Proxy-Based Solutions to Facilitate Mobile Applications

PROXY-BASED SOLUTIONS TO FACILITATE MOBILE APPLICATIONS

By

KUNPENG ZHANG

_____

A Thesis Submitted to The Honors College

In Partial Fulfillment of the Bachelor's degree
With Honors in

Computer Science

THE UNIVERSITY OF ARIZONA

MAY 2017

Approved by:

_____

Dr. Beichuan Zhang
Department of Computer Science

Abstract

Mobility causes problems in computer networking. For example, when a host moves to a new network and obtains a new IP address, its ongoing TCP sessions will break because TCP uses source and destination addresses and ports to identify packets belonging to the same session. In the first part of this project, we design and implement a solution to mobility-caused problems in general. This solution is independent of specific applications. In the second part, we come to a specific problem which is downloading emails from phone to laptop on an airplane. Old applications always assume that they have an internet connectivity which can connect to a specific server on the cloud, but this assumption breaks because of more and more mobility. Without internet access, local data can still be shared by the local connectivity. When people get on the airplane, they face this problem. Therefore, we come up with a solution to help share emails between the phone and the laptop on an airplane.

1 Introduction

In old days, when an application builds the connection to a host, the application will be given a host name or IP address, because the host address is unique. Nowadays, however, mobility breaks IP uniqueness. This brings up two problems. One is existing network connection breaks due to IP address changes. For example, when a host moves to a new network and obtains a new IP address, its ongoing TCP sessions will break because TCP uses source and destination addresses (and ports) to identify packets belonging to the same session. The first part of this thesis explores solutions of supporting host mobility. We propose a proxy-based solution to handle this problem. The goal is to establish a session between the client and the server, so that after changing the server's IP address, the same connection session still works. This solution works with all applications and transport protocols, and does not require to change applications. Key concepts in our design include the use of UDP tunnels and a sequence of private IP addresses to uniquely identify a host. We also implement a simple and incrementally deployable change to DNS as a mechanism to learn the address.

The other problem is that devices may not have internet connectivity, but data existing in local needs to be shared. Old applications always assume that they have an internet connectivity which can connect to a specific server on the cloud, but this assumption breaks due to mobility. When people are flying in an airplane, they do not have internet access, but the phone and the laptop can still share data in the local network. In the second part of the project, we move on to an application specific host mobility problem, which is downloading emails from a smart phone to a laptop on an airplane. This can help the people who updated their emails on their smart phones before boarding edit emails on their laptops while flying. Our solution aims to make the smart phone as a temporary email host. There will be no changes to email client applications. After the phone gets reconnection to the email server, the proxy will be able to synchronize all the editions to the server.

2 Host Mobility

2.1 Design

We propose to use DNS to learn tunnel end-point addresses, use a generic user-level program on the client host to mask the non-uniqueness of private IP addresses from transport protocols and applications, and use UDP tunnels (instead of IP-in-IP tunnels) to encapsulate packets. We build a client proxy running on the client side and a server proxy running on the server side to handle this UDP tunnels. We first use an example to illustrate the tunneling mechanism.

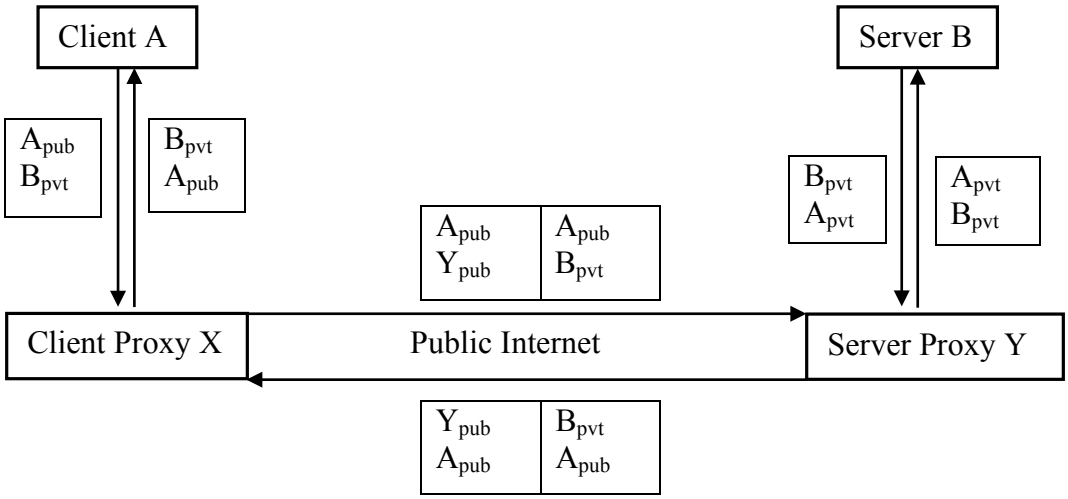


Figure.1

The DNS zone file needs to be modified, so that it includes the new resource record for server's private IP address. In Fig.1, let us assume that client A wants to access web server B. By querying B's DNS name, A learns B's 127.1.0.0/16 private IP address and Y's public IP address. A then sends packets to B's private IP address and the packets will be captured by client proxy X. X encapsulates its packets in UPD. The outer IP/UDP header is destined to server proxy Y. After Y received the packets, it removes the outer header and maps an unused 127.1.0.0/16 private address A_{pvt} to A_{pub}, then Y replaces the inner header source IP A_{pub} with A_{pvt} and sends the packets to server B.

In this whole framework, all the mobility is handled by the two proxies. When server B moves, only the public address of proxy Y is going to change. Server B will still hold its private IP address. DNS server will learn what is the new public IP address of proxy Y. From client A's point of view, after the DNS query, client A only connects to B's 127.1.0.0/16 private address, so the applications running on Client A are immune for any changes. Client proxy A learns what is proxy Y's new public IP address from DNS, then A will tunnel the traffic. Server B will not get interrupted, because all the packets from Client A will maintain the same source IP address. Therefore, this whole framework will keep the connection even if Server B moved.

2.2 Implementation

Two visual machines are used to simulate this framework. We name the client machine A, and the server machine B. The client-side proxy (cproxy) and the server-side proxy (sproxy) run on the client machine and the server machine. In order to relay the traffic to the proxy, TUN virtual network kernel devices [3] must be added to both machines. On the client side, TUN device has 127.1.0.2 as its IP address. We drive all the traffic to 127.1.0.0/16 via 127.1.0.2. On the server side, TUN device has 127.1.0.1 as the IP address, then we drive all the traffic to 127.1.0.0/16 via 127.1.0.1. In order to avoid the system treating 127.1.0.0/16 as loopback, we also need to change loopback netmask to 255.255.0.0. The full set-up scripts are following:

Client-side TUN setup script:

```
sudo ip tuntap add dev tun1 mode tun
sudo ip addr add 127.1.0.2/16 dev tun1
sudo ip link set tun1 up
sudo ip route change 127.1.0.0/16 via 127.1.0.2
sudo ifconfig lo netmask 255.255.0.0
```

Server-side TUN setup script:

```
sudo ip tuntap add dev tun1 mode tun
sudo ip addr add 127.1.0.1/16 dev tun1
sudo ip link set tun1 up
sudo ip route change 127.1.0.0/16 via 127.1.0.1
sudo ifconfig lo netmask 255.255.0.0
```

In both cproxy and sproxy, we need to connect to TUN devices. The following function will allocate TUN device [4]:

```
int tun_alloc(char *dev, int flags)
{
   struct ifreq ifr;
   int fd, err;
   char *clonedev = "/dev/net/tun";

   /* Arguments taken by the function:
    *
    * char *dev: the name of an interface (or '\0'). MUST have enough
    *   space to hold the interface name if '\0' is passed
    * int flags: interface flags (eg, IFF_TUN etc.)
    */

   /* open the clone device */
   if( (fd = open(clonedev, O_RDWR)) < 0 ) {
      return fd;
   }
```

```
   /* preparation of the struct ifr, of type "struct ifreq" */
   memset(&ifr, 0, sizeof(ifr));

   ifr.ifr_flags = flags;

   /* Flags: IFF_TUN   - TUN device (no Ethernet headers)
    *        IFF_TAP   - TAP device
    *
    *        IFF_NO_PI - Do not provide packet information
    *
    *  If flag IFF_NO_PI is not set each frame format is:
    *       Flags [2 bytes]
    *       Proto [2 bytes]
    *       Raw protocol(IP, IPv6, etc) frame.
    */


   if (*dev) {
      /* if a device name was specified, put it in the structure; otherwise,
       * the kernel will try to allocate the "next" device of the
       * specified type */
      strncpy(ifr.ifr_name, dev, IFNAMSIZ);
   }

   /* try to create the device */
   if( (err = ioctl(fd, TUNSETIFF, (void *) &ifr)) < 0 ) {
      close(fd);
      return err;
   }

   /* if the operation was successful, write back the name of the
    * interface to the variable "dev", so the caller can know
    * it. Note that the caller MUST reserve space in *dev (see calling
    * code below) */
   strcpy(dev, ifr.ifr_name);

   return fd;
}
```

In both cproxy and sproxy, we simply call “tun_alloc” to create a socket and build the connection between proxy and TUN device:

```
int tun_fd = tun_alloc(tun_name, IFF_TUN | IFF_NO_PI);
```

We also need to create a UDP socket and connect it to the remote proxy:

```
sock_fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
if(sock_fd < 0) die("creating socket");

if (bind(sock_fd, (struct sockaddr *) &localSA, sizeof(localSA)) != 0)
   die("bind()");

if(connect(sock_fd, (struct sockaddr *) &remoteSA, sizeof(remoteSA)) !=
0)
   die("connect()");
```

In our proxies, we use *select()* to handle the input from two sockets at the same time. Let us use the Fig.1 example. When cproxy receives packets from the *tun_fd* socket, it will replace the destination address in the header with server B's 127.1.0.0/16 private address, then cproxy will forward the packets to sproxy through UDP tunnel. After receiving the packets from UDP tunnel, sproxy will replace the source IP address with an unused 127.1.0.0/16 private address which is mapped to the client's public IP address. Before sproxy forwards packets to server B, it needs to re-calculate the checksum, because we changed the source IP address. If server B replies to the client, it will follow the same process, but the checksum recalculation will happen in cproxy. We use telnet to test our framework. In this case, Client A in Fig1 will be the telnet client. Server B in Fig1 will be the telnet daemon.

Source code for this implementation: https://github.com/zkpye9/HostMobility.git

2.3 Future Work

In this implementation, we face an issue with using 127.1.0.0/16 subnet. After sproxy modified the packets and sent them to server B, B doesn't have any responses. However, this framework works with any subnets other than 127.0.0.0/8. There are two possible reasons for this issue. First, we doubt that TUN device does not send the packets to server B. Second, server B itself doesn't have any responses to the incoming data. We think the former is more likely. In the future, in order to solve this, we can try the following:

- Debug the source code of TUN device and located the problem. Modify TUN device to make it support our requirements.
- Use other methods to derive traffic, such as firewall, instead of TUN/TAP device.

3 Share Emails in Local Network

3.1 Design

In this part, we focus on a specific problem which is downloading emails from a phone to a laptop on an airplane. We design a proxy-based solution to solve this problem.

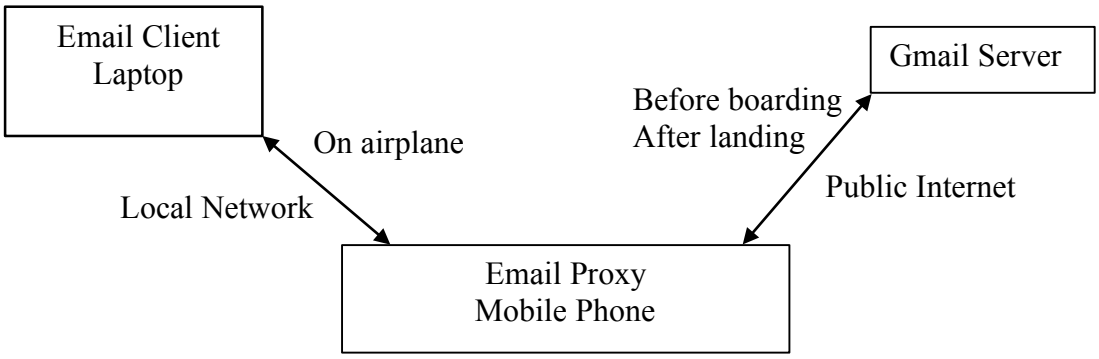


Figure.2

As shown in Fig.2, before boarding, the mobile phone connects to Gmail server and updates the emails. The email client on the laptop is configured to connect to the email proxy on the mobile phone. While flying, the email proxy on the mobile phone will become the email server for the email client on the laptop. The proxy will receive and reply all the IMAP commands that the client sends to it. All the email edits will be temporarily held in the proxy. Once the plane landed, the proxy will communicate with Gmail server and update all the edits.

3.2 Implementation

In order to test the whole framework, firstly, we plan to implement IMAP tunneling through the proxy. We build an IMAP proxy in Java, based on JavaMail API [2] and an open source email server called GreenMail [1].

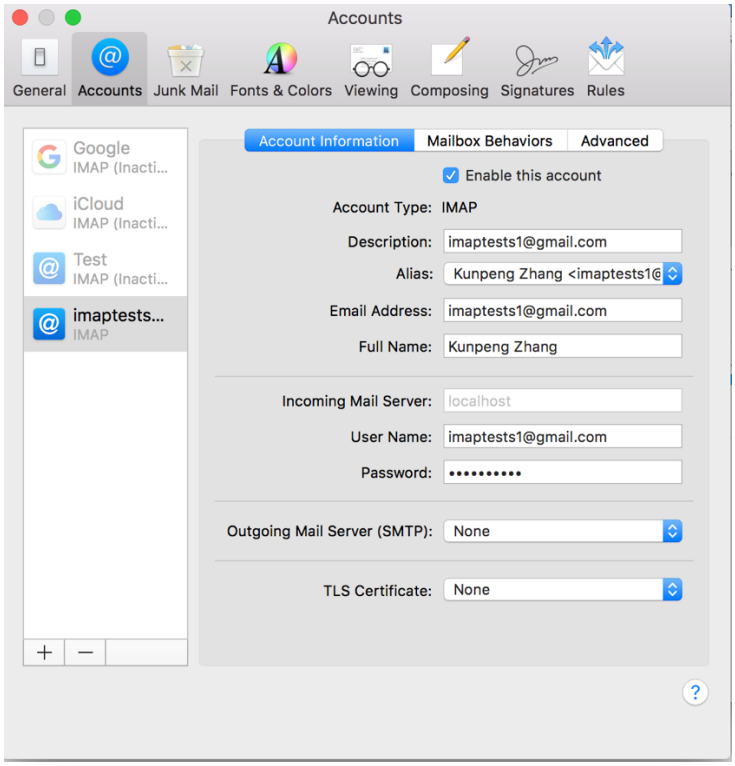
```
Properties props = new Properties();
props.setProperty("mail.store.protocol", "imaps");
Session session = Session.getInstance(props, null);
Store store = session.getStore();
store.connect("imap.gmail.com", "imaptests1@gmail.com", "helloworld");
Folder inbox = store.getFolder("INBOX");
inbox.open(Folder.READ_ONLY);
```

```
Message[] msg = inbox.getMessages();
```

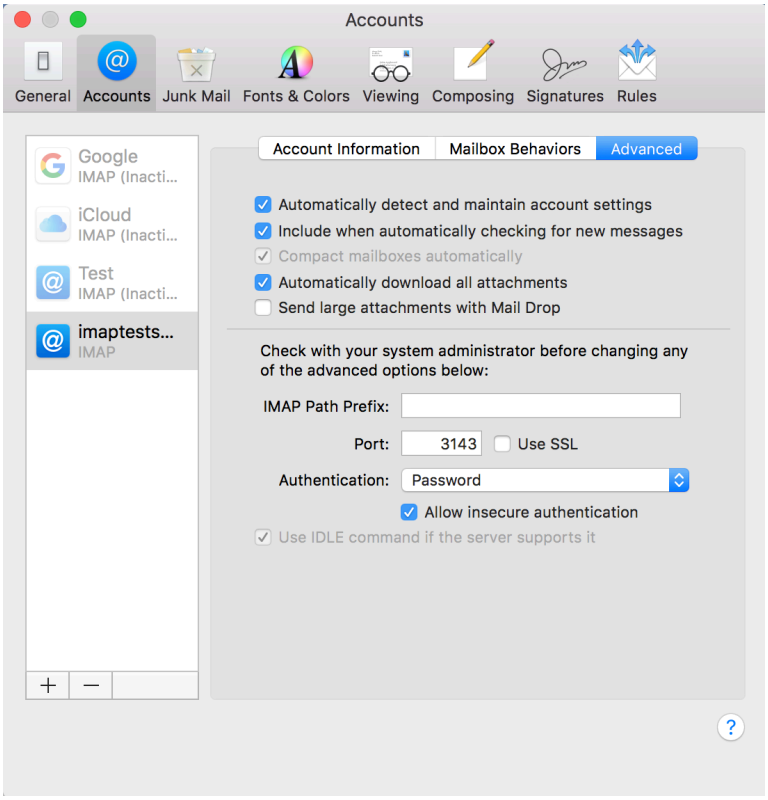
As shown above, in the proxy, we use JavaMail to build connections between the proxy and Gmail server. When the proxy runs, it needs to contact Gmail server to fetch and update all the emails. After getting an array of messages, we need to parse emails. JavaMail API provides methods for us to do so. For example, getting subject is `msg[i].getSubject()`. However, we need to be careful about parsing emails' contents. For some emails, the content is plain text, so we can simply use `getContent().toString()` to retrieve the text, but some emails warp the contents recursively in MimeMultipart objects. In order to handle this, we use a function for parsing email contents:

```
private static String getTextFromMimeMultipart(
        MimeMultipart mimeMultipart) throws Exception{
        String result = "";
        int count = mimeMultipart.getCount();
        for (int i = 0; i < count; i++) {
            BodyPart bodyPart = mimeMultipart.getBodyPart(i);
            if (bodyPart.isMimeType("text/plain")) {
                System.out.println("bodypart is plain text");
                result = result + "\n" + bodyPart.getContent();
                break;
            } else if (bodyPart.isMimeType("text/html")) {
                System.out.println("bodypart is html");
                String html = (String) bodyPart.getContent();
                result = result + "\n" + Jsoup.parse(html).text();
            } else if (bodyPart.getContent() instanceof MimeMultipart){
                System.out.println("body part is mimemultipart");
                result =result+
                        getTextFromMimeMultipart(
                        (MimeMultipart)bodyPart.getContent());
            }
        }
        return result;
    }
```

This function will recursively parse the email's content from wrapped MimeMultipart, until it finds plain text or HTML text. Then, we load those emails to GreenMail server. "`public static GreenMail gmail = new GreenMail()`" will initialize a static GreenMail server. " `gmail.start()` " will start the server. By default, the server will use port 3143 as IMAP service port. Next, we configure the email client to connect to the proxy.



Firstly, we need to configure the incoming mail server to the address of our proxy. In our case, the proxy runs on the same machine with the email client, so we set the incoming mail server to localhost.



Under the tag "Advance", IMAP port is set to 3143. After running the proxy, GreenMail server will handle the authentications command from the email client. After confirming the whole connection work, we start forking GreenMail. Originally, GreenMail as an email server will response every IMAP commands. Now, we change it to relay the commands to Gmail server. We choose the check command as an example. Every time GreenMail received a check commands, it will connect to the Gmail server to update the emails instead of checking emails locally.

Source code for this implementation: https://github.com/zkpye9/IMAP.git

3.3 Future Work

- For testing purpose, we only implement one IMAP command. The rest of IMAP commands need to be added in the features. GreenMail has already included all the IMAP commands. We need to modify it.

- We aim to immigrant the proxy program to mobile phones, then use local WIFI to build connections between phones and laptops. A new protocol may need to

be designed for the communication between the laptop and the phone in the local network. Since our proxy is developed in Java, firstly, we should try it on an Android smart phone. In the future, we need to explore the possibility of running it on iPhones.

- After the phone regains the internet connection, the synchronization to Gmail server is also a big part we should specifically handle in the future. The proxy runs on the phone should be added a function which detects whether the internet is reconnected. Once the internet connection is received, the proxy should automatically update the server. We believe JavaMail can help us build the connection to Gmail server and update the edits on the server.

4 Conclusion

Mobility brings challenges to the old network design. This proxy-based solution can generically solve these problems. In the first part, the design we purposed for the host mobility does not require any modifications to applications. It can help maintain the connection when the host moved. This can increase the host reliability. In the second part, our solution helps email data be shared among local networks on the airplane. It provides flexibilities to edit the emails saved in a local network.

Acknowledgements

I am grateful to Dr.Beichuan Zhang for his work on this research project and being my honors thesis advisor. We also thank Mr.Marcel May, the contributor of GreenMail, for providing help on building GreenMail.

References

[1] GreenMail: http://www.icegreen.com/greenmail/#

[2] JavaMail API: https://java.net/projects/javamail/pages/Home

[3] TUN/TAP: http://tuntaposx.sourceforge.net/

[4] TUN/TAP interface tutorial: http://backreference.org/2010/03/26/tuntap-interface-tutorial/