D\ \texttt{in}\ P$	Agent Definition Scope
	$\mid$	$X\langle\tilde{e}\rangle$	Agent Instance
	$\mid$	$\mathbf{0}$	Completed Process
	$\mid$	$s\,[S]\colon \vec{h}$	Session Endpoint Configuration
	$\mid$	$\mathsf{make}\ (Cred);\ P$	Credentials Creation
	$\mid$	$\mathsf{close}(k);\ P$	Session Closure

where:

u, u'	$::=$	$a, b, c \mid x, y, z$	Shared channels
k, k'	$::=$	$s, \overline{s} \mid x, y, z$	Session endpoints

This extension of π-calculus is asynchronous which means we need to have queues for message sending or receiving. We write $\vec{h}$ for a vector $h_1...h_n$, which describes the message queues. Shared channels are used to initialise the sessions and we use the $\overline{s}$ notation to describe the dual of the session s. Moreover, we use $u\,(x : S).P$ to accept a session requested by $\overline{u}\,(x : S).P$ and communications are carried out with $k\,!\langle e\rangle;\ P$ and $k \triangleleft l;\ P$ and their duals $k\,?(x);\ P$ and $k \triangleright \{l_1 :$

$P_1, ..., l_n\colon P_n\}$, respectively. The conditional and the parallel composition are self-explanatory and $(\boldsymbol{\nu}\, a\colon \langle S\rangle)\ P$ and $(\boldsymbol{\nu}\, s)\ P$ restrict the channel or the session to the scope of P. $\mathtt{def}\ D\ \mathtt{in}\ P$ defines a new agent, which is useful for recursions and $X\langle\tilde{e}\rangle$ defines a new instance of an agent. The new primitives in this modelling are the make (Cred); P and the $\mathsf{close}(k);\ P$, which create a new credential for the session delegation and close the session k, respectively.

8.1.2 Session Type Syntax

$$
\begin{array}{lll}
S & ::= & !\langle T\rangle; S \\
 & | & ?(T); S \\
 & | & \oplus\{l_1\colon S_1, ..., l_n\colon S_n\} \\
 & | & \&\{l_1\colon S_1, ..., l_n\colon S_n\} \\
 & | & \mathtt{end} \\
 & | & \mu\,\mathbf{t}.S \\
 & | & \mathbf{t}
\end{array}
$$

where:

$$
\begin{array}{lllr}
T & ::= & U \mid S & \text{Message types} \\
U & ::= & \mathsf{bool} \mid \mathsf{nat} \mid \mathsf{String} \mid \mathsf{CredT} \mid \mathsf{CloseSig} \mid \langle S\rangle & \text{Constant value types}
\end{array}
$$

$!\langle T\rangle; S$ and $?(T); S$ represent the output and the input, respectively, of a value of type T with the subsequent session of type S. $\oplus\{l_1\colon S_1, ..., l_n\colon S_n\}$and $\&\{l_1\colon S_1, ..., l_n\colon S_n\}$represent the selection type of one of the labels and its dual – the branching type. $\mu\,\mathbf{t}.S$ is the recursive type, whereas $\mathbf{t}$ is a type variable. Finally, $\mathtt{end}$ is the terminating type.

8.1.3 Reduction Rules

$$
\frac{}{\mathsf{close}(s);\ P \mid s\,[S]\colon \vec{h} \longrightarrow P}\ \text{Close}
$$

$$
\frac{Cred \text{ fresh in } P}{\mathsf{make}\ (Cred);\ P \longrightarrow P}\ \text{Make}
$$

All reduction rules are based on previous papers [30, 33, 36, 45] except these two which are a specific addition to this implementation. On the Close rule we can see that there is a session s with the respective queue, and after the close instruction this session disappears and the process continues. In the make rule, we simply want to ensure that the Credential is fresh for that protocol, in order to achieve several properties, stated later in this chapter. All the other reduction rules can be found in appendix C.1.

8.1.4 Typing rules

$$
\frac{\Theta; \Gamma \vdash P \triangleright \Sigma \cdot k\colon S}{\Theta; \Gamma \vdash P; \mathsf{close}(k) \triangleright \Sigma \cdot k\colon S; \mathtt{end}}\ \text{Close}
$$

The typing rules are identical to those of the previous papers and for further explanation, they should be consulted. All judgements are in the form $\Theta; \Gamma \vdash \mathrm{P} \triangleright \Sigma$ which can be read as: in the environment $\Theta; \Gamma$, process P has the typing Σ. The only addition on the typing rules is the rule related to the closing of a session: this rule simply adds an end type to whatever is before the close instruction. The list of all typing rules is in appendix C.2.

8.2 Resending Protocol

8.2.1 Case 1

This is the most used case in which there is a simple delegation between three parties and the passive party is waiting for a receiving instruction.

A - $[[P \mid s_{\text{AB}} : \langle S \rangle \vec{h}]]$

$$\text{A} = s_{\text{AB}}\,?(S', x_{\text{IPc}}, y_{\text{pc}}, z_{\text{CredA}});\ s_{\text{AB}}\,!\langle ACK \rangle;\ \mathsf{close}(s_{\text{AB}});\ \overline{y_{\text{pc}}}\,(x\colon S).x\,!\langle z_{\text{CredA}} \rangle;\ x \triangleright \{\textbf{Success} : x\,!\langle LM(S-S') \rangle, \textbf{Fail} : \mathsf{close}(x)\}$$

It starts by receiving the information regarding the session receiver, along with the credential, followed by sending an ACK confirming the reception. Then, it closes the session it has with B, followed a connection to the receiver and sending the credential. If the connection is accepted, it sends information about the lost messages; otherwise the new connection is closed.

Typing

$\Gamma \vdash \mathbf{0} \triangleright s_{\text{AB}} \colon \mathsf{end} \cdot x \colon \mathsf{end}$

$\Gamma \vdash x \triangleright \{\textbf{Success} : x\,!\langle LM(S-S') \rangle, \textbf{Fail} : \mathsf{close}(x)\}$
$\quad \triangleright s_{\text{AB}} \colon \mathsf{end} \cdot x \colon \&\{\textbf{Success} :!\langle String \rangle, \textbf{Fail} : \mathsf{end}\}$

$\Gamma \vdash x\,!\langle z_{\text{CredA}} \rangle;\ x \triangleright \{\textbf{Success} : x\,!\langle LM(S-S') \rangle, \textbf{Fail} : \mathsf{close}(x)\}$
$\quad \triangleright s_{\text{AB}} \colon \mathsf{end} \cdot x :!\langle CredT \rangle; \&\{\textbf{Success} :!\langle String \rangle, \textbf{Fail} : \mathsf{end}\}$

$\Gamma \vdash \overline{y_{\text{pc}}}\,(x\colon S).x\,!\langle z_{\text{CredA}} \rangle;\ x \triangleright \{\textbf{Success} : x\,!\langle LM(S-S') \rangle, \textbf{Fail} : \mathsf{close}(x)\} \triangleright s_{\text{AB}} \colon \mathsf{end}$

$\Gamma \vdash s_{\text{AB}}\,!\langle ACK \rangle;\ \mathsf{close}(s_{\text{AB}});\ \overline{y_{\text{pc}}}\,(x\colon S).x\,!\langle z_{\text{CredA}} \rangle;$
$\quad x \triangleright \{\textbf{Success} : x\,!\langle LM(S-S') \rangle, \textbf{Fail} : \mathsf{close}(x)\} \triangleright s_{\text{AB}} :!\langle ACK \rangle; \mathsf{end}$

$\Gamma \vdash s_{\text{AB}}\,?(S', x_{\text{IPc}}, y_{\text{pc}}, z_{\text{CredA}});\ s_{\text{AB}}\,!\langle ACK \rangle;\ \mathsf{close}(s_{\text{AB}});\ \overline{y_{\text{pc}}}\,(x\colon S).x\,!\langle z_{\text{CredA}} \rangle;$
$\quad x \triangleright \{\textbf{Success} : x\,!\langle LM(S-S') \rangle, \textbf{Fail} : \mathsf{close}(x)\} \triangleright s_{\text{AB}} :?(S, nat, nat, CredT); !\langle ACK \rangle; \mathsf{end}$

From this we can conclude that the session s_{AB} receives the session information, followed by the sending of the ACK. On the other hand, the connection with C is composed by the sending of the credential it has previously received, terminating by the branching type (with the sending of a String in the Success case).

B - $[[Q \mid \overline{s_{\text{AB}}} : \langle S' \rangle \vec{h'} \mid s'_{\text{BC}} : \langle S'' \rangle \vec{h''}]]$

$$\text{B} = \mathsf{make}\,(\text{CredA});\ s'_{\text{BC}}\,!\langle CredA \rangle;\ b\,(x_{\text{pc}}\colon S).\overline{s_{\text{AB}}}\,!\langle S', IPc, x_{\text{pc}}, CredA \rangle;\ \overline{s_{\text{AB}}}\,?(x_{\text{ACK}});\ \mathsf{close}(\overline{s_{\text{AB}}})$$

B starts by creating a fresh credential for that delegation, which is sent to C and later to A along with other important information on how to connect to C. In the end it receives an ACK from A indicating that everything went well, so it can close the session it has with A.

Typing

$\Gamma \vdash \mathbf{0} \triangleright \overline{s_{\text{AB}}} \colon \mathsf{end} \cdot s'_{\text{BC}} \colon \mathsf{end} \cdot x_{\text{pc}} \colon \mathsf{end}$

$\Gamma \vdash \overline{s_{\text{AB}}}\,?(x_{\text{ACK}});\ \mathsf{close}(\overline{s_{\text{AB}}}) \triangleright \overline{s_{\text{AB}}} :?(ACK); \mathsf{end} \cdot s'_{\text{BC}} \colon \mathsf{end} \cdot x_{\text{pc}} \colon \mathsf{end}$

$\Gamma \vdash \overline{s_{\text{AB}}}\,!\langle S', IPc, x_{\text{pc}}, CredA \rangle;\ \overline{s_{\text{AB}}}\,?(x_{\text{ACK}});\ \mathsf{close}(\overline{s_{\text{AB}}})$
$\quad \triangleright \overline{s_{\text{AB}}} :!\langle S, nat, nat, CredT \rangle; ?(ACK); \mathsf{end} \cdot s'_{\text{BC}} \colon \mathsf{end} \cdot x_{\text{pc}} \colon \mathsf{end}$

$\Gamma \vdash b\,(x_{\text{pc}}\colon S).\overline{s_{\text{AB}}}\,!\langle S', IPc, x_{\text{pc}}, CredA\rangle;\ \overline{s_{\text{AB}}}\,?(x_{\text{ACK}});\ \mathsf{close}(\overline{s_{\text{AB}}})$
$\quad \triangleright \overline{s_{\text{AB}}}:!\langle S, nat, nat, CredT\rangle; ?(ACK); \mathtt{end} \cdot s'_{\text{BC}}: \mathtt{end}$

$\Gamma \vdash s'_{\text{BC}}\,!\langle CredA\rangle;\ b\,(x_{\text{pc}}\colon S).\overline{s_{\text{AB}}}\,!\langle S', IPc, x_{\text{pc}}, CredA\rangle;\ \overline{s_{\text{AB}}}\,?(x_{\text{ACK}});\ \mathsf{close}(\overline{s_{\text{AB}}})$
$\quad \triangleright \overline{s_{\text{AB}}}:!\langle S, nat, nat, CredT\rangle; ?(ACK); \mathtt{end} \cdot s'_{\text{BC}}:!\langle CredT\rangle; \mathtt{end}$

From this typification we can infer that B sends the credential to C and that it also sends all the session information to A, followed by the receiving of the ACK.

C - $[[R \mid \overline{s'_{\text{BC}}} : \langle S^3\rangle \vec{h^3}]]$

C $= \overline{s'_{\text{BC}}}\,?(x_{\text{CredA}});\ (\boldsymbol{\nu}\, pc)\ (\bar{b}\,(pc\colon S).pc\,(x\colon S).x\,?(y_{\text{CredA}});$
$\mathtt{if}\ x_{\text{CredA}} == y_{\text{CredA}}\ \mathtt{then}\ x \triangleleft \textbf{Success};\ x\,?(x_{\text{LM}})\ \mathtt{else}\ x \triangleleft \textbf{Fail};\ \mathsf{close}(x);\ \mathsf{close}(pc))$

To conclude this case of the delegation, the session receiver starts by receiving the credential from the session sender. Then, it creates a new port *pc* to accept a new connection from the passive party. It then sends that port to B on the shared channel *b*, followed by the waiting of the connection from A. This process terminates by receiving the credential from A and checking if this credential matches with the one received from B: if it does, it accepts the information regarding the lost messages; otherwise the new session is closed, together with the opened port *pc*.

Typing

$\Gamma \vdash \mathbf{0} \triangleright \overline{s'_{\text{BC}}}: \mathtt{end} \cdot pc: \mathtt{end} \cdot x: \mathtt{end}$

$\Gamma \vdash x\,?(x_{\text{LM}}) \triangleright \overline{s'_{\text{BC}}}: \mathtt{end} \cdot pc: \mathtt{end} \cdot x:?(String); \mathtt{end}$

$\Gamma \vdash x \triangleleft \textbf{Success};\ x\,?(x_{\text{LM}}) \triangleright \overline{s'_{\text{BC}}}: \mathtt{end} \cdot pc: \mathtt{end} \cdot x: \oplus\{\textbf{Success}:?(String); \mathtt{end}, \textbf{Fail}: \mathtt{end}\}$

Let $Q = \mathtt{if}\ x_{\text{CredA}} == y_{\text{CredA}}\ \mathtt{then}\ x \triangleleft \textbf{Success};\ x\,?(x_{\text{LM}})\ \mathtt{else}\ x \triangleleft \textbf{Fail};\ \mathsf{close}(x);\ \mathsf{close}(pc)$

$\Gamma \vdash Q \triangleright \overline{s'_{\text{BC}}}: \mathtt{end} \cdot pc: \mathtt{end} \cdot x: \oplus\{\textbf{Success}:?(String); \mathtt{end}, \textbf{Fail}: \mathtt{end}\}$

$\Gamma \vdash x\,?(y_{\text{CredA}});\ Q \triangleright \overline{s'_{\text{BC}}}: \mathtt{end} \cdot pc: \mathtt{end} \cdot x:?(CredT); \oplus\{\textbf{Success}:?(String); \mathtt{end}, \textbf{Fail}: \mathtt{end}\}$

$\Gamma \vdash pc\,(x\colon S).x\,?(y_{\text{CredA}});\ Q \triangleright \overline{s'_{\text{BC}}}: \mathtt{end} \cdot pc: \mathtt{end}$

$\Gamma \vdash \bar{b}\,(pc\colon S).pc\,(x\colon S).x\,?(y_{\text{CredA}});\ Q \triangleright \overline{s'_{\text{BC}}}: \mathtt{end}$

$\Gamma \vdash \overline{s'_{\text{BC}}}\,?(x_{\text{CredA}});\ (\boldsymbol{\nu}\, pc)\ (\bar{b}\,(pc\colon S).pc\,(x\colon S).x\,?(y_{\text{CredA}});\ Q) \triangleright \overline{s'_{\text{BC}}}:?(CredT); \mathtt{end}$

Process C is typed with the receiving of a credential type from B and from the new connection with A it is typed as the reception of another credential, followed by the selection type in which it chooses the Success branch if both credentials match, or the Fail branch otherwise. In the success branch it receives a String referring the lost messages.

8.2.2 Case 2

This cases differs from the first one in that the passive party (A) has already finished its party of the session when the delegation takes place.

A - $[[P \mid s_{\mathrm{AB}} : \langle S \rangle \vec{h}]]$

$\mathrm{A} = s_{\mathrm{AB}}\,?(S', x_{\mathrm{IPc}}, y_{\mathrm{pc}}, z_{\mathrm{CredA}});\ \mathsf{close}(s_{\mathrm{AB}});\ \overline{y_{\mathrm{pc}}}\,(x\colon S).x\,!\langle z_{\mathrm{CredA}} \rangle;$
$x \triangleright \{\mathbf{Success} : x\,!\langle LM(S - S') \rangle, \mathbf{Fail} : \mathsf{close}(x)\}$

The only difference from party A in Case 1 is that it does not send an ACK after receiving the session information. Apart from that it is identical to the previous one.

Typing

$\Gamma \vdash \mathbf{0} \triangleright s_{\mathrm{AB}} \colon \mathsf{end} \cdot x \colon \mathsf{end}$

$\Gamma \vdash x \triangleright \{\mathbf{Success} : x\,!\langle LM(S - S') \rangle, \mathbf{Fail} : \mathsf{close}(x)\}$
$\quad \triangleright s_{\mathrm{AB}} \colon \mathsf{end} \cdot x \colon \&\{\mathbf{Success} :!\langle String \rangle, \mathbf{Fail} : \mathsf{end}\}$

$\Gamma \vdash x\,!\langle z_{\mathrm{CredA}} \rangle;\ x \triangleright \{\mathbf{Success} : x\,!\langle LM(S - S') \rangle, \mathbf{Fail} : \mathsf{close}(x)\}$
$\quad \triangleright s_{\mathrm{AB}} \colon \mathsf{end} \cdot x :!\langle CredT \rangle; \&\{\mathbf{Success} :!\langle String \rangle, \mathbf{Fail} : \mathsf{end}\}$

$\Gamma \vdash \overline{y_{\mathrm{pc}}}\,(x\colon S).x\,!\langle z_{\mathrm{CredA}} \rangle;\ x \triangleright \{\mathbf{Success} : x\,!\langle LM(S - S') \rangle, \mathbf{Fail} : \mathsf{close}(x)\} \triangleright s_{\mathrm{AB}} \colon \mathsf{end}$

$\Gamma \vdash s_{\mathrm{AB}}\,?(S', x_{\mathrm{IPc}}, y_{\mathrm{pc}}, z_{\mathrm{CredA}});\ \mathsf{close}(s_{\mathrm{AB}});\ \overline{y_{\mathrm{pc}}}\,(x\colon S).x\,!\langle z_{\mathrm{CredA}} \rangle;$
$\quad x \triangleright \{\mathbf{Success} : x\,!\langle LM(S - S') \rangle, \mathbf{Fail} : \mathsf{close}(x)\} \triangleright s_{\mathrm{AB}} :?(S, nat, nat, CredT); \mathsf{end}$

B - $[[Q \mid \overline{s_{\mathrm{AB}}} : \langle S' \rangle \vec{h'} \mid s'_{\mathrm{BC}} : \langle S'' \rangle \vec{h''}]]$

$\mathrm{B} = \mathsf{make}\ (\mathrm{CredA});\ s'_{\mathrm{BC}}\,!\langle CredA \rangle;\ b\,(x_{\mathrm{pc}}\colon S).\overline{s_{\mathrm{AB}}}\,!\langle S', IPc, x_{\mathrm{pc}}, CredA \rangle;\ \mathsf{close}(\overline{s_{\mathrm{AB}}})$

Similar to A, the only difference is that in this case, B does not receive any ACK.

Typing

$\Gamma \vdash \mathbf{0} \triangleright \overline{s_{\mathrm{AB}}} \colon \mathsf{end} \cdot s'_{\mathrm{BC}} \colon \mathsf{end} \cdot x_{\mathrm{pc}} \colon \mathsf{end}$

$\Gamma \vdash \overline{s_{\mathrm{AB}}}\,!\langle S', IPc, x_{\mathrm{pc}}, CredA \rangle;\ \mathsf{close}(\overline{s_{\mathrm{AB}}}) \triangleright \overline{s_{\mathrm{AB}}} :!\langle S, nat, nat, CredT \rangle; \mathsf{end} \cdot s'_{\mathrm{BC}} \colon \mathsf{end} \cdot x_{\mathrm{pc}} \colon \mathsf{end}$

$\Gamma \vdash b\,(x_{\mathrm{pc}}\colon S).\overline{s_{\mathrm{AB}}}\,!\langle S', IPc, x_{\mathrm{pc}}, CredA \rangle;\ \mathsf{close}(\overline{s_{\mathrm{AB}}}) \triangleright \overline{s_{\mathrm{AB}}} :!\langle S, nat, nat, CredT \rangle; \mathsf{end} \cdot s'_{\mathrm{BC}} \colon \mathsf{end}$

$\Gamma \vdash s'_{\mathrm{BC}}\,!\langle CredA \rangle;\ b\,(x_{\mathrm{pc}}\colon S).\overline{s_{\mathrm{AB}}}\,!\langle S', IPc, x_{\mathrm{pc}}, CredA \rangle;\ \mathsf{close}(\overline{s_{\mathrm{AB}}})$
$\quad \triangleright \overline{s_{\mathrm{AB}}} :!\langle S, nat, nat, CredT \rangle; \mathsf{end} \cdot s'_{\mathrm{BC}} :!\langle CredT \rangle; \mathsf{end}$

C - $[[R \mid \overline{s'_{\mathrm{BC}}} : \langle S^3 \rangle \vec{h^3}]]$

$\mathrm{C} = \overline{s'_{\mathrm{BC}}}\,?(x_{\mathrm{CredA}});\ (\boldsymbol{\nu}\,pc)\ (\bar{b}\,(pc\colon S).pc\,(x\colon S).x\,?(y_{\mathrm{CredA}});$
$\mathtt{if}\ x_{\mathrm{CredA}} == y_{\mathrm{CredA}}\ \mathtt{then}\ x \triangleleft \mathbf{Success};\ x\,?(x_{\mathrm{LM}})\ \mathtt{else}\ x \triangleleft \mathbf{Fail};\ \mathsf{close}(x);\ \mathsf{close}(pc))$

Typing

This peer is exactly the same as C in Case 1.

8.2.3 Case 3

This case is an attempt of a double delegation: A also wants to delegate its session with D to B. However the delegation is only performed after the first delegation is complete, which is triggered by the sending of a new credential that A creates.

A - $[[P \mid s_{\mathrm{AB}} : \langle S \rangle \vec{h} \mid s''_{\mathrm{AD}} : \langle S^4 \rangle \vec{h^4}]]$

$\mathrm{A} = \mathsf{make}\ (\mathrm{CredD});\ s_{\mathrm{AB}}\,!\langle CredD \rangle;\ s_{\mathrm{AB}}\,?(S', x_{\mathrm{IPc}}, y_{\mathrm{pc}}, z_{\mathrm{CredA}});\ \mathsf{close}(s_{\mathrm{AB}});\ \overline{y_{\mathrm{pc}}}\,(x\colon S).x\,!\langle z_{\mathrm{CredA}} \rangle;$ $x \triangleright \{\mathbf{Success} : x\,!\langle LM(S - S'), CredD \rangle, \mathbf{Fail} : \mathsf{close}(x)\}$

Since, in this case, A also tries to perform a delegation with B for the passive party D, it starts by creating a credential for the same purpose. Then, it outputs that credential to B and the rest is identical to the previous case with exception of the branching part: in case of a successful delegation it also sends the new credential to notify the receiver that another delegation is going to happen.

Typing

$\Gamma \vdash \mathbf{0} \triangleright s_{\mathrm{AB}} : \mathsf{end} \cdot s''_{\mathrm{AD}} : \mathsf{end} \cdot x : \mathsf{end}$

$\Gamma \vdash x \triangleright \{\mathbf{Success} : x\,!\langle LM(S - S'), CredD \rangle, \mathbf{Fail} : \mathsf{close}(x)\}$
$\triangleright s_{\mathrm{AB}} : \mathsf{end} \cdot s''_{\mathrm{AD}} : \mathsf{end} \cdot x : \&\{\mathbf{Success} : !\langle String, CredT \rangle, \mathbf{Fail} : \mathsf{end}\}$

$\Gamma \vdash x\,!\langle z_{\mathrm{CredA}} \rangle;\ x \triangleright \{\mathbf{Success} : x\,!\langle LM(S - S'), CredD \rangle, \mathbf{Fail} : \mathsf{close}(x)\}$
$\triangleright s_{\mathrm{AB}} : \mathsf{end} \cdot s''_{\mathrm{AD}} : \mathsf{end} \cdot x : !\langle CredT \rangle; \&\{\mathbf{Success} : !\langle String, CredT \rangle, \mathbf{Fail} : \mathsf{end}\}$

$\Gamma \vdash \overline{y_{\mathrm{pc}}}\,(x\colon S).x\,!\langle z_{\mathrm{CredA}} \rangle;\ x \triangleright \{\mathbf{Success} : x\,!\langle LM(S - S'), CredD \rangle, \mathbf{Fail} : \mathsf{close}(x)\}$
$\triangleright s_{\mathrm{AB}} : \mathsf{end} \cdot s''_{\mathrm{AD}} : \mathsf{end}$

$\Gamma \vdash s_{\mathrm{AB}}\,?(S', x_{\mathrm{IPc}}, y_{\mathrm{pc}}, z_{\mathrm{CredA}});\ \mathsf{close}(s_{\mathrm{AB}});\ \overline{y_{\mathrm{pc}}}\,(x\colon S).x\,!\langle z_{\mathrm{CredA}} \rangle;$
$x \triangleright \{\mathbf{Success} : x\,!\langle LM(S - S'), CredD \rangle, \mathbf{Fail} : \mathsf{close}(x)\}$
$\triangleright s_{\mathrm{AB}} : ?(S, nat, nat, CredT); \mathsf{end} \cdot s''_{\mathrm{AD}} : \mathsf{end}$

$\Gamma \vdash s_{\mathrm{AB}}\,!\langle CredD \rangle;\ s_{\mathrm{AB}}\,?(S', x_{\mathrm{IPc}}, y_{\mathrm{pc}}, z_{\mathrm{CredA}});\ \mathsf{close}(s_{\mathrm{AB}});\ \overline{y_{\mathrm{pc}}}\,(x\colon S).x\,!\langle z_{\mathrm{CredA}} \rangle;$
$x \triangleright \{\mathbf{Success} : x\,!\langle LM(S - S'), CredD \rangle, \mathbf{Fail} : \mathsf{close}(x)\}$
$\triangleright s_{\mathrm{AB}} : !\langle CredT \rangle; ?(S, nat, nat, CredT); \mathsf{end} \cdot s''_{\mathrm{AD}} : \mathsf{end}$

The main difference in the typification from the previous case is that it also sends the credential to B in before receiving the session information. Furthermore, it also sends the credential in the new established connection x in the Success branch.

B - $[[Q \mid \overline{s_{\mathrm{AB}}} : \langle S' \rangle \vec{h'} \mid s'_{\mathrm{BC}} : \langle S'' \rangle \vec{h''}]]$

$\mathrm{B} = \mathsf{make}\ (\mathrm{CredA});\ s'_{\mathrm{BC}}\,!\langle CredA \rangle;\ b\,(x_{\mathrm{pc}}\colon S).\overline{s_{\mathrm{AB}}}\,!\langle S', IPc, x_{\mathrm{pc}}, CredA \rangle;\ \mathsf{close}(\overline{s_{\mathrm{AB}}}) \mid \overline{s_{\mathrm{AB}}}\,?(x_{\mathrm{CredD}})$

Similar to the previous Case, with the exception that it receives the credential from A to perform the other delegation in parallel. However this delegation is ignored by B which continues its delegation process from A to C.

Typing

$\Gamma \vdash \mathbf{0} \triangleright \overline{s_{\mathrm{AB}}} : \mathsf{end} \cdot s'_{\mathrm{BC}} : \mathsf{end} \cdot x_{\mathrm{pc}} : \mathsf{end}$

$\Gamma \vdash \overline{s_{\mathrm{AB}}}\,!\langle S', IPc, x_{\mathrm{pc}}, CredA \rangle;\ \mathsf{close}(\overline{s_{\mathrm{AB}}}) \triangleright \overline{s_{\mathrm{AB}}} : !\langle S, nat, nat, CredT \rangle; \mathsf{end} \cdot s'_{\mathrm{BC}} : \mathsf{end} \cdot x_{\mathrm{pc}} : \mathsf{end}$

$\Gamma \vdash b\,(x_{\text{pc}}\colon S).\overline{s_{\text{AB}}}\,!\langle S', IPc, x_{\text{pc}}, CredA\rangle;\ \mathsf{close}(\overline{s_{\text{AB}}}) \triangleright \overline{s_{\text{AB}}}:!\langle S, nat, nat, CredT\rangle; \mathtt{end}\cdot s'_{\text{BC}}\colon \mathtt{end}$

$\Gamma \vdash s'_{\text{BC}}\,!\langle CredA\rangle;\ b\,(x_{\text{pc}}\colon S).\overline{s_{\text{AB}}}\,!\langle S', IPc, x_{\text{pc}}, CredA\rangle;\ \mathsf{close}(\overline{s_{\text{AB}}})$
$\quad \triangleright \overline{s_{\text{AB}}}:!\langle S, nat, nat, CredT\rangle; \mathtt{end}\cdot s'_{\text{BC}}:!\langle CredT\rangle; \mathtt{end} = \Sigma_1$

$\Gamma \vdash \overline{s_{\text{AB}}}\,?(x_{\text{CredD}}) \triangleright \overline{s_{\text{AB}}}:?(CredT); \mathtt{end} = \Sigma_2$

$\Gamma \vdash \mathsf{make}\ (CredA);\ s'_{\text{BC}}\,!\langle CredA\rangle;\ b\,(x_{\text{pc}}\colon S).\overline{s_{\text{AB}}}\,!\langle S', IPc, x_{\text{pc}}, CredA\rangle;\ \mathsf{close}(\overline{s_{\text{AB}}}) \mid \overline{s_{\text{AB}}}\,?(x_{\text{CredD}})$
$\quad \triangleright \Sigma_1\cdot\Sigma_2$

Process B is typed as the composition of a reception of a Credential type together with the same result of process B in Case 2.

C - $[[R \mid \overline{s'_{\text{BC}}} : \langle S^3\rangle \vec{h^3}]]$

$\text{C} = \overline{s'_{\text{BC}}}\,?(x_{\text{CredA}});\ (\boldsymbol{\nu}\,pc)\ (\bar{b}\,(pc\colon S).pc\,(x\colon S).x\,?(y_{\text{CredA}});$
$\mathtt{if}\ x_{\text{CredA}} == y_{\text{CredA}}\ \mathtt{then}\ x \triangleleft \textbf{Success};\ x\,?(x_{\text{LM}}, y_{\text{CredD}})\ \mathtt{else}\ x \triangleleft \textbf{Fail};\ \mathsf{close}(x);\ \mathsf{close}(pc))$

Very similar to the previous cases with the exception of the reception of a new credential along with the lost messages. This is what will trigger the new delegation between C and D.

Typing

$\Gamma \vdash \mathbf{0} \triangleright \overline{s'_{\text{BC}}}\colon \mathtt{end}\cdot pc\colon \mathtt{end}\cdot x\colon \mathtt{end}$

$\Gamma \vdash x\,?(x_{\text{LM}}, y_{\text{CredD}}) \triangleright \overline{s'_{\text{BC}}}\colon \mathtt{end}\cdot pc\colon \mathtt{end}\cdot x:?(String, CredT); \mathtt{end}$

$\Gamma \vdash x \triangleleft \textbf{Success};\ x\,?(x_{\text{LM}}, y_{\text{CredD}})$
$\quad \triangleright \overline{s'_{\text{BC}}}\colon \mathtt{end}\cdot pc\colon \mathtt{end}\cdot x\colon \oplus\{\textbf{Success} :?(String, CredT); \mathtt{end}, \textbf{Fail} : \mathtt{end}\}$

Let $Q = \mathtt{if}\ x_{\text{CredA}} == y_{\text{CredA}}\ \mathtt{then}\ x \triangleleft \textbf{Success};\ x\,?(x_{\text{LM}}, y_{\text{CredD}})\ \mathtt{else}\ x \triangleleft \textbf{Fail};\ \mathsf{close}(x);\ \mathsf{close}(pc)$

$\Gamma \vdash Q \triangleright \overline{s'_{\text{BC}}}\colon \mathtt{end}\cdot pc\colon \mathtt{end}\cdot x\colon \oplus\{\textbf{Success} :?(String, CredT); \mathtt{end}, \textbf{Fail} : \mathtt{end}\}$

$\Gamma \vdash x\,?(y_{\text{CredA}});\ Q \triangleright \overline{s'_{\text{BC}}}\colon \mathtt{end}\cdot pc\colon \mathtt{end}\cdot x:?(CredT); \oplus\{\textbf{Success} :?(String, CredT); \mathtt{end}, \textbf{Fail} : \mathtt{end}\}$

$\Gamma \vdash pc\,(x\colon S).x\,?(y_{\text{CredA}});\ Q \triangleright \overline{s'_{\text{BC}}}\colon \mathtt{end}\cdot pc\colon \mathtt{end}$

$\Gamma \vdash \bar{b}\,(pc\colon S).pc\,(x\colon S).x\,?(y_{\text{CredA}});\ Q \triangleright \overline{s'_{\text{BC}}}\colon \mathtt{end}$

$\Gamma \vdash \overline{s'_{\text{BC}}}\,?(x_{\text{CredA}});\ (\boldsymbol{\nu}\,pc)\ (\bar{b}\,(pc\colon S).pc\,(x\colon S).x\,?(y_{\text{CredA}});\ Q) \triangleright \overline{s'_{\text{BC}}}:?(CredT); \mathtt{end}$

As mentioned above, the single difference is the reception of a Credential type along with the String in the Success branch of the new session established between A and C.

D - $[[T \mid \overline{s''_{\text{AD}}} : \langle S^5\rangle \vec{h^5}]]$

$\text{D} = \mathbf{0}$

In this case, D does not do anything since the delegation is not simultaneous: after the first one terminates the second is triggered and operates as a Case 1 or Case 2 (if the passive party has finished its part of the session).

8.2.4 Case 4

We finally reach the most interesting and most difficult case: the double delegation. In this case B delegates the session with A to C and at the same time, A is also delegating its session with B to D.

A - $[[P \mid s_{\text{AB}} : \langle S\rangle\vec{h} \mid s''_{\text{AD}} : \langle S^4\rangle\vec{h^4}]]$

$$\begin{aligned}A = {} & \mathsf{make}\ (\text{CredB});\ s''_{\text{AD}}\,!\langle CredB\rangle;\ a\,(x_{\text{pd}}\colon S).s_{\text{AB}}\,!\langle S, IPd, x_{\text{pd}}, CredB\rangle \mid \\ & s_{\text{AB}}\,?(S', x_{\text{IPc}}, y_{\text{pc}}, z_{\text{CredA}});\ \mathsf{close}(s_{\text{AB}});\ \overline{y_{\text{pc}}}\,(x\colon S).x\,!\langle z_{\text{CredA}}\rangle; \\ & x \triangleright \{\textbf{Success} : x\,!\langle LM(S-S'), S', x_{\text{IPd}}, y_{\text{pd}}, CredB\rangle;\ \mathsf{close}(x);\ s''_{\text{AD}} \triangleleft \textbf{Success}, \\ & \quad \textbf{Fail} : \mathsf{close}(x);\ s''_{\text{AD}} \triangleleft \textbf{Fail};\ s''_{\text{AD}}\,!\langle \mathsf{close}(x_{\text{pd}})\rangle\}\end{aligned}$$

In this specific case, A starts by creating a new credential for one of the delegations, followed by the sending of that credential to D. Then, it receives the opened port *pd* that D is using for accepting a new connection and sends all the information regarding the session to B. Simultaneously, A is waiting in parallel for the receiving information to connect to C. After its reception, it closes its session with B, connects to C and sends the credential it had received from B. Again, if those credentials match, A sends the lost messages, together with the information for C to connect to D and closes the new established session with C, ending up by notifying D that everything went successfully. On the other hand, if the connection with C is not correctly established, it terminates that connection and asks D to close the new port *pd* and not to expect a new connection from C.

Typing

$\Gamma \vdash \mathbf{0} \triangleright s_{\text{AB}} : \mathsf{end}\cdot s''_{\text{AD}} : \mathsf{end}\cdot x : \mathsf{end}$

$\Gamma \vdash s''_{\text{AD}} \triangleleft \textbf{Fail};\ s''_{\text{AD}}\,!\langle \mathsf{close}(x_{\text{pd}})\rangle \triangleright s_{\text{AB}} : \mathsf{end}\cdot s''_{\text{AD}} : \oplus\{\textbf{Success} : \mathsf{end}, \textbf{Fail} :!\langle CloseSig\rangle\}\cdot x : \mathsf{end}$

$\Gamma \vdash x\,!\langle LM(S-S'), S', x_{\text{IPd}}, y_{\text{pd}}, CredB\rangle \triangleright s_{\text{AB}} : \mathsf{end}\cdot s''_{\text{AD}} : \mathsf{end}\cdot x :!\langle String, S, nat, nat, CredT\rangle$

Let $Q = x \triangleright \{\textbf{Success} : x\,!\langle LM(S-S'), S', x_{\text{IPd}}, y_{\text{pd}}, CredB\rangle;\ \mathsf{close}(x);\ s''_{\text{AD}} \triangleleft \textbf{Success},$
$\quad \textbf{Fail} : \mathsf{close}(x);\ s''_{\text{AD}} \triangleleft \textbf{Fail};\ s''_{\text{AD}}\,!\langle \mathsf{close}(x_{\text{pd}})\rangle\}$

$\Gamma \vdash Q \triangleright s_{\text{AB}} : \mathsf{end}\cdot s''_{\text{AD}} : \oplus\{\textbf{Success} : \mathsf{end}, \textbf{Fail} :!\langle CloseSig\rangle\}\cdot$
$\quad x : \&\{\textbf{Success} :!\langle String, S, nat, nat, CredT\rangle, \textbf{Fail} : \mathsf{end}\}$

$\Gamma \vdash x\,!\langle z_{\text{CredA}}\rangle;\ Q$
$\quad \triangleright s_{\text{AB}} : \mathsf{end}\cdot s''_{\text{AD}} : \oplus\{\textbf{Success} : \mathsf{end}, \textbf{Fail} :!\langle CloseSig\rangle\}\cdot$
$\quad x :!\langle CredT\rangle; \&\{\textbf{Success} :!\langle String, S, nat, nat, CredT\rangle, \textbf{Fail} : \mathsf{end}\}$

$\Gamma \vdash \overline{y_{\text{pc}}}\,(x\colon S).x\,!\langle z_{\text{CredA}}\rangle;\ Q$
$\quad \triangleright s_{\text{AB}} : \mathsf{end}\cdot s''_{\text{AD}} : \oplus\{\textbf{Success} : \mathsf{end}, \textbf{Fail} :!\langle CloseSig\rangle\}$

$\Gamma \vdash s_{\text{AB}}\,?(S', x_{\text{IPc}}, y_{\text{pc}}, z_{\text{CredA}});\ \mathsf{close}(s_{\text{AB}});\ \overline{y_{\text{pc}}}\,(x\colon S).x\,!\langle z_{\text{CredA}}\rangle;\ Q$
$\quad \triangleright s_{\text{AB}} :?(S, nat, nat, CredT); \mathsf{end}\cdot s''_{\text{AD}} : \oplus\{\textbf{Success} : \mathsf{end}, \textbf{Fail} :!\langle CloseSig\rangle\} = \Sigma_1$

$\Gamma \vdash \mathbf{0} \triangleright s_{\text{AB}} : \mathsf{end}\cdot s''_{\text{AD}} : \mathsf{end}\cdot x_{\text{pd}} : \mathsf{end}$

$\Gamma \vdash s_{\text{AB}}\,!\langle S, IPd, x_{\text{pd}}, CredB\rangle \triangleright s_{\text{AB}} :!\langle S, nat, nat, CredT\rangle; \mathsf{end}\cdot s''_{\text{AD}} : \mathsf{end}\cdot x_{\text{pd}} : \mathsf{end}$

$\Gamma \vdash a\,(x_{\text{pd}}\colon S).s_{\text{AB}}\,!\langle S, IPd, x_{\text{pd}}, CredB\rangle \triangleright s_{\text{AB}} :!\langle S, nat, nat, CredT\rangle; \mathsf{end}\cdot s''_{\text{AD}} : \mathsf{end}$

$\Gamma \vdash s''_{\text{AD}}\,!\langle CredB\rangle;\ a\,(x_{\text{pd}}\colon S).s_{\text{AB}}\,!\langle S, IPd, x_{\text{pd}}, CredB\rangle$
$\quad \triangleright s_{\text{AB}} :!\langle S, nat, nat, CredT\rangle; \mathsf{end}\cdot s''_{\text{AD}} :!\langle CredT\rangle; \mathsf{end} = \Sigma_2$

$\Gamma \vdash \mathsf{make}\ (CredB);\ s''_{\text{AD}}\,!\langle CredB\rangle;\ a\,(x_{\text{pd}}\colon S).s_{\text{AB}}\,!\langle S, IPd, x_{\text{pd}}, CredB\rangle \mid$
$\quad s_{\text{AB}}\,?(S', x_{\text{IPc}}, y_{\text{pc}}, z_{\text{CredA}});\ \mathsf{close}(s_{\text{AB}});\ \overline{y_{\text{pc}}}\,(x\colon S).x\,!\langle z_{\text{CredA}}\rangle;\ Q \triangleright \Sigma_1\cdot\Sigma_2$

The typified process A is what is expected from observing how the process works. In the session shared with D, it sends a credential type to let it know which peer to expect a connection with. Then, it is typified as a Selection type with nothing in the Success branch and a Closing signal in

the Fail branch. The session x between A and C is typed as the sending of a credential type and the branching with all the important session information in the Success branch and nothing in the Fail one. Finally, the session shared with B starts by sending all the session information regarding the delegation with D along with the credential type, followed by the reception of all the session information of the delegation with C.

B - $[[Q \mid \overline{s_{\mathrm{AB}}} : \langle S' \rangle \vec{h'} \mid s'_{\mathrm{BC}} : \langle S'' \rangle \vec{h''}]]$

$\mathrm{B} = \mathsf{make}\ (\mathrm{CredA})\ s'_{\mathrm{BC}}\,!\langle CredA \rangle;\ b\,(x_{\mathrm{pc}} \colon S).\overline{s_{\mathrm{AB}}}\,!\langle S', IPc, x_{\mathrm{pc}}, CredA \rangle \mid$
$\overline{s_{\mathrm{AB}}}\,?(S, x_{\mathrm{IPd}}, y_{\mathrm{pd}}, z_{\mathrm{CredB}});\ \mathsf{close}(\overline{s_{\mathrm{AB}}})$

The process above starts by creating the credential like usual, send it to C, receive the port that C has opened for the new connection and send all that information back to A. Simultaneously, it receives the information that A sent in order to connect to D, which is ignored by B. Finally B closes its session with A.

Typing

$\Gamma \vdash \mathbf{0} \triangleright \overline{s_{\mathrm{AB}}} \colon \mathtt{end} \cdot s'_{\mathrm{BC}} \colon \mathtt{end} \cdot x_{\mathrm{pc}} \colon \mathtt{end}$

$\Gamma \vdash \overline{s_{\mathrm{AB}}}\,!\langle S', IPc, x_{\mathrm{pc}}, CredA \rangle \triangleright \overline{s_{\mathrm{AB}}} \colon !\langle S, nat, nat, CredT \rangle; \mathtt{end} \cdot s'_{\mathrm{BC}} \colon \mathtt{end} \cdot x_{\mathrm{pc}} \colon \mathtt{end}$

$\Gamma \vdash b\,(x_{\mathrm{pc}} \colon S).\overline{s_{\mathrm{AB}}}\,!\langle S', IPc, x_{\mathrm{pc}}, CredA \rangle;\ \mathsf{close}(\overline{s_{\mathrm{AB}}}) \triangleright \overline{s_{\mathrm{AB}}} \colon !\langle S, nat, nat, CredT \rangle; \mathtt{end} \cdot s'_{\mathrm{BC}} \colon \mathtt{end}$

$\Gamma \vdash s'_{\mathrm{BC}}\,!\langle CredA \rangle;\ b\,(x_{\mathrm{pc}} \colon S).\overline{s_{\mathrm{AB}}}\,!\langle S', IPc, x_{\mathrm{pc}}, CredA \rangle;\ \mathsf{close}(\overline{s_{\mathrm{AB}}})$
$\quad \triangleright \overline{s_{\mathrm{AB}}} \colon !\langle S, nat, nat, CredT \rangle; \mathtt{end} \cdot s'_{\mathrm{BC}} \colon !\langle CredT \rangle; \mathtt{end} = \Sigma_1$

$\Gamma \vdash \mathbf{0} \triangleright \overline{s_{\mathrm{AB}}} \colon \mathtt{end}$

$\Gamma \vdash \overline{s_{\mathrm{AB}}}\,?(S, x_{\mathrm{IPd}}, y_{\mathrm{pd}}, z_{\mathrm{CredB}});\ \mathsf{close}(\overline{s_{\mathrm{AB}}}) \triangleright \overline{s_{\mathrm{AB}}} \colon ?(S, nat, nat, CredT); \mathtt{end} = \Sigma_2$

$\Gamma \vdash \mathsf{make}\ (CredA);\ s'_{\mathrm{BC}}\,!\langle CredA \rangle;\ b\,(x_{\mathrm{pc}} \colon S).\overline{s_{\mathrm{AB}}}\,!\langle S', IPc, x_{\mathrm{pc}}, CredA \rangle \mid$
$\quad \overline{s_{\mathrm{AB}}}\,?(S, x_{\mathrm{IPd}}, y_{\mathrm{pd}}, z_{\mathrm{CredB}});\ \mathsf{close}(\overline{s_{\mathrm{AB}}}) \triangleright \Sigma_1 \cdot \Sigma_2$

With A, B only sends and receives session information for both C and D. On the other hand, B only sends to C a credential type which is aimed for the delegation between A and C.

C - $[[R \mid \overline{s'_{\mathrm{BC}}} : \langle S^3 \rangle \vec{h^3}]]$

$\mathrm{C} = \overline{s'_{\mathrm{BC}}}\,?(x_{\mathrm{CredA}});\ (\boldsymbol{\nu}\, pc)\ (\overline{b}\,(pc \colon S).pc\,(x \colon S).x\,?(y_{\mathrm{CredA}});\ \mathtt{if}\ x_{\mathrm{CredA}}! = y_{\mathrm{CredA}}\ \mathtt{then}$
$\quad x \triangleleft \mathbf{Fail};\ \mathsf{close}(x);\ \mathsf{close}(pc)\ \mathtt{else}$
$\quad x \triangleleft \mathbf{Success};\ x\,?(x_{\mathrm{LM}}, S', x_{\mathrm{IPd}}, y_{\mathrm{pd}}, z_{\mathrm{CredB}});\ \overline{y_{\mathrm{pd}}}\,(y \colon S).y\,!\langle z_{\mathrm{CredB}} \rangle;$
$\qquad y \triangleright \{\mathbf{Success} : \mathbf{0}, \mathbf{Fail} : \mathsf{close}(y)\}\ )$

This process is slightly more complicated than the previous cases. It starts by receiving the credential from B, opening a new port *pc* for the new delegation, accepting a new connection and receiving a new credential from A. Yet again, if those credentials match, the session continues; otherwise it is closed. If it succeeds, there is a particular difference from the other cases: it receives the order to connect to D and so, it connects to it. Finally it sends the credential to D and starts a branching which closes the new connection with D in case of a failure or continues in case of success.

Typing

$\Gamma \vdash \mathbf{0} \triangleright \overline{s'_{\mathrm{BC}}} \colon \mathtt{end} \cdot pc \colon \mathtt{end} \cdot x \colon \mathtt{end} \cdot y \colon \mathtt{end}$

$\Gamma \vdash y \triangleright \{\mathbf{Success} : \mathbf{0}, \mathbf{Fail} : \mathsf{close}(y)\} \triangleright \overline{s'_{\mathrm{BC}}} \colon \mathtt{end} \cdot pc \colon \mathtt{end} \cdot x \colon \mathtt{end} \cdot y \colon \&\{\mathbf{Success} : \mathtt{end}, \mathbf{Fail} : \mathtt{end}\}$

$\Gamma \vdash y\,!\langle z_{\mathrm{CredB}}\rangle;\ y \triangleright \{\mathbf{Success} : \mathbf{0}, \mathbf{Fail} : \mathsf{close}(y)\}$
$\quad \triangleright \overline{s'_{\mathrm{BC}}} \colon \mathtt{end} \cdot pc \colon \mathtt{end} \cdot x \colon \mathtt{end} \cdot y \colon !\langle CredT\rangle; \&\{\mathbf{Success} : \mathtt{end}, \mathbf{Fail} : \mathtt{end}\}$

$\Gamma \vdash \overline{y_{\mathrm{pd}}}\,(y \colon S).y\,!\langle z_{\mathrm{CredB}}\rangle;\ y \triangleright \{\mathbf{Success} : \mathbf{0}, \mathbf{Fail} : \mathsf{close}(y)\} \triangleright \overline{s'_{\mathrm{BC}}} \colon \mathtt{end} \cdot pc \colon \mathtt{end} \cdot x \colon \mathtt{end}$

$\Gamma \vdash x\,?(x_{\mathrm{LM}}, S', x_{\mathrm{IPd}}, y_{\mathrm{pd}}, z_{\mathrm{CredB}});\ \overline{y_{\mathrm{pd}}}\,(y \colon S).y\,!\langle z_{\mathrm{CredB}}\rangle;\ y \triangleright \{\mathbf{Success} : \mathbf{0}, \mathbf{Fail} : \mathsf{close}(y)\}$
$\quad \triangleright \overline{s'_{\mathrm{BC}}} \colon \mathtt{end} \cdot pc \colon \mathtt{end} \cdot x \colon ?(String, S, nat, nat, CredT); \mathtt{end}$

$\Gamma \vdash x \triangleleft \mathbf{Success};\ x\,?(x_{\mathrm{LM}}, S', x_{\mathrm{IPd}}, y_{\mathrm{pd}}, z_{\mathrm{CredB}});\ \overline{y_{\mathrm{pd}}}\,(y \colon S).y\,!\langle z_{\mathrm{CredB}}\rangle;$
$\quad y \triangleright \{\mathbf{Success} : \mathbf{0}, \mathbf{Fail} : \mathsf{close}(y)\}$
$\quad \triangleright \overline{s'_{\mathrm{BC}}} \colon \mathtt{end} \cdot pc \colon \mathtt{end} \cdot x \colon \oplus\{\mathbf{Success} : ?(String, S, nat, nat, CredT); \mathtt{end}, \mathbf{Fail} : \mathtt{end}\}$

Let $Q = \mathtt{if}\ x_{\mathrm{CredA}} != y_{\mathrm{CredA}}\ \mathtt{then}$
$\quad x \triangleleft \mathbf{Fail};\ \mathsf{close}(x);\ \mathsf{close}(pc)\ \mathtt{else}$
$\quad x \triangleleft \mathbf{Success};\ x\,?(x_{\mathrm{LM}}, S', x_{\mathrm{IPd}}, y_{\mathrm{pd}}, z_{\mathrm{CredB}});\ \overline{y_{\mathrm{pd}}}\,(y \colon S).y\,!\langle z_{\mathrm{CredB}}\rangle;$
$\quad\quad y \triangleright \{\mathbf{Success} : \mathbf{0}, \mathbf{Fail} : \mathsf{close}(y)\}$

$\Gamma \vdash Q \triangleright \overline{s'_{\mathrm{BC}}} \colon \mathtt{end} \cdot pc \colon \mathtt{end} \cdot x \colon \oplus\{\mathbf{Success} : ?(String, S, nat, nat, CredT); \mathtt{end}, \mathbf{Fail} : \mathtt{end}\}$

$\Gamma \vdash x\,?(y_{\mathrm{CredA}});\ Q$
$\quad \triangleright \overline{s'_{\mathrm{BC}}} \colon \mathtt{end} \cdot pc \colon \mathtt{end} \cdot x \colon ?(CredT); \oplus\{\mathbf{Success} : ?(String, S, nat, nat, CredT); \mathtt{end}, \mathbf{Fail} : \mathtt{end}\}$

$\Gamma \vdash pc\,(x \colon S).x\,?(y_{\mathrm{CredA}});\ Q \triangleright \overline{s'_{\mathrm{BC}}} \colon \mathtt{end} \cdot pc \colon \mathtt{end}$

$\Gamma \vdash \overline{b}\,(pc \colon S).pc\,(x \colon S).x\,?(y_{\mathrm{CredA}});\ Q \triangleright \overline{s'_{\mathrm{BC}}} \colon \mathtt{end}$

$\Gamma \vdash \overline{s'_{\mathrm{BC}}}\,?(x_{\mathrm{CredA}});\ (\boldsymbol{\nu}\, pc)\ (\overline{b}\,(pc \colon S).pc\,(x \colon S).x\,?(y_{\mathrm{CredA}});\ Q) \triangleright \overline{s'_{\mathrm{BC}}} \colon ?(CredT); \mathtt{end}$

Regarding the connection with D, it sends a credential type and goes into a branching with no operations in both cases. With A, it receives a credential type followed by the selection of one of the two branches: if it is successful it receives all the session information along with the credential to connect to D. Finally, it just receives a credential from B.

D - $[[T \mid \overline{s''_{\mathrm{AD}}} : \langle S^5 \rangle \vec{h^5}]]$

$$\begin{aligned}
\mathrm{D} = {} & \overline{s''_{\mathrm{AD}}}\,?(x_{\mathrm{CredB}});\ (\boldsymbol{\nu}\, pd)\ (\overline{a}\,(pd \colon S).pd\,(y \colon S). \\
& \overline{s''_{\mathrm{AD}}} \triangleright \{\mathbf{Success} : y\,?(y_{\mathrm{CredB}}); \\
& \quad\quad \mathtt{if}\ x_{\mathrm{CredB}} == y_{\mathrm{CredB}}\ \mathtt{then}\ y \triangleleft \mathbf{Success}\ \mathtt{else}\ y \triangleleft \mathbf{Fail};\ \mathsf{close}(y);\ \mathsf{close}(pd), \\
& \quad \mathbf{Fail} : \overline{s''_{\mathrm{AD}}}\,?(\mathsf{close}(pd));\ \mathsf{close}(pd)\})
\end{aligned}$$

To conclude, D receives the credential from A, opens a new port *pd*, sends that information to A through their private channel *a* and waits for a new connection on that port. Afterwards, it goes into a branching to know whether the delegation between A and C went successfully. If it did not, it receives a closing signal from A and closes the port *pd*; otherwise, it accepts another credential and checks whether they match. If they do, everything continues normally; if not, the new session is closed along with the open port *pd*.

Typing

$\Gamma \vdash \mathbf{0} \triangleright \overline{s''_{\mathrm{AD}}} \colon \mathtt{end} \cdot pd \colon \mathtt{end} \cdot y \colon \mathtt{end}$

$\Gamma \vdash y \triangleleft \mathbf{Fail};\ \mathsf{close}(y);\ \mathsf{close}(pd) \triangleright \overline{s''_{\mathrm{AD}}} \colon \mathtt{end} \cdot pd \colon \mathtt{end} \cdot y \colon \oplus\{\mathbf{Success} : \mathtt{end}, \mathbf{Fail} : \mathtt{end}\}$

$\Gamma \vdash \mathtt{if}\ x_{\mathrm{CredB}} == y_{\mathrm{CredB}}\ \mathtt{then}\ y \triangleleft \mathbf{Success}\ \mathtt{else}\ y \triangleleft \mathbf{Fail};\ \mathsf{close}(y);\ \mathsf{close}(pd)$
$\quad \triangleright \overline{s''_{\mathrm{AD}}} \colon \mathtt{end} \cdot pd \colon \mathtt{end} \cdot y \colon \oplus\{\mathbf{Success} : \mathtt{end}, \mathbf{Fail} : \mathtt{end}\}$

$$\Gamma \vdash y\,?(y_{\text{CredB}});\ \texttt{if}\ x_{\text{CredB}} == y_{\text{CredB}}\ \texttt{then}\ y \triangleleft \textbf{Success}\ \texttt{else}\ y \triangleleft \textbf{Fail};\ \mathsf{close}(y);\ \mathsf{close}(pd)$$
$$\triangleright \overline{s''_{\text{AD}}} : \texttt{end} \cdot pd : \texttt{end} \cdot y : ?(CredT); \oplus\{\textbf{Success} : \texttt{end}, \textbf{Fail} : \texttt{end}\}$$

$$\text{Let } Q = \overline{s''_{\text{AD}}} \triangleright \{\textbf{Success} : y\,?(y_{\text{CredB}});\ \texttt{if}\ x_{\text{CredB}} == y_{\text{CredB}}\ \texttt{then}\ y \triangleleft \textbf{Success}\ \texttt{else}\ y \triangleleft \textbf{Fail};\ \mathsf{close}(y);\ \mathsf{close}(pd),\ \textbf{Fail} : \overline{s''_{\text{AD}}}\,?(\mathsf{close}(pd));\ \mathsf{close}(pd)\}$$

$$\Gamma \vdash Q \triangleright \overline{s''_{\text{AD}}} : \&\{\textbf{Success} : \texttt{end}, \textbf{Fail} : ?(CloseSig); \texttt{end}\} \cdot pd : \texttt{end} \cdot y : ?(CredT); \oplus\{\textbf{Success} : \texttt{end}, \textbf{Fail} : \texttt{end}\}$$

$$\Gamma \vdash pd\,(y : S).Q \triangleright \overline{s''_{\text{AD}}} : \&\{\textbf{Success} : \texttt{end}, \textbf{Fail} : ?(CloseSig); \texttt{end}\} \cdot pd : \texttt{end}$$

$$\Gamma \vdash \overline{a}\,(pd : S).pd\,(y : S).Q \triangleright \overline{s''_{\text{AD}}} : \&\{\textbf{Success} : \texttt{end}, \textbf{Fail} : ?(CloseSig); \texttt{end}\}$$

$$\Gamma \vdash \overline{s''_{\text{AD}}}\,?(x_{\text{CredB}});\ (\boldsymbol{\nu}\, pd)\ (\overline{a}\,(pd : S).pd\,(y : S).Q)$$
$$\triangleright \overline{s''_{\text{AD}}} : ?(CredT); \&\{\textbf{Success} : \texttt{end}, \textbf{Fail} : ?(CloseSig); \texttt{end}\}$$

This process is typed with the session shared with A as a receiver of a credential and the branching with the Closing Signal in the failure branch. However, with C it is typed as the reception of another credential and finalised by the selection type with no actions in either branches.

8.3 Forwarding Protocol

In the Forwarding Protocol, it is up to the session sender to forward the lost messages to the receiver before reconnection. This is quite beneficial especially in Case 2, where A does not need to connect to C just to resend the lost messages. Moreover, in case 4 there is no need for an intermediate connection and C connects to D directly – among other things explained in more detail later on.

8.3.1 Forwarding and Buffering functions

These functions are used all over the Forwarding protocol and can be modelled as:

$$fw\langle s,\overline{s}\rangle = \texttt{def}\ X(y,y') = y\,?(x);\ \texttt{if}\ x == ACK\ \texttt{then}\ y \triangleleft \textbf{Success}; y' \triangleleft \textbf{Success}; y'\,!\langle ACK\rangle\ \texttt{else}\ y \triangleleft \textbf{Fail}; y' \triangleleft \textbf{Fail}; y'\,!\langle x\rangle;\ X\langle y,y'\rangle\ \texttt{in}\ X\langle s,\overline{s}\rangle$$

$$buffer\langle\overline{s}\rangle = \texttt{def}\ Y(y) = y\,?(x);\ \texttt{if}\ x == ACK\ \texttt{then}\ y \triangleleft \textbf{Success};\ \texttt{else}\ y \triangleleft \textbf{Fail}; Y\langle y\rangle\ \texttt{in}\ Y\langle\overline{s}\rangle$$

The forwarding function is defined as the reception of some information through the session of the first argument. Afterwards, if that information is an ACK, it selects the successful branch on both sessions and outputs an ACK on the second session; otherwise, it selects the failing branches and outputs the received information on the second session, terminating by entering in recursion.

On the other hand, the buffer function starts by receiving on the session passed as argument and terminates if it is an ACK; otherwise it starts a recursion on the same session.

Forwarding and Buffering Typing

$$Y : S; \Gamma \vdash X\langle s,\overline{s}\rangle \triangleright s : S \cdot \overline{s} : S$$

Let $Q = \texttt{if}\ x == ACK\ \texttt{then}\ y \triangleleft \textbf{Success}; y' \triangleleft \textbf{Success}; y'\,!\langle ACK\rangle$
$\texttt{else}\ y \triangleleft \textbf{Fail}; y' \triangleleft \textbf{Fail}; y'\,!\langle x\rangle;\ X(y,y')$

$$Y : S; \Gamma \vdash Q \triangleright y : \oplus\{\textbf{Success} : \texttt{end}, \textbf{Fail} : S\} \cdot y' : \oplus\{\textbf{Success} :!\langle ACK\rangle, \textbf{Fail} :!\langle String\rangle; S\}$$

$$Y : S; \Gamma \vdash y\,?(x);\ Q \triangleright y :?(String); \oplus\{\textbf{Success} : \texttt{end}, \textbf{Fail} : S\} \cdot y' : \oplus\{\textbf{Success} :!\langle ACK\rangle, \textbf{Fail} :!\langle String\rangle; S\}$$

$$Y : S; \Gamma \vdash \texttt{def}\ X\langle y,y'\rangle = y\,?(x);\ Q \texttt{in}\ X\langle s,\overline{s}\rangle \triangleright s : \mu\,\mathbf{t}.?(String); \oplus\{\textbf{Success} : \texttt{end}, \textbf{Fail} : \mathbf{t}\} \cdot \overline{s} : \mu\,\mathbf{t}.\oplus\{\textbf{Success} :!\langle ACK\rangle, \textbf{Fail} :!\langle String\rangle; \mathbf{t}\}$$

$$Y : S; \Gamma \vdash Y\langle\overline{s}\rangle \triangleright \overline{s} : S$$

$$Y : S; \Gamma \vdash \texttt{if}\ x == ACK\ \texttt{then}\ y \triangleleft \textbf{Success};\ \texttt{else}\ y \triangleleft \textbf{Fail}; Y(y) \triangleright y : \oplus\{\textbf{Success} : \texttt{end}, \textbf{Fail} : S\}$$

$$Y : S; \Gamma \vdash y\,?(x);\ \texttt{if}\ x == ACK\ \texttt{then}\ y \triangleleft \textbf{Success};\ \texttt{else}\ y \triangleleft \textbf{Fail}; Y(y) \triangleright y :?(String); \oplus\{\textbf{Success} : \texttt{end}, \textbf{Fail} : S\}$$

$$Y : S; \Gamma \vdash \texttt{def}\ Y\langle y\rangle = y\,?(x);\ \texttt{if}\ x == ACK\ \texttt{then}\ y \triangleleft \textbf{Success};\ \texttt{else}\ y \triangleleft \textbf{Fail}; Y(y) \texttt{in}\ Y\langle\overline{s}\rangle \triangleright \overline{s} : \mu\,\mathbf{t}.?(String); \oplus\{\textbf{Success} : \texttt{end}, \textbf{Fail} : \mathbf{t}\}$$

The forwarding function is typed in the first argument as the recursion type of a reception of a String, followed by the selection of either terminating or continuing with the recursion. In addition, the second argument is typed as the recursion of the selection of either sending an ACK and terminating, or sending a String and continuing with the recursion.

Furthermore, the buffer function is typed as the recursion of receiving a string and selecting to either continue with, or terminate the recursion.

8.3.2 Case 1

A - $[[P \mid s_{\mathrm{AB}} : \langle S\rangle\vec{h}]]$

$\mathrm{A} = s_{\mathrm{AB}}\,?(x_{\mathrm{IPc}}, y_{\mathrm{pc}}, z_{\mathrm{CredA}});\ s_{\mathrm{AB}}\,!\langle ACK\rangle;\ \mathsf{close}(s_{\mathrm{AB}});\ \overline{y_{\mathrm{pc}}}\,(x\colon S).x\,!\langle z_{\mathrm{CredA}}\rangle;$
$x \triangleright \{\mathbf{Success} : \mathbf{0}, \mathbf{Fail} : \mathsf{close}(x)\}$

In this case, party A is very similar to the same case in the Resending protocol. The main difference is that A does not send information regarding the lost messages to C, which is done by B instead. Another major difference between this protocol and the resending one (in all the cases) is that S is not sent along with the IP address, port and credential: this is not needed since we do not need to calculate the session difference in order to get the lost messages – that is done internally between the sender and the receiver.

Typing

$\Gamma \vdash \mathbf{0} \triangleright s_{\mathrm{AB}} \colon \mathsf{end} \cdot x \colon \mathsf{end}$

$\Gamma \vdash x \triangleright \{\mathbf{Success} : \mathbf{0}, \mathbf{Fail} : \mathsf{close}(x)\} \triangleright s_{\mathrm{AB}} \colon \mathsf{end} \cdot x \colon \&\{\mathbf{Success} : \mathsf{end}, \mathbf{Fail} : \mathsf{end}\}$

$\Gamma \vdash x\,!\langle z_{\mathrm{CredA}}\rangle;\ x \triangleright \{\mathbf{Success} : \mathbf{0}, \mathbf{Fail} : \mathsf{close}(x)\}$
$\triangleright s_{\mathrm{AB}} \colon \mathsf{end} \cdot x \colon !\langle CredT\rangle; \&\{\mathbf{Success} : \mathsf{end}, \mathbf{Fail} : \mathsf{end}\}$

$\Gamma \vdash \overline{y_{\mathrm{pc}}}\,(x\colon S).x\,!\langle z_{\mathrm{CredA}}\rangle;\ x \triangleright \{\mathbf{Success} : \mathbf{0}, \mathbf{Fail} : \mathsf{close}(x)\} \triangleright s_{\mathrm{AB}} \colon \mathsf{end}$

$\Gamma \vdash s_{\mathrm{AB}}\,!\langle ACK\rangle;\ \mathsf{close}(s_{\mathrm{AB}});\ \overline{y_{\mathrm{pc}}}\,(x\colon S).x\,!\langle z_{\mathrm{CredA}}\rangle;$
$x \triangleright \{\mathbf{Success} : \mathbf{0}, \mathbf{Fail} : \mathsf{close}(x)\} \triangleright s_{\mathrm{AB}} \colon !\langle ACK\rangle; \mathsf{end}$

$\Gamma \vdash s_{\mathrm{AB}}\,?(x_{\mathrm{IPc}}, y_{\mathrm{pc}}, z_{\mathrm{CredA}});\ s_{\mathrm{AB}}\,!\langle ACK\rangle;\ \mathsf{close}(s_{\mathrm{AB}});\ \overline{y_{\mathrm{pc}}}\,(x\colon S).x\,!\langle z_{\mathrm{CredA}}\rangle;$
$x \triangleright \{\mathbf{Success} : \mathbf{0}, \mathbf{Fail} : \mathsf{close}(x)\} \triangleright s_{\mathrm{AB}} \colon ?(nat, nat, CredT); !\langle ACK\rangle; \mathsf{end}$

Once more, the difference in the typification is the removal of the sending of the String in the branching in comparison with case 1 of the Resending Protocol.

B - $[[Q \mid \overline{s_{\mathrm{AB}}} : \langle S'\rangle\vec{h'} \mid s'_{\mathrm{BC}} : \langle S''\rangle\vec{h''}]]$

$\mathrm{B} = \mathsf{make}\ (\mathrm{CredA});\ s'_{\mathrm{BC}}\,!\langle CredA\rangle;\ b\,(x_{\mathrm{pc}}\colon S).\overline{s_{\mathrm{AB}}}\,!\langle IPc, x_{\mathrm{pc}}, CredA\rangle;\ \overline{s_{\mathrm{AB}}}\,?(x_{\mathrm{ACK}});$
$\mathsf{fw}(\overline{s_{\mathrm{AB}}}, s'_{\mathrm{BC}});\ \mathsf{close}(\overline{s_{\mathrm{AB}}})$

This process starts by creating a credential, sending it, afterwards, to C. Then, it receives the information of the open port from C and sends all relative information to A. When everything is complete, it receives an acknowledgment from A and enters in the forwarding mode to send all lost messages to C. This process is complete after the forwarding by closing its session with A.

Typing

$\Gamma \vdash \mathbf{0} \triangleright \overline{s_{\mathrm{AB}}} \colon \mathsf{end} \cdot s'_{\mathrm{BC}} \colon \mathsf{end} \cdot x_{\mathrm{pc}} \colon \mathsf{end}$

$\Gamma \vdash \mathsf{fw}(\overline{s_{\mathrm{AB}}}, s'_{\mathrm{BC}});\ \mathsf{close}(\overline{s_{\mathrm{AB}}}) \triangleright \overline{s_{\mathrm{AB}}} \colon \mu\,\mathbf{t}.(?(String); \oplus\{\mathbf{Success} : \mathsf{end}, \mathbf{Fail} : \mathbf{t}\}) \cdot$
$s'_{\mathrm{BC}} \colon \mu\,\mathbf{t}.(\oplus\{\mathbf{Success} : !\langle ACK\rangle, \mathbf{Fail} : !\langle String\rangle; \mathbf{t}\}) \cdot x_{\mathrm{pc}} \colon \mathsf{end}$

$\Gamma \vdash \overline{s_{\mathrm{AB}}}\,?(x_{\mathrm{ACK}});\ \mathsf{fw}(\overline{s_{\mathrm{AB}}}, s'_{\mathrm{BC}});\ \mathsf{close}(\overline{s_{\mathrm{AB}}})$
$\triangleright \overline{s_{\mathrm{AB}}} \colon ?(ACK); \mu\,\mathbf{t}.(?(String); \oplus\{\mathbf{Success} : \mathsf{end}, \mathbf{Fail} : \mathbf{t}\}) \cdot$
$s'_{\mathrm{BC}} \colon \mu\,\mathbf{t}.(\oplus\{\mathbf{Success} : !\langle ACK\rangle, \mathbf{Fail} : !\langle String\rangle; \mathbf{t}\}) \cdot x_{\mathrm{pc}} \colon \mathsf{end}$

$\Gamma \vdash \overline{s_{\mathrm{AB}}}\,!\langle IPc, x_{\mathrm{pc}}, CredA\rangle;\ \overline{s_{\mathrm{AB}}}\,?(x_{\mathrm{ACK}});\ \mathsf{fw}(\overline{s_{\mathrm{AB}}}, s'_{\mathrm{BC}});\ \mathsf{close}(\overline{s_{\mathrm{AB}}})$
$\triangleright \overline{s_{\mathrm{AB}}} \colon !\langle nat, nat, CredT\rangle; ?(ACK); \mu\,\mathbf{t}.(?(String); \oplus\{\mathbf{Success} : \mathsf{end}, \mathbf{Fail} : \mathbf{t}\}) \cdot$

$s'_{\mathrm{BC}}\colon \mu\,\mathbf{t}.(\oplus\{\mathbf{Success}\ {:}!\langle ACK\rangle, \mathbf{Fail}\ {:}!\langle String\rangle; \mathbf{t}\})\cdot x_{\mathrm{pc}}\colon \mathtt{end}$

$\Gamma \vdash b\,(x_{\mathrm{pc}}\colon S).\overline{s_{\mathrm{AB}}}\,!\langle IPc, x_{\mathrm{pc}}, CredA\rangle;\ \overline{s_{\mathrm{AB}}}\,?(x_{\mathrm{ACK}});\ \mathsf{fw}(\overline{s_{\mathrm{AB}}}, s'_{\mathrm{BC}});\ \mathsf{close}(\overline{s_{\mathrm{AB}}})$
$\quad \triangleright\, \overline{s_{\mathrm{AB}}}\,{:}!\langle nat, nat, CredT\rangle; ?(ACK); \mu\,\mathbf{t}.(?(String); \oplus\{\mathbf{Success} : \mathtt{end}, \mathbf{Fail} : \mathbf{t}\})\cdot$
$\quad s'_{\mathrm{BC}}\colon \mu\,\mathbf{t}.(\oplus\{\mathbf{Success}\ {:}!\langle ACK\rangle, \mathbf{Fail}\ {:}!\langle String\rangle; \mathbf{t}\})$

$\Gamma \vdash s'_{\mathrm{BC}}\,!\langle CredA\rangle;\ b\,(x_{\mathrm{pc}}\colon S).\overline{s_{\mathrm{AB}}}\,!\langle IPc, x_{\mathrm{pc}}, CredA\rangle;\ \overline{s_{\mathrm{AB}}}\,?(x_{\mathrm{ACK}});\ \mathsf{fw}(\overline{s_{\mathrm{AB}}}, s'_{\mathrm{BC}});\ \mathsf{close}(\overline{s_{\mathrm{AB}}})$
$\quad \triangleright\, \overline{s_{\mathrm{AB}}}\,{:}!\langle nat, nat, CredT\rangle; ?(ACK); \mu\,\mathbf{t}.(?(String); \oplus\{\mathbf{Success} : \mathtt{end}, \mathbf{Fail} : \mathbf{t}\})\cdot$
$\quad s'_{\mathrm{BC}}\,{:}!\langle CredT\rangle; \mu\,\mathbf{t}.(\oplus\{\mathbf{Success}\ {:}!\langle ACK\rangle, \mathbf{Fail}\ {:}!\langle String\rangle; \mathbf{t}\})$

From this typification we can see that B sends to A all the important session information, followed by the reception of the ACK and terminating by the forwarding mode. This mode is typed in 8.3.1 and is basically the recursive type of receiving a string and selecting one of the branches (with the recursion on the Fail branch).
On the other hand, B sends a credential type to C and enters in the receiving part of the Forwarding. This is typed as a recursion of a selection type where the Success branch sends an ACK and the Fail branch sends a String and enters in the recursion again.

C - $[[R \mid \overline{s'_{\mathrm{BC}}} : \langle S^3\rangle \vec{h^3}]]$

$\mathrm{C} = \overline{s'_{\mathrm{BC}}}\,?(x_{\mathrm{CredA}});\ (\boldsymbol{\nu}\, pc)\ (\bar{b}\,(pc\colon S).\mathsf{buffer}(\overline{s'_{\mathrm{BC}}});\ pc\,(x\colon S).x\,?(y_{\mathrm{CredA}});$
$\mathtt{if}\ x_{\mathrm{CredA}} == y_{\mathrm{CredA}}\ \mathtt{then}\ x \triangleleft \mathbf{Success}\ \mathtt{else}\ x \triangleleft \mathbf{Fail};\ \mathsf{close}(x);\ \mathsf{close}(pc))$

Finally, C receives the credential from B, opens a new port *pc*, buffers the lost messages and only then waits for a connection from A. The order is different from the Resending protocol, which accepts the new connection and only then receives the lost messages. To conclude this process, C receives another credential from A and checks if they match: if they do, the process terminates and the delegation is complete; otherwise it closes both the new connection and the port *pc*.

Typing

$\Gamma \vdash \mathbf{0} \triangleright \overline{s'_{\mathrm{BC}}}\colon \mathtt{end}\cdot pc\colon \mathtt{end}\cdot x\colon \mathtt{end}$

$\Gamma \vdash x \triangleleft \mathbf{Fail};\ \mathsf{close}(x);\ \mathsf{close}(pc) \triangleright \overline{s'_{\mathrm{BC}}}\colon \mathtt{end}\cdot pc\colon \mathtt{end}\cdot x\colon \oplus\{\mathbf{Success} : \mathtt{end}, \mathbf{Fail} : \mathtt{end}\}$

Let $Q = \mathtt{if}\ x_{\mathrm{CredA}} == y_{\mathrm{CredA}}\ \mathtt{then}\ x \triangleleft \mathbf{Success}\ \mathtt{else}\ x \triangleleft \mathbf{Fail};\ \mathsf{close}(x);\ \mathsf{close}(pc)$

$\Gamma \vdash Q \triangleright \overline{s'_{\mathrm{BC}}}\colon \mathtt{end}\cdot pc\colon \mathtt{end}\cdot x\colon \oplus\{\mathbf{Success} : \mathtt{end}, \mathbf{Fail} : \mathtt{end}\}$

$\Gamma \vdash x\,?(y_{\mathrm{CredA}});\ Q \triangleright \overline{s'_{\mathrm{BC}}}\colon \mathtt{end}\cdot pc\colon \mathtt{end}\cdot x\,{:}?(CredT); \oplus\{\mathbf{Success} : \mathtt{end}, \mathbf{Fail} : \mathtt{end}\}$

$\Gamma \vdash pc\,(x\colon S).x\,?(y_{\mathrm{CredA}});\ Q \triangleright \overline{s'_{\mathrm{BC}}}\colon \mathtt{end}\cdot pc\colon \mathtt{end}$

$\Gamma \vdash \mathsf{buffer}(\overline{s'_{\mathrm{BC}}});\ pc\,(x\colon S).x\,?(y_{\mathrm{CredA}});\ Q \triangleright \overline{s'_{\mathrm{BC}}}\colon \mu\,\mathbf{t}.(?(String); \oplus\{\mathbf{Success} : \mathtt{end}, \mathbf{Fail} : \mathbf{t}\})\cdot pc\colon \mathtt{end}$

$\Gamma \vdash \bar{b}\,(pc\colon S).\mathsf{buffer}(\overline{s'_{\mathrm{BC}}});\ pc\,(x\colon S).x\,?(y_{\mathrm{CredA}});\ Q$
$\quad \triangleright\, \overline{s'_{\mathrm{BC}}}\colon \mu\,\mathbf{t}.(?(String); \oplus\{\mathbf{Success} : \mathtt{end}, \mathbf{Fail} : \mathbf{t}\})$

$\Gamma \vdash \overline{s'_{\mathrm{BC}}}\,?(x_{\mathrm{CredA}});\ (\boldsymbol{\nu}\, pc)\ (\bar{b}\,(pc\colon S).\mathsf{buffer}(\overline{s'_{\mathrm{BC}}});\ pc\,(x\colon S).x\,?(y_{\mathrm{CredA}});\ Q)$
$\quad \triangleright\, \overline{s'_{\mathrm{BC}}}\,{:}?(CredT); \mu\,\mathbf{t}.(?(String); \oplus\{\mathbf{Success} : \mathtt{end}, \mathbf{Fail} : \mathbf{t}\})$

The new session established with A is typed as the reception of a credential and a selection with the end type in both branches. Regarding the session with B, it is typed as the reception of a credential and as the buffering type, i.e. the recursion of receiving a String and a selection of either continuing or terminating the recursion.

8.3.3 Case 2

A - $[[P \mid s_{\text{AB}} : \langle S \rangle \vec{h}]]$

$$\text{A} = s_{\text{AB}}\,?(x_{\text{IPc}}, y_{\text{pc}}, z_{\text{CredA}});\ \mathsf{close}(s_{\text{AB}})$$

This case simply receives the information regarding the session receiver and closes the session with B, since it had already finished its part of the session, not needing to connect to C.

Typing

$$\Gamma \vdash s_{\text{AB}}\,?(x_{\text{IPc}}, y_{\text{pc}}, z_{\text{CredA}});\ \mathsf{close}(s_{\text{AB}}) \triangleright s_{\text{AB}} :?(nat, nat, CredT); \mathtt{end}$$

Regarding typification, s_{AB} receives the session information along with the credential type.

B - $[[Q \mid \overline{s_{\text{AB}}} : \langle S' \rangle \vec{h'} \mid s'_{\text{BC}} : \langle S'' \rangle \vec{h''}]]$

$$\text{B} = \mathsf{make}\ (\text{CredA});\ s'_{\text{BC}}\,!\langle CredA \rangle;\ b\,(x_{\text{pc}} \colon S).\overline{s_{\text{AB}}}\,!\langle IPc, x_{\text{pc}}, CredA \rangle;\ \mathsf{fw2}(\overline{s_{\text{AB}}}, s'_{\text{BC}});\ \mathsf{close}(\overline{s_{\text{AB}}})$$

In this case, B starts by creating the credential, sending it to C, receiving the new opened port *pc* and sending all that information back to A. Then, it uses the function fw2 instead of the fw, which is different in that it terminates the loop with the FIN signal instead of an ACK (since A has finished its part of the session). The process then terminates with the closing of the session with A.

Typing

$$\Gamma \vdash \mathbf{0} \triangleright \overline{s_{\text{AB}}} \colon \mathtt{end} \cdot s'_{\text{BC}} \colon \mathtt{end} \cdot x_{\text{pc}} \colon \mathtt{end}$$

$$\Gamma \vdash \mathsf{fw2}(\overline{s_{\text{AB}}}, s'_{\text{BC}});\ \mathsf{close}(\overline{s_{\text{AB}}}) \triangleright \overline{s_{\text{AB}}} \colon \mu\,\mathbf{t}.(?(String); \oplus\{\mathbf{Success} : \mathtt{end}, \mathbf{Fail} : \mathbf{t}\}) \cdot s'_{\text{BC}} \colon \mu\,\mathbf{t}.(\oplus\{\mathbf{Success} :!\langle ACK \rangle, \mathbf{Fail} :!\langle String \rangle; \mathbf{t}\}) \cdot x_{\text{pc}} \colon \mathtt{end}$$

$$\Gamma \vdash \overline{s_{\text{AB}}}\,!\langle IPc, x_{\text{pc}}, CredA \rangle;\ \mathsf{fw2}(\overline{s_{\text{AB}}}, s'_{\text{BC}});\ \mathsf{close}(\overline{s_{\text{AB}}}) \triangleright \overline{s_{\text{AB}}} :!\langle nat, nat, CredT \rangle; \mu\,\mathbf{t}.(?(String); \oplus\{\mathbf{Success} : \mathtt{end}, \mathbf{Fail} : \mathbf{t}\}) \cdot s'_{\text{BC}} \colon \mu\,\mathbf{t}.(\oplus\{\mathbf{Success} :!\langle ACK \rangle, \mathbf{Fail} :!\langle String \rangle; \mathbf{t}\}) \cdot x_{\text{pc}} \colon \mathtt{end}$$

$$\Gamma \vdash b\,(x_{\text{pc}} \colon S).\overline{s_{\text{AB}}}\,!\langle IPc, x_{\text{pc}}, CredA \rangle;\ \mathsf{fw2}(\overline{s_{\text{AB}}}, s'_{\text{BC}});\ \mathsf{close}(\overline{s_{\text{AB}}}) \triangleright \overline{s_{\text{AB}}} :!\langle nat, nat, CredT \rangle; \mu\,\mathbf{t}.(?(String); \oplus\{\mathbf{Success} : \mathtt{end}, \mathbf{Fail} : \mathbf{t}\}) \cdot s'_{\text{BC}} \colon \mu\,\mathbf{t}.(\oplus\{\mathbf{Success} :!\langle ACK \rangle, \mathbf{Fail} :!\langle String \rangle; \mathbf{t}\})$$

$$\Gamma \vdash s'_{\text{BC}}\,!\langle CredA \rangle;\ b\,(x_{\text{pc}} \colon S).\overline{s_{\text{AB}}}\,!\langle IPc, x_{\text{pc}}, CredA \rangle;\ \mathsf{fw2}(\overline{s_{\text{AB}}}, s'_{\text{BC}});\ \mathsf{close}(\overline{s_{\text{AB}}}) \triangleright \overline{s_{\text{AB}}} :!\langle nat, nat, CredT \rangle; \mu\,\mathbf{t}.(?(String); \oplus\{\mathbf{Success} : \mathtt{end}, \mathbf{Fail} : \mathbf{t}\}) \cdot s'_{\text{BC}} :!\langle CredT \rangle; \mu\,\mathbf{t}.(\oplus\{\mathbf{Success} :!\langle ACK \rangle, \mathbf{Fail} :!\langle String \rangle; \mathbf{t}\})$$

B sends the session information to A, composed by two natural numbers and one credential type, being then typed as the forwarding, sending part. The same way, the session with C is typed as the sending of a credential type, followed by the receiving part of the forwarding type. This forwarding type, for both the sending and receiving parts, is typed identically to the fw function typed previously.

C - $[[R \mid \overline{s'_{\text{BC}}} : \langle S^3 \rangle \vec{h^3}]]$

$$\text{C} = \overline{s'_{\text{BC}}}\,?(x_{\text{CredA}});\ (\boldsymbol{\nu}\, pc)\ (\bar{b}\,(pc\colon S).\mathsf{buffer2}(\overline{s'_{\text{BC}}});\ \mathsf{close}(pc))$$

Just like process A, this one is rather simple: it begins by receiving the credential used for the new delegation, opens the new port *pc*, sends the information about the port to B and buffers the lost messages. Since the delegation is not performed, C can close the new opened port.

Typing

$\Gamma \vdash \mathbf{0} \triangleright \overline{s'_{\text{BC}}}\colon \mathtt{end}\cdot pc\colon \mathtt{end}$

$\Gamma \vdash \mathsf{buffer2}(\overline{s'_{\text{BC}}});\ \mathsf{close}(pc) \triangleright \overline{s'_{\text{BC}}}\colon \mu\,\mathbf{t}.(?(String); \oplus\{\mathbf{Success} : \mathtt{end}, \mathbf{Fail} : \mathbf{t}\})\cdot pc\colon \mathtt{end}$

$\Gamma \vdash \bar{b}\,(pc\colon S).\mathsf{buffer2}(\overline{s'_{\text{BC}}});\ \mathsf{close}(pc) \triangleright \overline{s'_{\text{BC}}}\colon \mu\,\mathbf{t}.(?(String); \oplus\{\mathbf{Success} : \mathtt{end}, \mathbf{Fail} : \mathbf{t}\})$

$\Gamma \vdash \overline{s'_{\text{BC}}}\,?(x_{\text{CredA}});\ (\boldsymbol{\nu}\, pc)\ (\bar{b}\,(pc\colon S).\mathsf{buffer2}(\overline{s'_{\text{BC}}});\ \mathsf{close}(pc))$
$\quad \triangleright \overline{s'_{\text{BC}}}\colon ?(CredT); \mu\,\mathbf{t}.(?(String); \oplus\{\mathbf{Success} : \mathtt{end}, \mathbf{Fail} : \mathbf{t}\})$

As a consequence of the simplicity of this process, it is typed as the receiving of the credential and the buffer type adapted to terminate with the FIN signal instead of the ACK.

Auxiliary functions

$fw2\langle s, \bar{s}\rangle = \mathtt{def}\ X(y, y') = y\,?(x);\ \mathtt{if}\ x == FIN\ \mathtt{then}\ y'\,!\langle FIN\rangle;\ \mathbf{0}\ \mathtt{else}\ y'\,!\langle x\rangle;\ X\langle y, y'\rangle\ \mathtt{in}\ X\langle s, \bar{s}\rangle$

$buffer2\langle \bar{s}\rangle = \mathtt{def}\ Y(y) = y\,?(x);\ \mathtt{if}\ x == FIN\ \mathtt{then}\ \mathbf{0}\ \mathtt{else}\ X\langle y\rangle\ \mathtt{in}\ Y\langle \bar{s}\rangle$

Both these auxiliary functions are similar to the original ones except they terminate their recursion with the FIN signal and not with the ACK. Moreover, the typification is identical to the previous functions (section 8.3.1).

8.3.4 Case 3

A - $[[P \mid s_{\text{AB}} : \langle S \rangle \vec{h} \mid s''_{\text{AD}} : \langle S^4 \rangle \vec{h^4}]]$

$\text{A} = \mathsf{make}\ (\text{CredD});\ s_{\text{AB}}\,!\langle CredD\rangle;\ s_{\text{AB}}\,?(x_{\text{IPc}}, y_{\text{pc}}, z_{\text{CredA}});\ \mathsf{close}(s_{\text{AB}});\ \overline{y_{\text{pc}}}\,(x\colon S).x\,!\langle z_{\text{CredA}}\rangle;$
$x \triangleright \{\mathbf{Success} : \mathbf{0}, \mathbf{Fail} : \mathsf{close}(x)\}$

The passive party in case 3 of the forwarding protocol is pretty similar to the same case in the resending protocol, except in the successful case of the branching nothing happens, since we do not need to send the lost messages nor the credential for the delegation with D. The same applies for the typification where it does not send the String concerning the lost messages.

Typing

$\Gamma \vdash \mathbf{0} \triangleright s_{\text{AB}}\colon \mathtt{end}\cdot s''_{\text{AD}}\colon \mathtt{end}\cdot x\colon \mathtt{end}$

$\Gamma \vdash x \triangleright \{\mathbf{Success} : \mathbf{0}, \mathbf{Fail} : \mathsf{close}(x)\} \triangleright s_{\text{AB}}\colon \mathtt{end}\cdot s''_{\text{AD}}\colon \mathtt{end}\cdot x\colon \&\{\mathbf{Success} : \mathtt{end}, \mathbf{Fail} : \mathtt{end}\}$

$\Gamma \vdash x\,!\langle z_{\text{CredA}}\rangle;\ x \triangleright \{\mathbf{Success} : \mathbf{0}, \mathbf{Fail} : \mathsf{close}(x)\}$
$\quad \triangleright s_{\text{AB}}\colon \mathtt{end}\cdot s''_{\text{AD}}\colon \mathtt{end}\cdot x\colon !\langle CredT\rangle; \&\{\mathbf{Success} : \mathtt{end}, \mathbf{Fail} : \mathtt{end}\}$

$\Gamma \vdash \overline{y_{\text{pc}}}\,(x\colon S).x\,!\langle z_{\text{CredA}}\rangle;\ x \triangleright \{\mathbf{Success} : \mathbf{0}, \mathbf{Fail} : \mathsf{close}(x)\} \triangleright s_{\text{AB}}\colon \mathtt{end}\cdot s''_{\text{AD}}\colon \mathtt{end}$

$\Gamma \vdash s_{\text{AB}}\,?(x_{\text{IPc}}, y_{\text{pc}}, z_{\text{CredA}});\ \mathsf{close}(s_{\text{AB}});\ \overline{y_{\text{pc}}}\,(x\colon S).x\,!\langle z_{\text{CredA}}\rangle;$

$x \triangleright \{\textbf{Success} : \mathbf{0}, \textbf{Fail} : \mathsf{close}(x)\} \triangleright s_{\text{AB}} :?(nat, nat, CredT); \texttt{end} \cdot s''_{\text{AD}} : \texttt{end}$

$\Gamma \vdash s_{\text{AB}} \,!\langle CredD \rangle;\; s_{\text{AB}} \,?(x_{\text{IPc}}, y_{\text{pc}}, z_{\text{CredA}});\; \mathsf{close}(s_{\text{AB}});\; \overline{y_{\text{pc}}}\,(x\colon S).x\,!\langle z_{\text{CredA}} \rangle;$
$x \triangleright \{\textbf{Success} : \mathbf{0}, \textbf{Fail} : \mathsf{close}(x)\}$
$\triangleright\, s_{\text{AB}} :!\langle CredT \rangle; ?(nat, nat, CredT); \texttt{end} \cdot s''_{\text{AD}} : \texttt{end}$

B - $[[Q \mid \overline{s_{\text{AB}}} : \langle S' \rangle \vec{h'} \mid s'_{\text{BC}} : \langle S'' \rangle \vec{h''}]]$

$\text{B} = \mathsf{make}\,(\text{CredA});\; s'_{\text{BC}} \,!\langle CredA \rangle;\; b\,(x_{\text{pc}}\colon S).\overline{s_{\text{AB}}} \,!\langle IPc, x_{\text{pc}}, CredA \rangle \mid \overline{s_{\text{AB}}} \,?(x_{\text{CredD}});\; \mathsf{fw}(\overline{s_{\text{AB}}}, s'_{\text{BC}});$ $s'_{\text{BC}} \,!\langle x_{\text{CredD}} \rangle;\; \mathsf{close}(\overline{s_{\text{AB}}})$

As in all session senders, this case begins by creating a new credential for the delegation, send it to C, receive the port number that C had opened for the new connection and send all these to A. At the same time, B receives another credential indicating the desire for a new delegation, forwards all the lost messages from A to C, including the new delegation signal, and terminates the process with the closure of the session.

Typing

$\Gamma \vdash \mathbf{0} \triangleright \overline{s_{\text{AB}}} : \texttt{end} \cdot s'_{\text{BC}} : \texttt{end}$

$\Gamma \vdash s'_{\text{BC}} \,!\langle x_{\text{CredD}} \rangle;\; \mathsf{close}(\overline{s_{\text{AB}}}) \triangleright \overline{s_{\text{AB}}} : \texttt{end} \cdot s'_{\text{BC}} :!\langle CredT \rangle; \texttt{end}$

$\Gamma \vdash \mathsf{fw}(\overline{s_{\text{AB}}}, s'_{\text{BC}});\; s'_{\text{BC}} \,!\langle x_{\text{CredD}} \rangle;\; \mathsf{close}(\overline{s_{\text{AB}}}) \triangleright \overline{s_{\text{AB}}} : \mu\,\mathbf{t}.(?(String); \oplus\{\textbf{Success} : \texttt{end}, \textbf{Fail} : \mathbf{t}\}) \cdot$
$s'_{\text{BC}} : \mu\,\mathbf{t}.(\oplus\{\textbf{Success} :!\langle ACK \rangle, \textbf{Fail} :!\langle String \rangle; \mathbf{t}\}); !\langle CredT \rangle; \texttt{end}$

$\Gamma \vdash \overline{s_{\text{AB}}} \,?(x_{\text{CredD}});\; \mathsf{fw}(\overline{s_{\text{AB}}}, s'_{\text{BC}});\; s'_{\text{BC}} \,!\langle x_{\text{CredD}} \rangle;\; \mathsf{close}(\overline{s_{\text{AB}}})$
$\triangleright\, \overline{s_{\text{AB}}} :?(CredT); \mu\,\mathbf{t}.(?(String); \oplus\{\textbf{Success} : \texttt{end}, \textbf{Fail} : \mathbf{t}\}) \cdot$
$s'_{\text{BC}} : \mu\,\mathbf{t}.(\oplus\{\textbf{Success} :!\langle ACK \rangle, \textbf{Fail} :!\langle String \rangle; \mathbf{t}\}); !\langle CredT \rangle; \texttt{end} = \Sigma_1$

$\Gamma \vdash \mathbf{0} \triangleright \overline{s_{\text{AB}}} : \texttt{end} \cdot s'_{\text{BC}} : \texttt{end} \cdot x_{\text{pc}} : \texttt{end}$

$\Gamma \vdash \overline{s_{\text{AB}}} \,!\langle IPc, x_{\text{pc}}, CredA \rangle \triangleright \overline{s_{\text{AB}}} :!\langle nat, nat, CredT \rangle; \texttt{end} \cdot s'_{\text{BC}} : \texttt{end} \cdot x_{\text{pc}} : \texttt{end}$

$\Gamma \vdash b\,(x_{\text{pc}}\colon S).\overline{s_{\text{AB}}} \,!\langle IPc, x_{\text{pc}}, CredA \rangle \triangleright \overline{s_{\text{AB}}} :!\langle nat, nat, CredT \rangle; \texttt{end} \cdot s'_{\text{BC}} : \texttt{end}$

$\Gamma \vdash s'_{\text{BC}} \,!\langle CredA \rangle;\; b\,(x_{\text{pc}}\colon S).\overline{s_{\text{AB}}} \,!\langle IPc, x_{\text{pc}}, CredA \rangle$
$\triangleright\, \overline{s_{\text{AB}}} :!\langle nat, nat, CredT \rangle; \texttt{end} \cdot s'_{\text{BC}} :!\langle CredT \rangle; \texttt{end} = \Sigma_2$

$\Gamma \vdash \mathsf{make}\ (CredA);\; s'_{\text{BC}} \,!\langle CredA \rangle;\; b\,(x_{\text{pc}}\colon S).\overline{s_{\text{AB}}} \,!\langle IPc, x_{\text{pc}}, CredA \rangle \mid s_{\text{AB}} \,?(x_{\text{CredD}});\; fw(\overline{s_{\text{AB}}}, s'_{\text{BC}});$
$s'_{\text{BC}} \,!\langle CredD \rangle;\; \mathsf{close}(\overline{s_{\text{AB}}}) \triangleright \Sigma_1 \cdot \Sigma_2$

This process is typed regarding the session with A as the sending of all the information for the new connection with C and by the reception of a credential type, concluding with the forwarding type - sending part. Concerning the session with C, it sends a credential and has the type of the forwarding function - receiving part. This session terminates with the sending of another credential.

C - $[[R \mid \overline{s'_{\text{BC}}} : \langle S^3 \rangle \vec{h^3}]]$

$\text{C} = \overline{s'_{\text{BC}}} \,?(x_{\text{CredA}});\; (\boldsymbol{\nu}\, pc)\; (\bar{b}\,(pc\colon S).\mathsf{buffer}(\overline{s'_{\text{BC}}});\; \overline{s'_{\text{BC}}} \,?(x_{\text{CredD}});\; pc\,(x\colon S).x\,?(y_{\text{CredA}});$
$\texttt{if}\ x_{\text{CredA}} == y_{\text{CredA}}\ \texttt{then}\ x \triangleleft \textbf{Success}\ \texttt{else}\ x \triangleleft \textbf{Fail};\ \mathsf{close}(x);\ \mathsf{close}(pc))$

Firstly, it receives a credential from B, followed by the opening of a new port *pc*. After that, it sends the information regarding the port to B using the shared channel *b*, starts the buffering of the lost messages and receives another credential indicating the subsequent delegation. The waiting

for a new connection is then ensued, which results in the receiving of one credential to check if it matches the one expected. Again, if the credentials match, the delegation is concluded and the following one is triggered; otherwise the new session is closed together with the port pc.

Typing

$\Gamma \vdash \mathbf{0} \triangleright \overline{s'_{\mathrm{BC}}} \colon \mathtt{end} \cdot pc \colon \mathtt{end} \cdot x \colon \mathtt{end}$

$\Gamma \vdash x \triangleleft \textbf{Fail};\ \mathsf{close}(x);\ \mathsf{close}(pc) \triangleright \overline{s'_{\mathrm{BC}}} \colon \mathtt{end} \cdot pc \colon \mathtt{end} \cdot x \colon \oplus\{\textbf{Success} : \mathtt{end}, \textbf{Fail} : \mathtt{end}\}$

Let $Q = \mathtt{if}\ x_{\mathrm{CredA}} == y_{\mathrm{CredA}}\ \mathtt{then}\ x \triangleleft \textbf{Success}\ \mathtt{else}\ x \triangleleft \textbf{Fail};\ \mathsf{close}(x);\ \mathsf{close}(pc)$

$\Gamma \vdash Q \triangleright \overline{s'_{\mathrm{BC}}} \colon \mathtt{end} \cdot pc \colon \mathtt{end} \cdot x \colon \oplus\{\textbf{Success} : \mathtt{end}, \textbf{Fail} : \mathtt{end}\}$

$\Gamma \vdash x\,?(y_{\mathrm{CredA}});\ Q \triangleright \overline{s'_{\mathrm{BC}}} \colon \mathtt{end} \cdot pc \colon \mathtt{end} \cdot x \colon ?(\mathit{CredT}); \oplus\{\textbf{Success} : \mathtt{end}, \textbf{Fail} : \mathtt{end}\}$

$\Gamma \vdash pc\,(x \colon S).x\,?(y_{\mathrm{CredA}});\ Q \triangleright \overline{s'_{\mathrm{BC}}} \colon \mathtt{end} \cdot pc \colon \mathtt{end}$

$\Gamma \vdash \overline{s'_{\mathrm{BC}}}\,?(x_{\mathrm{CredD}});\ pc\,(x \colon S).x\,?(y_{\mathrm{CredA}});\ Q \triangleright \overline{s'_{\mathrm{BC}}} \colon ?(\mathit{CredT}); \mathtt{end} \cdot pc \colon \mathtt{end}$

$\Gamma \vdash \mathsf{buffer}(\overline{s'_{\mathrm{BC}}});\ \overline{s'_{\mathrm{BC}}}\,?(x_{\mathrm{CredD}});\ pc\,(x \colon S).x\,?(y_{\mathrm{CredA}});\ Q$
$\quad \triangleright \overline{s'_{\mathrm{BC}}} \colon \mu\,\mathbf{t}.(?(\mathit{String}); \oplus\{\textbf{Success} : \mathtt{end}, \textbf{Fail} : \mathbf{t}\}); ?(\mathit{CredT}); \mathtt{end} \cdot pc \colon \mathtt{end}$

$\Gamma \vdash \overline{b}\,(pc \colon S).\mathsf{buffer}(\overline{s'_{\mathrm{BC}}});\ \overline{s'_{\mathrm{BC}}}\,?(x_{\mathrm{CredD}});\ pc\,(x \colon S).x\,?(y_{\mathrm{CredA}});\ Q$
$\quad \triangleright \overline{s'_{\mathrm{BC}}} \colon \mu\,\mathbf{t}.(?(\mathit{String}); \oplus\{\textbf{Success} : \mathtt{end}, \textbf{Fail} : \mathbf{t}\}); ?(\mathit{CredT}); \mathtt{end}$

$\Gamma \vdash \overline{s'_{\mathrm{BC}}}\,?(x_{\mathrm{CredA}});\ (\boldsymbol{\nu}\, pc)\ (\overline{b}\,(pc \colon S).\mathsf{buffer}(\overline{s'_{\mathrm{BC}}});\ \overline{s'_{\mathrm{BC}}}\,?(x_{\mathrm{CredD}});\ pc\,(x \colon S).x\,?(y_{\mathrm{CredA}});\ Q)$
$\quad \triangleright \overline{s'_{\mathrm{BC}}} \colon ?(\mathit{CredT}); \mu\,\mathbf{t}.(?(\mathit{String}); \oplus\{\textbf{Success} : \mathtt{end}, \textbf{Fail} : \mathbf{t}\}); ?(\mathit{CredT}); \mathtt{end}$

Regarding the typification of this process, it receives a credential type from C and enters in the recursion of the buffering type. It is then concluded by the reception of another credential. Concerning the session with A, it is typed as the reception of one credential followed by the selection of two conclusive types.

D - $[[T \mid \overline{s''_{\mathrm{AD}}} : \langle S^5 \rangle \vec{h^5}]]$

D = $\mathbf{0}$

Like in the Resending protocol, party D is not supposed to do anything since the succeeding delegation only occurs after the first one is concluded, entering in one of the two previous cases.

8.3.5 Case 4

A - $[[P \mid s_{\mathrm{AB}} : \langle S \rangle \vec{h} \mid s''_{\mathrm{AD}} : \langle S^4 \rangle \vec{h^4}]]$

A = $\mathsf{make}\ (\mathrm{CredB});\ s''_{\mathrm{AD}}\,!\langle \mathit{CredB} \rangle;\ a\,(x_{\mathrm{pd}} \colon S).s_{\mathrm{AB}}\,!\langle \mathit{IPd}, x_{\mathrm{pd}}, \mathit{CredB} \rangle \mid s_{\mathrm{AB}}\,?(x_{\mathrm{IPc}}, y_{\mathrm{pc}}, z_{\mathrm{CredA}});$ $s''_{\mathrm{AD}}\,!\langle x_{\mathrm{IPc}}, y_{\mathrm{pc}}, z_{\mathrm{CredA}} \rangle;\ \mathsf{close}(s_{\mathrm{AB}})$

In the double delegation case, A begins by creating a new credential for the delegation with D and sends it that credential. It then receives the port for the new session and sends all that information back to A. Simultaneously, it receives the information regarding the delegation to C and forwards that information to D to notify it that C is the one who is going to establish a connection, not B like originally planned. This process terminates with the closure of the session with B.

Typing

$\Gamma \vdash \mathbf{0} \triangleright s_{\text{AB}} : \mathsf{end} \cdot s''_{\text{AD}} : \mathsf{end}$

$\Gamma \vdash s''_{\text{AD}} \,!\langle x_{\text{IPc}}, y_{\text{pc}}, z_{\text{CredA}} \rangle;\ \mathsf{close}(s_{\text{AB}}) \triangleright s_{\text{AB}} : \mathsf{end} \cdot s''_{\text{AD}} : !\langle nat, nat, CredT \rangle; \mathsf{end}$

$\Gamma \vdash s_{\text{AB}} \,?(x_{\text{IPc}}, y_{\text{pc}}, z_{\text{CredA}});\ s''_{\text{AD}} \,!\langle x_{\text{IPc}}, y_{\text{pc}}, z_{\text{CredA}} \rangle;\ \mathsf{close}(s_{\text{AB}})$
$\quad \triangleright s_{\text{AB}} : ?(nat, nat, CredT); \mathsf{end} \cdot s''_{\text{AD}} : !\langle nat, nat, CredT \rangle; \mathsf{end} = \Sigma_1$

$\Gamma \vdash \mathbf{0} \triangleright s_{\text{AB}} : \mathsf{end} \cdot s''_{\text{AD}} : \mathsf{end} \cdot x_{\text{pd}} : \mathsf{end}$

$\Gamma \vdash s_{\text{AB}} \,!\langle IPd, x_{\text{pd}}, CredB \rangle \triangleright s_{\text{AB}} : !\langle nat, nat, CredT \rangle; \mathsf{end} \cdot s''_{\text{AD}} : \mathsf{end} \cdot x_{\text{pd}} : \mathsf{end}$

$\Gamma \vdash a\,(x_{\text{pd}} : S).s_{\text{AB}} \,!\langle IPd, x_{\text{pd}}, CredB \rangle \triangleright s_{\text{AB}} : !\langle nat, nat, CredT \rangle; \mathsf{end} \cdot s''_{\text{AD}} : \mathsf{end}$

$\Gamma \vdash s''_{\text{AD}} \,!\langle CredB \rangle;\ a\,(x_{\text{pd}} : S).s_{\text{AB}} \,!\langle IPd, x_{\text{pd}}, CredB \rangle$
$\quad \triangleright s_{\text{AB}} : !\langle nat, nat, CredT \rangle; \mathsf{end} \cdot s''_{\text{AD}} : !\langle CredT \rangle; \mathsf{end} = \Sigma_2$

$\Gamma \vdash \mathsf{make}\ (CredB);\ s''_{\text{AD}} \,!\langle CredB \rangle;\ a\,(x_{\text{pd}} : S).s_{\text{AB}} \,!\langle IPd, x_{\text{pd}}, CredB \rangle \mid s_{\text{AB}} \,?(x_{\text{IPc}}, y_{\text{pc}}, z_{\text{CredA}});$
$\quad s''_{\text{AD}} \,!\langle x_{\text{IPc}}, y_{\text{pc}}, z_{\text{CredA}} \rangle;\ \mathsf{close}(s_{\text{AB}}) \triangleright \Sigma_1 \cdot \Sigma_2$

This process is typed considering its session with B as the sending of the delegation information concerning D, followed by the reception of the delegation information concerning C. Regarding the session with D, it is typed as the sending of a credential type, followed by the sending of the delegation information regarding C.

B - $[[Q \mid \overline{s_{\text{AB}}} : \langle S' \rangle \vec{h'} \mid s'_{\text{BC}} : \langle S'' \rangle \vec{h''}]]$

B $= \mathsf{make}$ (CredA); $s'_{\text{BC}} \,!\langle CredA \rangle;\ b\,(x_{\text{pc}} : S).\overline{s_{\text{AB}}} \,!\langle IPc, x_{\text{pc}}, CredA \rangle \mid \overline{s_{\text{AB}}} \,?(x_{\text{IPd}}, y_{\text{pd}}, z_{\text{CredB}});$ $\mathsf{fw}(\overline{s_{\text{AB}}}, s'_{\text{BC}});\ s'_{\text{BC}} \,!\langle x_{\text{IPd}}, y_{\text{pd}}, z_{\text{CredB}} \rangle;\ \mathsf{close}(\overline{s_{\text{AB}}})$

B is responsible for creating a new credential, sending it to C, receiving the opened port *pc* and sending all this information to A. At the same time, it receives the information regarding the delegation with D, starts the forwarding mode and sends to C the information regarding the other delegation. The session $\overline{s_{\text{AB}}}$ is then closed.

Typing

$\Gamma \vdash \mathbf{0} \triangleright \overline{s_{\text{AB}}} : \mathsf{end} \cdot s'_{\text{BC}} : \mathsf{end}$

$\Gamma \vdash s'_{\text{BC}} \,!\langle x_{\text{IPd}}, y_{\text{pd}}, z_{\text{CredB}} \rangle;\ \mathsf{close}(\overline{s_{\text{AB}}}) \triangleright \overline{s_{\text{AB}}} : \mathsf{end} \cdot s'_{\text{BC}} : !\langle nat, nat, CredT \rangle; \mathsf{end}$

$\Gamma \vdash \mathsf{fw}(\overline{s_{\text{AB}}}, s'_{\text{BC}});\ s'_{\text{BC}} \,!\langle x_{\text{IPd}}, y_{\text{pd}}, z_{\text{CredB}} \rangle;\ \mathsf{close}(\overline{s_{\text{AB}}})$
$\quad \triangleright \overline{s_{\text{AB}}} : \mu\,\mathbf{t}.(?(String); \oplus\{\mathbf{Success} : \mathsf{end}, \mathbf{Fail} : \mathbf{t}\}) \cdot$
$\quad\ s'_{\text{BC}} : \mu\,\mathbf{t}.(\oplus\{\mathbf{Success} : !\langle ACK \rangle, \mathbf{Fail} : !\langle String \rangle; \mathbf{t}\}); !\langle nat, nat, CredT \rangle; \mathsf{end}$

$\Gamma \vdash \overline{s_{\text{AB}}} \,?(x_{\text{IPd}}, y_{\text{pd}}, z_{\text{CredB}});\ \mathsf{fw}(\overline{s_{\text{AB}}}, s'_{\text{BC}});\ s'_{\text{BC}} \,!\langle x_{\text{IPd}}, y_{\text{pd}}, z_{\text{CredB}} \rangle;\ \mathsf{close}(\overline{s_{\text{AB}}})$
$\quad \triangleright \overline{s_{\text{AB}}} : ?(nat, nat, CredT); \mu\,\mathbf{t}.(?(String); \oplus\{\mathbf{Success} : \mathsf{end}, \mathbf{Fail} : \mathbf{t}\}) \cdot$
$\quad\ s'_{\text{BC}} : \mu\,\mathbf{t}.(\oplus\{\mathbf{Success} : !\langle ACK \rangle, \mathbf{Fail} : !\langle String \rangle; \mathbf{t}\}); !\langle nat, nat, CredT \rangle; \mathsf{end} = \Sigma_1$

$\Gamma \vdash \mathbf{0} \triangleright \overline{s_{\text{AB}}} : \mathsf{end} \cdot s'_{\text{BC}} : \mathsf{end} \cdot x_{\text{pc}} : \mathsf{end}$

$\Gamma \vdash \overline{s_{\text{AB}}} \,!\langle IPc, x_{\text{pc}}, CredA \rangle \triangleright \overline{s_{\text{AB}}} : !\langle nat, nat, CredT \rangle; \mathsf{end} \cdot s'_{\text{BC}} : \mathsf{end} \cdot x_{\text{pc}} : \mathsf{end}$

$\Gamma \vdash b\,(x_{\text{pc}} : S).\overline{s_{\text{AB}}} \,!\langle IPc, x_{\text{pc}}, CredA \rangle \triangleright \overline{s_{\text{AB}}} : !\langle nat, nat, CredT \rangle; \mathsf{end} \cdot s'_{\text{BC}} : \mathsf{end}$

$\Gamma \vdash s'_{\text{BC}} \,!\langle CredA \rangle;\ b\,(x_{\text{pc}} : S).\overline{s_{\text{AB}}} \,!\langle IPc, x_{\text{pc}}, CredA \rangle$
$\quad \triangleright \overline{s_{\text{AB}}} : !\langle nat, nat, CredT \rangle; \mathsf{end} \cdot s'_{\text{BC}} : !\langle CredT \rangle; \mathsf{end} = \Sigma_2$

$\Gamma \vdash \mathsf{make}\ (CredA);\ s'_{\text{BC}} \,!\langle CredA \rangle;\ b\,(x_{\text{pc}} : S).\overline{s_{\text{AB}}} \,!\langle IPc, x_{\text{pc}}, CredA \rangle \mid \overline{s_{\text{AB}}} \,?(x_{\text{IPd}}, y_{\text{pd}}, z_{\text{CredB}});$
$\quad \mathsf{fw}(\overline{s_{\text{AB}}}, s'_{\text{BC}});\ s'_{\text{BC}} \,!\langle x_{\text{IPd}}, y_{\text{pd}}, z_{\text{CredB}} \rangle;\ \mathsf{close}(\overline{s_{\text{AB}}}) \triangleright \Sigma_1 \cdot \Sigma_2$

The left part of the parallel composition is typed for $\overline{s_{\mathrm{AB}}}$ as the reception of the delegation information followed by the forwarding type - sending part. For s'_{BC} it is typed as the receiving part of the forwarding type, followed by the sending of the succeeding delegation.
On the other hand, the right part of the parallel composition is typed as the sending of the first delegation information for the session shared with A, whereas for the session shared with C, it is typed as the sending of a credential type. The resulting typification is the composition of both left and right parts of the parallelisation.

C - $[[R \mid \overline{s'_{\mathrm{BC}}} : \langle S^3 \rangle \vec{h^3}]]$

$$\mathrm{C} = \overline{s'_{\mathrm{BC}}}\,?(x_{\mathrm{CredA}});\ (\boldsymbol{\nu}\,pc)\ (\bar{b}\,(pc\colon S).\mathsf{buffer}(\overline{s'_{\mathrm{BC}}});\ \overline{s'_{\mathrm{BC}}}\,?(x_{\mathrm{IPd}}, y_{\mathrm{pd}}, z_{\mathrm{CredB}});\ \mathsf{close}(pc);\ \overline{y_{\mathrm{pd}}}\,(x\colon S).x\,!\langle z_{\mathrm{CredB}}\rangle;\ x \triangleright \{\mathbf{Success} : \mathbf{0}, \mathbf{Fail} : \mathsf{close}(x)\})$$

Starting by receiving a credential from B, it opens a new port pc, notifies B of that port and starts buffering the lost messages. Then, it receives information regarding the new delegation taking place, so it can close pc. Now, it needs to connect to D, send the credential it had just received and start branching according to the correctness of that credential. This is the one thing that distinguishes the Forwarding protocol from the Resending one: there is no need of an intermediate connection and C connects directly to D after receiving the notification to do so.

Typing

$\Gamma \vdash \mathbf{0} \triangleright \overline{s'_{\mathrm{BC}}}\colon \mathsf{end} \cdot pc\colon \mathsf{end} \cdot x\colon \mathsf{end}$

$\Gamma \vdash x \triangleright \{\mathbf{Success} : \mathbf{0}, \mathbf{Fail} : \mathsf{close}(x)\} \triangleright \overline{s'_{\mathrm{BC}}}\colon \mathsf{end} \cdot pc\colon \mathsf{end} \cdot x\colon \&\{\mathbf{Success} : \mathsf{end}, \mathbf{Fail} : \mathsf{end}\}$

$\Gamma \vdash x\,!\langle z_{\mathrm{CredB}}\rangle;\ x \triangleright \{\mathbf{Success} : \mathbf{0}, \mathbf{Fail} : \mathsf{close}(x)\}$
$\quad \triangleright \overline{s'_{\mathrm{BC}}}\colon \mathsf{end} \cdot pc\colon \mathsf{end} \cdot x :!\langle CredT\rangle; \&\{\mathbf{Success} : \mathsf{end}, \mathbf{Fail} : \mathsf{end}\}$

$\Gamma \vdash \overline{y_{\mathrm{pd}}}\,(x\colon S).x\,!\langle z_{\mathrm{CredB}}\rangle;\ x \triangleright \{\mathbf{Success} : \mathbf{0}, \mathbf{Fail} : \mathsf{close}(x)\} \triangleright \overline{s'_{\mathrm{BC}}}\colon \mathsf{end} \cdot pc\colon \mathsf{end}$

$\Gamma \vdash \overline{s'_{\mathrm{BC}}}\,?(x_{\mathrm{IPd}}, y_{\mathrm{pd}}, z_{\mathrm{CredB}});\ \mathsf{close}(pc);\ \overline{y_{\mathrm{pd}}}\,(x\colon S).x\,!\langle z_{\mathrm{CredB}}\rangle;$
$\quad x \triangleright \{\mathbf{Success} : \mathbf{0}, \mathbf{Fail} : \mathsf{close}(x)\} \triangleright \overline{s'_{\mathrm{BC}}} :?(nat, nat, CredT); \mathsf{end} \cdot pc\colon \mathsf{end}$

$\Gamma \vdash \mathsf{buffer}(\overline{s'_{\mathrm{BC}}});\ \overline{s'_{\mathrm{BC}}}\,?(x_{\mathrm{IPd}}, y_{\mathrm{pd}}, z_{\mathrm{CredB}});\ \mathsf{close}(pc);\ \overline{y_{\mathrm{pd}}}\,(x\colon S).x\,!\langle z_{\mathrm{CredB}}\rangle;$
$\quad x \triangleright \{\mathbf{Success} : \mathbf{0}, \mathbf{Fail} : \mathsf{close}(x)\}$
$\quad \triangleright \overline{s'_{\mathrm{BC}}}\colon \mu\,\mathbf{t}.(?(String); \oplus\{\mathbf{Success} : \mathsf{end}, \mathbf{Fail} : \mathbf{t}\}); ?(nat, nat, CredT); \mathsf{end} \cdot pc\colon \mathsf{end}$

$\Gamma \vdash \bar{b}\,(pc\colon S).\mathsf{buffer}(\overline{s'_{\mathrm{BC}}});\ \overline{s'_{\mathrm{BC}}}\,?(x_{\mathrm{IPd}}, y_{\mathrm{pd}}, z_{\mathrm{CredB}});\ \mathsf{close}(pc);\ \overline{y_{\mathrm{pd}}}\,(x\colon S).x\,!\langle z_{\mathrm{CredB}}\rangle;$
$\quad x \triangleright \{\mathbf{Success} : \mathbf{0}, \mathbf{Fail} : \mathsf{close}(x)\}$
$\quad \triangleright \overline{s'_{\mathrm{BC}}}\colon \mu\,\mathbf{t}.(?(String); \oplus\{\mathbf{Success} : \mathsf{end}, \mathbf{Fail} : \mathbf{t}\}); ?(nat, nat, CredT); \mathsf{end}$

$\Gamma \vdash \overline{s'_{\mathrm{BC}}}\,?(x_{\mathrm{CredA}});\ (\boldsymbol{\nu}\,pc)\ (\bar{b}\,(pc\colon S).\mathsf{buffer}(\overline{s'_{\mathrm{BC}}});\ \overline{s'_{\mathrm{BC}}}\,?(x_{\mathrm{IPd}}, y_{\mathrm{pd}}, z_{\mathrm{CredB}});\ \mathsf{close}(pc);\ \overline{y_{\mathrm{pd}}}\,(x\colon S).$
$\quad x\,!\langle z_{\mathrm{CredB}}\rangle;\ x \triangleright \{\mathbf{Success} : \mathbf{0}, \mathbf{Fail} : \mathsf{close}(x)\})$
$\quad \triangleright \overline{s'_{\mathrm{BC}}} :?(CredT); \mu\,\mathbf{t}.(?(String); \oplus\{\mathbf{Success} : \mathsf{end}, \mathbf{Fail} : \mathbf{t}\}); ?(nat, nat, CredT); \mathsf{end}$

Regarding the connection with B, it is typed as the reception of a credential type and entering into the buffering type. This session is finalised with the reception of the information of the existence of another delegation. Conversely, in the session shared with D, it sends a credential and then branches between two empty actions.

D - $[[T \mid \overline{s''_{\text{AD}}} : \langle S^5 \rangle \vec{h^5}]]$

$$\text{D} = \overline{s''_{\text{AD}}}\,?(x_{\text{CredB}});\ (\boldsymbol{\nu}\,pd)\ (\overline{a}\,(pd\colon S).\overline{s''_{\text{AD}}}\,?(x_{\text{IPc}}, y_{\text{pc}}, z_{\text{CredA}});\ pd\,(x\colon S).x\,?(y_{\text{CredB}});$$
$$\texttt{if}\ x_{\text{CredB}} == y_{\text{CredB}}\ \texttt{then}\ x \triangleleft \textbf{Success}\ \texttt{else}\ x \triangleleft \textbf{Fail};\ \mathsf{close}(x);\ \mathsf{close}(pd))$$

Similarly to the previous party, it begins by receiving a credential from A and then opens a new port *pd* for the passive party to connect and notifies A of that port through the channel a. The reception of the notification of the proceeding delegation is ensued, so D can become aware that is C who is going to connect. It is just left to accept the new connection in *pd*, receive the credential from the passive party and selecting to either continue or abort the session according to the credential matching.

Typing

$\Gamma \vdash \mathbf{0} \triangleright \overline{s''_{\text{AD}}} \colon \texttt{end} \cdot pd \colon \texttt{end} \cdot x \colon \texttt{end}$

$\Gamma \vdash x \triangleleft \textbf{Fail};\ \mathsf{close}(x);\ \mathsf{close}(pd) \triangleright \overline{s''_{\text{AD}}} \colon \texttt{end} \cdot pd \colon \texttt{end} \cdot x \colon \oplus\{\textbf{Success} : \texttt{end}, \textbf{Fail} : \texttt{end}\}$

Let $Q = \texttt{if}\ x_{\text{CredB}} == y_{\text{CredB}}\ \texttt{then}\ x \triangleleft \textbf{Success}\ \texttt{else}\ x \triangleleft \textbf{Fail};\ \mathsf{close}(x);\ \mathsf{close}(pd)$

$\Gamma \vdash Q \triangleright \overline{s''_{\text{AD}}} \colon \texttt{end} \cdot pd \colon \texttt{end} \cdot x \colon \oplus\{\textbf{Success} : \texttt{end}, \textbf{Fail} : \texttt{end}\}$

$\Gamma \vdash x\,?(y_{\text{CredB}});\ Q \triangleright \overline{s''_{\text{AD}}} \colon \texttt{end} \cdot pd \colon \texttt{end} \cdot x \colon ?(CredT); \oplus\{\textbf{Success} : \texttt{end}, \textbf{Fail} : \texttt{end}\}$

$\Gamma \vdash pd\,(x\colon S).x\,?(y_{\text{CredB}});\ Q \triangleright \overline{s''_{\text{AD}}} \colon \texttt{end} \cdot pd \colon \texttt{end}$

$\Gamma \vdash \overline{s''_{\text{AD}}}\,?(x_{\text{IPc}}, y_{\text{pc}}, z_{\text{CredA}});\ pd\,(x\colon S).x\,?(y_{\text{CredB}});\ Q \triangleright \overline{s''_{\text{AD}}} \colon ?(nat, nat, CredT); \texttt{end} \cdot pd \colon \texttt{end}$

$\Gamma \vdash \overline{a}\,(pd\colon S).\overline{s''_{\text{AD}}}\,?(x_{\text{IPc}}, y_{\text{pc}}, z_{\text{CredA}});\ pd\,(x\colon S).x\,?(y_{\text{CredB}});\ Q \triangleright \overline{s''_{\text{AD}}} \colon ?(nat, nat, CredT); \texttt{end}$

$\Gamma \vdash \overline{s''_{\text{AD}}}\,?(x_{\text{CredB}});\ (\boldsymbol{\nu}\,pd)\ (\overline{a}\,(pd\colon S).\overline{s''_{\text{AD}}}\,?(x_{\text{IPc}}, y_{\text{pc}}, z_{\text{CredA}});\ pd\,(x\colon S).x\,?(y_{\text{CredB}});\ Q)$
$\quad \triangleright \overline{s''_{\text{AD}}} \colon ?(CredT); ?(nat, nat, CredT); \texttt{end}$

The session shared with A is typed as the reception of a credential type, followed by the reception of the delegation information. Likewise, the new established session shared with C is typed as the receiving of a credential followed by the selection of one of the two empty branches.

8.4 Protocols Correctness

The three properties defined in SJ paper [32] were Linearity, Liveness and Session Consistency. In order to guarantee correctness in the new protocols, it is needed to add an extra property regarding Security. To sum up the resulting four properties are:

P1: Linearity

– There is always a single receiver for each control message sent.

– There are no concurrent inputs or outputs.

P2: Liveness

– *Deadlock Freedom*: There are no circular dependencies between actions.

– *Readiness*: The server is always ready for reconnection.

– *Stuck Freedom*: Only after all the lost messages are identified is that the session being delegated is closed.

P3: Session Consistency The structure and order of the delegated session are preserved by the inexistence of lost or duplicated messages.

P4: Session Security

– *Freshness*: The credentials are fresh for only the current session.

– *Credential Checking*: A session is only completely delegated if the credential that the passive party has matches the one from the session receiver. If this is not the case, an error occurs and the session is closed.

– *Attackers' Protection*: There is protection against attacks from the Network, achieving message authentication, confidentiality and integrity.

8.4.1 Formalised Properties

Definition 1 (Session Sender)**.**
P is called a session sender if $P \equiv s\,!\langle Cred\rangle;\ P_1;\ s'\,!\langle ..Cred..\rangle;\ P_2$

The session sender is responsible for creating the new credentials and sending them to the session receiver and to the passive party.

Definition 2 (Session Receiver)**.**
P is called a session receiver if $P \equiv s\,?(Cred);\ P_1;\ s'\,?(Cred);\ P_2$

On the other hand, the session receiver starts by receiving the credential from the sender, performs some intermediate operations and receives another credential from the passive party. As we saw previously, if those credentials match, the session continues; otherwise the connection is closed.

Definition 3 (Passive Party)**.**
P is called a passive party if $P \equiv s'\,?(..Cred..);\ P_1;\ s\,!\langle Cred\rangle;\ P_2$

The final type of peer is the passive party which starts by receiving the credential from the sender, connects to the receiver and sends that credential back to it.

Notation. *Normally we use A for passive party, B for session sender and C for session receiver.*

Proposition 1 (Linearity)**.** *There is always a single receiver for each control message sent and there are no concurrent inputs or outputs.*

In the formalisation of the property, we call one party as A and other as B, but this can be whatever parties stated in sections 8.2 and 8.3.

The first part can be formalised as:
$\forall P_A \equiv (\boldsymbol{\nu}\, \tilde{a})\ (s\,!\langle e\rangle;\, Q_A \mid R_A)$ where $\nexists s \in R_A$

$\exists Q_B, R_B$ s.t. $P_B \equiv (\boldsymbol{\nu}\, \tilde{b})\ (s\,?(x);\, Q_B \mid R_B)$ where $\nexists s \in R_B$

which states that if A is sending some information and it has no parallel actions that use the same session, then there is always a B that receives that information with no parallel actions using the same session as well.

On the other hand, the part referring no concurrent inputs or output can be written as:
$\forall P_A \equiv (\boldsymbol{\nu}\, \tilde{a})\ (s\,!\langle e\rangle;\, Q_A \mid R_A)$, $\nexists P_B$ s.t. $P_B \equiv (\boldsymbol{\nu}\, \tilde{b})\ (s\,!\langle e\rangle;\, Q_B \mid R_B)$ where $\nexists s \in R_A, R_B$

and, $\forall P_A \equiv (\boldsymbol{\nu}\, \tilde{a})\ (s\,?(x);\, Q_A \mid R_A)$, $\nexists P_B$ s.t. $P_B \equiv (\boldsymbol{\nu}\, \tilde{b})\ (s\,?(x);\, Q_B \mid R_B)$ where $\nexists s \in R_A, R_B$

which is similar to the previous one but explicitly says that whenever A is sending something over session s, there is no other peer using the same session for sending other information. The same applies for receiving operations.
The formal proof of the Linearity proposition can be found in Proof 1.

Proposition 2 (Liveness)**.** *This property is divided in Deadlock Freedom, Readiness and Stuck Freedom.*

Deadlock Freedom:
$\forall P_A, P_B$ s.t. $P_A \equiv s\,?(x);\, R_A$ and $P_B \equiv s'\,?(y);\, R_B$

If $R_A \equiv\ ...\ s'\,!\langle x\rangle\ ...\ \Rightarrow R_B \not\equiv\ ...\ s\,!\langle y\rangle\ ...$

We assume that there are two sessions s and s' shared with the same peer and both of them are performing a receiving operation on each of the sessions. If in the continuation of one peer there is a send operation on the other session, the other peer cannot be sending on the other session as well, i.e. there cannot be a circular dependency. This is proved in Proof 2.

Readiness:
$\forall port\,(x\colon S)$, $\exists P; port\,(x\colon S)$ s.t. $P \xrightarrow{*} (\boldsymbol{\nu}\, port)$

This means that whenever there is an accept for a new session on the shared channel *port*, there must be the creation of that *port* before.
Readiness is proved in Proof 3.

Stuck Freedom:
In order to better explain this protocol, we divided it into the several cases of the delegation.

– *Delegation Case 1:*
$\forall \mathsf{close}(s)$, $\exists P; \mathsf{close}(s)$ s.t. $P \xrightarrow{*} s\,!\langle ACK\rangle$ or $P \xrightarrow{*} s\,?(ACK)$

It is shown here that whenever there is a close of a session, there has to be either a receiving or a sending of an ACK before the closing of that session.

– *Other Delegation Cases - Resending:*
In the resending protocol we assume it is B who resends the lost messages to C, so s_{AB} can be closed after the sending of the Session information (S)

– *Other Delegation Cases - Forwarding:*
$\forall \mathsf{close}(s)$, P session sender, $\exists P; \mathsf{close}(s)$ s.t. $P \xrightarrow{*} \mathsf{fw}(s, s')$

This last case of the Stuck Freedom property states that the session sender (see Definition 1) only closes its session after performing the forwarding operation (fw).
The formal proof for Stuck Freedom can be found in Proof 4.

Proposition 3 (Session Security)**.** *This property concerns the new security part of SJ and is composed by Freshness, Credential Checking and Attacker's Protection.*

Freshness:
$\forall$ protocols, $\exists P \equiv \mathsf{make}\ (Cred)$ s.t. P is a session sender

This property refers that every session sender in the protocols listed in sections 8.2 and 8.3 creates a new credential for that specific session, which is fresh by the reduction rules. Freshness is proved in Proof 5.

Credential Checking:
$\forall$ protocols, $\exists P_C \equiv s\,?(x_{\text{Cred}});\ P; port\,(x\colon S).x\,?(y_{\text{Cred}});$
$\quad\quad\texttt{if}\ x_{\text{Cred}} == y_{\text{Cred}}\ \texttt{then}\ x \triangleleft \textbf{Success}; Q\ \texttt{else}\ x \triangleleft \textbf{Fail}; \mathsf{close}(x);$

and, $\exists P_A \equiv \overline{port}\,(x\colon S).x\,!\langle y_{\text{Cred}}\rangle;\ x \triangleright \{\textbf{Success} : Q, \textbf{Fail} : \mathsf{close}(x)\}$

s.t. P_C is the session receiver and P_A is the passive party

Finally, this last property states that in all protocols stated previously, if the party is the session receiver it starts by receiving a credential from the session sender followed by accepting a new connection on the opened *port*. Then it receives another credential from the passive party and if both credentials match, the Success branch is selected and the process continues; otherwise the new session is closed. On the other hand, if the party is the passive party, it starts by requesting a connection in *port* followed by sending the credential it had received from the session sender. If the credential is accepted it follows the Success branch, otherwise goes to the Fail branch which also closes the new session created between the passive party and the session receiver.
The proof of Credential Checking can be found in Proof 6.

Attacker's Protection:
All connections are protected from attacks of the network because, according to this modelling, all existing channels are private and by the definition of private channels they achieve security properties similar to the TLS. In order to proof this property, one should model all protocols in a lower level and one suitable modelling from the protocols would be using the Applied Pi Calculus [9], which is left as future work.
The proof that the current modelling achieves the mentioned security properties can be found in 7.

8.4.2 Proving Protocols Correctness

In order to prove Protocols Correctness, we will use A_1, A_2, ..., A_n for the rest of the current protocol as a shortcut so it is clear and easier to read. We will also write the two parties that are

interacting in different lines but the meaning is the parallel interaction of those parties. Once again this is to avoid too complex notations and create fluency in reading these proofs.
So, as an example, whenever there is:

$A \longrightarrow A_1 \longrightarrow A_2$ .....
$B \longrightarrow B_1 \longrightarrow B_2$ .....

it should be read as:
$A \mid B \longrightarrow A_1 \mid B_1 \longrightarrow A_2 \mid B_2$ .....

To conclude, we just selected the most interesting examples to proof, being the remaining ones either similar or sometimes identical.

Proof 1 (Linearity)**.**

- *Case 1 - Forwarding:*

There are no parallel actions so we can exclude R_A and R_B. Then we can easily see that every message sent has only one receiver:

A with B

$A \xrightarrow{*} s_{\text{AB}}\,?(x_{\text{IPc}}, y_{\text{pc}}, z_{\text{CredA}});\, s_{\text{AB}}\,!\langle ACK\rangle;\, A_1 \longrightarrow s_{\text{AB}}\,!\langle ACK\rangle;\, A_1 \longrightarrow A_1$

$B \xrightarrow{*} \overline{s_{\text{AB}}}\,!\langle IPc, x_{\text{pc}}, CredA\rangle;\, s_{\text{AB}}\,?(x_{\text{ACK}});\, B_1 \longrightarrow s_{\text{AB}}\,?(x_{\text{ACK}});\, B_1 \longrightarrow B_1$

From this we can see that B sends some information to A, which it receives, sending back an ACK confirming the reception. Then B receives the ACK and both continue with their processes.

A with C

Let Q = $\texttt{if}\ x_{\text{CredA}} == y_{\text{CredA}}\ \texttt{then}\ x \triangleleft \textbf{Success}\ \texttt{else}\ x \triangleleft \textbf{Fail};\ \mathsf{close}(x);\ \mathsf{close}(pc)$

$A \xrightarrow{*} x\,!\langle z_{\text{CredA}}\rangle;\, x \triangleright \{\textbf{Success} : \mathbf{0}, \textbf{Fail} : \mathsf{close}(x)\} \longrightarrow x \triangleright \{\textbf{Success} : \mathbf{0}, \textbf{Fail} : \mathsf{close}(x)\} \longrightarrow \mathbf{0}$

$C \xrightarrow{*} x\,?(y_{\text{CredA}});\, Q \longrightarrow Q \longrightarrow \mathbf{0}$

A sends the credential to C and then, according to the same credential, C selects one of the branches. Both processes are then concluded.

B with C

$B \xrightarrow{*} s'_{\text{BC}}\,!\langle CredA\rangle;\, B_1 \longrightarrow B_1$

$C = \overline{s'_{\text{BC}}}\,?(x_{\text{CredA}});\, C_1 \longrightarrow C_1$

We can easily see that what B sends C receives, concluding that Delegation Case 1 (Forwarding) preserves the Linearity property since there is always a receiver for each message sent.

- *Case 3 - Forwarding:*

A with B

$A \xrightarrow{*} s_{\text{AB}}\,!\langle CredD\rangle;\, s_{\text{AB}}\,?(x_{\text{IPc}}, y_{\text{pc}}, z_{\text{CredA}});\, A_1 \longrightarrow s_{\text{AB}}\,?(x_{\text{IPc}}, y_{\text{pc}}, z_{\text{CredA}});\, A_1 \longrightarrow A_1$

$B \xrightarrow{*} \overline{s_{\text{AB}}}\,?(x_{\text{CredD}});\, B_1 \mid \overline{s_{\text{AB}}}\,!\langle IPc, x_{\text{pc}}, CredA\rangle \longrightarrow B_1 \mid \overline{s_{\text{AB}}}\,!\langle IPc, x_{\text{pc}}, CredA\rangle \longrightarrow B_1 \mid \mathbf{0}$

A sends the credential of D to B and then B sends all the session information to A. With this last sending the right branch of B terminates and only the left branch continues, whereas A continues with its process.

A with C

Identical to previous case

B with C

$B \xrightarrow{*} s'_{\text{BC}}\,!\langle CredA\rangle;\, B_1 \mid B_2 \longrightarrow B_1 \mid B_2 \xrightarrow{*} B_1 \mid s'_{\text{BC}}\,!\langle x_{\text{CredD}}\rangle;\, B_3 \longrightarrow B_1 \mid B_3$

$C = \overline{s'_{\text{BC}}}\,?(x_{\text{CredA}});\, C_1 \longrightarrow C_1 \xrightarrow{*} \overline{s'_{\text{BC}}}\,?(x_{\text{CredD}});\, C_2 \longrightarrow C_2$

B starts by sending the credential of A to C. Afterwards, after several reductions, the other branch of B sends the credential of D to C as well. Dual to this, C starts by receiving the first credential and after several steps, it receives the other one.

We can then conclude that Delegation Case 3 (Forwarding) also preservers the Linearity property.

Proof 2 (Deadlock Freedom)**.**

Assuming the processes are deadlocked, then there must be at least two processes A and B sharing two sessions s and s' with each other such that: $A \equiv s\,?(x);\, R_A$ and $B \equiv s'\,?(y);\, R_B$ with $R_A \xrightarrow{*} s'\,!\langle x\rangle$ and $R_B \xrightarrow{*} s\,!\langle y\rangle$.
If we see, for example, Case 1 (Forwarding) and assume that in party A, x is also shared with B, then both s_{AB} and x are shared with it. However there are no circular dependencies, i.e there is no $A \xrightarrow{*} s_{\text{AB}}\,?(z);\, R_A$ and $B \xrightarrow{*} x\,?(y);\, R_B$ with both $R_A \xrightarrow{*} x\,!\langle z\rangle$ and $R_B \xrightarrow{*} s_{\text{AB}}\,!\langle y\rangle$. Henceforth, there is no deadlock and by *reductio ad absurdum* we proved there is no deadlock between A and B in Case 1 – Forwarding, achieving the Deadlock Freedom property.
All the remaining cases are similar.

Proof 3 (Readiness)**.**

- *Delegation Cases 1, 2 and 3:*

In these cases C is the server, so we just need to check for all accepts on a certain port if that port was previously declared as new.

$\forall C_1, pc$ if $C \xrightarrow{*} (\boldsymbol{\nu}\, pc)\ (C_1)$ then $C_1 \xrightarrow{*} pc\,(x\colon S).C_2$

From this we can see that before accepting x on port pc, a new pc was instantiated, hence Delegation Cases 1, 2 and 3 (both Resending and Forwarding) preserve the Readiness property.

- *Delegation Case 4:*

In this case both C and D are servers, and once more we need to check if the ports were declared as new before accepting on those ports.

$\forall C_1, pc$ if $C \xrightarrow{*} (\boldsymbol{\nu}\, pc)\ (C_1)$ then $C_1 \xrightarrow{*} pc\,(x\colon S).C_2$

$\forall D_1, pd$ if $D \xrightarrow{*} (\boldsymbol{\nu}\, pd)\ (D_1)$ then $D_1 \xrightarrow{*} pd\,(y\colon S).D_2$

In C we can see that before accepting x on port pc, a new pc was instantiated, whereas in D a new port pd was declared before accepting y. Therefore, Delegation Case 4 (both Resending and Forwarding) also preserves the Readiness property.

Proof 4 (Stuck Freedom)**.**

- *Case 1 - Resending:*

$$A \xrightarrow{*} s_{\mathrm{AB}}\,!\langle ACK\rangle;\ \mathsf{close}(s_{\mathrm{AB}});\ A_1 \longrightarrow \mathsf{close}(s_{\mathrm{AB}});\ A_1 \longrightarrow A_1$$

$$B \xrightarrow{*} \overline{s_{\mathrm{AB}}}\,?(x_{\mathrm{ACK}});\ \mathsf{close}(\overline{s_{\mathrm{AB}}}) \longrightarrow \mathsf{close}(\overline{s_{\mathrm{AB}}}) \longrightarrow \mathbf{0}$$

Like stated previously, A sends an ACK to B, so it can close the session s_{AB} and proceed with the rest of its process. On the other hand B receives that ACK and closes its part of the session. From this we can conclude that Delegation Case 1 (Resending) achieves the Stuck Freedom property.

- *Case 4 - Forwarding:*

$$B \xrightarrow{*} \mathsf{fw}(\overline{s_{\mathrm{AB}}}, s'_{\mathrm{BC}});\ B_1 \longrightarrow B_1$$

$$B_1 \xrightarrow{*} \mathsf{close}(\overline{s_{\mathrm{AB}}}) \longrightarrow \mathbf{0}$$

In case of a Forwarding protocol we only need to check if the session sender forwards the lost messages before it closes the session with the passive party. As we can see, B forwards the lost messages from A to C before it closes the session s_{AB}. As a consequence we can infer that Delegation Case 4 (Forwarding) achieves the Stuck Freedom property.

All other Forwarding cases are identical to this one so we do not need to repeat the same procedure to prove their correctness.

Proof 5 (Security – Freshness)**.**

- *Case 1 - Resending:*

$$B = \mathsf{make}\ (CredA);\ s'_{\mathrm{BC}}\,!\langle CredA\rangle;\ B_1 \longrightarrow s'_{\mathrm{BC}}\,!\langle CredA\rangle;\ B_1 \longrightarrow B_1$$

$$B_1 \xrightarrow{*} \overline{s_{\mathrm{AB}}}\,!\langle S', IPc, x_{\mathrm{pc}}, CredA\rangle;\ B_2 \longrightarrow B_2$$

Before delegation, B creates a new credential specific for the current session, which it sends to both C and A (for A it sends together with other important session information). So, we can conclude that Delegation Case 1 (Resending) preserves the Freshness property since the credential is fresh (according to make reduction rules).

In all other protocols the procedure is the same, except in Cases 3 and 4 (in both resending and forwarding) as the party A also creates a new credential.

Proof 6 (Security – Credential Checking)**.**

- *Case 2 - Resending:*

Let P = $x \triangleright \{\textbf{Success} : x\,!\langle LM(S - S')\rangle, \textbf{Fail} : \mathsf{close}(x)\}$
and Q = $\texttt{if}\ x_{\text{CredA}} == y_{\text{CredA}}\ \texttt{then}\ x \triangleleft \textbf{Success};\ x\,?(x_{\text{LM}})\ \texttt{else}\ x \triangleleft \textbf{Fail};\ \mathsf{close}(x);\ \mathsf{close}(pc)$

$A \xrightarrow{*} \overline{y_{\text{pc}}}\,(x\colon S).x\,!\langle z_{\text{CredA}}\rangle;\ P$
$\longrightarrow x\,!\langle z_{\text{CredA}}\rangle;\ P$
$\longrightarrow P$
$\longrightarrow \mathbf{0}$

$C = \overline{s'_{\text{BC}}}\,?(x_{\text{CredA}});\ C_1 \longrightarrow C_1$
$C_1 \xrightarrow{*} pc\,(x\colon S).x\,?(y_{\text{CredA}});\ Q$
$\longrightarrow x\,?(y_{\text{CredA}});\ Q$
$\longrightarrow Q$
$\longrightarrow \mathbf{0}$

Before A and C interact, C receives a credential from B (shown in C reduction). Then A connects to C via p_c and establishes a new session x. To conclude, A sends the credential it has on C and C compares this credential with the one it had received from B and if they match, the session is continued; otherwise the session is closed along with the port p_c.
We can therefore conclude that Delegation Case 2 (Resending) achieves the Credential Checking property.

All other protocols are identical to this one, except Case 4 (Forwarding) in which party C performs a credential checking with D, and Case 4 (Resending) where A does a credential checking with C and then C does another one with D. However, these checking are similar to the one stated before so there is no need to repeat the procedure to prove its correctness.

Proof 7 (Security – Attackers' Protection)**.**

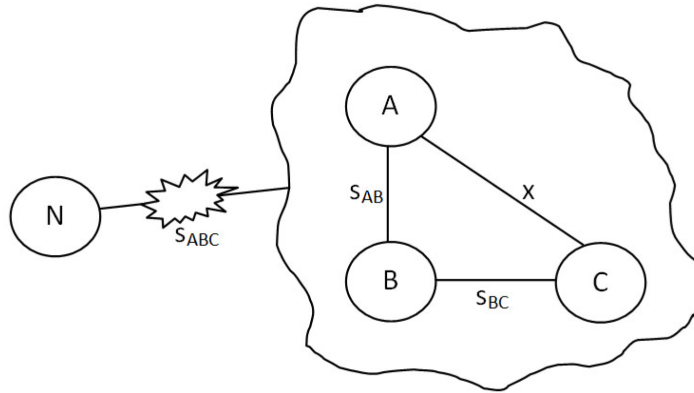


Figure 8.1: Environment of possible attacker in Case 1

In order to proof that this modelling achieves protection from attackers, we can formalise the Network (attacker) and interact with the other parties.

N = $s_{\text{ABC}}\,?(S', x_{\text{IPc}}, y_{\text{pc}}, z_{\text{CredA}});\ s_{\text{ABC}}\,!\langle ACK\rangle;\ \overline{y_{\text{pc}}}\,(z\colon S).z\,!\langle z_{\text{CredA}}\rangle;$
$z \triangleright \{\textbf{Success} : z\,!\langle LM(S - S')\rangle, \textbf{Fail} : \mathbf{0}\}$

The Network starts by receiving the information of the delegation, sends the acknowledgment and connects to the session receiver. Note that the network does not close the session after the ACK since in a point of view of an attacker, it would want to remain connected to all parties. Afterwards, it sends the credential received in the first message and, in case of success, behaves like a normal peer; otherwise it does not close the connection for the same reasons as those stated before.

As we can see, this attacker remains on hold forever, since it will not receive anything from s_{ABC}. All three parties interact with each other on private channels and do not have any channel for communication with the Network. Hence, we can guarantee protection from this attacker and with this we proved this security property holds on Delegation Case 1. This case can be seen on Figure 8.1.

All other attackers should be modelled in similar ways according to the different cases of the delegation protocol.

Chapter 9

Evaluation

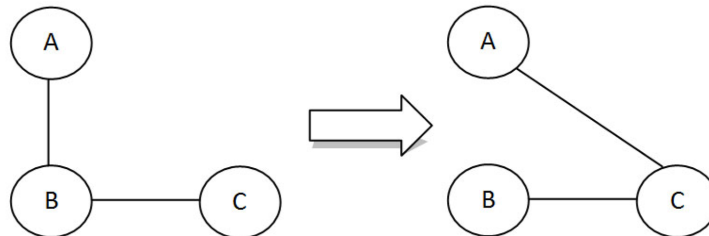
9.1 Implementation

In order to evaluate all delegation cases and also the new security features implemented, several tests were developed, one for each of the delegation case. Most of these tests were adapted from the existing ones Raymond Hu had implemented previously, so we can show in more detail how delegation works, with special incidence on security.
All evaluation tests were successfully completed.

The most important methods of these evaluation tests can be found in appendix B. The full source code can be found in [1].

9.1.1 Delegation Case 1

Case 1 is the most common type of delegation where party A is waiting for a receiving operation while B is delegating the session to C. In order to illustrate this case, we implemented all three parties that send several messages in order to trace the flow of the sessions between those peers.



The protocols for all parties are:

```
% A
final noalias protocol p_b { cbegin.?(String).!<String>.!<String>.?(String) }

% B
final noalias protocol aux { ?(String).!<String> }
final noalias protocol p_a { sbegin.!<String>.?(String).@(aux) }
final noalias protocol p_c { cbegin.!<@(aux)> }

% C
final noalias protocol p_a { ?(String).!<String> }
final noalias protocol p_b { sbegin.?(@(p_a)) }
```

We shall note that the last receive in A's protocol is what distinguishes this case from case 2. Since it is waiting for a receive operation, it has not finished is part of the session and the delegation may take place in order for A and C to communicate directly.

A receives a message from B, sends a reply to him, followed by sending another message which is going to be delegated to C. This protocol finalises with the reception of a message from C (after the session is delegated) without A being aware that the message was actually sent by C. On the other hand, C is aware of the delegation and after receiving the message from A, it sends a reply to her, terminating the protocol.

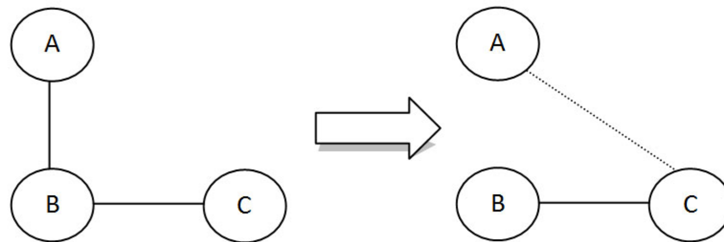
The output from those messages is the following:

```
% A
Received from B: Message from B to A!
Received from B: Message from C to A! (aware of delegation)

% B
Received from A: Message from A to B!

% C
Received from A (delegated by B): Message from A to B!
```

As mentioned before, A receives the second message from C, although she thinks she is connected to B. Nevertheless, C knows she is sending the message to A. This is, as stated in previous chapters, one of the most important features of session delegation — being transparent to the passive party.

Regarding security, A needs to login to B to establish the initial session, whereas B needs to login to C. Then, internally, B generates a new fresh credential for the session being delegated and A connects to C directly without the need for the user to input any login details. These details are always checked with the SRP protocol before establishing a new TLS connection, achieving all security properties explained in chapter 4 and section 4.3.

9.1.2 Delegation Case 2



This delegation case is the simplest, since A has already finished her part of the session and his waiting for an acknowledgment. What distinguishes this case from the previous case is that after A sends the information to B, she cannot wait for any message in reply. Hence the protocol is:

```
final noalias protocol p_b { cbegin.?(String).!<String>.!<String> }
```

The first receive is just for presenting a message from B, the second one is the sending of a message to B, and the third one is what is going to be delegated to C. Since after the delegation there are no receives, we can assure we are in presence of case 2 of the delegation.

On the other hand, the protocols of both B and C are the following:

```
% B
final noalias protocol aux { ?(String) }
final noalias protocol p_a { sbegin.!<String>.?(String).@(aux) }
final noalias protocol p_c { cbegin.!<@(aux)> }
```

```
% C
final noalias protocol p_a { ?(String) }
final noalias protocol p_b { sbegin.?(@(p_a)) }
```

B starts by sending a message to A, followed by receiving a message from the same peer and delegating the next message to C. Then, C is just entitled to receive the session with A delegated by B and receive a new message from A, which A sent thinking it was for B.

The resulting output is shown below:

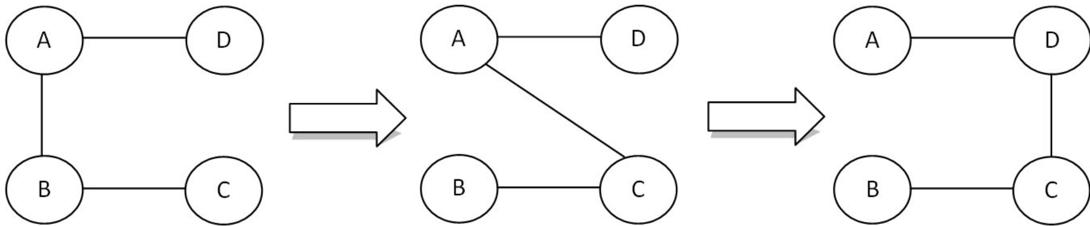
```
% A
Received from B: Message from B to A!

% B
Received from A: Message from A to B!

% C
Received from A (delegated by B): Message from A to B!
```

We can easily see that A sends a message to C, which she thought it was for B. Apart from that, this example is trivial and it was just implemented to show the difference between cases 1 and 2.

Regarding security it is exactly the same as case 1.

9.1.3 Delegation Case 3

For testing case 3 of delegation, all four parties were implemented to simulate this case as accurate as possible. This case distinguishes from the double delegation in that this case only attempts to delegate simultaneously, but in fact it only triggers another delegation after the first one has occurred i.e., like two case 1 delegations occurring sequentially.



The protocols for this case are the following:

```
% A
final noalias protocol aux { ?(String).!<String> }
final noalias protocol p_d { cbegin.@(aux) }
final noalias protocol p_b { cbegin.?(String).!<String>.?(String).!<@(aux)> }

% B
final noalias protocol aux { ?(String).!<String>.?(?(String).!<String>) }
final noalias protocol p_a { sbegin.!<String>.@(aux) }
final noalias protocol p_c { cbegin.!<@(aux)>.?(String) }

% C
final noalias protocol p_d { ?(String).!<String> }
final noalias protocol p_a { ?(String).!<String>.?(@(p_d)) }
final noalias protocol p_b { sbegin.?(@(p_a)).!<String> }
```

```
% D
final noalias protocol p_a { sbegin.!<String>.?(String) }
```

A connects to B and D, hence they need to have dual protocols. The same thing applies to C with B and all intermediate connection protocols. A starts by delegating its session with B to D, followed by B delegating to C its session with A. Then C receives both A and D, being D unaware of anything.

The output for tracing the delegations was the following:

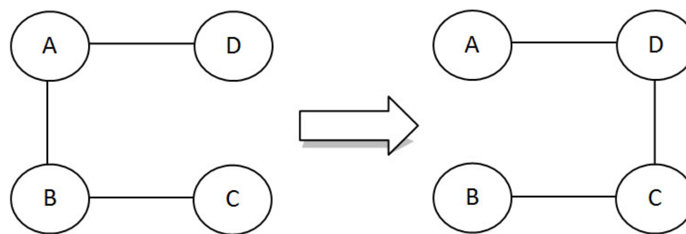
```
% A
Received from B: Message from B to A!
Received from B: Message from C to A! (aware of delegation)

% B
Received from C: Message from C to B!

% C
Received from A (delegated by B): Message from A to B!
Received from D (delegated by A) (second delegation): Message from D to A!

% D
Received from A: Message from C to D! (aware of delegation)
```

Analysing this output we can see that A thinks she is connected to B but instead she receives a message from C. The same thing happens in D where he receives a message from C thinking he is connected to A. However, the sender of both these messages was aware of the delegation.

Regarding security, in the beginning of the connection it asks for user name and password in order to connect to the other parties. Then these details are checked via the SRP protocol and a TLS connection is created amongst the peers.

9.1.4 Delegation Case 4

In order to test the most complex delegation case of SJ — simultaneous double delegation — one main test was implemented. The test consists of the four parties of the delegation protocol.



The protocols of the party A are the following:

```
final noalias protocol aux { ?(String).!<String>.?(String) }
final noalias protocol p_b { cbegin.!<String>.@(aux) }
final noalias protocol p_d { cbegin.!<@(aux)>.?(String) }
```

These protocols interact with the ones from class B (among others):

```
final noalias protocol aux { ?(String).!<String> }
final noalias protocol p_a { sbegin.?(String).!<String>.@(aux) }
```

```
final noalias protocol p_c { cbegin.?(String).!<@(aux)> }
```

Class C is composed by the following protocols:

```
final noalias protocol p_d { ?(String).!<String> }
final noalias protocol p_b { sbegin.!<String>.?(@(p_d)) }
```

Finally, peer D has the protocols:

```
final noalias protocol p_c { ?(String).!<String>.?(String) }
final noalias protocol p_a { sbegin.?(@(p_c)).!<String> }
```

As we can see, the protocol *p_b* in class A is dual to *p_a* in class B; the protocol *p_d* (A) is dual to *p_a* (D); *p_c* (B) to *p_b* (C) and *p_d* (C) to *p_c* (D). This has to be the case in order for all the parties to interact.

Then, A delegates its session with B to D and B delegates its session with A to C (performing the double delegation), followed by the sending of several Strings in order to show the flow of the messages. The results were the following:

```
% A
Received from D: Message from D to A!

% B
Received from A: Message from A to B!
Received from C: Message from C to B!

% C
Received from D (delegated by B): Message from D to C! (aware of delegation)

% D
Received from C (delegated by A): Message from B to A!
Received from C (delegated by A): Message from C to D! (aware of delegation)
```

It is easily shown that the first message received by D was sent by someone unaware of the delegation, which shows one important feature of the delegation: being transparent to the passive party. The other two messages received by C and D were sent by each other, being aware of the delegation since they were the session receivers.

Regarding security, the login (username and password) is asked in the beginning of the connection, being this asked twice in party A, since there is a connection to both B and D. This could be done with one single login, but in order to simulate real life applications, one user may have different login details in different servers and that is why we opted to ask once for each connection.

9.1.5 Running Test classes

In order to execute these test programs, one should input the following instructions:

- **D**, with arguments: *d d <portD>*
- **C**, with arguments: *d d <portC>*
- **B**, with arguments: *d d <hostC> <portC> <portB>*
- **A**, with arguments: *d d <hostB> <portB> [<hostD> <portD>]*

Obviously party D is only executed in cases 3 and 4 and the last two arguments from instruction A should be removed if we are executing either of the other two cases.

Like in the Online Shopping example, in order to run these testing programs with the secure protocol we should change 'd d' by 's s'.

9.2 Protocols Correctness

The evaluation of the theoretical part of this project concerns the protocols correctness that was successfully proved in section 8.4.2.

We started by modelling all delegation protocols into π-calculus and formalised every property regarding their correctness. The properties are as following: linearity, liveness, session consistency and session security. This last property was specifically created for this new implementation of SJ and it was also proved along with all the other properties.

Since everything was successfully proved, we ensure the theoretical part of the implementation also succeeded in the correctness evaluation.

9.3 Session Java

Programming in SJ is a rather simple process since it is an extension of Java. However, seeing that it is still under development, there are some features that are not yet implemented, making it sightly harder for SJ's programs to work at first. One thing that could also be implemented is some type of auxiliary program that could help debugging SJ's programs since debugging in distributed programs is quite difficult. Nonetheless, it is a great parallel and distributed programming language, which is hoped to become known in industry in the future, specially now that secure features were developed. This programming language was compared with other parallel and distributed languages and achieved some remarkable results [13].

Regarding the modifications in Session Java runtime, the entire code was easily understandable, modular and extensible, which made the task easier to perform. Furthermore, the code is very well documented which was also very helpful.

After implementing the new Secure SJ, the code also remained modular and extensible, since the modifications consisted mostly on adding some methods (and classes) and restructuring parts of the main code. The new code was also well documented, so that anyone can use this code in order to add further extensions on the language.

To conclude, programming in Secure SJ is relatively easy and for anyone wanting to extend this language's runtime, the task will also be easy. Secure SJ has a great potential for becoming an important distributed programming language as it is rapidly growing in all its new versions.

Chapter 10

Conclusion and Further Work

Before this project was started, SJ was vulnerable to the network or any other attacker. We started by researching the best possible way of solving this issue by studying several different protocols and by trying to combine them in order to achieve proper security for SJ.

After that was implemented, we formalised all four cases of the delegation protocol in both Forwarding and Resending modes and decided how security could be added on those delegation protocols. Once this was decided, we implemented everything and then performed all sorts of testing. When we were satisfied with the results, we moved into a more theoretical approach.

In order to prove the correctness of the protocols, we modelled all delegation cases into π-calculus. Then we type-checked those protocols and formalised all correctness properties in a mathematical notation, to allow us to better prove their correctness. It was just left to prove their correctness, which we successfully did.

To prove SJ can be used for more important applications now that it includes security features, we created several examples, one of which with a confidential credit card information. We then executed that example with and without the security protocols activated in order to show their importance and found out that the credit card information could be easily eavesdropped and modified in the unsecure execution of the program.

We hope this project is a pioneer in security improvements in the future versions of SJ, i.e. that in future versions security issues will be taken more seriously into account. We also hope that Secure SJ can be more trustily used in industry and not just for academic purposes.

Further Work

Regarding further work, as it was mentioned before, the TLS implementation should be changed in order to include the SRP protocol in the cipher suite list. This way, an additional key exchange could be avoided and the SRP would be part of the key exchange phase of the TLS protocol. Currently, the best way of doing so, is by implementing the server side of the Bouncy Castle as it was discussed in section 6.1. Moreover, credentials should be improved so as to stop being randomly generated and should contain an expiry timestamp in order to revoke old credentials. This way, the username could be something like the process ID with the time it was generated. There should also be more research concerning the need of limiting the re-delegation by depth like related works currently do. At the time, it did not seem necessary but for future versions this may be something to add in case it becomes a requirement of the user.

Additionally, regarding the theoretical part of this project, proving the Session Consistency property was left as future work as well as modelling the delegation protocols in applied π-calculus in order to prove the security properties in a more realistic scenario.

Bibliography

[1] A Secure Session-Based Distributed Programming Language: Homepage. `http://www.doc.ic.ac.uk/~nma08/project`.

[2] AES Homepage. `http://csrc.nist.gov/encryption/aes`.

[3] Chapel homepage. `http://chapel.cs.washington.edu`.

[4] Fortress homepage. `http://projectfortress.sun.com`.

[5] Secure Socket Layer Article. `http://www.windowsecurity.com/articles/Secure_Socket_Layer.html`.

[6] Sieve of Eratosthenes. `http://www.nist.gov/dads/HTML/sieve.html`.

[7] SJ Homepage. `http://www.doc.ic.ac.uk/~rh105/sessionj.html`.

[8] WS-Trust. Web service trust language: Homepage. `https://www.ibm.com/developerworks/webservices/library/specification/ws-trust/`.

[9] Martn Abadi and Cdric Fournet. Mobile Values, New Names, and Secure Communication, 2001.

[10] Onur Acıiçmez and Çetin Kaya Koç. Trace-Driven Cache Attacks on AES. Cryptology ePrint Archive, Report 2006/138, 2006.

[11] Onur Acıiçmez, Werner Schindler, and Çetin K. Koç. Cache Based Remote Timing Attack on the AES. In *Topics in Cryptology CT-RSA 2007, The Cryptographers Track at the RSA Conference 2007*, pages 271–286. Springer-Verlag, 2007.

[12] M. Ahsant, J. Basney, and O. Mulmo. Grid delegation protocol. In *Proceedings of the Workshop on Grid Security Practice and Experience, vol. YCS-2004-380*, pages 81–91, July 2004.

[13] Nuno Alves. Session-Based Distributed Programming in Java. *Imperial College London - Independent Study Option, Term I.*

[14] Olav Bandmann, Mads Dam, and Babak Sadighi Firozabadi. Constrained delegation, 2002.

[15] Moritz Becker, Cedric Fournet, and Andrew Gordon. Design and semantics of a decentralized authorization language. In *CSF '07: Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 3–15, Washington, DC, USA, 2007. IEEE Computer Society.

[16] Andi Bejleri, Raymond Hu, and Nobuko Yoshida. Session-based Programming for Parallel Algorithms. *Proceedings of the ETAPS workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software PLACES'09.*

[17] Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, and James J. Leifer. Cryptographic Protocol Synthesis and Verification for Multiparty Sessions. *CSF 09*, 2009.

[18] C. Neuman and T. Yu and S. Hartman and K. Raeburn. The Kerberos Network Authentication Service (V5). *RFC 4120, July 2005.*

[19] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[20] D. Taylor, T. Wu, N. Mavrogiannopoulos, and T. Perrin. Using the Secure Remote Password (SRP) Protocol for TLS Authentication. *RFC 5054, Nov. 2007.*

[21] DES. Data Encryption Standard. In *FIPS PUB 46, Federal Information Processing Standards Publication*, pages 46–2, 1977.

[22] John DeTreville. Binder, a logic-based security language. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 105, Washington, DC, USA, 2002. IEEE Computer Society.

[23] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session Types for Object-Oriented Languages. In Dave Thomas, editor, *ECOOP'06*, volume 4067 of *LNCS*, pages 328–352. Springer-Verlag, 2006.

[24] T. Dierks and E. Rescorla. The TLS Protocol Version 1.2. *RFC 5246, August 2008.*

[25] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[26] Jean Dollimore, Tim Kindberg, and George Coulouris. *Distributed Systems: Concepts and Design (4th Edition) (International Computer Science Series).* Addison Wesley, May 2005.

[27] E. Rescorla. HTTP Over TLS. *RFC 2818, May 2000.*

[28] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. *SIGOPS Oper. Syst. Rev.*, 40(4):177–190, 2006.

[29] David Gourley, Brian Totty, Marjorie Sayer, Sailu Reddy, and Anshu Aggarwal. *HTTP: The Definitive Guide.* 2002.

[30] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP '98: Proceedings of the 7th European Symposium on Programming.* Springer-Verlag, 1998.

[31] Raymond Hu, Nobuko Yoshida, Andi Bejleri, and Kohei Honda. The SJ Framework for Transport-Independent, Type-Safe, Object-Oriented Communications Programming. *Draft*, 2009.

[32] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-Based Distributed Programming in Java. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 516–541, Berlin, Heidelberg, 2008. Springer-Verlag.

[33] Raymond Hu, Nobuko Yoshida, and Kohei Honda. ESP: π-Calculus with Event-Enriched Session-Types. *Draft*, 2009.

[34] Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network Security: Private Communication in a Public World, Second Edition.* Prentice Hall PTR, April 2002.

[35] Ninghui Li, Benjamin Grosof, and Joan Feigenbaum. A practically implementable and tractable delegation logic, 2000.

[36] Dimitris Mostrous and Nobuko Yoshida. Session-based communication optimisation for higher-order mobile processes. In *TLCA '09: Proceedings of the 9th International Conference on Typed Lambda Calculi and Applications*, pages 203–218, Berlin, Heidelberg, 2009. Springer-Verlag.

[37] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, October 2000.

[38] R. Khare and S. Lawrence. Upgrading to TLS Within HTTP/1.1. *RFC 2817, May 2000.*

[39] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26:1484–1509, 1997.

[40] William Stallings. *Cryptography and Network Security (4th Edition)*. Prentice Hall, November 2005.

[41] Michal Wegiel and Chandra Krintz. XMem: Type-Safe, Transparent, Shared Memory for Cross-Runtime Communication and Coordination. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 327–338, New York, NY, USA, 2008. ACM.

[42] M. Weiland. Chapel, Fortress and X10: Novel Languages for HPC. Technical Report HPCxTR0706, The HPCx Consortium, 2007.

[43] Thomas Wu. The secure remote password protocol. In *In Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium*, pages 97–111, 1998.

[44] Thomas Wu. A real-world analysis of kerberos password security. In *In Symposium on Network and Distributed Systems Security (NDSS '99)*, 1999.

[45] Nobuko Yoshida and Vasco T. Vasconcelos. Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication. *Electron. Notes Theor. Comput. Sci.*, July 2007.

Appendix A

Delegation Implementation

A.1 Session sending

```
public void delegateSession(SJAbstractSocket s, SJSessionType st) throws SJIOException
    {
  if (ser.zeroCopySupported())
    ser.writeReference(s); // Or use pass (like sendChannel).
  else {
    SJSessionProtocols sp = s.getSJSessionProtocols();

    if (!(sp instanceof SJSessionProtocolsImpl))
      throw new SJIOException("[SJSessionProtocolsImpl] Incompatible session peer for
          delegation: " + sp);

    ser.writeByte(DELEGATION_START);
    ser.writeBoolean(s instanceof SJRequestingSocket); // Original requestor.

    SJCredentials cred = new SJCredentials();

    // If HTTPS, creates credentials and sends to C
    if (s.getConnection().getTransportName().equals(Constants.HTTPS)) {
      cred = generateUserPassword();
      System.out.println("Cred: " + cred.getUser() + " " + cred.getPwd() + " " + cred.
          getIndex() + " " + Arrays.toString(cred.getSalt()));
      ser.writeObject(cred);
    }

    int port = receiveInt(); // Should handle delegation case 3 (the two preceding
        delegation protocol messages are forwarded by peer).

    SJSerializer foo = sp.getSerializer();
    String hostName = ser.getConnection().getHostName();

    if (hostName.equals("localhost") || hostName.equals("127.0.0.1")) { // Factor out
        constants.
      try {
        hostName = InetAddress.getLocalHost().getHostAddress(); // Main Runtime routines
            are now using IP addresses rather than host names.
      }
      catch (UnknownHostException uhe) {
        throw new SJRuntimeException(uhe);
      }
    }

    // If HTTPS, sends credentials back to A, otherwise sends just normal information
```

```
    if (s.getConnection().getTransportName().equals(Constants.HTTPS)) {
      // B -> A: <IPc, Pc, CredA>
      foo.writeControlSignal(new SJDelegationSignal(hostName, port, cred.getUser(), cred
          .getPwd()));
      cred = null; // Deletes credential from memory to prevent memory leaks
    }
    else // B -> A: <IPc, Pc>
      foo.writeControlSignal(new SJDelegationSignal(hostName, port, null, null));

    forwardUntilACKOrFINInclusive(foo, ser);

    // Maybe need to close sp (and corresponding socket).
    s.getSerializer().close();
    SJRuntime.closeSocket(s);
  }
}
```

A.2 Session receiving

```
public SJAbstractSocket receiveSession(SJSessionType st, SJSessionParameters params)
    throws SJIOException {
  if (ser.zeroCopySupported()) {
    try {
      return (SJAbstractSocket) ser.readReference(); // Can use ordinary receive (like
          receiveChannel).
    }
    catch (SJControlSignal cs) { // May need revision to handle certain delegation
        cases.
      throw new SJIOException(cs);
    }
  }
  else {
    byte flag = receiveByte(); // Handles delegation by peer?

    if (flag != DELEGATION_START) { // Maybe replace by a proper control signal. Then
        would need to manually handle SJDelegationSignal.
      throw new SJIOException("[SJSessionProtocolsImpl] Unexpected flag: " + flag);
    }

    boolean origReq;
    SJProtocol p = null;
    LinkedList<SJMessage> bar = null;
    SJConnection conn = null;
    boolean simdel;
    SJTransportManager sjtm = null;
    SJAcceptorThreadGroup atg = null;

    try {
      try {
        if (!lostMessages.isEmpty()) {
          SJMessage m = lostMessages.remove(0);
          byte t = m.getType();

          if (t != SJ_BOOLEAN) {
            throw new SJIOException("[SJSessionProtocolsImpl] Expected boolean, not: " +
                t);
          }

          origReq = m.getBooleanValue();
```

```
    }
    else {
      origReq = ser.readBoolean();
    }
  }
  catch (SJControlSignal cs) {
    throw new SJIOException(cs);
  }

  if (s.getConnection().getTransportName().equals(Constants.HTTPS)) {
    try {
      SJCredentials cred = (SJCredentials) receive();

      PasswordFile pwf = new PasswordFile(Constants.DEFAULT_PASS);
      pwf.add(cred.getUser(), cred.getPwd(), cred.getSalt(), cred.getIndex());

      System.out.println("Received Cred: " + cred.getUser());

      cred = null; // Delete credentials from memory to prevent leaks
    } catch (ClassNotFoundException e) {
      throw new SJIOException("[SJSessionProtocolsImpl] Unable to receive
          credentials: " + e);
    } catch (IOException e) {
      throw new SJIOException("[SJSessionProtocolsImpl] Unable to read password file
          : " + e);
    }
  }
  sjtm = SJRuntime.getTransportManager();
  p = new SJProtocol(SJRuntime.getTypeEncoder().encode(st));
  atg = SJRuntime.getFreshAcceptorThreadGroup(params);

  ser.writeInt(atg.getPort()); // Sends new opened port (pc)
  bar = new LinkedList<SJMessage>();

  simdel = bufferLostMessagesUntilACKOrFIN(ser, bar); // Includes final control
      message.

  SJMessage m = bar.get(bar.size() - 1);

  // Delegation case 4
  if (simdel && origReq) {
    SJDelegationSignal ds = (SJDelegationSignal) bar.get(bar.size() - 1).getContent
        ();

    conn = sjtm.openConnection(ds.getHostName(), ds.getPort(), params, ds.
        getUserName(), ds.getPwd());
  }
  else {
    if (!(m.getType() == SJ_CONTROL && m.getContent() instanceof SJFIN)) {
      conn = atg.nextConnection(); // FIXME: if the passive peer has no compatible
          setups, we will hang.
    }
  }
  bar.remove(bar.size() - 1); // Remove the final control message (SJDelegationACK).
}
finally {
  if (sjtm != null && atg != null) {
    sjtm.closeAcceptorGroup(atg.getPort());
    SJRuntime.freePort(atg.getPort()); // Gives the session port the Runtime bound
        for us.
  }
```

```
    }

    SJAbstractSocket foo;

    if (origReq)
      foo = new SJRequestingSocket(p, SJSessionParameters.DEFAULT_PARAMETERS);
    else
      foo = new SJAcceptingSocket(p, SJSessionParameters.DEFAULT_PARAMETERS);

    SJRuntime.bindSocket(foo, conn);

    ((SJSessionProtocolsImpl) foo.getSJSessionProtocols()).lostMessages = bar;

    if (conn != null) { // Delegation case 2.
      try { // Duals the operations at the end of reconnectToDelegationTarget. But not
           sure if this should be done here.
        foo.setHostName(InetAddress.getByName(conn.getHostName()).getHostAddress());

        if (simdel && origReq) {
          foo.getSerializer().writeInt(foo.getLocalPort()); // Mirrors the actions of
               reconnectToDelegationTarget.
        }
        else {
          foo.setPort(foo.getSerializer().readInt()); // This could also be given by the
                delegator, passive party's session port shouldn't change - except for
               delegation case 4?
        }
      }
      catch (SJControlSignal cs) {
        throw new RuntimeException("[SJSessionProtocolsImpl] Shouldn't get in here: " +
             cs);
      }
      catch (UnknownHostException uhe) {
        throw new RuntimeException("[SJSessionProtocolsImpl] Shouldn't get in here: " +
             uhe);
      }
    }
    return foo;
  }
}
```

Appendix B

Delegation Tests

B.1 Case 1

B.1.1 A

```
public static void main(String[] args) throws Exception
{
  final noalias protocol p_b { cbegin.?(String).!<String>.!<String>.?(String) } // The
      last receive makes this case 1 instead of 2

  final noalias SJService c_b = SJService.create(p_b, args[2], Integer.parseInt(args
      [3]));
  final noalias SJSocket s_b;

  SJSessionParameters params = createSJSessionParameters(args[0], args[1]);

  try (s_b) {
    s_b = c_b.request(params);

    System.out.println("Received from B: " + s_b.receive());

    s_b.send("Message from A to B!");
    s_b.send("Message from A to B!"); //Delegated message.

    System.out.println("Received from B: " + s_b.receive());
  }
  finally { }
}
```

B.1.2 B

```
public static void main(String[] args) throws Exception {
  final noalias protocol aux { ?(String).!<String> }
  final noalias protocol p_a { sbegin.!<String>.?(String).@(aux) }
  final noalias protocol p_c { cbegin.!<@(aux)> }

  final noalias SJServerSocket ss;

  try (ss) {
    SJSessionParameters params = createSJSessionParameters(args[0], args[1]);
    ss = SJServerSocketImpl.create(p_a, Integer.parseInt(args[4]), params);
    final noalias SJService c_c = SJService.create(p_c, args[2], Integer.parseInt(args
        [3]));
```

```
    noalias SJSocket s_a;
    final noalias SJSocket s_c;

    try (s_a, s_c) {
      s_a = ss.accept();
      s_c = c_c.request(params);

      s_a.send("Message from B to A!");
      System.out.println("Received from A: " + s_a.receive());

      s_c.pass(s_a);
    }
    finally { }
  }
  finally { }
}
```

B.1.3 C

```
public static void main(String[] args) throws Exception {
  final noalias protocol p_a { ?(String).!<String> }
  final noalias protocol p_b { sbegin.?(@(p_a)) }

  final noalias SJServerSocket ss;

  try (ss) {
    ss = SJServerSocketImpl.create(p_b, Integer.parseInt(args[2]),
        createSJSessionParameters(args[0], args[1]));

    final noalias SJSocket s_b;
    final noalias SJSocket s_a;

    try (s_b, s_a) {
      s_b = ss.accept();

      s_a = (@(p_a)) s_b.receive(s_b.getParameters());
      System.out.println("Received from A (delegated by B): " + s_a.receive());

      s_a.send("Message from C to A! (aware of delegation)");
    }
    finally { }
  }
  finally { }
}
```

B.2 Case 2

B.2.1 A

```
public static void main(String[] args) throws Exception {
  final noalias protocol p_b { cbegin.?(String).!<String>.!<String> }

  final noalias SJService c_b = SJService.create(p_b, args[2], Integer.parseInt(args
      [3]));
  final noalias SJSocket s_b;

  SJSessionParameters params = createSJSessionParameters(args[0], args[1]);
```

```
  try (s_b) {
    s_b = c_b.request(params);

    System.out.println("Received from B: " + s_b.receive());

    s_b.send("Message from A to B!");
    s_b.send("Message from A to B!"); //Delegated message. No further receives (case 2)
  }
  finally { }
}
```

B.2.2 B

```
public static void main(String[] args) throws Exception {
  final noalias protocol aux { ?(String) }
  final noalias protocol p_a { sbegin.!<String>.?(String).@(aux) }
  final noalias protocol p_c { cbegin.!<@(aux)> }

  final noalias SJServerSocket ss;

  try (ss) {
    SJSessionParameters params = createSJSessionParameters(args[0], args[1]);

    ss = SJServerSocketImpl.create(p_a, Integer.parseInt(args[4]), params);
    final noalias SJService c_c = SJService.create(p_c, args[2], Integer.parseInt(args
        [3]));
    noalias SJSocket s_a;
    final noalias SJSocket s_c;

    try (s_a, s_c) {
      s_a = ss.accept();
      s_c = c_c.request(params);

      s_a.send("Message from B to A!");
      System.out.println("Received from A: " + s_a.receive());

      s_c.pass(s_a);
    }
    finally { }
  }
  finally { }
}
```

B.2.3 C

```
public static void main(String[] args) throws Exception {
  final noalias protocol p_a { ?(String) }
  final noalias protocol p_b { sbegin.?(@(p_a)) }

  final noalias SJServerSocket ss;

  try (ss) {
    ss = SJServerSocketImpl.create(p_b, Integer.parseInt(args[2]),
        createSJSessionParameters(args[0], args[1]));

    final noalias SJSocket s_b;
    final noalias SJSocket s_a;
```

```
    try (s_b, s_a) {
      s_b = ss.accept();

      s_a = (@(p_a)) s_b.receive(s_b.getParameters());
      System.out.println("Received from A (delegated by B): " + s_a.receive());
    }
    finally { }
  }
  finally { }
}
```

B.3 Case 3

B.3.1 A

```
public static void main(String[] args) throws Exception {
  final noalias protocol aux { ?(String).!<String> } // The send makes this session a
      case 1 instead of case 2.
  final noalias protocol p_d { cbegin.@(aux) }
  final noalias protocol p_b { cbegin.?(String).!<String>.?(String).!<@(aux)> } // The
      first send is unnecessary.

  final noalias SJService c_d = SJService.create(p_d, args[2], Integer.parseInt(args
      [3]));
  final noalias SJService c_b = SJService.create(p_b, args[4], Integer.parseInt(args
      [5]));

  noalias SJSocket s_d;
  final noalias SJSocket s_b;
  SJSessionParameters params = createSJSessionParameters(args[0], args[1]);

  try (s_b, s_d) {
    s_d = c_d.request(params);
    s_b = c_b.request(params);

    System.out.println("Received from B: " + (String) s_b.receive());
    s_b.send("Message from A to B!");
    System.out.println("Received from B: " + (String) s_b.receive());

    s_b.pass(s_d);
  }
  finally { }
}
```

B.3.2 B

```
public static void main(String[] args) throws Exception {
  final noalias protocol aux { ?(String).!<String>.?(?(String).!<String>) }
  final noalias protocol p_a { sbegin.!<String>.@(aux) }
  final noalias protocol p_c { cbegin.!<@(aux)>.?(String) }

  final noalias SJServerSocket ss;

  SJSessionParameters params = createSJSessionParameters(args[0], args[1]);

  try (ss) {
    ss = SJServerSocketImpl.create(p_a, Integer.parseInt(args[4]), params);
```

```
    final noalias SJService c_c = SJService.create(p_c, args[2], Integer.parseInt(args
        [3]));
    noalias SJSocket s_a;
    final noalias SJSocket s_c;

    try (s_a, s_c) {
      s_a = ss.accept();
      s_c = c_c.request(params);

      s_a.send("Message from B to A!");

      s_c.pass(s_a);

      System.out.println("Received from C: " + (String) s_c.receive());
    }
    finally { }
  }
  finally { }
}
```

B.3.3 C

```
public static void main(String[] args) throws Exception {
  final noalias protocol p_d { ?(String).!<String> }
  final noalias protocol p_a { ?(String).!<String>.?(@(p_d)) }
  final noalias protocol p_b { sbegin.?(@(p_a)).!<String> }

  final noalias SJServerSocket ss;

  try (ss) {
    ss = SJServerSocketImpl.create(p_b, Integer.parseInt(args[2]),
        createSJSessionParameters(args[0], args[1]));

    final noalias SJSocket s_b;
    final noalias SJSocket s_a;
    final noalias SJSocket s_d;

    try (s_b, s_a, s_d) {
      s_b = ss.accept();

      s_a = (@(p_a)) s_b.receive(s_b.getParameters());
      System.out.println("Received from A (delegated by B): " + (String) s_a.receive());
      s_a.send("Message from C to A! (aware of delegation)");

      s_d = (@(p_d)) s_a.receive(s_b.getParameters());
      System.out.println("Received from D (delegated by A) (second delegation): " + (
          String) s_d.receive());
      s_d.send("Message from C to D! (aware of delegation)");

      s_b.send("Message from C to B!");
    }
    finally { }
  }
  finally { }
}
```

B.3.4 D

```
public static void main(String[] args) throws Exception {
  final noalias protocol p_a { sbegin.!<String>.?(String) }

  final noalias SJServerSocket ss;

  try (ss) {
    ss = SJServerSocketImpl.create(p_a, Integer.parseInt(args[2]),
        createSJSessionParameters(args[0], args[1]));
    final noalias SJSocket s_a;

    try (s_a) {
      s_a = ss.accept();

      s_a.send("Message from D to A!");
      System.out.println("Received from A: " + (String) s_a.receive());
    }
    finally { }
  }
  finally { }
}
```

B.4 Case 4

B.4.1 A

```
public static void main(String[] args) throws Exception {
  final noalias protocol aux { ?(String).!<String>.?(String) }
  final noalias protocol p_b { cbegin.!<String>.@(aux) }
  final noalias protocol p_d { cbegin.!<@(aux)>.?(String) }

  final noalias SJService c_d = SJService.create(p_d, args[2], Integer.parseInt(args
      [3]));
  final noalias SJService c_b = SJService.create(p_b, args[4], Integer.parseInt(args
      [5]));

  final noalias SJSocket s_d;
  noalias SJSocket s_b;
  SJSessionParameters params = createSJSessionParameters(args[0], args[1]);

  try (s_d, s_b) {
    s_d = c_d.request(params);
    s_b = c_b.request(params);

    s_b.send("Message from A to B!");

    s_d.pass(s_b);
    System.out.println("Received from D: " + (String) s_d.receive());
  }
  finally { }
}
```

B.4.2 B

```
public static void main(String[] args) throws Exception {
  final noalias protocol aux { ?(String).!<String> }
```

```
  final noalias protocol p_a { sbegin.?(String).!<String>.@(aux) }
  final noalias protocol p_c { cbegin.?(String).!<@(aux)> }

  final noalias SJServerSocket ss;

  try (ss) {
    SJSessionParameters params = createSJSessionParameters(args[0], args[1]);

    ss = SJServerSocketImpl.create(p_a, Integer.parseInt(args[4]), params);
    final noalias SJService c_c = SJService.create(p_c, args[2], Integer.parseInt(args
        [3]));
    final noalias SJSocket s_c;
    noalias SJSocket s_a;

    try (s_c, s_a) {
      s_c = c_c.request(params);
      s_a = ss.accept();

      System.out.println("Received from A: " + (String) s_a.receive());
      s_a.send("Message from B to A!");

      System.out.println("Received from C: " + (String) s_c.receive());
      s_c.pass(s_a);
    }
    finally { }
  }
  finally { }
}
```

B.4.3 C

```
public static void main(String[] args) throws Exception {
  final noalias protocol p_d { ?(String).!<String> }
  final noalias protocol p_b { sbegin.!<String>.?(@(p_d)) }

  final noalias SJServerSocket ss;

  try (ss) {
    ss = SJServerSocketImpl.create(p_b, Integer.parseInt(args[2]),
        createSJSessionParameters(args[0], args[1]));
    final noalias SJSocket s_b;
    final noalias SJSocket s_d;

    try (s_b, s_d) {
      s_b = ss.accept();

      s_b.send("Message from C to B!");

      s_d = (@(p_d)) s_b.receive(s_b.getParameters()); // Important to specify
          parameters, otherwise uses default
      System.out.println("Received from D (delegated by B): " + (String) s_d.receive());
      s_d.send("Message from C to D! (aware of delegation)");
    }
    finally { }
  }
  finally { }
}
```

B.4.4 D

```
public static void main(String[] args) throws Exception {
  final noalias protocol p_c { ?(String).!<String>.?(String) }
  final noalias protocol p_a { sbegin.?(@(p_c)).!<String> }

  final noalias SJServerSocket ss;

  try (ss) {
    ss = SJServerSocketImpl.create(p_a, Integer.parseInt(args[2]),
        createSJSessionParameters(args[0], args[1]));
    final noalias SJSocket s_a;
    final noalias SJSocket s_c;

    try (s_a, s_c) {
      s_a = ss.accept();

      s_c = (@(p_c)) s_a.receive(s_a.getParameters()); // Important to specify
          parameters, otherwise uses default
      s_a.send("Message from D to A!");

      System.out.println("Received from C (delegated by A): " + (String) s_c.receive());
      s_c.send("Message from D to C! (aware of delegation)");
      System.out.println("Received from C (delegated by A): " + (String) s_c.receive());
    }
    finally { }
  }
  finally { }
}
```

Appendix C

π-Calculus

C.1 Reduction rules

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \text{ R-Eval}$$

$$\frac{}{E[\mathtt{not}(\text{tt})] \longrightarrow E[\text{ff}]} \text{ R-Not}$$

$$\frac{s \text{ fresh in } P, Q}{a\,(x\colon S).P \mid \overline{a}\,(x\colon S').Q \longrightarrow (\boldsymbol{\nu}\, s)\ (P\{s/x\} \mid Q\{\overline{s}/x\} \mid s\,[S]\colon \varepsilon \mid \overline{s}\,[S']\colon \varepsilon)} \text{ R-Init}$$

$$\frac{}{s\,!\langle v\rangle;\ P \mid s\,[!\langle T\rangle; S]\colon \vec{h} \mid \overline{s}\,[S']\colon \vec{h'} \longrightarrow P \mid s\,[S]\colon \vec{h} \mid \overline{s}\,[S']\colon \vec{h'}\cdot v} \text{ R-Send}$$

$$\frac{}{s\,?(x);\ P \mid s\,[?(T); S]\colon v\cdot\vec{h} \longrightarrow P\{v/x\} \mid s\,[S]\colon \vec{h}} \text{ R-Receive}$$

$$\frac{1 \le i \le n}{s \triangleleft l_i;\ P \mid s\,[\oplus\{l_1\colon S_1, ..., l_n\colon S_n\}]\colon \vec{h} \mid \overline{s}\,[S']\colon \vec{h'} \longrightarrow P_i \mid s\,[S_i]\colon \vec{h} \mid \overline{s}\,[S']\colon \vec{h'}\cdot l_i} \text{ R-Select}$$

$$\frac{1 \le i \le \mathtt{min}(m, n)}{s \triangleright \{l_1\colon P_1, ..., l_m\colon P_m\} \mid s\,[\&\{l_1\colon S_1, ..., l_n\colon S_n\}]\colon l_i\cdot\vec{h} \longrightarrow P_i \mid s\,[S_i]\colon \vec{h}} \text{ R-Branch}$$

$$\frac{}{\mathtt{if}\ \text{tt}\ \mathtt{then}\ P\ \mathtt{else}\ Q \longrightarrow P} \text{ R-If-true}$$

$$\frac{}{\mathtt{if}\ \text{ff}\ \mathtt{then}\ P\ \mathtt{else}\ Q \longrightarrow Q} \text{ R-If-false}$$

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \text{ R-Par}$$

$$\frac{P \longrightarrow P'}{(\boldsymbol{\nu}\, a\colon \langle S\rangle)\ P \longrightarrow (\boldsymbol{\nu}\, a\colon \langle S\rangle)\ P'} \text{ R-Restr-a}$$

$$\frac{P \longrightarrow P'}{(\boldsymbol{\nu}\, s)\ P \longrightarrow (\boldsymbol{\nu}\, s)\ P'} \text{ Restr-s}$$

$$\frac{X(\tilde{x}) = P \in D}{\texttt{def}\ D\ \texttt{in}\ (X\langle\tilde{v}\rangle \mid Q) \longrightarrow \texttt{def}\ D\ \texttt{in}\ P\{\tilde{v}/\tilde{x}\} \mid Q}\ \text{R-Instance}$$

$$\frac{P \longrightarrow P'}{\texttt{def}\ D\ \texttt{in}\ P \longrightarrow \texttt{def}\ D\ \texttt{in}\ P'}\ \text{R-Def-scope}$$

$$\frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q}\ \text{R-Struct}$$

C.2 Typing rules

$$\frac{\Gamma \vdash e \triangleright U' \quad U' \leq U}{\Gamma \vdash e \triangleright U}\ \text{T-Sub-u}$$

$$\frac{}{\Gamma \cdot x \colon U \vdash x \triangleright U}\ \text{T-Var-u}$$

$$\frac{}{\Gamma \cdot u \colon \langle S\rangle \vdash u \triangleright \langle S\rangle}\ \text{T-Shared-chan}$$

$$\frac{}{\Gamma \vdash \text{tt} \triangleright \mathsf{bool}}\ \text{T-Bool-true}$$

$$\frac{}{\Gamma \vdash \text{ff} \triangleright \mathsf{bool}}\ \text{T-Bool-false}$$

$$\frac{\Gamma \vdash e \triangleright \mathsf{bool}}{\Gamma \vdash \texttt{not}(e) \triangleright \mathsf{bool}}\ \text{T-Not}$$

$$\frac{\Theta; \Gamma \vdash P \triangleright \Sigma' \quad \Sigma' \leq \Sigma}{\Theta; \Gamma \vdash P \triangleright \Sigma}\ \text{T-Sub-s}$$

$$\frac{\Gamma \vdash u \triangleright \langle S\rangle \quad \Theta; \Gamma \vdash P \triangleright \Sigma \cdot x \colon S}{\Theta; \Gamma \vdash u\,(x \colon S).P \triangleright \Sigma}\ \text{T-Accept}$$

$$\frac{\Gamma \vdash u \triangleright \langle S\rangle \quad \Theta; \Gamma \vdash P \triangleright \Sigma \cdot x \colon \overline{S}}{\Theta; \Gamma \vdash \overline{u}\,(x \colon \overline{S}).P \triangleright \Sigma}\ \text{T-Request}$$

$$\frac{\Gamma \vdash e \triangleright U \quad \Theta; \Gamma \vdash P \triangleright \Sigma \cdot k \colon S}{\Theta; \Gamma \vdash k\,!\langle e\rangle;\ P \triangleright \Sigma \cdot k \colon !\langle U\rangle; S}\ \text{T-Send-u}$$

$$\frac{\Theta; \Gamma \cdot x \colon U \vdash P \triangleright \Sigma \cdot k \colon S}{\Theta; \Gamma \vdash k\,?(x);\ P \triangleright \Sigma \cdot k \colon ?(U); S}\ \text{T-Receive-u}$$

$$\frac{S \neq \texttt{end} \quad \Theta; \Gamma \vdash P \triangleright \Sigma \cdot k \colon S'}{\Theta; \Gamma \vdash k\,!\langle k'\rangle;\ P \triangleright \Sigma \cdot k \colon !\langle S\rangle; S' \cdot k' \colon S}\ \text{T-Send-s}$$

$$\frac{\Theta; \Gamma \vdash P \triangleright \Sigma \cdot k \colon S' \cdot x \colon S}{\Theta; \Gamma \vdash k\,?(x);\ P \triangleright \Sigma \cdot k \colon ?(S); S'}\ \text{T-Receive-s}$$

$$\frac{1 \leq i \leq n \quad \Theta; \Gamma \vdash P \triangleright \Sigma \cdot k \colon S_i}{\Theta; \Gamma \vdash k \triangleleft l_i;\ P \triangleright \Sigma \cdot k \colon \oplus\{l_1 \colon S_1, ..., l_n \colon S_n\}} \text{ T-Select}$$

$$\frac{\forall i.1 \leq i \leq n \quad \Theta; \Gamma \vdash P_i \triangleright \Sigma \cdot k \colon S_i}{\Theta; \Gamma \vdash k \triangleright \{l_1 \colon P_1, ..., l_n \colon P_n\} \triangleright \Sigma \cdot k \colon \&\{l_1 \colon S_1, ..., l_n \colon S_n\}} \text{ T-Branch}$$

$$\frac{\Gamma \vdash e \triangleright \mathsf{bool} \quad \Theta; \Gamma \vdash P \triangleright \Sigma \quad \Theta; \Gamma \vdash Q \triangleright \Sigma}{\Theta; \Gamma \vdash \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q \triangleright \Sigma} \text{ T-If}$$

$$\frac{\Theta; \Gamma \vdash P \triangleright \Sigma \quad \Theta; \Gamma \vdash Q \triangleright \Sigma'}{\Theta; \Gamma \vdash P \mid Q \triangleright \Sigma \cdot \Sigma'} \text{ T-Par}$$

$$\frac{\Theta; \Gamma \cdot u \colon \langle S \rangle \vdash P \triangleright \Sigma}{\Theta; \Gamma \vdash (\boldsymbol{\nu}\, u \colon \langle S \rangle)\ P \triangleright \Sigma} \text{ T-Restr-a}$$

$$\frac{\tilde{x} = \tilde{x}_U \cdot \tilde{x}_S \quad \Theta \cdot X \colon \tilde{U} \cdot \tilde{S}; \Gamma \cdot \tilde{x}_U \colon \tilde{U} \vdash P \triangleright \tilde{x}_S \colon \tilde{S} \quad \Theta \cdot X \colon \tilde{U} \cdot \tilde{S}; \Gamma \vdash Q \triangleright \Sigma}{\Theta; \Gamma \vdash \texttt{def}\ X(\tilde{x}) = P\ \texttt{in}\ Q \triangleright \Sigma} \text{ T-Def-in}$$

$$\frac{\Sigma \text{ completed} \quad \tilde{x} = \tilde{x}_U \cdot \tilde{x}_S \quad \Gamma \vdash \tilde{x}_U \triangleright \tilde{U}}{\Theta \cdot X \colon \tilde{U} \cdot \tilde{S}; \Gamma \vdash X\langle \tilde{x} \rangle \triangleright \Sigma \cdot \tilde{x}_S \colon \tilde{S}} \text{ T-Process-var}$$

$$\frac{\Sigma \text{ completed}}{\Theta; \Gamma \vdash \mathbf{0} \triangleright \Sigma} \text{ T-Nil}$$