Authorisation and Authentication of Processes in Distributed Systems

CO42019 Honours Project

Undergraduate Project Dissertation

Submitted in partial fulfilment of the requirements of Napier University for the degree of Bachelor of Science with Honours in Networked Computing

Ewan Gunn
05013468
BSc (Hons) Network Computing

Supervisor:
Prof. William Buchanan

Second Marker:
Dr. Jose Munoz

Authorship Declaration

I, Ewan Gunn, confirm that this dissertation and the work presented in it are my own achievement.

1. Where I have consulted the published work of others this is always clearly attributed.
2. Where I have quoted from the work of others the source is always given. With the exception of such quotations this dissertation is entirely my own work.
3. I have acknowledged all main sources of help.
4. If my research follows on from previous work or is part of a larger collaborative research project I have made clear exactly what was done by others and what I have contributed myself.
5. I have read and understand the penalties associated with plagiarism.

Signed

Name: Ewan Gunn

Matric No.: 05013468

ABSTRACT

Communications over a network from a specific computer have become increasingly more suspect, with the increase of various security breaches in operating systems. This has allowed malicious programs such as worms, trojans, zombies and bots to be developed that exploit these security holes and run without the user being any wiser about the infection on their computer.

The current work in the field of anti-virus protection focuses on detecting and removing any malicious software or spyware from a computer. This is proving effective, however it is merely a way of treating the symptoms instead of the illness.

This project presents a hypothesis based on these situations, and attempts to prove the effectiveness of a protocol developed specifically to provide preventative measures to stop the spread of malicious software, based on authentication and subsequent authorisation.

Tools such as encryption, hashing, and digital certificates were investigated and marked for use in providing the protocol to prove the hypothesis, and a further investigation took place of the common principles in security in the computing paradigm such as the CIA and AAA sets of principles, which provided a specific context within which a protocol could be constructed. A discussion was made of the only protocol that was close to a solution to the hypothesis, Kerberos, along with any usefulness that that protocol might have in the situations the hypothesis is based in.

This was followed by a design of a new protocol, consisting of a methodology of protocol design used heavily in industry – that of communication analysis and finite state machines. A further proof-of-concept program was designed as well, to provide a facility to test the effectiveness and efficiency of the protocol. In all design considerations, the evaluation of such a system was a priority, and steps were taken at the design stage to provide an easy method to collect data results.

The system was implemented in a proof-of-concept program using an open-source alternative to the .NET framework developed by Microsoft, called mono. This development environment is cross platform and fully compliant with all versions of .NET provided by Microsoft, thereby providing a cross-platform solution to the problem described above. Specific concerns faced in implementation of such a protocol were raised, and measures taken to overcome these concerns presented, along with decisions made on options available in the implementation.

An analysis was made of the efficiency of the resulting system, by taking measurements of the time taken between request conception and the subsequent

request completion. Baseline measures were made on this using a simple client/server program developed during implementation that had the option of using the system or not, with the option not to use the system. These were compared to measurements made of the same system, however with the option to use the authorisation service enabled. A conclusion and discussion of the surprising results followed.

Lastly a critque of the project is made, along with a discussion of a theoretical situation where this system might prove beneficial; a general discussion on the benefits of promoting preventative measures for malicious software spread and any further work that could be carried out specifically on the ideas and work presented in this project.

Table of Contents

Table of Figures

ACKNOWLEDGEMENTS

I would like to thank Professor William Buchanan, for his guidance and support throughout this project. In addition, I would like to thank Dr. Jose Munoz for being part of the marking process.

1. Introduction

1.1 Project Overview

Worms, Trojans and other non-authorised network traffic have always been a concern for system administrators world wide (McDowell, "Now That We Are So Well-Educated...", 2006), and this has been highlighted even more with the new 'targeted Trojan' attack (Aaronson, "For the Love of Money..." 2005). The propagation of this self-distributing malware can cripple any network it infects in a very short space of time. Whilst the majority of this malware can be detected using common virus protection software, it would be far more beneficial to halt the distribution of these programs before they reached the stage of infecting another computer.

One of the key tenets of security is the idea of authentication, that is the process of confirming that the claims made by a principle are true and correct (Lampson *et al,* "Authentication in Distributed Systems", 1992), (Waldo, "A Formal Semantics for the Logic of Authentication"), and (Woo and Lam, "Authentication for Distributed Systems", 1992). It is this idea that is at the core of containing self-propagating code – only code that can be authenticated and subsequently authorised should be allowed to run.

1.2 Background

At present, authentication and authorisation of network traffic depends on the user, not on the process running. This allows virtually any program to access the network and internet, including self propogating code and malicious software, as long as the user the process is running under is allowed access to the network device.

This can be represented by the formal authentication language SVO (see *Authentication Semantics* in the Literature Review, below). The current way of thinking, which is evident even through the formal language itself, is that

$$C \rightarrow U,$$

read as "C speaks for U", i.e. that for any communication channel C, a user U has authorised that communication channel to speak on its behalf. This means that any messages coming across C are automatically assumed to be from U.

This precludes any malicious software which is running under the user U, although without the knowledge of that user. A more complete representation should be

$$C \rightarrow P \rightarrow U,$$

namely "C speaks for P speaks for U" for any process P. This then requires any process P to authenticate itself to U, and U can then explicitly give authorisation to that process to speak on its behalf over C.

The hypothesis is then

> "*can an efficient system be created for authenticating and authorising processes for access to the communication channel that will not interfere in the normal workings of the communication*",

the communication channel in this case being a network device. The aim of this project is to investigate this hypothesis, and test any implementation of the resulting investigative work.

1.3 Aims and Objectives

The following aims were defined to measure the effectiveness of the project.

1. Design a protocol for process authorisation and authentication
2. Design a proof-of-concept implementation for the protocol
3. Implement the proof-of-concept design
4. Implement a testing methodology to test the protocol and the proof-of-concept system
5. Evaluate the system from the results of the testing methodology

To carry out these aims, further objectives were set.

1. Investigate current security principles for guidelines on authentication and authorisation
2. Investigate any current work in the area of authentication and authorisation

1.4 Thesis Structure

Chapter 1	**Introduction**. Defining aims and objectives, along with background information
Chapter 2	**Theory**. Commonly used technology and terminology beneficial in understanding the context of the project
Chapter 3	**Literature Review**. Investigation into current work in the areas

	defined in the Introduction
Chapter 4	**Design**. Outlines of the design of the protocol, and the proof-of-concept implementation thereof
Chapter 5	**Implementation**. A discussion of the implementation, any challenges faced, and the overcoming of those challenges
Chapter 6	**Evaluation**. The testing of the protocol and the proof-of-design implementation, including the methodology behind the testing and the results
Chapter 7	**Conclusions**. A critical analysis of the project, a discussion on further work, and a description of the ideal scenario where the development of the project would be beneficial
Chapter 8	**References**. Documentation of all resources used in this project.
Chapter 9	**Appendices**. Including source code for all classes involved, screenshots of the protocol testing, and the collated results of the testing of the proof-of-concept implementation

2. Theory

2.1 Introduction

Authentication and authorisation have many established tools to aid in their implementation, due in part to the increase in awareness of the need for these principles in computing security. This chapter investigates a number of these tools, focusing mainly on encryption and the logical extension of that: digital signing and digital certificates. Other tools involved in large scale communication systems are investigated as well, concentrating on threading and the terminology involved in differentiating a process from a program.

2.2 Process or Program?

It is important to make a distinction between a process and a program, as it is relevant in the following discussion. A program is an executable file that contains a set of instructions that are carried out by a processor. A process is a program under execution in a particular state, where the state is determined by the attributes and variables declared in the running of that program (Tully, “Software Process Models and Programs”, 1989).

A good analogy is between that of a class and an object in software engineering. A class is the framework, or blueprint, for an object, where the attributes and methods are defined that the eventual object can manipulate. The object itself is an instantiated class with a certain set of values in the attributes. The object exists only as long as the program it is declared in is in state of execution, much like a process only existing as long as it is being executed.

2.3 Threading

Threading is a procedure whereby one process spawns multiple concurrent operations (called *forking*) which operate as if individual processes, however share the run-time environment of the parent process (Bennet, T, “A Thread Implementation Project Supporting and Operating Systems Course”, 2007).

Threading is a useful concept in todays computing environment, as it allows one process to emulate the operation of multiple processes.

2.4 Message Delivery

There are a number of semantics that can be used for implementing message delivery in a protocol (Ravindran, “Structural Complexity and Execution Efficiency of Distributed Application Protocols”, 1993). First is the 'at-least-once' semantics, where the protocol guarantees that the message is delivered at least one time. This semantic is used primarily in communications where any duplicate work carried out due to more than one message arriving has no effect on the state of the system (for example in online banking – if there is an option to have another copy of your latest statement delivered to your house, there is no effect on the system if the message is delivered twice, and you receive two copies of that statement).

Secondly there is at-most-once semantics. This is used where duplicate work does affect the system, and so must be minimized. Using the online-banking example, above, if you wanted to transfer money from one account to another, you would use at-most-once semantics. This would guarantee that the operation (moving money from one account to another) happened at most once. If it doesn't happen at all, then there is no big loss – the entire transaction may be repeated. If it happens twice, however, then there you transfer money twice, which would not be wanted.

Thirdly, there is exactly-once semantics, where the system ensures that any message sent is delivery one time, and one time only. This is important in critical systems, such as a nuclear power station – the message to drop the control rods into the reaction chamber by a specific amount must only be done by that specific amount. Using at-least-once semantics means that the rods could be dropped by multiple values, and using at-most-once semantics means that it may not be dropped at all.

Laslty, there is unimportant message delivery, where the delivery status is of no importance. This is like streaming media via UDP – if a packet of data is lost on the way, it is of no importance, because the loss of that information will not have a great effect on the outcome.

2.5 Encryption

There are two methods of encryption that are important in computing: symmetric and asymmetric cryptography. The differentiation between the two is important when discussing certificates and authentication.

In both cases, the general idea is thus: a sender A (commonly called Alice) wishes to send a message to the recipient, B (commonly called Bob) without any eavesdroppers understanding what that message is (the eavesdropper is referred to as Eve, and represents the entire easvesdropping community).

The process involves two main parts – an algorithm for conducting the encryption, and a key to make sure that the encryption is unique. A plaintext message has the algorithm applied to it, using the key as a parameter. The result is a ciphertext message.

This ciphertext message then has the reverse algorithm applied, along with a key, to return the plaintext message. Thus during transit the message is garbled, and cannot be understood without knowing the algorithm and they key used in that algorithm.

2.5.1 Symmetric Cryptography

In symmetric cryptography, only one key exists to encrypt and decrypt the messages. As an example, using a Ceaser cipher (which is the simplest shifted alphabet cipher) two alphabets exist in order, however one is 'shifted' a number of places left or right. For example:

A B C D E F ...

D E F G H I ...

represents a Ceaser cipher with a key of '3 places', and the algorithm is shift the second alphabet a number of places to the right.

To decrypt any message enciphered with this procedure, the same key must be used ('3 places') and the reverse algorithm applied (shift to the left). Thus we get:

D E F G H I ...

A B C D E F ...

returning the ciphertext to the original cleartext.

The main problem with symmetric cryptography is the need to have a secure channel to transmit the key through. If this secure channel is compromised, then the key is discovered by Eve, and she can then decrypt any messages sent by Alice and Bob.

2.5.2 Asymmetric Cryptography

Asymmetric cryptography is a relatively recent discovery, as it relies on the high mathemtical power that computers can provide to implement complicated algorithms.

The defining difference between symmetric and asymmetric cryptography is the use of multiple keys. Instead of having one key to encrypt and decrypt, there are two keys – one to encrypt, and one to decrypt.

Asymmetric cryptography came about as a response to the need to transmit the key in a secure channel, by removing that need. Of the two keys generated, only one can decihper and message encrypted with the other, and vice versa. It is not possible to decrypt a message with the same key it was encrypted with. This is best explained in an example.

Alice wishes again to send a message to Bob. She has two keys, called her public and private keys. Bob also has two keys, is public and private keys. Alice's public key is the only key that decrypt any message encrypted with her private key, and her private key is the only key that can decrypt messages encrypted with her public key. The same is true of Bob.

The hint is in the name of each key – public and private. Alice and Bob can publish their public keys as much as they like, as any message that is destined for them can only be decrypted with their private key (which they keep hidden).

So to send a message to Bob, Alice encrypts it with Bobs public key. Eve knows Bobs public key, but this is of no use to her, as she would need Bobs private key to decrypt. When Bob receives the message from Alice, he can then decrypt it with his private key.

The only problem with this procedure is the time taken for the algorithms to conduct the encryption and decryption. On average, symmetric cryptopgraphy is around 5000 times faster than asymmetric cryptography.

2.6 Certificates

An added benefit of the public key encryption system was that of digital signing – being able to guarantee who sent the message, and the message integrity, using a combination of asymmetric key cryptography, and hashing (see below).

This, combined with a need for a directory of public keys (similar in concept to a telephone directory) lead to the development of a number of *public key infrastructures* being developed. The core idea behind a public key infrastructure is to allow people to confirm that a public key they have (or a document signed by the equivalent private key) does indeed belong to (or originate from) the person named as the signatory. The natural progression of this was towards an *identity certificate* (sometimes called a *public key certificate*) that contains the information of the named person the certificate will belong to, combined with a signature from a trusted third party to confirm the identity of the certificate holder.

This certificate can then be passed to any relevant parties, who can then confirm the validity of the certificate by verifying it with the trusted third party. (This is usually done by pre-installing the *root certificates* of these trusted third parties onto new operating systems, or built in to browsers. Root certificates are self-signed, relying on their reputation to confirm their trustworthiness).

2.6.1 X.509

The main certificate used today is the *X.509 certificate* which was originally developed by the predecessor to the ITU-T, the CCITT, as part of the X.500 series of directory services. As such, it follows the strict hierarchy defined in X.500, with the pinnacle of the hierarchy being the root certificates, from which all other certificates are verified. As the primary purpose of the X.509 certificate was to provide identity confirmation within the X.500 protocol, the main feature used for stating the identity of a certificate holder was a *distinguished name*, as defined in X.500. These days, the *alternative name* section is widely used, which can be defined as an email address, an IP address, or a domain name (Housley et al, “Internet X.509 Public Key Infrastructure Certificate and CRL Profile”, 2002).

The X.509 certificate system is used in the *secure sockets layer* and its descendant *transport layer security*, which define secure communications across the internet for a wide variety of uses, such as browsing or email. The fact that a user is not aware of the majority of certificates passed to it is due to the pre-installed root certificates, which automatically allow any verified certificate passed to the browser to be authenticated. A user is only really aware of this process when an incorrect or out of date certificate gets delivered to the browser, and an action is required (such as in the case of the Napier University email system – the IP address defined in the certificate does not match the IP address of the server, and some browsers ask whether to allow the certificate anyway).

2.6.2 XKMS

The XKMS system is being developed by the W3C and is looking to replace the X.509 system of certificates, as the X.509 system is merely adapted to use on the internet, as opposed to being directly developed for it.

XKMS stands for XML Key Management Specification, and utilises XML based certificates instead of plaintext certificates. This allows far wider adaptability and scalability due to the nature of XML based documents, best illustrated by the fact that an XKMS certificate can have an X.509 certificate embedded into it, for use with legacy environments (Ford et al, “XML Key Management Specification”, 2001). This is a huge benefit for transition.

2.6.3 Certificate Revocation Lists

A certificate revocation list is used to revoke the authority of specific certificates. This list is issued by a certification authority, and it is up to the system checking the certificates to check this list either periodically, or each time a request is made for a certificate. If this CRL is not checked, then a certificate may be classed as valid, when in fact it has lost its authority (Wohlmacher, “Digital Certificates”, 2000).

A certificate is added to the CRL in a few circumstances, but mainly if the private key associate with the public key held in the certificate is compromised. This is why CRLs are an important part of a public key infrastructure, as they provide a high level of trust in a certificate (if a CRL is used within a system).

Most certificate checking systems do not check these CRLs, unless it is explicitly declared in a policy document.

Certificates can also be placed on *hold* in these lists, which temporarily blocks the authenticity of the certificates and may be recinded at any time.

2.7 Hashing

Hashing is much like encryption, where data is turned into a garbled form. The difference between hashing and encryption, however, is that hashing is a *one-way* function. Once a hash value for a piece of data has been calculated, it is not possible to reconstruct the data from the hash value.

It is, however, possible for two dissparate pieces of data to have the same hash value. This is called a *collision* (Hasan *et al*, “Chisel”, 2006), and can be utilised by Eve during replay attacks where the hash value is sent along with the data. New data could be altered to provide the same hash value that the original data has, and then sent in its place in a replay or injection attack. This vulnerability can be overcome somewhat by using more recent hashing algorithms that provide a far larger number of possible hash values, thus reducing the collision space dramatically.

2.8 Which System?

There are many references throughout this document to the term 'system', with various meanings. Where ambiguities might arise, it has been explicitly explained which system is being referred to. On the whole, 'system' refers to the authorisation system under investigation, consisting of the authorisation service and the client process requesting authorisation. When discussing the implementation, the 'system' also refers to the test server.

2.9 Conclusion

This section covered the main technologies currently available in the field of security and distributed systems. This is important as it allows a solid foundation to build upon for the design of the protocol, and is necessary in understanding some of the current work involved in the field.

It can be seen that the ideal system would use an at-least-once message delivery system, to make sure that the process is authenticated and authorised at least once. Using at-most-once semantics would allow for some messages to be lost, to the detriment of the user. The result of multiple executions of the result of a message from the client to the server would give a trivial duplication of the authorisation status of the process, and so can be included.

Again in the ideal system, the service would require a certificate to be passed to it to aid the authentication of the process. This certificate would be checked against any relevant certificate revocation lists to confirm the current status of the certificate passed. A certificate could be put on a *hold* status locally within the corporation if there is a known security hole, which would allow ad hoc defence again insecure products. When a security patch is applied, the *hold* status could be rescinded.

The hash value of the program the process is run from is also a piece of information that is useful to calculate. This precludes a malicious entity replacing the program with an insecure version, as the hash values will not match. The authorisation service can then deny access to the communication channel for that process.

3. Literature Review

3.1 Introduction

The main emphasis of this project is on the authorisation and authentication of processes, with the aim of allowing (or disallowing) those processes access to the network. However, it is only when the larger picture of the entirety of computing security is illustrated that the need for such authorisation is emphasised.

From the Theory section, above, a number of current, widely used technologies have been discussed, and common terminology used. This chapter discusses the higher level principle sets used in constructing technological solutions to security in computing and corporate security policies, using the CIA and AAA techniques as examples, although references to other considerations of principles are given.

An investigation took place on any existing system that had a similar philosophy defined in the project aims, and the only close result was Kerberos, an authentication system for entities. Kerberos is explained and investigated, and opinions offered on the effeciveness of the protocol in authorisation and authentication of processes.

3.2 Methods of Security

The following two sections illustrate and explain the two most common sets of principles for computer security. There are other aspects, such as dependability (Abrams, "An Integrated Framework...", 1989), survivability (Krings et al, "Security and Survivability...", 2005), diversity (O'Donnell and Sethu, "Network Security", 2005), redundancy (Littlewood and Strigini, "Redundancy and Diversity in Security", 2004), reliability (Zhuang et al, "Reliability and Security", 2004) and non-repudiation (Taipei, "Security Protocols", 2006); however most of these are still under research, or otherwise considered in specialist cases.

3.2.1 CIA

The first method investigated is that of CIA: confidentiality, integrity, and availability. This is perhaps the most common of the two sets of security principles, and is widely referenced in both technical documents and corporate security policies. The following three paragraphs explain the theory behind each of these principles.

Confidentiality

This is the maintenance of authorised disclosure of data, i.e. only those who the information is destined for are allowed to view it. By maintaining this as a primary concern, users of the system can remain fairly certain that their secure communications have not been opened and viewed before hand (Gürses et al, "Eliciting Confidentiality Requirements in Practice", 2005). Confidentiality is usually maintained using encryption to encipher the data being transmitted, and under modern public key algorithms, confidentiality can also provide non-repudiation – if the recipient can only decrypt the data using the senders public key, then that data must have been encrypted using his private key. This of course does not preclude the fact that the private key might have been stolen, or the data otherwise being encrypted using the senders private key without his knowledge or consent. The majority of the time, however, a single shared key is created between the two parties across a channel secured by asymmetric encipherment, and then a shared key for symmetric key is created. Again, as this shared key is session-unique, it can provide non-repudiation.

Integrity

Integrity is the maintenance of authorised alteration of data – ensuring that the data has not been altered during transmission or storage (Gouda et al, "Hop Integrity in Computer Networks", 2002). The most effective method of achieving this is through message hashing – creating a fixed length *fingerprint* of a piece of data. This fingerprint can be forwarded along with the data, and checked against the result of running the same procedure against the received data. To prevent tampering of the data *and* the hash signature, one can use public key encryption again. The message is first hashed, and the resulting hash signature is encrypted with the senders private key. The encrypted signature is appended to the data itself, and the entirety is encrypted again using the recipients public key. As only the senders public key can decrypt the hash signature (which itself is only accessible via the recipients private key) which authenticates the sender, and provides security against the hash signature being altered.

Availability

Maintaining authorised access to data is referred to as availability (Siponen and Oinas-Kukkonen, "A Review of Information Security Issues and Respective Research Contributions", 2007). It is possible to have the most secure system possibly imaginable, consisting of a powered off computer existing in a welded shut safe, but that does not allow the data or services to be used as intended. The resources must be made available as and when required by authorised entities, whether that be processes or users. The concept of availability extends to the clear, original version of the resource – un-encrypted. Again, it is possible to construct a cipher algorithm that is so complicated that it cannot be cracked for hundreds of years – even with the exponential increase in computing power – but it may take years to encrypt or decrypt a piece of data. Availability has a notion of timeliness associated with it for just this reason.

3.2.2 AAA

The “triple A” guidelines for security are perhaps more important to the subject in question, as it covers both authentication and authorisation directly, along with auditing or accounting. As such, it will be discussed in much greater depth.

Authentication

There are many aspects to authentication, and it is an area under an increasing amount of research due to the rise in recent threats such as phishing. An authentication system may not use a complete system for authentication, however there are a number of key areas that should be discussed.

Content Authentication

Is the message true and correct? This can be confirmed by message hashing. Hashing is the process of creating an arbitrary output of data from an input. If two hash values are the same, there is a very high chance (though not 100%) that the two inputs were the same. This is mainly used for validation that a message's content hasn't been changed, by sending the hash along with the message, and the recipient carrying out a hash function on that message and comparing the hash received with the hash generated. This provides protection against a number of man-in-the-middle attacks, where there is a malicious third party intercepting all communications between two principles. It cannot, however, protect against other types of attack, such as replay attack, where the idea is to use exactly what was sent from one principal to another at a later time (Lampson *et al*, “Authentication in Distributed Systems”, 1992).

Origin Authentication

Did the stated origin really send this? This can be done at the initial creation of a session. One method of confirming origin authentication is by creating a separate session to the supposed origin of the communication and interrogating the principle there. Whilst this is still vulnerable to man-in-the-middle attacks, it can get round such problems as IP spoofing. Research has been conducted on larger scale origin authentication with regards to the internet (Aiello et al, “Origin Authentication in Interdomain Routing”, 1981).

Identity Authentication

Is that origin who they say they are? This is generally carried out with the use of a certification authority outside of the principles involved, such as a trusted third party certificate server. Generally this authentication takes place using multifactor authentication, which takes the form of *something you are*, *something you have*, and *something you know*. For example, access to a military establishment might require authentication through fingerprint analysis (*something you are*), a swipe card (*something you have*), and a PIN (*something you know*). The majority of times, especially with computing, only single-factor authentication is used in the form of *something you know* (a password to log on, for example), and this is referred to as weak authentication as opposed to strong authentication which uses two (or more) of these factors. For example, companies are increasingly issuing key fobs or swipe cards for users to log on to their computers, which is something the user has along with something they know in the form of a password.

There is also research being carried out on a fourth factor of authentication in the form of *someone you know* (Brainard et al, "Privacy and Authentication", 2006), which means that someone needs to confirm your identity along with any of the other factors of authentication. This has been implemented mildly in the form of certificate servers.

Authentication Semantics

There have been a number of attempts to formalise the logical process of authentication over the course of the past 20 years, starting with (Burrows et al, "A Logic of Authentication", 1989). The semantics most commonly used now are what is referred to as the SVO notation, after its authors Syverson and van Oorschot. The SVO notation unified the increasing number of logical notations for authentication that had been developed (Syverson and van Oorschot, "A Unified Cryptographic Protocol Logic", 1996). The majority of predecessors were all based around the BAN notation (Burrows et al, "A Logic of Authentication", 1989), but implement some added features that were missing from BAN.

Mutual Authentication

When two parties both authenticate themselves to each other through whatever means, it is referred to as mutual authentication. This ensures that the client can trust the server but, perhaps more importantly, the server can trust the client. Very few protocols use mutual authentication – in fact many common protocols such as TLS rely on single authentication of the server (Abdalla et al, "Provably Secure Password-Based Authentication in TLS", 2006).

Mutual authentication can be taken further, where the client and the server send periodic challenges to the other. The most common way of doing this is where the client and the server generate their own public/private key pair, and send the public key to the other. When a challenge is made, the challenger encrypts a simple piece of data with the challengees public key – if the challengee returns the correct data decrypted, then the challenger can confirm that the challengee is the same entity that originated the transaction.

The need for this kind of periodic authentication check is mainly used for long transmissions of sensitive data such as a financial transaction. With less time-intensive transmissions a simple method of passing certificates between the parties involved is more efficient (Chang and Chang, "An Efficient Authentication Protocol for Mobile Satellite Communication Systems", 2004).

Authorisation

Authorisation is the act of confirming that an entity has sufficient rights to continue with its actions. This is not to be confused with authentication – an entity can be authenticated, but still not have authorisation, just as an entity may not be authenticated and still have authorisation.

Authorisation could be considered the most important aspect of security, as this actually allows entities to do things (or not) (Thompson et al, "Certificate-Based Authorization Policy in a PKI Environment", 2003).

Two forms of authorisation exist, depending on their actions. The first is where only entities listed are allowed to access resources or services. The second is where all entities except those listed are allowed to access resources or services. The first employs what is called a *whitelist* and the second a *blacklist* (Xie et al, "Privacy and Authentication", 2006).

Obviously, the whitelist is more secure than the blacklist – a new virus could be spread that hasn't yet been identified and still be authorised under a blacklist, whereas under a whitelist any new entity requiring access to services must be investigated first before being allowed onto the list.

The downside of a whitelist is the added administration that is involved – new versions of software and other mild modifications on already existing rules on the whitelist must be updated. One option is to use regular expressions for list items, however this also detracts from security – a specific virus could name itself something that matches a common entry in a whitelist, for example.

Accounting/Auditing

There are two aspects to accounting and auditing. The first is auditing the security policy and its implementation directly, that is querying it at regular intervals to make sure that the rules created to apply a security policy are still in effect. (Firesmith, "Engineering Security Requirements", 2003).

The second is an audit trail of all entities that interact with these rules – whether or not they are authorised for access to a service, for example. This is the type of accounting usually thought of when referring to auditing and audit trails, and is useful not just to the organisation in question. It is this type of logging that can be used in court for defence or prosecution for any cases of of a breach of the law regarding computing security (Allinson, "Audit Trails in Evidence", 2003).

3.3 Kerberos

Methods already exist to authenticate principles in a network. The primary one used is Kerberos, which uses mutual authentication between a client and server (Steiner *et al*, "Kerberos", 1988).

Kerberos is beneficial in that it provides integrity (as described above) to the communications, and also helps prevent replay-attacks (a form of man-in-the-middle attack, where a specific message is stored by an eavesdropper for later use) and eavesdropping. (Kohl *et al*, "The Evolution of the Kerberos Authentication Service", 1994)

Kerberos utilises the Needham-Schroeder model of encryption over large networks (Needham and Schroeder, "Using Encryption for Authentication in Large Networks of Computers", 1978) with the addition of timestamps to provide the authentication and uniqueness of each message to prevent replay.

Entities are authenticated in Kerberos, with no necessary distinction between a user and a process. However,

3.6 Conclusion

Using both certificates and hash values in the authentication procedure would give strong authentication to a process requesting access to the communication channel, in the form of something the process has (the certificate) and something the process is (the hash value). This is stronger authentication than the majority of operating systems use to identify the users logged into the systems, which is beneficial as the majority of processes being executed under a users privileges have nothing to do with the user – they are mainly core processes or derivative processes.

A whitelist, as opposed to a blacklist, will provide a greater level of security for the system. Whilst this would mean more administration for the sytems administrator, it provides a greater degree of granularity, allowing each individual user to have their own allowed list of processes. This is especially important in a large corporation where different departments require different access to different programs.

Kerberos could be adapted for use with the proposed protocol, but it is a distributed authentication protocol, requiring a remote server to handle the authentication. The proposed protocol would exist entirely locally on a computer, and as such Kerberos would need to be implemented on each computer specifically for this purpose. It is far more efficient to develop a lightweight system to do this instead.

4. Design

4.1 Introduction

From the literature review, it can be seen that various sets of principles exist to aid in the construction of security implementations, and this information can be used when designing the proposed system.

This chapter outlines two main areas. The first is the design of the underlying protocol, using accepted design methodologies used in constructing protocols for industry and academia, based around the communication system desired, and a subsequent model of the system in a number of finite state machines.

Secondly, a design of a proof-of-concept system for implementing and testing the protocol is presented, with considerations made at that early stage for the results that will be required for the subsequent analysis.

Lastly there is a brief discussion on an outline for a testing methodology, as it is good to have this information at an early stage, to ease the design and the evaulation itself.

4.2 Protocol Design

There are many methods for the design methodology of protocols, however the following is the one recommended by the International Standards Organisation, and results in an almost complete document in itself for presentation to that body. It relies heavily on finite state machines, a notation where a description of a finite number of states are illustrated that a system may exist in, and the required steps taken to move from one state to another.

4.2.1 Requirement Analysis

The first step is to define the requirements of the system.

1. The architecture will be that of a client/server system.
2. The client will be required to pass a number of pieces of information to the server for verification:
   - The username of the user running the process
   - The name of the process
   - The certificate of the program

   - The hash signature of the program

3. The server will be required to pass a number of pieces of information to the client for verification:

   - The certificate of the server

4. Both the client and server will be required to authenticate the other.

5. Any failure of validation of credentials will result in discontinuation of communications.

   - Should the server fail to be verified, the log should reflect this so the system administrator can investigate.

6. The server should log all requests, for full auditing purposes.

7. The exchange may be aborted at any time, by the server or the client.

8. Messages between the client and server, and vice versa, must have guaranteed delivery.

The distinction between a program and a process is important here (as discussed in the Theory section, above) as the requirement must take into account that a user (or malicious process) can install a similarly named program onto the operating system. The authentication system must be able to deal with that, taking into account not only the user running the process, and the process name itself, but also the path of the program being run. For example: the computer contains a certified version of a program called WebBrowse, with the executable program WB.exe. There is a new version of the product available that has not yet been installed on the computer, and contains some new features that a user wants to use. It is possible for the user to download the new version and install a copy local to them, and the new version has the executable WB.exe. Unless the path of the program is declared as part of the registration, it is possible that this new, non-certified, version of the program is allowed access to the network.

4.2.2 Communications Diagram

From the above requirements, a diagram can be constructed that will illustrate the communications between the client and the server during a request. This diagram does not illustrate what the client and server does with the requests, merely the required order of messages.

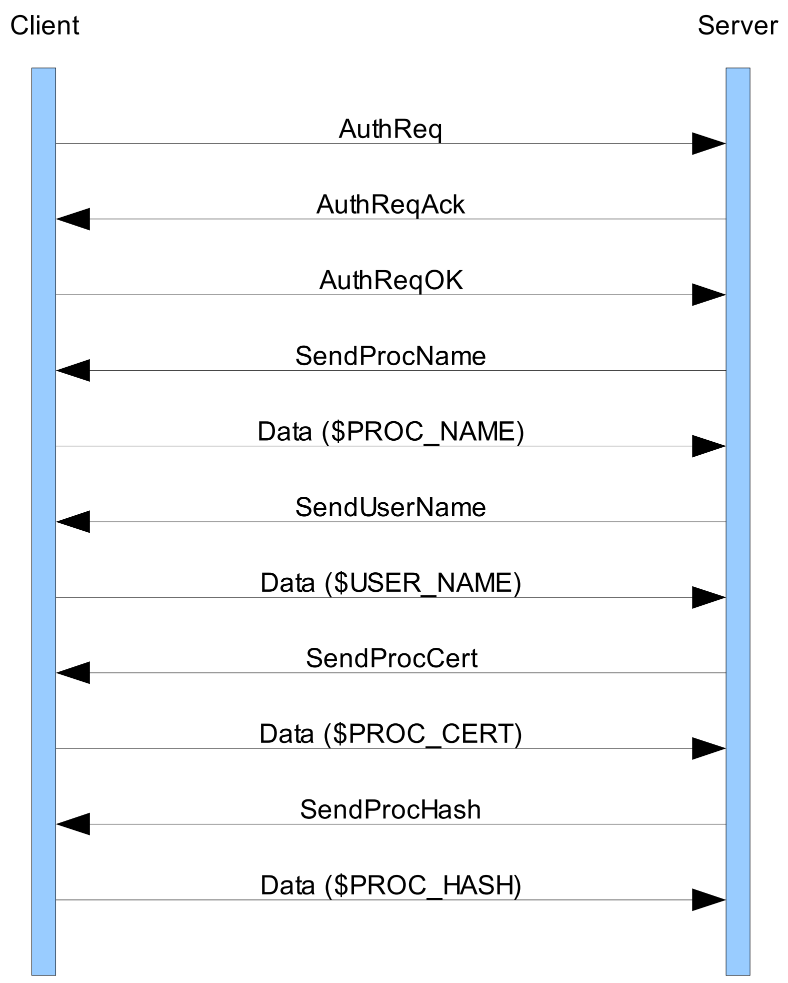


Fig 4.2.1: The first stage of the request

Figure *4.2.1* represents the initial request from the client to the server, consisting of a request for authorisation, and the server querying the client for the required information. Once that is complete, the server then validates the information, and either notifies the client that the credentials are not valid, or that they are, and continues with the exchange.

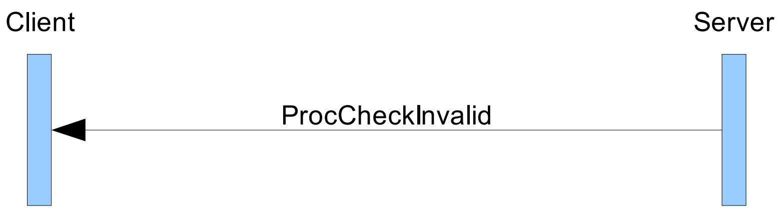


Fig 4.2.2: A possible reply from the server on the credentials from the client

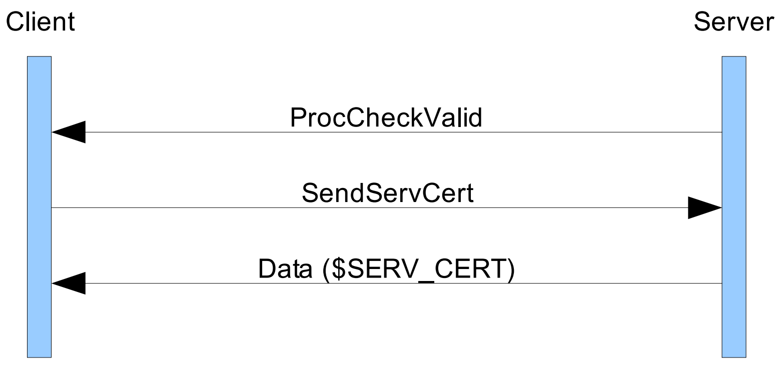


Fig 4.2.3: The client requesting the credentials from the server

Figures *4.2.2* and *4.2.3* show part of the exchange represents the client querying the server for the required information for validation. As with the server, above, the client then either notifies the server that the credentials are invalid, or continues.

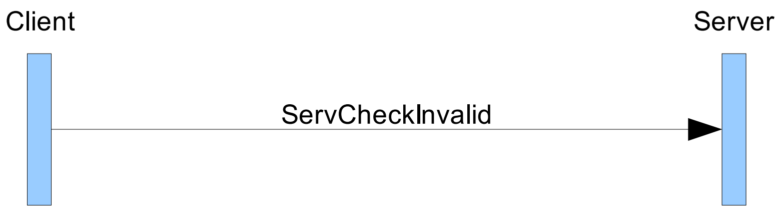


Fig 4.2.4: A possible response from the client to the server regarding validation

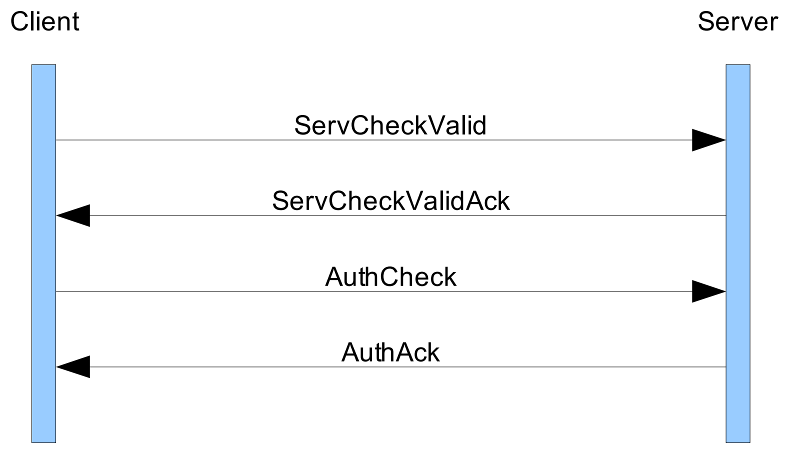


Fig 4.2.5: The final request for authorisation from the client

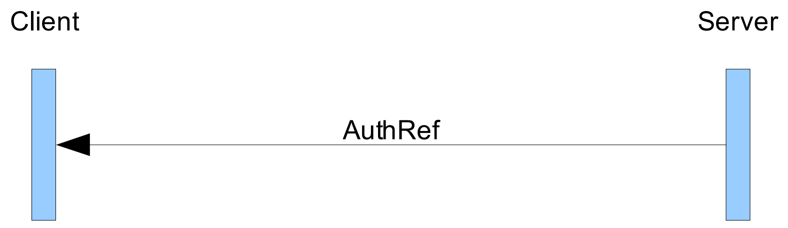


Fig 4.2.6: A possible response from the server to the client regarding authorisation

Finally, as shown in figures *4.2.5* and *4.2.6*, the client requests the authorisation check to see if it is allowed to communicate over the network. Again, the server may deny such a request if the process/username combination is not held on the server, or if it is, it notifies the client it may continue.

4.2.3 Finite State Machine

Although the above communication diagrams allow us to visualise the order of communications, it does not show us the exact output (or outgoing event) given a specific input (or incoming event), nor does it show the exact flow of the protocol. For example, in the diagrams above it was helpful to not show that an 'Abort' message may be sent at virtually any time, which will revert both client and server to their initial states – the diagrams would have had to have been broken down significantly to allow for that, making them unreadable. As it is, the rejection of credentials or authorisation for access to the network is shown only to illustrate the possible 'main' events.

The resulting diagrams in the above section do, however, let a more complete picture be built of the various states that both client and server may be in at any given time, though not concurrently. These state transition diagrams may seem simple in comparison to larger complete systems required for programming, and that is due to the nature of what is being designed – a protocol. Protocols (generally) contain specific linear requirements, and as such the flow of transitions through a finite state machine are demonstrably linear also. A finite state machine simply illustrates where a system may be in only one of a finite number of possible states, and is therefore ideal for designing logical flows of control most commonly found in computing.

Figures *4.3.1* and *4.3.2* represent the flow of control in the client and server using the finite state machine notation. The initial state is represented by the state with the bold outline. Each transition is represented by a line connecting two states, the arrow pointing to the end state during transition. The labels on the transitions are of the form "Incoming Event: Predicate [Action] Outgoing Event" where the predicate, the action, and the outgoing event are all optional. The incoming event is required, as it is the trigger between states.

The predicate is described as "p*n*", where *n* is a number. The predicate definition is listed after the diagram, and in its form "p*n*" in the diagram shows where the predicate is true. In the form "^p*n*", this shows where the predicate is false, the caret (^) representing logical negation. So a line "^p01: [0] Abort" reads as 'if the predicate p01 is false, do action [0], and send an outgoing event 'Abort'".

Actions are again listed after the diagram, and represent the actions taken by either party when that incoming event arrives, if the predicate allows.

Figure *4.3.1* is that of the server. The linearity can be seen as there is only one incoming event that will move the system to the next state, until completion. Each 'Data' event represents pure data that arrives, as shown in the previous communication diagrams. The contents of these 'Data' events is dependent on their position in the exchange, and at each stage both parties will know what they represent, as it is in the definition of the exchange.

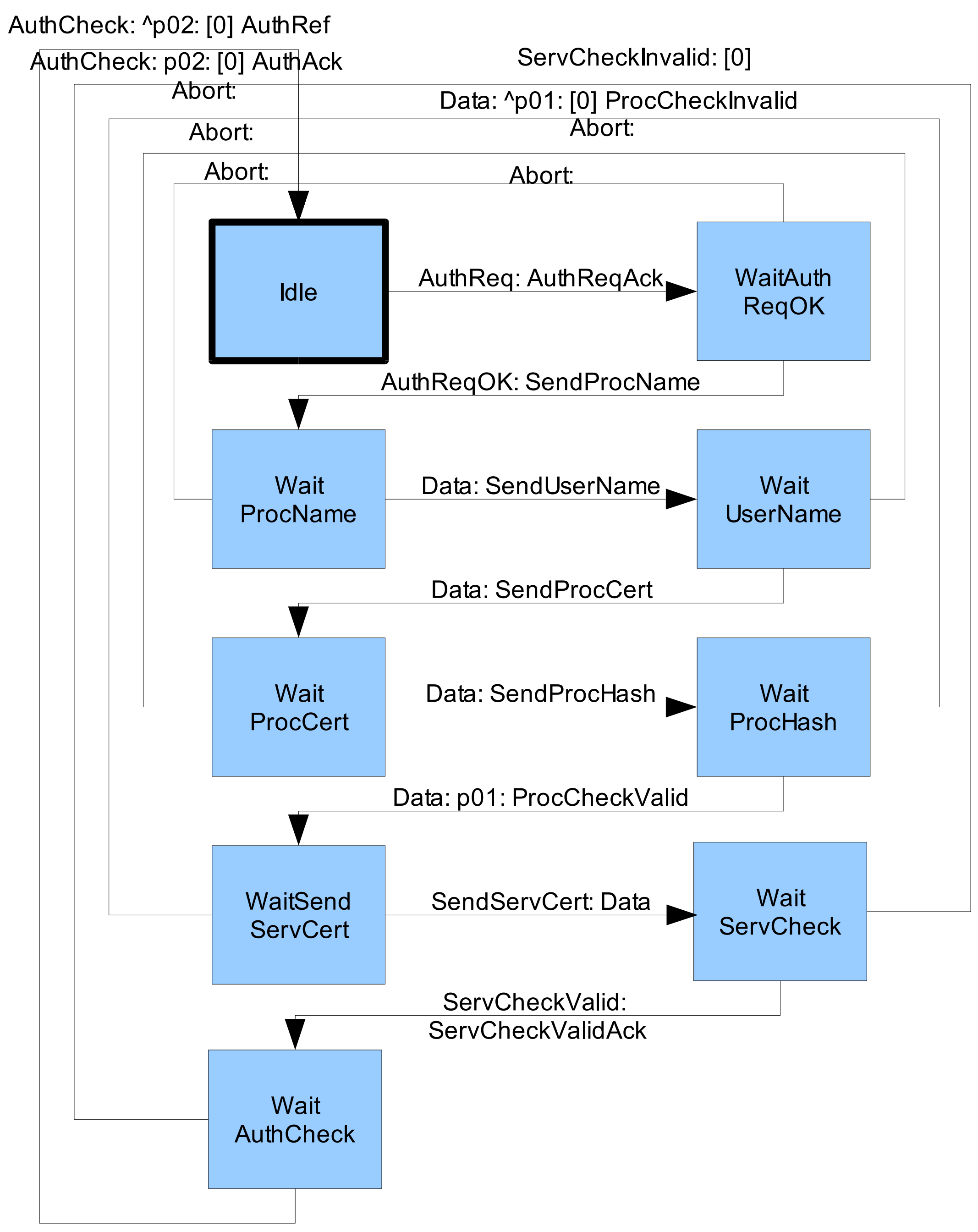


Fig 4.3.1: The finite state machine for the server

Predicates:

- p01: True if the credentials of the client are validated
- p02: True if the process is allowed access to the network

Actions:

- [0]: Write the event to the log

Figure *4.3.2* is that of the client. As in the case of the server, above, the contents of the 'Data' events are dependant on the state of the system.

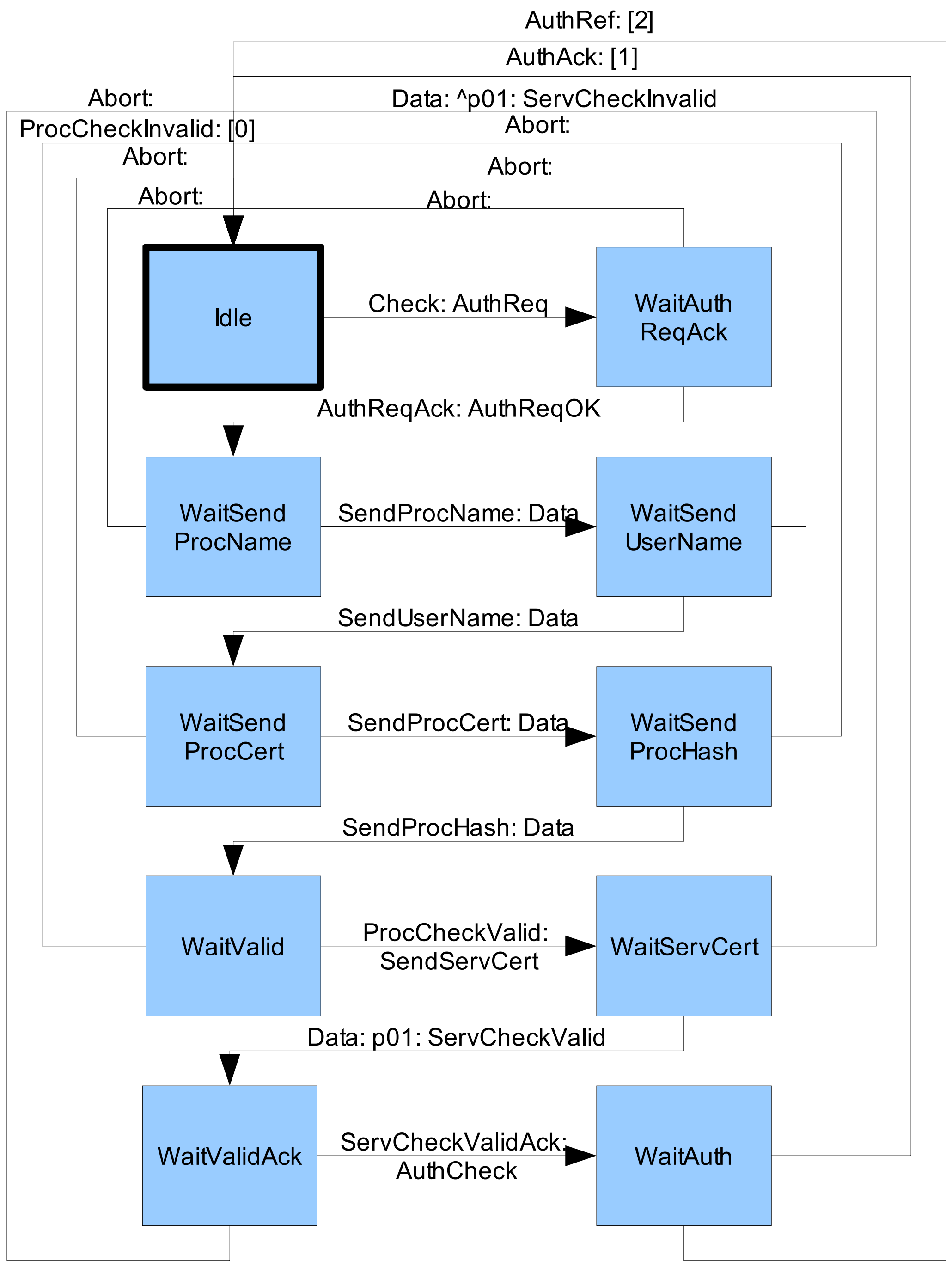


Fig 4.3.2: The finite state machine for the client

Predicates:

- p01: True if the credentials of the server were validated

Actions:

- [0]: Return that the credentials have been rejected
- [1]: Return that the process is authorised
- [2]: Return that the process is not authorised

From this, a tool called PRIDE was used to emulate the protocol, to test for completeness (see appendices J and K for configuration files for PRIDE).

4.3 Proof of Concept Design

Once the protocol has been designed, a method for implementing it can then be constructed. There are a number of key aspects of the requirements that can only be dealt with in an implementation of the protocol:

- Logging the outcome of a request
- Creating a checksum of the requesting program
- The list of allowed process and user combinations
- Collecting the required credentials for presentation
- A large number of requests may be received simultaneously, and the implementation must be able to handle this

From these specific requirements, the structure of a proof-of-concept implementation emerges.

Logging the outcome of a request is trivial, requiring minimal coding to write the outcome to a file. This can be included as a method within the server program.

Creating a checksum of a file can be difficult, and is best approached by creating a separate object for this facility. Depending on the language of the implementation, this object may contain the entire algorithm for hashing, or it may simply consist of a wrapper for a library implementation of the hashing algorithm.

There are a number of ways of implementing a list of allowed process and username combinations, including two-dimensional arrays; linked lists; either a program object containing a list of users that object is allowed to access, or the inverse – a user object containing a list of programs that that user is allowed to access; or as a database, again referenced by program then user, or user then program. Given the nature of this system as a whitelist as described in the literature review, it is best to go for the latter option, as it allows easier maintenance of the list should a user leave an organisation – it is one entry to delete. There is a possibility that if implemented any other way a single program and user combination may exist that refers to the user who has left. Whilst this access is still relatively benign (the program must have been 'okayed' by a systems administrator to be on the list in the first place) any outdated information may provide a vulnerability if discovered. For the proof-of-concept implementation, however, it is more prudent to use a user object containing a list of programs, emulating the equivalent referencing in a database, but simplifying the practise. Indeed, a database might only be required for large multi-user servers running multiple shell accounts – otherwise having the details stored in memory might give the best balance between speed and resource use.

Collecting the various credentials again is trivial, requiring minimal coding to interrogate a file for the information required, and formatting it correctly.

Lastly, to handle a large number of requests the system must implement a form of threading, as discussed in the literature review. The proof-of-concept implementation will use a thread-per-request model, which is the simplest method of threading to implement. To provide for this, a management object should be created to manage incoming connections, and a service object to process the request called from that management object.

At the client end, only one object is required, which will handle credential gathering and negotiations via the protocol.

4.3.1 UML Diagram

It is now possible to sketch a basic UML diagram to show the structure of the server program.

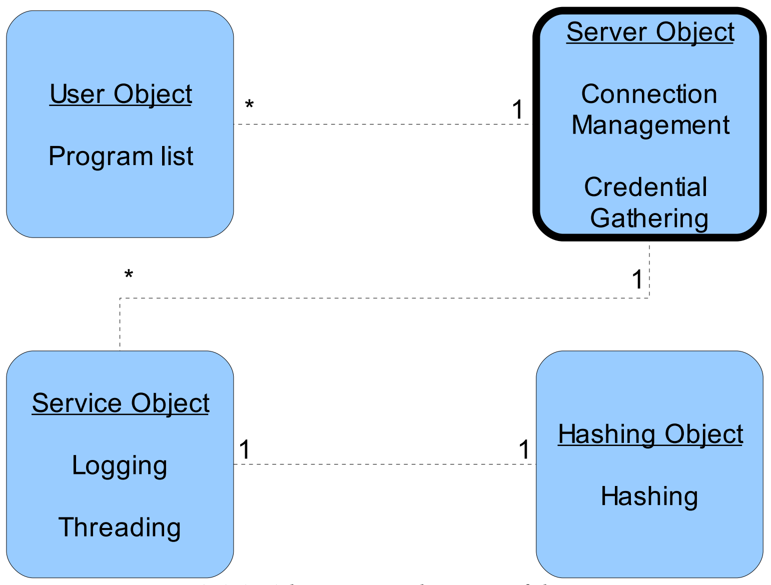


Fig 4.3.1.1: A basic UML diagram of the server

The entry point for the program will be the Server Object (denoted in figure *4.3.1.1* as such by the bold outline) which will then add a list of User Objects to a local list, and retrieve the credentials required, storing them in memory. These User Objects contain a list of programs that the user associated with that object is allowed to run. When an incoming connection is detected, the Server Object creates a Service Object to handle that request, and passes the connection over. This Service Object then starts a separate thread for that request. When required, the Service Object creates a Hashing Object to handle hashing for the request. Once the request is completed, the Service Object logs the outcome of the request for auditing, and closes the thread.

4.4 Testing Design

It is important at this point to design the testing and evaluation framework, to see if the implementation works, and how well.

To see if it 'works' is relatively simple, and will be covered in the evaluation section directly.

However, to see how well it works requires some more planning, around a number of key points.

- What is the effect on the time taken to communicate over the network?
- How does this effect change with an increased number of requests?

In evaluation, specific tools are required to measure the various aspects of the testing. Thus, a program is needed that can be adjusted to test for various variables against the implementation.

First, the control functionality of the test. This should be something fairly basic and simple, such as passing a message from a client to a server. This will remain the same in each experiment. The client and the server should note the time the procedure started, and the time it was completed (i.e. when the client starts the procedure to send the message, whether with the system to test or not, and when the server finally receives the message). This is the measurement that will be used in evaluating the system.

The experiments that would be run then would be thus:

- Control experiments
  - Various amounts of messages without the system
- Evaluation experiments
  - Various amounts of messages with the system

This information could then be plotted on a chart, and an analysis of the information made.

As for designing the test programs, it must be ensured that the programs are as near identical as possible, so as not to affect the results other than those areas required for analysis. To this effect, it would be wise to use the same program with different parameters.

4.5 Conclusions

The major relevent parts of the system have been designed, which (as with any software) eases the implementation. Following a systematic protocol design methodology provides a direct flow-of-control structure that can be implemented almost directly in the implementation.

Further, by considering the methodology used for gathering results at this stage, it is possible to add this functionality at a design stage, including (but not limited to) the format of the output.

5. Implementation

5.1 Introduction

It was decided to use the Microsoft .NET framework for implementation, using the free Mono open-source implementation of .NET, to make the program as cross-platform as possible. The language chosen within .NET was C#, which has very good network component support and threading support.

This chapter outlines the various challenges involved when implementing the protocol in a linear format with branches in the control structure. A discussion takes place of choices made when implementing the system that incorporates the protocol, specifically regarding areas of the implementation that are already widely used, such as methods for validating certificates.

Lastly, implementation options for the test system are discussed, including what method of timing was used.

5.2 Protocol Implementation

The protocol is encapsulated in the services offered by the system. During construction, the original method of many levels of nested if-else structures was deemed too complicated both to follow during coding and debugging, but also in terms of complexity for the program to follow. As the protocol can be aborted at any time, each condition checked for the latest response from the client and if the message received was not "Abort" then it would continue with the next stage of retrieving a response from the client and evaluating it. If the response was an "Abort" message, then the condition would run the code, which in this case was empty, and then exit the condition, thus implementing the "Abort" sequence.

```
if (latestInput == "Abort") {

} else {

  // Continue protocol, update latestInput

  if (latestInput == "Abort") {

  } else {

    // Continue protocol, update latestInput

    if (latestInput == "Abort") {
      ...

}

// End the connection
```

It was then decided to use a series of non-nested if-else statements, with the use of the `goto` statement to provide the exit functionality. This provides a more linear and less intensive method of analysing the incoming messages from the client.

```
  if (latestInput == "Abort") {

    goto end;
  } else {

    // Continue protocol

  }

  // Update latestInput

  if (latestInput == "Abort") {

    goto end;

  } else {

    ...

  end:

  // End the connection
```

It is then possible to point to a specific block in the implementation of the protocol and match it up to a corresponding state in the state transition diagram.

The use of the `goto` function has long been considered a sign of weak programming (Dijkstra, "Go To Statement Considered Harmful", 1968), which is why the original implementation did not utilise it. However, this is one of the few situations where such a use is not only useful, but recommended.

5.3 PROOF-OF-CONCEPT IMPLEMENTATION

Whilst full certificates have been included as part of the protocol, they will be left out of the proof-of-concept system, for a number of reasons. First, to accurately reflect the efficiency of the system, any outside influence over the time taken to complete a request needs to be limited – the ideal scenario of checking certificates would involve a third party certificate server, and the negotiations with this server are variable in time depending on network traffic or the implementation of the certificate server, which could skew the results. Secondly, there are numerous implementations for certificate checking, and indeed many programming languages contain classes for manipulating the most common certificate types as part of their core libraries. Any work done to implement a certificate checking method for the system alone would not be constructive.

From further work during the design (and the choice of a specific programming language) it is now possible to construct a full UML diagram for the server *(Fig. 5.3.1).*

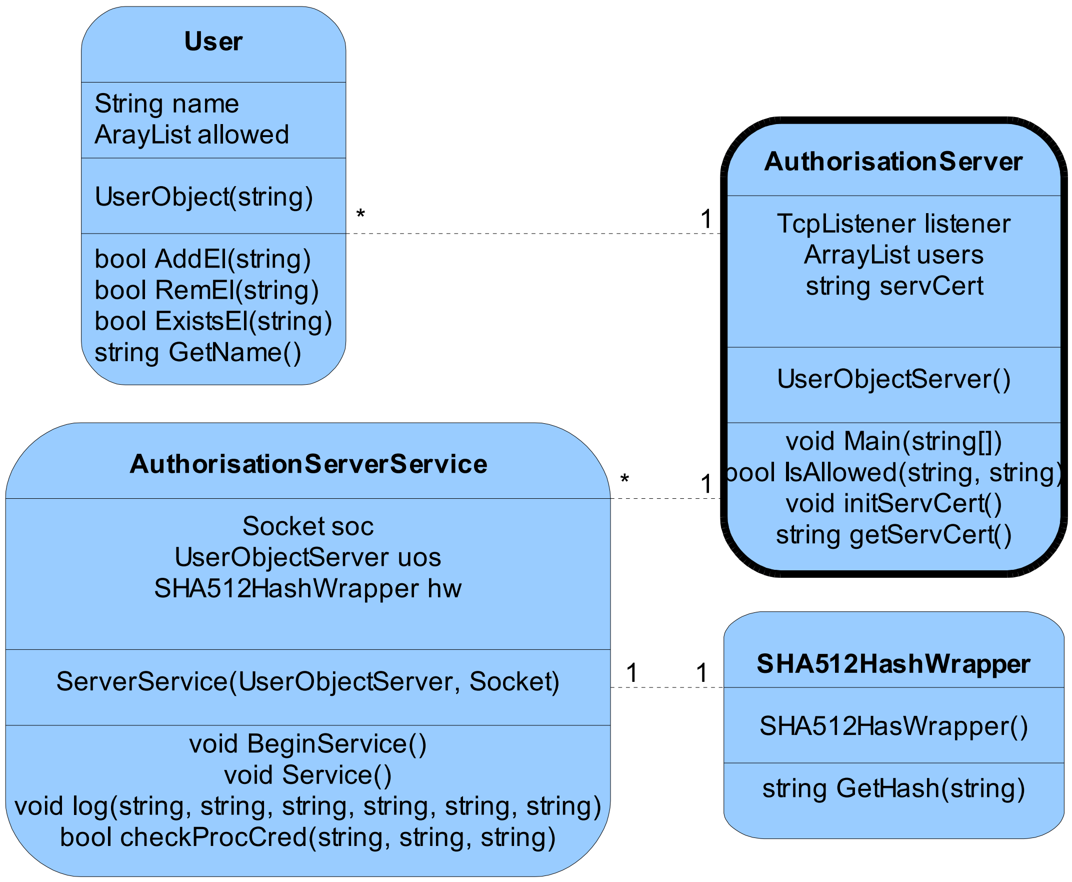


Fig. 5.3.1: A complete UML diagram of the service

It was decided not to use a graphical user interface at any stage in the program, as an aim of this system is to be easily integrated into any existing code with minimal effort. Also, a GUI adds nothing to the overall usability of system and, in the case of the authorisation service, could in fact detriment operational efficiency.

5.4 Testing Implementation

Two simple programs were constructed. The first, the server, listens for incoming connections, opens them on a new thread, receives the message and displays the time the message was received at. The second, the client, takes two parameters – whether to use the authorisation system or not, and the number of connections (threads) to make. The first thing the new connection does (whether or not it will check the authorisation server) is to display the number of the connection (for tracking) and the time the connection was initialised. If the authorisation system is used, then for each connection a request is made to the authorisation server before the client sends the message to the server (which is simply the connection number, so the connection can be tracked at the other end).

This then gives two sets of information: a list of connections and when they were initialised, from the client; and a list of connections and when the message was received, from the server. From these two pieces of information, the difference in time can be calculated, giving us the total time per connection for completion. An average time can be worked out and used for analysis.

The time base used is simply the hardware clock time – unlike systems that operate over the network and require some sort of central timing service, that is already provided in a purely local system. Plus, as the measurements are purely comparative, there is no need to provide incredibly accurate timing functions, as long as the time measured has a suitable grain. The .NET function DateTime.Ticks returns “the number of 100-nanosecond intervals that have elapsed since 12:00:00 midnight, January 1, 0001” (sic) (Microsoft, “DateTime.Ticks Property (System)”). Upon initial testing, no value declared by the server is the same as that declared by the client, so the granularity of the time function was deemed sufficient.

5.5 Conclusions

The programming required in the implementation was simple in comparison to larger applications, due to the nature of the implementation – that of a simple service to provide authentication and authorisation using a simple protocol. Sometimes, however, the simplicity of a system makes it all the more powerful, as this system is believed to be.

The simplicity of the system makes gathering results of experiments easier, however this does not necessarily mean that the outcome of the analysis of those results is any easier.

6. Evaluation

6.1 Introduction

Due to the considerations taken in the design for the subsequent evaluation, and the choices made in the implementation, the results gathered were of a level that can be analysed with relative ease, given the large quantity of data collected.

This chapter discusses the methodology used to evaluate the system and protocol, including limitations of the run-time environment that were not expected. Following this is an analysis of the data collected, which was collated, and has been presented in chart formats.

Lastly there is a discussion of the surprising conclusions made whilst analysing the results.

6.2 Methodology

The protocol itself has been tested during the implementation to make sure that it works correctly, in that it allows registered processes to access the device, and rejects non-registered or non-authenticated processes. This was carried out by running the test program from various locations, with various hash signatures. In the cases where either the hash signature was rejected, or the path of the program was not one of the paths registered, the response was a rejection. In all other cases (with a correctly registered path, and a correct hash signature) the process was authorised to run. (See Appendix H).

However, the main aspect of the hypothesis, which is also the more difficult to analyse, is the efficiency of the protocol. As mentioned in the design, above, the programs output the time between the request being initiated, and the request being completed. From this we can work out the time of a connection, from initial request to completion, for each connection running concurrently.

The range of concurrent connections chosen was aimed to stress-test the protocol and the proof-of-concept implementation, and as such is highly irregular compared to normal client/server communications. The idea of the protocol is to use it once when a process starts, and thereafter before any communication is made, the code returned by the authorisation service is checked. Even if the process spawns multiple threads, each thread will still use the results of the initial check.

In the testing situation, however, the processes spawn multiple threads and each of those threads is checking the server directly. There is one limitation on this which was discovered, and that is the runtime environment for the program (mono) has a defined upper limit of the number of times a single file can be opened. This was discovered when trying to use the authorisation service for requests over 1000 concurrent connections – the runtime environment would throw exceptions for each thread that failed to gain access to the file. This was deemed a useful piece of information to note in terms of limitations of the system, so a binary search was conducted in the test space between 1000 concurrent connection (where no exceptions were noted) to 2000 concurrent connections (where the first exceptions arose). However, there seems to be some form of caching involved in the runtime environment, as two tests of the same number of concurrent connections could give both positive and negative results for the exceptions. This made the upper limit impossible to investigate, and so the last always-negative result of 1000 was used as the upper limit for testing.

The blocks chosen then were in the ranged 10-100 in steps of ten, and 100-1000 in steps of one hundred. The reason for the split stepping was to see the efficiency of the system at a low number of concurrent connections, as this was most likely in a real-world scenario where the system might be used. The higher stepping was used purely for stress testing, and as discussed above is highly unrealistic.

Lastly, two scenarios were used whilst conducting the testing. The first was using an operating environment running a minimal number of processes, and steps were taken to make sure that anything that could supply a varying degree of operating resources use during testing was stopped. This was denoted the 'basic' scenario. The second scenario consisted of a normal running operating environment using a varying amount of operating resources during the testing of the programs, to simulate the use in a normal environment by a normal user. This was denoted the 'usual' scenario.

Experiments for each step were conducted in the basic scenario with and without the authorisation system, and in the usual scenario with and without the authorisation system. The whole series of experiments was conducted three times to gain a more representative average of the results. This resulted in the experimental matrix in table *6.2.1.*

	Basic						Usual					
	With			Without			With			Without		
	A	B	C	A	B	C	A	B	C	A	B	C
10												
20												
30												
40												
50												
60												
70												
80												
90												
100												
200												
300												
400												
500												
600												
700												
800												
900												
1000												

Table 6.2.1: The experimental matrix

The experiments were carried out, and the results in the output files of the various programs were collated into spreadsheets (both the raw data and the collated spreadsheets are included in the submission material). The average time was calculated and plotted onto a final spreadsheet. This final spreadsheet represents the results.

Two metrics were used. The first was the average time to complete a request. The second was the percentage of lost requests, either through TCP timeouts or thread failure. Again, as mentioned above, these losses in the higher set experiments are purely for stress testing, and it is highly unlikely such a scenario would exists in a full implementation.

6.3 Results

The comparisons following are between the same experiment set run against the same conditions, one set using the system and the other not using the system. Thus we can compare the efficiency of the system compared to the baseline (without the system).

The results represented in charts *6.3.1* and *6.3.2* were surprising, as it implied that using the system is generally *more* efficient than without, with lower average times to complete a request than without the system. Unfortunately, at the lowest end of the scale (where a proper implementation would most likely exist, between 10 and 30 concurrent requests) the system is less efficient, between 33% and 200% slower than without the system.

However, this average time is calculated using the packets that arrived. It could be that the system loses far more requests than without. Charts *6.3.3* and *6.3.4* compare the percentage of lost requests.

Again, the results are surprising. Fewer requests are lost using the system than without, at all levels of concurrent requests. This is even more surprising given that all the experiments at each level were carried out three times, which should have normalized any spikes and anomalies. The results on the whole, however, are very erratic, and could point to outside influences that were not immediately apparent when conducting these experiments.

Charts *6.3.1* through *6.3.4* show the results when the authorisation system was used in what was called a 'basic' scenario, which was so restricted in the outside processes running that it does not represent what could be classed as a 'usual' scenario. The results represents in charts *6.3.5* through *6.3.8* were based in the 'usual' scenario, described above, to see if the system running under normal circumstances fared any better or worse to the base results.

In the chart *6.3.5,* over 6.5% of requests made in 10 connections were lost. This must be an anomaly, as at such low numbers it is highly unlikely to lose any requests. It is possible (as we are now working in the 'usual' scenario, with unpredictable resource usage) that this would happen occasionally, but in this case it must be taken as an anomaly.

Again, generally, the system is more efficient than without.

It is possible to compare the two scenarios side by side, to see if any pattern can be spotted within those results. Doing this allows an approximation to be made between the performance of the system (and without) across the two scenarios.

Chart *6.3.9* is less erratic than those preceding, and is what is expected of such an increase of connections – as the number of connections increases linearly, the time taken to carry out those requests increases exponentially. However once the range exceeds 90 concurrent connections, that trend dissipates and become as erratic as before. However on the whole, the system again proves more efficient that without.

As before, the number of lost requests must also be analysed to provide a clearer picture, and as expected at relatively low levels the two scenarios provide much the same results, and as the number of concurrent connections increases, it takes longer to process a request in the 'usual' scenario (chart *6.3.11*). It is surprising to note that until the highest of this set of results, the use of the system under 'usual' conditions is more efficient than without the use of the system in a 'basic' scenario.

Even at higher levels (chart *6.3.12*), the system is more efficient than without, although not as drastic as in the lower set levels. However, apart from one spike at the 700 set, the system under the 'usual' scenario is more efficient even that itself in the 'basic' scenario.

At upper levels of the test set represented in chart *6.3.13*, the system eventually starts to break down and loses more packets in the 'usual' scenario compared to any other set. However, that trend doesn't continue into the stress test sets, as shown in chart *6.3.14*.

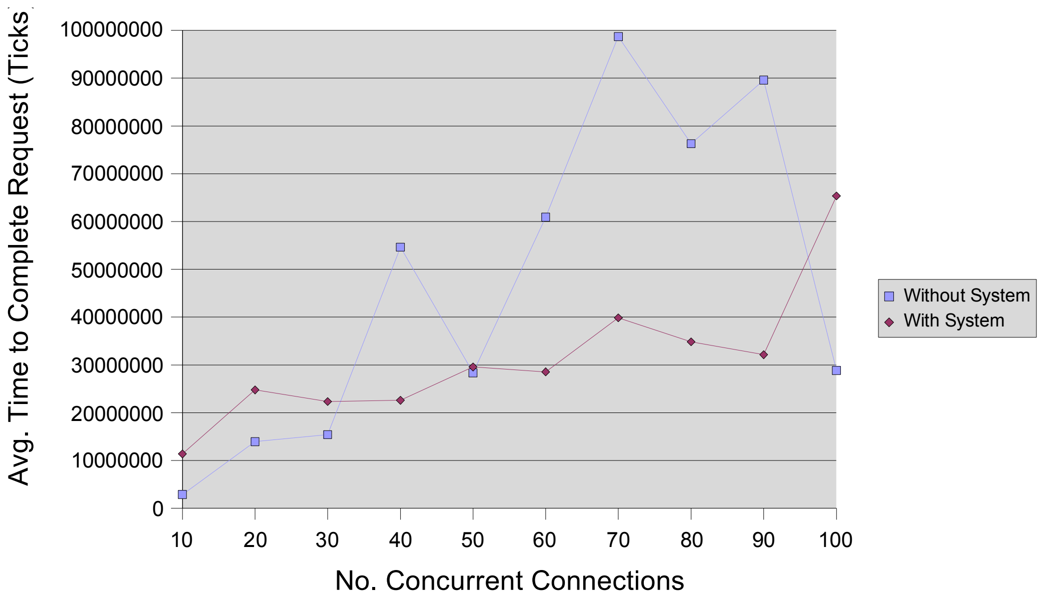


Chart 6.3.1: Avg. time to complete a connection in the basic scenario (10-100)

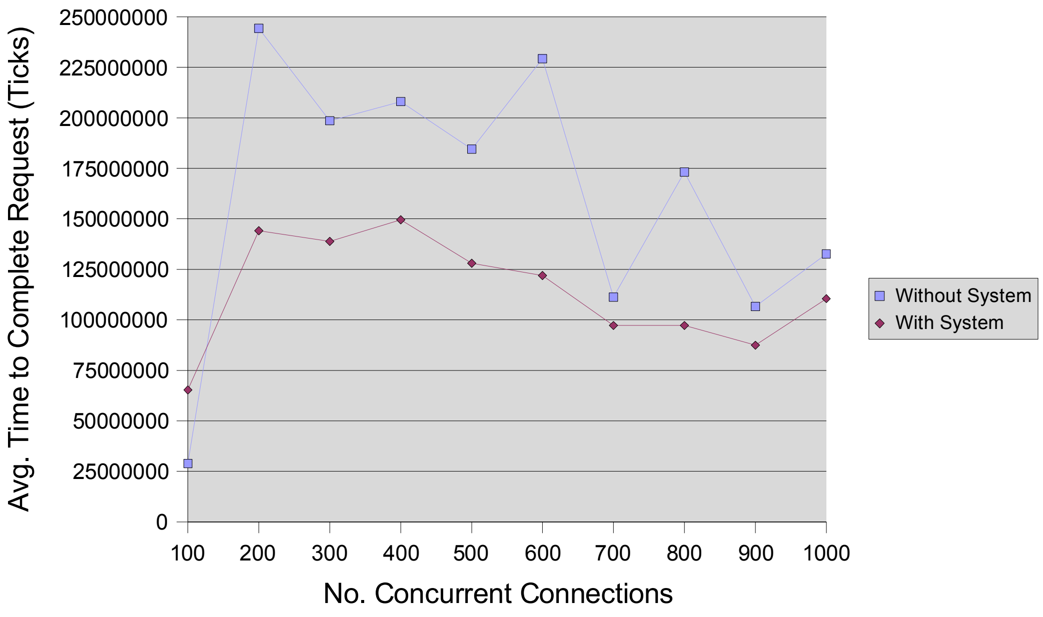


Chart 6.3.2: Avg. time to complete a connection in the basic scenario (100-1000)

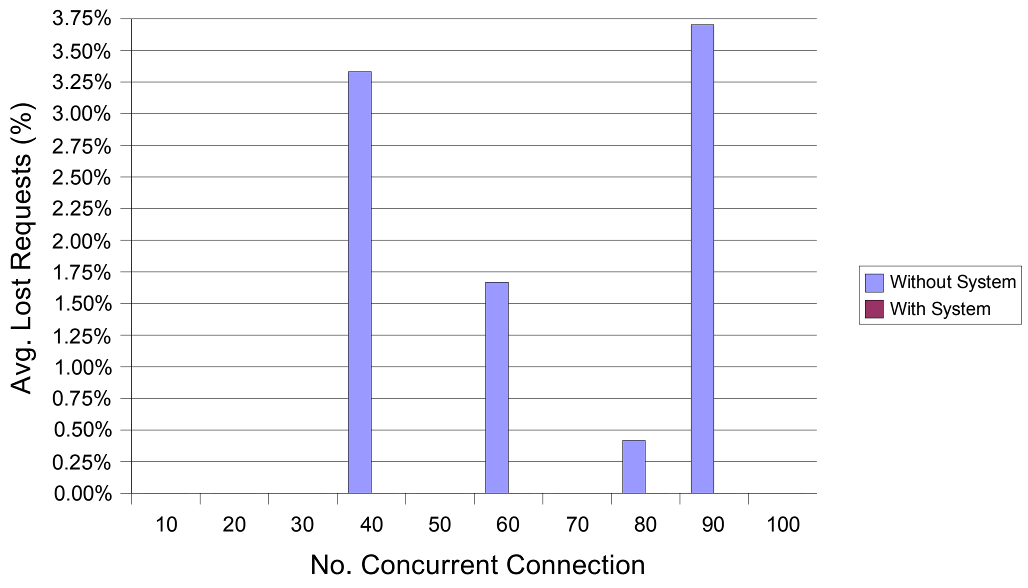


Chart 6.3.3: Avg. number of lost requests in the basic scenario (10-100)

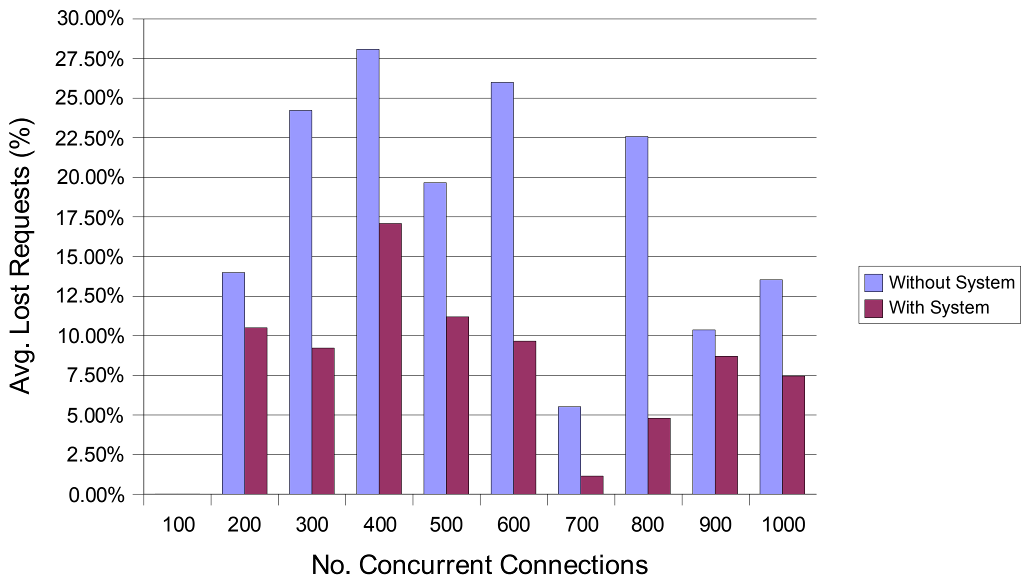


Chart 6.3.4: Avg. number of lost requests in the basic scenario (100-1000)

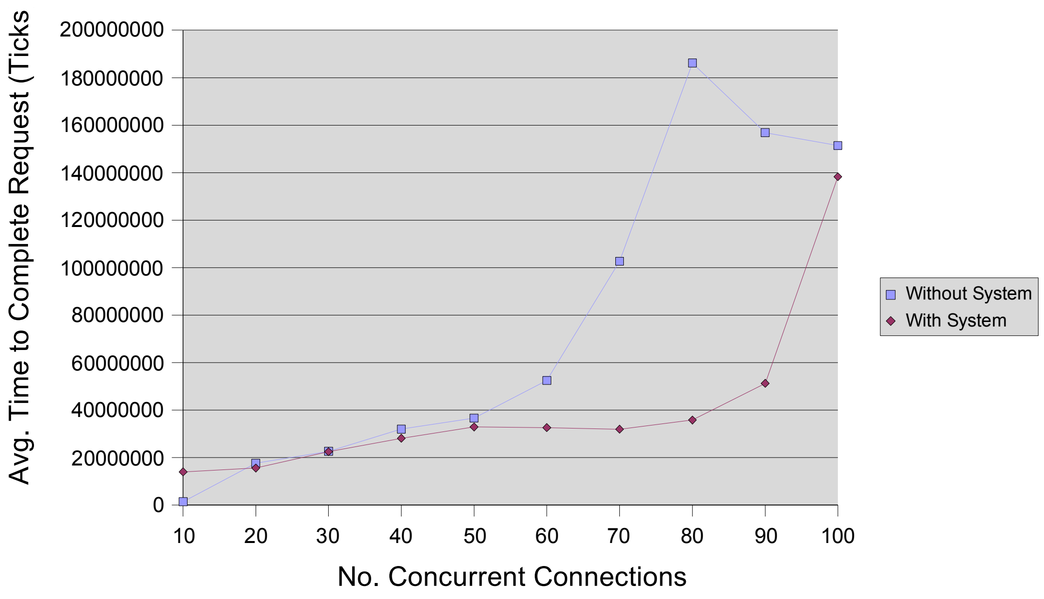


Chart 6.3.5: Avg. time to complete a connection in the usual scenario (10-100)

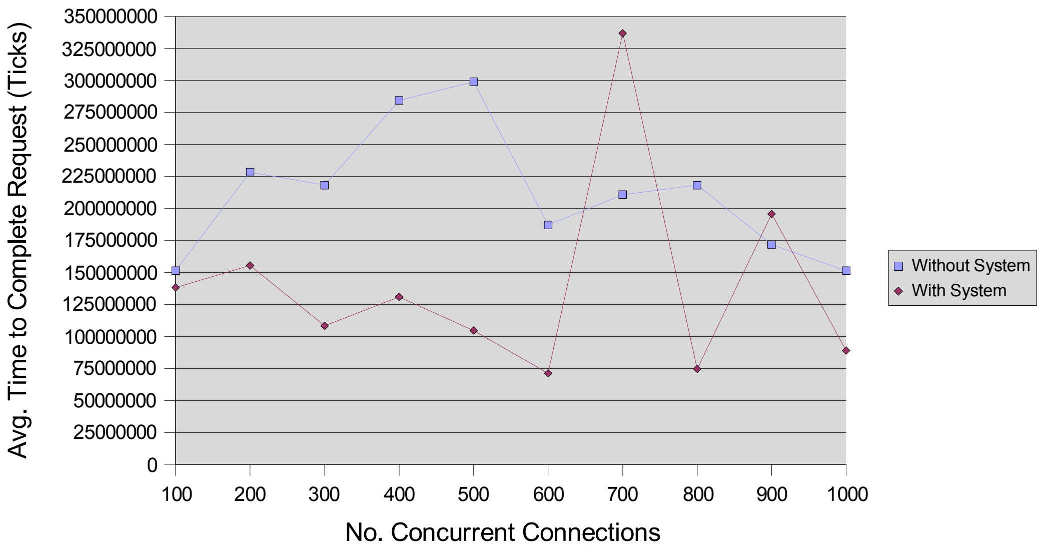


Chart 6.3.6: Avg. time to complete a connection in the usual scenario (100-1000)

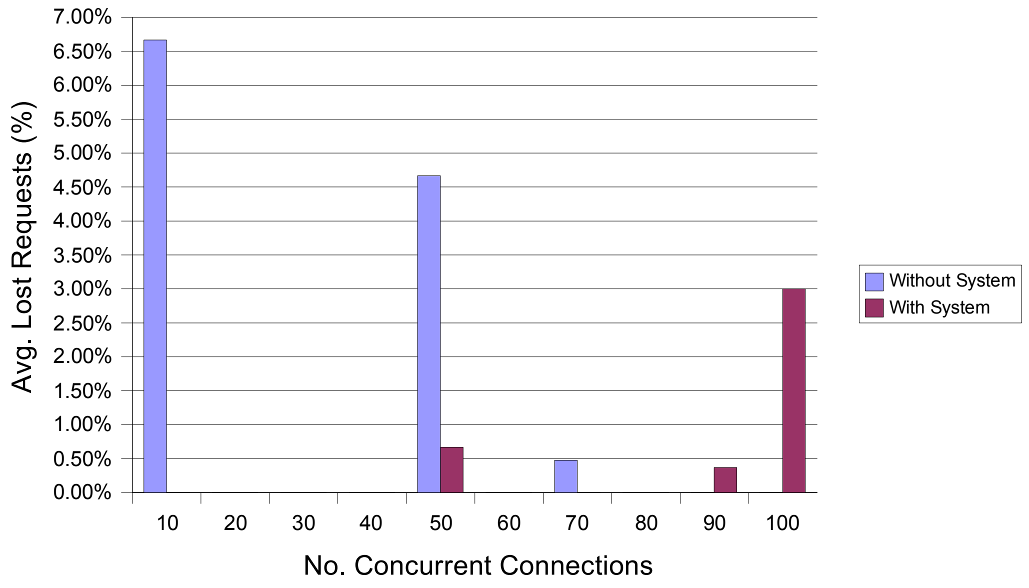


Chart 6.3.7: Avg. number of lost requests in the usual scenario (10-100)

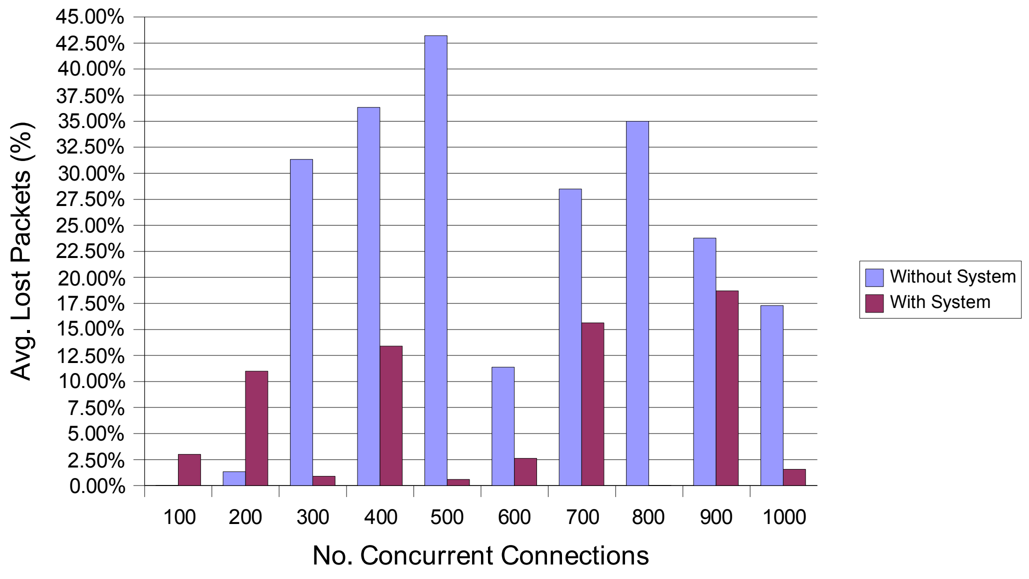


Chart 6.3.8: Avg. number of lost requests in the usual scenario (100-1000)

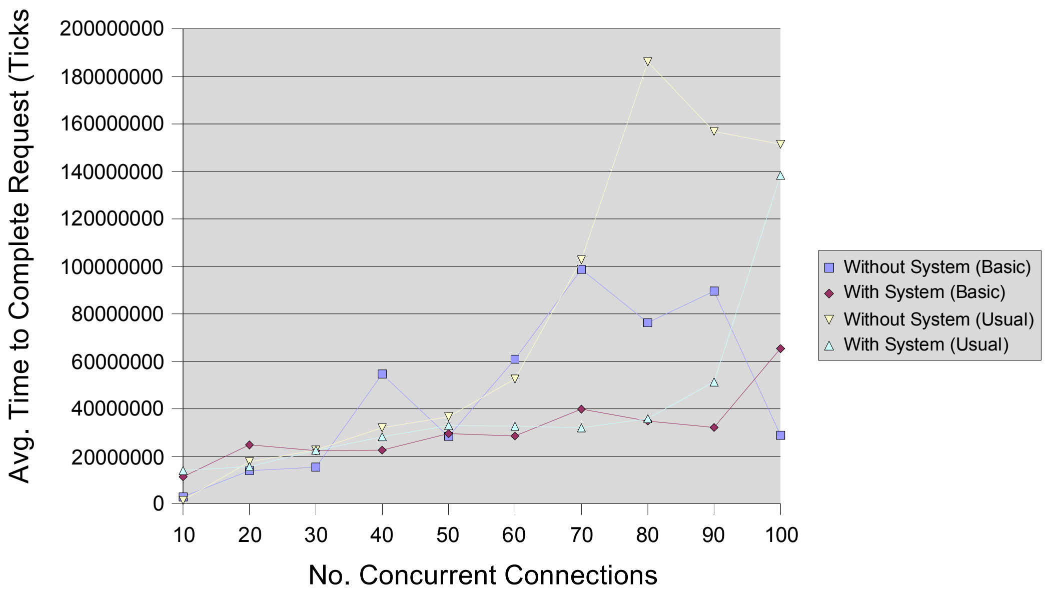


Chart 6.3.9: Avg. time to complete a connection in both scenarios (10-100)

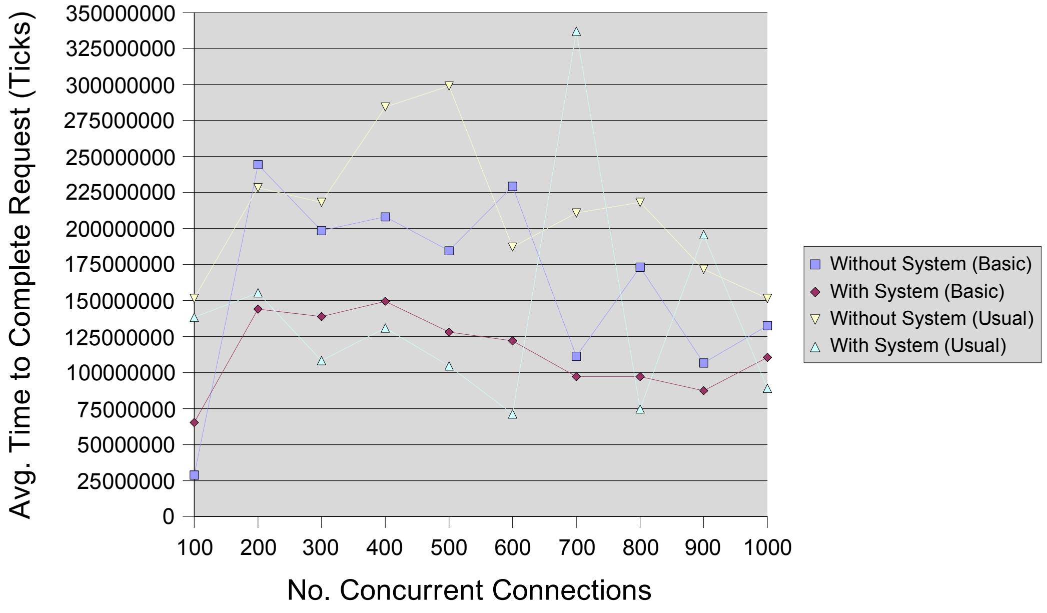


Chart 6.3.10: Avg. time to complete a connection in both scenarios (100-1000)

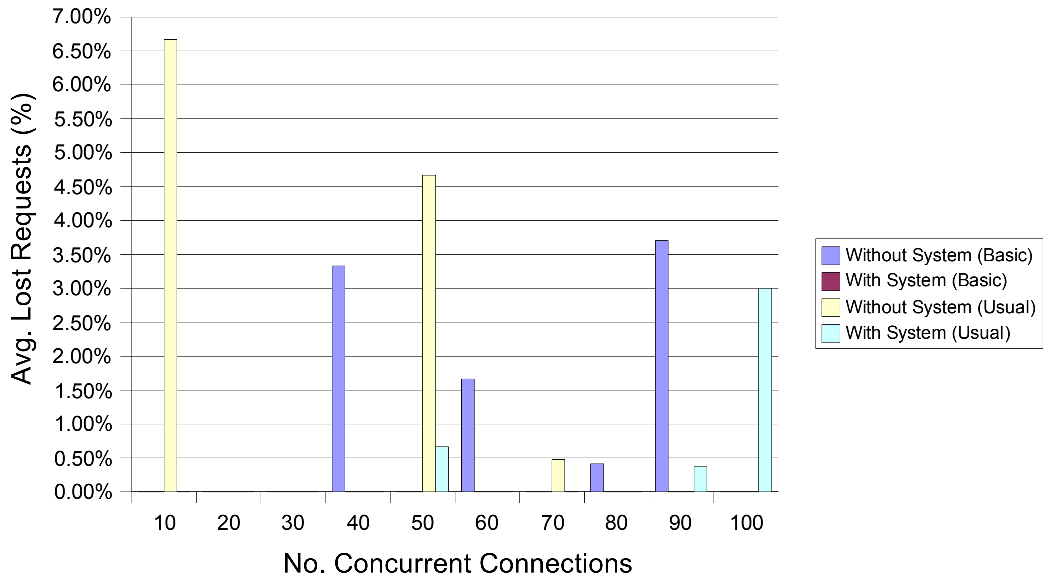


Chart 6.3.11: Avg. Lost Requests in both scenarios (10-100)

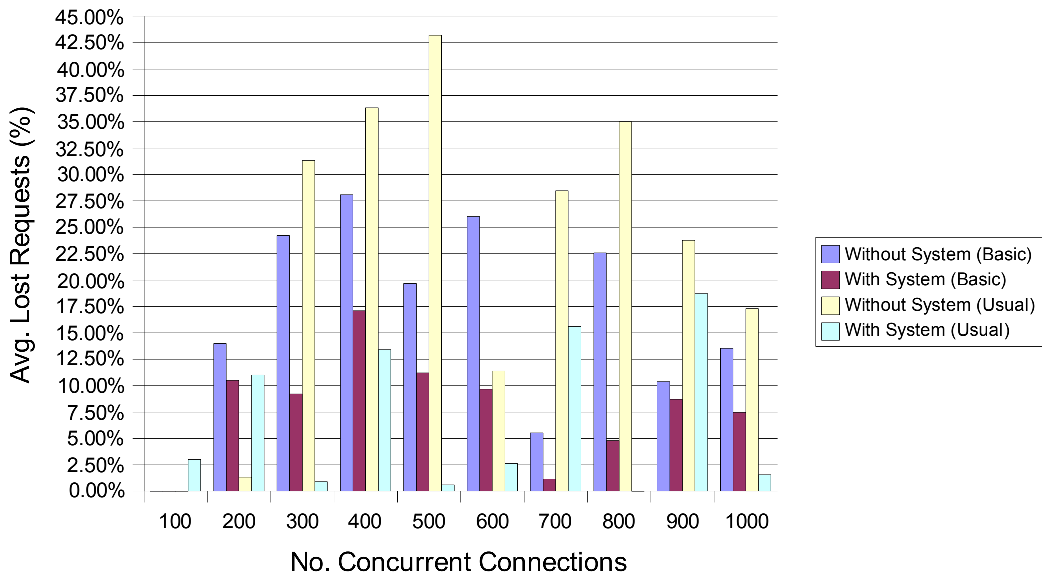


Chart 6.3.12: Avg. Lost Requests in both scenarios (100-1000)

6.3 General Discussion

The aim of the experiments was to find out how inefficient the system was over just a basic scenario without any authorisation taking place. The idea was to compare the figures, and present a case where the hypothesis defined in the introduction could have a positive response, along with a value (for example it is 20% slower that the baseline, but that is still within reasonable bounds given the speeds under discussion). The results, however, show that the system is more efficient under virtually every circumstance that a scenario where the authorisation system isn't being used. This is not only surprising, but thought provoking.

It would be remiss not to discuss the possibilities behind why this could happen. The system simply emulates a large number of client processes (in this case by using threads) connecting to a simple server – the only message passed between the two is a number, corresponding to the number of the process. Unlike a number of processes connecting to the server at once, the use of multiple threads means that all those threads are existing and being run by the same run-time environment, instead of separate environments, as in a proper use. The reason threads were chosen was for the simplicity in conducting experiments, and for being the only reasonable way to conduct concurrent testing – any other alternative would have involved sequential instantiation of the program, which would not have given the scope required for testing. These threads, as mentioned, exist in one run-time environment by design of the environment, to save on resources. However, that same resource saving could cause resource limitations for each thread.

This in itself is not enough to cause the skewed results, but when considered with the following, provides an adequate answer.

When the system is not in use, each client thread directly talks to the server, which creates its own thread for the client to communicate through. Each thread is created at (approximately) the same time, and is dealt with by the processor at (approximately) the same time. There is a one-to-one mapping between the threads. However, when the authorisation system is in use, it requires that the threads communicate with the authorisation service first, before then communicating with the server. At any one time, some threads will be in the process of communicating with the authorisation service, and the remainder will be in the process of communicating with the server. Thus, the resources are not being utilised by all the threads at the same time, there is a split in the resource use. The only time there was an issue with resource use when using the authorisation service was when too many threads were trying to read from the hash signature file at one time.

This is one possibility.

Another possibility is that there is some form of resource clean-up carried out by the operating system or runtime-environment during use – the more resources being used, the more rigorous the environment is in cleaning up resources to make sure there are sufficient resources for all processes running. This could mainly explain the disparity between the 'basic' scenario and the 'usual' scenario, where, when the authorisation service was used, there were better results than in the 'basic' scenario, when the authorisation service was not used. If true, it could also give some explanation to the results within each scenario, where the authorisation services provided better results than without – as the authorisation service was using more resources (as it was being used) the environment was more rigorous in 'cleaning-up' after each thread (this could have been the OS or the runtime environment).

Further testing would have to be conducted to get more representative results, and a different implementation of the testing would be advisable as well – one that does not use threads, but separate individual processes. This would require a large number of entries into the registered process list, as each process would need to be registered, and have access to its own hash signature.

6.4 Conclusion

The system works as intended, and seems to provide more reliable connections when used than when not used, along with the protocol itself working as intended. As discussed above, there are a number of reasons why this sort of behaviour was detected, but without further investigation a conclusive answer cannot be given.

In principle, this protocol could be implemented in a large corporate network as it stands, however before any commercialisation can be made it would require further testing, and implementing some of the further work, described below.

7. Conclusions

7.1 Introduction

This project was aimed at creating a protocol that could be used to prevent unwanted network usage, through implementing a whitelist of allowed process and user combinations. Secondary to this authorisation was the authentication of the processes requiring outside network access, ensuring that if they are held on the allowed list, that they are still the same process.

This chapter provides a critical analysis of the project as a whole. Firstly a critical analysis is made of the whole design and implementation process, including examples where other options may have been made to alter the results. A discussion on further work that could be carried out in this area follows, offering alternative means over which the protocol can be conducted and areas where tighter security may be added with lower-level operating system integration.

A theoretical situation is presented where such a system would not only be feasible, but desired, and where alterations may need to be made to allow the system to operate in such an environment. Lastly there is a general discussion on the area of prevention of malicious process communication.

7.2 Critical Analysis

There are a number of aspects of the implementation that could be implemented in a more efficient manner – for example implementing at-most-once semantics for message delivery to the authorisation service, and indeed to the test server as well. This would negate any loss of requests, and would affect the overall average time taken to deal with a request.

Further, the use of TCP for conducting intra-computer communications provides overheads that would not be associated with other means of inter-process communications. For example, the use of a 'whiteboard' (a shared space in memory for processes to pass information to each other), as recommended by Microsoft for providing inter-process communications would provide added speed for the system, or developing a proprietary inter-process communication system. The design of such was beyond the scope of the hypothesis however.

The testing carried out could have been conducted on a wider scale, to get a more representative average time and average of lost requests, perhaps conducting 10 or more experiments for each set level, as opposed to the three that were used. This would provide a more normalised set of results, which would allow more complete analysis of the system.

There are a number of issues involved with this system, especially through the use of TCP. First, the messages passed between the client and the service are subject to man-in-the-middle attacks, although these would be harder to accomplish than over network communications, given the local properties of this system. As such, a Diffie-Helman key exchange system through an initial public key tunnel could be used to exchange keys and allow encrypted traffic (much like what happens in TLS). This would be especially required during some of the further work discussed below.

The protocol relies on the requesting process providing all its credentials, and the authorisation service checks those credentials. The certificate must be passed by the requesting process to the service, however the hash signature should be stored along with the registration details of the process on the server. Currently, a malicious entity could alter the program, recompile it, and alter the hash signature file for the program to match the new, altered, program. When the server then requests the hash signature and compares the provided hash signature to the one the server calculates itself, it will return that the hash signatures match. This is correct, but the purpose of including the hash signature is to ensure that the program has not been altered between the time it was registered and the time the corresponding process requests authorisation.

The easiest solution to this is to include the hash signature as part of the registration of the process in the service.

The original hypothesis was:

> "*can an efficient system be created for authenticating and authorising processes for access to the communication channel that will not interfere in the normal workings of the communication*",

and a conclusive answer can now be given. Yes, a system can be created for authenticating and authorising processes for access to a communication channel, and yes efficient implementations can be made that do not interfere in the normal workings of the user, the operating environment, or any other involved parties/processes.

7.3 Further Work

As mentioned above, changing the protocol used to communicate between the requesting process and the authorisation service could be beneficial. However, an added bonus of using the TCP method is that the authorisation service could exist on a central server, and an interface provided on each client to gather the required credentials together. This would provide a central service which would be easier to maintain, and also provide the same results no matter which client a user is logged on to. This is important in a large corporate network, as users may move between computers, and with thousands of computers on the network, would be a nightmare to configure each one for each user.

One way of doing this would be to use remote method invocation, which would use at-most-once semantics, and allow easy integration of an interface on each host to allow the service to analyse the program running to get the hash signature from it. However, RMI has some security vulnerabilities that would have to be explicitly addressed in any implementation, to counter man-in-the-middle attacks, perhaps using TLS or a similar technology.

Lastly, an implementation of the protocol could exist that, once the outcome of the request is determined, will block access to any communication channels (network devices or modems) at a lower level, with the co-operation of the OS. To a certain extent, this would initially look like a process-level user-stateful firewall, however with the added bonus of authentication of the processes, it can prevent modified programs existing in the same location as a process registered to access any devices.

Another aspect that could be added is the inclusion of group support, to allow a slightly less granular control over the system. This would provide systems administrators with the ability to apply general group-wide policies in the system, which would benefit large corporations where entire departments need access to specific processes.

7.4 Theoretical Situation

This system is, on the whole, impractical in modern environments. It requires that the process invoking the request to the authorisation service respects the outcome of that request.

A theoretical situation where this system would be beneficial would be where this system is used for all processes, and they all respect the outcome. In that way, a system administrator can provide process-level access to users, and guarantee that the user will not be able to circumvent the system.

If further work was carried out to provide low level access to block processes, then this would automatically mean that any process that does not use the system, or does not respect the authority of the authorisation service, will already be blocked. This prevents the spread or use of any non-explicitly run processes by the user, such as worms, trojans and, most importantly, zombies and bots.

7.5 General Discussion

The need for a preventative measure to the problem of malicious self propogating code is something that deserves more research, whatever the outcome. Although there is an entire security industry being built around vulnerabilities and flaws in software, there remains a core problem of badly designed and implemented code, due to software developers divorcing themselves from the responsibility of security. (Schneier, “Do We Really Need a Security Industry”, 2007). Solutions closer to the core problem, like the one discussed in this project, is a step in the right direction.

8. References

Aaronson, L, “For love of money – malicious hacking takes an ominous turn”, *IEEE Spectrum*, Volume 42, Number 11, Pages 17-19, IEEE Press, November 2005

Abdalla, M, Bresson, E, Chevassut, O, Möller, B, Pointcheval, D, “Provably Secure Password-Based Authentication in TLS”, *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications security*, pages 35-45, 2006

Abrams, M, “An Integrated Framework for Security and Dependability”, *Proceeding of the 1998 Workshop on New Security Paradigms (NSPW '98)*, pages 22-29, 1989.

Aiello, W, Ioannidis, J, McDaniel, P “Origin Authentication in Interdomain Routing”, *Conference on Computer and Communications Security*, Pages 165-178, ACM Press, 2003

Bennet, T, “A Thread Implementation Project Supporting and Operating Systems Course”, *Journal of Computing Sciences in Colleges*, Vol. 22, No. 5, May 2007

Booth, K, “Authentication of Signatures Using Public Key Encryption”, *Communications of the ACM*, Volume 24, Number 11, pages 772-774, November 1981.

Brainard, J, Juels, A, Rivest, R L, Szydlo, M, Yung, M, “Privacy and Authentication: Fourth-Factor Authentication: Somebody You Know”, *Proceedings of the 13h ACM Conference on Computer and Communications Security CCS 2006*, Pages 168-178, ACM Press, October 2006

Burrows, M, Adabi, M, and Needham, R, “A Logic of Authentication”, *Proceedings of the Royal Society of London, Series A, Mathematical and Physical Sciences*, Vol. 426, No. 1871, pages 233-271, December 1989

Chang, Y, and Chang, C, “An Efficient Authentication Protocol for Mobile Satellite Communication Systems”, *ACM SIGOPS Operating Systems Review*, Vol. 39, No. 1, January 2005.

Dijkstra, E, “Go To Statement Considered Harmful”, *Communications of the ACM*, Vol. 11, No. 3, pages 147-148, March 1968.

Firesmith, D, “Engineering Security Requirements”, *Journal of Object Technology*, Vol. 2, No. 1, February 2003.

Ford, W., P. Hallam-Baker, B. Fox, B. Dillaway, B. LaMacchia, J. Epstein, and J. Lapp, “XML Key Management Specification (XKMS)”, March 2001, latest version is available at http://www.w3.org/TR/xkms/

Gouda, M, Elnozahy, E, Huang, C, McGuire, T, “Hop Integrity in Computer Networks”, *IEEE/ACM Transactions on Networking*, Vol. 10, No. 3, pages 308-319, June 2002.

Gürses, S, Jahnke, J, Obry, C, Onabajo, A, Santen, T, Price, M, “Eliciting Confidentiality Requirements in Practice”, *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 101-116, 2005.

Hasan, J, Cadambi, S, Jakkula, V, and Chakradhar, S, “Chisel: A Storage-efficient, Collision-free Hash-based Network Processing Architecture”, *Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 203-215, 2006

Housley, R, Polk, W, Ford, W, Solo, D, “Internet X.509 Public Key Infrastructure Certificate and CRL Profile draft-ietf-pkix-new-part1-12.txt”, http://www.ietf.org/internet-drafts/draft-ietf-pkix-new-part1-12.txt, April 2002, accessed 17th April 2007.

Lampson, B, Abadi, M, Burrows, M, and Wobber, E, “Authentication in Distributed Systems: Theory and Practice”, *ACM Trans. Computer Systems*, Vol. 10, No. 4, November 1992

Littlewood, B, and Strigini, L, “Redundancy and Diversity in Security”, *Proceeding of European Symposium on Research in Computer Security Conference*, Pages 423-438, 2004

Krings, A, Oman, P and Azadmanesh, A, “Security and Survivability of Networked Systems”, *Proceedings of the 38th Annual Hawaii International Conference on*

System Sciences (HICSS'05), Pages 308, 2005

Kohl, J, Neuman, C, and Ts'o, T, "The Evolution of the Kerberos Authentication Service", *Distributed Open Systems*, pages 78-94, September 1994.

McDowell, K, "Now that we are all so well-educated about spyware, can we put the bad guys out of business?", *Proceedings of the 34th annual ACM SIGUCCS conference on User services*, Pages 235-239, ACM Press, 2006

Microsoft, "DateTime.Ticks Property (System)", *MSDN*, accessed 5th May 2007 at http://msdn2.microsoft.com/en-us/library/system.datetime.ticks.aspx

Needham, R, and Schroeder, M, "Using Encryption for Authentication in Large Networks of Computers", *Communcations of the ACM*, Vol. 2, No. 1, December 1978

O'Donnell, A J, and Sethu, H, "Network security: On Achieving Software Diversity for Improved Network Security using Distributed Coloring Algorithms" *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 121-131, 2004

Ravindran, K, and Lin, X, "Structural Complexity and Execution Efficiency of Distributed Application Protocols", *Conference Proceedings on Communications Architectures, Protocols and Applications*, pages 160-169, 1993

Schneier, B, "Do We Really Need a Security Industry?", *Wired.com*, accessed 5th March 2007 at http://www.wired.com/politics/security/commentary/securitymatters/2007/05/securitymatters_0503

Siponen, M, and Oinas-Kukkonen, H, "A Review of Information Security Issues and Respective Research Contributions", *ACM SIGMIS Database*, Vol. 38, No. 1, pages 60-80, February 2007

Steiner, G, Neuman, C, Schiller, J, *Kerberos: An Authentication Service for Open Network Systems*, March 1988

Syverson, P, and van Oorschot, P, *A Unified Cryptographic Protocol Logic*, NRL

Publication 5540-227, Naval Research Lab, 1996

Taipei, T, "Security protocols: Certified mailing lists", *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security* Pages 46-58, ACM Press, March 2006

Thompson, M, Essiari, A, and Mudumbai, S, "Certificate-Based Authorization in a PKI Environment", *ACM Transactions on Information and System Security*, Vol. 6, No. 4, pages 566-588, November 2003

Tully, C, "Software Process Models and Programs: Observations on Their Nature and Context", *Proceedings of the 4th Interntational Software Process Workshop on Representing and Enacting the Software Process*, pages 159-162, 1989.

Waldo, J, *A Formal Semantics for the Logic of Authentication*, Sun Microsystems Laboratories, Burlington, USA

Wahlmacher, P, "Digital Certificates: A Survey of Revocation Methods", *Proceedings of the 2000 ACM Workshops on Multimedia*, pages 111-114, 2000.

Woo, T and Lam, S, "Authentication for Distributed Systems", *Computer*, Volume 25, Number 1, pages 39-52, January 1992

Xie, M, Yin, H, and Wang, H, "Privacy and Authentication: An Effective Defence Against Email Spam Laundering", *Proceedings of the 13th ACM conference on Computer and Communications Security*, pages 179-190, 2006

Zhuang, X, Zhang, T, Lee, H S, and Pande, S, "Reliability and Security: Hardware Assisted Control Flow Obfuscation For Embedded Processors", *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 292-307, 2004

9. APPENDICES

A. AUTHORISATIONSERVER.CS

```
using System;
using System.Collections;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Threading;

namespace ProcessAuthorisation {

  public class AuthorisationServer {

    /* Listener for an incoming client connection
     */
    static TcpListener listener;

    /* List of users
     */
    ArrayList users = new ArrayList();

    /* The server certificate
     */
    static string servCert;

    /* Main method. Creates a new AuthorisationServer and starts
listening for incoming connection
     */
    public static void Main (string[] args) {

      AuthorisationServer server = new AuthorisationServer();

      listener = new TcpListener(9999);
      listener.Start();

      // For each new connection, create a new ServerService to
provide the service
      for(;;) {
        Socket soc = listener.AcceptSocket();
        AuthorisationServerService client = new
AuthorisationServerService(server, soc);
        client.BeginService();
      }

    }

    /* Test constructor.
     */
     public AuthorisationServer() {

        initServCert();

        // Create a new test User
        User U = new User("kasyx");


U.AddEl("/home/kasyx/backup/uni/honours/Code/Submission/TestClient
/TestClient.exe");
        users.Add(U);

     }

     /* Check to see if a program run by a user is allowed access
to the network
      */
     public bool IsAllowed (string procName, string userName) {

        bool result = false;

        foreach (User current in users) {

           if (current.GetName() == userName) {

              if (current.ExistsEl(procName)) {
                 result = true;
              }

           }

        }

        return result;

     }

     /* Facility to initialise the certificate
      */
     public void initServCert() {

        servCert = "ServCert";

     }

     /* Retrieve the server certificate
      */
     public string getServCert() {

        return servCert;

     }
```

```
    }

}
```

B. USER.CS

```
using System;
using System.Collections;

namespace ProcessAuthorisation {

  public class User {

    /* The name of the user
     */
    string name;

    /* A list of processes this user is allowed to run
     */
    ArrayList allowed;

    /* Constructor.
     *
     * String name - the username of the user
     */
    public User (string name) {

      this.name = name;
      allowed = new ArrayList();

      allowed.Add("test");

    }

    /* Add a process to the list
     */
    public bool AddEl (string subject) {

      try {
        allowed.Add(subject);
        return true;
      } catch (Exception e) {
        System.Console.WriteLine(e);
        return false;
      }

    }

    /* Remove a process from the list
     */
    public bool RemEl (string subject) {

      try {
        allowed.Remove(subject);
        return true;
        } catch (Exception e) {
          System.Console.WriteLine(e);
          return false;
        }

      }

      /* Check to see if a process exists in the list
       */
      public bool ExistsEl (string subject) {

        bool result = false;

        foreach (string current in allowed) {
          if (current == subject) {
            result = true;
          }
        }
        return result;

      }

      /* Return the name of this object
       */
      public string GetName () {

        return this.name;

      }
  }
}
```

C. AuthorisationServerService.cs

```
using System;
using System.Net;
using System.Net.Sockets;
using System.IO;
using System.Collections;
using System.Threading;
using Hashing;

namespace ProcessAuthorisation {

  public class AuthorisationServerService {

     /* Socket for the connection
      */
     Socket soc;

     /* UserObjectServer for the system
      */
     AuthorisationServer uos;

     /* Hashing wrapper
      */
     SHAHashWrapper hw;

     /* Default constructor. Constructs a ServerService to carry
out the checks
      */
     public AuthorisationServerService(AuthorisationServer uosrv,
Socket s) {

       this.soc = s;
       this.uos = uosrv;
       hw = new SHAHashWrapper();

     }

     /* Begins the service on a new thread, allowing the
originating process to continue its work
      */
     public void BeginService() {

       Thread t = new Thread(new ThreadStart(Service));
       t.Start();

     }

     /* The service itself
      */
     private void Service () {
```

```
try {

  // Create new communication stream for the connection
  Stream s = new NetworkStream(soc);

  // Create reading and writing mechanisms for the stream
  StreamReader sr = new StreamReader(s);
  StreamWriter sw = new StreamWriter(s);

  sw.AutoFlush = true;

  // Begin the protocol

  // State: Idle

  // Incoming Event: AuthReq
  string incoming = sr.ReadLine();

  if (incoming == "Abort")
    goto end;

  if (incoming != "AuthReq") {
    sw.WriteLine("Abort");
    goto end;
  }

  // Outgoing Event: AuthReqAck
  sw.WriteLine("AuthReqAck");

  // New State: WaitAuthReqOK

  // Incoming Event: AuthReqOK
  incoming = sr.ReadLine();

  if (incoming == "Abort")
    goto end;

  if (incoming != "AuthReqOK") {
    sw.WriteLine("Abort");
    goto end;
  }

  // Outgoing Event: SendProcName
  sw.WriteLine("SendProcName");

  // New State: WaitProcName

  // Incoming Event: Data($PROC_NAME)
  incoming = sr.ReadLine();

  if (incoming == "Abort")
```

```
            goto end;

          string procName = incoming;

          // Outgoing Event: SendUserName
          sw.WriteLine("SendUserName");

          // New State: WaitUserName

          // Incoming Event: Data($USER_NAME)
          incoming = sr.ReadLine();

          if (incoming == "Abort")
            goto end;

          string userName = incoming;

          // Outgoing Event: SendProcCert
          sw.WriteLine("SendProcCert");

          // New State: WaitProcCert

          // Incoming Event: Data($PROC_CERT)
          incoming = sr.ReadLine();

          if (incoming == "Abort")
            goto end;

          string procCert = incoming;

          // Outgoing Event: SendProcHash
          sw.WriteLine("SendProcHash");

          // New State: WaitProcHash

          // Incoming Event: Data($PROC_HASH)
          incoming = sr.ReadLine();

          if (incoming == "Abort")
            goto end;

          string procHash = incoming;

          // Outgoing Event: ^p01: ProcCheckInvalid
          if (!checkProcCred(procName, procCert, procHash)) {
            sw.WriteLine("ProcCheckInvalid");
            Log(procName, userName, procCert, procHash,
uos.getServCert(), "0");
            goto end;
          }

          // Outgoing Event: p01: ProcCheckValid
          sw.WriteLine("ProcCheckValid");

          // New State: WaitSendServCert

          // Incoming Event: SendServCert
          incoming = sr.ReadLine();

          if (incoming == "Abort")
            goto end;

          if (incoming != "SendServCert") {
            sw.WriteLine("Abort");
            goto end;
          }

          // Outgoing Event: Data($SERV_CERT)
          sw.WriteLine(uos.getServCert());

          // New State: WaitServCheck

          // Incoming Event: ServCheckInvalid
          incoming = sr.ReadLine();

          if (incoming == "Abort")
            goto end;

          if (incoming == "ServCertInvalid") {
            Log(procName, userName, procCert, procHash,
uos.getServCert(), "1");
            goto end;
          }

          // Incoming Event: ServCheckValid
          if (incoming != "ServCertValid") {
            sw.WriteLine("Abort");
            goto end;
          }

          // Outgoing Event: ServCertValidAck
          sw.WriteLine("ServCheckValidAck");

          // New State: WaitAuthCheck

          // Incoming Event: AuthCheck
          incoming = sr.ReadLine();

          if (incoming == "Abort")
            goto end;

          if (incoming != "AuthCheck") {
            sw.WriteLine("Abort");
            goto end;
          }

          // Outgoing Event: p02: AuthAck
          if (uos.IsAllowed(procName, userName)) {
            sw.WriteLine("AuthAck");
            Log(procName, userName, procCert, procHash,
uos.getServCert(), "2");
          } else {
            // Outgoing Event: ^p01: AuthRef
            sw.WriteLine("AuthRef");
            Log(procName, userName, procCert, procHash,
uos.getServCert(), "3");
          }

          end:

          // Close the reader and writer
          s.Close();

        } catch (Exception e) {

          // Display an error if the stream cannot be created
          Console.WriteLine(e.Message);

        }

        // Close the stream
        soc.Close();

      }

      /* Facilities for logging
       */
      public void Log (string pn, string un, string pc, string ph,
string sc, string c) {

        Console.WriteLine("{0}:{1}:{2}:{3}:{4}:{5}", pn, un, pc,
ph, sc, c);

      }

      /* Check the credentials of the process
       */
      public bool checkProcCred(string procName, string procCert,
string procHash) {

        if (hw.GetHash(procName) == procHash)
          return true;

        return false;
      }
```

```
    }

}
```

D. SHAHashWrapper.cs

```
using System;
using System.Text;
using System.Security.Cryptography;
using System.IO;

namespace Hashing {

  public class SHAHashWrapper {

    SHA512Managed SHhash = new SHA512Managed();

    public SHAHashWrapper () {}

    public string GetHash (string fsl) {

      // Open the file at the location given - read only, we
aren't wanting to change anything
      FileStream fs = new FileStream(fsl, FileMode.Open,
FileAccess.Read);

      byte[] Hash;

      Hash = SHhash.ComputeHash(fs);

      // Convert into base 16
      string hashValue = "";

      foreach(byte b in Hash) {
        hashValue += String.Format("{0:x2}", b);
      }

      // Close the file
      fs.Close();

      return hashValue;

    }

  }

}
```

E. ClientService.cs

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;

namespace ProcessAuthorisationClient {

  public class AuthorisationClient {

     /* The username running the process
      */
     string userName;

     /* The process name
      */
     string procName;

     /* The TCP connection
      */
     TcpClient client;

     /* The auth code
      */
     int auth;

     /* The process certificate
      */
     string procCert;

     /* The process hash
      */
      string procHash;

     /* Constructor
      *
      * Fetches the username and process name from the system,
then fetches the certificate and hash based on
      * the process location. Creates a new connection to the
server.
      */
     public AuthorisationClient() {

       userName =
System.Security.Principal.WindowsIdentity.GetCurrent().Name.ToStri
ng();
       procName =
System.Reflection.Assembly.GetExecutingAssembly().Location;
       procCert = getProcCert();
       procHash = getProcHash();
```

```
        auth = AuthCheck();

      }

      /* Return the current authorisation
       */
      public int GetAuth() {

        return this.auth;

      }

      /* Allow for rechecking of the authorisation
       */
      public void RecheckAuth() {

        this.auth = AuthCheck();

      }

      /* Checks for authorisation with the server.
       * Returns a value depending on the results of the checks
       */
      public int AuthCheck() {

        int tempAuth = 0;

        try {

          // Connect to the AuthorisationServer
          client = new TcpClient("127.0.0.1", 9999);

          // Create a communication stream
          Stream s = client.GetStream();

          // Create a reader and writer for the communication
stream
          StreamReader sr = new StreamReader(s);
          StreamWriter sw = new StreamWriter(s);

          sw.AutoFlush = true;

          // Begin protocol

          // State: Idle

          // Outgoing Event: AuthReq
          sw.WriteLine("AuthReq");

          // New State: WaitAuthReqAck

          // Incoming Event: AuthReqAck
```

```
string incoming = sr.ReadLine();

if (incoming == "Abort")
  goto end;

if (incoming != "AuthReqAck") {
  sw.WriteLine("Abort");
  goto end;
}

// Outgoing Event: AuthReqOK
sw.WriteLine("AuthReqOK");

// New State: WaitSendProcName

// Incoming Event: SendProcName
incoming = sr.ReadLine();

if (incoming == "Abort")
  goto end;

if (incoming != "SendProcName") {
  sw.WriteLine("Abort");
  goto end;
}

// Outgoing Event: Data($PROC_NAME)
sw.WriteLine(procName);

// New State: WaitSendUserName

// Incoming Event: SendUserName
incoming = sr.ReadLine();

if (incoming == "Abort")
  goto end;

if (incoming != "SendUserName") {
  sw.WriteLine("Abort");
  goto end;
}

// Outgoing Event: Data($USER_NAME)
sw.WriteLine(userName);

// New State: WaitSendProcCert

// Incoming Event: SendProcCert
incoming = sr.ReadLine();

if (incoming == "Abort")
  goto end;
```

```
if (incoming != "SendProcCert") {
  sw.WriteLine("Abort");
  goto end;
}

// Outgoing Event: Data ($PROC_CERT)
sw.WriteLine(procCert);

// New State: WaitSendProcHash

// Incoming Event: SendProcHash
incoming = sr.ReadLine();

if (incoming == "Abort")
  goto end;

if (incoming != "SendProcHash") {
  sw.WriteLine("Abort");
  goto end;
}

// Outgoing Event: Data($PROC_HASH)
sw.WriteLine(procHash);

// New State: WaitValid

// Incoming Event: ProcCheckValid
incoming = sr.ReadLine();

if (incoming == "Abort")
  goto end;

// Incoming Event: ProcCheckInvalid
if (incoming == "ProcCheckInvalid") {
  tempAuth = 0;
  goto end;
}

// Outgoing Event: SendServCert
sw.WriteLine("SendServCert");

// New State: WaitServCert

// Incoming Event: Data($SERV_CERT)
incoming = sr.ReadLine();

if (incoming == "Abort")
  goto end;

string servCert = incoming;
```

```
// Outgoing Event: ^p01: ServCheckInvalid
if (!ServCheckCred(servCert)) {
   sw.WriteLine("ServCheckInvalid");
   tempAuth = 1;
   goto end;
}

// Outgoing Event: p01: ServCheckValid
sw.WriteLine("ServCertValid");

// New State: WaitValidAck

// Incoming Event: ServCheckValidAck
incoming = sr.ReadLine();

if (incoming == "Abort")
   goto end;

if (incoming != "ServCheckValidAck") {
   sw.WriteLine("Abort");
   goto end;
}

// Outgoing Event: AuthCheck
sw.WriteLine("AuthCheck");

// New State: WaitAuth

//  Incoming Event: AuthAck
incoming = sr.ReadLine();

if (incoming == "Abort")
   goto end;

if (incoming != "AuthAck" && incoming != "AuthRef") {
   sw.WriteLine("Abort");
   goto end;
}

if (incoming == "AuthAck") {
   tempAuth = 2;
   goto end;
}

// Incoming Event: AuthRef
if (incoming == "AuthRef") {
   tempAuth = 3;
   goto end;
}

end:
        // Close the communication stream
        s.Close();

      } catch (Exception e) {

        // Display any exceptions
        Console.WriteLine(e.Message);

      }

      // Close the connection
      client.Close();

      // Return the authorisation code
      return tempAuth;

    }

    /* Check the credentials of the server
     */
    public bool ServCheckCred (string servCert) {
      return true;
    }

    /* Retrieve the certificate of the process
     */
    public string getProcCert() {

      return "My certificate";

    }

    /* Retrieve the hash signature of the process
     */
    public string getProcHash() {

      // Convert the current path to a char array
      char[] procNameAsChar = procName.ToCharArray();

      bool found = false;
      int pos = 0;

      // While not found the period denoting file extension
      while (!found) {
        // Iterate backward through the array
        for (int i = procNameAsChar.Length - 1; i >= 0; i--) {
          // Mark the position of the period, and say it's been
found
          if (procNameAsChar[i] == '.') {
            found = true;
            pos = i;
          }
```

```
         }
       }

       // Create a new char array with size up to the position of
the period
       char[] hashLocAsChar = new char[pos];

       // Transfer the details accross
       for (int i = 0; i < pos; i++) {
         hashLocAsChar[i] = procNameAsChar[i];
       }

       // Convert the new location to a string
       string hashLoc = new string(hashLocAsChar);
       // Append the new extension
       hashLoc += ".sha512";

       // Read the hash signature stored in the file in
       FileStream file = new FileStream(hashLoc, FileMode.Open,
FileAccess.Read);
       StreamReader sr = new StreamReader(file);
       procHash = sr.ReadToEnd();
       procHash = procHash.Replace("\n","").Replace("\r","");
       sr.Close();
       file.Close();

       // Return the hash signature
       return procHash;

     }

   }

}
```

F. TestClient.cs

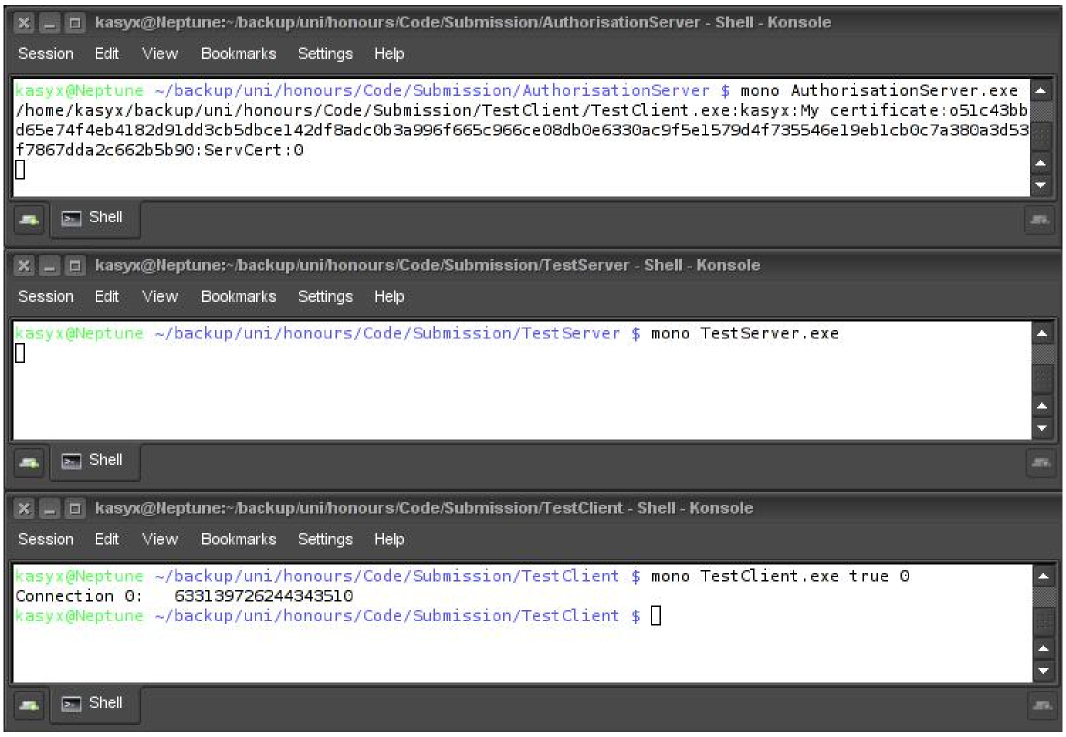
```
using System;
using System.Threading;
using System.Net;
using System.Net.Sockets;
using System.IO;
using ProcessAuthorisationClient;

namespace ProcessAuthorisationTestingClient {

  public class TestClient {

    /* Main method. Takes two parameters: whether or not to use
the AuthorisationServer,
     * and how many connections to make.
     */
    public static void Main (String[] args) {

      bool use = false;
      int connections = Convert.ToInt32(args[1]);;

      if (args[0] == "true")
        use = true;

      TestClientService testing;

      for (int i = 0; i <= connections; i++) {
        testing = new TestClientService(use, i);
        testing.BeginService();
      }

    }

  }

  /* The service for the test message
   */
  public class TestClientService {

    bool use;
    int no;

    /* Constructor. Initialises whether to use the
AuthorisationServer or not,
     * and the number of this connection
     */
    public TestClientService (bool u, int n) {

      use = u;
      no = n;
    }

    /* Start the service
     */
    public void BeginService() {

      Thread t = new Thread(new ThreadStart(Service));
      t.Start();

    }

    /* The service itself
     */
    public void Service() {

      // Display to console the time when the connection was
initialised
      Console.WriteLine("Connection " + no + ":\t" +
DateTime.Now.Ticks);

      // If we use the AuthServer
      if (use) {

        AuthorisationClient comms = new AuthorisationClient();

        if (comms.GetAuth() == 2) {
          TcpClient client = new TcpClient("127.0.0.1", 10001);

          // Create a communication stream
          Stream s = client.GetStream();

          // Create a reader and writer for the communication
stream
          StreamWriter sw = new StreamWriter(s);

          sw.AutoFlush = true;

          sw.WriteLine(no);

          s.Close();
          client.Close();
        }

      // If we don't
      } else {
        TcpClient client = new TcpClient("127.0.0.1", 10001);

        // Create a communication stream
        Stream s = client.GetStream();

        // Create a reader and writer for the communication
stream
        StreamWriter sw = new StreamWriter(s);

        sw.AutoFlush = true;

        sw.WriteLine(no);

        s.Close();
        client.Close();
      }
    }
  }
}
```

G. TESTSERVER.CS

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Threading;

namespace ProcessAuthorisationTestingServer {

  public class TestServer {

     /* Main method. No arguments expected
      */
     public static void Main (String[] args) {

        // Start a new listener
        TcpListener listener = new TcpListener(10001);
        listener.Start();

        // For every connection that comes in, start a new service
for it
        for(;;) {
          Socket soc = listener.AcceptSocket();
          TestServerService client = new TestServerService(soc);
          client.BeginService();
        }

     }

  }

  /* The service for the server to provide
   */
  public class TestServerService {

     Socket sock;

     /* Constructor. Assigns the socket to this object
      */
     public TestServerService(Socket soc) {

        sock = soc;

     }

     /* Start the service
      */
     public void BeginService() {

        Thread t = new Thread(new ThreadStart(Service));
      t.Start();

    }

    /* The service itself
     */
    public void Service() {

      // Create new communication stream for the connection
      Stream s = new NetworkStream(sock);

      // Create reading and writing mechanisms for the stream
      StreamReader sr = new StreamReader(s);

      string incoming = sr.ReadLine();

      // Print to console the time of the connection, plus the
connection number
      Console.WriteLine("Connection " + incoming + ":\t" +
DateTime.Now.Ticks);

      // Close the connection
      s.Close();

    }

  }

}
```

H. Protocol Testing Results

The image below shows the output of the first of the tests of the protocol, where the credentials (hash signature) of the requesting process is invalid. It can be seen that the AuthorisationServer, (top Konsole), has returned a value '0', indicating that the process has invalid credentials, and that the requesting process has honoured this by not proceeding any further and attempting to make a connection to its own server (middle Konsole).


kasyx@Neptune:~/backup/uni/honours/Code/Submission/AuthorisationServer - Shell - Konsole
Session Edit View Bookmarks Settings Help
Shell

```
kasyx@Neptune ~/backup/uni/honours/Code/Submission/AuthorisationServer $ mono AuthorisationServer.exe
/home/kasyx/backup/uni/honours/Code/Submission/TestClient/TestClient.exe:kasyx:My certificate:o51c43bb
d65e74f4eb4182d91dd3cb5dbce142df8adc0b3a996f665c966ce08db0e6330ac9f5e1579d4f735546e19eb1cb0c7a380a3d53
f7867dda2c662b5b90:ServCert:0
```

kasyx@Neptune:~/backup/uni/honours/Code/Submission/TestServer - Shell - Konsole
Session Edit View Bookmarks Settings Help
Shell

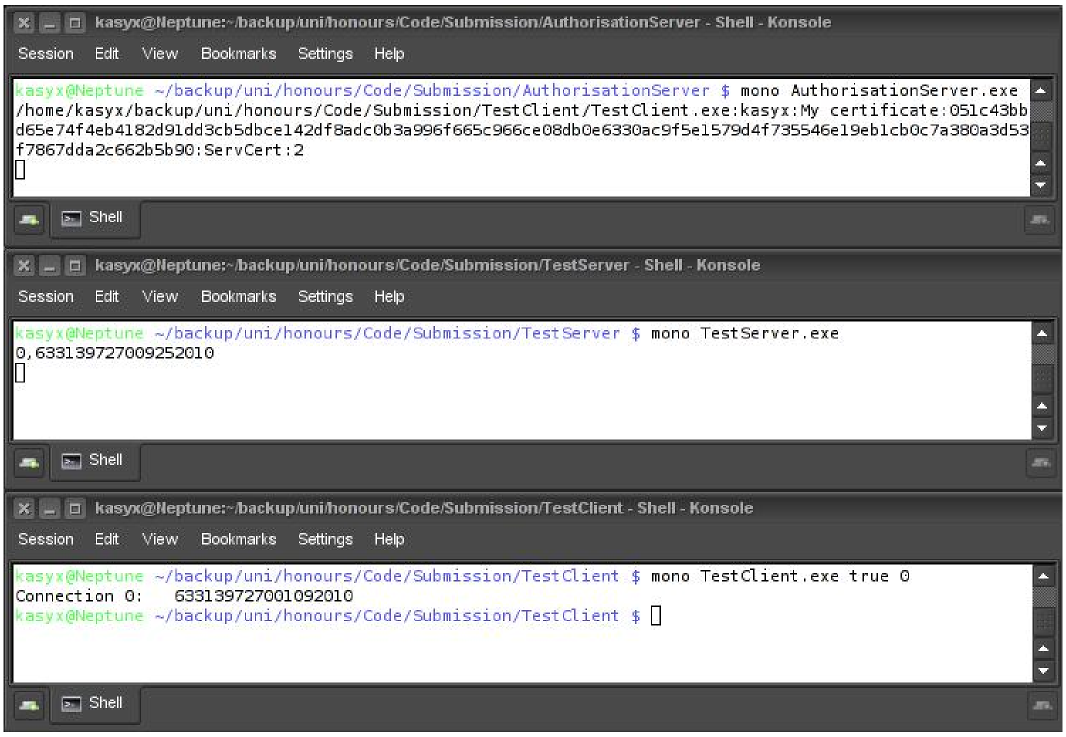
```
kasyx@Neptune ~/backup/uni/honours/Code/Submission/TestServer $ mono TestServer.exe
```

kasyx@Neptune:~/backup/uni/honours/Code/Submission/TestClient - Shell - Konsole
Session Edit View Bookmarks Settings Help
Shell

```
kasyx@Neptune ~/backup/uni/honours/Code/Submission/TestClient $ mono TestClient.exe true 0
Connection 0:   633139726244343510
kasyx@Neptune ~/backup/uni/honours/Code/Submission/TestClient $
```

Fig H.1: The output of three programs, testing incorrect credentials of a registered process

Next, a test was carried out for a correctly registered process with correct credentials.


kasyx@Neptune:~/backup/uni/honours/Code/Submission/AuthorisationServer - Shell - Konsole
Session Edit View Bookmarks Settings Help
Shell

```
kasyx@Neptune ~/backup/uni/honours/Code/Submission/AuthorisationServer $ mono AuthorisationServer.exe
/home/kasyx/backup/uni/honours/Code/Submission/TestClient/TestClient.exe:kasyx:My certificate:051c43bb
d65e74f4eb4182d91dd3cb5dbce142df8adc0b3a996f665c966ce08db0e6330ac9f5e1579d4f735546e19eb1cb0c7a380a3d53
f7867dda2c662b5b90:ServCert:2
```

kasyx@Neptune:~/backup/uni/honours/Code/Submission/TestServer - Shell - Konsole
Session Edit View Bookmarks Settings Help
Shell

```
kasyx@Neptune ~/backup/uni/honours/Code/Submission/TestServer $ mono TestServer.exe
0,633139727009252010
```

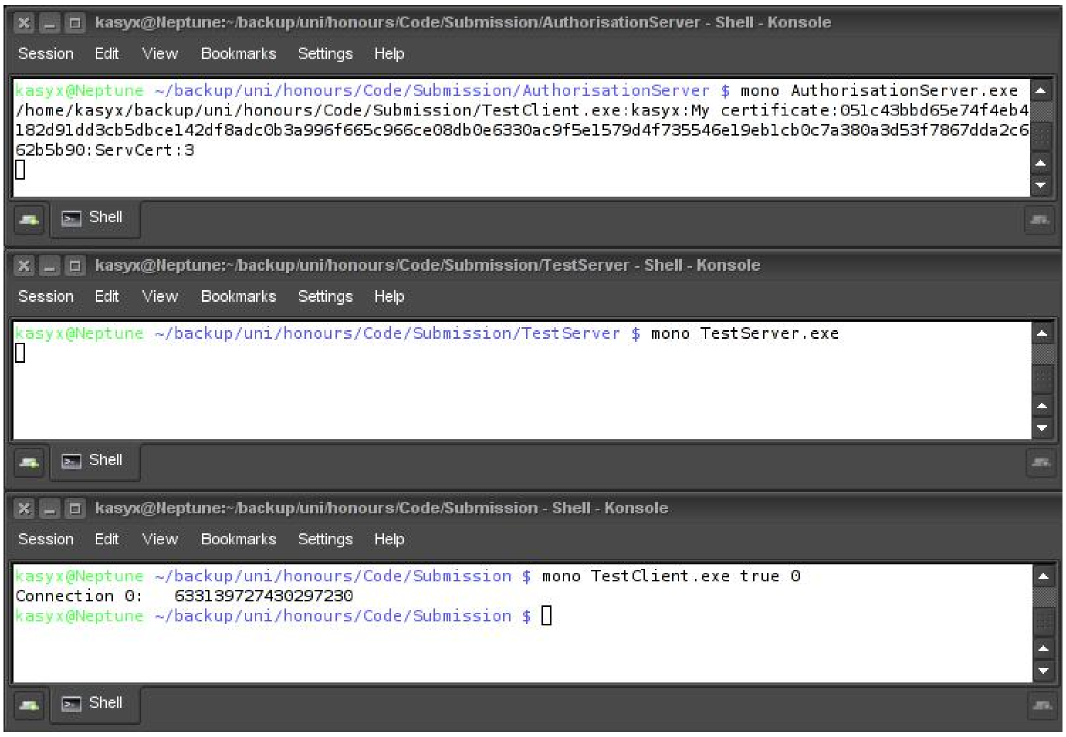
kasyx@Neptune:~/backup/uni/honours/Code/Submission/TestClient - Shell - Konsole
Session Edit View Bookmarks Settings Help
Shell

```
kasyx@Neptune ~/backup/uni/honours/Code/Submission/TestClient $ mono TestClient.exe true 0
Connection 0:    633139727001092010
kasyx@Neptune ~/backup/uni/honours/Code/Submission/TestClient $
```

Fig H.2: The output of three programs, testing correct credentials of a registered process

As can be seen, the AuthorisationServer has responded with code '2', indicating that the process is registered, and has supplied credentials that were verified. The requesting process has then gone on to communicate with its server.

Lastly, a non-registered process with correct credentials was tested.


kasyx@Neptune:~/backup/uni/honours/Code/Submission/AuthorisationServer - Shell - Konsole
Session Edit View Bookmarks Settings Help
Shell
kasyx@Neptune:~/backup/uni/honours/Code/Submission/TestServer - Shell - Konsole
Session Edit View Bookmarks Settings Help
Shell
kasyx@Neptune:~/backup/uni/honours/Code/Submission - Shell - Konsole
Session Edit View Bookmarks Settings Help
Shell

```
kasyx@Neptune ~/backup/uni/honours/Code/Submission/AuthorisationServer $ mono AuthorisationServer.exe
/home/kasyx/backup/uni/honours/Code/Submission/TestClient.exe:kasyx:My certificate:051c43bbd65e74f4eb4
182d91dd3cb5dbce142df8adc0b3a996f665c966ce08db0e6330ac9f5e1579d4f735546e19eb1cb0c7a380a3d53f7867dda2c6
62b5b90:ServCert:3
```

```
kasyx@Neptune ~/backup/uni/honours/Code/Submission/TestServer $ mono TestServer.exe
```

```
kasyx@Neptune ~/backup/uni/honours/Code/Submission $ mono TestClient.exe true 0
Connection 0:    633139727430297230
kasyx@Neptune ~/backup/uni/honours/Code/Submission $
```

Fig. H.3: The output of three programs, tesing correct crednetials of a non-registered process

Again, the requesting process has honoured the response of the Authorisation server, and has not proceeded further to talk to its own server.

I. Collated Data Results

These results are the collated results used for presenting the graphs in the Evaluation section.

	Basic System		
	Avg. Time for Connection		
No. Concurrent Connections		Without System	With System
	10	2886741.33	11378845.87
	20	13955496.53	24782114.13
	30	15436526.93	22342337.42
	40	54632902.64	22597289.6
	50	28344494.93	29578591.57
	60	60913124.94	28558197.33
	70	98700850.59	39858385.07
	80	76291872.61	34837311.47
	90	89574634.34	32145007.88
	100	28838503.25	65354806.61
		Without System	With System
	100	28838503.25	65354806.61
	200	244365034.76	144136249.47
	300	198598710.57	138843107.04
	400	208089299.14	149535212.25
	500	184572183.7	128062590.03
	600	229256761.64	122005575.18
	700	111281790.88	97205371.48
	800	173217842.31	97227073.27
	900	106630033.35	87426449.65
	1000	132601317.21	110524770.6

Table I.1: The average time for a connection on the basic system, with and without the authorisation system being used

	Basic System		
	Avg. Lost Requests		
No. Concurrent Connections		Without System	With System
	10	0.00%	0.00%
	20	0.00%	0.00%
	30	0.00%	0.00%
	40	3.33%	0.00%
	50	0.00%	0.00%
	60	1.67%	0.00%
	70	0.00%	0.00%
	80	0.42%	0.00%
	90	3.70%	0.00%
	100	0.00%	0.00%
		Without System	With System
	100	0.00%	0.00%
	200	14.00%	10.50%
	300	24.22%	9.22%
	400	28.08%	17.08%
	500	19.67%	11.20%
	600	26.00%	9.67%
	700	5.52%	1.14%
	800	22.58%	4.79%
	900	10.37%	8.70%
	1000	13.53%	7.47%

Table I.2: The average percentage of lost requests on the basic system, with and without the authorisation system being used

	Avg. Time for Connection		
No. Concurrent Connections		Without System	With System
	10	1460988.8	14012142.93
	20	17699985.07	15654233.6
	30	22683370.67	22515300.98
	40	32031731.2	28210532.27
	50	36667302.56	32946830.8
	60	52535763.2	32640943.64
	70	102691164.4	31999305.14
	80	186169428.8	35836414.93
	90	156825560.65	51303491.04
	100	151412675.84	138261493.52
		Without System	With System
	100	151412675.84	138261493.52
	200	228362038.13	155401289.18
	300	218223691.08	108313842.14
	400	284425872.69	130936271.77
	500	299034587.2	104619099.07
	600	187108671.75	71249226.04
	700	210824254.67	336951462.14
	800	218213416.63	74663847.63
	900	171729913.82	195774548.65
	1000	151412536.08	89026607.61

Table I.3: The average time for a connection on a usual system, with and without the authorisation system being used

	Usual System		
	Avg. Lost Packets		
No. Concurrent Connections		Without System	With System
	10	6.67%	0.00%
	20	0.00%	0.00%
	30	0.00%	0.00%
	40	0.00%	0.00%
	50	4.67%	0.67%
	60	0.00%	0.00%
	70	0.48%	0.00%
	80	0.00%	0.00%
	90	0.00%	0.37%
	100	0.00%	3.00%
		Without System	With System
	100	0.00%	3.00%
	200	1.33%	11.00%
	300	31.33%	0.89%
	400	36.33%	13.42%
	500	43.20%	0.60%
	600	11.39%	2.61%
	700	28.48%	15.62%
	800	35.00%	0.00%
	900	23.78%	18.70%
	1000	17.30%	1.57%

Table I.4: The average percentage of lost requests on a usual system, with and without the authorisation system being used

J. AuthProtoServ.st

```
# CW Client
# Ewan Gunn - 05013468


# Define the incoming events

define eventsin
begin

      AuthReq           Client      /* Initial request */
      AuthReqOK   Client      /* End of three-way-handshake */
      Data        Client      /* A required piece of data */
      SendServCert      Client      /* A request for the
certificate of the server */
      ServCheckValid    Client      /* The client has validated
the credentials of the server */
      ServCheckInvalid  Client      /* The client has rejected the
credentials of the server */
      AuthCheck   Client      /* Request the authorisation code */
      Abort       Client      /* Abort the request */

end


# Define the outgoing events if required

define eventsout
begin

      AuthReqAck  Client      /* Acknowledge the authorisation
request */
      SendProcName      Client      /* Send the name of the
process */
      SendUserName      Client      /* Send the username running
the process */
      SendProcCert      Client      /* Send the certificate of the
process */
      SendProcHash      Client      /* Send the hash signature of
the process */
      ProcCheckValid    Client      /* The credentials of the
process have been validated */
      ProcCheckInvalid  Client      /* The credentials of the
process have been rejected */
      Data  Client      /* A required piece of data */
      ServCheckValidAck Client      /* Acknowledge the validity of
the server credentials */
      AuthRef     Client      /* The request has been refused */
      AuthAck     Client      /* The request has been accepted */
```

```
end


#  Define the states

define states
begin

      Idle  /* The server is inactive */
      WaitAuthReqOK     /* Wait for the AuthReqOK event */
      WaitProcName      /* Wait for the data event corresponding
to the process name */
      WaitUserName      /* Wait for the data event corresponding
to the username */
      WaitProcCert      /* Wait for the data event corresponding
to the process certificate */
      WaitProcHash      /* Wait for the data event corresponding
to the process hash signature */
      WaitSendServCert  /* Wait for the SendServCert event */
      WaitServCheck     /* Wait for the response of the
credentials check */
      WaitAuthCheck     /* Wait for the AuthCheck event */

end


# Define the predicates if required

define predicates
begin

      p01         /* If TRUE credentials of the client are valid
*/
      p02         /* If TRUE the process is allowed to access the
network */

end


# Define the actions if required

define actions
begin
# In UNIX programming, popen is a function similar to fopen,
except that the
# first argument is a UNIX command

      [0]   /* Log the request */


end
```

```
#  Define the state variable name and the initial state

define statevarname  ServerVar     initialstate Idle

table ServerTable
begin

      state Idle
            eventin AuthReq AuthReqAck WaitAuthReqOK

      state WaitAuthReqOK
            eventin AuthReqOK SendProcName WaitProcName
            eventin Abort Idle

      state WaitProcName
            eventin Data SendUserName WaitUserName
            eventin Abort Idle

      state WaitUserName
            eventin Data SendProcCert WaitProcCert
            eventin Abort Idle

      state WaitProcCert
            eventin Data SendProcHash WaitProcHash
            eventin Abort Idle

      state WaitProcHash
            eventin Data p01: ProcCheckValid WaitSendServCert;
                   ^p01: [0] ProcCheckInvalid Idle

      state WaitSendServCert
            eventin SendServCert Data WaitServCheck
            eventin Abort Idle

      state WaitServCheck
            eventin ServCheckValid ServCheckValidAck WaitAuthCheck
            eventin ServCheckInvalid [0] Idle

      state WaitAuthCheck
            eventin AuthCheck p02: [0] AuthAck Idle;
                              ^p02: [0] AuthRef  Idle



 end
```

K. AUTHPROTOCLIENT.ST

```
# CW Client
# Ewan Gunn - 05013468


# Define the incoming events

define eventsin
begin


      AuthReqAck  Server      /* Acknowledgement of the AuthReq
event */
      SendProcName      Server      /* Send the process name */
      SendUserName      Server      /* Send the username */
      SendProcCert      Server      /* Send the process
certificate */
      SendProcHash      Server      /* Send the process hash
signature */
      ProcCheckValid    Server      /* The credentials of the
process are valid */
      ProcCheckInvalid  Server      /* The credentials of the
process are invalid */
      Data  Server      /* A required piece of data */
      ServCheckValidAck Server      /* Acknowledgement of the
ServCheckValid event */
      AuthRef     Server      /* Authorisation is refused */
      AuthAck     Server      /* Authorisation is given */
      Abort Server      /* Abort the request */
      Check User  /* Simulated initiation usually conducted by the
client program */

end


# Define the outgoing events if required

define eventsout
begin

      AuthReq     Server      /* Initiate the request */
      AuthReqOK   Server      /* Last PDU of the handshake */
      Data  Server      /* A required piece of data */
      SendServCert      Server      /* Ask for the server
certificate */
      ServCheckValid    Server      /* The credentials of the
server are valid */
      ServCheckInvalid  Server      /* The credentials of the
server are invalid */
```

```
      AuthCheck   Server      /* Request final authorisation */

end


#  Define the states

define states
begin

      Idle  /* The client is inactive */
      WaitAuthReqAck    /* Wait for the AuthReqAck event */
      WaitSendProcName  /* Wait for the SendProcName event */
      WaitSendUserName  /* Wait for the SendUserName event */
      WaitSendProcCert  /* Wait for the SendProcCert event */
      WaitSendProcHash  /* Wait for the SendProcHash event */
      WaitValid   /* Wait for the results of the validation of
credentials */
      WaitServCert      /* Wait for the server certificate */
      WaitValidAck      /* Wait for the acknowledgement of the
server validity */
      WaitAuth    /* Wait for final authorisation */

end


# Define the predicates if required

define predicates
begin

      p01         /* If TRUE the server credentials are valid */
end


# Define the actions if required

define actions
begin
# In UNIX programming, popen is a function similar to fopen,
except that the
# first argument is a UNIX command

      [0]   /* Return that the credentials have been rejected */
      [1]   /* Return that the process is authorised */
      [2]   /* Return that the process is not authorised */

end


#  Define the state variable name and the initial state
```

```
define statevarname  ServerVar     initialstate Idle

table ServerTable
begin

      state Idle
            eventin Check AuthReq WaitAuthReqAck

      state WaitAuthReqAck
            eventin AuthReqAck AuthReqOK WaitSendProcName
            eventin Abort Idle

      state WaitSendProcName
            eventin SendProcName Data WaitSendUserName
            eventin Abort Idle

      state WaitSendUserName
            eventin SendUserName Data WaitSendProcCert
            eventin Abort Idle

      state WaitSendProcCert
            eventin SendProcCert Data WaitSendProcHash
            eventin Abort Idle

      state WaitSendProcHash
            eventin SendProcHash Data WaitValid
            eventin Abort Idle

      state WaitValid
            eventin ProcCheckValid SendServCert WaitServCert
            eventin ProcCheckInvalid [0] Idle

      state WaitServCert
            eventin Data p01: ServCheckValid WaitValidAck;
                  ^p01: ServCheckInvalid Idle

      state WaitValidAck
            eventin ServCheckValidAck AuthCheck WaitAuth
            eventin Abort Idle

      state WaitAuth
            eventin AuthAck [1] Idle
            eventin AuthRef [2] Idle

 end
```