Adaptive Firewalls for Grid Computing

Dawei Yao

Kongens Lyngby 2005

Abstract

Grid technology is getting more and more popular now. One of the challenges to deploy Grid applications into the existing environment is to configure the Grid firewalls. The state of art technology is to open the ports as many as needed. Such firewall policy is so risky, and a more dynamic way for controlling the firewall is needed. In this project we propose a secure and dynamic mechanism to adaptively control the firewall for Grid computing. Also, an implementation is made to verify our ideas.

Preface

This thesis is submitted to fulfill the requirements of the Master of Science in Computer System Engineering. The project was done by Dawei Yao during the period February 2005 to August 2005 at the department of Informatics and Mathematical Modeling (IMM), Technical University of Denmark (DTU). The work was supervised by Professor Robin Sharp.

At here, I'd like to thank Professor Robin Sharp for leading me into this special and exciting research area, and for his good ideas and encouragement for better results.

A special thank goes to my parents and wife - Ling Bai, who offered me meticulous support during the whole process.

Lyngby, August 2005

Dawei Yao

Contents

CHAPTER 1

Introduction

1.1 Background

Grid is a system that coordinates distributed resources using standard, open, general-purpose protocols and interfaces to deliver nontrivial qualities of service.[3] As WWW technology allows people to share the content, Grid helps people to share the computing resources, including data, application, and experimental facilities through the Internet. Such share could even be transparent, so that the user does not even need to consider the location of the facilities. The ultimate goal of Grid is that finally the computational power could be used in the same way as the electric power in the power plants.

The rate of increase in the popularity of Grid system depends on how fast we can solve the challenges imposed when integrating the technology into the existing infrastructure. One of the biggest challenges is security. The Grid security includes many disciplines; The functionalities that should be offered include authentication, delegation, single sign-on, credential life span and renewal, authorization, privacy, confidentiality, message integrity, policy exchange, secure logging, manageability and firewall traversal.

The Grid security requirements are modeled in a concept of *"The Secure Grid Society"*, which uses virtual organizations to model the relationship between entities in such a society. The concept of the Virtual Organizations (VOs) is *"flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions, and resources[4]"*. Virtual organizations provide bridges of trust between different entities that belong to separate policy domains. The entities that belong to the same virtual organization could talk with each other in a trusted environment - virtual organization domain.

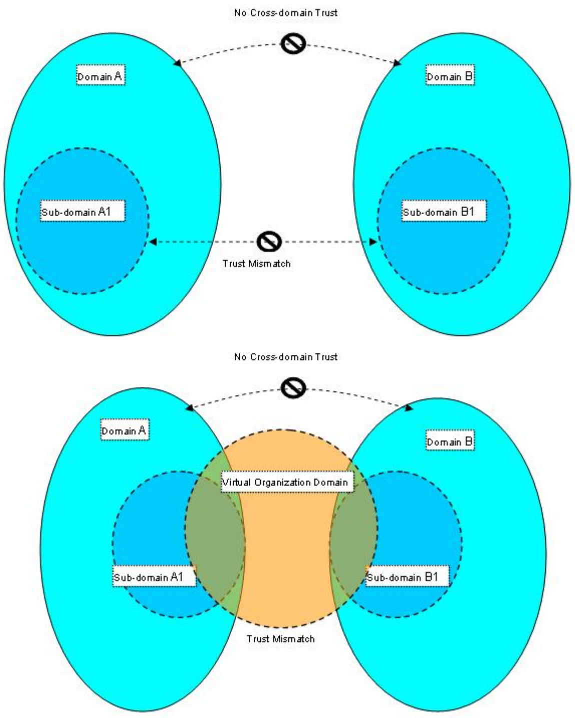


Figure 1.1: Virtual organization to bridge sub-domains .

To fix Grid applications into the currently existing environment, and provide services through the open network, one of the common tasks is to set up the firewalls. Because of the traffic feature of some services, firewalls are normally required to open a big number of ports for incoming connections. A good example is the *"Grid Resource Acquisition and Management (GRAM)"* service. The service requires both parties to open some ports for incoming connection. But some of the ports are kept open all the time even though they are not in use. This is dangerous and an intelligent way of opening and closing firewall by needs should be proposed.

1.2 Motivation

To control a firewall remotely through open network, in a dynamic, secure and intelligent way for Grid computing, we propose a secure protocol on top of IP layer for the remote authenticated user to open and close the firewall. The challenges for our research are:

1. What are the main threats for the remote firewall control, and how to defend the attacks?
2. How to make the protocol robust against system and communication failures?
3. How to integrate the proposed protocol with the existing Grid platforms transparently?

The objective of this project is to study the current state of the art in Grid systems and firewalls. Based on this understanding, design mechanisms for controlling the firewalls adaptively in a secure manner.

1.3 Terminology

The following terms are used in this document:
Ephemeral Port:
A non-deterministic port assigned by the system in the untrusted port range (> 1024).

Controllable Ephemeral Port:
An ephemeral port selected by the Globus Toolkit libraries that is constrained to a given port range.

Grid Service Ports:
Static ports for well-known Grid services

GT2:
Globus Toolkit version 2.x. This release contains only services based on preweb services technology.

GT3:
Globus Toolkit version 3.x. This release contains both pre-web services technology and web services-based technology based on OGSI.

GT4:
Globus Toolkit version 4.x. This release contains both pre-web services technology and web services-based technology based on WSRF.

PRE-WS Services:
Globus services predating the adoption of web services. I.e. the services present in GT2, but also found in GT3 and GT4.

Well-known Port:
A port number registered with IANA .

WSRF:
The Web Services Resource Framework. A set of proposed Web services specifications that define a rendering of the WS-Resource approach in terms of specific message exchanges and related XML definitions. These specifications allow the programmer to declare and implement the association between a Web service and one or more stateful resources.[5]

GSI:
Grid Security Infrastructure, A service used by the Globus Toolkit for secure authentication and communication over an open network. GSI services include mutual authentication and single sign-on. [11]

OGSI:
Open Grid Service Infrastructure, A Global Grid Forum(GGF) standard that defines the core semantics of a transient Web service, including naming, lifetime, and exposing service state. [9]

OGSA:
Open Grid Services Architecture, An integration of Grid and Web services technologies that defines standard interfaces and behaviors for distributed system integration and management. [17]

Chapter 2

Grid Firewall Requirements

Our motivation is to propose an adaptive mechanism for Grid firewalls. But before start, the requirements of Grid computing to its underlying firewall are the first factors we should consider. Such requirements varies with different technology architectures that are adopted during the evolution of Grid. In this chapter we will explore such requirements to present a clear view .

2.1 Grid Security

Grid computing, during the process of evolvement, has adopted two different technology mainstreams - the *"Pre Web Service (Pre-WS) Technologies"* and *"Web Service(WS) Technologies "*. The associated security technologies are also not the same.

2.1.1 *"PRE-WS"* Grid Architecture: Hourglass

The Grid Architecture at the beginning is based on the principles of the *"hourglass model"*, as shown in Figure 2.1. The idea is to build different high-level behaviors based on the small number of core abstractions and protocols at the narrow neck of the *"hourglass"*.

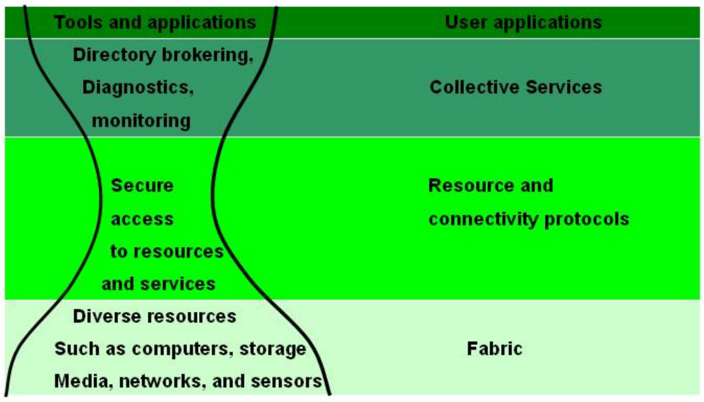


Figure 2.1: The layered hourglass Grid Architecture.

The *"Fabric Layer"* wraps the local, resource-specific operations as sharing operations, to offer the interfaces for local control from higher layer. The *"Connectivity Layer"* defines core Grid-specific network protocols. Also, it includes basic security solutions, such as single sign-on and delegation. The *"Resource Layer"* makes the sharing of single resources possible. But, it does not concern the issues of global state and atomicity of actions between distributed resources. The main purpose of *"Collective Layer"* is to coordinate multiple resource sharing, such as monitoring, data replication, and scheduling. The Grid application is built in the top layer of such model, utilizing services from other layers beneath.

2.1.2 Globus Toolkit Version 2

The Globus Toolkit is a collection of clients and services to enable Grid computing. It is an open source software toolkit used for building Grid systems and applications. Globus Toolkit is being developed by the Globus Alliance and many others all over the world.[7] The toolkit offers a platform for Grid application development, and is popularly used in many places.

Until now, there have been four versions of the Globus Toolkit - *"GT1"*, *"GT2"*, *"GT3"*, and *"GT4"*. *"GT1"* and *"GT2"* use *"PRE-WS"* technologies. *"GT3"* and *"GT4"* adopt *"WS-technologies"*, and will be discussed later. *"GT2"* addresses the Grid technology requirements imposed by *" Hourglass"* model. It implements most of the features in *"Fabric Layer"*, *"Connectivity Layer"*, and *"Resource Layer"*, but little in *"Collective Layer"*.

The security in *"GT2"* is implemented as components in connectivity layer, and is defined by *"Grid Security Infrastructure (GSI)"*. *GSI* defines the services for single sign-on authentication, communication protection, and restricted delegation. *Single sign-on* allows a user to authenticate once and then create a proxy credential that a program can use to authenticate with any remote service on the user's behalf. [6] The Delegation allows the creation and communication to a remote service of a delegated proxy credential that the remote service can use to act on the user's behalf,perhaps with various restrictions; All these services are tightly coupled with an underlying security framework, such as *"SSL with X.509/PKI"*, and *"DCE with Kerberos"*[20].

2.1.3 WS Grid Architecture:OGSA

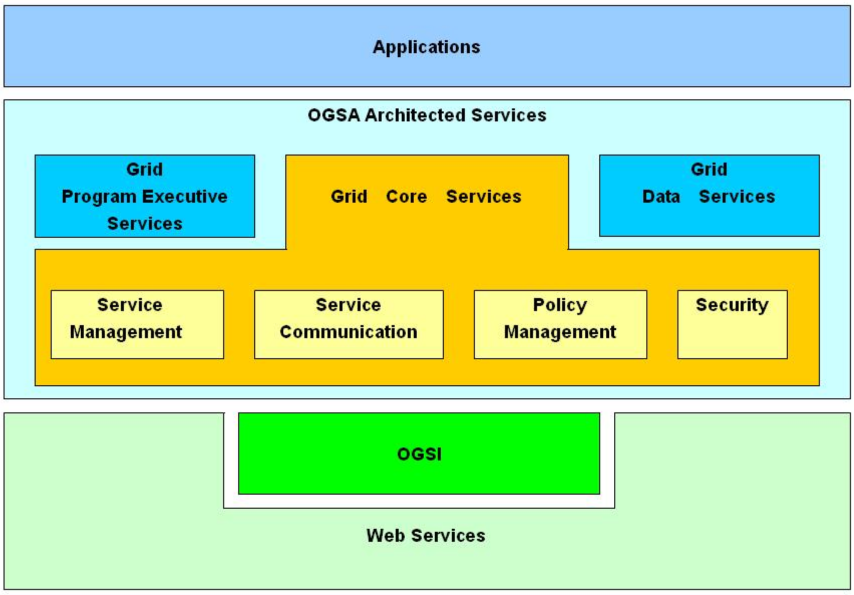


Figure 2.2: The main architecture of OGSA.

With the emergence of web services, a new architecture with service orientation feature is introduced to Grid computation field - *"Open Grid Services Architecture"*. *OGSA* is *"An integration of Grid and web services technologies that defines standard interfaces and behaviors for distributed system integration and* *management."*[9] As illustrated in Figure 2.2, the three principle elements of OGSA include:

- OGSA Services
- OGSI
- Web Services

The *"Open Grid Services Infrastructure (OGSI)"* is *"A Global Grid Forum standard that defines the core semantics of a transient web service, including naming, lifetime, and exposing service state. "*[9] The specification defines grid services based on the mechanisms of web services like XML and WSDL to specify standard interfaces, behaviors, and interaction for all grid resources. OGSI extends the definition of web services to provide capabilities for dynamic, stateful, and manageable web services that are required to model the resources of the grid.[22]. OGSI defines essential building blocks for constructing distributed system.

Based on *OGSI* and web services, the core Grid services are developed. *"Service Management"* provides automated help in managing the deployed services, such as installation, maintenance and monitoring. *"Service Communication"* supports the fundamental communication between Grid services. *"Policy Services"* create a general framework for creation, administration, and management of policies and agreements for system operation.[22]

"Security Services" are built from XML-based security protocol components, which specify the security architecture in a manner that is agnostic to the actual underlying mechanisms. These mechanism-agnostic approaches allow the same basic assertion formats and protocols to be deployed with different underlying security infrastructures. An infrastructure could be built on top of *WS* security without any knowledge about the underlying mechanisms [23]. Such an approach could offer good scalability for the system, since the underlying infrastructure could be replaced with another one to meet changed requirements without affecting the existing web services that are built on top of the previous infrastructure.

2.1.4 Globus Toolkit Version 3 and 4

"GT3" was born with a new *GSI*, which differs quite a lot from the former one. It offers the functionality of authentication, identity federation, dynamic entities and delegation, message-level security, management of overlaid trust domains,

and security service abstraction. *"GT3"* and *"GT4"* contain both *"PRE-WS"* and *"WS"* based components. In *"GT3"*, the web service components are based on the *"Open Grid Services Infrastructure (OGSI)"* specification. In *"GT4"* the web services components are based on the *"Web Service Resource Framework (WSRF)"* specification. [8]

2.2 GT Firewall Requirements

The necessity of using a firewall is to restrict traffic between a protected network and the open network, such as Internet. Especially, for some valuable computing resources or experimental utilities running as Grid service providers, most of the time, security must be firstly assured before performing any further operations.

The *Globus Toolkit*, as a standardized development platform for Grid applications, is raising its popularity as time goes, and has been widely used now. The toolkit includes many service components that require the firewalls to let the traffic go through. The character of traffic differs from service to service and from version to version.

The main services of the *Globus Toolkit* include:

- Job Management - *GRAM*. It defines protocols for creating and interacting with a managed job. *GRAM* jobs are typically associated with computational resources. [11]

- Data Movement - *GridFTP*. It is a high-performance, secure, reliable data transfer protocol optimized for high-bandwidth wide-area networks.[24]

- Monitoring and Discovery - *MDS*. *MDS* is a distributed service for publishing and discovering dynamic information about distributed resources. [14]

Grid Security Infrastructure (GSI) support the authentication and message protection for the above services (see in figure 2.3).

Normally, firewalls exist between the communication entities. All the traffic of such services must be enabled by the firewalls between the ends. At the same time, firewalls should allow some supporting services, such as *GSI-enabled OpenSSH, MyProxy, and MPICH-G2.*

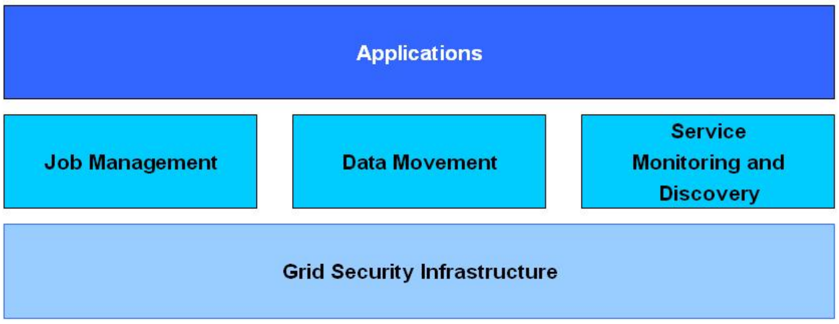

Applications
Job Management
Data Movement
Service Monitoring and Discovery
Grid Security Infrastructure

Figure 2.3: The service architecture of Globus Toolkit .

In this section, we explore the services of different versions of Globus Toolkit, their traffic character and corresponding firewall requirements.

2.2.1 Globus Authentication and Message Protection (GSI)

GSI is used in Globus Toolkit for user authentication and message protection. *GSI* adopts SSL and X.509/PKI certificates to provide *GSI protections* for:

- Secure Mutual Authentication
- Integrity Protection
- Encryption
- Secure Delegation

The infrastructure needs the support from *GSI-enabled OpenSSH*, which has the incoming connections to TCP port 22.

2.2.2 Job Initiation and Management (GRAM)

The *"Grid Resource Acquisition and Management (GRAM)"* service offers the methods for job creation and job management from remote hosts. The mechanism of protocol is different in versions of *GT*. The *"PRE-WS"* GRAM protocol, which could be found in *"GT2"*, *"GT3"* and *"GT4"*, generates connection traffic initiated from client for job creation, and traffic from both ends for job management.

The *WSRF* and *OGSI GRAM* Protocols, which could be found in *"GT3"* and *"GT4"*, generate connection traffic initiated from client for job creation and job management. But the file staging will still result in reverse connections as in *"PRE-WS GRAM"* .

2.2.3 Data Movement (GridFTP)

The *Globus GridFTP* server appears in all versions of the toolkit and is a *"PRE-WS"* component. The network traffic characteristics of all versions are identical. The protocol contains two kinds of channels, control channel and data channel. The connection of control channel is initiated by the client. The data channel could have single connection or multiple connections for parallel transfer. The single connection starts from the client, but the multiple connections start from the source of data. This could mean *"bi-directional"* connections.

2.2.4 Monitoring and Discovery Service (MDS)

MDS offers a means to monitor the status of the machines and find the proper ones for use. The protocol and generated traffic differs in *"PRE-WS MDS"* , *"OGSI and WSRF MDS"*.

The *"PRE-WS MDS"* architecture has two main components: *"Grid Resource Information Servers (GRISs)"* and *"Grid Information Index Servers (GIISs)"*. *"GRISs"* run on resources and respond directly to any queries. *"GRISs"* also typically register themselves to one or more GIISs. This allows users to query a *"GIIS"*, see all the available resources in an organization and then query the resource's GRIS directory or through the *"GIIS"*. [8]

Traffic in both direction between *"GRIS"* and *"GIIS"* should be allowed. Also, the traffic from Client to *"GRIS"* and *"GIIS"* should be permitted.

For *"OGSI and WSRF MDS"*, the model is similar to the *"PRE-WS MDS"* model. But we only need to allow the traffic from client to the service port.

2.2.5 Client Site Firewall Requirements

From the above analysis of services, we could draw the following conclusions.

- Clients need the permission to connect freely from ephemeral ports on host to all ports at server site.
- Since the GRAM service uses callbacks to client; client sites should allow incoming connections and restrict the incoming port range.

2.2.6 Server Site Firewall Requirements

Sites that host Grid services often host or act as Grid clients. For example, retrieving files needed by the job. So, the server firewall configuration should meet all the client site firewall requirements.

2.2.6.1 Allowed Incoming Ports

A server site should allow incoming connections to the *"well-known Grid Service Ports"* as well as ephemeral ports.[8]

For *"PRE-WS"* services these ports are:

- *22/tcp for GSI-ENABLED OPENSSH*
- *2119/tcp for GRAM*
- *2135/tcp for MDS*
- *2811/tcp for GridFTP.*

For *"WS-based"* services these ports are:

- *22/tcp for GSI-ENABLED OPENSSH*
- *2811/tcp for GridFTP*
- *8080/tcp (*"GT3"*) or 8443/tcp (*"GT4"*) for GRAM and MDS.*

2.2.6.2 Allowed Outgoing Ports

Server sites should allow outgoing connections freely from ephemeral ports at the server site to ephemeral ports at client sites as well as to Grid Service Ports at other sites.[8]

From the above discussion, it is obvious that to construct a firewall policy that can meet the Grid firewall requirements, administrators need to open several *well-known Grid Service Ports*, and a range of ephemeral ports on server sites for incoming connections. Even, the client sites need to open some ephemeral ports for callback connections. Without dynamic firewall control, such ports will be kept open even when there are no activities at all. This is dangerous!!! Some backdoor programs could listen to such ports for incoming connections.

An adaptive firewall at each end, which could open and close ports based on service needs, could solve such problem. The firewall will open the ports when it receives authenticated requests. Moreover, the firewall will close the ports when there are no service activities on those ports.

An alternative is to reduce the number of open firewall ports. A piece of software called *"Nexus Proxy"* offers port tunneling by multiplexing and tunneling all *Globus* communications through a specified port. A *"Nexus Proxy"* incoming server runs outside the firewall. At the same time, a *"Nexus Proxy"* outgoing server runs as a daemon process inside the firewall. But, the use of such proxy process could impose heavy system burden and become a bottleneck when there is a heavy load of communication.

2.3 Summary

In this chapter, we present two different architectures of Grid - *"Hourglass"* and *"OGSA"* , as well as their security mechanisms. Also, we introduce the services in these two architectures and their traffic feature and corresponding requirements to firewalls.

There are two distinct architectures of Grid - *"PRE-WS"* and *"WS"* . The security mechanisms that they use are also different. The services in each architectures that are implemented in Globus Toolkit are almost the same, but with different characters of traffic. Thus , the firewall requirements off *"PRE-WS"* and WS architecture are not the same.

Another factor that should be paid attention to is that the client side needs to open some ports for incoming connections, which is obviously dangerous.

Chapter 3

The Threats and Requirements

The threats to Adaptive Grid Firewall Protocol (named *" AGF-protocol "* in the following discussion) could be attacks, communication failures and system failures. They are all environmental factors that could exert some influences on the normal operations of the protocol. In this chapter we will discuss about the threats and requirements for an adaptive firewall.

3.1 Attacks

Before we design AGF-Protocol for firewall open and close, we need to research the potential classes of network attacks. Such attack patterns could suggest where protections need to be tightened during the design process. We expand our discussion based on the presumption that the primary key of each ends would not be compromised. AGF-Protocol is not designed to deal with the key compromise vulnerability.

The classes of network attacks that we consider relevant include *"Man in the middle attacks"*, *"brute force attacks"*, and *"dictionary attacks"*.

3.1.1 Man in the middle attacks

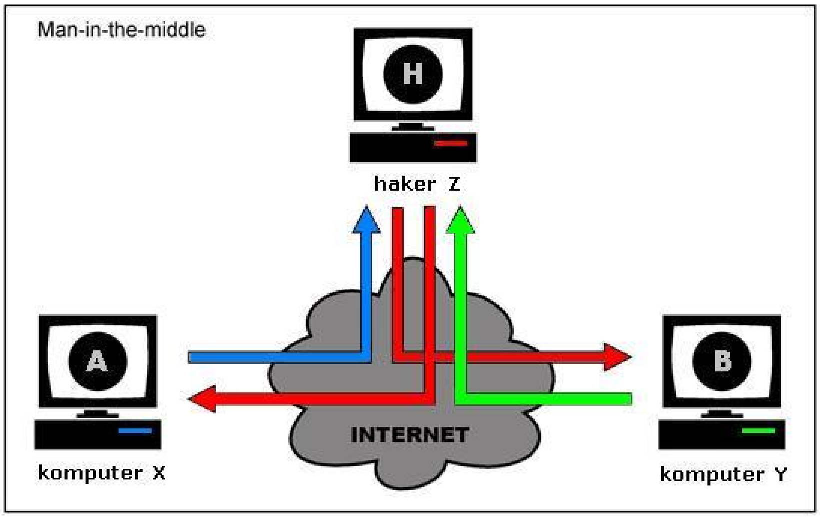


Figure 3.1: The process of Man in the Middle Attack.

A man in the middle attack (MITM) is an attack in which an attacker is able to read, insert and modify at will, messages between two parties without either party knowing that the link between them has been compromised.[10] The MITM attack(figure 3.1) may include one or more of: *“Eavesdropping”*, *“substitution attack”*, *“replay attack”*, and *“denial of service attack”*.

3.1.1.1 Eavesdropping

Eavesdropping includes traffic analysis and possibly a known plaintext attack. The characteristic of such attack is intercepting and reading messages by unintended recipient.

The threats of such attack to AGF-Protocol are that an adversary could find out the exact port number that a client wants to open. Then, it’s possible for him to race and setup connection earlier than the client and performs further attacks. An adversary could also combine eavesdropping with other attacks such as substitution attack to disguise as a client and open the firewall ports at will.

To protect the communication from such attack, a security service of confidentiality, which is normally encryption, could be used. For AGF-Protocol, the messages are encrypted partially. Parts of the message, which are used for identifying the message and insensitive, such as message type, and user ID, are exposed to the communication channel. But, some parts of the message, which are sensitive, such as the expected port to open, session key, are encrypted with strong encryption algorithm.

3.1.1.2 Substitution attack

An adversary could substitute the messages maliciously without the notice from either party. Such attack could change the meaning of messages, and cause the abnormal behavior of the victim party.

If the content of messages could be changed without noticed, then it's a nightmare for both server and client. For AGF-Protocol, if the expected port number could be modified by adversary on fly, the server side firewall would open a port which is appointed by the attacker. The client, thinking the server has opened the desired port, will encounter communication failure, since the port is not opened actually. Thus, it seems like that the client is helping the attacker to open any service ports within the allowed service port range.

To protect AGF-Protocol from such attack, we use message authentication code *(MAC)*. A *MAC* algorithm accepts as input a secret key and an arbitrary-length message to be authenticated, and outputs a *MAC* (sometimes known as a tag). The *MAC* value protects both a message's integrity as well as its authenticity, by allowing verifiers (who also possess the secret key) to detect any changes to the message content. [16] *MAC*, in AGF-Protocol, is designed as an encrypted digest of the whole message.

3.1.1.3 Replay attacks

The adversary could save parts or all of the data transmissions and resend or delay the message at a detected pattern to the party, disguising the contrary one.

For AGF-Protocol, such attack could cause the server to open the same firewall ports that a client has ever opened to the attacker. There are two ways to prevent replay attacks, one is session tokens, and another is time stamping.

The main idea of time stamping is that: The communication parties could only accept the authenticated time stamped messages within a reasonable time tolerance. The messages stamped out of synchronization will not be accepted.

One of the defects of such protocol is that we must synchronize securely the clocks at both ends, which could be difficult.

AGF-Protocol is designed with the idea from one-time password and pseudorandom number sequence. It could be seen as a kind of session token method. Both ends share a same primary key. For each session, the server will generate a challenge to the client and negotiate a one-time session key for message encryption. Each message is numbered with the random number in the pseudorandom sequence, which is generated based on session key and primary key.

Thus, an adversary can not reuse the intercepted messages to gain the access to ports.

3.1.1.4 Denial of service attacks

The most popular type of such attack is to cause the consumption or overload of system or network resources. The adversary could utilize the design faults of protocol to keep the system too busy to serve any other clients.

For AGF-Protocol, such attack could cause the server fails to serve any requests from client, and keep the firewall closed all the time.

To avoid such attacks, AGF-Protocol is designed to only open one service thread for identical request messages. Also, the message authentication code could prevent an adversary from generating random request messages without the knowledge of primary key. The maximum number of such requests is the number of all the user-port pairs, which most of the time is relatively small in scale .

3.1.2 Distributed Denial of service attacks

DDoS attacks is a kind of DoS attacks, but use many hosts on the Internet (Figure 3.2). In a distributed attack, the attacking computer hosts are often zombie computers with broadband connections to the Internet that have been compromised by viruses or Trojan horse programs that allow the perpetrator to remotely control the machine and direct the attack, often through a botnet. With enough such slave hosts, the services of even the largest and most well-connected websites can be denied.[15]

The effect of such attack is the same as DoS attack. The message authentication code could also help at here to lower down the number of valid requests to the

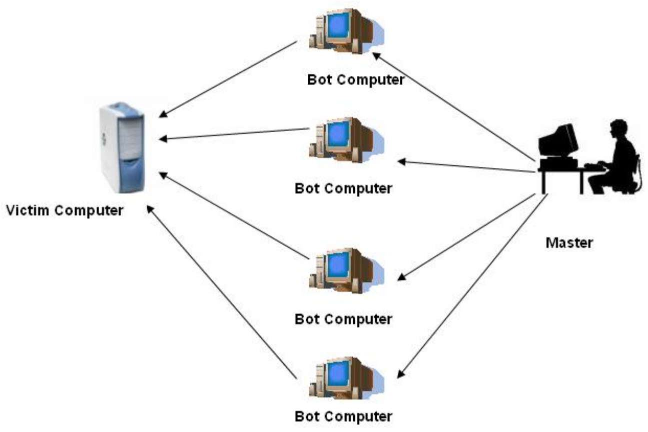


Figure 3.2: The process of Distributed Dinial of Service Attack.

number of all the user-port pairs. As long as the server could handle all the requests from all the legitimate users at the same time, then it could handle such attacks.

3.1.3 Brute force attacks

The adversary could decrypt the data transmission by trying a large number of possible encryption keys.

For AGF-Protocol, such attacks, if success, could decrypt all the communications, and disclose the session key or even primary key. Then the attacker could perform any kinds of attacks which are prevented by encryption and random number generation technique.

To avoid such attack, we could only depend on the strong encryption method. AES could offer strong encryption for 128-bits, 256-bits or even longer. We adopt AES as encryption method for AGF-Protocol.

3.1.4 Dictionary attacks

Dictionary attack refers to the general technique of trying to guess some secret by running through a list of likely possibilities, often a list of words from a dictionary.[12]

The effects of such attack are the same as the effects of brute force attacks. One of the most important feature of such attacks is that most of the possibilities that they searches are human related. To defend AGF-Protocol against it, we insulate all the information sent between two ends from human. The information has nothing to do with any human kind, they are just generated randomly. For example, we could generate primary key and session key from a hardware random number generator, which use resistor or semiconductor noise as the source of randomness.

3.2 Communication Failure

We should never make an assumption that the communication channel could be seen as an ideal pipeline, especially for the connection through open networks. A message, during the transmission, needs to go through many *"decision makes"* before it could actually reach its final destination.

As the OSI network reference model shows, a message needs to go through several layers from top to bottom at hosting machine before it is actually turned into electronic signals and sent out. During the transmission, at each node of the route, the signals have to be processed and passed to the next node. At the destination host, such signals will be turned back to the original message by going through several layers from bottom to top. Also, during transmission, the message could be divided into many packets and reassembled at the destination machine.

The message could be received correctly if and only if all the above steps run as expected. But, there could be many errors. For example, the electronic signals could be affected by the environmental electromagnetic field. Also, the packets could be directed to a wrong place because of software or hardware design faults of the nodes. Some overloaded nodes, such as routers, would delay or drop the packet.

In total, the messages, during the transmission, could get corrupted, delayed, or even lost.

The protocol that we designed should also be able to withstand such communication failures. Otherwise, the session will be so easy to break down. Then, it would be hard for a client to keep the firewall open during a long time, because, the connection is considered not stable enough to perform such a task.

To deal with communication failures, we adopted some traditional ideas from protocol design, such as resending and time out mechanisms.

3.3 System Failure

A system failure is *"an internally detected error from which recovery is not possible. Rather than continue to operate, risking data integrity, the operating system halts the computer"*.[18]

We can not stop system failure. How could the protocol get recovered from system failure? The answer is *"saving the status variables"*. As for AGF-Protocol, after the current session is finished, it is important for both sides to store the status variables into history files for use in the next session. In order to recover from system failure, both entities will also store the status variables at critical steps into history files. With such files, the client and server could synchronize to start from the same point.

Another question is: *"How could the implementation guarantee that the firewall is closed, when the operating system restart after a system failure?"* It is an obvious risk, if the firewall is kept open when the system restarts, only because the implementation does not ever have a chance to close it. Such problem could be solved, if the firewall rules that are modified are temporary, say only in the memory, when firewall is running. Such rules will NEVER be written into the firewall rule configuration files. Once the system restarts, the firewall will restart and load the stable rules which are stored in files.

Here comes another question: *"How could the implementation guarantee that it will close the ports anyway when it encounters a program faults, or it is stopped suddenly?"* Well, it's like a question of *"Who could you depend on when your gate guard gets sick?"* My answer is *"The gate could close by itself when nobody opens it."* We could have another process, which is simple but robust, to monitor the firewall. The process could open and close the firewall and must get continuous keep-alive messages for keeping the firewall open; otherwise it will close the firewall as soon as time out.

3.4 Adaptive GRID Firewall Requirements

Based on the discussion about Globus Toolkit firewall requirements and the potential threats and failures. We found the following requirements, with the consideration of integrating the implementation into the existing Grid services:

- The protocol should be resistant to the network attacks mentioned above
- The protocol should be robust enough to deal with potential communication and system failures, including:
  - Message loss, corruption and delay
  - Abnormal quit from program, such as system reboot and sudden system power off.
  - The firewall should close when there is a failure.
- The implementation of the protocol should be able to integrate into the existing *"GT2"*, *"GT3"* and *"GT4"* applications. It should be configured as transparent as possible to applications in the upper layer

3.5 Referenced Idea: Port Knocking

To design an adaptive firewall , one of the solutions could be to open only one service port on the firewall for the AGF-Protocol server to receive incoming *"firewall-open"* requests. The disadvantage of such method is that there still exists a *" hole "* in the firewall. For a backdoor program, it could get some chances to receive incoming connections through this port, when our service program goes wrong and release the port.

An alternative is closing all the ports for incomming connections. The port is only opened for a client when it could authenticate to the server. The capability of penetrating the firewall is a must to deal with such situation.

Port Knocking is one of the solution for stealthy authentication across closed ports. port knocking refers to a method of communication between two computers in which information is encoded, and possibly encrypted, into a sequence of port numbers.This sequence is termed the knock. [25]

The server presents no open ports to outside and *"silently"* monitors all connection attempts. Only the client which could *"knock"* correctly could trigger the server to open the expected port for it. Such knock is a sequence of TCP *"SYN"* packets with specific port numbers on them. The monitor inside the firewall could detect such knock by checking the firewall log file of incomming TCP packets.If it detect a successful knock, it will open the required port(s).

The good thing for such method is that it provides a mechanism to communicate to the processes inside firewall through closed ports. But the bad thing is that it splits the request into small packets. The receiver has to have the capability to *" reassemble "* the *"SYN"* packets. The communication parties have to make sure the order of receiving the packets keeps the same as the order of sending the packets, otherwise the decoded information will be different as sent. If the packet is sent through a complex route, the order could not be guaranteed. Also, since the packets are not authenticated, the server can not disdinguish a packet from a forged one. The attacker can insert *"SYN"* with random port number but same IP address as the client has, to prevent the server from receiving correct knocking sequence.

To design AGF-Protocol , we take advantage of the goodness of Port Knocking and use authenticated message within one IP packet. By using *"PCAP"* library, we could detect incomming IP packets through a closed firewall. The solution will be discussed in more detail in the following chapter.

3.6 Summary

As listed, the AGF-Protocol should fulfil all the requirements about attacks and failures. The attacks include eavesdropping, substitution attacks, replay attacks, DoS, DDoS, brute force attacks, and dictionary attacks. At the same time, AGF-Protocol should also be designed to withstand communication failure(message corruption, delay and loss) and system failure(OS halts,program failure).It is also important that the protocol is feasible for integrating it with the existing Grid architectures.

Chapter 4

Conceptual Design

The design process of adaptive Grid firewall includes three phases: conceptual design, protocol design, architecture design and protocol modeling.

Conceptual design includes some basic ideas about how to solve the challenges given by requirements about network attacks and system/communication failures. The protocol design concerns more about the details, such as message format and behavior of the two ends. The architecture design gives the architectural view of integrating protocol implementation into the existing Grid service architecture.

The conceptual design has referenced many ideas from the state of the art network security technologies and traditional communication protocol design principles. The design gives the general view and direction of problem solving against the requirement challenges.

4.1 Network Security Challenges

4.1.1 Eavesdropping

Eavesdropping is characterized as intercepting and reading messages by unintended recipient. The threats of such attack to AGF-Protocol have been discussed in the former chapter. At here, we presents a more detailed solution.

To defend the data transition against eavesdropping, the protocol uses AES encryption algorithm for a fast and strong protection. The advantage of AES is that it could offer at least 128-bit security. And the key length could vary based on different computational configurations and needs. Also the number of encryption cycle can be extended to offer the option for trade-off between performance and security level.

The AES is used to protect some sensitive parts of the message, including the expected port number to open, the content of challenge, the content of response, and the session key. And it is used as the generation algorithm for message authentication code (MAC).

The information, such as user ID and message type, is exposed to the eavesdropper. But, such information must be exposed, so that the communication parties could recognize and process the messages. Even though eavesdroppers could get the knowledge about communication pattern through such information, and could substitute the messages at correct time. It will be no use for them, if such behavior could be detected.

4.1.2 Substitution attack

The character of substitution attack is changing the message without notice.

The basic idea to defend the message against such attacks is protecting the integrity. The integrity is the characteristic of data being accurate and complete. The message authentication code (MAC) could offer the service of both integrity and authenticity.

To defend the communication against IP falsification, we bind the IP address with the message also. The technique in AGF-Protocol is appending the message with host IP address, and the primary key. Then generate the hash value and encrypt the value with current session key. The process is shown in figure 4.1.

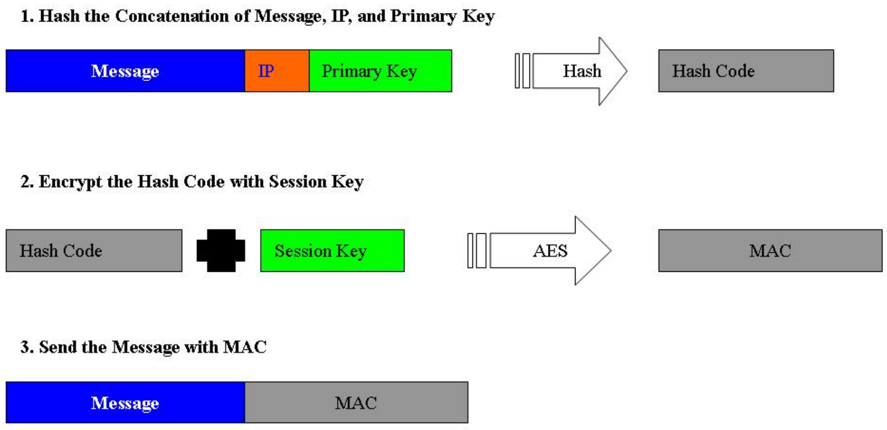

1. Hash the Concatenation of Message, IP, and Primary Key
Message
IP
Primary Key
Hash
Hash Code
2. Encrypt the Hash Code with Session Key
Hash Code
Session Key
AES
MAC
3. Send the Message with MAC
Message
MAC

Figure 4.1: The process of MAC generation.

The recipient receives the message, appends the message with source IP address and its primary key, and generates the hash value B. Then it decrypts the MAC with session key and get a hash value A. If the received decrypted hash value A is the same as the received generated hash value B, the recipient could know that the message is originated from authenticated counterparty, and is not tampered during transmission. The process is shown in figure 4.2

4.1.3 Replay attacks

One of the premises for the success of replay attack is that the communication parties allow the reuse of messages that is ever sent or accept the delayed message.

As designed, the protocol does not allow identical messages in the whole session. Only the first message that the client sent to the server could be the same as last session, for sake of failure recovery. But the reply to such a message is a new challenging message, which changes randomly every time, thus the adversary could not do anything further.

The protocol use Challenge/ Response mechanism before it sets up the session and opens the firewall ports. The challenge is encrypted and varies every time when establishing the session. The challenge includes a new session key for use in this session. As discussed above, the MACs for the same content of the message between different sessions will be different, because of the different

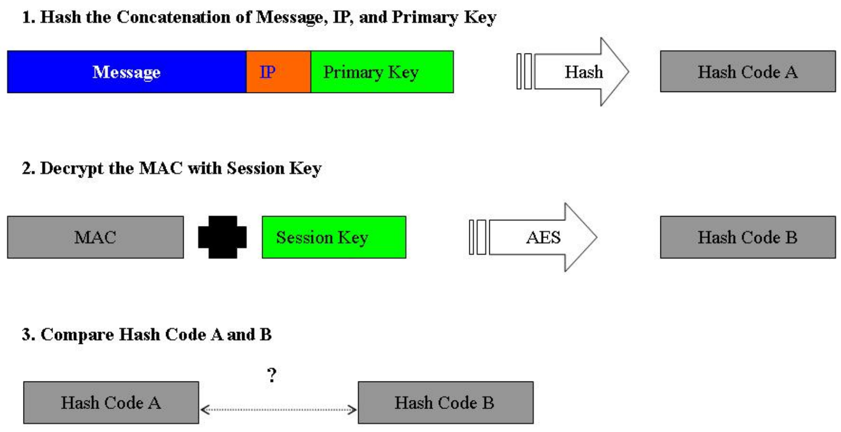

1. Hash the Concatenation of Message, IP, and Primary Key
Message
IP
Primary Key
Hash
Hash Code A
2. Decrypt the MAC with Session Key
MAC
Session Key
AES
Hash Code B
3. Compare Hash Code A and B
Hash Code A
?
Hash Code B

Figure 4.2: The process of MAC verification.

session key used. Thus, the messages, which are attached with MAC and sent in different sessions, are different. Thus, the reuse of messages between sessions is not possible.

It is also not possible to reuse the messages in one session. The process is shown in figure 4.3. All the messages within one session are attached with different random number from a same pseudorandom number sequence, which are generated with the primary key and session key. Although, some messages, such as keep-alive messages, are identical, the number attached with each of them is different, and can not be replayed.

For the messages that arrive late, the parties will ignore them. For a message A that has ever been received, if it could be considered as the resent message because of communication failure, the party will resend the reply of it - message B, which has ever been sent. Once the party has received the successive message - message C, which is the *"reply of its reply"*, it will continue and ignore message A, if such message is received during the successive steps. Thus, a replay attack can not get success because it can not *"drive"* the party to go one step further.

4.1.4 Denial of service attacks

Since the system could only accept the new messages in the whole process, except the first one. To consume the system resource, an adversary performing DoS attack, could only take advantage of the first message feature. He keeps sending the intercepted old first message to the server. The message is attached

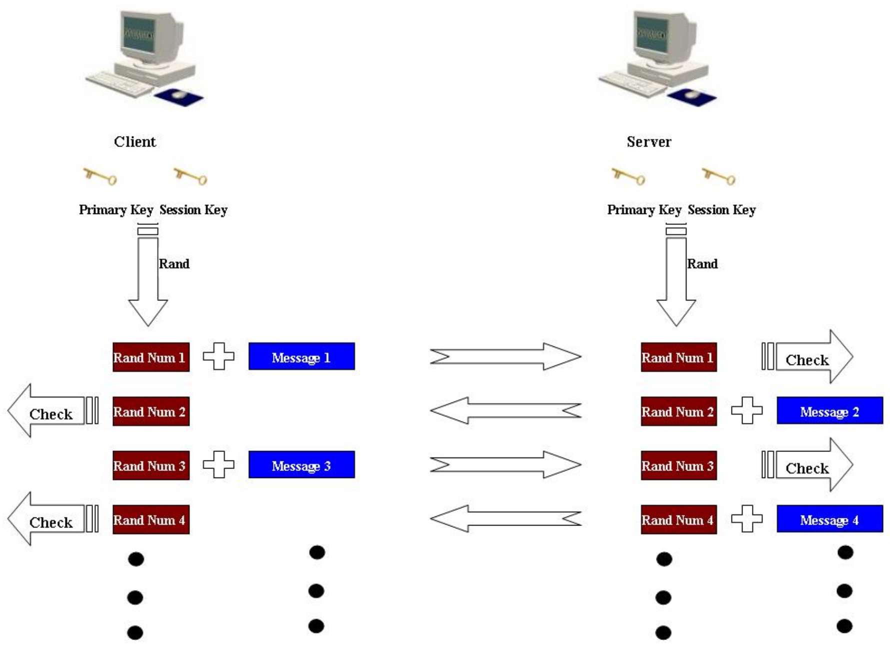


Figure 4.3: Pseudorandom Number Sequence Generation and Verification.

with the digest of the combination of original IP address, key, and the message itself. It is not possible for the attacker to change the message to look like a new request message, without violating the integrity. We name such message as *"start message"*.

After receiving the start message, if it is correct, the server will generate a challenge and wait for the response until time out. During the waiting process, any identical messages will be ignored. Thus, there is only one thread created for identical messages. The number of such valid *"request messages"* is the same as the number of *"user - port"* pairs that a server could handle, which normally is a relatively small number. Thus, such kind of attack is degraded to a low level. Even if the attack starts a distributed denial of service attack, the maximum number of servant threads that could be raised by such attack is the same as the maximum valid number of *"user - port "* pairs for the server. The server could also degrade such kind of attacks.

4.1.5 Brute Force Attacks

The communication is protected by AES. Until now, we have not seen any report about the possibility of cracking an AES encrypted cyphertext by brute force.

4.1.6 Dictionary attack

The random number attached to each message is picked from a pseudorandom number sequence, and is 160 bits long. The primary key and session key are also generated randomly. Nothing in the communication is generated from personalized information, such as name, birthday or human selected password. It's hard to have a dictionary for the random numbers used in the protocol.

4.2 Communication Failure

Communication Failure could include message loss, message delay and message corruption. Based on the principle of traditional design, we could use simplex communication, message retransmission, time out and maximum trying times to guarantee the transmission of messages. The mechanisms could handle message loss and delay. Both ends ignore the corrupted messages.

In simplex communication, the process will not continue until the party has received the expected message in time or the receiving process is timeout. Such method could synchronize the communication parties within the transmission process simply but stably.

The message retransmission, together with time out mechanism, could make the protocol robust against message loss and delay. The party will try to resend the message if it can not receive its replay in time. If the maximum trying times is reached, it will abandon the communication and quit.

4.3 System Failure

The protocol could save the status of communication as a history into history file. When the session of communication gets success from the first message to the last one, the history will be updated. When there is a failure in the process, such history will not be updated, and we use it, then, as a starting point for recovery. The more detailed discussion could be found in the section of protocol design.

4.4 Summary

In this chapter, we present the basic design ideas to solve the problems imposed by adaptive Grid firewall requirements.

- AES encryption for eavesdropping and brute force attacks
- MAC for substitution attack
- Challenge/Response plus Pseudorandom Number Sequence for replay attack
- Single Thread for DoS and DDoS
- Random big number for dictionary attacks
- Timeout/Resend for communication failure
- Status Saving/Restore for system failure

The encryption and pseudorandom number sequence generation is performed base on a key pair that's shared between the two communication parties. A key pair includes a primary key and session key. The primary key is kept stable, while the session key is negotiated at the beginning of every session and changes in different sessions.

Chapter 5

Protocol Design

The protocol design includes the design for the formats of messages, and the behavior of two ends when they have received or failed to receive such messages. In this chapter we will present you an insight of the different types of messages, the meaning of each part in a message, and the process of a whole session for opening and closing a firewall.

5.1 Message Design

Based on the ideas from conceptual design, we could extract the message format, divide a message into several parts and set the relationship between them. And finally we get the general message format and message types as design results.

1. Use random sequence generator to generate random number, and attach it to every message

2. Use key pair, including primary key K_{prim} and session key K_{sess}. The primary key keeps stable, while the session key changes from session to session.

3. The random sequence generator is a function of the sequence number n, K_{prim} and K_{sess}.

$X_n = Rand(n, K_{prim}, K_{sess})$

4. A message M should include the message type, user ID, random sequence number n, corresponding random number X_n , the encrypted expecting port number EEP , and the optional encrypted content EC .

$M = MessageType + UserID + n + X_n + EEP + EC$

5. The content of the message, such as the expected port number, will be encrypted by the K_{sess} with a symmetrical encryption algorithm. The expression is:

$EC = Enc(Content, K_{sess})$

6. For each message M , the sender should make a digest of the concatenation of the source IP address, the primary key K_{prim} , and M , then encrypt this digest with the session key K_{sess}. ,using symmetrical encryption algorithm for the concern of performance.

$Sig(M) = Enc(Dig(M + IP + K_{prim}), K_{sess})$

The function Enc is the encryption function, such as *AES*. The Dig is the digest algorithm, such as *MD5*. The result could be seen as kind of signature of the message, and is sent, of course, together with this message.

7. The general format of the message is:

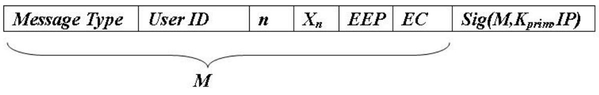


Figure 5.1: General Format of the Message.

8. There are eight types of messages, with the name:

- ASK: The first message of session, sent from client to server. The message informs the server that a client wants to open a port.
- CHAL: The second message of the session, sent from server to client. The message contains a challenge, which includes new session key nK_{sess} and new random sequence number nn. The challenge is encrypted with primary key K_{prim} . As for the Client, it needs to decrypt the challenge, generate a new random number based on the given nK_{sess} , nn. The function used is:

$X_{nn} = Rand(nn, K_{prim}, nK_{sess})$

- PORT: The third message of the session, sent from client to server. The message is sent with X_{nn} and nn. It changes EEP to a new one —the expected port number is encrypted with the new session key nK_{sess}.

  $EEP = Enc(expectedport, nK_{sess})$

- OPEN: The fourth message of the session, sent from server to client. The message is sent with X_{nn+1} and $nn + 1$. The port number is encrypted with nK_{sess}. The message is telling the client that its request to that port could be granted. If the client could send a KEEP message, the firewall at server site will open the port for it.

- KEEP: The fifth message of session, sent from client to server. The message is sent with X_{nn+2} and $nn + 2$. The port number is encrypted with nK_{sess}. The message is a kind of keep-alive message, telling the server that the client wants to keep the port open. If the server is not able to get this message, it will not open the firewall. Such type of message will be sent during the session, to keep the connection alive. If the server is not able to get this message within a given period of time, it will close the firewall port.

- ACK: The sixth message of the session, sent from server to client. The message is sent with X_{nn+3} and $nn + 3$. The port number is encrypted with nK_{sess}. The message is telling the client that the expected port has been opened at the server side. The client should keep sending the KEEP messages, with the successive random numbers in sequence.

- TERM: The message offers a positive way for a client to close the remote firewall port. It is sent from client to server. The port number is encrypted with nK_{sess}. The message is telling the server that the client wants to quit the connection and closes the port.

- TACK: The last massage that is sent in the whole session. After this message, the process terminates. The server sends it to client after it receives TERM and closes the firewall port. The port number is encrypted, the same as before. This message is telling the client that the server has received its terminating request, and closed the firewall successfully.

5.2 Client and Server Behavior Design

After we have finished with the design of message types and their semantic meanings, we need to define the behavior of the client and server about how to

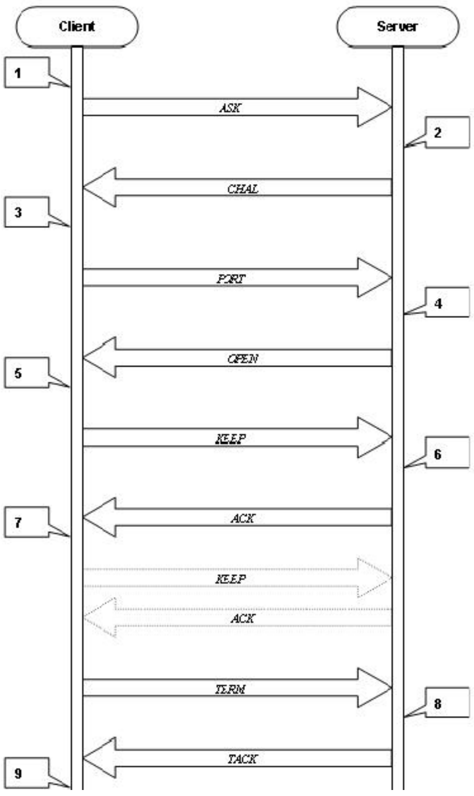


Figure 5.2: Message Transmission in One Session.

process the received messages, and how to deal with system and communication failures. In this section we will discuss about the behavior of client and server at each point that is labeled in figure 5.2

5.2.1 Communication Start: ASK

The session is initiated by the client. It loads the system status from a history file. As mentioned before, the history mechanism is designed for system recovery. Such file contains the system status information of last successful session, including the expected port - $ExpPort$, $userID$, K_{prim}, K_{sess} , X_n and n.

In the process there are several state variant, which are used in the whole process.

- *c_curn: client variable, current random sequence number n for sending*
- *c_curx: client variable, current random number X_n for sending*
- *c_chkn: client variable, random sequence number for checking*
- *c_chkx: client variable, random number for checking*
- *c_curUID: client variable, current user ID*
- *c_curKsess: client variable, current session key*
- *c_expPort: client variable, expected port number*

The process sets the variables as follows:

$c\_curn = n$

$c\_curx = X_n$

$c\_chkn = n + 1$

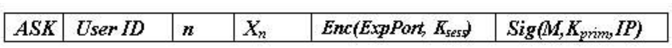
$c\_chkx = Rand(n + 1, K_{prim}, K_{sess})$

$c\_curUID = userID$

$c\_curKsess = K_{sess}$

$c\_expPort = ExpPort$

Base on such information, it sends an ASK message:



ASK	*User ID*	*n*	X_n	*Enc(ExpPort, K_{sess})*	*Sig(M,K_{prim},IP)*

Figure 5.3: ASK message format.

After sending the message, the client will calculate X_{n+1} and $n+1$ to check the reply from server.

$chkx = X_{n+1}$

$chkn = n + 1$

Also, the client will start a timer to count time out. If a reply can not arrive in time, it will try to send the same message again. If it still can not get reply after trying for several times, say *TryOutTimes*, the client will give up, report a communication failure and quit the process. We define such a process as wait and try process, in short *"W/T process"*. The client uses such process when sending all types of messages, including ASK, PORT, KEEP, and TERM. In the following discussion, we will use *"W/T process"* to reference such behavior.

5.2.2 Reply and Challenge: CHAL

As a server, when it has received such an ASK message, it starts a new thread to process the request. We call such a process as service process. The service process first checks the user ID, if such ID is legitimate, the process will then load the service status from a history file of this user. Such file contains the system status information of last session, including the values of status variants as below.

There are several state variants in the thread, which are used in the whole session.

- *s_curn: server variable, current random sequence number n for sending*
- *s_curx: server variable, current random number X_n for sending*
- *s_chkn: server variable, random sequence number for checking*
- *s_chkx: server variable, random number for checking*
- *s_curUID: server variable, current user ID*
- *s_curKsess: server variable, current session key*
- *s_processSuc: server Boolean variable, indicate whether the process get success in the whole session*
- *s_oldchkn: server variable, the old random sequence number for checking*
- *s_oldchkx: server variable, the old random number for checking*
- *s_oldKsess: server variable, the old session key*

- *s_expPort: server variable, expected port number*

After loading, the service process checks the variable *s_processSuc*. The variable indicates whether the last session is a successful communication process. If it's true, the service process should accept the ASK message with the successive random number and the ASK message with the old random number as well. If it's false, the service process should only accept the message with old random number.

Because, as designed, the service process considers it to be successful if the session is terminated by the client and the service process sends the last message - TACK. Then the process will save the *s_processSuc* as true into the history file. But, if the client could not get the TACK, it will recognize the session as unsuccessful, and will send the old ASK message next time when it starts.

If the service process receives a successive message, it updates *s_oldchkn*, *s_oldchkx*, *s_oldKsess* and *s_processSuc* to the current correspondings, and save them into history file, which is:

$s\_oldchkn = s\_chkn$

$s\_oldchkx = s\_chkx$

$s\_oldKsess = s\_curKsess$

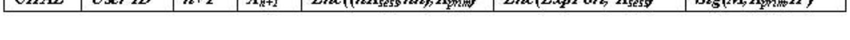
$s\_processSuc = false$

If the service process receives an old message, instead, it will update the current session key, current n, and current x based on old values, as follows:

$s\_curKsess = s\_oldKsess$

$s\_curn = s\_oldchkn + 1$

$s\_curx = Rand(s\_curn, K_{prim}, s\_curKsess)$



CHAL	User ID	$n+1$	X_{n+1}	$Enc((nK_{sess}, nn), K_{prim})$	$Enc(ExpPort, K_{sess})$	$Sig(M, K_{prim}, IP)$

Figure 5.4: CHAL message format.

Afterwards, the service process generates randomly a new 160-bits big number as the new session key nK_{sess}, and select randomly a small integer nn. These two values form a pair, and are encrypted together with the K_{prim} as the content of CHAL message(see figure 5.4).

The reason to encrypt the challenge with primary key, instead of the session key, is to defense from the old session key compromise. Since the new session key are transmitted from the wire, if we use old session key for protection, once the old key is compromised , all the successive messages could be decrypted. With a primary key, which has never been transmitted through wire, it's not possible to compromise the current communication, even if the old session key is discovered.

After sending this message, the service process changes the checking variables to the newly generated values, and sets a timer.

$s\_chkn = nn$

$s\_chkx = Rand(s\_chkn, K_{prim}, nK_{sess})$

If it can not receive a correct PORT message in time, the service process will terminate itself. If there are some incorrect messages from the client, the service process will resend the CHAL message, for the concern of message loss or message corruption in both directions.

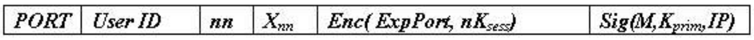
5.2.3 Response with Port Number: PORT

When the client receives the CHAL message in time, it will check the random number and random sequence number with its checking variable: c_chkx and c_chkn Then it decrypts the content and extracts the challenge

$(nKsess, nn) = Dec(Enc((nK_{sess}, nn), K_{prim}), K_{prim})$

The client needs to calculate the new X_{nn} with the received nn , nK_{sess}. and K_{prim} that it owns.

$X_{nn} = Rand(nn, K_{prim}, nK_{sess})$



PORT	User ID	nn	X_{nn}	Enc(ExpPort, nK_{sess})	Sig(M,K_{prim},IP)

Figure 5.5: PORT message format.

The PORT message should be sent with the calculated X_{nn}. , and encrypted expected port number with session key , as follow(see figure 5.5):

This mechanism could guarantee that only after the client has received the challenge correctly, could a correct PORT message be created. And, only the party with nK_{sess} could know the expected port number.

After sending PORT, the client updates its checking variable c_chkx and c_chkn:

$$c\_chkn = nn + 1$$

$$c\_chkx = Rand(nn + 1, K_{prim}, nK_{sess})$$

And then it starts the *"W/T process"*.

5.2.4 Open is Possible: OPEN

When the service process receives PORT correctly, it checks the random sequence number, and random number against checking variables, and decrypts the EEP for port number. If such port number is allowed by firewall policy to be assigned to the user, it will send OPEN message, otherwise quit the process. If it is allowed, the service process then updates its variables and sends OPEN to client:

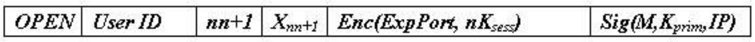
$$s\_curKsess = nK_{sess}$$

$$s\_curn = nn + 1$$

$$s\_curx = Rand(s\_curn, K_{prim}, s\_curKsess)$$

$$s\_chkn = s\_chk + 2$$

$$s\_chkx = Rand(s\_chkn, K_{prim}, s\_curKsess)$$



OPEN	*User ID*	*nn+1*	X_{nn+1}	*Enc(ExpPort,* nK_{sess}*)*	*Sig(M,*K_{prim}*,IP)*

Figure 5.6: OPEN message format.

The message OPEN is shown in figure 5.6:

After sending the message, the service process starts a timer, and waits for KEEP or TERM message. If it can not get either of these two types of messages correctly, the service process will terminate itself.

If the process receives PORT message that is just processed, it could be possibly because the client has not received the OPEN message in time, and send the PORT message again. The process, then, will resend the OPEN message, but will not reset the timer. Other received messages are ignored.

An attacker could perform a replay attack here, but since the timer will not be reset for resending, the service process will finally terminate when time out or continue if a correct TERM or KEEP is received from client.The replay attack can not get success. The processing of KEEP is discussed in step 6, and the processing of TERM is discussed in step 8.

5.2.5 Keep Alive: KEEP

When the client receives OPEN message correctly, it checks the message the same way as before, and updates the status variables:

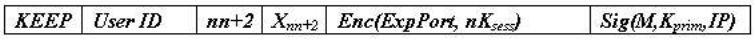
$c\_curKsess = nK_{sess}$

$c\_chkn = c\_chkn + 2$

$c\_chkx = rand(c\_chkn, K_{prim}, c\_curKsess)$

$c\_curn = c\_chkn - 1$

$c\_curx = rand(c\_curn, K_{prim}, c\_curKess)$



KEEP	User ID	nn+2	X_{nn+2}	$Enc(ExpPort, nK_{sess})$	$Sig(M, K_{prim}, IP)$

Figure 5.7: KEEP message format.

After this, it checks whether the user wants to quit without opening the firewall. If it is not the case, the client sends KEEP message as follow and start *“W/T process”*. Otherwise it sends TERM, the process is the same as the process when the user wants to quit after the firewall is opened, as discussed in the last part of step 7. The KEEP message is shown in figure 5.7.

5.2.6 Acknowledge and Continue: ACK

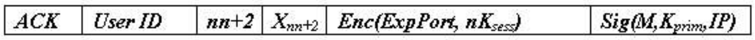
When the service process receives the KEEP message correctly, it will check the message the same way as before, and update the status variables:

$s\_curn = s\_curn + 2$

$s\_curx = Rand(s\_curn, K_{prim}, s\_curKsess)$

$s\_chkn = s\_chk + 2$

$s\_chkx = Rand(s\_chkn, K_{prim}, s\_curKsess)$



ACK	User ID	nn+2	X_{nn+2}	Enc(ExpPort, nK_{sess})	Sig(M,K_{prim},IP)

Figure 5.8: ACK message format.

Then, the process opens the firewall and sends ACK message (see figure 5.8) to the client.

After this, the service process sets the timer and starts to wait for another KEEP or TERM message. If it could not get correct KEEP or TERM message in time, the service process will close the firewall and terminate itself.

If the process receives KEEP message that is just processed, it could be possibly because the client has not received the ACK message in time, and send KEEP for the second time. The process, then, will resend the ACK message. Other received messages are ignored. An attacker could perform a replay attack here, as ever. But the attack could not influence too much because the timer will not be reset.

5.2.7 Continue or Terminate? TERM

When the client receives the ACK correctly, it will check the message and update the variable,

$c\_chkn = c\_chkn + 2$

$c\_chkx = rand(c\_chkn, K_{prim}, c\_curKsess)$

$c\_curn = c\_chkn - 1$

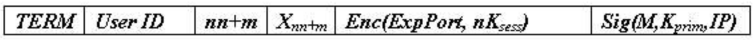
$c\_curx = rand(c\_curn, K_{prim}, c\_curKess)$

Then it waits for a period of time and checks whether the user wants to close the firewall, if not, the client goes back to step 5 to send the KEEP message.

The client - server system just repeats between step 5, 6 and 7 until

1. the user wants to quit

2. communication time out



TERM	*User ID*	*nn+m*	X_{nn+m}	*Enc(ExpPort, nK$_{sess}$)*	*Sig(M,K$_{prim}$,IP)*

Figure 5.9: TERM message format.

When the user wants to quit, the client will send a TERM message as shown in figure 5.9, and start a *"W/T process"*:

5.2.8 Acknowledgement to Termination: TACK

When the service process receives TERM message correctly, it checks the TERM message against checking variables, and closes the firewall. Afterwards, it updates the status variables as follows:

$s\_curn = s\_curn + 2$

$s\_curx = Rand(s\_curn, K_{prim}, s\_curKsess)$

$s\_chkn = s\_chk + 2$

$s\_chkx = Rand(s\_chkn, K_{prim}, s\_curKsess)$

And then save the status variables into the history file with the following values:

$s\_curn : nn + m + 1$

$s\_curx : Rand(nn + m + 1, K_{prim}, nK_{sess})$

$s\_chkn : nn + m + 2$

$s\_chkx : Rand(nn + m + 2, K_{prim}, nK_{sess})$

$s\_curUID : userID$

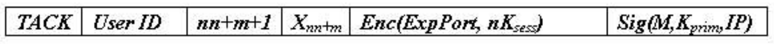
$s\_curKsess : nK_{sess}$

$s\_processSuc : true$

$s\_oldchkn : n$

$s\_oldchkx : X_n$

$s\_oldKsess : K_{sess}$



TACK	User ID	nn+m+1	X_{nn+m}	Enc(ExpPort, nK_{sess})	Sig(M,K_{prim},IP)

Figure 5.10: TACK message format.

After saving, it sends the message TACK as shown in figure 5.10, to acknowledge the termination request.

After sending, the service process terminates.

5.2.9 Save Before Termination

When the client receives TACK correctly, it checks the message, and updates status variables:

$c\_chkn = c\_chkn + 2$

$c\_chkx = rand(c\_chkn, K_{prim}, c\_curKsess)$

$c\_curn = c\_chkn - 1$

$c\_curx = rand(c\_curn, K_{prim}, c\_curKess)$

Then, the client saves the status variables with values as follow:

$c\_curn : nn + m + 2$

$c\_curx : Rand(nn + m + 2, K_{prim}, nK_{sess})$

$c\_chkn : nn + m + 3$

$c\_chkx : Rand(nn + m + 32, K_{prim}, nK_{sess})$

$c\_curUID : userID$

$c\_curKsess : nK_{sess}$

After saving the variables, the client process has accomplished its mission and terminates.

After it has opened the firewall, if there's a communication time out at server side, the service process will suppose the client has quit the process suddenly. It then closes the firewall and terminates, without saving the status variables.

We could have a more detailed view of the whole communication process as shown in figure 5.11

5.3 Summary

Eight types of messages are designed in the protocol. A session includes the following main steps:

- *"ASK"* is sent from client, the first message for a session.
- *"CHAL"* is sent from server to challenge the client.
- *"PORT"* is sent from client then to reply such challenge and indicate the expected port number that the client wants to open.
- *"OPEN"* is sent from server after validation, indicating the firewall could be opened for such port.
- *"KEEP"* is sent from client, saying that the client surely want to open the firewall. The server then opens the firewall.
- *"ACK"* is sent from server, indicating the firewall has opened, and the client should continue to send the keep-alive message.
- *"KEEP"* and *"ACK"* are continuously sent between client and server, until ...

- *"TERM"* is sent from client, when the client wants to close the firewall. The server will close the firewall when such message is received.
- *"TACK"* is sent from server, indicating the server has closed the firewall, the session could be closed now.

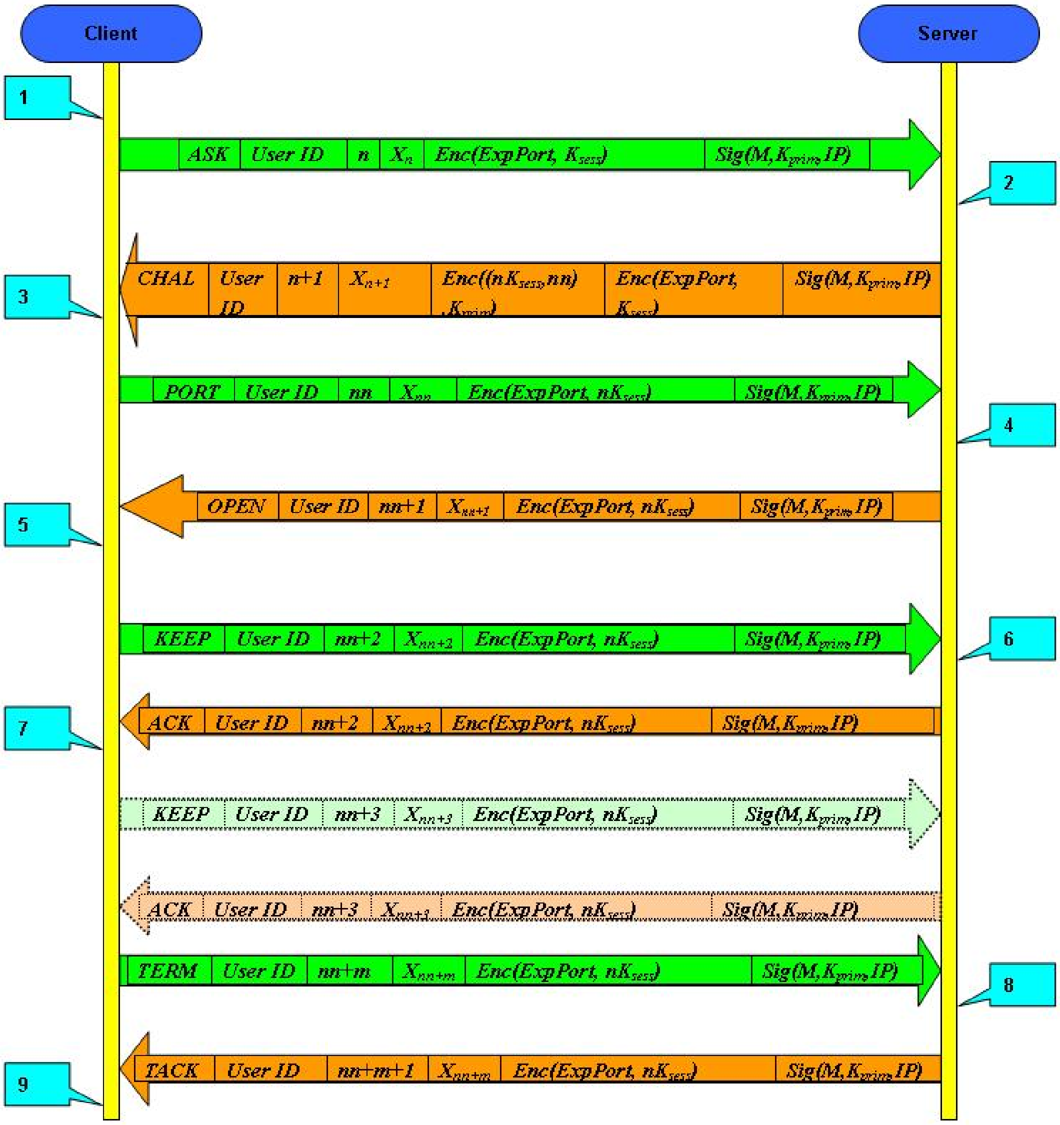

Client
Server
1
ASK | User ID | n | X_n | Enc(ExpPort, K_{sess}) | Sig(M,K_{prim},IP)
2
3
CHAL | User ID | n+1 | X_{n+1} | Enc((nK_{sess},nn) ,K_{prim}) | Enc(ExpPort, K_{sess}) | Sig(M,K_{prim},IP)
PORT | User ID | nn | X_{nn} | Enc(ExpPort, nK_{sess}) | Sig(M,K_{prim},IP)
4
5
OPEN | User ID | nn+1 | X_{nn+1} | Enc(ExpPort, nK_{sess}) | Sig(M,K_{prim},IP)
6
KEEP | User ID | nn+2 | X_{nn+2} | Enc(ExpPort, nK_{sess}) | Sig(M,K_{prim},IP)
7
ACK | User ID | nn+2 | X_{nn+2} | Enc(ExpPort, nK_{sess}) | Sig(M,K_{prim},IP)
KEEP | User ID | nn+3 | X_{nn+3} | Enc(ExpPort, nK_{sess}) | Sig(M,K_{prim},IP)
ACK | User ID | nn+3 | X_{nn+3} | Enc(ExpPort, nK_{sess}) | Sig(M,K_{prim},IP)
TERM | User ID | nn+m | X_{nn+m} | Enc(ExpPort, nK_{sess}) | Sig(M,K_{prim},IP)
8
9
TACK | User ID | nn+m+1 | X_{nn+m} | Enc(ExpPort, nK_{sess}) | Sig(M,K_{prim},IP)

Figure 5.11: Detailed Message Transmission of One Session.

Chapter 6

Architecture Design and Protocol Modeling

The architecture design gives the architectural view of integrating protocol implementation into the existing Grid service architecture. The protocol modeling is the implementation of ideas in conceptual design in forms of state machines. We use a state machine modeling and verification tool — UPPAAL to simulate and test the design.

6.1 Architecture Design

The aim of the architecture design is to fit the protocol implementation into the Grid platform seamlessly.

As a firewall control mechanism, AGF-Protocol runs upon IP network layer. The firewall ports need to be opened at both client and server sides before a Grid service could really get start. From an architecture overview, the implementation is in the basic layer for Grid service, as shown in figure 6.1

In order to open the remote firewall before accessing the services, the client needs first authenticate itself to the firewall by AGF-Protocol, gets the ports

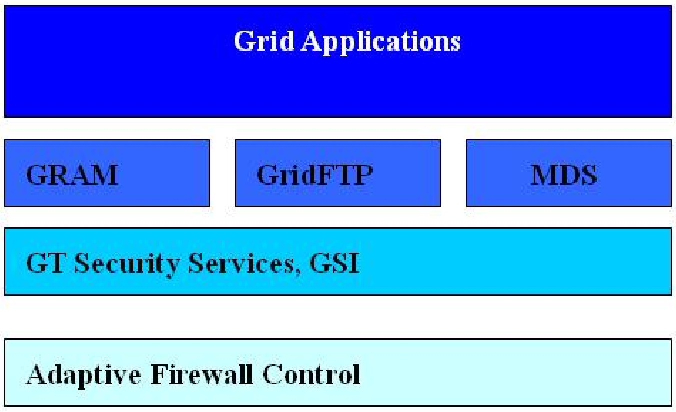


Figure 6.1: Adaptive Firewall Control Protocol in the Whole GT Architecture

opened, and connects to the services behind the firewall and starts a higher level session.

During the high-level session, Keep-Alive and Acknowledgement messages will be exchanged between client and the firewall to keep the ports open. If the client quit abnormally, for example when the service gets stuck. The client will not send Keep-Alive messages any more, and the server firewall ports will be closed automatically when time is out for receiving such messages.

When the high-level session terminates, the client will inform the firewall to close the corresponding ports that have been opened for it, and quit the process.

One of the possibilities for integrating such mechanism into existing Grid environment is using *"Wrappers"*. A wrapper is a script, which could combine an existing Grid application, no matter service or client, with the implementation of Adaptive Firewall Control protocol, and make them run in parallel.

It is possible to configure the wrapper about the destination and port numbers that a Grid client needs to open. Sometimes, a Grid service needs to act as a client to get the sub-services from other Grid sites. Then, we could configure the wrapper of a Grid service like a Grid client, so that when such service starts, it opens the remote firewalls, and closes them when it quits.

Because of time limit, we haven't explored to find other possibilities. Such work should be done in the future.

6.2 Protocol Modelling

The design of the protocol is an evolutionary process like a spiral. The design need to be improved based on the test of it and redesign. The process includes several recursive steps as follows:

1. model the design
2. test the model
3. find the defects in design from modeling result
4. modify the design and return to step 1

The main purpose of protocol modeling is for testing the ideas in protocol design. So that we could have a detailed view of the protocol, find the limitations for some original ideas, and modify the design to be better.

All these activities are performed on a simulation platform - UPPAAL.

6.2.1 UPPAAL and Modelling

Uppaal is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types. [19] It is appropriate for communication protocols that can be modeled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or shared variables. [21]

Uppaal consists of a description language, a simulator and a model-checker.

- The description language is a non-deterministic guarded command language with data types (e.g. bounded integers, arrays, etc.). [21] It is used as a modeling language to describe protocol behavior as networks of timed automata.
- The simulator is a validation tool which enables protocol examination during early design stage. The platform offers a virtual representation of the dynamic behavior of the modeled protocol. Many faults during the design are found in such simulation, such as deadlock.

- The model-checker can check invariant and reachability properties by exploring the state-space of a system. [21] Some of the protocol features are checked by this tool. The following protocol features are turned into requirement specification expressions and checked:
  - The protocol is robust from message loss
  - The protocol is robust from message repeat
  - The protocol is robust from message delay

6.2.2 State Machines

The protocol is modeled in four state machines:

- Client
- Server
- Timer for Client
- Timer for Server.

A protocol can not run without its environment, which includes:

- User at client side
- Communication Media from client to server
- Communication Media from server to client

All the above three entities are modeled as independent process, which communicate with others by shared variables and channels. The channel here, in UPPAAL modeling language, is a kind of special data type for modeling event raising and receiving. Such mechanism is used in protocol modeling message sending/receiving and synchronization between processes.

The client and server models describe the client/server behaviors in the protocol. These two models simulate the behavior of encryption, decryption, pseudorandom number generation, message sending and listening, time counting and so on. The models are attached as appendix in figure A.2 and figure A.1.

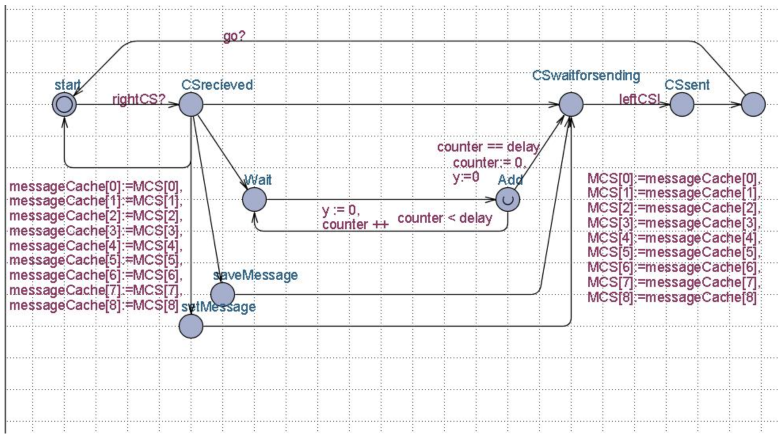


Figure 6.2: The state machine of Communication Media from Client to Server

The client and server are modeled to communicate with each other through the communication media processes. When the server sends a message out, it puts the data of message into a globally shared array and notifies the communication media through a channel. The communication media will then notify the client that there comes a message from server. When the client sends a message out, it is similar, as shown in figure 6.3 and figure 6.2

Each communication media takes care of the communication of one direction. Both of them are designed with the ability to lost, delay and resend messages. So that we could test some protocol features as mentioned before. Some other features, such as the abilities to defense from substitution attack and replay attack, are difficult to model and are out of the scope of this project. We prove such features by other means.

The time counting processes are modeled as independent state machines, like *"watches on the wall"*, as figure 6.5 and figure 6.4 represent. The client and server can set their timers by special channels, and get notification from other channels when time is out.

The client user (figure 6.6) is modeled as a user with random behaviors, requesting the client to open or close the firewall randomly as time goes.

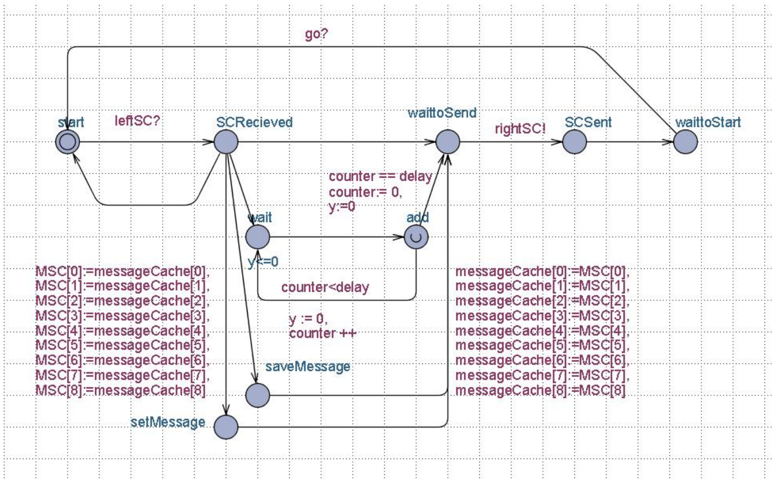


Figure 6.3: The state machine of Communication Media from Server to Client

6.3 Summary

The implementation of AGF-Protocol could be integrated into existing Grid environment by *"Wrappers"*. The protocol are modeled in timed automata and tested in UPPAAL.The tested features include robustness from message loss, repeat and delay. Other features are hard to model, and will be proved by other methods.

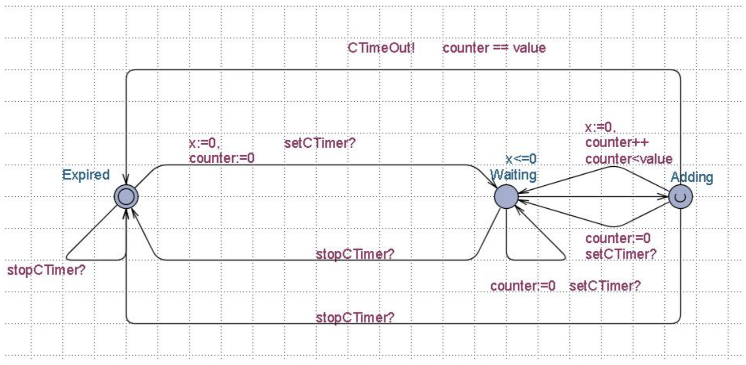


Figure 6.4: The state machine of Client Timer Process

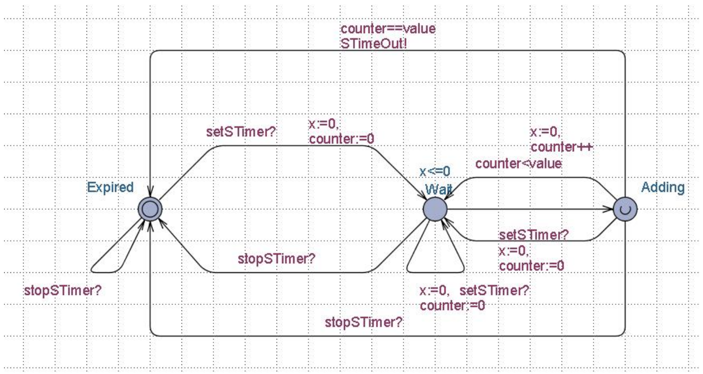


Figure 6.5: The state machine of Server Timer Process

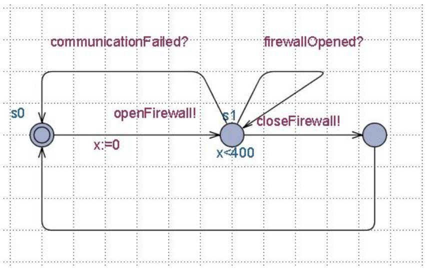


Figure 6.6: The state machine of Client User Process

Chapter 7

Protocol Implementation

7.1 Software Architecture

The protocol is implemented in C++, compiled and tested on Linux system. The software suit includes a Key pair generator, a server, and a client.

7.1.1 Key Generation

The key pair generator is designed to generate user ID based on given user name, create corresponding key pair for the server and client, so that, they can communicate for the first time(figure 7.1). The key pair will be stored together with other items in the history file, which will be updated after each session.

7.1.2 Server

The server and client are implemented with the help of pseudorandom bit generation, UNIX socket and packet filtering technology.

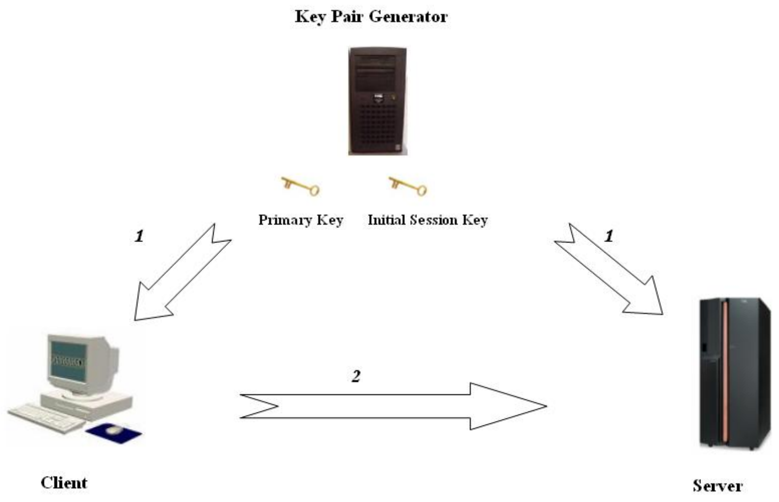


Figure 7.1: Key pair generation Architecture

The program implements pseudorandom number generator, following the Federal Information Processing Standard 186 (FIPS 186), which is originally designed for DSA (Digital Signature Algorism) private keys. The algorithm could generate 160-bit pseudorandom number sequence based on the given prime and seed, which is, in our application, the $K_{prim}$and K_{sess}.

The library *"PCAP"* is a *"Packet Capture library, provides a high level interface to packet capture systems. All packets on the network, even those destined for other hosts, are accessible through this mechanism"*. [13]The advantage of using such library, is that it could work in front of the software firewalls. Because of the requirements, the system must be able to catch the packets even though the software firewall has blocked all types of traffics. The communication could be seen as a kind of knocks to the firewall. On correct knocks, the firewall will open.

The server uses multithread technology to process the requests from multiple clients. The architecture of server is shown in figure 7.2. For each user, there is a servant thread and an event queue created for it. The *"Timer Thread"* could handle all the time counting requests from servant threads, distinguishing each other by user ID. Then, when time is out, it will push the timeout event into the corresponding event queue.

The message and time out processing of each servant thread is the same as the one of *"Client Thread"*, which will be explained later. The *"PCAP packet In-*

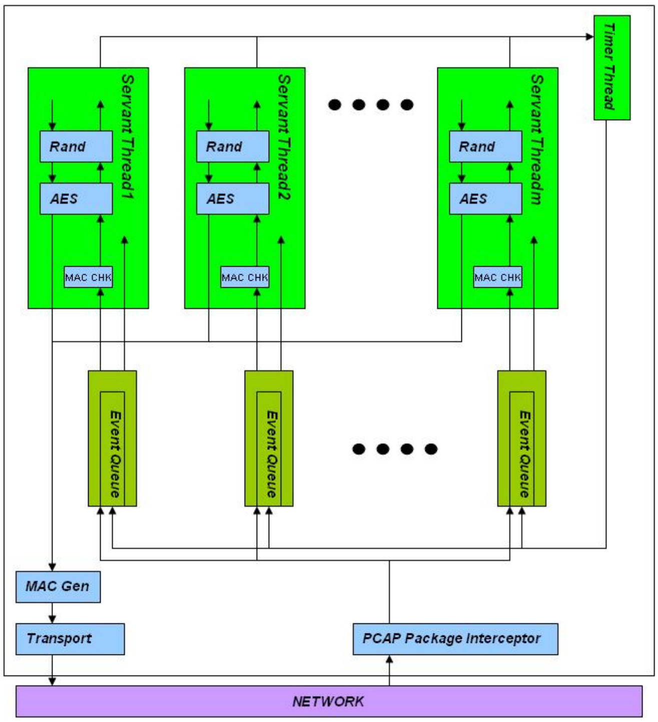


Figure 7.2: Server Software Architecture

terceptor" module for server has the ability of recognizing messages for different servant threads by user ID, and could dispense them to different event queues.

7.1.3 Client

The architecture of client is designed as in figure 7.3. There are three threads in the runtime. *"User IO Thread"* serves as a user interface gets the command of open/close firewall, and prints the current status on to the screen. The *"Timer thread"* waits the requests from *"Client Thread"* - set, stop and reset. When time is out, it will create a timeout event and push it to the *" Event Queue"*. *" Client Thread"* is the brain of whole program, gets the events from event queue, checks and generates messages, and sends them out.

"Client Thread" manages the process of message generation. It collects all the necessary information, such as message types and contents, and then utilizes the

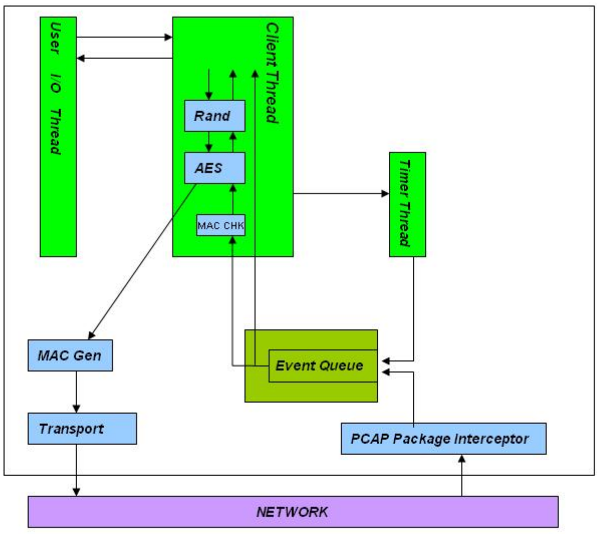
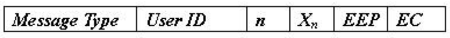
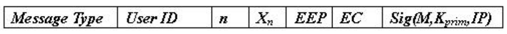


Figure 7.3: Client Software Architecture



Message Type	User ID	n	X_n	EEP	EC

Figure 7.4: The Message Created by Client Thread



Message Type	User ID	n	X_n	EEP	EC	Sig(M,K_{prim},IP)

Figure 7.5: The Message Sent Through Network

"Rand" module to generate the pseudorandom number in sequence and encrypt the content by *"AES"* module. The message that is generated is shown in figure 7.4.

Then the message is passed to *"MAC gen"* module, to attach message authentication code to it. Afterwards, the message is turned into network byte-order and sent as an IP packet by module " *Transport"*. The format of the message is shown in figure 7.5.

" Client Thread" gets the new messages and time out events from *"Event Queue"*. When a new relevant message is captured from network, it will be pushed as an event into event queue. The *"PCAP packet Interceptor"* is pro-

grammed with filter function, so that only the messages that belong to AGF-Protocol would be put into *“Event Queue”*. Having received the message, the *“Client Thread”* decrypts and checks the message by module *“MAC CHK”*, *“AES”*, *“Rand”* to test message authentication code, the content, and check the pseudorandom number.

7.2 Test

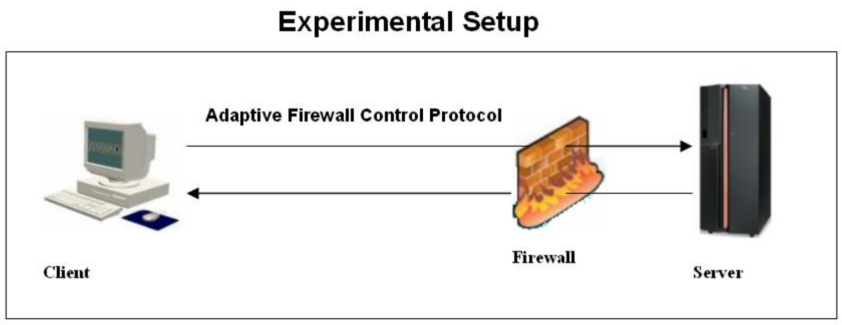


Figure 7.6: Test Enviroment Setup

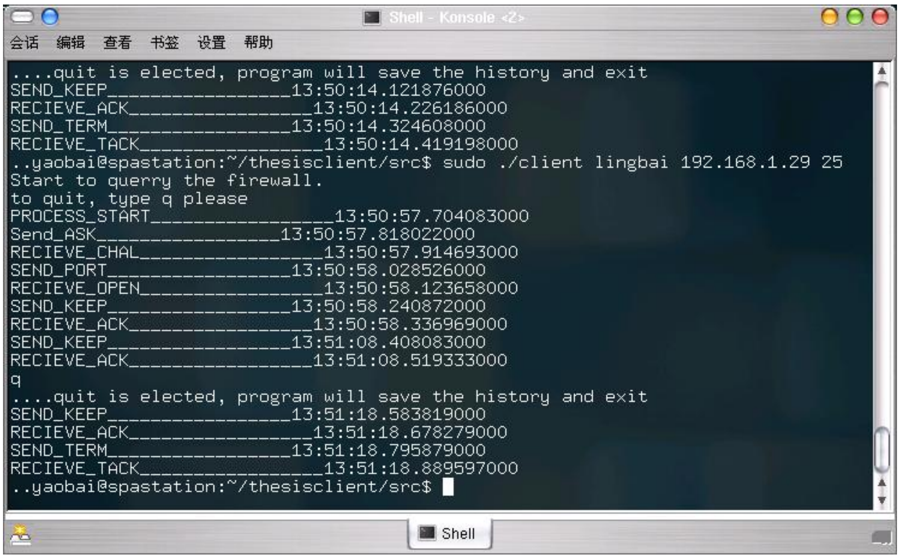
As shown in figure 7.6 Once started, the client will load the history files, which contain the key pairs, and send *“ASK”* messages to server. The server, having loaded the history file, will authenticate the clients. Once the client has authenticated itself, the server opens the firewall by modifying the firewall rules tempararyly of *“iptables”* .

Figure 7.7 and Figure 7.8 show the whole process of a session. The client at 192.168.1.1 tryes to open port 25 at 192.168.1.29 using a user name *“lingbai”*, whose user ID and key pair has already deployed at both client and server side. Every time there’s a message received or sent, both sides will record the time of receiving and sending, as the time values shown in Figure7.7 and Figure 7.8.

The test is performed on two machines with the following configuration:

Server:

- *Intel Celeron 2.2G (M)*



```
....quit is elected, program will save the history and exit
SEND_KEEP_________________13:50:14.121876000
RECIEVE_ACK_________________13:50:14.226186000
SEND_TERM_________________13:50:14.324608000
RECIEVE_TACK_________________13:50:14.419198000
..yaobai@spastation:~/thesisclient/src$ sudo ./client lingbai 192.168.1.29 25
Start to querry the firewall.
to quit, type q please
PROCESS_START_________________13:50:57.704083000
Send_ASK_________________13:50:57.818022000
RECIEVE_CHAL_________________13:50:57.914693000
SEND_PORT_________________13:50:58.028526000
RECIEVE_OPEN_________________13:50:58.123658000
SEND_KEEP_________________13:50:58.240872000
RECIEVE_ACK_________________13:50:58.336969000
SEND_KEEP_________________13:51:08.408083000
RECIEVE_ACK_________________13:51:08.519333000
q
....quit is elected, program will save the history and exit
SEND_KEEP_________________13:51:18.583819000
RECIEVE_ACK_________________13:51:18.678279000
SEND_TERM_________________13:51:18.795879000
RECIEVE_TACK_________________13:51:18.889597000
..yaobai@spastation:~/thesisclient/src$ 
```

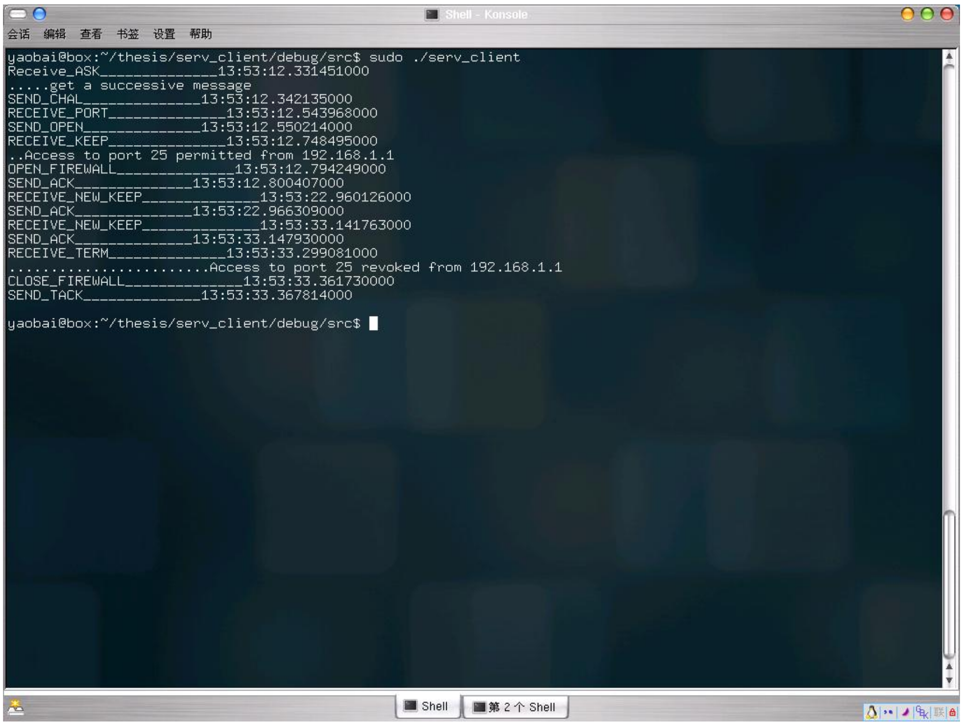
Figure 7.7: The Client Output

- *256MB DDR Ram*
- *100MB/10MB Ethernet Adapter*
- *LINUX 2.6*

Client:

- *Intel Pentum II 333*
- *224MB Ram*
- *10MB Ethernet*
- *LINUX 2.6*

The performance for opening a firewall port is tested. The time counting starts from the begining of client process until when it receives the first *"ACK"* message, which means that the remote firewall port has been opened. We get the following data:



```
yaobai@box:~/thesis/serv_client/debug/src$ sudo ./serv_client
Receive_ASK______________13:53:12.331451000
.....get a successive message
SEND_CHAL______________13:53:12.342135000
RECEIVE_PORT______________13:53:12.543968000
SEND_OPEN______________13:53:12.550214000
RECEIVE_KEEP______________13:53:12.748495000
..Access to port 25 permitted from 192.168.1.1
OPEN_FIREWALL______________13:53:12.794249000
SEND_ACK______________13:53:12.800407000
RECEIVE_NEW_KEEP______________13:53:22.960126000
SEND_ACK______________13:53:22.966309000
RECEIVE_NEW_KEEP______________13:53:33.141763000
SEND_ACK______________13:53:33.147930000
RECEIVE_TERM______________13:53:33.299081000
........................Access to port 25 revoked from 192.168.1.1
CLOSE_FIREWALL______________13:53:33.361730000
SEND_TACK______________13:53:33.367814000

yaobai@box:~/thesis/serv_client/debug/src$ █
```

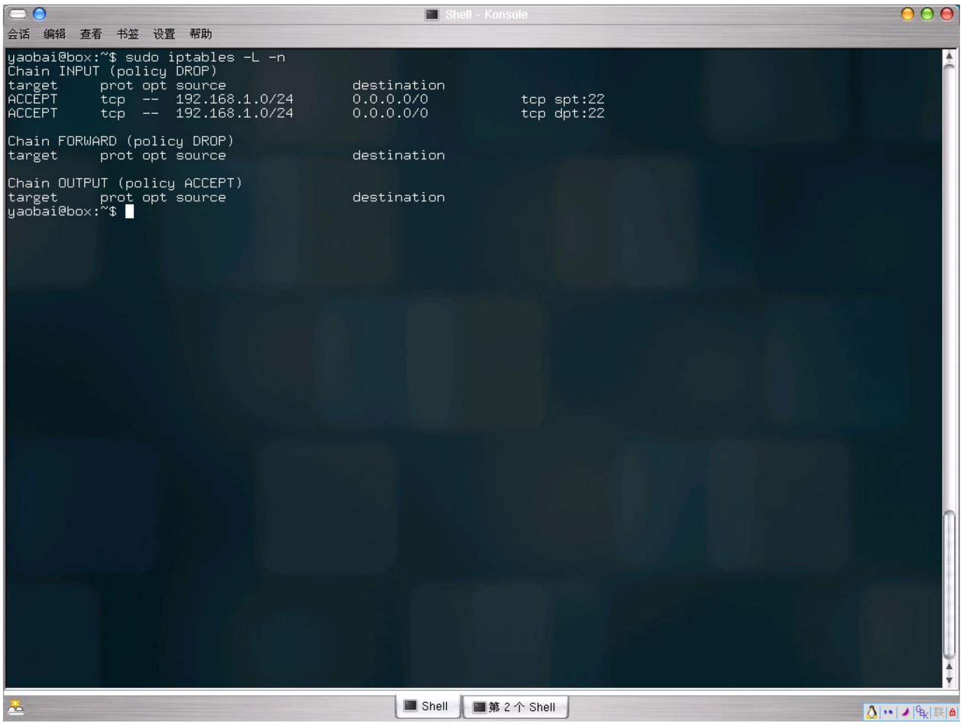
Figure 7.8: The Server Output

No.	Time	No.	Time
001	0.56s	006	0.58s
002	0.58s	007	0.58s
003	0.58s	008	0.58s
004	0.58s	009	0.52s
005	0.59s	010	0.58s

The average time to open a port is 0.57 second.

Figure 7.9, 7.10 and 7.11 show the status of firewall rules before, during and after the session. As we can see, at the beginning , the firewall does not accept any incomming connection except port 22.

After the client is authenticated to server, a new chain ,named " *KNOCKERS_eth0*" is added to the fireall rules, to open port 25 for incomming connection from 192.168.1.1.



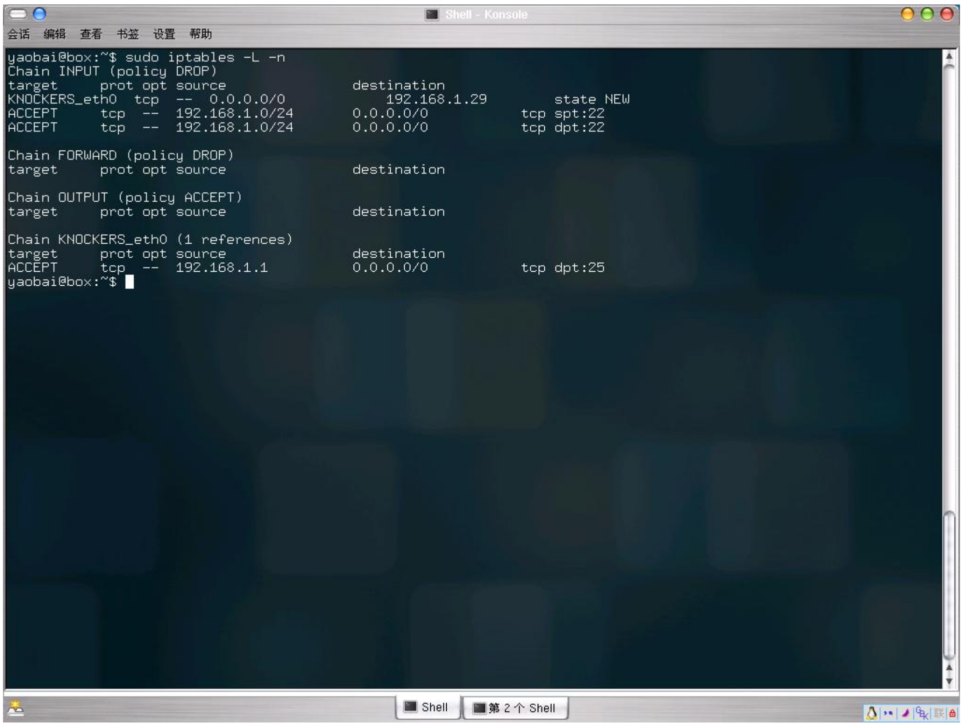
```
yaobai@box:~$ sudo iptables -L -n
Chain INPUT (policy DROP)
target     prot opt source               destination
ACCEPT     tcp  --  192.168.1.0/24       0.0.0.0/0           tcp spt:22
ACCEPT     tcp  --  192.168.1.0/24       0.0.0.0/0           tcp dpt:22

Chain FORWARD (policy DROP)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
yaobai@box:~$
```

Figure 7.9: The Status of IPTABLES before Client Connects

Then, after the client send *"TERM"* message, the firewall closes port 25 by removing rule *"ACCEPT tcp – 192.168.1.1 0.0.0.0 tcp dtp:25"* from chain*" KNOCKERS_eth0"*.

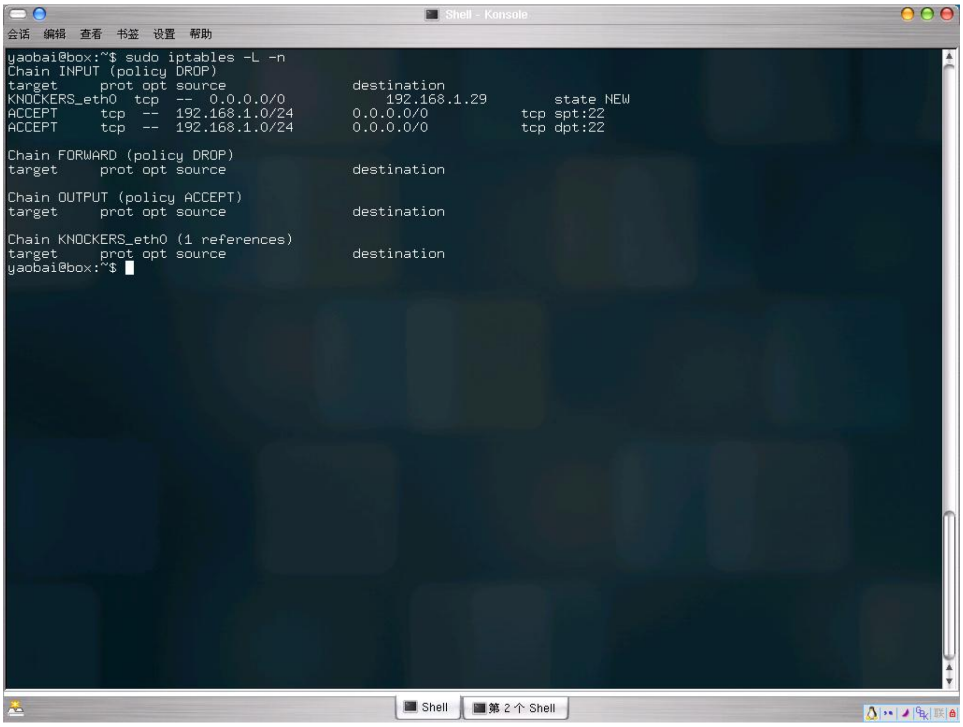
7.3 Summary

The software suit includes three programs: Key pair generator, server and client. The pseudorandom number generation follows the standard of FIPS 186. The communication between client and server could penetrate software firewall with the help of *"PCAP"* library. We have performed tests on performance and the feasibility of opening and closing firewall to verify the design ideas. But the tests about the robustness of the protocol from attacks and failures are not performed because of time and environment limits. The current work has only proved such features in theory.



```
yaobai@box:~$ sudo iptables -L -n
Chain INPUT (policy DROP)
target     prot opt source               destination
KNOCKERS_eth0  tcp  --  0.0.0.0/0            192.168.1.29        state NEW
ACCEPT     tcp  --  192.168.1.0/24       0.0.0.0/0           tcp spt:22
ACCEPT     tcp  --  192.168.1.0/24       0.0.0.0/0           tcp dpt:22

Chain FORWARD (policy DROP)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination

Chain KNOCKERS_eth0 (1 references)
target     prot opt source               destination
ACCEPT     tcp  --  192.168.1.1          0.0.0.0/0           tcp dpt:25
yaobai@box:~$
```

Figure 7.10: The Status of IPTABLES when firewall is Opened



```
yaobai@box:~$ sudo iptables -L -n
Chain INPUT (policy DROP)
target     prot opt source               destination
KNOCKERS_eth0  tcp  --  0.0.0.0/0            192.168.1.29        state NEW
ACCEPT     tcp  --  192.168.1.0/24       0.0.0.0/0           tcp spt:22
ACCEPT     tcp  --  192.168.1.0/24       0.0.0.0/0           tcp dpt:22

Chain FORWARD (policy DROP)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination

Chain KNOCKERS_eth0 (1 references)
target     prot opt source               destination
yaobai@box:~$ 
```

Figure 7.11: The Status of IPTABLES when the firewall port is Closed

Chapter 8

Conclusion and Future Work

The current work is basically around finding the requirements for a Grid Adaptive Firewall, proposing a conceptual design and offering a simple implementation.

We analysis the different firewall requirements of services in two different architectures of Grid - Hourglass and OGSA. Then the possible threats to AGF-Protocol, when it is exposed to open network, are collected, which include several types of related attacks, as well as system failures and communication failures. Based on the above steps, we draw a conclusion on the requirements for adaptive Grid firewall.

To design a protocol that could meet all the requirements, we use several mechanisms in conceptual design: AES, MAC, Challenge-Response, Primary key/session key, FIPS 186 Pseudorandom Number Sequence Generation, Time-out/Resend , and so on.

Eight types of messages are designed in the protocol. We design the format of messages and the behavior of communicaiton parties during one session. We also give the suggestions for integrating the implementation into existing Grid environment by *"Wrappers"*.

We have model and tested the protocol in timed automata with the help of a verification tool – UPPAAL.The robustness from message loss, repeat and delay are tested.

A simple implementation to verify ideas is developed, which includes three programs: Key pair generator, server and client. The client and server could penetrate software firewall with the support of *"PCAP"* library.

Some simple tests are performed on the implementation. The robustness of the protocol from attacks and failures are proved in theory. Further tests about such features should be done in the future.

Currently, the protocol is , in theory, immune from man in the middle attacks, brute force attacks, and dictionary attacks. At the same time, it is robust against system and communication failure. The implemenation could penetrate a software firewall and open/close the firewall ports as requested.

The current implemenation could only open one port in each session, a future work should improve it to open a range of ports or multiple ports in one session.

The firewall that we use for test is a LINUX *"iptables"* firewall. Some of the commercial implementation use hardware firewall instead. One of the future work is to find a proper way of controlling the hardware firewall as well.

The current implementation does not offer a way to encrypt the *"history file"*, which contains key pair information. In the future, a utility to protect such files should be found.

The current implementation does not deal with too much about the key management and distribution. To design a mechanism that can be compliant with Grid authentication schemes and key delegation would be a future work to do.

Appendix A

Timed Automatas of Protocol

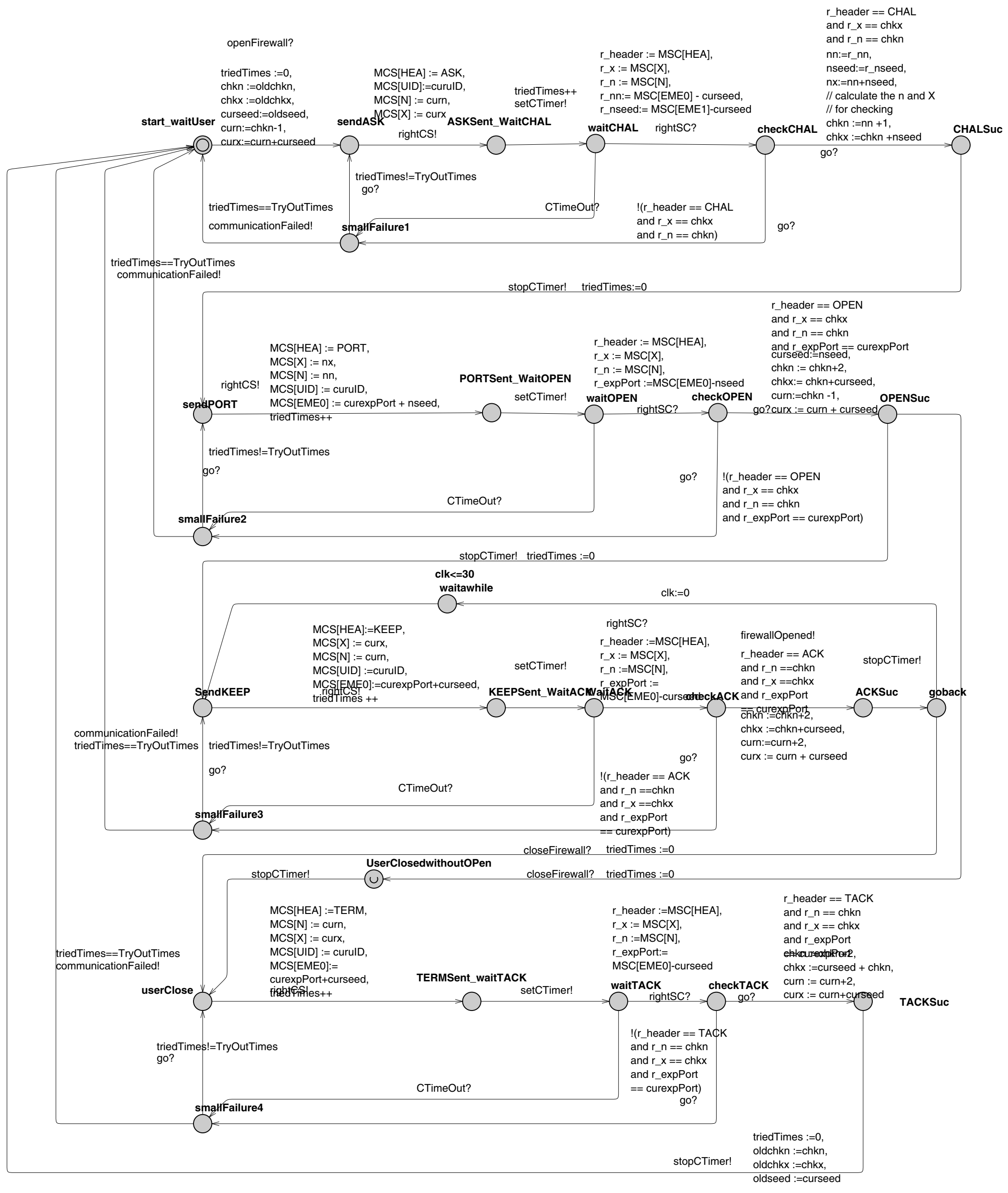
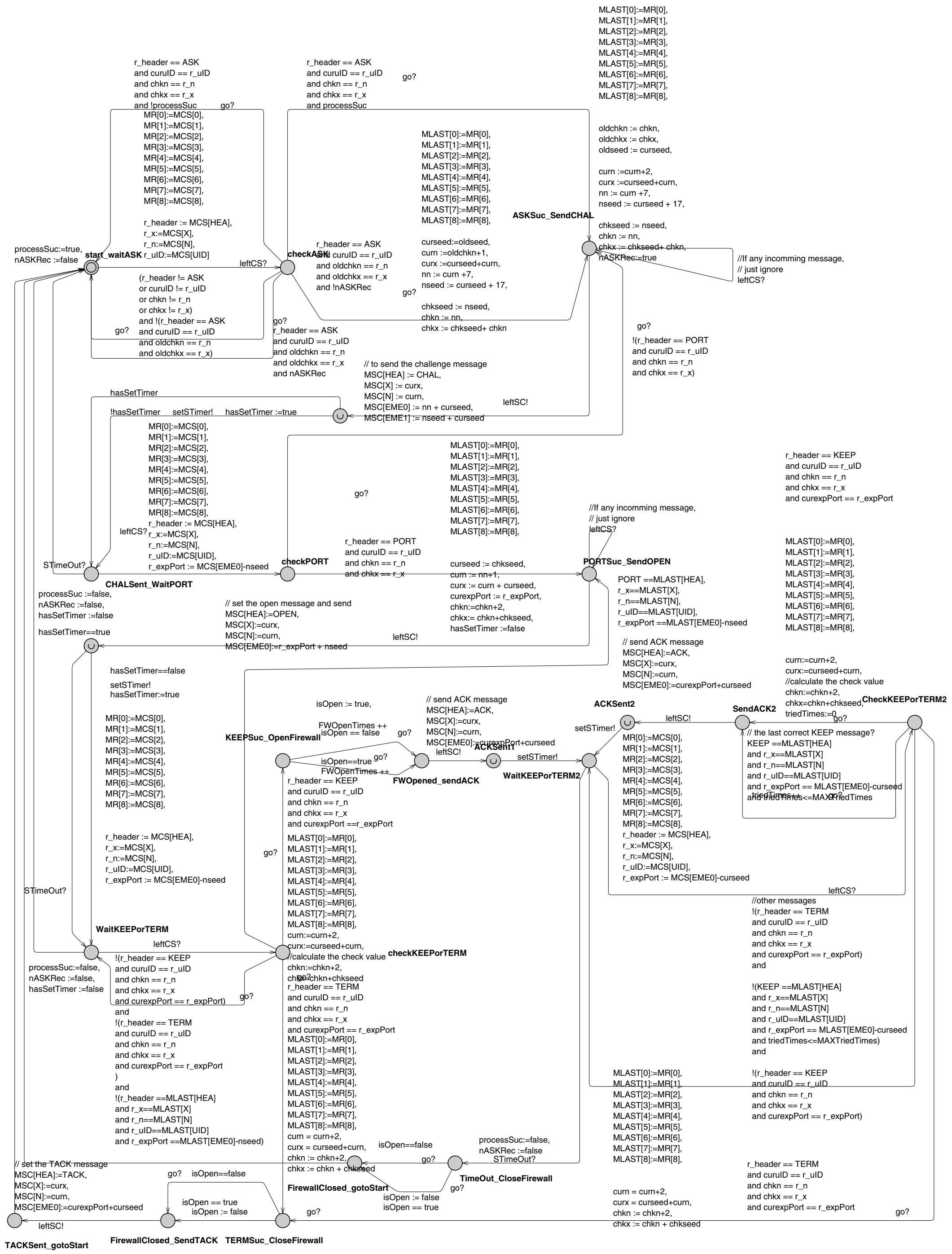


Figure A.1: The state machine of Client Process



start_waitASK
checkASK
ASKSuc_SendCHAL
CHALSent_WaitPORT
checkPORT
PORTSuc_SendOPEN
WaitKEEPorTERM
checkKEEPorTERM
KEEPSuc_OpenFirewall
FWOpened_sendACK
ACKSent1
WaitKEEPorTERM2
ACKSent2
SendACK2
CheckKEEPorTERM2
TimeOut_CloseFirewall
FirewallClosed_gotoStart
TACKSent_gotoStart
FirewallClosed_SendTACK
TERMSuc_CloseFirewall
r_header == ASK and curuID == r_uID and chkn == r_n and chkx == r_x and !processSuc
go?
MR[0]:=MCS[0], MR[1]:=MCS[1], MR[2]:=MCS[2], MR[3]:=MCS[3], MR[4]:=MCS[4], MR[5]:=MCS[5], MR[6]:=MCS[6], MR[7]:=MCS[7], MR[8]:=MCS[8],
r_header := MCS[HEA], r_x:=MCS[X], r_n:=MCS[N], r_uID:=MCS[UID]
processSuc:=true, nASKRec :=false
leftCS?
(r_header != ASK or curuID != r_uID or chkn != r_n or chkx != r_x) and !(r_header == ASK and curuID == r_uID and oldchkn == r_n and oldchkx == r_x)
r_header == ASK and curuID == r_uID and oldchkn == r_n and oldchkx == r_x and nASKRec
r_header == ASK and curuID == r_uID and chkn == r_n and chkx == r_x and processSuc
r_header == ASK and curuID == r_uID and oldchkn == r_n and oldchkx == r_x and !nASKRec
MLAST[0]:=MR[0], MLAST[1]:=MR[1], MLAST[2]:=MR[2], MLAST[3]:=MR[3], MLAST[4]:=MR[4], MLAST[5]:=MR[5], MLAST[6]:=MR[6], MLAST[7]:=MR[7], MLAST[8]:=MR[8],
curseed:=oldseed, curn :=oldchkn+1, curx :=curseed+curn, nn := curn +7, nseed := curseed + 17,
chkseed := nseed, chkn := nn, chkx := chkseed+ chkn
oldchkn := chkn, oldchkx := chkx, oldseed := curseed,
curn :=curn+2, curx :=curseed+curn, nn := curn +7, nseed := curseed + 17,
chkseed := nseed, chkn := nn, chkx := chkseed+ chkn, nASKRec:=true
//If any incomming message, // just ignore leftCS?
!(r_header == PORT and curuID == r_uID and chkn == r_n and chkx == r_x)
// to send the challenge message MSC[HEA] := CHAL, MSC[X] := curx, MSC[N] := curn, MSC[EME0] := nn + curseed, MSC[EME1] := nseed + curseed
leftSC!
hasSetTimer
!hasSetTimer setSTimer! hasSetTimer :=true
MR[0]:=MCS[0], MR[1]:=MCS[1], MR[2]:=MCS[2], MR[3]:=MCS[3], MR[4]:=MCS[4], MR[5]:=MCS[5], MR[6]:=MCS[6], MR[7]:=MCS[7], MR[8]:=MCS[8], r_header := MCS[HEA], r_x:=MCS[X], r_n:=MCS[N], r_uID:=MCS[UID], r_expPort := MCS[EME0]-nseed
STimeOut?
processSuc :=false, nASKRec :=false, hasSetTimer :=false
r_header == PORT and curuID == r_uID and chkn == r_n and chkx == r_x
curseed := chkseed, curn := nn+1, curx := curn + curseed, curexpPort := r_expPort, chkn:=chkn+2, chkx:= chkn+chkseed, hasSetTimer :=false
//If any incomming message, // just ignore leftCS?
PORT ==MLAST[HEA], r_x==MLAST[X], r_n==MLAST[N], r_uID==MLAST[UID], r_expPort ==MLAST[EME0]-nseed
// set the open message and send MSC[HEA]:=OPEN, MSC[X]:=curx, MSC[N]:=curn, MSC[EME0]:=r_expPort + nseed
hasSetTimer==true
hasSetTimer==false
setSTimer! hasSetTimer:=true
r_header := MCS[HEA], r_x:=MCS[X], r_n:=MCS[N], r_uID:=MCS[UID], r_expPort := MCS[EME0]-nseed
processSuc:=false, nASKRec :=false, hasSetTimer :=false
!(r_header == KEEP and curuID == r_uID and chkn == r_n and chkx == r_x and curexpPort == r_expPort) and !(r_header == TERM and curuID == r_uID and chkn == r_n and chkx == r_x and curexpPort == r_expPort) and !(r_header ==MLAST[HEA] and r_x==MLAST[X] and r_n==MLAST[N] and r_uID==MLAST[UID] and r_expPort ==MLAST[EME0]-nseed)
isOpen := true, FWOpenTimes ++
isOpen == false
isOpen==true FWOpenTimes ++
r_header == KEEP and curuID == r_uID and chkn == r_n and chkx == r_x and curexpPort ==r_expPort
curn:=curn+2, curx:=curseed+curn, //calculate the check value chkn:=chkn+2, chkx:=chkn+chkseed
// send ACK message MSC[HEA]:=ACK, MSC[X]:=curx, MSC[N]:=curn, MSC[EME0]:=curexpPort+curseed
setSTimer!
r_header == TERM and curuID == r_uID and chkn == r_n and chkx == r_x and curexpPort == r_expPort
curn = curn+2, curx = curseed+curn, chkn := chkn+2, chkx := chkn + chkseed
isOpen==false
processSuc:=false, nASKRec :=false STimeOut?
isOpen := false isOpen == true
// set the TACK message MSC[HEA]:=TACK, MSC[X]:=curx, MSC[N]:=curn, MSC[EME0]:=curexpPort+curseed
isOpen==false
isOpen == true isOpen := false
r_header == KEEP and curuID == r_uID and chkn == r_n and chkx == r_x and curexpPort == r_expPort
tried Times:=0
// the last correct KEEP message? KEEP ==MLAST[HEA] and r_x==MLAST[X] and r_n==MLAST[N] and r_uID==MLAST[UID] and r_expPort == MLAST[EME0]-curseed and triedTimes<=MAXTriedTimes
triedTimes++
//other messages !(r_header == TERM and curuID == r_uID and chkn == r_n and chkx == r_x and curexpPort == r_expPort) and !(KEEP ==MLAST[HEA] and r_x==MLAST[X] and r_n==MLAST[N] and r_uID==MLAST[UID] and r_expPort == MLAST[EME0]-curseed and triedTimes<=MAXTriedTimes) and !(r_header == KEEP and curuID == r_uID and chkn == r_n and chkx == r_x and curexpPort == r_expPort)
r_header == TERM and curuID == r_uID and chkn == r_n and chkx == r_x and curexpPort == r_expPort

Figure A.2: The state machine of Server Process

Bibliography

[1] Robin Sharp, IMM, DTU, November 2001: OSI and other Layered Architectures: Principles and Implementation

[2] Robin Sharp, Lynby, 2002 : Principles of Protocol Design 2nd Edition

[3] Ian Foster and Carl Kesselman : The GRID 2 , P46, Morgan Kaufmann , 2004

[4] The Anatomy of the Grid: Enabling Scalable Virtual Organizations. I. Foster, C. Kesselman, S. Tuecke, 8 Aug 2005 < *http* : *//www.globus.org/research/papers/anatomy.pdf* >

[5] The WS-Resource Framework< *http* : *//www.globus.org/wsrf/specs/ws* − *wsrf.pdf* >

[6] Ian Foster and Carl Kesselman : The GRID 2 , P56 , Morgan Kaufmann , 2004

[7] Globus Toolkit web site ,06.Aug.2005 < *http* : *//www.globus.org/toolkit/* >

[8] Von Welch, NCSA/U. of Illinois: Globus Firewall Requirements,8/Jul/05 < *http* : *//www* − *unix.globus.org/toolkit/security/firewalls/* >

[9] Ian Foster and Carl Kesselman : The GRID 2 , P667, Morgan Kaufmann , 2004

[10] Man in the middle attack, Wikipedia 29 July 2005 < *http* : *//en.wikipedia.org/wiki/Man_in_the_middle_attack* >

[11] Ian Foster and Carl Kesselman : The GRID 2 , P662 , Morgan Kaufmann , 2004

[12] Dictionary attack, Wikipedia ,20 July 2005.< *http : //en.wikipedia.org/wiki/Dictionary_attack* >

[13] Man page of PCAP , 06.Aug.2005< *http : //www.tcpdump.org/pcap3_man.html* >

[14] Ian Foster and Carl Kesselman : The GRID 2 , P665 , Morgan Kaufmann , 2004

[15] Distributed denial-of-service attacks, 20 August 2005 < *http : //en.wikipedia.org/wiki/Denial_of_service#Distributed_denial − of − service_attacks* >

[16] Message authentication code ,3 August 2005< *http : //en.wikipedia.org/wiki/Message_authentication_code* >

[17] Ian Foster and Carl Kesselman : The GRID 2 , P667 , Morgan Kaufmann , 2004

[18] HP technical documentation , chapter 1 , glossary of terms , 16.Aug.2005 < *http : //docs.hp.com/en/*32650 − 90887/*ch*01*.html* >

[19] UPPAAL Home 10/April/2005 < *http : //www.uppaal.com/* >

[20] Ian Foster and Carl Kesselman : The GRID 2 , P366 , Morgan Kaufmann , 2004

[21] UPPAAL Introduction 10/April/2005 < *http : //www.uppaal.com/* >

[22] Jay Unger , Matt Haynos ,IBM: A visual tour of Open Grid Services Architecture , 29 Jun 2005< *http : //www −* 128*.ibm.com/developerworks/grid/library/gr − visual/* >

[23] Ian Foster and Carl Kesselman : The GRID 2 , P367 , Morgan Kaufmann , 2004

[24] TWiki SpGlossary, 25 Aug 2005 < *http : //wiki.astrogrid.org/bin/view/SSVO/SpGlossary* >

[25] Martin Krzywinski, 25 Aug 2005 : PORTKNOCKING < *http : //www.portknocking.org/* >

[26] Vinay Bansal , Dept. of Computer Science , Duke University: Policy Based Firewall for GRID Security

[27] GRuediger Berlich, Ursula Epting, Jos Van Wezel, 2003: Grid Computing, Clusters and Security

[28] A.Menezes, P.van Oorschot, and S.Vanstone, CRC Press, 1996: Handbook of Applied Cryptography

[29] N. Haller, Bellcore, C.Metz, May 1996: RFC 1938 - A One-Time Password System

[30] N. Haller, Bellcore, C.Metz, P.Nesser, M.Straw, Feb.1998: RFC 1938 - A One-Time Password System

[31] Ian Foster, 17 Jan 2005:A Globus Toolkit Primer $< www.globus.org/toolkit/docs/4.0/key/GT4_Primer_0.6.pdf >$

[32] Borja Sotomayor ,2003: The Globus Toolkit 3 Programmer's Tutorial $< http : //www - unix.globus.org/toolkit/license.html] >$

[33] Gerd Behrmann, Alexandre David, and Kim G. Larsen, Department of Computer Science, Aalborg University, Denmark, 17th November 2004: A Tutorial on Uppaal