Integrating Email Service with Opportunistic Network

Kunkun Wang
Abstract

Integrating Email Service with Opportunistic Network

Kunkun Wang

Haggle is a networking architecture for content-centric opportunistic communication. Legacy networks and legacy services are possible to be integrated into Haggle. A typical legacy service is email service. The key problem for integrating email service is that data is opportunistically transmitted to any node and any node which has Internet connection could be a potential email gateway. Therefore, one email may find many gateways in Haggle and duplicate emails will be sent to a destination. An email gateway can pick up email data from Haggle and forward this email through the Internet.

To solve the above problem, this thesis proposes a method that each email gateway sends a copy of an email to a public email account when it gets a new email, and then all the gateways can check the public email account to see whether the new email they get is a duplicate or not. In the implementation phase, two gateway programs which respectively use the POP3 protocol and the IMAP protocol are implemented for duplicate email checking. Experiments were conducted afterwards to compare and evaluate the performance and scalability of these two programs.
Acknowledgement

It was my pleasure to do my master thesis in the Communication Research Group, Department of Information Technology, at Uppsala University. I would like to thank all the people who have helped me in my pursuit of this master thesis. First of all I would like to recognize my supervisor Fredrik Bjurefors, and my reviewer Christian Rohner, who always gave me help and advice. Vasilij Savin and Tong Xiaoyu also gave me a lot of help. Erik Nordstrom gave me a lot of suggestions at the beginning of my project, and all the staff of Communication Research Group provided help and support. Without their help, I could not have completed my thesis alone. At last, I would like to thank my parents, my girlfriend and friends who are always with me.
## Contents

- Acknowledgement ................................................. 1
- Contents .................................................................. 3
- 1. Introduction ...................................................... 4
  - 1.1 Delay Tolerant Networking Architecture (DTN) .......... 4
  - 1.2 Opportunistic Networks ..................................... 4
    - 1.2.1 Concepts of Opportunistic Networks ............... 4
    - 1.2.2 Advantages and Usage of Opportunistic Network ...... 5
    - 1.2.3 Two Scenarios of Opportunistic Networks .......... 6
    - 1.2.4 Haggle Project ............................................. 6
  - 1.3 Project Background and Objective ....................... 7
  - 1.4 Outline of the Thesis ....................................... 7
- 2. Background ....................................................... 8
  - 2.1 Introduction to Haggle ..................................... 8
  - 2.2 Haggle Overview .......................................... 8
    - 2.2.1 Unified Metadata ....................................... 9
    - 2.2.2 The Relation Graph ................................... 10
    - 2.2.3 Search Based Resolution ............................ 10
    - 2.2.4 Forwarding ................................................. 12
  - 2.3 Related Work ................................................ 12
- 3. Email Service in Opportunistic Network .................. 13
  - 3.1 Problems to Be Solved .................................... 13
  - 3.2 Solution ......................................................... 14
    - 3.2.1 Integration into Haggle ............................... 14
    - 3.2.2 Email Sending .......................................... 15
  - 3.3 Implementation .............................................. 15
    - 3.3.1 General settings ....................................... 15
    - 3.3.2 Program Implementation .................................. 16
- 4. Experiment and Evaluation .................................. 17
  - 4.1 Email Format .................................................. 17
  - 4.2 Comparing POP3 with IMAP ............................. 19
  - 4.3 IMAP Performance on Multiple Email Senders .......... 21
  - 4.4 Scalability of IMAP Program for Multiple Gateways. ...... 22
- 5. Conclusions and Future work ............................... 25
  - 5.1 Conclusions and Discussion .............................. 25
  - 5.2 Future work ................................................... 25
- References .............................................................. 27
- Appendix ................................................................. 29
1. Introduction

1.1 Delay Tolerant Networking Architecture (DTN)

Nowadays, the Internet has become an indispensable part of people’s lives. Everyday billions of people use different methods such as cable, Wi-Fi, and so on to connect to internet everyday. Internet which is based on TCP/IP has made our life much more convenient. But, with the increasing needs of people and the development of different technologies, sometimes Internet could not fulfill people’s requirements.

TCP/IP could provide very good service if the following requirements are satisfied [3]:

- An end-to-end path exists between data source and destination;
- The Maximum round trip time (RTT) between any node pairs in a network should not be excessive;
- The packet loss during transmission should be small.

Unfortunately, a class of challenged networks, such as Terrestrial Mobile Networks, Exotic Media Networks, Military Ad-Hoc Networks, Sensor/Actuator Networks [6] and so on which violate one or more of these requirements, are becoming more and more important and may not be well served by the end-to-end TCP/IP model. All these classes of challenged networks have path and link characteristics which are High Latency, Low Data Rate, Network disconnection and Long Queuing Delay [3]. As the path and link characteristics of challenged networks have departed from the Internet, researchers proposed a challenged network oriented network architecture which is called delay tolerant networking architecture (DTN) [3].

DTN is based on message switching, and a “bundle layer” [3] is adopted for Message aggregates. The function of the “bundle layer” is similar to an Internet gateway, but the difference is that the “bundle layer” provides service for message switching instead of packet switching, so the “bundle layer” which in some sense is a router which is called “bundle forwarder” or DTN gateway.

1.2 Opportunistic Networks

1.2.1 Concepts of Opportunistic Networks

Opportunistic networks are developed from DTN, so it contains most of the features of DTN. In some sense we can consider opportunistic networks as self-organized networks with common features of DTN. There is not a clear definition of opportunistic networks, but the main characteristics of opportunistic networks are as follows:
Full link path is not required between source node and target node;
Communication is realized by the contact opportunity of nodes movement;
Network is self-organized by nodes, and no base-station is required..

Opportunistic routing between nodes is based on the “store-carry-forward” routing pattern. A typical opportunistic routing is shown in Figure 1-1:

- At time T1, the source node (S) wants to send data to the target node (T). However, they are not in the same network domain so that they are not able to communicate with each other directly. Therefore, S packs its data into a message and sends it to the next hop (node 1). Node 1 gets this message, stores it and then waits for an opportunity to forward it.
- After a period of time (at T2), when all nodes move to different places some nodes meet in a new network domain. Node 1 finds node 4, and then forwards the message that is from S.
- After another period of time (at T3), all nodes have move to new places. Node 4 meets node T in a new network domain, and then forwards message to T. T unpacks the message and gets data, and then the whole process of transmission is completed.

1.2.2 Advantages and Usage of Opportunistic Networks
According to the description above, an opportunistic network is not reliable if you want to send data to a specific node. Also it is impossible to maintain a stable connection between nodes. Although opportunistic networks are not reliable for data transmission, they are still very useful for many applications.

With the development of technology, wireless technologies such as Bluetooth, Wi-Fi, and so on are equipped in various mobile devices (mobile phone, PDA, mp3 player, etc.) which has increased the use of wireless ad hoc network applications. These applications are widely used for building sensor networks, data sharing, Internet collaborating etc. [4]. However, in a realistic environment, the traditional communication model of mobile ad hoc networks (MANET) which requires at least one path existing from the data source to target node is not able to build a structured and fully connected network. So for much of the time the whole network is disconnected for reasons which include signal attenuation, link loss, low density,
node movement. As a result, communication failure is increased. Opportunistic networks which do not require a full network connection can solve the problems for many application fields.

In an opportunistic network, communication opportunities arise from node movement to forward messages in a node-by-node way. The whole message forwarding process does not require a fully connected network. As the needs of self-organized networks increase so rapidly, many researchers have shown their interest in opportunistic networks. Although opportunistic networks are still in the developing step, some applications have already been set up. The following lists some typical applications of opportunistic networks [4]:

- Wild animal tracing;
- Handheld devices network organizing or mobile networking;
- Car networks;
- Network transmission in remote regions.

1.2.3 Two Scenarios of Opportunistic Networks

For instance, a destructive earthquake taking place in a city may completely destroy many buildings including base stations for any kind of network access. Thousands of people may be buried under ruins. In order to rescue those people who are buried, traditionally, we would have to spend a lot of time and efforts to search everywhere. And it happens again and again that people are found dead after taking them out of ruins because they were buried too long. If mobile phone supports opportunistic networks, people would be able to send out his information by using his mobile phone. Then no matter who receives their message, they can get a timely rescue to save their lives. A similar case exists in other disasters (flood, debris flow, fire in high building, etc.).

A second scenario is set on a train from Uppsala to Stockholm. I am planning to visit Drottingholm, but I do not know how to get there. I have a laptop in my hand, but I do not have a GPRS connection (I don’t want to pay for it, because that is so expensive.). The lucky thing is that my laptop supports one sort of opportunistic network, what I can do then is send a message about transportation from Stockholm Central Station to Drottingholm, and hope some guys could receive my message and provide me an answer. On the train, people around me are mostly Swedish. If any one of the Swedish guys receives my message is quite probably give me a very good answer.

1.2.4 Haggle Project

Nowadays, a mobile network user may have one or more handheld devices with big storage capacity. The user does random moves everyday, so we can regard him as a networking node with big data-carrying capacity. The devices have various local networking interfaces which allow him to exchange data with other nodes in the mobile network. Furthermore, he is able to
access one or more global networks (e.g. Internet, GPRS, CDMA), which differ in price, bandwidth, and availability. Both global and local networks may provide occasional transmission opportunities to transfer data. The user’s mobility allows him to carry data to other places, occasionally forward data to other node, and finally the data reach its destination. Researchers from Cambridge University and Intel Research call this model as Pocket Switching Network (PSN) [7]. Both MANET and DTN could not fulfill their requirements for PSN, so they designed a network architecture called Haggle which is an innovative paradigm for autonomic opportunistic communication. This study on Haggle then got funding support from European Commission. This project has seven European participant universities or institutions, one of which is Uppsala University [9]. The Haggle version designed by Uppsala University which is a novel search-based network architecture [5] is different to the above Haggle.

1.3 Project Background and Objective
In Haggle, both local networks and global networks are called legacy networks, and Haggle tries to make legacy services provided by the legacy networks available to mobile users. Email service is a very typical legacy service, and the key problem to integrate email service is that data is opportunistically transmitted to any nodes and any nodes which have internet connection could be a potential email forwarder so that one email may find many forwarders in Haggle and duplicate emails will be sent to destination. This thesis aimed to integrate email service into Haggle. The problems to be solved are listed as follows:

- Avoiding transmission of user’s email account name and password in Haggle for security reason.
- Avoiding sending duplicate email copies to same email destination.
- Controlling email sending cost in a tolerable scale.

1.4 Outline of the Thesis
The rest of this thesis is organized as follows:

Chapter 2 introduces concepts and working mechanism of Haggle. Chapter 3 proposes a solution to integrate legacy email service and provides its implementation. Chapter 4 provides the experiments results and the evaluation for the implementation. Chapter 5 gives some conclusions and discussions as well as some possible future works.
2. Background

2.1 Introduction to Haggle

Haggle is a networking architecture for content-centric opportunistic communication [16]. Here content-centric in other words can be data-centric or content oriented, which means that data is forwarded based on the interest of nodes in content instead of host addresses. In this case, Haggle nodes are allowed to exchange data directly between themselves whenever they meet in Haggle domain. As the main task of Haggle nodes is propagating data, applications in any mobile nodes try to take advantage of any connection opportunities in that case. Communication in Haggle is inherently asynchronous since connection opportunities arise unpredictably.

2.2 Haggle Overview

Su et al. presented the Haggle architecture as they designed in [1]. The researchers at Uppsala University in the same project share the basic goals and principles of the architecture, but completely redesigned this architecture based on lessons they learned from the previous work. They used a flat metadata namespace with searching which improves the flexibility and late-binding mechanism of the system.

Haggle provides a push-based data dissemination service [5] where each mobile user could declare his own interests in data (e.g. transmit picture, provide specific service.), and the data exchange takes place when data content matches the interest. The content to exchange is based on the interests that users declare.

A typical Haggle application is PhotoShare. As shown in Figure 2-1 (source: [5]), a user with a mobile phone on which there is camera could very easily share photos taken with its camera if PhotoShare is already installed in the phone. Any number of key words can be added to the photo’s metadata as the annotation of this photo. Then this photo could be published in the Haggle network and users who have interests in matching with this metadata will receive it. All the received photos will be automatically shown in the mobile phone window. In this figure, the mobile phone user has interests in photos with key words "Banana", "Orange" and "Apple", so he received three photos which respectively named "Banana", "Orange" and "Apple". Each time the user receives photos opportunistically because there might be more than one photo that matches his interests. Therefore, if he publishes same interests in another time, he might receive different photos. The other part of an interest is weight which denotes how much the user want the photo. The higher weight means the bigger wish that the user want the photo.
All the Haggle applications do not need to simultaneously run with Haggle network which means
that they may run intermittently and concurrently. Users can turn on or shut down applications at
any time when the Haggle network service is already running. When the user turn on an
application, default functions will register this application with Haggle. After that, data and
interests from the user will be registered in Haggle as well. Unless explicitly removed by
applications, registered data and interest will persistently exist in Haggle. Therefore,
dissemination work could continue when users put their mobile phone into pocket with the
application running.

2.2.1 Unified Metadata

Haggle uses a unified metadata format to do matching between different types of data. The
metadata format is different from traditional packet headers that enable addressing and
multiplexing/demultiplexing based on TCP/IP. The metadata itself defines the "address" and how
the data is propagated in a Haggle network demultiplexed to applications respectively determined
by the operations that do searching and filtering against interest. Relation graph [5] which is a
logical structure of information and is stored in a place called Haggle note’s data store is used for
searching in Haggle. In the Haggle relation graph, the format of the items is called data object
which is a tuple (metadata; data). The data part in this tuple is optional, but in most cases, a
normal file (e.g. a photo, a Text document, an email, or a small flash file) is put in the data. The
attribute can be directly extracted from data or manually given by user.

Figure 2-1 Example of photo share
2.2.2 The Relation Graph

Relation graph is maintained in every Haggle node. In Haggle, the relation between a pair of data objects is determined by their shared attributes. If a pair of data objects have relation, this would mean that they have at least one attribute in their metadata which is the same. The more same attributes exist, the stronger their relation is.

The users who are interested in specific data objects set up his own interests in his phone and put them in a special data objects. This type of data objects is called node description. Attributes of a node description are not about data, but instead are the combined application interests of this node. Figure 2-2 (source: [5]) describes how nodes are represented in a relation graph. Some data objects in a relation graph represent nodes, and node descriptions are logically mapping to a node space.

2.2.3 Search Based Resolution

The search based resolution [5] gives a rank system to wishful receiver nodes. A receiver node can obtain a high rank only relies on how well it matches to a specific item of data. Relation graph will be updated when specific event takes place (e.g. new data objects or interests are registered, new data objects are received). In these cases, search-based primitive which is called event driven resolution will be invoked. Figure 2-3 (source: [5]) illustrates how event driven resolution is invoked.

Figure 2-2 Relationships between data objects and nodes in a relation graph
In Figure 2-3 (a), two haggle nodes exchange their node descriptions when they meet. Their meet triggers \( r_d \) (resolve data objects) and then nodes push resolved and ranked data objects to their neighbour. In Figure 2-3 (b), a new data object is inserted by an application. This insert operation triggers \( r_n \) (resolve nodes) and only the match neighbour gets this data object.

Figure 2-4 (source: [5]) illustrates how wishful receiver nodes are ranked in a relation graph according to how well these nodes match a specific data object. In this example, a query intends to find a node with interests on a specific data object. Two nodes have shown their interest in this data object because a number of attributes are shared with this data object. One node has a higher rank than the other one.

In the vice versa case, the rank of data object depends on how much the nodes want to get this object. During the meet of two nodes, each node could query its relation graph and data is exchanged in the order of how much the receiver node is interested in this data.

In some cases, the number of received data objects could be very large if the attributes that are shared are too few. However, the query can be limited to only the top \( n \) ranked nodes by setting
minimal weights and ranks through changing weighting function. In this way, data exchanging is controlled, and in some sense a type of congestion control is achieved.

2.2.4 Forwarding

Basically, as mentioned above, a Haggle node exchange data objects according to interests. If a node does have interest in the data objects from its encountered nodes, data objects will be forwarded to it from the other node. We therefore refer to this type of forwarding as interest-based forwarding [5].

Sometimes, a node can't forward data objects because the other node has no interest in its data objects. In this case, interest-based forwarding is not sufficient. In this case, Haggle can use a forwarding algorithm to compute delegate forwarders who are not in the current interest communities. Delegate forwarders are not interested in the data, but they are delegated data objects from the node inside the interest communities. These data objects on behalf of their interest communities, and thus forwarding these data objects is called delegate forwarding [5].

In the logical node plane, metrics which can, for instance, use the number of encounters relate the nodes to each other. Forwarding algorithms may maintain their own metrics. Figure 2-5 (source: [5]) depicts how delegate forwarding works. In this figure, the local node does not have direct contact with the target receiver. If a co-located neighbour is found that it has a very large chance to encounter the target node according to the calculation of the delegate algorithm, data objects will then be delegated to this node.

2.3 Related Work

Previously, some work was done for integrating legacy networks and legacy applications into opportunistic networks. In [33], a general description was presented on how to integrate legacy networks and legacy applications in to Haggle. And in advance, the authors of [1] described how to integrate web applications especially email application into Haggle, and the corresponding email program for Haggle was written by Daniel Aldman at Uppsala University. In other words, it was possible to write email among Haggle Users, but not possible to write email from a Haggle user to an internet user. In [32], the authors introduced a novel mobile social networking middleware named MobiClique which was designed and implemented by them. Mobiclique allows mobile phone users to synchronize online social profiles in opportunistic networks. Its experiment on using it in Haggle environment was presented in [34]. The other social service integration in Haggle which was called Opportunistic Twitter developed by EPFL University was mentioned in [34] as well. In [34], the authors described their design, implementation and experiment. And the experiment showed that mobile users without internet connection were able to use Twitter over Haggle. Both Mobiclique and Opportunistic Twitter use Ad-hoc way for opportunistic contacts.
3. Email Service in Oppotunistic Network

As is mentioned in Chapter 1, this thesis aims to integrate Internet email service into Haggle. In other words, any node in a Haggle network should be able to send an email no matter it has Internet connection or not. If a node with Internet connection provides email service and publish its service as an interest (e.g. set attributes as email service, send email) into the Haggle network so that other nodes could send their email data objects to this node and then get email sent to its destination address.

Figure 3-1 illustrates how we integrate legacy applications/services and legacy network. There are many legacy applications, and here we use email to describe the solution. Application proxy is a proxy that is used to integrate legacy applications. All the data sent by legacy applications will be converted to data objects and then sent into Haggle network by Email proxy. Network Proxy is a proxy that is used to integrate legacy networks. All the data exchange between legacy networks and Haggle network will be handled by Network Proxy.

3.1 Problems to Be Solved

Before realizing this proxy, some problems are needed to be solved:

- All the data objects are in plain text format, and hence not encrypted so that it is not secure to send user name and password in the current Haggle network. Network Proxy should be able to send email to destination without getting user name and password from sender node;
- Because the contact between nodes in a Haggle network is unpredictable, the email data object exchanging may take place many times if the sender node contacts many other nodes providing email service. As a result, many duplicate copies of the email will be sent to the same target email address, which should be avoided.
- The cost of the Network Proxy for sending an email should be tolerable for the email service provider. First of all, the Network Proxy should avoid occupying too much memory; otherwise the usage of other functions might be influenced. Second, the sending time should be restricted to a tolerable interval.

![Figure 3-1 Integrating legacy network and legacy application](image)
3.2 Solution

For integrating email service, we named the Network Proxy as “Gateway” and the Application Proxy as “Email proxy” (Figure 3-2). An email client (e.g. Outlook, Thunderbird,) firstly sends email to the Email proxy of current node through legacy protocols, and then the Email proxy packs the email into a data object and then forward this data object in the Haggle network. When the data object reaches a node with a Gateway that provides email service, the Gateway takes and unpacks this data object and convert it into an email package. Then the Gateway sends this email to the destination in the Internet through legacy protocols.

3.2.1 Integration into Haggle

From Figure 3-2, it is easy to find that the Gateway, Email proxy and Email data object are the three key parts for integrating email service into Haggle. Following settings were set up to help integrating Gateway and Email proxy into Haggle, and disseminating email data objects in Haggle:

- Both Gateway and Email proxy were implemented as Haggle application, so both of them needed to find and get an application handle before they started. Once they got the handle, they were registered in Haggle.
- To help realizing data object exchange, a special attribute named “gateway” was added into each email data object’s metadata. When the Gateway program started, it would publish an interest named “gateway”. When Haggle found that the Gateway’s interest matched one data object’s attribute, this data object was sent to the Gateway program.
3.2.2 Email Sending

The idea to realize the email sending and solve problems mentioned above is down as shown in Figure 3-3:

- At first, the Gateway receives an email data object from Haggle network, and then unpacks this data object into an email package. Gateway then extracts receiver’s email address and email subject from the email package.

- Second, two public email addresses – “hagglepub1” and “hagglepub2” are used. The Gateway uses SMTP protocol [17] [18] [19] [20] [21] to log into hagglepub1 and sends a new mail in which the sender is hagglepub1 and the receiver is the email address extracted from the email package above. The extracted subject will be put in the new subject, and the new subject looks like: “FW: subject”. The content of this new email is the whole email we get from the email data object. In this case, hagglepub1 simply acts as an email forwarder.

- Hagglepub1 sends a copy of the email (only with a unique subject) to hagglepub2. This copy is used for avoiding duplicate copies of the same email from being sent. At each time before hagglepub1 forwards an email, the Gateway would use POP3 protocol [23] [24] [25] [26] or IMAP protocol to check for a copy of the email at the inbox of hagglepub2 to make sure that no duplicate copies are sent.

- Hagglepub1 only sends emails with subject is for two reasons:
  - We only need to check the subject instead of checking the whole email which would save a lot of time.
  - The POP3 protocol retrieves new emails from the email server, and hence no copies are left in the server. We would need to resend the emails we have checked to hagglepub2 again. If hagglepub2 keeps the whole email for all emails, the resending work would be very time consuming. Even in this case, the resending work is still time consuming with the increased email numbers at hagglepub2.

- A modified subject line is used to ensure that the email at hagglepub2 is unique. The subject consists of a UID, the sender’s email address and the original email subject. The UID which includes the data object create time [31] and a random number (generated by a function for 32-bit random number, normally has 9 digits or 10 digits) is created by the Email proxy when it is converting original email message to data object.

- IMAP protocol doesn’t care whether the email sent from hagglepub1 is subject only, because IMAP protocol supports an inner search function to find email with a specific subject. In this case, the performance of IMAP protocol should be more efficient than POP3 protocol.

- If the email checking result from hagglepub2 is that the email hasn’t been sent, the new email will be sent to destination from the hagglepub1.

3.3 Implementation

3.3.1 General settings
The programming language is C language, and Openssl [13] is used for SSL and TLS authentication in SMTP, POP3 and IMAP login.

At the beginning, the two public email addresses are Gmail email addresses which are hagglepub1@gmail.com and hagglepub2@gmail.com because Gmail email service is stable. To log into Gmail SMTP server, SSL or TLS [10] [11] [12] [14] [15] authentication is required, and the port for TLS/STARTTLS is 587. To log into Gmail POP3 server, SSL authentication is required, and the port for SSL is 995 [22]. Unfortunately, a lot of problems were encountered during experiments:

- Gmail has a number of sending limits in place to prevent abuse of their system, and to help fight spam. If one of the mail accounts reaches an abuse limit, the account will be temporarily unable to send mail. If Gmail was accessed via POP3 or IMAP clients (like Microsoft Outlook), it is possible to send an email message to a maximum of 100 email addresses at a time. In this case, Gmail can't be used as tens of thousands of emails will be sent during an experiment.
- Gmail allows keeping a copy in email box when accessing email via POP3 clients, but all the copies are marked as read already. And if an email is marked as read already, the POP3 client won't read this email again so that the duplicate emails can't be found in an experiment. Because email gateways might need to check whether an email has been sent for many times, but the Gmail email service can't satisfy the requirement, I have to write a email resending function in my program to send all retrieved email back to hagglepub2. Even in this case, it still has problem because the program can only check at most 100 emails as described in section 3.2.
- Although Gmail declares that it is using IMAP version 4rev1, its server can not identify some IMAP version 4rev1 commands [30].

In contrast, the UU email server does not have above problems:

- UU email server doesn't have sending limits.
- UU email server can keep email copy as unread which can reduce code lines and reduce program running time,
- UU email server can well support all the commands of IMAP version 4rev1.

At last I turn to Uppsala University (UU) email service.

### 3.3.2 Program Implementation

“Gateway” was implemented by program “smtp” which includes a POP3 program or an IMAP program. A lot of functions were both written in the POP3 program and IMAP program, and the details of 10 main functions can be found in the appendix part of this thesis.

“Email Proxy” was implemented by program “haggleproxy”(written by Daniel Aldman, a slight modification was done to the code used in my solution).
4. Experiment and Evaluation

In order to do experiment, a program called “smtp1” was written to periodically generate emails. In my experiments, emails were mainly generated every 1 second, 2 seconds, or 5 seconds. Program “smtp1” and “haggleproxy” were both installed on experiment nodes. Both POP3 program and IMAP program were installed at the gateway to check and forward email.

Three groups of experiments were designed to evaluate the performance of the system:

1. In order to see the difference of the performance at the gateway for the POP3 protocol and the IMAP protocol, 1 node and 1 gateway were used in the experiments.

2. In order to see the performance at the gateway for the IMAP protocol with different numbers of email senders and different email numbers at the email server, experiments were done both for 1 node and 1 gateway as well as 5 nodes and 1 gateway.

3. In order to see the scalability of IMAP program on finding duplicate emails, experiments were done for 1 node and 2 gateways.

In order to do evaluation, both POP3 program and IMAP program outputted a log file named “tracelog.txt”. Each line of the log file includes UID of an email, the time that the program received the email data object, the time that the program finished the whole process of handling an email and the status that showed the email was a new one or an old one.

4.1 Email Format

As was described in section 3.2, email receivers could definitely know that the email that he received was a forwarded email (Figure 4-1).

The email forwarder as well as the original email sender’s email address can be both found from the email content (Figure 4-2).

Figure 4-3 and Figure 4-4 show the email format that was sent to the hagglepub2 which was used to check emails. As was described in section 3.2, the first part of the email subject is the UID, the second part is the sender’s email address, and the third part is the original email subject. Moreover, the text content is empty.

Additionally, in order to distinguish different test nodes that generate emails and the email number they generate in the experiment, the node number and the email number is added to the subject. And my own UU email accounts were used to take the place of hagglepub1 and hagglepub2 in the experiment.
4.2 Comparing POP3 with IMAP

In this experiment, duplicate emails were added in and all the duplicate emails can be found by both programs, and node generates 1 email every 2 seconds. In the beginning of the experiment at two hagglepub2s have zero emails in the inbox and totally 50 emails are sent. As a result, no big difference of performance is found.

In order to show difference, the email numbers at the hagglepub2 is increased to 350. Figure 4-5 and Figure 4-6 both show the delay distribution of using IMAP and POP3, from which a very obvious difference can be found in performance between IMAP and POP3. Figure 4-5 depicts how fast the gateway can handle new emails, and Figure 4-6 depicts how fast the gateway can handle new email as well as duplicate emails. Obviously, duplicate emails can be handled much faster than new emails, and IMAP can handle all the emails much faster than POP3. The reason for these differences is that the POP3 program has to check duplicate emails by retrieving emails from the server and the IMAP program only needs to check duplicate emails by using inner search function of IMAP protocol.
Figure 4-5 Delay distribution for IMAP and POP3

Figure 4-6 Delay distribution for IMAP and POP3 with 5% duplicate emails
4.3 IMAP Performance on Multiple Email Senders

In order to evaluate the performance on multiple email senders, we used an experiment setup with 5 nodes and 1 gateway. Every node generated 1 email every 2 seconds, and 100 emails in total. Additionally, we did some other experiments by changing email generating rate to 1 second and 5 seconds, but we found no difference on performance at other generating rates compare to the rate at 2 seconds.

Totally 20 experiments were done. The number of emails in the inbox of hagglepub2 increased from 0 to 10000. In other word, before the first experiment, no emails existed in the inbox of hagglepub2, and after the first experiment was done, 500 emails existed in the inbox of hagglepub2. And during the following experiments, the number of emails in the inbox of hagglepub2 increased 500 after each experiment was done.

Figure 4-7 shows the delay distribution of the first experiment, the tenth experiment and the last experiment. From this figure, we could find that all the three curves have the same trend, and more than 95% of the emails can be handled in 1 second or less which is similar to the IMAP curve showed in Figure 4-6. The tenth experiment with 4500 emails already in hagglepub2 has the best performance; the first experiment with 0 emails in hagglepub2 performs worse than the tenth experiment; the last experiment with 9500 emails already in hagglepub2 performs worst. And from all the 20 experiment results, we can not find any trend of performance that is related to the number of emails in the inbox of hagglepub2. Therefore, we can conclude that the performance of IMAP program won’t be influenced by the numbers of senders, but we can’t conclude that the performance will be influenced by the number of emails in hagglepub2. Actually we found that all programs performance much better during night hours than working hours (9am. -5pm.), so we can guess that the performances mainly depend on the email server load.

![Figure 4-7 Delay distribution for IMAP with amount of emails in the inbox of hagglepub2](image-url)
4.4 Scalability of IMAP Program for Multiple Gateways.

In the first experiment, duplicate emails were sent after all the emails that had been sent, so the scenario was ideal for handling duplicate emails. Email server needs time to handle email request. If the email server hasn’t been able to send email to hagglepub2 when new email checking request come, new email will be regarded as new email even if it is an old email. In the real world, there won’t be only one gateway connecting to Internet in Haggle network, and if the second gateway receives the same email data object which has already been handled by the first gateway before the email server sends this email to a public email account, duplicate emails will be sent to the receiver. In this experiment, we want to know how the delay between two gateways that receive emails impacts the undiscovered duplicate email rates (Figure 4-8).

Two computers were used to do this experiment. The computer1 generated 1 email every 5 seconds as well as provided gateway service, and the computer2 only provided gateway service. The data transmission time span between computer1 and computer2 is about 0.28 second. A delay from 0.3 second to 1.5 second is set in the program at computer2. The experiments were done both during night hours and working hours and computer1 totally generated 500 emails during each experiment.

![Figure 4-8 Email check for multiple gateways](image)
Figure 4-9 illustrates the *duplicate email rates* during night hour experiments. In each experiment, totally 500 emails were sent to both gateway1 and gateway2. Gateway1 firstly checked and sent emails, so hagglepub2 received 500 new emails from gateway1. Gateway2 checked and sent emails after a preset delay, so ideally all the 500 emails sent by gateway2 were supposed to be discovered as duplicate emails after the gateway program checked emails at hagglepub2. If the preset delay was short enough, some duplicate emails couldn’t be discovered. Then we used the number of undiscovered duplicate emails to divide 500, and called the result duplicate email rate. We can observe that the duplicate email rate increases drastically when the delay changes from 0.5 second to 0.4 second, so we can conclude that most emails can be handled within 0.8 second. When the delay is 1.5 second, several duplicate emails still can not be avoided. Combining the previous experiments, we can conclude that if the time interval between the two gateways receives the same email data object is more than 2 seconds, duplicate emails can nearly be avoided. But we can not conclude the time interval that we can completely avoid duplicate email since the longest email handling time vary from less than 1 second to more than 4 seconds in different experiments.

Figure 4-10 depicts the duplicate email rates during working hour experiments. Since the server load varies from time to time, the results of these experiments are not reliable, but we can take them as a reference. From this figure, we could find that emails can be handled much slower than that of night hour.
Figure 4-10 Rates of undiscovered duplicate emails using different delays during working hours
5. Conclusions and Future work

5.1 Conclusions and Discussion

The gateway programs that we have presented in this thesis are found useful. From the experimental results and evaluation shown in the last chapter, the following conclusions can be proposed:

- The number of email senders and the email sending frequency of a sender do not influence the performance of gateway.
- The number of emails in the inbox of the hagglepub2 will influence the performance of the gateway if it is based on POP3, but won’t influence the performance of the gateway when the gateway is using IMAP program. Therefore it is better to implement the gateway using IMAP.
- Email server load is the main reason that influences the performance of the gateway when the gateway program is using IMAP.
- In the real world, the time interval between email arrivals at two gateways shouldn’t be too short otherwise duplicate email will be generated. The best time interval should be more than 2 seconds.

In order to control server load, private email servers should be built for Haggle before deploying gateway program in haggle. One email server may be too busy to handle emails. Multiple email servers should be built and gateway program can automatically select the most free server when the program starts.

All the duplicate email checks are done at the email server, so it is possible that sometimes the server load is too high to provide good service. To avoid this problem, an alternative way is that email server uses P2P method for email check. The email server could build a P2P network where every Haggle node is responsible to verify emails with the closest UID.

5.2 Future work

The email gateway program that we presented in this thesis is just one step that we integrate legacy network service into Haggle. There is still much work can be done in the future. Possible future work could be:

- Doing research on how to avoid the time interval less than 2 seconds between email arrivals at two gateways.
- Building private and multiple email servers and writing a program to help automatically select most free server.
- Both POP3 program and IMAP program are not able to carry email attachments, and this can be done in the future.
● Using P2P method for email Check.
References


[22] Gmail: Configuring other mail clients. http://mail.google.com/support/bin/answer.py?hl=en&answer=13287.
Appendix

Main functions of IMAP program:

- `void create_interest(haggle_handle_t hh)`. This function is used to create haggle interest to help this program finding email data objects.
- `static email email_create(struct dataobject *dObj)`. This function is used to convert email data objects to regular email format.
- `static int newDataObject(haggle_event_t *e, void *arg)`. This function will retrieve email data objects from Haggle network and check the validity of data objects. If a data object is an email data object, this function will call email_create and then check email at email server. New email will be sent to destination if it hasn’t been sent.
- `void base64(char *dbuf, char *buf128, int len)`. This function is used to encode email account username and password.
- `void send_line(SSL *ssl, char *cmd)`. This function is used to send commands to email server.
- `void recv_line(SSL *ssl)`. This function is used receive anything that from email server.
- `void sendemail1(char *email, char **subj, int len)`. This function is used to send new email to destination.
- `int sendemail(char *email, char *body)`. This function is used to send resending emails that has been checked.
- `int pop(char *subject)`. This function is used to check email at email server. It will retrieve all the mails from email server, and compare their subjects with the new email.
- `char *findsub(char *s1)`. This function is used to find email subject from the emails.
- `int open_socket(struct sockaddr *addr)`. This function is used to build network socket connection.
- `int main(int argc, char *argv[])`. The main function will firstly find Haggle, register this program, start haggle event loop, publish interest and call newDataObject function to retrieve data object, The program will keep running one start, and only terminate when main function receives Haggle terminate signal or force quit signal (Ctrl-C).

Main functions of POP3 program:

- `void create_interest(haggle_handle_t hh)`. This function is used to create haggle interest to help this program finding email data objects.
- `static email email_create(struct dataobject *dObj)`. This function is used to convert email data objects to regular email format.
- `static int newDataObject(haggle_event_t *e, void *arg)`. This function will retrieve email data objects from Haggle network and check the validity of data objects. If a data object is an email data object, this function will call email_create and then check email at email server. New email will be sent to destination if it hasn’t been sent.
- `void base64(char *dbuf, char *buf128, int len)`. This function is used to encode email account
username and password.

- void send_line(SSL *ssl, char *cmd). This function is used to send commands to email server.
- void recv_line(SSL *ssl). This function is used receive anything that from email server.
- void sendemail1(char *email, char **subj, int len). This function is used to send new email to destination.
- int sendemail(char *email, char *body). This function is used to send resending emails that has been checked.
- int Imapemail(char *subject). This function is used to search and check email subject directly at email server by using IMAP inner search function instead of retrieving emails from email server.
- char *findsub(char *s1). This function is used to find email subject from the emails.
- int open_socket(struct sockaddr *addr). This function is used to build network socket connection.
- int main(int argc, char *argv[]). The main function will firstly find Haggle, register this program, start haggle event loop, publish interest and call newDataObject function to retrieve data object. The program will keep running one start, and only terminate when main function receives Haggle terminate signal or force quit signal (Ctrl-C).