An Ownership-based Message Admission Control Mechanism for Curbing Spam

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Hongxing Geng

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

Unsolicited e-mail has brought much annoyance to users, thus, making e-mail less reliable as a communication tool. This has happened because current email architecture has key limitations. For instance, while it allows senders to send as many messages as they want, it does not provide adequate capability to recipients to prevent unrestricted access to their mailbox. This research develops a new approach to equip recipients with ability to control access to their mailbox.

This thesis builds an ownership-based approach to control mailbox usage employing the CyberOrgs model. CyberOrgs is a model that provides facilities to control resources in multi-agent systems. We consider a mailbox to be a precious resource of its owner. Any access to the resource requires its owner's permission. Thus, we give recipients a capability to manage their valuable resource - mailbox. In our approach, message senders obtain a permission to send messages through negotiation. In this negotiation, a sender makes a proposal and the intended recipient evaluates the proposal according to their own policies. A sender's desired outcome of a negotiation is a contract, which conducts the subsequent communication between the sender and the recipient. Contracts help senders and recipients construct a long-term relationship.

Besides allowing individuals to control their mailbox, we consider *groups*, which represent organizations in human society, in order to allow organizations to manage their resources including mailboxes, message sending allowances, and contracts.

A prototype based on our approach is implemented. In the prototype, policies are separated from the mechanisms. Examples of policies are presented and a public policy interface is exposed to allow programmers to develop custom policies. Experimental results demonstrate that the system performance is policy-dependent. In other words, as long as policies are carefully designed, communication involving negotiation has minimal overhead compared to communication in which senders deliver messages to recipients directly.

Acknowledgements

It is very difficult to find a finite set of people to express my gratitude to. In some ways, it turns out to be more difficult than writing any chapter of this thesis. I have received support and assistance from a number of people; however, I would like to use this opportunity to give special thanks to the following people.

It would not have been possible to complete this work without support of my supervisor, Professor Nadeem Jamali. Tons of thanks go to him. I have benefited a lot from his invaluable advices and insightful comments. Many times when I faced challenges which slowed my progress, it was he who helped me overcome them. In addition to providing me academic advices, he also offered me many suggestions on my personal life and my career development.

I would like to thank three professors in my committee. Professor Simone Ludwig, my internal committee member, deserves my deeped gratitude. Thanks for her willingness to be my committee member. She knew this work from earlier stage and attended presentations on it and gave me many suggestions. I would thank Professor Aryan Saadat Mehr for his willingness to be my external examiner despite a very busy schedule and for offering valuable advice. I would like to thank Professor Mark Keil, who chaired the committee. I appreciate everyone's hard work in helping me improve this thesis.

My colleague, Xinghui Zhao, gave me invaluable suggestions on the first draft of this thesis. It was not as readable as it is now. I appreciate her suffering in reading those drafts.

Many thanks go to office staff of the Department of Computer Science at University of Saskatchewan. They provide a very supportive working environment and selfishless assistance to every graduate student. I would particularly like to thank Jan Thompson, Heather Webb, and Maureen Desjardins.

I am deeply indebted to my family. My wife, Ran Ding, always supported and encouraged me. During my school year, she was committed a lot to our family. I have to thank my son, Eric Geng. His smile reminds me of my responsibilities which pushes me forward.

This thesis is dedicated to my family.

Contents

List of Tables

List of Figures

List of Abbreviations

AA	The Actor Architecture
ABM	Attention Bond Mechanism
APWG	Anti-Phishing Working Group
ASRG	Anti-Spam Research Group
CA	Central Authority
CAPTCHA	Completely Automatic Public Turing test to tell Computers and Humans Apart
CSV	Certified Server Validation
DMTP	Differentiate Mail Transport Protocol
EP	Explicit Policy system: system with negotiation using explicit polices
ESP	Email Service Provider
HIP	Human Interactive Proof
IM2000	Internet Mail 2000
IP	Implicit Policy system: system with negotiation using implicit polices
IRTF	Internet Research Task Force
ISP	Internet Service Provider
META	Message Enhancements for Transmission Authorization
MH	Message Handler
MQ	Message Queue
MTA	Mail Transfer Agent
SMTP	Simple Mail Transfer Protocol
SN	System with Negotiation
SPF	Sender Policy Framework
UI	User Interface

Chapter 1

Introduction

Email has brought many benefits to users; however unsolicited or unwanted bulk email creates an onerous burden for both legitimate users and Email Service Providers (ESPs), which include consumer Internet Service Providers (ISPs), (such as AOL and Telus), free ESPs (such as Gmail and Hotmail), and organizations (such as governments, universities, and companies). Rampant unsolicited bulk email affects every Internet user so that email has become an unreliable communication mechanism, and Internet users are losing their trust on email even the Internet [10]. Spam research is becoming an important research area lately and is attracting significant interest, including from researchers and developers.

1.1 Spam

There are various informal spam definitions. [60] defines spam as email which advertises products. In [46], spam has characteristics that a huge number of copies of the same message are sent to recipients who would not be willing to receive it. [52] defines spam as "unsolicited bulk email." "Bulk email" means a large volume of email including the same content. This definition considers a message as spam if and only if two conditions are satisfied: the message is unsolicited and it is bulk. "Unsolicited" email may not always be spam. For example, an email containing a customer's bill information from a credit card company is not spam. Similarly, "bulk" email may not always be spam either. For instance, bulk email sent from a mailing list to its subscribers is not spam. In this thesis, spam refers to any "unsolicited messages," which means that any message that recipients do not wish to receive is spam, no matter whether it is a bulk message or not.

1.2 Cost of Spam

Spam has brought and continues to bring significant cost to legitimate email users as well as ESPs. It is estimated that spam accounts for 85% email traffic today and this percentage will be up to 90% at the end of 2007 if the current increase trend continues [18]. Spam has cost business 50 billion dollars around the world in 2005 [31]. Not only does spam demand significant time and attention from legitimate individual users, but it also affects ESPs.

First of all, spam floods mailbox of legitimate users, which makes it hard for them to go through their email. It takes additional time and attention for legitimate users to find their wanted email, which reduces productivity. Even worse, legitimate email may be deleted accidentally. Second, one type of spam, called phishing email, intends to cheat email users. Phishing email [10] attempts to deceive recipients into disclosing their confidential information, such as private bank information, credit card number, and social insurance number. Afterwards, spammers can abuse this information to commit crime. Consequently, fraudulent email causes users to lose their trust in Internet transactions, which hurts e-businesses. Experts estimate that 50% of phishing email deceives Internet users by spoofing eBay and Paypal [49]. Figure 1.1(a) shows an example[1] of phishing email which imitates eBay. Figure 1.1(b) presents the fraudulent website, directed by the link in the phishing email, luring eBay users to disclose their password. Finally, spammers, along with hackers and virus writers, usually disperse spam by intruding computers which are not theirs[2]. Currently, more than 80% of spam worldwide is sent by remote-controlled zombie PCs [56].

Spam brings significant cost to ESPs as well. An ESP may lose its reputation because spammers find it is easier and cheaper to use its email services to send spam [21]. Secondly, ESPs are forced to invest greater resources, such as network bandwidth, anti-spam technologies, storage servers, and customer support, etc. in an effort to fight spam. For example, ESPs have to invest more effort on inhibiting incoming spam and dealing with customers' complaints. Last, but not the least, ESPs may suffer from being added to black lists [53] if spammers initiate a large numbers of spam through their email servers.

1.3 Spammer Incentives

In this section, we examine why spammers spam. [39] suggests that incentives have been the "cornerstone of human existence". For example, the fact that parking is more expensive in downtown of a city is not because the government of the city is cash-strapped, but because the city government intends to optimize usage of parking lots. The intent of spammers is not just to disturb email users but to make profit.

However, current email infrastructure allows sending email for free, which allows malicious spammers to advertise their products through email for free. The only cost that spammers have to pay is the use of a computer and a subscription to access the Internet. However, compared to a large volume of spam they send, the marginal cost can be negligible. It is estimated that spammers can make profit even if only one recipient out of every 100,000 purchases their advertised products [25].

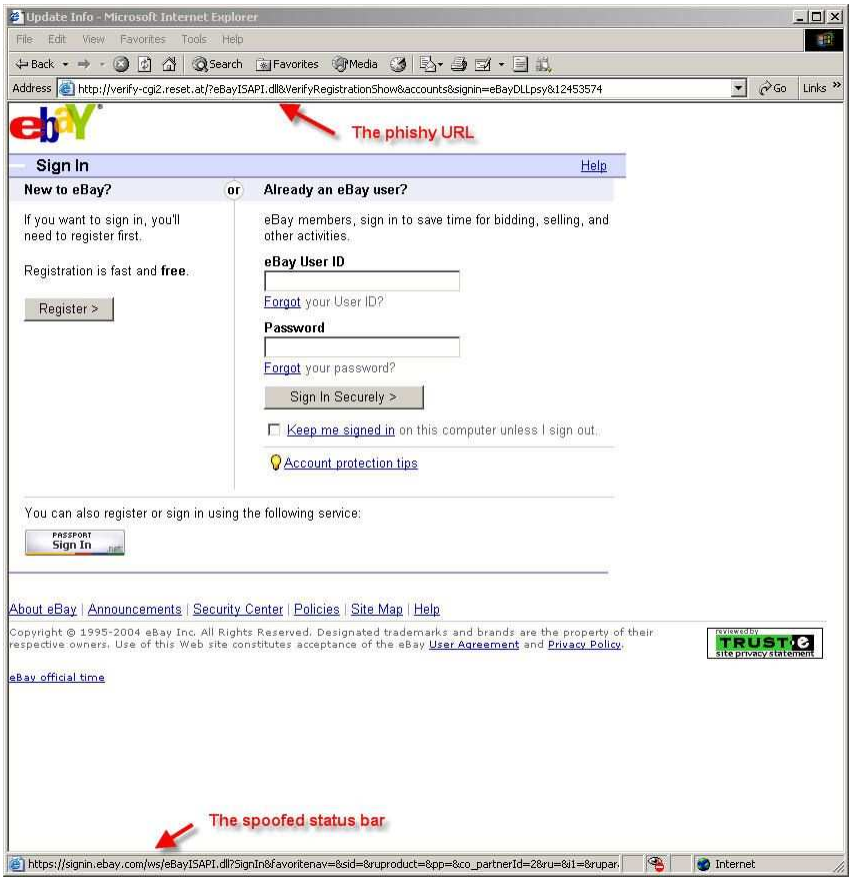
[1] The example is from the official website of Anti-Phishing Working Group [24]
[2] Those compromised computers are called zombie PCs.

Dear eBay User,
During our regular update and verification of the accounts,
we couldn't verify your current information.
Either your information has changed or it is incomplete.
If the account information is not updated to current information
within 5 days then, your access to bid or buy on eBay will be suspended.
go to the link below,
and re-enter your account information.

Click here to update your account.
Please Do Not Reply To This E-Mail As You Will Not Receive A Response
Thank you
Accounts Management

Copyright©1995-2005 eBay Inc.

(a) A phishing email



(b) The fraudulent website directed by the link in the phishing email lures legitimate users to disclose their personal information

Figure 1.1: A phishing email and the forgery website

1.4 Research Objectives

Current email infrastructure uses SMTP (Simple Mail Transfer Protocol) [32] to transfer email among users. At the time when SMTP was designed, Internet users trusted each other. Because SMTP assumes that Internet users are reliable, it has inherent shortcomings since it was born. First of all, SMTP does not impose constraints on senders. Not only can senders transmit as many messages as they want, but they can also send them for free. Moreover, senders do not need to stay online after they click the *send* button. Even worse, senders can modify the *Return-Path*, a header in the SMTP, which gives spammers an opportunity to cheat recipients. SMTP does not provide facilities to authenticate senders. On the one hand, current email infrastructure provides significant power to senders; on the other hand, it does not equip recipients with capabilities to control their mailbox access except that recipients can delete messages in their mailboxes. Recipients can only passively accept incoming email. Subsequently, senders can abuse recipients arbitrarily if they want. The primary goal of this research is to present an approach to solve the challenging spam problem. In our approach, we think of a mailbox as a type of precious resource. The owner of a mailbox can manage its mailbox consumption. We construct an ownership-based approach for controlling and sharing mailbox resources along with other message-related resources.

Our approach has the following characteristics:

Granting more mailbox access control capabilities to recipients: As we mentioned before, current email infrastructure is essentially sender-driven. Recipients cannot control their incoming messages. Approaches such as spam filters [54, 26, 27, 53] usually work at the recipient side by allowing recipients to block incoming spam. Our approach is to enable recipients to react to spam positively. This means that whether or not a message is transmitted to a recipient is determined by the recipient.

Controlling messages on both the sender-side and the recipient-side: Until now, most anti-spam research has focused on defending spam at the recipient side. For instance, spam filters [22, 23, 54] are usually installed at the recipient side. Signature-based approaches [62, 61, 44, 7] authenticate senders through allowing recipients to check additional headers inserted by senders. DiffMail [11] and TCP damping [40] work at the recipient side as well. In DiffMail, recipients differentiate senders into three categories. Recipients use different strategies to manage incoming messages from different groups. TCP damping allows recipients to slow down network traffic at the TCP level. In our approach, we aim to make sender accountable for message transmission. The underlying principle is that it is a sender's responsibility to deliver relevant messages because a sender knows exactly the content of its messages in advance, but recipients do not.

Presenting a flexible mechanism for spam control: One approach to control spam is to discourage spammers from spamming by making sending spam expensive. Market-based solutions, assigning

costs in terms of micropayment, human interaction, or computational cost [1, 13, 15, 34, 55], are based on this idea. These approaches require senders to invest some effort in sending messages. However, these solutions are not flexible. For example, Zmail [34] requires senders to pay a small amount of money called e-pennies when they deliver a message; however, besides setting fixed price for their incoming messages in terms of e-pennies, recipients cannot manage more. We want to equip recipients with more flexibility in controlling incoming messages.

Constructing a long-term relationship between senders and recipients: At present, no approach attempts to build a long-term relationship between senders and recipients. All proposed approaches only accommodate temporary relationships between senders and recipients. For each individual incoming message, they build a new relationship on the fly. After communication has completed, the relationship disappears. In our approach, we enable participants to build a long-term relationship in order to avoid creating a relationship for each individual message.

In this thesis, we present an ownership-based approach for spam control. We think of a mailbox as a type of precious resource, which others cannot consume freely. Anyone interested in consuming a mailbox resource must get consent from its owner. Senders can get consent by negotiating with the potential recipients. The product of a negotiation process is a *contract*, which is a written consent regulating the privileges and responsibilities of both the sender and the recipient. The generated contract controls the successive communication between the sender and the recipient. By employing contracts, we can combat spam not only on the recipient side but also on the sender side. Meanwhile, contracts also construct a long-term relationship between senders and recipients.

1.5 Contribution

The contributions of this work are as follows:

- An ownership-based approach for controlling access to mailbox resources
- Prototype implementation
- Separation of policy from mechanism
- Introduction of *group* ownership rights
- Case studies with example scenarios
- Analysis and experiments to assess feasibility

1.6 Outline

The rest of this thesis is organized as follows:

Related work is discussed in Chapter 2. Chapter 3 describes our approach - an ownership-based approach to message admission control. Prototype design and implementation are elaborated in Chapter 4. Chapter 5 presents analytical models and experimental results. Finally, conclusion and future work are discussed in Chapter 6.

Chapter 2

Related Work

Spam research has received significant attention from the research community recently. For example, anti-spam work groups have been founded, such as the Anti-Spam Research Group (ASRG) of the Internet Research Task Force (IRTF) and Anti-Phishing Working Group (APWG) [24]. Many ongoing anti-spam projects are carried out by industrial research labs. In addition to technological approaches, nations and communities of nations such as the United States, Canada, European Union, and Australia have enacted laws and legislation to deter spam [45]. In this thesis, we focus on technological approaches rather than legislative approaches.

There are a variety of anti-spam techniques. We classify them into five groups: filter-based approaches, signature-based approaches, market-based approaches, protocol-level approaches, and other approaches. Each class of approaches has its own advantages and disadvantages. Under specific circumstances, two or more approaches may work more effectively when they work together.

2.1 Filters

We group filters into two categories: non-statistical filters and statistical filters. Generally, non-statistical filters determine whether a message is spam based on their contact lists. On the other hand, statistical filters evaluate a message according to their knowledge, which is acquired through training.

2.1.1 Non-statistical Filters

In this section, we introduce four types of non-statistical filters, which are whitelists, blacklists, spamtraps, and greylisting. Generally, through maintaining a contact list, the first three approaches check the list to determine whether an incoming message from a particular sender should be delivered to the intended recipients. Greylisting works based on the common observation that spammers do not attempt to send messages twice to the same recipient if the first sending attempt fails.

Whitelists

A common type of non-statistical filter uses whitelists, (also known as safelists) which contain lists of contacts. Recipients regard messages from these contacts as desirable. Contacts in a whitelist can be individual email addresses, IP addresses, or domains. Whitelists can be used in two different modes: exclusive and non-exclusive. In the exclusive mode, only messages from those in the whitelist can be delivered to the inbox folder. Messages from others are blocked or deposited into the spam folder. In the non-exclusive case, messages from contacts in the whitelist are delivered to recipients directly without being checked by statistical spam filters. This may lead to false positives meaning that filters misclassify legitimate email as spam. Statistical filters have this inherent problem. We will discuss it shortly.

The main disadvantage of whitelists is that the cost of list maintenance can be very high. If the maintenance tasks are carried out by individual users, not all users are willing to update the list frequently because it takes time to do so. If the ESPs are responsible for updating whitelists, it is no doubt that maintenance time will increase. However, if a whitelist is not updated in a timely manner, it will not work effectively. Finally, the whitelists can be explored by spammers [19].

Blacklists

The opposite of whitelists is blacklists [53]. In contrast to whitelists, messages from contacts in a blacklist are blocked or deleted or sent to the spam folder. Like whitelists, contacts in a blacklist can be individual email addresses, IP addresses and entire domains. In practice, blacklists are not used independently. They are combined with statistical filters to avoid false negatives, which happen when spam is misclassified as legitimate email. Blacklists can help reduce percentage of false negatives in statistical filters.

The major problem of blacklists is that it is challenging to update the list instantaneously to reflect the latest information. Spammers always change their IP addresses and domains to send spam effectively and to avoid being tracked. Therefore, a large number of email addresses, IP addresses, and domain names can be sources of spam. The effort to maintain such a large volume of information is demanding. Moreover, a large amount of information demands very large storage space. Finally, innocent IP addresses and domains may be added into blacklists unexpectedly when spammers use compromised *zombie computers* to send messages, which may lead blacklists to become ineffective.

Spamtraps

A spamtrap is a *honeypot* [27, 51] used to collect spam. Usually, spamtraps are fake email addresses which are used exclusively for capturing spam rather than for general communication purpose. Spamtraps are stored in a location where legitimate email users cannot find them but email address

harvesters can. Normally, email address harvesters are programs which run automatically to search for email addresses in the Internet. Spamtraps are typically used to build blacklists: the collected email addresses are black listed.

The controversial aspect of spamtraps is that they are vulnerable to the *backscatter* email attack, which may cause legitimate email to be considered as spam. For example, a malicious spammer can send email claiming to be from a spamtrap to legitimate users, who will be considered as spammers by the spamtrap if they or their mail systems respond to the false email.

Greylisting

Based on the observation that spammers do not attempt to send messages again once the first attempt fails, but legitimate users do, greylisting was invented. MTAs (Mail Transfer Agents) which employ the greylisting technique temporarily reject any messages from unknown senders and then accept it at the second. Because legitimate senders try to send messages again and spammers usually do not retry, legitimate messages get through but spam does not. The information checked by greylisting-compatible mail servers is a triplet which consists of the IP address of the connecting host, the envelope sender address, and the envelope recipient address [26].

The main disadvantage of greylisting is that legitimate messages can be lost [38]. Greylisting assumes that the legitimate email servers will make the second attempt if the first fails; however, in practice, some email servers may not retry, causing some legitimate messages to be lost. Another assumption that greylisting is based on is that spammers' email servers will not make a second connection attempt. Thus, it is vulnerable to be attacked by spammers if they intend to do so. One minor disadvantage of greylisting is that it introduces unnecessary delay for legitimate email users.

2.1.2 Statistical Filters

Statistical spam filters [23, 22, 54] are content-based filters using machine-learning techniques, which have to be trained to work effectively. Because training and learning processes take time, the accuracy of recognizing spam increases over time. Commonly, a statistical filter maintains a collection of words (also known as "Bag-of-words") which are most likely to be used in spam. An incoming email containing words in the collection is treated as spam.

One of earliest statistical filters was proposed by Sahami *et. al.* in 1998. After Paul Graham published his classic article, "A plan for SPAM," [22] statistical filters became popular[1]. Researchers from Microsoft research employed more comprehensive machine learning systems to attack the spam problem [19, 20]. To make a machine learning system work effectively, first of all, a large volume of

[1]Most of them are Bayesian-based filters

training data, in which messages are identified as spam or regular email, is collected. The second step is to analyze characteristics of each message and to use these characteristics to train the system. The analysis not only examines what spam looks like but also determines the features of regular mail. The characteristics might include words in the message as well as the time when the messages are sent.

Unfortunately, statistical spam filters have some problems. First of all, they suffer from false positives and false negatives. False positives may not be acceptable by certain users, because the misclassified legitimate email may contain critical information. False negatives still bring annoyance to normal email users. To avoid these two problems, statistical filters often work with a whitelist and a blacklist. Second of all, spammers can use techniques to adapt new statistical filters rapidly, which leads to a filter that works one day but will likely not work the next. One example can be found in [20], which describes an ongoing escalation of technology taking place in the battle between spammers and anti-spam researchers and developers. For example, the deluge of image spam has recently increased significantly. However, current statistical filters do not address image spam because of the high cost of processing images to determine whether the email is spam. Image spam brings a new challenge to anti-spam researchers and developers. Figure 2.1 shows an example of image spam.

Figure 2.1: An example of image spam

2.2 Signature-based approaches

Email spoofing is one of the fraudulent email activities in which the envelope sender address (also known as *Return Path*) is forged to appear as though the email has a different source address. Email from spammers claiming to be from one legitimate contact may easily coax recipients to disclose their private information such as password, banking information, and credit card number. Email spoofing has become one of the biggest challenges in common internet activities and reduces email to a less reliable and trustable communication medium.

Authentication mechanism can be built to avoid email spoofing. However, SMTP does not provide any facilities to authenticate senders. Signature-based approaches attempt to deal with this problem. The basic idea behind signature-based approaches is that before transmitting a message, the sender signs the message in some way. At the other side, the recipient can effectively authenticate the message. For example, the recipient can authenticate the identity of the sender through querying the sending DNS server.

There are various signature-based proposals. In this section, we discuss Yahoo DomainKeys [62], Sender Policy Framework (SPF) [61, 50], Message Enhancements for Transmission Authorization (META) Signatures [37], and Certified Server Validation (CSV) [3].

2.2.1 DomainKeys

DomainKeys developed by Yahoo is an email authentication system for prohibiting email spoofing. On the sender server side, the domain owner creates a pair of keys: one is a public key and another is a private key. The private key is stored in the email server and is used to sign outgoing messages. The public key is published in the DNS server and is used by recipients to verify their incoming messages.

When a sender uses a DomainKeys-compatible outbound email server to send a message, the pre-stored private key will be used to generate a digital signature for that message. The digital signature will be attached to that message as a header entry. At the recipient server side, the email server extracts the digital signature and the claimed *From: Domain* header from the received email and retrieves the public key from the DNS server for the *From: Domain*. The recipient-side email server uses the public key to verify the digital signature. If the signature is approved, the recipient-side email server will deliver the email to the recipient. Otherwise, the email will be flagged, blocked, or deleted

Cisco's Identified Internet Mail (IIM) [7] works in the same way as Yahoo DomainKeys. And lately these two solutions have joined into Domain Keys Identified Mail (DKIM).

DomainKeys (or DKIM) has difficulties when email forwarders are involved in communication. Since email forwarders may considerably modify the message, the attached signature may become

invalid on the recipient-side. Hence, the recipient email server cannot effectively verify the signature. Another shortcoming of DomainKeys is that generating and verifying digital signatures lead to needless communication and computation overhead.

2.2.2 Sender Policy Framework

Sender Policy Framework (SPF) extends SMTP by allowing domain owners to publish SPF records on the domain's DNS server. The SPF records regulate whether a machine in that domain is warranted to send email or not. Once the recipient receives a message claiming from that domain, the recipient can query the DNS server of the claimed domain to check whether the message is actually coming from the claimed domain or not and whether the sending machine is permitted to deliver messages by that domain.

Having the same drawback of DomainKeys, SPF does not work well if mail forwarding is involved. In this case, even if the machine that the sender use is permitted to send email; however, if email is eventually transmitted by an email forwarder in that domain, email may fail to be delivered. Moreover, SPF is not applied to the case in which email users tend to change their ISP or computers quite often in that domain owners have to re-publish SPF records for them.

Sender ID[44] proposed by Microsoft is heavily based on SPF and employs the same mechanism to prevent email address forgery.

2.2.3 Message Enhancements for Transmission Authorization Signatures

Email may pass through multiple mail servers from senders to recipients. Message Enhancements for Transmission Authorization (META) Signatures require each mail server en route to add a cryptographic signature to the email message. Consequently, the subsequent mail server can authenticate the message by querying the previous mail server. This establishes a chain of trust until the message arrives at the destination.

Although META signatures solve the forwarding problem in DomainKeys and SPF, the requirement that all involved mail servers should be META-compatible makes it impossible for incremental deployment. If one mail server is not enforce META signatures specification, the trust chain would break and the META authentication would fail. This causes META to be a close system.

2.2.4 Certified Server Validation

Certified Server Validation (CSV) validates a SMTP session by querying the sending domain and a reputation service. The validation process consists of three steps. Firstly, the recipient's mail server examines whether the IP address of the sender's mail server matches the domain name's IP address. Secondly, the recipient's mail server determines whether the sender's mail server is

allowed to transmit email from that domain. Finally, the recipient's mail server queries a reputation service such as Spamhaus.org [52], which computes reputation scores for domains, to evaluate the reputation of the sender's domain. Whether the message is delivered to the recipient is based on the results of the above steps.

2.2.5 Signature-based Approach Summary

In summary, signature-based approaches only resolve one aspect of spam problems - the email spoofing problem, in which spammers deceive victims to trust them and disclose important personal information to the spammers. However, If spammers do not attempt to hide their identity, these approaches become ineffective.

2.3 Market-based approaches

Since it is so cheap to spread a deluge of spam, spammers can abuse recipients arbitrarily and thus debate benefits of recipients. To reverse this situation, various market-based approaches were proposed in order to make the marginal cost of communication more expensive. Pricing communication is not a novel idea. As early as 1995, MacKie-Mason and Varian argued that pricing can make the usage of the scarce Internet resource more efficient and improve public benefits [43]. Current situation is that while an amount of information increases significantly, the attention supplying is almost fixed. The market mechanism can make demand and supply more balanced. The basic idea behind market-based solutions is that senders must demonstrate that they have committed some efforts to sending email. Commitment can be micropayment, computational cost, Human Interactive Proofs (HIPs) [55], or Attention Bond Mechanism (ABM) [42] (also called sender at risk). For legitimate users, the cost that has to pay is very limited, but for spammers whose economy pattern is based on their abilities to disperse a deluge of spam, the cost may become extremely high.

2.3.1 Micropayment

One of earliest micropayment approaches, E-stamps [57], was proposed by Brad Templeton in 1995. E-stamps are issued by a digital money bank, and each e-stamp has an expiration date. If a sender intends to deliver a message, an e-stamp is attached to the message before sending. The recipient only accepts messages with e-stamps and rejects messages without one. After accepting a message, the recipient can choose to redeem the attached e-stamp from the bank if the message is an unsolicited junk email, or to let it expire if the message is from a legitimate user. In the e-stamp utopia, normal email users do not redeem e-stamps from the bank except when they receive spam.

The immoral spammers cannot afford the massive burden of email users consistently redeeming their e-stamps, they might stop sending spam.

From an economic perspective, Kraut *et al.* [33] proposed that a variable pricing scheme, combined with targeting information[2], rather than a flat pricing rate will improve the benefits to both senders and recipients.

The main obstacle to deploying a micropayment scheme is that it is impossible to construct a micropayment infrastructure because the very small amounts of money make transactions non-profitable. Another barrier is that this scheme is too *impolite* to be practical. For example, an email from an unknown person may aim to assist the recipient with a query, which the intended recipient has posted in a forum. If the recipient asks for e-stamps, the stranger may decline to offer the assistance.

2.3.2 Computational Challenge

There is an alternative to real-world money called proof-of-work [13], which was first proposed by Dwork and Naor in 1992. The basic idea is that by computing a complex function, senders present evidence that they have put computational effort for sending the email. It is not a heavy burden for normal email users because the computational cost is so small that it can be ignored. However, the processing time is a limited resource so that spammers cannot afford the computational cost for massive volume of email, thus prohibiting spammers from distributing spam.

One of the most popular proof-of-work systems is Hashcash [4]. Hashcash requires senders to calculate a cryptographic function, which is expensive for senders to compute but is relatively cheap for recipients to verify. To prevent from double-spending hashcash and pre-computed hashcash, the hashcash function involves two important elements: the destination email address and the timestamp.

Analysis by Laurie *et al.* [35] concludes that proof-of-work does not work from the economic and security perspectives, respectively. However, Liu *et al.* [41] argue that proof-of-work does work if combined with reputation mechanisms.

2.3.3 Human Interactive Proofs

Human Interaction Proofs (HIPs) (also known as "Completely Automatic Public Turing test to tell Computers and Humans Apart"(CAPTCHAs) [59], or Reverse Turing Test) attempt to prevent the situation where email is sent automatically by machines. PC World estimates that around 50-80 percent of spam is sent from *zombie PCs*, which are unprotected personal computers compromised by spammers and used to send spam. Currently, the volume of spam dispersed by zombie PCs is

[2]Senders know information about recipients in advance

still increasing because of out-of-control *botnets*[3]. To ensure email is originated by human beings instead of machines, HIPs are used to tell human beings and computers apart.

Figure 2.2: Gimpy from the CAPTCHA project

A common type of HIPs is a distorted image of a sequence of letters and digits. Figure 2.2 shows a CAPTCHA example[4] from the CAPTCHA project official website. As we see, the obscured characters are easy for humans to recognize, but it is hard or impossible for computers to recognize them automatically. Another type of CAPTCHA is called *pix*, which presents a series of images and asks human beings to determine the difference or similarity among them. An example of pix is given in Figure 2.3. However, character-based or image-based HIPs may bring barriers to visually impaired users to use email services normally. To solve this problem, audio-based HIPs are proposed [58]. The audio HIP puzzles are designed to be easy for human beings to solve, but too hard to be decoded by computers. A typical audio HIP puzzle is a spoken list of letters or numbers with intended noise, reverberation, etc.

The shortcoming of HIP is that it is vulnerable to a relay attack in which spammers may employ cheap human solvers to solve HIPs [9]. A recent study [6] reveals that computers may even beat human beings at single character recognitions in HIPs.

2.3.4 Attention Bond Mechanisms

Attention Bond Mechanisms [42] (ABM, also called Sender at risk) requires the sender to place a bond[5] on a third party. The recipient can claim the posted bond or request the third party

[3]botnets are software applications which run automatically over the Internet
[4]Gimpy is one type of CAPTCHA in the CAPTCHA project.
[5]Notice that a bond is not a warranty.

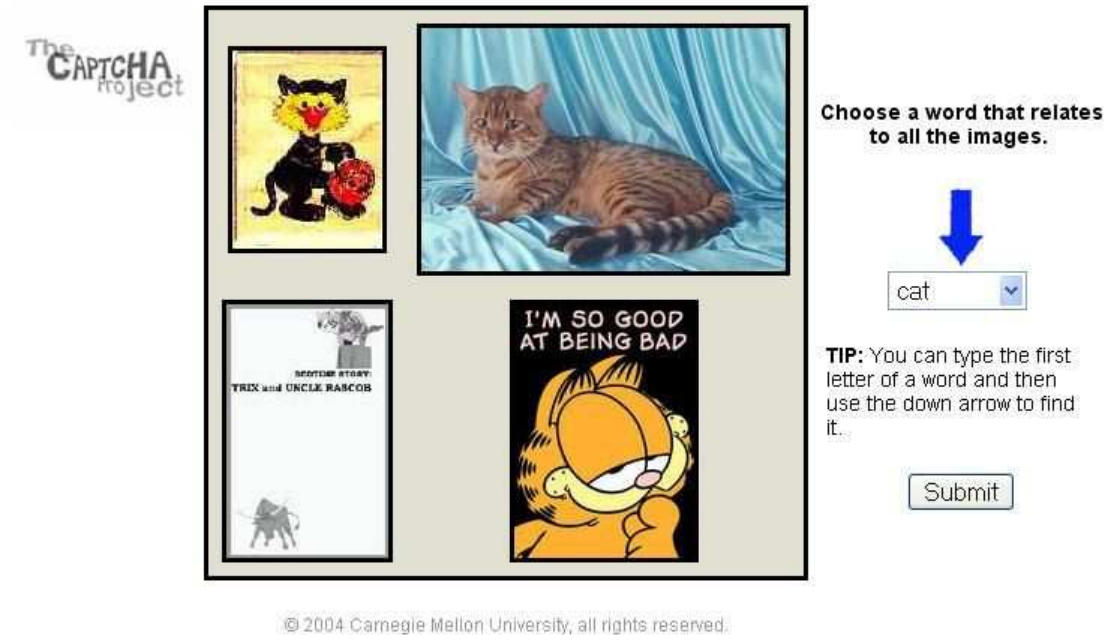


Figure 2.3: Pix from the CAPTCHA project

to release it. The purpose of ABM is to make communication valuable and improve the mutual benefits of both senders and recipients.

ABM works as follows. If the sender intends to send a message, the sender must post a bond on a third party. After receiving the message, the recipient examines whether the received message is desirable. If yes, the recipient will ask the third party to manumit the bond; otherwise, the recipient will claim the posted bond as compensation for its attention or time.

Like the micropayment solution, ABM may be considered impolite if legitimate users intend to send email to recipients, who use ABM-enhanced mail server. For example, for companies who leave their email addresses on their websites, ABM may deter potential customers to contact the companies, thus causing them to lose businesses.

2.4 Protocol-level approaches

In this section, we introduce three protocol level approaches. DiffMail attempts to modify current SMTP protocol. Internet Mail 2000 (IM2000) tries to replace SMTP protocol completely. Tarpit works at SMTP level or even lower at TCP level.

2.4.1 DiffMail

DiffMail [11] uses Differentiate Mail Transport Protocol (DMTP) [12] to transport messages. DMTP provides facilities to enable recipients to control their incoming messages. DiffMail categorizes senders into three classes: well-known spammers, regular contacts, and unclassified sources. Recipients handle each group differently. Messages from the regular contact class are processed in the same way as in the current email architecture. Messages from the well-known spammers are rejected. When a message is from unclassified sources, contrast to current email systems which deliver messages to the recipient-side servers, a DMTP-compatible mail server keeps the complete message on itself, and tries to send the original message which is encapsulated by an *envelope* (a type of short message) to the recipient-side mail server. The recipient-side server checks if the sender comes from unclassified sources. If yes, the recipient-side server only accepts the envelope message and delivers it to the recipient. At this point, the complete original message is kept in the sender-side mail server. If the recipient decides to accept the original message, it can retrieve the message from the sender-side mail server at its convenient time using facilities provided by DMTP.

DMTP attempts to grant extra capabilities to recipients and modify current SMTP. However, it still depends on whitelists and blacklists or similar services to work productively.

2.4.2 Internet Mail 2000

Internet Mail 2000 (IM2000) [5] is a project which intends to design a new email infrastructure and replaces the current protocol - SMTP completely. The core concept is that mail storage is the sender's responsibility rather than recipients'. A complete message from the sender is stored on the sender's ESP and occupies the sender's disk quota. The recipient is informed by a brief notification from the sender's ESP. The sender's ESP must always stay online in order that the recipient can retrieve the message at her convenient time. It is determined by the recipient if the message should be retrieved, which prohibits spammers from vanishing after sending spam. After downloading the message, the recipient can deliver an acknowledgement message to the sender's ESP in order to notify it to delete the downloaded message. Otherwise, the sender's ESP will retransmit the brief notification periodically.

Unfortunately, it is not reasonable to completely replace current SMTP protocol in one night. Hence, to deploy IM2000 becomes an incredible mission.

2.4.3 Tarpits

Tarpits assume that spammers are impatient email users. The typical method used by a tarpit is that the recipient-side mail server deliberately slows down the response to connection requests from the sender-side mail server [40]. To reach this goal, the recipient-side mail server can reduce the

advertisement windows, fake congestion, or intentionally wait for a while and then respond to the request. Commonly, spammers will give up the sending attempt if they find it is hard to construct a connection with the recipient-side mail server. This kind of tarpit works at the TCP level.

Another type of tarpit is SMTP tarpit [14, 17], which intentionally postphones sending SMTP greeting banner to the client. In SMTP, the client has to wait for receiving the greeting banner before it sends any data to the server. However, spammers are too impatient to wait for the greeting banner. Thus, the server can promptly identify the connecting request coming from spammers and decline the request.

Like greylisting, in practice, some email servers cannot process tarpits correctly causing legitimate email to be lost.

2.5 Other approaches

In this section, we present three anti-spam approaches: disposable email addresses, Spamalot, and an approach of blocking outgoing spam.

2.5.1 Disposable Email Addresses

Gabber *et al.* [16] proposed solving spam problems using *extended email addresses*, which are generated by appending extensions to original (or *core*) email addresses. Extended email addresses are disposable. Each user has multiple extended email addresses but only one core email address, which is not used for communication purpose but only for generating extended email addresses. For example, alice@example.com is a core email address, which is not used for communication. alice+a1b2c3@example.com is an extended email address, which is created through adding a1b2c3 to the core email address. Any one who wants to send email to the intended recipient, they must have an extended email address of the recipient. The sender requests one extended email address through a mechanism called *handshake*. Once the sender obtains an extended email address, the sender can use it to transmit email to the recipient. However, if a user realizes that one of her extended email addresses is compromised, she can discard the victim address right away. Therefore, the contact using the discarded email address cannot transmit email to her. However, normal communication with other regular contacts will not be affected. The contact which uses the compromised one must acquire a new one through another handshake process. Currently, ESPs such as Spamgourmet and Gmail have used disposable email addresses to solve spam problems.

The main drawback of disposable email addresses is that it is hard for some users to remember the extensions. Another shortcoming is that it is not easy for system administrator to manage email accounts.

2.5.2 Spamalot

In [8], Cranor *et al.* advocate for combatting spam by counterattacking them. In 2006, researchers from University of Illinois at Chicago built an anti-spam toolkit named Spamalot system [47] to exhaust spammers' resources to cause fruitless communication for spammers. Spamalot system uses intelligent agents, currently including three agents: *Arthur*, *Patsy*, and *Lancelot*, to answer back spammers to achieve the goal of consuming spammers' resources. By aggressively responding spammers, the authors argue that spam can be greatly reduced.

However, this approach assumes that it is human being not machines to process the feedback. If spammers have software which can interact with the Spamalot automatically, the spamalot will fail.

2.5.3 Sender-side Approaches

Almost all previous works focus on curbing spam at the recipient side. Nevertheless, spammers have many incentives to disperse spam through ESPs [21]. Increasing costs stimulate ESPs to thwart outgoing spam. One approach [63] proposed by Zhong *et al.* describes that by installing a spam filter at the ESP server, each outgoing message is scored a spam-likelihood value. According to the assigned score, the ESP server select a corresponding computational task for the sender. The message is not sent until the computational result has been presented by the sender. In another approach [21], the ESP servers can use HIPs, computational challenges, and micropayment to deter spammers to spread spam through them. This approach has the same problems as the recipient-side approaches when they use the same techniques.

2.6 Chapter Summary

This chapter reviews related anti-spam works including filter-based approaches, signature-based approaches, market-based approaches, protocol-level approaches, and three other approaches. Filter-based approaches separate and block spam. Signature-based approaches provide email authentication mechanism for current email infrastructure. The idea behind market-based approaches is to make the marginal cost of sending email more expensive; thus, spammers are not willing to distribute spam aimlessly. Protocol-level approaches attempt to modify or completely replace SMTP. A disposable email address can be discarded if the owner find it was compromised. Spamalot argues that email users should aggressively respond to spammers to exhaust their resources. Stopping outgoing spam by ESPs can undoubtedly reduce a large volume of outgoing spam.

Chapter 3

Ownership-based Message Admission Control

We create an approach by specializing CyberOrgs [29], which is a model for resource bounded distributed systems. Each cyberorg in the CyberOrgs model owns a certain amount of resources and can control how resources are to be consumed. In this thesis, we control a mailbox as a resource of its owner. Thus, the CyberOrgs model is desirable to be used to build our approach.

The organization of this chapter is as follows: In Section 3.1, we present the CyberOrgs model briefly. Section 3.2 describes our approach - ownership-based message admission control mechanism for curbing spam. We address negotiation ingredients and products in Section 3.3. Section 3.4 elaborates four communication scenarios. In Section 3.5, we introduce a self-protective mechanism to prevent an undesirable situation, in which malicious senders may produce too many error messages. Finally, the last section summarizes our approach from a different perspective.

3.1 CyberOrgs Model

CyberOrgs [29] is a model for resource control in multi-agent systems. In the CyberOrgs model, a cyberorg is a resource boundary which encapsulates a certain amount of resources and a collection of concurrent computations. Computations inside one cyberorg can only consume resources bound to the same cyberorg. Cyberorgs organize computations and resources as a tree hierarchy. Except the root cyberorg, each cyberorg is accommodated in another cyberorg, which is the parent cyberorg of the former one. Before being encapsulated, the intending cyberorg must negotiate a *contract* with a potential parent cyberorg. The contract stipulates the *types* and *quantities* of resources, which will be available to the encapsulated cyberorg, and their costs. To satisfy its needs, the encapsulated cyberorg can purchase resources from its parent cyberorg according to the contract between them. The currency used by cyberorgs is called *eCash*.

CyberOrgs provides three primitives to facilitate resource control, such as `isolate`, `assimilate`, and `migrate`.

- `isolate`

As shown in Figure 3.1(a), using `isolate` primitive, one cyberorg can spawn another cyberorg, which is encapsulated by the original cyberorg. The new cyberorg consists of actors, eCash, messages, and certain amount of resources, which are from the creating cyberorg. Moreover, a new contract is generated to regulate the trade of resources between them.

- `assimilate`

Figure 3.1(b) illustrates the `assimilate` primitive. Obviously, an assimilation process is the opposite of an isolation process. As shown in the figure, the exterior cyberorg dissolves the inner cyberorg; at the same time, the outside one acquires actors, eCash, messages, and resources which were possessed by the inside one. We should notice that the contract between two cyberorgs does not exists any more.

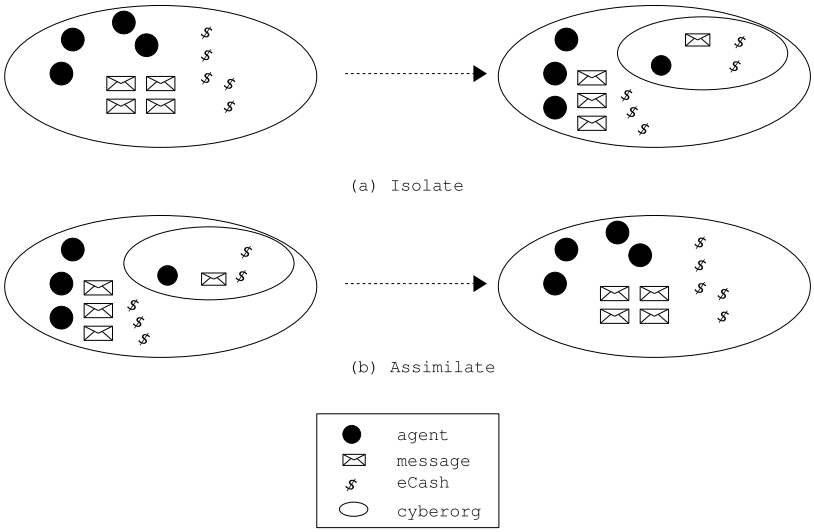


Figure 3.1: Cyberorg Primitives: isolation and assimilation

- `migrate`

Depicted by Figure 3.2, one cyberorg can migrate into another cyberorg. This happens when a cyberorg comprehends that its resource requirement cannot be satisfied by current contract with its parent cyberorg. Thus, it intends to negotiate a new contract with others to satisfy its needs. To reach its goal, the intended cyberorg must carry out a series of tasks. First of all, the cyberorg has to find one potential cyberorg which can provide sufficient resources. Second of all, the intended one must achieve a contract with the potential one through a negotiation process. Finally, once a desired contract is generated, the intended cyberorg migrates to the destination cyberorg, which provides a certain amount of resources to the new encapsulated cyberorg and gets paid according to the contract.

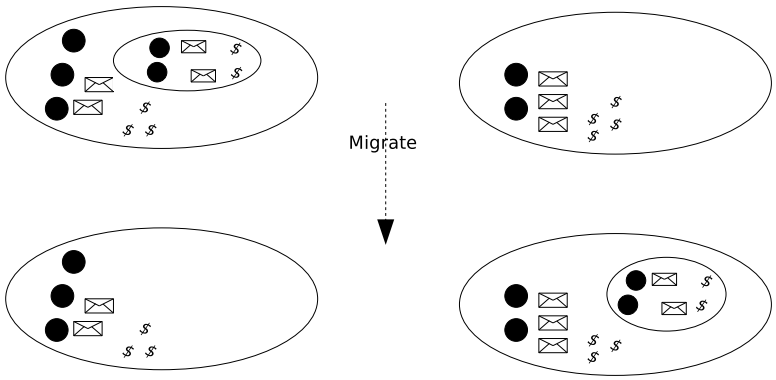


Figure 3.2: Cyberorg primitives: migration

In the CyberOrgs model, resources can be controlled using the above primitives. Resources flow from the root cyberorg to the leaves in the cyberorg hierarchy, and the eCash flows in the opposite direction. A cyberorg can acquire resources from any cyberorg as long as there is a contract between them.

3.2 Our Approach

The ownership-based message admission control mechanism is constructed on the CyberOrgs model. In our approach, we consider a mailbox to be a precious resource of its owner. Not only does a mailbox abstract physical entities, such as hard drive storage, CPU cycles, and network bandwidth, but it also represents time and attention of its owner. Mailbox owners have capabilities to control their mailbox access. Anyone who wants to send messages to a mailbox has to get permission from its owner. On the one hand, a sender can consume a mailbox by delivering messages to the recipient. On the other hand, to prevent from being accessed freely, the recipient manages its mailbox by controlling an access admission. To send a message, a sender has to get access right from the recipient. After holding a permission, it can send messages to the recipient.

3.2.1 Overview

In this section, we describe our approach briefly. The reminder of Section 3.2 illustrates our approach in detail.

In our approach, we have two types of communicating entities called cells and groups. Communication among communicating entities is through sending and receiving messages. A communicating entity can join a group; the later is called the parent group of the former, and the former is called the group member of its parent group. A cell can have multiple parent groups; however, a group can only belong to one group. Group members can consume resources of their parent group.

A sender must have "write" permission of a recipient's mailbox before it sends messages to the recipient. A sender who attempts to get "write" permission initiates a negotiation process with the recipient. For the sender, the desired outcome of the negotiation process is a *contract*, which represents the "write" permission and stipulates the subsequent communication between the sender and the recipient. Whether a contract is reached is based on both the sender's and the recipient's policies. A typical negotiation process should carry out the following three steps:

- According to its policy, the sender makes a proposal, which may include payment or favor which the sender would like to do for the recipient.
- The recipient evaluates the proposal based on its policy.
- If the proposal is accepted, a contract will be generated. The sender can send messages to the recipient according to the contract. Otherwise, the recipient declines the proposal by responding a *refusal message*, which is an unanticipated product of the negotiation. If the sender still intends to send messages, it should make a better proposal and start a new negotiation process.

The rest of this section is organized as follows: Section 3.2.2 introduces communicating entities. The primitives are discussed in Section 3.2.3. Section 3.2.4 discusses currency used in our approach. The detailed negotiation process is described in Section 3.3.

3.2.2 Communicating Entities

In this section, we introduce communicating entities. There are two kinds of communicating entities, which are *cells* and *groups*. We use cyberorgs to represent communicating entities. Thus, each communicating entity has its own resources and computations. The communicating entities, which are sources and targets of messages, can interact with each other through sending and receiving messages.

As we mentioned before, a cell can join multiple groups. It is incredible to imagine that a cell is splitted into several parts and each part is a member of a group. Hence, in our approach, when a cell joins a group, rather than the cell itself enters into the group, it creates a proxy, which enters into the group and represents the creating cell. The proxy is called a *satellite*. In this section, we describe cells, groups, and satellites.

Cells

One basic communicating entity is a *cell* which can abstract human users or mail user agents, which are used to send and receive messages. A *cell* is a cyberorg which encapsulates one mailbox and one user interface agent, which provides an interface for sending or receiving messages or for

management purposes (such as deleting messages and creating new message folders) to human users. Figure 3.3 depicts a cell. The black dot inside the cell is the user interface agent. Notice that there is a mailbox in the cell.

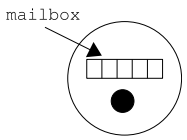


Figure 3.3: Communicating entity: a cell

A mailbox is a resource which is controlled by the cell which encapsulates it. The encapsulated *agent* has "read", "write", and "delete" access right to the associated mailbox. Any incoming messages to the agent must be stored in the affiliated mailbox. Originators of messages who intend to deposit messages to a mailbox have to obtain "write" permission from the cell which encapsulates the mailbox. As long as a sender has a *contract* with a recipient, outgoing messages to the recipient will go to the mailbox of that recipient directly.

A cell can play various roles when it is under different circumstances.

- Sender: a cell which initiates a negotiation process for intending to send messages
- Recipient: a cell which examines proposals for receiving messages
- Subscriber: a cell which commences a negotiation process for intending to receive messages.
- Publisher: a cell which evaluates proposals for dispersing messages

Besides from the mailbox resource, a cell may have other resources, such as policies and contracts. Policies are used for the negotiation purpose. Contracts stipulate resource usages between two sides. We will discuss these two resources shortly.

A cell has a unique name but may have multiple *aliases* and can use its name or its aliases to communicate with others. In fact, an alias is the name of one of the cell's satellite in our approach.

Groups

Another basic communicating entity is a *group* which corresponds to an organization in human society. A *group* is a cyberorg which contains one mailbox, one outgoing message queue (MQ), one user interface agent and may encapsulate other communicating entities. The user interface agent is for communication and administration purposes. Each group member has a contract with its parent group, which determines resource usages, such as access permission to the mailbox and MQ of the parent group. Basically, in a group, members can read messages from the mailbox and the MQ and can write messages into the MQ; whereas the user interface agent can read, write,

and delete messages from the mailbox and the MQ. In a group, different group members may have different access rights to the mailbox and the MQ according to their own contracts with the parent group. For example, one member is able to read messages from the mailbox but does not have privileges to write to the MQ; however, another can deposit messages into the MQ but cannot read the mailbox. The access rights are regulated by the contract between a group member and the group.

If a group itself wants to send a message to a recipient, the message will be stored into the mailbox of the recipient provided that there is an existing contract between them. However, if a group member wants to send a message to a recipient through its parent group, the message will be stored in the MQ of its parent group. Whether the message is sent or not is determined by the parent group.

A typical group is illustrated in Figure 3.4. The triangle is a satellite, which will be discussed shortly.

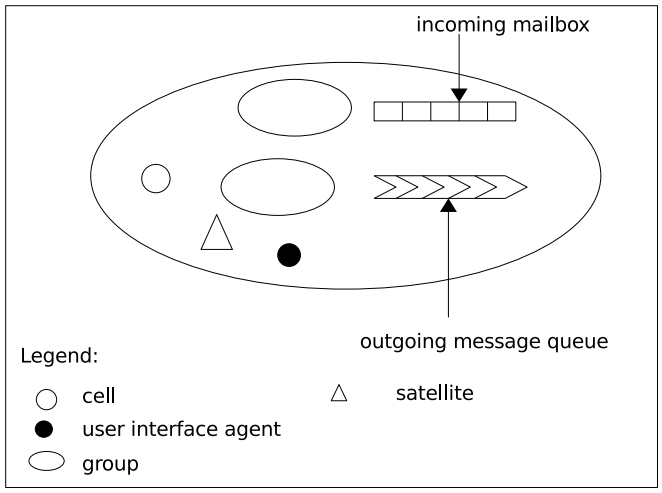


Figure 3.4: Communicating entity: a group

In addition to a mailbox and an outgoing message queue, a group may possess policies and contracts, which are resources that can be used by itself and its group members.

Like a cell, as a communicating entity, a group can be a sender, a recipient, a subscriber, or a publisher. Unlike a cell, a group cannot join multiple groups. In other words, a group can only have one parent group (if any).

Satellites

As we suggested before, when a cell attempts to join a group, it creates a proxy, which is a duplicate of the creating cell. The proxy is called a *satellite*, which represents the creating cell in the hosting group. The name of the satellite is an alias of the creating cell. Because satellites are copies of

their creating cell, satellites are cyberorgs in our approach. Before joining a group, the satellite must reach a contract with the potential hosting group. Because a cell can join multiple groups, it has corresponding number of satellites.

A satellite should hold the following properties:

- possesses a unique name
- must exist in a group
- represents the creating cell in the hosting group
- can communicate with the creating cell

However, a satellite cannot create another satellite.

3.2.3 Primitives

As we mentioned before, one cell may determine to join another group. By contrast, a cell can choose to depart one of its parent groups if any. Similarly, a group may join or depart its parent group if it has one. We discuss these primitives in this section.

Cells and Groups

A cell can join a group meaning that the cell creates a satellite, which migrates to the destination group and represents the creating cell in the hosting group. We call the hosting group the parent group of the creating cell. The creating cell and its satellites can communicate with each other by sending and receiving messages. The join process can be described as follows (Figure 3.5):

- The intending cell negotiates a contract with a potential parent group.
- The agent in the intending cell duplicates itself .
- The intending cell creates a satellite which encapsulates the created agent.
- The created satellite migrates to the potential group.

Because a satellite is a cyberorg, it holds a contract with its parent group.

When a cell decides to depart one of its parent groups, it sends a message to the corresponding satellite. Upon receiving the message, the satellite migrates out of its parent group.

Relationship between Groups

Unlike cells, a group can *join* another group through migrating to it directly, provided that the former group has reached a contract with the target group. On the other side, when its parent group cannot guarantee its resource satisfaction, an encapsulated member will *depart* its parent group using the `migrate` primitive of the CyberOrgs model.

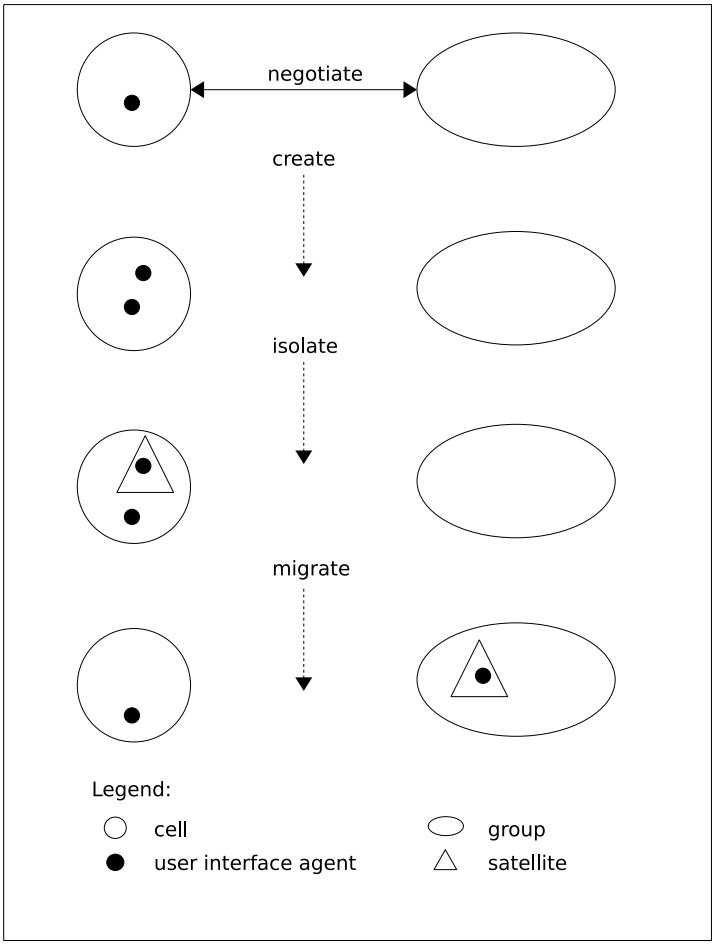


Figure 3.5: Primitive: a cell joins a group

Example

Let us suppose Alice, our exemplary user, has an email client on her machine, which can be used to send and receive email. Meanwhile, she is an employee of an organization called A and she has a free email account on an ESP named B. Under this circumstance, Alice can communicate with others through her email accounts of both A and B. Afterwards, she quits her job in A and get another job in the organization C. At this point, Alice cannot use her email account in A but can start interaction with the new email account in the organization C; however, the free email account is not affected.

3.2.4 Currency

As we mentioned in Section 3, when a sender attempts to deliver a message to a recipient, the sender must negotiate with the recipient. In the negotiation process, the sender presents a proposal

to compensate the recipient for consuming its mailbox. One ingredient of the proposal is a certain amount of money which the sender would like to pay. In our approach, like the CyberOrgs model, the currency used by cells and groups is called *eCash*. The minimum unit of *eCash* is called e-penny, which is a type of "token money". In other words, e-penny is a type of proof-of-work [41], which assures recipients that messages are worth reading in that senders have spent certain amount of effort on transmitting messages. Efforts may be computational cost, human effort, or real money. After the recipient accepts the offer, the payment can go to the recipient or a central authority in a real system.

3.3 Negotiation

In this section, we present the negotiation process in detail. In our daily life, for example, when an individual wants to buy a car, she has an acceptable maximum price in her mind. She would purchase a car if the real price is below the acceptable maximum price. This is her buying policy. For a particular car, car dealers have a desirable minimum price in their mind. They would sell the car if buyers are likely to accept a price higher than the minimum price. This is car dealers' sale policy. Customers and car dealers reach an agreement with a win-win price.

Likewise, in our approach, senders and recipients reach a contract based on their policies. Before a negotiation, a sender and a recipient must have policies. On the one hand, a sender generates a proposal according to its own policies. On the other hand, a recipient evaluates a proposal based on its own policies. We describe policies in Section 3.3.1. Section 3.3.2 elaborates how to generate a proposal based on a given policy. The products of a negotiation process is discussed in Section 3.3.3.

3.3.1 Policies

For negotiation purposes, there are two types of policies which are *sending policies* and *receiving policies*. Sending policies are used by senders to make proposals. On the other side, receiving policies are used by recipients to evaluate proposals. Thus, to send messages, a sender must possess sending policies. Otherwise, it cannot interact with others. If a recipient does not want to evaluate proposals, it may not have receiving policies, which means that the recipient is likely to permit all incoming messages to go through. Each communicating entity maintains a policy pool, which consists of its sending policies and receiving policies. One selects a policy from its policy pool to meet its needs when it participates in a negotiating process.

Policies can be arbitrarily complex. For example, a simple policy can be: one e-penny for one message, while a more complex policy can look like: ten e-pennies for ten messages within two days, which means that sending or receiving ten messages must occur within two days and the price of

ten messages is ten e-pennies.

A sender or a recipient may maintain *particular policies* for a special counterpart. For example, a sender may configure a particular policy for Bob: three e-pennies for one message when the recipient is Bob. Similarly, a recipient may hold a particular policy like: three e-pennies for receiving one message if the sender is Alice. On the one hand, when delivering a message, if the sender owns a particular policy for the recipient, the sender will make a proposal based on this particular policy. On the other hand, the recipient will use a particular policy to assess the proposal from the particular sender, if it has one.

As we suggested before, a communicating entity can be a member of a group. In this case, the communicating entity can use policies of its parent group. It can use these policies to make a proposal as a sender or evaluate a proposal as a recipient. However, because policies are also resources, whether the group's policies can be used by a member is determined by the contract between them. The contract is generated through a negotiation process at the time when that member intends to join the group.

3.3.2 Proposals

Proposals are built by senders based on their sending policies. After sending policies are settled, a sender is regulated by its sending policies when making a proposal. Basically, the sender is not likely to make better proposals compared to its policies. For example, if a sender holds a sending policy like: I would like to pay two e-pennies for sending one message, it is not reasonable for the sender to make a better offer such that three e-pennies for sending one message, but a worse offer such that one e-penny for one message is possible.

Corresponding to particular policies, a sender makes a particular proposal when it has a particular policy for a special recipient. When the sender wants to make a proposal to a recipient, it examines whether there is a particular policy for the recipient. If yes, the sender will select the particular one to make a proposal. Otherwise, the sender will make a proposal based on its general policies.

3.3.3 Negotiation Products

In a negotiation process, whether or not a recipient accepts a sender's proposal, it should deliver a response message to the sender. According to the response, two different types of products are generated. If the recipient accepts the sender's proposal based on its receiving policies, the desired product, a contract, is generated, which stipulates communications between participants. However, a contract might become invalid after it accomplishes its mission. For instance, one contract has a requirement that the recipient must accept ten messages from the sender. After the recipient accepts the tenth messages from the sender, the contract will not work any more. If the sender attempts

to deliver more, it has to re-negotiate a contract with the target. However, when a communicating entity is a member of a group, it can employ contracts of its parent group to transmit messages as long as the group consents to do so. We will discuss this point in Section 3.4.

Obviously, a proposal may be declined by the recipient, we call it an unacceptable proposal. A declining message acknowledged by the recipient is called a *refusal message*, which is not desirable and not relevant to normal communication. The refusal messages are another product of a negotiation process.

3.4 Communication Scenarios

In this section, we present four communication scenarios as follows:

1. Neither the sender nor the recipient belongs to a group.
2. The sender does but the recipient does not belong to a group.
3. The sender does not but the recipient does belong to a group.
4. Both the sender and the recipient belong to a group.

When the sender belongs to a group, it can send two types of messages through its parent group: *personal messages* and *group messages*. Personal messages represent the sender itself; however, group messages represent opinions of its parent group. A group message needs to be evaluated by the group and other group members to determine whether to be sent or not. Furthermore, when sending personal messages, group members use contracts of their parent group; To regulate its members' behavior, a group sets a *message quota* for each member. A message quota specifies that the maximum number of personal messages of a member can be sent through its parent group. If the number of personal messages of one sender has sent rises above its message quota, it cannot send more except that it requests more message quota. Message quota can be used by ESPs to stop outgoing spam [21]. For example, an ESP can set message quota for its email users. Thus, spammers cannot spread spam through this ESP arbitarily. Hotmail has already this type of limitation for its users in 2003 [48].

3.4.1 Neither The Sender Nor The Recipient Belongs to A Group

When neither the sender nor the recipient belongs to a group, we further have two basic situations: if there is an existing contract between the sender and the recipient, and if there is no existing contract between the sender and the recipient. In the first case, the sender uses the existing contract to transmit messages. In the second one, the sender has to initiate a negotiation process. The sender can transmit messages to the recipient once a contract between them is generated. We call this scenario Scenario I.

3.4.2 The Sender Belongs to A Group

In this scenario, the sender belongs to a group but the recipient does not. The sender can send personal messages and group messages through its parent group.

If the sender attempts to transmit an personal message through its parent group, the message will be delivered if and only if:

- there is an existing contract between the parent group of the sender and the recipient, and
- the sender does not run out of its message quota.

Otherwise, the parent group of the sender has to negotiate a contract with the recipient or the sender solicits more message quota from its parent group.

In addition to sending personal messages, a sender can send group messages as well. If a sender wants to send a group message, the group message is stored into the MQ of its parent group. Because a group message represents the opinion of the group, whether a group message is sent is determined by the group and other group members. If the group determines to send the group message, the group becomes a sender and this case turns into the Scenario I, where neither the sender nor the recipient belongs to a group.

3.4.3 The Recipient Belongs to A Group

When a sender does not belong to a group but a recipient does, the message will be sent if and only if there is an existing contract between the sender and the recipient. Otherwise, the sender has to negotiate a contract with the recipient. This case is the same as the Scenario I.

In this scenario, we do not require that there is an existing contract between the sender and the parent group of the recipient, because if the recipient is a cell, which may have multiple parent groups, the situation would become too complex to meet the requirement.

3.4.4 Both The Sender And The Recipient Belong to A Group

If both a sender and a recipient belong to a group, we also have two cases. If the sender wants to send a group message, which represents its parent group or all members in the same group, the group message will be deposited into the MQ. Whether the group message is sent is determined by the group. Under this circumstance, the group becomes a sender and this scenario falls into the scenario described in Section 3.4.3, where the sender does not belong to a group but the recipient does.

If a sender intends to send an personal message through its parent group, the message will be sent if and only if:

- there is an existing contract between the parent group of the sender and the recipient, and

- the sender does not run out of its message quota.

From the above scenario analysis, we should notice that whether the recipient belongs to a group or not does not affect communication between the sender and the recipient.

3.4.5 Communication Scenario Examples

In this section, we give two examples to illustrate the four communication scenarios. In the first example, we describe how communicating entities interact. In the second example, we use our approach to model real world email communication.

Example 1 (Figure 3.6). Consider Alice, represented by a cell, wants to send a message to Bob, represented by another cell. Alice belongs to two groups: one is *student* and another is *staff*. Bob is a member of *faculty* group. Alice holds a contract with Bob. At the same time, *student* and *staff* also have a contract with Bob separately. The message quotas of Alice in *student* and *staff* are 50 and 30, respectively. Under this circumstance, Alice can transmit messages to Bob directly. If she like, she also can send personal messages to Bob through *student* group and *staff* group as long as the number of messages she has sent through these two groups does not exceed 50 and 30, respectively.

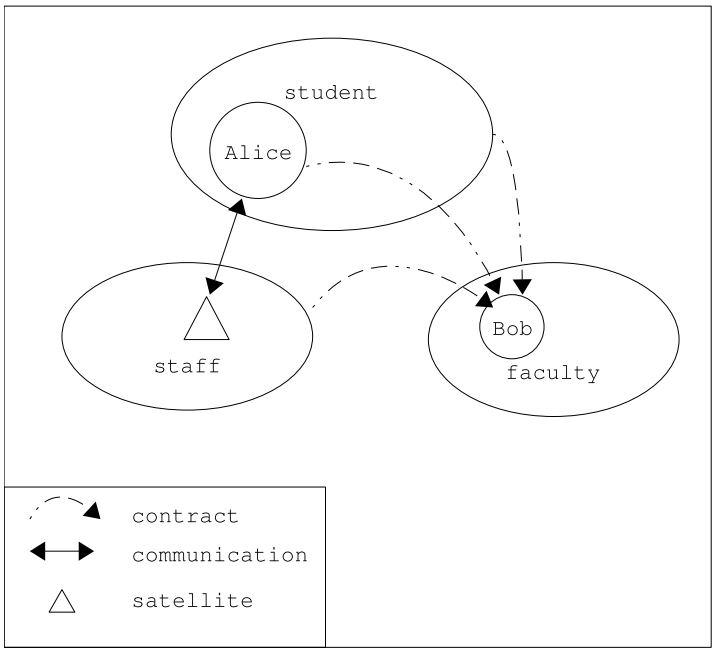


Figure 3.6: Example: communication between two cells

Example 2 (Figure 3.7) In this example, Alice has an email account in Gmail. Bob has an email account in YahooMail. If Alice wants to send email to Bob, Alice must have a contract with Bob; at the same time, Gmail must have a contract with YahooMail. When an email goes from Alice to Bob, that email use the contract between Alice and Bob. However, when Gmail sends an email to YahooMail, the communication between them is regulated by the contract between them. Under this circumstance, Gmail is a sender and YahooMail is a recipient. If there is no existing contract between Gmail and YahooMail, Alice cannot send email to Bob.

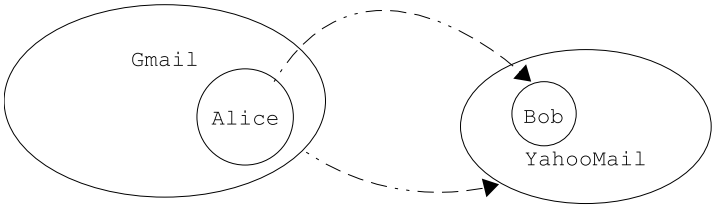


Figure 3.7: Example: communication between two email users

3.5 Penalties

As suggested in Section 3.3, it is possible that the system is flooded by numerous refusal messages if senders make a large volume of unacceptable proposals. A large number of refusal messages are undesirable in that they waste resources. This back door may be exploited by spammers to abuse the system in real world. Under this circumstance, senders should have right for making a proposal. If a sender does not hold such right, the sender cannot make a proposal. We call this right *meta-contract*. We introduce a self-protective mechanism called the penalty mechanism. A communicating entity is considered as abnormal provided that it makes proposals without a meta-contract.

Recall that in the CyberOrgs model, there is a root cyberorg, which encapsulates all other cyberorgs in the system. In our approach, there is a global cyberorg, which contains and manages all cells and groups in the system. Each communicating entity has *penalty policies*. Like policies in Section 3.3.1, penalty policies can be specified for a particular cell or group. Typical penalties can be *blocking* penalties and *financial* penalties. If one abnormal cell or group violates the penalty policy of another cell or group, the later one will apply penalties to the former one by reporting to the global cyberorg, which may either block any sending attempt from the abnormal one to the reporter or appropriate the abnormal one a certain amount of eCash dependent on the penalty policy of the reporter.

3.5.1 Penalty Policy Example

In this example, one cell has a penalty policy like: if anyone sends me 50 unacceptable proposals per day, I will block it for 10 days. With this constraint, the cell or group who sends more than 50 non-acceptable proposals daily is considered as the abnormal one by the policy holder, which declines any message from the abnormal one for ten days. A financial penalty policy can be like: if anyone sends me 50 non-acceptable proposals, I will claim ten e-pennies for compensation.

3.6 Chapter Summary

This chapter presents an ownership-based message admission control mechanism. The approach is built from the CyberOrgs model. In our approach, every communicating entity is a cyberorg. A mailbox is considered as a precious resource of its owner. Access to the mailbox is controlled by its owner. If one intends to send messages to another, the former must negotiate a contract with the later. A contract represents access permission, which allows a sender to deliver messages to a recipient. To negotiate a contract, both the sender and the recipient must possess policies. Whether the contract is generated or not is based on their policies. We introduce *group*s, which are in analogous with organizations in human society. Groups can control their resource consumption. By joining a group, a communicating entity can use resources of its parent group. To avoid too many refusal messages, our approach has a penalty mechanism, which discourages communicating entities deliberately deteriorate system performance.

Chapter 4

Prototype Design and Implementation

In this chapter, we describe a proof-of-concept prototype implementation based on our approach discussed in the previous chapter. The prototype is developed using the Actor Architecture (AA) [30], which is an implementation of the Actor model [2] of concurrent computation.

The organization of this chapter is as follows: We briefly introduce the Actor model and Actor Architecture in Section 4.1. Section 4.2 presents the system architecture. Section 4.3 introduces the types of messages in our implementation. We distinguish between normal messages, which constitute user communication, from auxiliary messages. The implementation of cyberorgs including cells and groups is described in Section 4.4. Message handlers, which provide communication services to communicating entities, are introduced in Section 4.5. Section 4.6 discusses the central authority responsible for issuing eCash and enabling contract accountability. Section 4.7 and Section 4.8 elaborate how cyberorgs select a policy when they have multiple choices and how policies are separated from mechanisms. The implementation of communication patterns is discussed in Section 4.9. The last section summarizes this chapter. In this chapter, we use cyberorgs and communicating entities interchangeably.

4.1 Actor Model and Actor Architecture

The Actor model offers an object oriented foundation for concurrent computation. An actor encapsulates state, behavior, a message buffer, and a thread of control (Figure 4.1). Actors can be viewed an active communicating object. In this thesis, we think of actors as independent processes which can run concurrently.

Because the internal behavior of an actor is encapsulated, it cannot be observed from outside directly. Actors interact with the external world only through sending and receiving asynchronous messages. Each actor has a unique name and a message buffer to receive messages in. Actors compute by processing messages received in their message buffer. An actor will wait if its message buffer is empty. An actor can only communicate with actors for which it has names. An actor a can know another actor b's name in one of the following ways:

- b's name is contained in a message received by a

- b's name was provided to a at its own creation time
- a created b

To respond to a message, an actor performs a local computation (which may be represented by any computer program) and three primitive actor operations:

- send messages: an actor may send messages to other actors.
- create actors: an actor may create new actors with particular behaviors.
- become ready to accept a message: an actor becomes ready to process the next message in its message buffer.

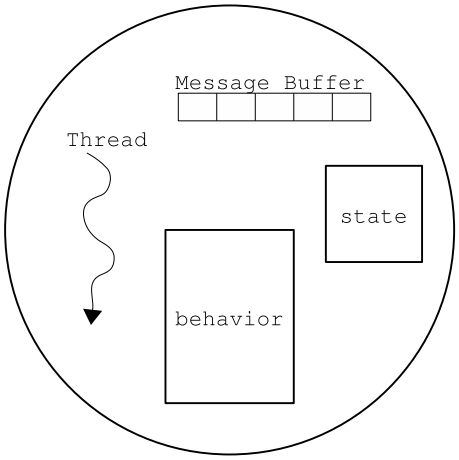


Figure 4.1: An actor

Actor Architecture (AA) is a Java-based implementation of the Actor model. AA is a middleware system architecture, which provides an Actor execution environment and an Actor API to implement actors. An Actor execution environment on a computer node is called an AA platform. Each actor executes on an AA platform, which provides communication services to actors.

Because actors naturally model communicating concurrent processes, we use actors to implement the communicating entities in our system.

4.2 System Architecture

There are three layers in our system architecture. From top to down, they are user interface (UI) layer, cyberorg layer, and message handler (MH) layer, shown in Figure 4.2. At the sender side, messages flow from the UI layer to the MH layer, and at the recipient side, messages flow in the opposite direction.

The UI layer presents graphic user interfaces or command user interfaces to allow human users to exchange messages with others and perform management tasks, such as configuring policies and purchasing e-pennies.

The cyberorg layer hosts cells and groups. As we mentioned in Chapter 3, cells and groups are communicating entities, which provide communication facilities and management facilities to the UI layer. In our prototype, cyberorgs cannot communicate with each other directly. By contrast, each cyberorg must have an associated message handler, which is located in the MH layer. Cyberorgs interact with the external world through their affiliated message handlers.

The MH layer provides communication services to cyberorgs in the cyberorg layer. Moreover, message handlers maintain information on behalf of associated cyberorgs. For example, policies of a cyberorg are stored on its associated message handler so that the message handler can make a proposal or evaluate a proposal on behalf of the cyberorg automatically. Recall that cyberorgs use eCash to trade in resources; each cyberorg's eCash account is also maintained by its corresponding message handler. We discuss these information in detail in Section 4.5.

Moreover, there is a Central Authority (CA) in the MH layer as well. One responsibility of the CA is to issue eCash. Another purpose of the CA is to ensure contract accountability. As we suggested in Chapter 3, one product of a negotiation process is a contract. In our prototype, each participant holds a copy of the contract. A third copy of the contract is deposited on the CA for accountability purposes. CA is transparent to cyberorgs. Only message handlers are aware of the existence of the CA.

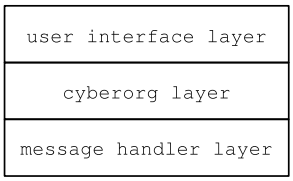


Figure 4.2: System architecture: three layers

4.2.1 Overview

As shown in Figure 4.3, in our prototype, equivalent to email accounts, cyberorgs are sources and targets of messages. Corresponding to real email servers, message handlers play the server role in our prototype. To simplify our discussion, we refer to cyberorgs as *clients* of their corresponding message handlers, which are *local* message handlers of the cyberorgs. The CA, which is a component absent in existing email systems, issues eCash and supports contract accountability for the system.

A message handler runs on a computer node (or on an AA platform). However, one computer node may host multiple message handlers. Each message handler is a server which provides

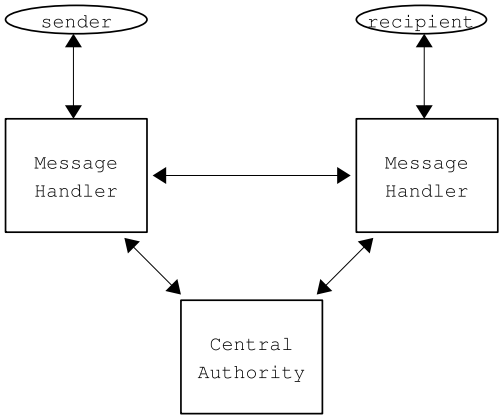


Figure 4.3: Big picture of the prototype

messaging services to its clients.

The next section describes types of messages used in our prototype. The remainder sections elaborate the implementation of our prototype in detail.

4.3 Messages

In our prototype, messages are categorized into two groups: *system messages* and *user messages*. System messages are invisible to cyberorgs. Message handlers and the CA transfer system messages among them to carry out fundamental functionalities, such as storing contract, purchasing eCash, etc. System messages are not delivered to cyberorgs.

User messages can be sent and read by cyberorgs. They are further grouped into four classes, including *normal messages*, *request messages*, *response messages*, and *error messages*.

Normal messages Normal messages come from senders and target to recipients. Normal messages contain general purpose communication information between senders and recipients.

Request Messages A sender issues a request message to a recipient when the former intends to send a message to the later. Each request message contains a proposal.

Response Messages A recipient sends a response message which corresponds to a request message. The response message indicates whether the recipient accepts or rejects the sender's proposal.

Error Messages Error messages are response messages. When a response message declines the sender's proposal, we call it an error message in our implementation. A sender should not raise too many unacceptable proposals, because they would result in the same number of error messages.

4.4 Cyberorgs

In this section, we introduce the implementation of cyberorgs. Cyberorgs, including cells and groups in the system, are implemented using actors. Besides communication and negotiation, cyberorgs have capabilities to settle their sending and receiving policies, to join and depart a group, and to purchase e-pennies from their message handlers, etc.

4.4.1 Cells

A cell can be naturally implemented by an actor, which has a thread of execution and a message buffer to receive messages. Each cell must have an associated message handler to send and receive messages. To affiliate with a message handler, a cell must register itself to one by requesting a message handler to accept it. Once receiving the registration request, the target message handler adds the requester to its client pool. Afterwards, the cell can communicate with the external world through its associated message handler.

In addition to sending and receiving messages, a cell needs to configure its policies, which are stored on its local message handler. On the one hand, as a message sender, a cell can be a publisher. A publisher provides messaging services to others. On the other hand, as a message recipient, a cell can be a subscriber. A subscriber subscribes messaging services provided by publishers. Under the publisher/subscriber circumstance, the subscriber should initiate a negotiation process to receive messages.

If a cell run out of its eCash, it has to purchase some from its local message handler. Otherwise, it can not send messages. When a cell intends to join or depart one of its parent groups, it sends a request to its local message handler, which, in turn, processes the request on behalf of the cell. We will discuss it in detail shortly.

4.4.2 Groups

Like a cell, a group must affiliate a message handler to interact with the external world. In order to associate with a message handler, a group must register itself like a cell does. As communicating entities, a group also holds sending and receiving policies, which are kept on its local message handler.

A group may permit cells or other groups to join it or may allow its members to leave it. In our implementation, these group primitives are carried out through the two APIs: `joinAGroup` and `departAGroup` exposed by the group class. Because the members of a group is variable, the group maintains a list to track its members.

Group members can use resources of their parent group. In our implementation, resources that can be used directly are policies. In other words, a member can use policies of its parent

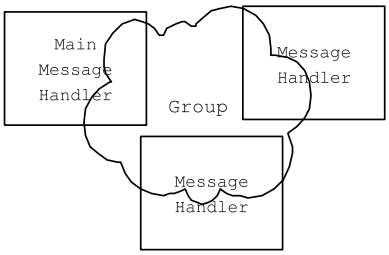


Figure 4.4: A distributed group which has only one main message handler

group to make proposals or evaluate proposals. Group members use contracts of their parent group indirectly.

A member can send messages through its parent group. Members can send two types of messages: personal messages and group messages. Because group members consume resources which are stipulated by the contracts between them and their parent group, their parent group must have a mechanism to impose constraints on its members to prevent its resources from being abused. To achieve this goal, as we suggested in Chapter 3, a group restricts the maximum number of personal messages that each member can send by setting message quotas. Each group member has a default message quota. Furthermore, a member may have particular message quota for particular recipients. For example, Alice in the student group, which has a contract with the faculty group, can send 10 messages to a recipient in the faculty group; however, her message quota maybe 20 when a recipient is in the staff group. Hence, each group member may have multiple message quotas. A group can configure arbitrarily complex message quota for each member. As long as the number of messages have sent does not exceed its message quota, personal messages from that member will be sent.

A member may issue a group message, which is stored in the outgoing message queue of the parent group. Whether the group message is sent is determined by *group message policies* of the parent group. For instance, a group message policy can be like: a group message can be sent if more than fifty percent of group members accepts it. When a member sends a group message, the message does not consume message quotas of that member. We will describe group message policies further in Section 4.9.

As shown in Figure 4.4, a group may consists of members from multiple message handlers. In this case, we say the group is a *distributed* group. Although a distributed group spans among multiple message handlers, only one message handler, which is called its *main* message handler, provides messaging services to the group.

4.5 Message Handler

To interact with the external world, each communicating entity must have an associated message handler. Besides providing communication services, a message handler also is responsible for maintaining information for its clients, such as policies, contracts, eCash account, etc. In this section, we delineate message handlers' chores on managing clients' information. The communication services will be discussed in a separated Section 4.9. Message handlers are implemented using actors.

This section is organized as follows: Section 4.5.1 discusses the policy pool, which is used to maintain clients' policies. The contract pool, which is employed to manage clients' contracts, is described in Section 4.5.2. Each cyberorg has an eCash account, which is addressed in Section 4.5.3. Section 4.5.4 discusses how a message handler monitors behaviors of its clients; therefore the message handler can make some decisions on behalf of its clients. How a message handler prevents others from abusing its resources is illustrated in Section 4.5.5.

4.5.1 Policy Pool

A message handler manages a policy pool, which contains sending and receiving policies of its clients. In addition to `add` and `remove` policies from the pool, the most important task of the policy pool is to select a policy to make a proposal for senders or evaluate a proposal for recipients.

Upon accepting a message from its client, if the message does not consist of a proposal, the local message handler generates a proposal for it automatically based on policies of the client. This task is delegated to its policy pool. The interface used for this purpose in the implementation is as follows:

```
public Proposal makePpl(String sender, String recipient)
```

The `sender` argument specifies the message sender, who has to make a proposal if there is no available contract between the sender and the recipient. The `recipient` argument specifies the message recipient. As we mentioned in Chapter 3, the sender prefers making a particular proposal to making a general proposal if there is a particular sending policy for the recipient. If the `recipient` parameter is presented, the `makePpl` method will choose a particular policy rather than a general policy to make a proposal if there is a particular one.

When a recipient-side message handler receives a proposal from a sender, it delegates the evaluation task to its policy pool, which selects a receiving policy according to the sender/recipient pair and employ the selected policy to assess the proposal. The interface which carries out the task in the implementation is:

```
public boolean checkPpl(String recipient, String sender, Proposal ppl)
```

The `sender` and `recipient` arguments have the same meaning as in the method `makePpl`. The `ppl` arguments specifies the proposal to be evaluated.

In our prototype implementation, policies have two categories: *implicit* policies and *explicit* policies. Recipients or senders have implicit policies for handling individual incoming or outgoing messages. These policies lead to implicit negotiations followed by implicit contracts for individual messages with senders or recipients. Explicit policies are employed when a sender attempts to construct a long-term relationship with a recipient; thus the sender does not need to negotiate with the recipient for every message. These policies lead to explicit negotiations followed by explicit contracts for multiple messages with recipients. An example of an implicit policy is: two e-pennies for sending one message. "100 e-pennies for sending 60 messages" is an example of an explicit policy. We will discuss implicit contracts and explicit contract in the next section.

4.5.2 Contract Pool

Recall that senders can offer implicit proposals using implicit policies. An acceptable implicit proposal leads to a *implicit contract*, which is used only once. In our implementation, implicit contracts are not stored in the contract pool because they are transient and only control one message communication. An *explicit contract* is generated by a negotiation using explicit policies. An explicit contract exists for a while to regulate subsequent communication between a sender and a recipient. In the following, when we say contracts, we mean explicit contracts.

After a contract is generated, both the sender-side and the recipient-side message handlers holds a copy of the contract, which is stored in their contract pools. Besides from storing contracts, a contract pool is responsible for two important functions: contract-checking and contract-generating.

At the sender side, when a sender wants to send a message, the sender-side message handler checks whether there is an existing contract between the sender and the recipient. If yes, the message will be delivered using the existing contract. If not, the sender-side message handler will activate a negotiation on behalf of the sender. The contract-checking assignment will be carried out by its contract pool using the following interface:

```
public int checkContract(String sender, String recipient)
```

At the recipient side, if the message handler receives a normal message from a remote message handler, the recipient-side message handler also needs to determine whether there is an existing contract between the recipient and the sender. If yes, the message will be transmitted to the recipient immediately. If not, the message will be discarded. If the message handler accepts a request message from a remote message handler, it designates its contract pool to generate a contract for the sender and the recipient. The generated contract is stored in both the recipient-side and the sender-side message handler. A third copy is deposited in the CA for the accountability purpose, which will be illustrated in Section 4.6. The contract-generating interface looks as follows:

```
public Contract generateContract(RequestMsg rm)
```

The argument `rm`, which encapsulates a proposal, indicates that a contract is generated from a request message. The request message also contains information of the sender.

4.5.3 eCash Account

Each cyberorg has an eCash account on its related message handler, which records the balance of the account. When a cyberorg wants to purchase e-pennies, it issues a request to its associated message handler. If the message handler has sufficient eCash, it will sell the requested number of e-pennies to the intended cyberorg. Otherwise, the message handler will purchase e-pennies from the CA and then sell them to the intended cyberorg.

Like a joint banking account in real world, an eCash account can be shared by multiple cyberorgs. Each eCash account has a main owner. Joint owners can be added to the eCash account. In a negotiation process, the sender designates which eCash account is used in its proposal. The eCash account has capabilities to check if the claimed sender is permitted to use it or not.

4.5.4 Behavior Monitor

In the real world, people always make decisions based on their previous behaviors and acquired knowledge. Similarly, in our prototype, cyberorgs need to know their previous behaviors to determine how to interact with others in the future. Each cyberorg has its own policy for making decisions. Policies used to monitor the behavior of cyberorgs are called *trigger* policies. An *event* is triggered once a certain condition is satisfied, which is defined in a trigger policy. For example, commonly, a sender may send messages to particular recipients regularly. To maximize its benefits, the sender may negotiate a new contract with recipients when special conditions defined in their trigger policies are satisfied.

To detect whether an event should be triggered, cyberorgs have to trace their own behavior, whereas it is too trivial to be efficient. In our prototype, we warrant the message handlers to do these chores for their clients. Trigger policies of a cyberorg are kept on its local message handler, which monitors behaviors of its clients. For example, if a cyberorg holds a trigger policy like: if I have sent 10 messages to Bob since yesterday, I would like to negotiate a contract with Bob, which allows me to send him 20 messages per day. In this case, the message handler will start a negotiation process with Bob provided that the regulated condition has been reached.

Implementation

Although the behavior monitor mechanism can detect any kinds of behavior of cyberorgs and can make any kinds of event be triggered, in our implementation, we only allow message handlers

representing their clients to initiate a negotiation.

As we mentioned in Section 4.5.1, cyberorgs has two types of sending policies: one type is implicit and another type is explicit. If the sender wants to communicate with the recipient frequently, the sender may be tired of using implicit policies for each communication, although it is the sender's message handler not the sender itself to start a negotiation process. In this case, the sender would like construct a long-term relationship using explicit policies.

Each cyberorg has its own distinct trigger policies, which are kept on its associated message handler. At the same time, a message handler also maintains a default trigger policy for its clients in order to provide facilities to clients who do not possess their own trigger policies. Like sending and receiving policies, trigger policies can be customized for particular cyberorgs.

In our implementation, the message handler counts the number of messages that the monitored client has sent. If the number meets the defined number in the trigger policy, the message handler will start a negotiation on behalf of the monitored client. For example, if Alice holds a trigger policy like: If I have sent 10 messages to Bob, I would like to negotiate a contract with him. Under this policy, when Alice's message handler detects that the number of messages goes to Bob from Alice surpasses 10, it will start a negotiation process automatically on behalf of Alice.

4.5.5 Penalties

As we suggested in the previous chapter, unacceptable proposals may cause unproductive communication. In our prototype, it is message handler's task to regulate the behavior of its clients. The abnormal clients are punished by their local message handlers. In turn, an abnormal message handler, which emits too many unacceptable proposals, is punished by other message handlers. To achieve this goal, a message handler maintains two tables. One is the *threshold table*, which tracks the number of unacceptable proposals that its clients can send. Another is the *penalty table*, which records which clients and which message handlers have been punished or are being punished.

A message handler may customize special thresholds for an individual client in the threshold table. For example, a message handler may allow one client to send 10 unacceptable proposals, but does not permit another to send 5. Once the number of unacceptable proposals a client has sent rises above its threshold, the client is put into the penalty table.

Each message handler has its own penalty policies. Members in the penalty table are punished according to penalty policies. For example, if a penalty policy is: blocking any message, then the message from the punishing member will be discarded by the message handler which holds this policy. At the sender side, the message handler examines whether the sender is in the penalty table. If yes, the message will be discarded; otherwise, the message will be delivered to the recipient-side message handler. At the recipient side, the message handler determines whether the remote message handler is in its own penalty table. If yes, any messages from the remote message handler will be

discarded; otherwise, messages will be evaluated or be delivered to the recipient.

Penalty Implementation

We construct a penalty mechanism to discourage users from deliberately forging unacceptable proposals. We implement two kinds of penalties to solve this problem: the financial penalty and the temporary blocking penalty. Corresponding to the two penalties, we settle two types of thresholds: the financial threshold and the temporary blocking threshold. In the following, we first describe the financial penalty and the temporary blocking penalty on the message handler level. Finally, we illustrate these two penalties on the system level.

Financial Penalty The first type of threshold is the financial threshold. We use t_f to denote it. If the number of refusal messages caused by a client beats t_f, the message handler will catch this event and punish the client by deducting some e-pennies from the client's eCash account. The message handler manages how many e-pennies should be decremented from the client's eCash account. Moreover, in a message handler, its clients may share a global t_f or the message handler may set up different financial penalties for different clients. In other words, clients may be punished individually. For example, the message handler may want to give more financial penalty to clients who make unacceptable proposals more frequently.

Temporary Blocking Penalty Another type of threshold is the temporary blocking threshold. We denote it as t_b. Generally, t_b is greater than t_f. When the total number of refusal messages caused by a client surpasses t_b, the message handler will block any messages from that client for a limited time period t. After this time period expires, the client is eligible to send messages again. However, if the same user continues to make unacceptable requests and the number of refusal messages exceeds t_b again, the message handler stop providing services to the client for a time period $2 \times t$. Next time will be $2 \times (2 \times t)$, and so on.

Like the financial threshold, the message handler holds a global t_b for all its clients; however, the message handler may also designate an individual value for a particular client. Again, the time period t may be shared or individual.

Punishing Message Handlers In previous discussions, we elaborated how to block a malicious cyberorg who purposely makes unacceptable requests on the message handler level. However, we have another problem: what if a message handler does not control unacceptable requests from its affiliated users? In this situation, the irresponsible message handler is punished by other message handlers. Otherwise, if others allow the abnormal one to make unsatisfactory requests, the abnormal one will waste resources of them, such as computational resources, memory, and network bandwidth.

Hence, if a message handler does not control its abnormal clients, the reckless one will risk to be penalized by others.

Message handlers use temporary blocking penalty rather than financial penalty to punish abnormal ones. A message handler may set a global temporary blocking threshold t_b or set specific t_b for particular message handlers. Again, the blocking time period t may be shared or individual.

4.6 Central Authority

There is a third party called the central authority (CA) in the system. One purpose of the CA is for issuing e-pennies. Only message handlers know the existence of the CA, which is transparent to cyberorgs. Cyberorgs buy e-pennies from their affiliated message handlers, which, in turn, purchase e-pennies from the CA.

Another purpose of the CA is for contract accountability. When a contract is produced by a negotiation process, it is necessary to store a copy of the contract to the CA for the accountability purpose to prevent the recipient and the sender from denying their responsibility.

4.6.1 eCash Account

Each message handler owns an eCash account on the CA. Recall that each cyberorg owns an eCash account on its associated message handler. Unlike cyberorgs' eCash account, a message handler's eCash account cannot be shared by multiple message handlers. The relationship between these two different types of eCash account is described as follows:

Let us suppose the total amount of e-pennies of the CA has issued is S. There are m message handlers in the system. The number of affiliated cyberorgs of jth message handler is n. Let us use b_{ij} to denote the balance of the eCash account of ith cyberorg on its related message handler j, and B_j to denote the balance of the eCash account of jth message handler on the CA. The number of e-pennies which are not sold by jth message handler is denoted by e_j. We have:

$$e_j + \sum_{i=1}^{n} b_{ij} = B_j \tag{4.1}$$

$$\sum_{j=1}^{m} B_j = S \tag{4.2}$$

From the equation 4.1 and equation 4.2, we can infer that when the balance of the eCash account of a cyberorg is alternated by n e-pennies, the balance of the eCash account of its associated message handler is also incremented or decremented by n correspondingly.

4.7 Policy Selection

Recall that a cyberorg may have multiple sending and receiving policies. Conditions become more complex when a cyberorg is a member of a group, which may be a member of another group. A cyberorg can also use the policies of its parent group, even the policies of the parent group of its parent group. The hierarchy of CyberOrgs brings a new problem: policy selection. For example, Alice has her own sending and receiving policies. She also belongs to the Student group, which, in turn, is a member of the University group. Each group has its own sending and receiving policies. This situation brings a complicated problem: which policies should Alice adopt when she makes or evaluates proposals.

In our implementation, the basic rule is that cyberorgs prefer particular policies to general policies. In other words, as long as a cyberorg can find a particular policy, they will adopt the particular one. Otherwise, the cyberorg will choose a general policy. In Section 4.7.1, we utilize an example to illustrate our implemented selection strategy.

4.7.1 Policy Selection Example

We use Figure 4.5 as an example to elaborate our discussion. In Figure 4.5, a cyberorg *c1* is a member of two groups: *g2* and *g3*. At the same time, *g2* and *g3* are located in a bigger group *g1*.

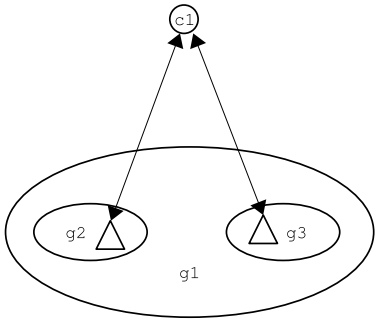


Figure 4.5: Policy selection example

Let's suppose the cyberorg *c1* intends to select a policy to interact with another cyberorg *c2*. *c1* holds some policies. Each group in the figure also possesses its own policies. The local message handler of *c1* selects a policy for *c1* by carrying out the following steps:

1. Determining whether there is a particular policy for *c2* in *c1*'s polices. If yes, the selecting process is done and go to step 7.

2. Randomly selecting a group from *g1*, *g2*, and g3.

3. Checking whether there is a specific policy for *c2* in the chosen group's policies. If yes, go to step 7.

4. Choosing one of other groups and repeating step 3

5. Checking whether *c1* has a global policy, if yes, go to step 7

6. Repeating step 5 using g1/g2/g3 [1] instead of *c1*.

7. Completing the selection process.

In step 2, we randomly choose a group from three groups, because we intend to treat each group equally. The policy selection strategy may be various in practice. Different users may have different preferences when it comes to selecting a policy. For example, one cyberorg may use policies of its parent group first and then choose its own policies. We only present one possible policy selection strategy here. More interesting strategies need to be developed.

4.8 Policy Implementation

In this section, we discuss how our implementation separates policies from mechanisms and present several policy examples to illustrate our solution. In Section 4.8.1, we expose the public policy interface, which is used to develop novel policies. In Section 4.8.2, we present four policy examples using pseudo-code to show developers how to develop a new policy.

4.8.1 Interface

To separate policies from mechanisms, we expose a public policy interface, which can make and evaluate a proposal, to developers, who can create their own policies for their own or public purposes.

The public policy interface looks as follows:

```
public abstract class Policy implements Serializable {
    int type;     //The type of policy
    int epennies; //Number of e-pennies

    //Each policy can make a proposal
    public abstract Proposal makePpl();
    //Each policy can evaluate a proposal
    public abstract boolean checkPpl(Proposal ppl);
}
```

[1] which means whatever order

The above code snippet indicates that each policy possesses two fields and two methods. The `type` field describes the type of the policy. The `epennies` field specifies the number of e-pennies. The method `makePpl` can generate a proposal based on the policy. Whether a proposal is acceptable is determined by the `checkPpl` method.

4.8.2 Implemented Policies

In this section, we present four policy examples. They are *implicit policy*, *message policy*, *day expiration policy*, and *message per minute policy*.

Implicit Policy As we mentioned above, implicit policies regulate one message communication. In our implementation, an implicit policy describes the price of one message. The code snippet looks like as follows:

```
public class ImpPolicy extends Policy {

    public boolean checkPpl(Proposal ppl) {
       ......
       int proposedEp = get the number of e-pennies from the proposal;
       if  (proposedEp >= epennies)
            return true;

        return false;
    }

    public Proposal makePpl() {
        ImpPpl ppl = new ImpPpl();
        ppl.setEp(epennies);
        return ppl;
    }
}
```

To simplify our implementation, when making a proposal on behalf of its client, the policy will use a called "lazy" strategy to parametrize the proposal. Like the above code, the policy uses its "epennies" value to fill in the field of the made proposal rather than a value which is less than the "epennies" value.

Message Policy A message policy specifies that how many e-pennies should be provided to compensate for a given number of messages. For example, "10 e-pennies for 10 messages" and "20 e-pennies for 10 messages" are message policies.

```
public class MsgPolicy extends Policy {
    int    nrMsgs; // The number of messages

    public boolean checkPpl(Proposal p) {
       ......
       int proposedEp = get the number of e-pennies from the proposal;
       int proposedMsgs = get the number of messages from the proposal;
       if  ((proposedEp/proposedMsgs) >= (epennies/nrMsgs))
            return true;

        return false;
    }

    public Proposal makePpl() {
        MsgPpl ppl = new MsgPpl();
        ppl.setEp(ep);
        ppl.setNumOfMsg(nrMsgs);

        return ppl;
    }
}
```

Day Expiration Policy An expiration policy controls that given a number of e-pennies, how many days a sender can deliver messages to a recipient. A day expiration policy, such as "100 e-pennies for 30 days," indicates that the sender offers 100 e-pennies to send messages to the recipient for 30 days or the recipient ask 100 e-pennies for receiving messages for 30 days.

```
public class DayPolicy extends Policy {
    int nrDays;

    public boolean checkPpl(Proposal p) {
        int proposedEp = get the number of e-pennies from the proposal;
        int proposedDays = get the number of days from the proposal;
```

```
        if ((proposedEp/proposedDays) >= (epennies/nrDays))
            return true;

        return false;
    }

    public Proposal makePpl() {
        DayPpl dp = new DayPpl();
        dp.setDays(nrDays);
        dp.setType(type);

        return dp;
    }
```

Message Per Minute Policy Message per minute policy describes how many messages a sender can send to a recipient per minute within a time period. For example, a message per minute policy can be such as: "10 e-pennies for 2 messages per minute for 5 minutes".

```
public class MinLtdMsgPolicy extends MsgPolicy {
    int mins;

    public boolean checkPpl(Proposal p) {
        int proposedMins = get proposed minutes from the proposal;
        int proposedMsgs = get the number of messages from the proposal;
        int proposedEp = get the number of e-pennies from the proposal;

        if ((proposedMins <= mins) && (proposedMsgs <= nrMsgs))
            return true;

        return false;
    }

    public Proposal makePpl() {
        MinLtdMsgPpl ppl = new MinLtdMsgPpl();
        ppl.setEp(ep);
        ppl.setMins(mins);
        ppl.setNumOfMsg(nrMsgs);
```

```
        return ppl;
    }
}
```

The above policy implementation examples show us how policies are separated from mechanisms. They also show developers how to implement their own policies. That is, developers must implement `checkPpl` and `makePpl` methods.

4.8.3 Proposals

Notice that in each `makePpl` method above, the proposal is built from its corresponding policy. In other words, for each policy, we can make a corresponding proposal. It is obvious that a message policy cannot generate a day expiration proposal, and vice versa.

4.9 Communication

In this section, we narrate how cyberorgs interact with each other through message handlers and how eCash flows among individual participants. As we mentioned, a sender can send personal messages or group messages if it belongs to a group. We elaborate how to transmit these two types of messages in Section 4.9.1 and Section 4.9.2 separately. Section 4.9.3 addresses what happens when a recipient-side message handler receives a message from a remote message handler. The detailed transaction is described in Section 4.9.4.

4.9.1 Personal Messages

A sender can deliver personal messages through two different ways. First, the sender itself can transmit personal messages to the recipient. Secondly, the sender can transmit personal messages through its parent group if it has one.

Let us see the first case in the first place. When a sender intends to transmit a message, the complete message arrives at the sender's local message handler first. Afterwards, the complete message is deposited in the message list in its local message handler. Next, the message handler carries out the following steps to accomplish the communication attempt:

- Checking whether the sender is in the penalty table. If yes, the message will be discarded and the message handler relinquish the sending attempt.
- Checking whether there is a contract between the sender and the recipient. If yes, the message will be delivered to the recipient-side message handler and return.

- Checking whether there is a trigger policy whose condition is satisfied. If yes, the message handler will select a explicit policy to make a proposal. Otherwise, the message handler will select an implicit policy to make a proposal.
- Obtaining the eCash account which is responsible for paying for the message
- Retrieving the balance of the eCash account. If the funds are not sufficient, the message handler will abandon this sending attempt.
- Sending a request message which encapsulates the generated proposal to the recipient-side message handler and waiting for a response
- Receiving and checking the response message from the recipient-side message handler. If the proposal is accepted, the complete message will be delivered and return.
- Checking whether the number of refusal messages has reached the penalty threshold. If yes, the sender will be inserted into the penalty table.

For the second case, the personal message is stored into the outgoing message queue of its parent group. Recall that cyberorgs cannot communicate with each other directly. Associated message handlers provide messaging services to them. Therefore, the message goes to the parent group through the following path: *the sender's local message handler* → *the group's main message handler* → *the group*. Whether the message is sent is determined by two factors:

- there is an existing contract between the group and the recipient, and
- the sender does not run out of its message quota.

We should notice that if the first condition is not met, the arrival of the personal message makes the group trigger a negotiation process.

4.9.2 Group Messages

If a sender wants to send a group message which represents its parent group, the message will be stored in the outgoing message queue of its parent group. Other members of the same group can vote on the message. The voting result determines whether the message is sent or not. The group evaluates the result of vote based on its group message policies and makes a final determination.

In our implementation, we developed two types of group message policies: *coauthor policy* and *committee policy*. We call messages using a coauthor policy *coauthor messages* and messages using a committee policy *committee messages* in the following. A coauthor message has multiple authors. In other words, besides from the first author (initiator), a coauthor message has one or more coauthors. The coauthor message will be sent to the recipient if and only if all coauthors accept it.

A committee message belongs to a committee which is comprised of several members. Each member in the committee can vote on it. The committee message also has a property which describes under what condition the message is to be sent. For example, the condition may be how many percents of the committee members consent the message or how many committee members should agree on sending the message. We expose the public group message policy interface, thus, developers can develop their own group message policies.

In our implementation, to send a group message, the initiator composes and transmits the message to its parent group through its related message handler. The group message is stored in the outgoing message queue of the group. Then, the group sends a query message to all involved coauthors or committee members, which are members of the group. Next, the group waits for response from related participants. The coauthor message will be delivered provided that all coauthors have accepted it. The committee message will be transmitted as long as conditions defined in the committee policy is true. For example, if there are five members in a committee, the definition of a committee policy can be "at least three members in the committee should accept the message". Once three members have acknowledged the message, the committee message will be transferred.

Interchanging group messages among cyberorgs introduces a new communication pattern: *many-to-one*. Here "many" represents a group of senders, and "one" means a recipient. When a recipient is also a group, a variable pattern, *many-to-many*, is introduced.

4.9.3 Message Receiving

In our prototype, a recipient may receive two types of messages from a sender: request messages and normal messages. When receiving a message from remote message handlers, the message handler of the recipient examines the type of the message. If the message is a request message, it determines whether the proposal attached with the request message is acceptable based on the recipient's policies if any. If the proposal is acceptable, a contract will be generated. A copy of the contract will be sent to the CA and to the remote message handler. Otherwise, the message handler will transmit a refusal message to the sender on behalf of the recipient. If there is no policy related to the recipient on it, the message handler will transfer the request message to the recipient, who makes the final decision on whether accepting the proposal or not.

If the message is a normal message, the message handler checks whether the remote message handler is in its penalty table. If yes, the message will be abandoned. Next, it checks whether there is an existing contract between the sender and the recipient. If yes, the message will be delivered to the recipient. Otherwise, the message will be discarded right away.

At the sender side, a sender may receive refusal messages from a recipient. In this case, the recipient or its message handler sends a response message to the sender-side message handler. The response message suggests whether the sender's proposal is accepted or not. If the response message

is a refusal message, the sender-side message handler will try to make a better proposal on behalf of the sender. If the sender-side message handler cannot generate a better proposal, the refusal message will be sent to the sender and let the sender make a final decision: offering a better proposal or abandoning communication attempt with the recipient.

4.9.4 Transactions

To reach a contract, the sender needs to pay eCash to compensate the recipient for its mailbox consumption. In this transaction, eCash flows from the sender to the recipient. To maintain the consistency of equation 4.1 and equation 4.2, the same amount of eCash also flows from the sender-side message handler to the recipient-side message handler. In other words, in a communication process, modifying an eCash account of a cyberorg causes three other eCash account to be changed. They are accounts of both message handlers and the account of another participant. The same discussion also applies to the publisher-subscriber pattern; however, in this case, the subscriber is the negotiation initiator and the publisher receives e-pennies.

Cyberorgs can purchase e-pennies from their associated message handlers, which, in turn, purchase e-pennies from the CA. In this transaction, eCash flows from the CA to the eCash account of the message handlers. In the meanwhile, the same amount of eCash flows from the eCash account of the message handlers to the eCash account of the buyers.

To prevent false contracts, the contract is generated in the recipient-side message handler. Upon being generated, the recipient-side message handler sends one copy to the sender-side message handler and one copy to the CA, respectively. After receiving the copy of the contract, the sender-side message handler also send a request to the CA to add the same contract. The CA add the contract if and only if it receives two requests: one is from the sender-side message handler and another is from the recipient-side message handler. Otherwise, the system provides senders an opportunity to add false contracts. Likewise, the CA will delete a contract if and only if it receives one request from the sender-side message handler and one request from the recipient-side message handler.

4.10 Deployment

The core of our implementation is message handlers, which can correspond to real email servers. In Section 4.10.1, we study three possible scenarios in which message handlers can be applied. Message handlers can be deployed incrementally in the Internet, which is discussed in Section 4.10.2.

4.10.1 Scenarios of Using Message Handlers

Message handlers can be used in three possible ways to reduce spam, including independent email clients, plug-ins of current email clients or current email servers, and email servers. In the following, we use an email client as an example to illustrate how to use message handlers in real world. Usages as plug-ins or email servers are similar.

Message handlers can be used as an independent email client program. When a message arrives at an email client running on a computer, the email client checks whether there is an attached proposal with the incoming message. If there is an attached proposal, it assesses the proposal based on policies which has already been deposited on it by users. If the proposal is accepted, the message will be stored into the local inbox folder. Otherwise, the email client will automatically send the message sender a short message that asks the sender to make a better proposal. If there is no attached proposal, the email client checks whether there is an existing contract with the sender. If yes, the message will be stored into the local inbox folder. Otherwise, the email client assumes that the message comes from an email server or email client which is not compatible with it. Whether the message is accepted or not can be determined by other approaches, such as filters.

When the email client sends a message, it checks whether there is an existing contract with the recipient. If there is one contract, the email client knows that the recipient uses the same type of email systems. Otherwise, the email client will attach a proposal with the outgoing message and send it. If recipients use traditional email systems, the recipient can ignore the attached proposal. Otherwise, if recipients use our email clients, they can process messages using our mechanism.

We can develop an email client program based on the above discussion and allow it to be downloadable. To be used effectively, the program must present a user-friendly user interface to make users customize their policies easily.

4.10.2 Incremental Deployment

In this section, we discuss how to deploy message handlers incrementally. In our discussion, message handlers are considered as email servers.

When sending a message, a message handler sends it with a proposal if there is no existing contract between a sender and a recipient. If a recipient-side system is a current email system, it ignores the attached proposal. If the recipient-side system is a message handler, the message can be handled using our mechanism.

As a recipient, a message handler checks whether there is a proposal attached with the incoming message. If yes, the message handler realize that the incoming message is sent by another message handler. In this case, the message is processed using our mechanism. If there is no attached proposal, the message handler determines whether there is an existing contract between the message

sender and the recipient. If no contract exists, the message handler assumes that the incoming message is sent by a traditional email system. In this case, the incoming message is checked by other techniques, such as filters [54, 20] or tarpits [14, 17].

Through the above description, we know that message handlers can be deployed incrementally in the Internet. After more message handlers are deployed, a message handler can work more effectively.

4.11 Chapter Summary and Discussion

In this chapter, we present a prototype implementation of the ownership-based message admission control mechanism discussed in the previous chapter. In our implementation, cyberorgs, which are communicating entities, are implemented using actors. Rather than communicating each other directly, cyberorgs interact with each other through their associated message handlers, which provide messaging services to cyberorgs. We introduce a central authority, which issues eCash and ensures the accountability of contracts.

In addition to facilitating communication, a message handler maintains information of its clients, including policies, contracts, eCash accounts, and trigger policies. An eCash account can be shared by multiple cyberorgs. To avoid being overwhelmed by refusal messages, each message handler manages a penalty table, which records abnormal message handlers and its abnormal clients. Entities in the penalty table are punished through either financial penalties or temporary blocking penalties.

eCash is not real money although it can be. In our implementation, a buyer does nothing more than only sending a request to its related message handler to purchase e-pennies. Upon receiving the request, the message handler deposits the required number of e-pennies into the buyer's eCash account. In real implementations, eCash can be made through multiple ways, such as solving CAPTCHAs [59] or computing a challenge. In this case, the central authority is a server which issues CAPTCHAs or challenges and authenticates the answers. eCash also can be purchased from the central authority using real money. In our implementation, we do not provide a mechanism to authenticate eCash. However, eCash authentication is one of most serious security problems in practice. For example, malicious hackers may commit to forge e-pennies. Thus, the central authority must provide authentication services in real systems.

We introduce *group* concept corresponding to organizations in human society. Groups can control consumption of their resources. A group member can use policies of its parent group. In Section 4.7, we present our policy selection strategy. However, in practice, individual users can have different preferences when choosing a policy. For example, Alice may choose a policy from one of her parent groups first instead of hers. A group uses message quota mechanism to restrict

its members' communication behavior. A group message has a relevant group message policy. A group message can be sent as long as the condition defined in its relevant group message policy is satisfied.

We address four examples of policies in Section 4.8. Developers can develop more useful and interesting policies by extending our public policy interface. Policies can be arbitrarily complex. By employing the policy interface, we separate policies from mechanisms.

Our approach is not a complete solution to spam problems. By contrast, this approach is an complement of existing approaches. In our approach, we do not classify whether a message is spam or a regular email. Thus, our approach precludes false positives, which are major problems of filters, which are widely deployed at present. In our approach, as long as spammers are willing to send spam, they must negotiate contracts with recipients. The contracts stipulate the cost of sending messages.

Chapter 5

Analysis and Experimental Results

In our approach, we add a negotiation process into normal communication to control message admission. Since negotiation introduces computation and communication overhead, in this chapter, we construct analytical models to analyze our prototype system and design and carry out a series of experiments to validate our analytical results.

The purpose of this chapter is to provide evidence that as long as we design policies carefully, a negotiation system has almost the same message processing capacity as a current email system. In this chapter, we examine the system in three scenarios. They are when the sender sends messages directly to the recipient, when the sender sends messages using implicit policies, and when the sender sends messages using explicit policies. To simplify our discussion, when a system is in the first scenario, we call the system the direct system. We call the last two the implicit policy system (IP) and the explicit policy system (EP), respectively. An IP system is a system with negotiation using implicit policies. An EP system is a system with negotiation using explicit policies. An IP system or an EP system is called a system with negotiation (SN). In the following, we use SN systems to represent IP systems and EP systems if we do not need to distinguish them from each other.

In an SN system, the analysis of a recipient-side message handler is straightforward. For example, on receiving a message, if there is an existing contract between the sender and the recipient, the message handler will deliver the message to the recipient; otherwise, the message handler will discard the message. Thus, in this chapter, we only focus on analyzing sender-side message handlers. In the following, when we say "message handler" or "sender," we mean "sender-side message handler" unless we specifically add "recipient-side" before "message handler."

The organization of this chapter is as follows. Section 5.1 constructs analytical models based on queuing theory and presents analytical results. In Section 5.2, we design and carry out several experiments to validate our analytical results. Finally, the last section summarizes this chapter.

5.1 Analytical Models

In this section, we construct analytical models to analyze how negotiation impacts on system performance. In Section 5.1.1, we briefly introduce the queuing theory [36], which is used to build and analyze our analytical models. In Section 5.1.2 and Section 5.1.3, we analyze our system from the processor and network perspectives.

5.1.1 Queuing theory

In this section, we concisely introduce queuing theory [28] and notations used in our analysis. In the queuing theory, the following quantities are measured:

λ, the job arrival rate

R, the average response time, that is, the time interval between arrival time and departure time

S, the average service requirement

W, the average waiting time, that is, the time interval between arrival time and the instant beginning to receive service

U, the utilization of the system

N, the number of jobs in the system including jobs waiting to receive service and jobs receiving service

Additionally, following equations hold true:

$$R = W + S \tag{5.1}$$

$$W = N * S \tag{5.2}$$

$$N = \lambda * R \tag{5.3}$$

and

$$U = \lambda * S \tag{5.4}$$

Equation 5.1 shows that the average response time is equal to the sum of the average waiting time and the average service requirement. When a job arrives, its average waiting time is equal to the product of the number of jobs in the system and the average service requirement (as shown in equation 5.2). Equation 5.3 and Equation 5.4 are the *Little's law* and the *utilization law*, respectively. The assumption under which these two laws are made is that jobs are not lost due to insufficient buffers. Under this circumstance, the throughput of a system is equal to the job arrival rate of the system. The Little's law describes that the average number of jobs in a system is equal

to the product of the job arrival rate and the average response time. The utilization law means that the utilization of a system is equal to the product of the job arrival rate and the average service requirement in that system.

In the rest of this section, we employ the above notations and equations to analyze our prototype system. When it comes to describing a direct system, we add a subscript d to the notations, such as λ_d. Likewise, we add a subscript s to the notations to represent a quantity in an SN system.

5.1.2 Processor Analysis

In the processor analysis, we analyze the utilization of an SN system. Figure 5.1 shows the observed system.

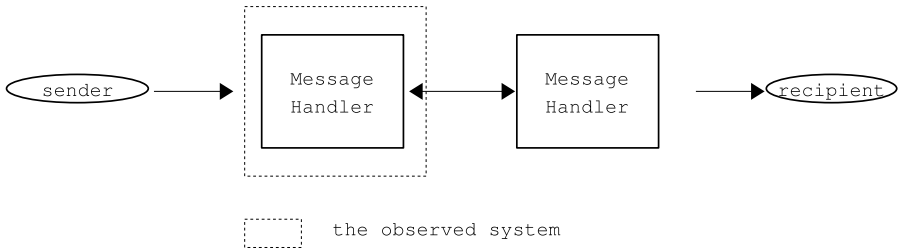


Figure 5.1: The observed system - message handler

In our analysis, we assume that senders have sufficient eCash so that they do not need to purchase e-pennies, which causes extra communication and computation. Another assumption is that senders always make acceptable proposals. So there is no error messages in the system. To describe our system more accurately, we define the following additional quantities:

S_c, the average service requirement when there is an existing contract in the system

S_n, the average service requirement when there is no existing contract in the system

P_c, the probability that there is an existing contract in the system

P_n, the probability that there is no existing contract in the system

From these measurements, we can define the average service requirement for an SN system:

$$S_s = P_c \times S_c + P_n \times S_n \tag{5.5}$$

Obviously, the average service requirement of an IP system is $S_s = S_n$.

Utilization Analysis

We determine how negotiation impacts the system utilization. To achieve this goal, we measure that when direct and SN systems have the same utilization, what message arrival rate these two systems respectively have.

$$\lambda_d S_d = \lambda_s S_s \tag{5.6}$$

Combined with the equation 5.5, we have:

$$\frac{\lambda_d}{\lambda_s} = \frac{P_c \times S_c + P_n \times S_n}{S_d} \tag{5.7}$$

From the equation 5.7, we can infer that when both systems have the same utilization, the ratio of the message arrival rates is $\frac{P_c \times S_c + P_n \times S_n}{S_d}$. Notice that $P_c + P_n = 1$ and $S_n \geq S_c \geq S_d$. We can further conclude that if a direct system has the same utilization as an SN system, its message arrival rate is higher.

Response Time Analysis

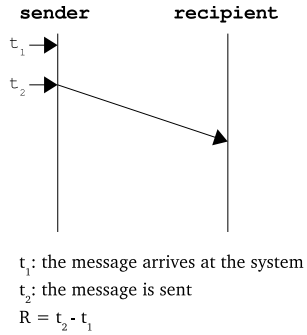


Figure 5.2: Processor response time analysis - a direct system

In a direct system, response time is the time interval between the arrival time (t_1) and the departure time (t_2). This time interval consists of two ingredients: one is the waiting time, and another is the service requirement. As shown in Figure 5.2, the response time can be expressed as follows:

$$R_d = t_2 - t_1 \tag{5.8}$$

The analysis of response time is more complex for an SN system. Illustrated in Figure 5.3, if there is an existing contract between the sender and the recipient, the response time is $R_n = t_2 - t_1$. If there is no existing contract, the response time consists of three parts:

- T_a, the time interval between the message arrival time and the time when a corresponding request message is sent
- T_w, the time interval between the time when a request message is sent and the time when the corresponding response message is received
- T_l, the time interval between the time when the corresponding response message is received and the message departure time

Further, T_w includes a round-trip network latency, denoted by T_D, and the recipient *thinking* time: $T_{rt} = t_4 - t_3$. Summarizing the above analysis, the processor response time of an SN system is:

$$R_s = P_c \times T_a + P_n \times (T_a + T_D + T_{rt} + T_l) \tag{5.9}$$

Notice that $P_c + P_n = 1$, the above equation can be simplified as follows:

$$R_s = T_a + P_n \times (T_D + T_{rt} + T_l) \tag{5.10}$$

From the equations 5.10 and 5.8, we can conclude that when P_n is significantly small, the average response time of an SN system is almost the same as the average response time of a direct system. In other words, if a sender needs to negotiate a contract more frequently, the average response time of an SN system will increase. Thus, the response time is policy-dependent in an SN system.

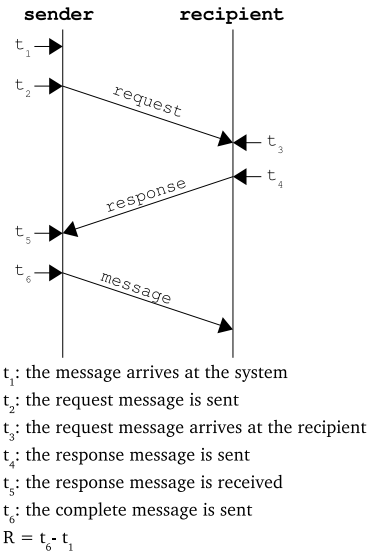


Figure 5.3: Processor response time analysis - an SN system

5.1.3 Network Analysis

In our network analysis, the observed system is a network connection, illustrated in Figure 5.4.

The following quantities are measured:

S_a, the average service requirement of application messages

S_e, the average service requirement of service messages

P_c, the probability that there is an existing contract in the system

P_n, the probability that there is no existing contract in the system

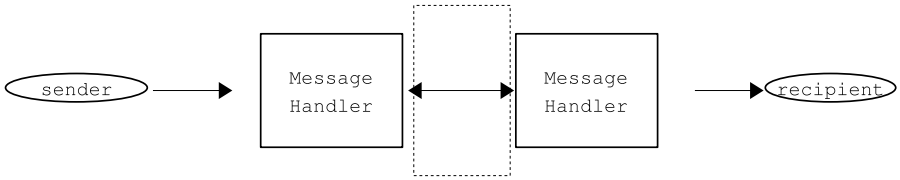


Figure 5.4: The observed system: network connection

In an SN system, when sending a message, if there is no existing contract between a sender and a recipient, three messages should be processed, including the normal message, one request message, and one response message. We call normal messages application messages and the last two messages service messages in our analysis. Compared to variable length of application messages, the length of service messages is almost constant for all application messages. Hence, we assume that the average service requirement of service messages is the same. Therefore, the average service requirement for an SN system is as follows.

$$S_s = P_c \times S_a + P_n \times (S_a + 2S_e) \tag{5.11}$$

We should notice that the sum of P_c and P_n is 1, thus the above equation becomes

$$S_s = S_a + 2P_n \times S_e \tag{5.12}$$

Moreover, the average service requirement for a direct system is:

$$S_d = S_a \tag{5.13}$$

Network Utilization

Similar to the utilization analysis of processors, we assume that the utilization of an SN system is the same as the utilization of a direct system. Combined the equation 5.6 with the equations 5.12 and 5.13, we have:

$$\frac{\lambda_d}{\lambda_s} = 1 + \frac{2P_n \times S_e}{S_a} \tag{5.14}$$

From the above equation, we can infer that when both systems have the same utilization, the ratio of the message arrival rates is $1 + \frac{2P_n \times S_e}{S_a}$.

Compared to the average service requirement of normal messages, the average service requirement of extra messages is smaller. Therefore, if P_n is very small, the fraction $\frac{2P_n \times S_e}{S_a}$ can be ignored. This means that when the probability that there is no existing contract in the system is significantly small, the network utilization of both systems is almost the same.

Network Response Time

In a direct system, the network response time of a message is the sum of the message service requirement and the network latency[1], denoted by T_D in this thesis. Hence, the network average response time of a direct system is:

$$R_d = S_a + T_D \tag{5.15}$$

In an SN system, if there is no existing contract between a sender and a recipient, a message will not be sent until a negotiation process completes and a contract is reached. If there is an existing contract, the message will be sent right away. Thus, the network average response time can be expressed as follows:

$$R_s = P_c \times (S_a + T_D) + P_n \times (S_a + 2 \times S_e + 3 \times T_D) \tag{5.16}$$

Because of $P_c + P_n = 1$, the above equation can be simplified as:

$$R_s = S_a + T_D + 2 \times P_n \times (S_e + T_D) \tag{5.17}$$

From the above equation, we can conclude that if P_n is trivial, the part "$2 \times P_n \times (S_e + T_D)$" becomes insignificant, the network average response time of an SN system will be almost the same as the network average response time of a direct system. Put differently, the network average response time of an SN system depends on the probability that there is an existing contract between the sender and the recipient. This probability, in turn, is dependent on policies.

5.2 Experimental Results

In this section, we present our experimental results. We carry out experiments and compute system utilization and system performance of our prototype system. We use simulated fixed-length messages to examine our system. Two assumptions are made. One is that senders have sufficient eCash; and another is that senders always make acceptable proposals.

In our system, per message overhead is significant; but per user overhead or per node overhead may be insignificant. In other words, our system may not affect per user overhead or per node overhead considerably; but influences per message overhead significantly. Therefore, our experiments focus on examine per message overhead instead of per user or per node overhead. The same discussion applies to system workload. This means our system may not affect per user workload or per node workload.

This section is organized as follows: Section 5.2.1 addresses how to compute the system processing time, system busy time, and system utilization. The rest of this section discusses experimental results and our analysis.

[1] Here, network latency is one-way delay, not round-trip

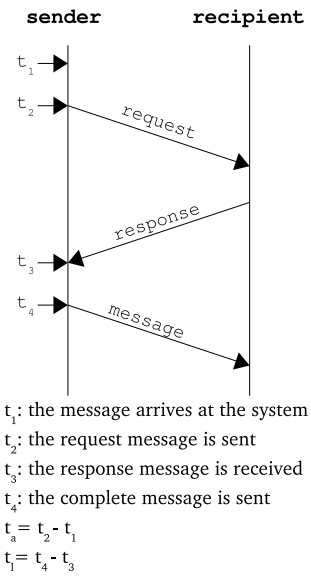


Figure 5.5: Communication pattern in an SN system

5.2.1 Analysis

When a message arrives at a direct system or an SN system which has an existing contract between a sender and a recipient, the service requirement is the time interval between the arrival time and the departure time. However, if a message arrives at an SN system which does not have a contract between the sender and the recipient, the service requirement for this message includes two time intervals: the interval between the arrival time and the time when the corresponding request message is sent; and the interval between the time when the corresponding response message is received and when it departs. We use t_a to denote the first interval and t_l to denote the second interval. Figure 5.5 illustrates this scenario.

Thus, the service requirement for this message is:

$$S_n = t_a + t_l \tag{5.18}$$

Suppose that the number of messages that has been sent within the observed time period T is C. The observed time period T is the time interval between the first message arrival time and the last message departure time. If the time intervals for the i^{th} message are denoted by t_{a_i} and t_{l_i}. Hence, an SN system busy time is:

$$B = \sum_{i=1}^{C} t_{a_i} + \sum_{i=1}^{C} t_{l_i} = \sum_{i=1}^{C} (t_{a_i} + t_{l_i}) \tag{5.19}$$

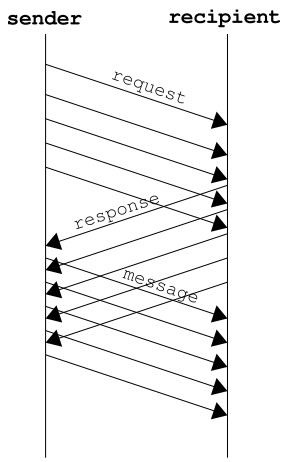


Figure 5.6: Communication pattern of an IP system

According to the queuing theory, the system utilization is:

$$U = \frac{B}{T} \tag{5.20}$$

Thus, within the observed time period T, the system utilization is:

$$U = \frac{\sum_{i=1}^{C}(t_{a_i} + t_{l_i})}{T} \tag{5.21}$$

For a direct system, each message service requirement does not include the second time interval, thus

$$U_d = \frac{\sum_{i=1}^{C} t_{a_i}}{T} \tag{5.22}$$

In an IP system, the communication pattern can be depicted in Figure 5.6. The figure indicates that each message needs a negotiation process. Thus, a system utilization of an IP system can be expressed using equation 5.21.

In an EP system, the communication pattern is shown in Figure 5.7. The busy time is:

$$B_e = P_c \times \sum_{i=1}^{C} t_{a_i} + P_n \times \sum_{i=1}^{C}(t_{a_i} + t_{l_i}) \tag{5.23}$$

Notice $P_c + P_n = 1$, the above equation becomes:

$$B_e = \sum_{i=1}^{C} t_{a_i} + P_n \times \sum_{i=1}^{C} t_{l_i} \tag{5.24}$$

Thus, the system utilization of an EP system is:

$$U_e = \frac{\sum_{i=1}^{C} t_{a_i} + P_n \times \sum_{i=1}^{C} t_{l_i}}{T} \tag{5.25}$$

In an EP system, when a message arrives, if there is no existing contract in the system, this message triggers a negotiation process. Following messages have to wait until the negotiation

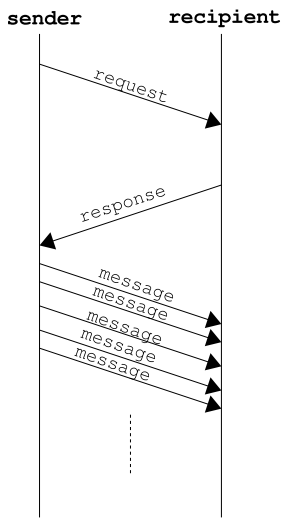


Figure 5.7: Communication pattern of an EP system

process completes and a contract is reached. If they do not wait, each message will trigger a negotiation process, which is not the negotiation purpose. However, in the Actor Architecture, actors cannot be suspended except by the Actor platform. Therefore, in our experiments, when we examine an EP system, if a message arrives and there is no existing contract at that instant time, the sender will stop sending messages for a while. This time period must guarantee that the negotiation process can complete. In our experimental environment, we choose this time period as 300 milliseconds.

Experiments for examining system utilization, busy time, and processing time are based on the above analysis. Experiments for examining processor response time are based on the analysis in Section 5.1.2.

5.2.2 Experiment Environments

We carry out experiments on two computers. Each computer runs an actor platform. Each actor platform has one message handler and one cell. In our experiments, one cell as a sender delivers messages to another, which plays the recipient role. The sender runs on a laptop computer, which has an Intel P4 1.9 GHz CPU and 768MB memory. The recipient runs on a desktop computer, which has an Intel Core Duo 1.86 GHz CPU and 2GB memory. Both computers run Ubuntu Linux 7.04.

In the following figures and tables, prefixes *D*, *IP*, and *EP* have the following meaning:

- *D* means a direct system.
- *IP* means an IP system

- *EP* means an EP system.

The suffix *0S* means a sender sends messages without any delay. In other words, we allow the sender to send messages as fast as possible. *1S* means the sender sends a message every 1 millisecond. The *100* or *200* means frequency of negotiating a contract, such as *100* means every 100 messages and *200* means every 200 messages. For example, *EP0S-100* means that in an EP system, the sender sends messages as fast as possible; and every 100 messages, the sender needs to negotiate a contract.

5.2.3 Experimental Results

In this section, we present experimental results related to system utilization, total processing time, busy time, and response time.

Utilization

The data collected from the experiments is presented in Table 5.1 and Figure 5.8. As shown in Figure 5.8, system utilization decreases as message arrival rate decreases. When the sender sends messages as fast as possible, the utilization of direct systems is higher than the utilization of SN systems. This is because we do not control the message arrival rate, the operating system job scheduling takes time.

When the system sends a message every 1 millisecond, the IP system has the highest utilization. This is consistent with our analysis. Considering an inserted 300 milliseconds delay for each negotiation when we carry out *EP0S-100* and *EP0S-200* experiments, it is not surprising that the utilization of EP systems is lower than the utilization of IP systems. If we deduct delays from the EP system, the utilization of EP systems is higher than the utilization of the direct system, which is also consistent with our analysis.

The utilization decreases as the number of messages increases, but not much, which is also due to the operating system job scheduling.

Processing Time

The total processing time is the time interval between the first message arrival time and the final message departure time. The experimental data is shown in Table 5.2 and Figure 5.9. As shown in the figure, the total processing time of each system appears to be linear with the number of messages. Obviously, the total processing time increases as the message arrival rate decreases.

Considering the added delays, it is not surprising that the experiments with *EP1S-100* has the biggest total processing time, because in comparison to the *EP1S-200*, *EP1S-100* has more inserted delays. However, after we deduct these delays, the total processing time of *EP1S-100* and *EP1S-200*

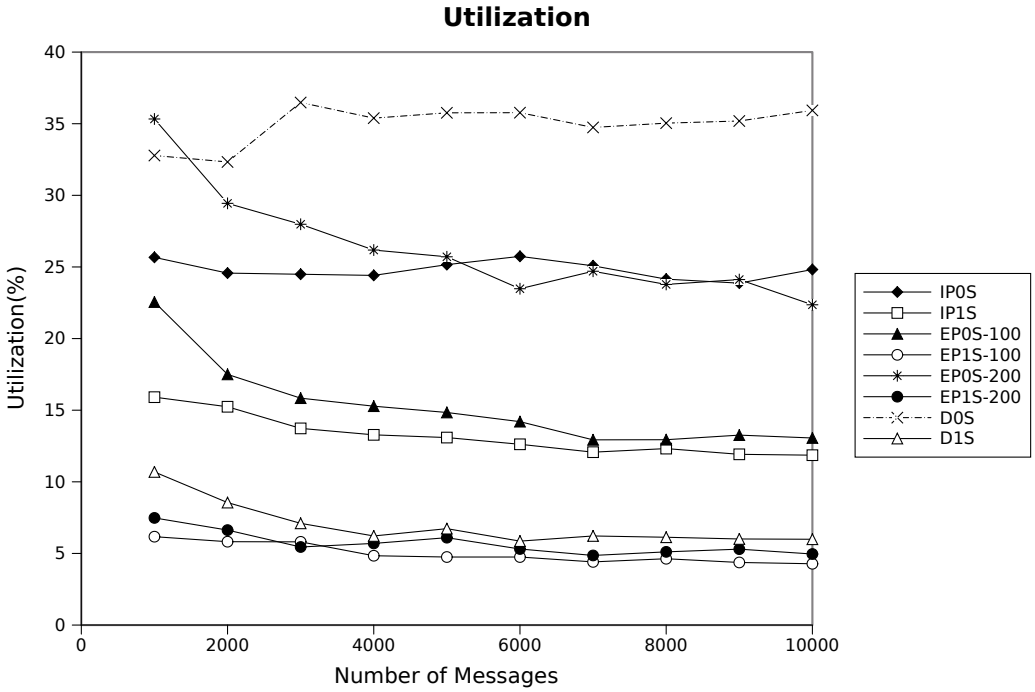


Figure 5.8: Comparison of the system utilization

Table 5.1: Experimental data on system utilization

Messages	D0S	D1S	IP0S	IP1S	EP0S-100	EP1S-100	EP0S-200	EP1S-200
1000	32.78	10.69	25.67	15.91	22.55	6.17	35.33	7.48
2000	32.33	8.55	24.57	15.24	17.5	5.82	29.44	6.63
3000	36.48	7.1	24.49	13.73	15.84	5.81	27.98	5.45
4000	35.39	6.22	24.41	13.28	15.28	4.84	26.18	5.7
5000	35.76	6.73	25.16	13.09	14.84	4.75	25.71	6.1
6000	35.77	5.86	25.74	12.62	14.2	4.75	23.48	5.31
7000	34.74	6.22	25.08	12.07	12.93	4.41	24.7	4.86
8000	35.04	6.13	24.15	12.32	12.94	4.63	23.78	5.11
9000	35.19	6.01	23.87	11.92	13.26	4.37	24.12	5.3
10000	35.93	5.99	24.82	11.86	13.06	4.28	22.36	4.96

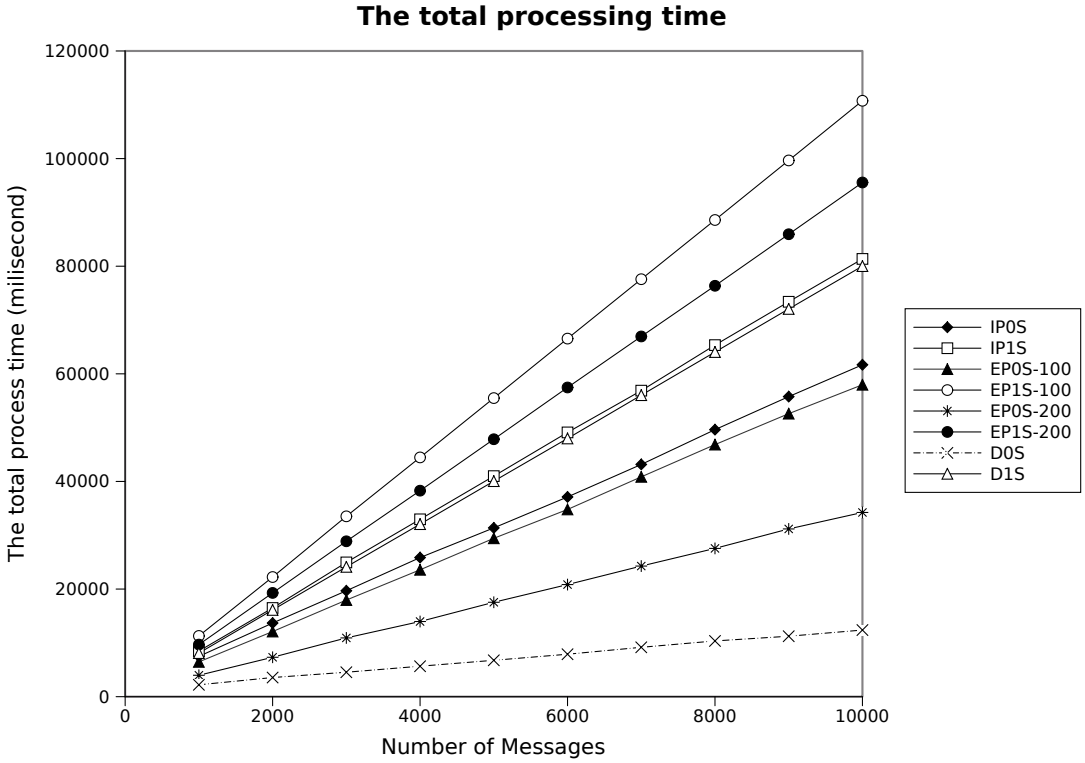


Figure 5.9: Comparison of the processing time I: the processing time that a system takes when processing a given number of messages

is between the total processing time of *IP1S* and the total processing time of *D1S*. In terms of total processing time, the percentage difference from *D1S* to *IP1S* is up to 4.7%. In other words, if users using our system still want to obtain the same message processing capacity as a direct system, the users must own or purchase a computer which has a 1.99 GHz CPU approximately (Recall that we used a 1.9 GHz CPU computer to carry out experiments).

Figure 5.9 also shows that as frequency of negotiation increases, total processing time increases as well provided that other conditions are the same. We can conclude that the total processing time is dependent on policies.

Figure 5.10 is an inverse version of Figure 5.9. It shows that how many messages a system can handle given an amount of processor time. Obviously, using the same amount of processor time, a direct system can process more messages than others; and an IP system processes fewer messages.

Busy Time

Figure 5.11 and Table 5.3 illustrate system busy time. Similar to the total processing time, the system busy time of each system appears to be linear as the number of messages increases. The SN system keeps the processor busier than the direct system. It is obvious that *IP0S* has the longest

Table 5.2: Experimental data on total processing time: Time is millisecond

Messages	D0S	D1S	IP0S	IP1S	EP0S-100	EP1S-100	EP0S-200	EP1S-200
1000	2208	8099	7488	8481	6467	11290	3981	9683
2000	3559	16118	13686	16480	12102	22237	7323	19283
3000	4544	24123	19682	24953	17953	33517	10921	28879
4000	5660	32061	25852	32962	23578	44465	13970	38286
5000	6762	40053	31367	40990	29432	55492	17545	47846
6000	7891	47986	37120	49113	34794	66547	20834	57463
7000	9178	56029	43160	56873	40853	77585	24274	66945
8000	10359	64049	49623	65338	46843	88589	27556	76362
9000	11236	72059	55763	73391	52595	99657	31175	85944
10000	12383	80016	61681	81352	57967	110748	34235	95558

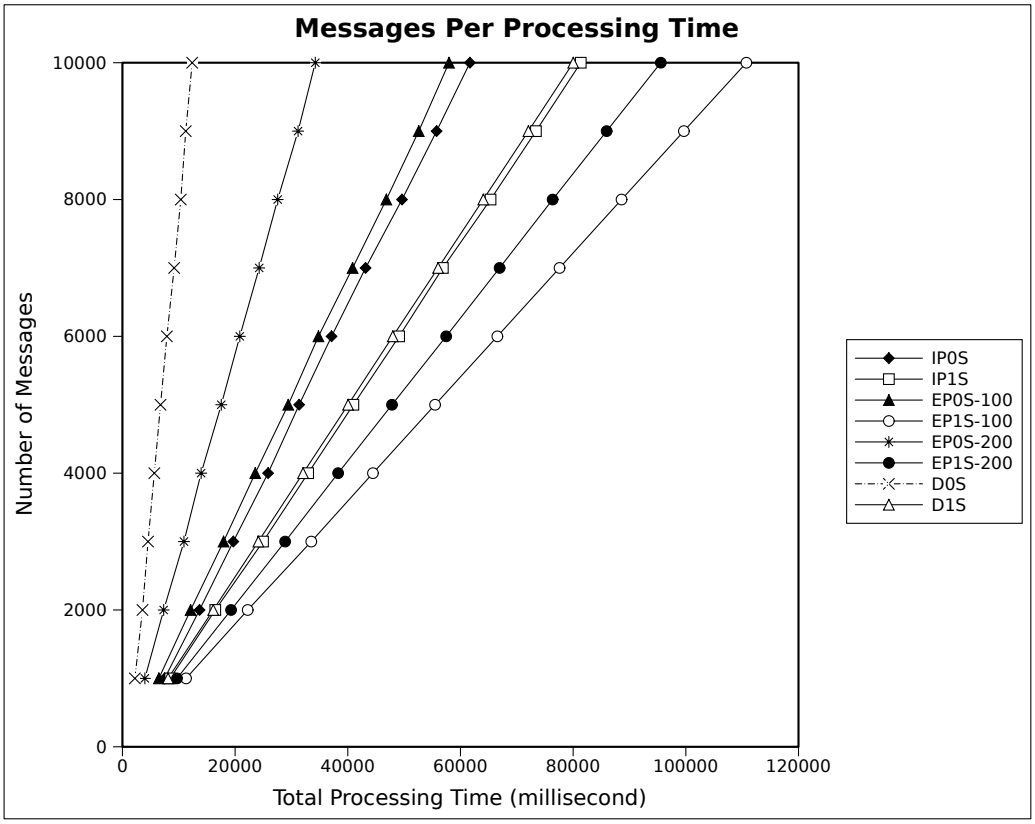


Figure 5.10: Comparison of the processing time II: the number of messages that a system can handle given an amount of processor time

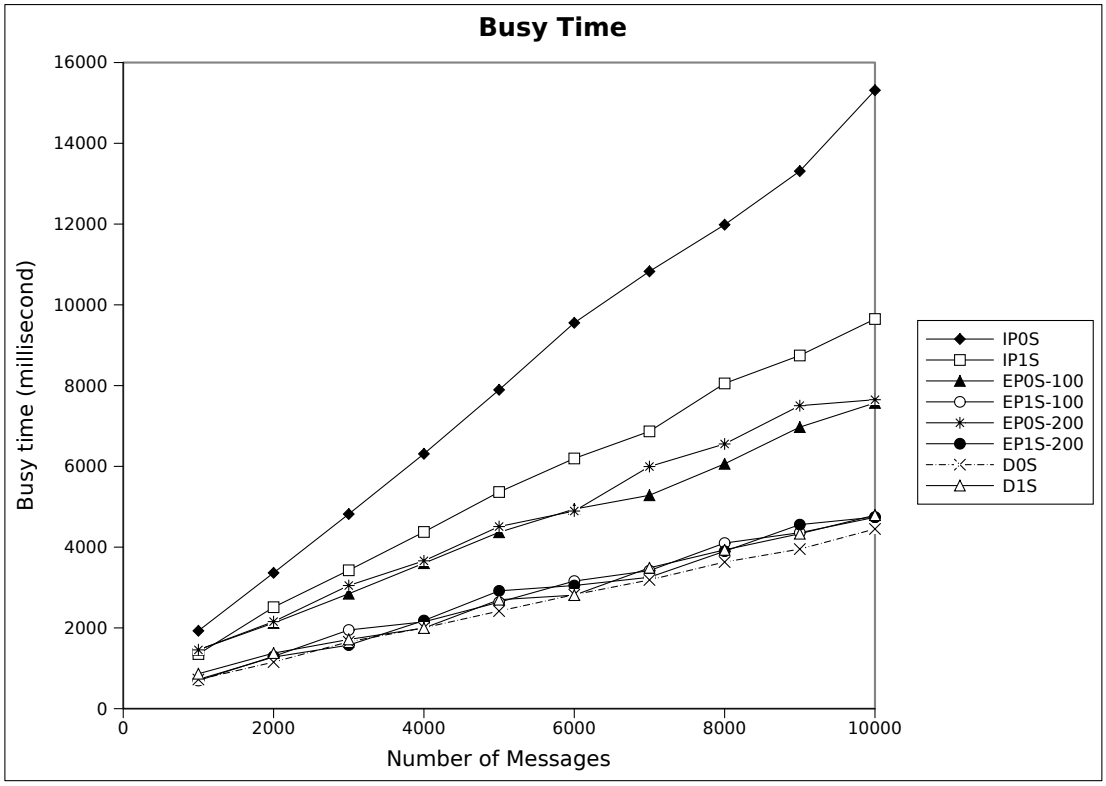


Figure 5.11: Comparison of the system busy time

busy time provided that other conditions are the same. As the message arrival rate decreases, the system busy time will decrease as well. Like total processing time, system busy time is also policy-dependent.

Response Time

Figure 5.12 shows the relationship between the frequency of negotiation and the response time. The number on the x-axis means the frequency of negotiation. For example, *100* means a negotiation process is needed for every 100 messages. We carry out 12 experiments to compare response time under different conditions. The collected data is listed in the Table 5.4. In each experiment, the sender sends 1,000 messages to the recipient. The figure indicates that as the ratio of message to contract increases, the response time of a negotiation system decreases.

We should notice that when a negotiation occurs every 25 or more messages, response time of negotiation systems is almost stable and it decreases asymptotically close to direct systems. This means that if we design our policies carefully, a system with negotiation can have minimal overhead in terms of response time.

Table 5.3: Experimental data on system busy time: Time is millisecond

Messages	D0S	D1S	IP0S	IP1S	EP0S-100	EP1S-100	EP0S-200	EP1S-200
1000	723	865	1928	1353	1458	696	1456	725
2000	1153	1376	3363	2513	2121	1295	2156	1278
3000	1657	1714	4819	3427	2846	1949	3050	1574
4000	2003	1996	6310	4376	3602	2150	3661	2183
5000	2417	2698	7895	5365	4368	2637	4510	2918
6000	2823	2811	9554	6196	4942	3160	4894	3050
7000	3188	3488	10828	6867	5284	3419	5996	3254
8000	3631	3932	11984	8054	6063	4101	6555	3898
9000	3952	4332	13311	8746	6972	4358	7503	4558
10000	4450	4791	15314	9649	7571	4739	7651	4744

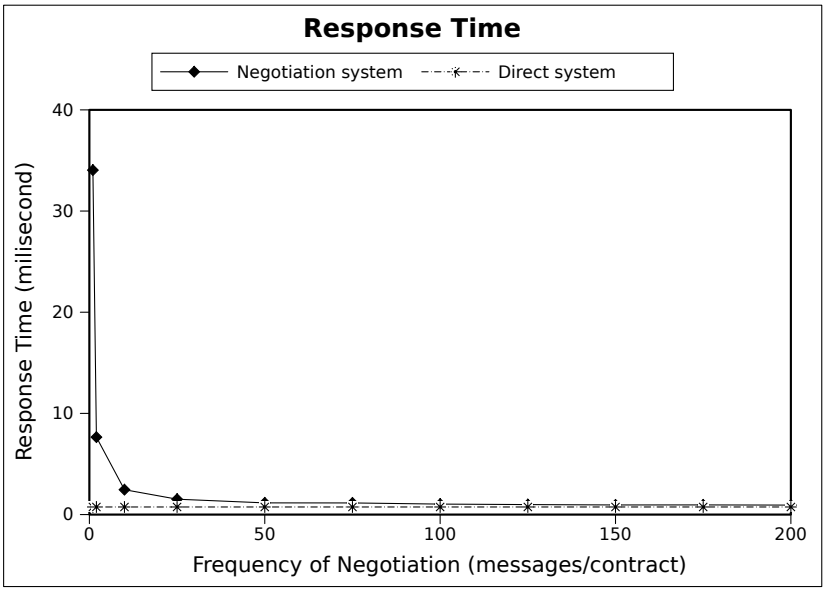


Figure 5.12: The response time vs. frequency of negotiation

Table 5.4: Frequency of negotiation vs. Response time: Time is millisecond

Messages/Contract	Response time
1	34.045
2	7.653
10	2.459
25	1.52
50	1.155
75	1.149
100	1.035
125	0.987
150	0.95
175	0.954
200	0.936
Direct System	0.75

System Overhead

We compute system overhead in terms of the number of extra messages, which include request messages, response messages, and system messages. In this discussion, the system includes both the sender-side message handler and the recipient-side message handler.

According to the discussion in Section 4.9.4, in an IP system, to send a message, 2 extra messages are generated in addition to one request message and one response message. These messages are responsible for updating the eCash accounts. In an EP system, besides one request message and one response message, 4 more extra messages are introduced. 2 of them modify eCash accounts and another two messages add a contract. Therefore, in an IP system, the extra messages are $4n$ if the number of normal messages is n. In an EP system, the extra messages are $6n$ if the number of negotiation is n. However, if a contract expires, the system will generate two more extra messages to delete the contract in the CA. The above analysis indicates that the system overhead is policy-dependent. More specifically, the fewer the number of negotiation times, the less the system overhead.

5.3 Discussion

In this chapter, although we carry out experiments on two computers (users), experimental results may be meaningful to multiple computers (users), because, in our system, system performance is

not related to the number of users in systems, but is related to the number of messages. On the other hand, more users in our system do not imply more messages. For instance, some users may send messages frequently, but others do not. Thus, the scalability of our system is not affected by the number of users in system. Moreover, in real deployment, our system can work as a plug-in of current email systems, leading our system to probably have the same scalability of current systems.

Our experiments do not study human user effort when human users use our system. Here, we discuss how our system may affect human user effort in real deployment. In direct systems, users do not need to install policies and purchase eCash; but they have to delete spam manually. Users using our system need to install policies and purchase eCash; However, usually, policies are installed when users start to use our system and they are not change frequently. eCash can be purchased in bulk. Hence, compared to spam deleting time, the time that human users should take maybe constant. Furthermore, installing policies and purchasing eCash can be completed by our systems automatically.

One limitation of our experiments is that simulated workload is used to examine our systems. However, we can analyze our system using real email workload when they are deployed. Starting from one node, we can deploy our system incrementally and examine workload of each node. Moreover, we can analyze how our systems help reduce spam effectively when they are widely deployed.

5.4 Chapter Summary

In this chapter, we employ the queuing theory to examine our prototype system discussed in the previous chapter. We construct analytical models to understand our implementation. Experiments are carried out using simulated workload to determine the system performance and check the accuracy of analytical models.

Through our analysis and experiments, we demonstrate that the system performance is policy-dependent, including utilization, busy time, processing time, and response time. It provides evidence to encourage users to design policies carefully in order to reduce negotiation. Moreover, the system busy time and system processing time increase linearly with the number of messages. This feature indicates if message service providers using our system attempt to improve system throughput, they must invest more on better performance computers.

Chapter 6

Conclusion and Future Work

This chapter makes a conclusion for this work and briefly introduces our future work.

6.1 Conclusion

Current email infrastructure does not enable recipients to control their mailbox consumption. Moreover, it does not restrict senders to deliver messages either. Our work intends to fill in this gap through introducing negotiation.

The purpose of this work is to provide a flexible mechanism to reduce spam and expose a public police interface to allow developers to develop customized policies. We construct an ownership-based message admission control mechanism for curbing spam by specializing the CyberOrgs model, which is a model for resource control in multi-agent systems. In our work, communicating entities are represented by cyberorgs. A mailbox, which is considered as a private resource of its owner, can be controlled in the boundary of a cyberorg. A sender obtains permission to access a recipient's mailbox through negotiation. In a negotiation process, a sender builds a proposal according to its own policies and submits it to the recipient. The recipient evaluates the proposal based on its own policies to determine whether the proposal is accepted. If the proposal is approved, a contract is generated, which regulates the following communication between the sender and the recipient. Otherwise, a refusal message is sent to the sender. A contract help the sender and the recipient construct a long-term relationship.

We build a prototype implementation in this work. We implement a negotiation mechanism and four policy examples. A public policy interface is exposed to programmers to allow them to develop more interesting policies. In our prototype implementation, we consider a possible security problem, in which malicious attackers may raise a large number of unacceptable proposals that waste resources. Our system counters this situation through a self-protective mechanism - a penalty mechanism.

We carry out experiments and present experimental results. The experimental results show system performance is policy-dependent. In other words, we have opportunities to design and develop policies carefully to make the system work efficiently. Our system was examined using

simulated workload instead of real email workload; however, we should analyze our system in terms of email clients or email servers in the Internet environment.

Our approach is not a complete spam control solution, but it is a supplement to existing approaches. Combined with other approaches, such as filter-based approaches or market-based approaches, our approach will work more effectively.

6.2 Future Work

This work creates a new approach for message admission control. In our prototype implementation, we do not consider one important issue - the security issue. However, to be deployed practically, the security issue must be addressed. Our future work should focus on solving the most infamous attack - Denial-of-Service (DoS) attack. Furthermore, because negotiation and communication involve eCash and contracts, malicious attackers may make our system work ineffectively through fabricating false eCash and false contracts. The new future implementation must add an authentication mechanism to resolve this problem as well.

We introduce a *group*, which is analogous with an organization in human society. Through groups, we may do more interesting researches on "many-to-one" and "many-to-many" communication mechanisms in the future.

In Chapter 3, we introduce a satellite, which is a proxy of a cell. In the future, we will introduce a *satellite of a group*, which encapsulates satellites of its group members.

References

[1] Martin Abadi, Andrew Birrell, Mike Burrows, Frank Dabek, and Ted Wobber. Bankable postage for network services. In *Proceedings of the 8th Asian Computing Science Conference*, Mumbai, India, December 2003.

[2] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[3] Mutual Internet Practices Association. Certified server validation. http://mipassoc.org/csv/.

[4] Adam Back. Hashcash. http://www.hashcash.org/.

[5] Daniel Bernstein. Internet mail 2000. http://www.im2000.org.

[6] Kumar Chellapilla, Kevin Larson, Patrice Simard, and Mary Czerwinski. Computers beat humans at single character recognition in reading based human interaction proofs (hips). In *Second Conference on Email and Anti-Spam (CEAS) 2005 Proceedings*, Mountain View, CA, July 2005.

[7] Cisco. Identified internet mail. http://www.identifiedmail.com/.

[8] Lorrie Faith Cranor and Brian A. LaMacchia. Spam! *Commun. ACM*, 41(8):74–83, 1998.

[9] Petmail Design. Hire people to solve captcha challenges. http://petmail.lothar.com/design.html#auto34.

[10] Christine E. Drake, Jonathan J. Oliver, and Eugene J. Koontz. Anatomy of a phishing email. In *First Conference on Email and Anti-Spam (CEAS) 2004 Proceedings*, Mountain View, CA, July 2004.

[11] Zhenhai Duan, Yingfei Dong, and Kartik Gopalan. Diffmail: A differentiated message delivery architecture to control spam. Technical Report TR-041025, Computer Science Department, Florida State University, USA, October 2004.

[12] Zhenhai Duan, Yingfei Dong, and Kartik Gopalan. Dmtp: Controlling spam through message delivery differentiation. In *In Proc. Networking 2006*, Coimbra, Portugal, May 2006.

[13] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *In Lecture Notes in Computer Science 740 (Proceedings of CRYPTO'92)*, pages 137–147, 1993.

[14] Tobias Eggendorfer. Reducing spam to 20% of its original value with a smtp tar pit simulator. In *Proceedings of MIT Spam Conference 2007*, March 2007.

[15] Scott E. Fahlman. Selling interrupt rights: A way to control unwanted e-mail and telephone calls. *IBM Systems Journal*, 41(4):759–766, 2002.

[16] Eran Gabber, Markus Jakobsson, Yossi Matias, and Alain J. Mayer. Curbing junk e-mail via secure classification. In *FC '98: Proceedings of the Second International Conference on Financial Cryptography*, pages 198–213, London, UK, 1998. Springer-Verlag.

[17] Cara Garretson. Tarpits deter impatient spammers. *Network World*, April 2007.

[18] Sharon Gaudin. 90% of e-mail will be spam by year's end. *Information Week*, March 2007.

[19] Joshua Goodman. Spam: Technologies and policies. *White Paper*, November 2003.

[20] Joshua Goodman, Gordon V. Cormack, and David Heckerman. Spam and the ongoing battle for the inbox. *Commun. ACM*, 50(2):24–33, 2007.

[21] Joshua Goodman and Robert Rounthwaite. Stopping outgoing spam. In *Proceedings of EC'04*, New York, New York, USA, May 2004.

[22] Paul Graham. A plan for spam. http://www.paulgraham.com/ spam.html.

[23] Paul Graham. Better bayesian filtering. In *Proceedings of the 2003 Spam Conference*, 2003.

[24] Anti-Phishing Working Group. Phishing archive. http://www.antiphishing.org/.

[25] Saul Hansell. Internet is losing ground in battle against spam. *New York Times*, April 2003.

[26] Evan Harris. The next step in the spam control war: Greylisting. http://www.greylisting.org/ articles/whitepaper.shtml, August 2003.

[27] Unspam Technologies Inc. Project honey pot. http://projecthoneypot.org/.

[28] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, 1991.

[29] Nadeem Jamali. *CyberOrgs: A Model for Resource Bounded Complex Agents*. PhD thesis, University of Illinois at Urbana-Champaign, 2004.

[30] Myeong-Wuk Jang. *The Actor Architecture Manual*. Department of Computer Science, University of Illinois at Urbana-Champaign, 2004.

[31] Gregg Keizer. Spam costs businesses worldwide $ 50 billion. *Information Week*, February 2005.

[32] John Kelnsin. Rfc 2821 - simple mail transfer protocol, April 2001.

[33] Robert E. Kraut, Shyam Sunder, Rahul Telang, and James Morris. Pricing electronic mail to solve the problem of spam. *Human-Computer Interaction*, 20:195–223, 2005.

[34] Benjamin J. Kuipers, Alex X. Liu, Aashin Gautam, and Mohamed G. Gouda. Zmail: Zero-sum free market control of spam. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW'05)*, 2005.

[35] Ben Laurie and Richard Clayton. Proof-of-work proves not to work. In *The Third Annual Workshop on Economics and Information Security*, May 2004.

[36] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.

[37] William Leibzon. Message enhancements for transmission authorization. http://www.metasignatures.org/.

[38] John R. Levine. Experience with greylisting. In *Second Conference on Email and Anti-Spam (CEAS) 2005 Proceedings*, Mountain View, CA, July 2005.

[39] Steven Levitt and Stephen Dubner. *Freakonomics: A Rogue Economist Explores the Hidden Side of Everything*. William Morrow, 2005.

[40] Kang Li, Calton Pu, and Mustaque Ahamad. Resisting spam delivery by tcp damping. In *First Conference on Email and Anti-Spam (CEAS) 2004 Proceedings*, Mountain View, CA, July 2004.

[41] Debin Liu and L Jean Camp. Proof-of-work can work. In *The Fifth Workshop on the Economics of Information Security (WEIS 2006)*, June 2006.

[42] Thede Loder, Marshall Van Alstyne, and Rick Wash. An economic answer to unsolicited communication. In *EC '04: Proceedings of the 5th ACM conference on Electronic commerce*, New York, NY, USA, 2004. ACM Press.

[43] Jeffrey K. MacKie-Mason and Hal R. Varian. Pricing the internet. pages 269–314, 1995.

[44] Microsoft. Sender id. http://www.microsoft.com /mscorp/safety/technologies/ senderid/default.mspx.

[45] Evangelos Moustakas, Chandrasekaran Ranganathan, and Penny Duquenoy. Combating spam through legislation: A comparative analysis of us and european approaches. In *Second Conference on Email and Anti-Spam (CEAS) 2005 Proceedings*, Mountain View, CA, July 2005.

[46] Scott Hazen Mueller. What is spam. http://spam.abuse.net/overview/whatisspam.shtml.

[47] Peter Nelson, Kenneth Dallmeyer, Lucasz Szybalski, Tom Palarz, and Michael Wieher. Spamalot: A toolkit for consuming spammers' resources. In *Third Conference on Email and Anti-Spam (CEAS) 2006 Proceedings*, Mountain View, CA, July 2006.

[48] BBC News. Hotmail limits to fight spam. http://news.bbc.co.uk/2/hi/technology/2890661.stm.

[49] Brian Prince. Report: Spamming soared in 2006. *eWeek*, December 2006.

[50] Sender Policy Framework project. Introduction to what spf is. http://www.openspf.org/.

[51] The Honeynet Project. The honeynet project. http://www.honeynet.org/.

[52] The Spamhaus Project. The definition of spam. http://www.spamhaus.org/definition.html.

[53] The Spamhaus Project. The spamhaus block list. http://www.spamhaus.org/sbl/index.lasso.

[54] Mehran Sahami, Susan Dumais, David Heckerman, and Eric Horvitz. A bayesian approach to filtering junk e-mail. In *In AAAI'98 Workshop on Learning for Text Categorization*, July 1998.

[55] Patrice Y. Simard, Richard Szeliski, Josh Benaloh, Julien Couvreur, and Iulian Calinov. Using character recognition and segmentation to tell computer from humans. In *ICDAR '03: Proceedings of the Seventh International Conference on Document Analysis and Recognition*, Washington, DC, USA, 2003. IEEE Computer Society.

[56] Tom Spring. Spam slayer: Slaying spam-spewing zombie pcs. *PC World*, June 2005.

[57] Brad Templeton. E-stamps. http://www.templetons.com/brad/spam/estamps.html.

[58] Luis von Ahn, Manuel Blum, Nicholas J. Hopper, and John Langford. The captcha project. http://www.captcha.net/.

[59] Luis von Ahn, Manuel Blum, and John Langford. Telling humans and computers apart automatically. *Commun. ACM*, 47(2):56–60, 2004.

[60] Webopedia. Spam. http://www.webopedia.com/ TERM/s/spam.html.

[61] Meng Wong and Wayne Schlitt. Rfc 4408 - sender policy framework (spf) for authorizing use of domains in e-mail, version 1, April 2006.

[62] Yahoo. Domainkeys: Proving and protecting email sender identity. http://antispam.yahoo.com/domainkeys.

[63] Zhenyu Zhong, Kun Huang, and Kang Li. Throttling outgoing spam for webmail services. In *Second Conference on Email and Anti-Spam (CEAS) 2005 Proceedings*, Mountain View, CA, July 2005.