Bulk Historic IP2ORG2IP Attribution

Marwan Mostafa
University of Saskatchewan
Saskatoon, Saskatchewan Canada
mam024@usask.ca
11305332

Addison McAuley
University of Saskatchewan
Saskatoon, Saskatchewan, Canada
amm637@usask.ca
11295940

ABSTRACT

IP to Organization attribution plays an integral role in computer networks. Today, many services are available for public use to allow single current, bulk current, as well as single historic IP attribution. Services that allow for historic bulk attribution are less common, with Team Cymru and BTTF being few examples of a service of this kind. In this paper, we will introduce our own attempt at a historic bulk attribution service, IP2ORG2IP, and go over both highlights, and low-lights of our implementation.

1 INTRODUCTION

In computer networks, IP to organization attribution and organization to IP attribution play a key role in both network measurement, and cyber-security. There are many resources available for various means of IP attribution, including general whois[4] services provided by Regional Internet Registries (RIR)[7], Team Cymru's bulk attribution service[3], and the implementation proposed and completed in our chosen paper, 'Back to the Future' (BTTF) whois[5, 11]. RIR whois services are useful when attributing single, current IP addresses, however, these services are usually not capable of doing bulk attribution. At the time BTTF was published, Team Cymru's implementation had allowed bulk attribution, along with single historic IP attribution, but no historical, bulk attribution. This lead to the implementation of BTTF whois, which allows for any combinations of single or bulk, and current or historic IP attribution.

Our goal with this paper is to examine the methodology that was outlined in BTTF, and attempt to create our own implementation of a historic bulk attribution service. In addition to IP to Organization attribution, we attempted to implement an Organization to IP attribution service using a similar methodology to the IP to Organization attribution.

To summarize, this paper will go over the following:

- Introduce some general background on IP attribution services, and some of the related concepts
- Cover the methodology for our implementation of a historic, bulk attribution service
- Highlight some key results of our implementation, as well as limitations, and future work that could be done

2 BACKGROUND DISCUSSION

In this section, we will cover some of the services already available, as well as some general background information on some key concepts related to IP attribution.

2.1 Autonomous Systems (AS)

AS's can be most simply described as a system of IP prefixes that all operate under a common routing policy within said system. When viewing the Internet from a network layer perspective, the Internet can be described as a collection of AS's[12]. Further guidelines for AS creation and registration are highlighted in RFC 1930[6]. The importance piece to look at for this RFC is the criteria for considering an AS:

- AS's must be used for exchanging external routing information with other AS's
- AS's should place as many IP prefixes possible within a single AS
- AS's are only needed if your routing policy is different to the border gateway peers

In order for an AS to manage routing within the system and exchange information with other AS's, they need some way to identify themselves and their neighbours. This is done by using AS Numbers (ASN). These numbers get assigned to RIR's by the Internet Assigned Number Authority, with the RIR's further allocating the ASN's to the network operators in their RIR[8].

2.2 Regional Internet Registry whois

RIR's are the associations responsible for managing, distributing, and registering internet number resources[9]. This includes IP address space, along with ASN's. There are currently 5 RIR's, those being:

- African Network Coordination Centre (AFRINIC)
- Asia-Pacific Network Coordination Centre (APNIC)
- American Registry for Internet Numbers (ARIN) (includes USA, Canada, many Caribbean and North Atlantic Islands)
- Latin American and Caribbean Internet Addresses Registry (LACNIC)
- Reseaux IP Europeens Network Coordination Centre (RIPE NCC) (includes Europe, the Middle East, parts of Central Asia)

Each of these RIR's provide whois services for IP prefixes that are found within their region. These whois services are typically reserved for single, current IP attribution requests. Although in theory these services could be used to handle bulk current requests, the raw output produced makes this impractical[13], and other methods must be relied upon.

2.3 Team Cymru whois

To address the lack of bulk IP attribution services available for the general public, Team Cymru[3] had developed a service that allowed for bulk current attribution. At the time that BTTF was published, this only provided current attribution, but since then, they have added functionality for historic IP data. Although at this point in time, Team Cymru's service appears to have more functionality present, the information and methodology for this is not well documented, so we will not comment further on this service.

2.4 BTTF whois

The primary motivation behind our implementation was the BTTF whois service proposed and implemented by Fiebig et al.[5, 11]. At the time of publishing, Team Cymru had only had bulk current attribution, with no historical capabilities. As mentioned in 2.3, this has since changed, but due to the detailed methodology description of BTTF, we found it to be the most suitable service to build on. Along with this, Team Cymru's service utilizes data directly from RIR's, which with the time allotted for our project, was not feasible to request access for. Along with this, the act of actually compiling that data in an easy to work with format could be an entire project on it's own.

Instead of using data directly from RIR's, BTTF utilizes two datasets, the CAIDA RouteViews Aggregate Dataset[2], and the CAIDA Inferred AS to Organization Mapping Dataset[1].

2.4.1 CAIDA RouteViews Aggregate (prefix2asn). The CAIDA RouteViews Aggregate data maps IPv4 and IPv6 addresses to ASN's, starting from 2005-05-09 and 2007-01-01 respectively[2]. These mappings are derived from data collected by the University of Oregon's RouteViews Project[10]. Although we would likely be able obtain more accurate data if we went through the task of aggregating this data ourselves, we felt it would turn the project into a data aggregation project, as opposed to a network research project. Along with this, the aggregated CAIDA data has an easy to work with format, and the ASN information is formatted exactly as needed for the ASN2ORG dataset, leading us to go with the CAIDA aggregates.

The IPv4 dataset, which contains the bulk of the addresses, has a text file for each day since May 9th, 2005 to the present day. For each of these files, each line of text contains an IPv4 Address, the mask used for the network, and the ASN that advertised that IP prefix. The IPv6 dataset follows a similar format, however, the bulk of the data is from the IPv4 addresses, so this was our primary focus in the implementation.

2.4.2 CAIDA Inferred AS to Organization Mapping Dataset (asn2org). The CAIDA Inferred AS to Organization Mapping data maps AS's to organizations, utilizing whois data directly from the five RIR's. This data spans from April 2004 to the present day, with quarterly resolution[1]. This infrequent resolution can unfortunately lead to less accurate results, with some IP changes going unnoticed if they get changed within the quarterly resolution period, and some changes not being reflected until the next quarterly resolution. In order to simplify the implementation, along with avoiding downloading the entire dataset to our machine, we utilized the API that CAIDA provides for their ASN2ORG dataset. Another benefit of using the API is that it returns a formatted JSON response, allowing us to easily extract the data using our bash scripts, which will be described in 3.2.

3 IMPLEMENTATION METHODOLOGY

For our implementation methodology, we will walk through a step-by-step process of how we designed and eventually implemented our service. Starting with our Design Goals and Technologies used then we'll move on to the procedure we designed and how we implemented that procedure to come to the product we have today.

3.1 Design Goals

To start our design goals we made a list of mandatory features that were required for our product to be successful. Mandatory features consisted of:

- Attribution service for IPv4 addresses
- Attribution service for Organization names
- The service must provide a Historic lookup
- The service must be able to perform bulk requests

With our main goals for our service out of the way we listed some quality of life goals for the product:

- Client takes an input text file that holds all requests
- Clear output for client responses

3.2 Technologies Used

To implement this project we decided to use the following technologies, Our main language of choice is C to create the logic for our product. the libraries used were:

- standard C library
- standard C library for I/O and strings
- C library for sockets and network operations
- pthreads library for threads

To communicate with the APIs and get the required data we used bash scripts which consisted of the following commands/tools:

- curl
- grep
- echo
- gzip
- rm
- tr
- cat
- sed
- wget
- touch
- jq

3.3 General Procedure

The general procedure for our implementation will follow the client-server communication model, where our client will be fed the requested input, format a packet based on the protocol outline and the arguments given, and send it to the server, then wait for a response. The server will wait for the request from the client, once received the server will carry out the requested operation and return the results in a formatted packet following the protocol.

3.4 Protocol

Our client and server followed a specific protocol for communication which we will outline here. To start, all communication done was over a TCP connection and the packet sent over followed a specific structure. The structure was as follows, client packets were an array of 16 uint16_t. The first block is for request type, 0 for IP2ORG, and 1 for ORG2IP, the second block is for year, the third block is for month, and the fourth block is for day. The rest of the

12 blocks are for IP addresses or Organization names. The example client packet is as follows: (if it was ORG2IP, indexes 4 to 15 would hold individual chars of an Organization name)

- index 0: 0 (reqtype = IP2ORG)
- index 1: 2020 (year)
- index 2: 1 (month)
- index 3: 1 (day)
- index 4: 128 (IPv4 128.233.0.0/16)
- index 5: 233 (IPv4)
- index 6: 0 (IPv4)
- index 7: 0 (IPv4)
- index 8: 16 (prefix)
- index 9: 0
- index 10: 0
- index 11: 0
- index 12: 0
- index 13: 0
- index 14: 0
- index 15: 0

Our server packet is an array of 32 uint16_t which will contain the returned Org name or IP for the request.

3.5 Client Procedure

Now we will be going more in-depth on the client-specific procedure. The client starts by checking the arguments and storing them. The arguments that the client takes in are as follows:

- Machine name that server runs on
- Input file name that contains requests
- Port that the server is listening on

After storing the arguments the client establishes a connection with the server then enters a while loop. the while loop exits once all requests from the input file have been sent, in the loop the client reads a line from the input file and then formats a packet using the arguments and following the protocol. The client first converts the packet from host to network then sends the packet to the server and waits for a response, once the response is received it converts the packet from network to host then displays the response, and moves on to the next line in the file, it repeats this until the end of the file then exits.

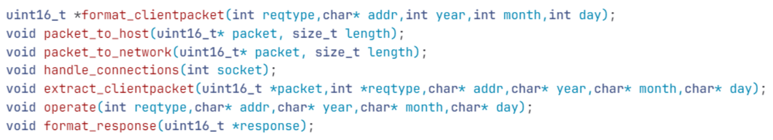
3.6 Input File

The structure of our client input file is as follows, each line is a request, and arguments are separated by white space. The arguments needed in order are request type (0 for IP2ORG and 1 for ORG2IP), address (IPv4 address with prefix or Organization name with no white spaces), year, month, day. Some example inputs could be:

- 0 128.233.0.0/16 2020 01 01
- 1 usask 2020 01 01

3.7 Server Procedure

The Server performs most of the work for the service, the procedure is as follows; the server has two components, a main thread and a connection thread. Server also takes one argument, the port that it should bind on.



```
uint16_t *format_clientpacket(int reqtype,char* addr,int year,int month,int day);
void packet_to_host(uint16_t* packet, size_t length);
void packet_to_network(uint16_t* packet, size_t length);
void handle_connections(int socket);
void extract_clientpacket(uint16_t *packet,int *reqtype,char* addr,char* year,char* month,char* day);
void operate(int reqtype,char* addr,char* year,char* month,char* day);
void format_response(uint16_t *response);
```

Figure 1: prog.h function prototypes

3.7.1 Main Thread. The main thread consists of the setup code that sets up the server to accept connections. The main thread enters an infinite while loop, constantly waiting for new connections. Once a connection is accepted, the main thread creates a new connection thread for that connection, which handles the requests for that connection.

3.7.2 Connection Thread. The connection thread enters a while loop that lasts until the connection is broken. The connection thread then waits until it receives a packet/request from the client. Once a packet is received it waits for the mutex, allowing only one connection thread to access the operate functionality at a time. Once the thread gets the mutex it begins working on its request. The thread first converts the packet from network to host, then it extracts the info from the packet. After extracting the info from the packet it operates and performs the requested service. The thread then formats a response packet, converts it from host to network, and sends the response back to the client. The thread repeats this for every request until the connection closes and the thread is killed.

3.8 Code Explanation

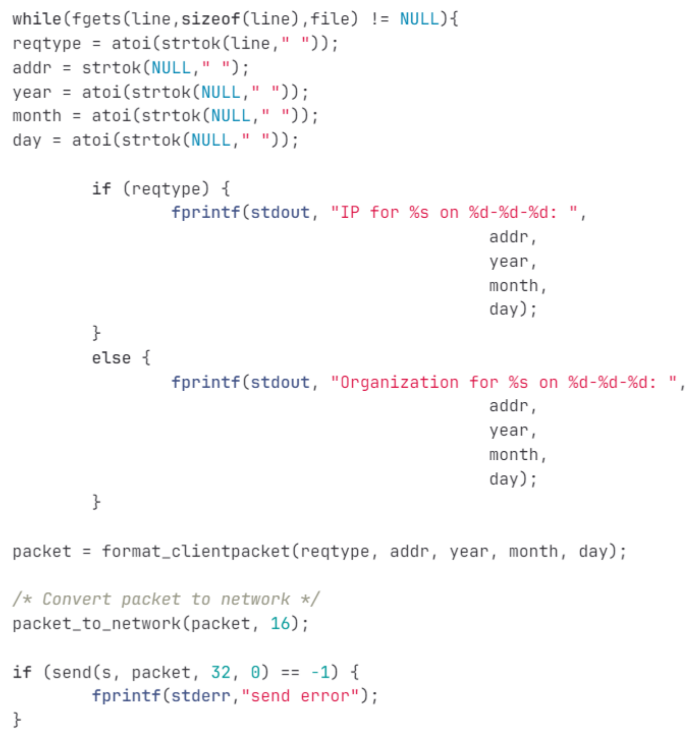
having a well-laid-out plan for implementation was crucial for our product to be well-implemented. Now that we have laid out our procedures we will explain our code and how each piece of code does its job

3.8.1 Code structure. How we laid out our code structure is as follows:

- prog.h contains all header files and function prototypes
- client.c contains all the client communication code
- format_packet.c contains all code for working with packets (formatting,extracting)
- packet_conversion contains code for converting code packets to network or host
- server.c contains the server code for setting up sockets and get the server ready to accept connections
- server_ops.c contains code for main server operations
- iterate.c contains code that works with the main API's to get the results
- ip_to_asn.sh bash script to download proper dataset table for date given
- asn_to_org.sh bash script to communicate with API to get org name from ASN
- org_to_asn bash script to communicate with API to get ASN of org

3.8.2 prog.h. prog.h contains all of our headers and function prototypes, the headers used are as follows:

- <stdlib.h>



```
while(fgets(line,sizeof(line),file) != NULL){
reqtype = atoi(strtok(line," "));
addr = strtok(NULL," ");
year = atoi(strtok(NULL," "));
month = atoi(strtok(NULL," "));
day = atoi(strtok(NULL," "));

        if (reqtype) {
                fprintf(stdout, "IP for %s on %d-%d-%d: ",
                                                addr,
                                                year,
                                                month,
                                                day);
        }
        else {
                fprintf(stdout, "Organization for %s on %d-%d-%d: ",
                                                addr,
                                                year,
                                                month,
                                                day);
        }

packet = format_clientpacket(reqtype, addr, year, month, day);

/* Convert packet to network */
packet_to_network(packet, 16);

if (send(s, packet, 32, 0) == -1) {
        fprintf(stderr,"send error");
}
```

Figure 2: client.c main loop

- <stdio.h>
- <string.h>
- <stdint.h>
- <sys/types.h>
- <sys/socket.h>
- <netdb.h>
- <arpa/inet.h>
- <netinet/in.h>
- <fcntl.h>
- <unistd.h>

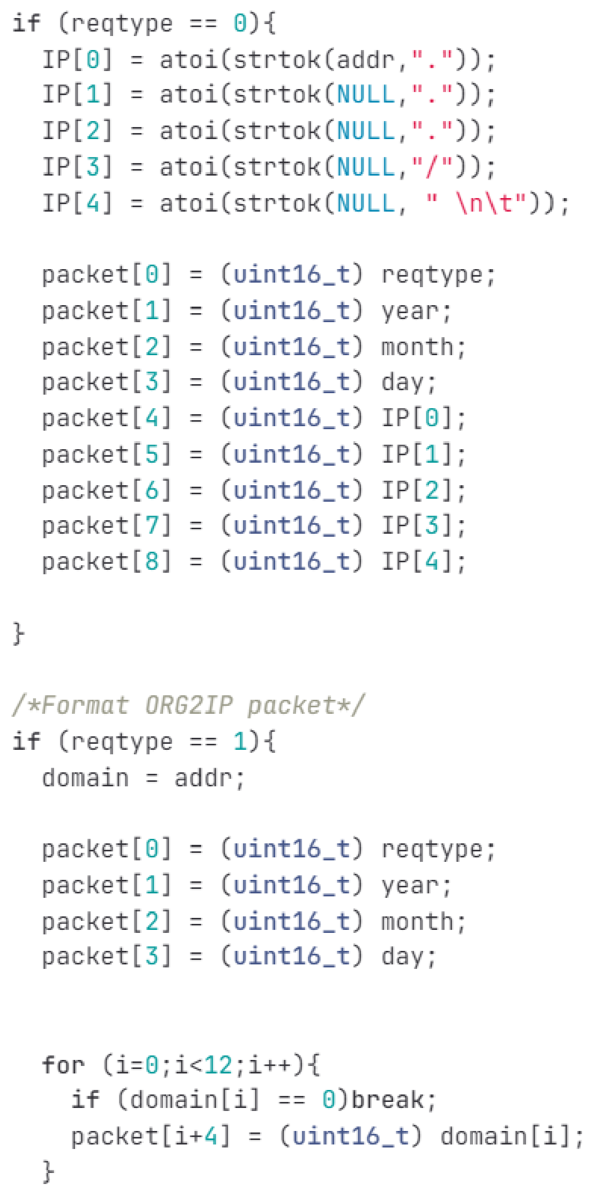
If you zoom in on Figure 1 you can view our function prototypes, format_clientpacket is found in format_packet.c and takes in a set of required arguments and returns a formatted packet following the protocol. packet_to_host takes in a pointer to a packet (array) and its size and converts the packet from network to host byte order. packet_to_network takes in a pointer to a packet (array) and its size and converts the packet from host to network. handle_connections takes the listening socket and handles all incoming connections on that socket. extract_clientpacket takes a packet and pointers to storage variables, it then extracts the info from the packet and stores it into the variables. operate is the main functionality of the server and will be explained later in further detail. format_response takes a pointer to a buffer and formats a server response packet.

3.8.3 client.c. client.c contains all the code for setting up a connection with the server, iterating through the input file, sending/receiving packets, and the main function for the client executable. client.c starts by checking the argument count and ensuring there is only 3 arguments being inputted. It then stores the arguments in the appropriate variables. client.c then sets up the socket required. The arguments required for socket() are found using gethostbyname_r and getaddrinfo. The client then connects to the server using the socket and moves on to its main functionality loop. If you look at Figure 2 you can see the code for the main loop, beforehand the input file is opened using fopen. Each line of the input file is iterated and the individual arguments of the line are stored in their respective variables. format_clientpacket is then called with the extracted arguments and a packet is created and stored into the packet variable. packet_to_network is then called on the packet and the packet is ready for sending. The packet is then sent to the server using the socket and send() and the client waits for a response from the server using recv() on the socket. Once the client receives a response it prints out the proper output based on the request type and moves on to the next line in the input file. The client repeats these steps until there are no more lines in the input file.

3.8.4 format_packet.c. format_packet.c deals with all the packet formatting and extraction. All code that deals with creating packets based on the protocol, and extracting info from packets is found here. format_packet.c contains the functions format_clientpacket, extract_clientpacket.c, and format response which are used by client.c or server_ops.c. Looking at Figure 3 we see the source code for format_clientpacket which takes in arguments and places them in the proper index following the protocol outlined in 3.4. It then returns the newly formatted packet. extract_clientpacket takes pointers to storage variables and goes through the packet, storing the respective arguments in the packet following the protocol to the storage variable. format_response takes a pointer to an array, gets the found IP/ORG name that was placed in an output.txt file by opening the file and reading the first line, (done by the operate function) then places each char into its own index of the provided response packet array.

3.8.5 packet_conversion.c. packet_conversion.c contains the codes for converting a packet from network to host and vice versa. Both functions inside (packet_to_network and packet_to_host) work the same way except the convert to different byte order. This is done by looping through the array and setting the packet indexes to themselves in the converted byte order.

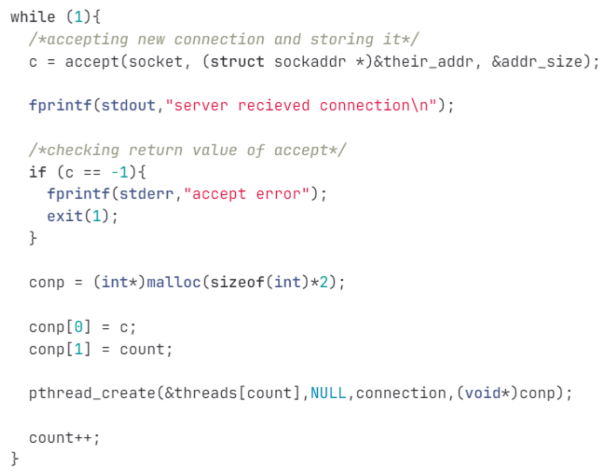
3.8.6 server.c. server.c contains the code for setting up the required sockets, binding them, listening for connections, and the main function for the server executable. server.c starts by making sure only one argument (port) is entered, then it creates a socket using socket(). The arguments required for socket() are found using gethostbyname_r and getaddrinfo. Once the socket is created it is bound to the entered port using bind() and it starts listening for connections by calling listen(). server.c the calls handle_connections on the created socket and hands off the main execution thread to that function for the rest of its run time.



```
if (reqtype == 0){
  IP[0] = atoi(strtok(addr,"."));
  IP[1] = atoi(strtok(NULL,"."));
  IP[2] = atoi(strtok(NULL,"."));
  IP[3] = atoi(strtok(NULL,"/"));
  IP[4] = atoi(strtok(NULL, " \n\t"));

  packet[0] = (uint16_t) reqtype;
  packet[1] = (uint16_t) year;
  packet[2] = (uint16_t) month;
  packet[3] = (uint16_t) day;
  packet[4] = (uint16_t) IP[0];
  packet[5] = (uint16_t) IP[1];
  packet[6] = (uint16_t) IP[2];
  packet[7] = (uint16_t) IP[3];
  packet[8] = (uint16_t) IP[4];

}

/*Format ORG2IP packet*/
if (reqtype == 1){
  domain = addr;

  packet[0] = (uint16_t) reqtype;
  packet[1] = (uint16_t) year;
  packet[2] = (uint16_t) month;
  packet[3] = (uint16_t) day;


  for (i=0;i<12;i++){
    if (domain[i] == 0)break;
    packet[i+4] = (uint16_t) domain[i];
  }
```

Figure 3: format_packet.c (format_clientpacket())

3.8.7 server_ops.c. server_ops.c contains all the main operation logic of the server. It contains two functions (handle_connections and connection). handle_connections is the main thread, its in a constant while loop waiting for connections on the passed-in socket. It waits for connections by calling accept() on the socket, which is a blocking function. Once a connection is received it creates a thread using pthread_create and passes the connection socket as the argument to the thread. This can all be seen in Figure 4. The next function is connection, which is the main logic that is run by any new connection thread. connection is in while loop for as long as the connection is alive. It takes in the connection file descriptor, then calls recv() on it which blocks until a packet



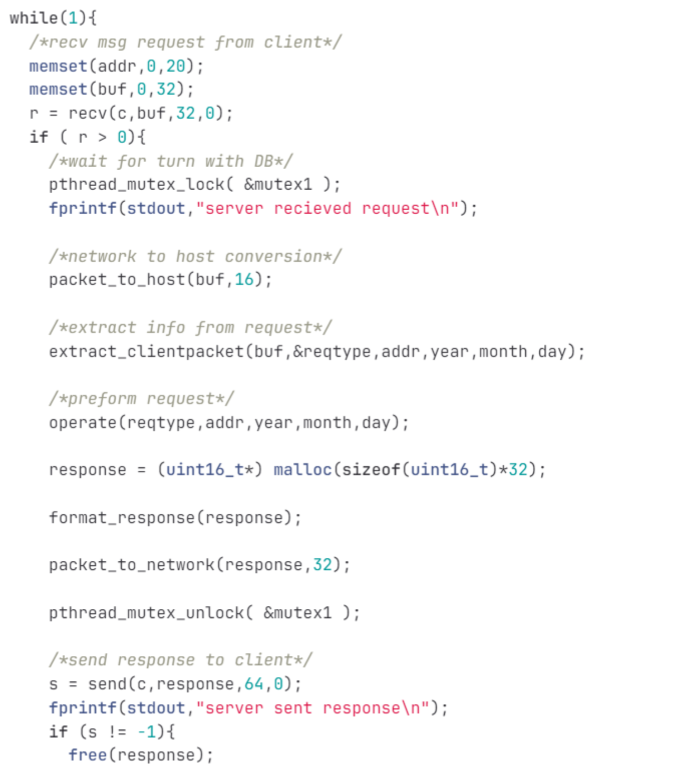
```
while (1){
  /*accepting new connection and storing it*/
  c = accept(socket, (struct sockaddr *)&their_addr, &addr_size);

  fprintf(stdout,"server recieved connection\n");

  /*checking return value of accept*/
  if (c == -1){
    fprintf(stderr,"accept error");
    exit(1);
  }

  conp = (int*)malloc(sizeof(int)*2);

  conp[0] = c;
  conp[1] = count;

  pthread_create(&threads[count],NULL,connection,(void*)conp);

  count++;
}
```

Figure 4: server_ops.c (handle_connections())

is received from the connected client. Once a packet is received it waits for the mutex by calling pthread_mutex_lock. When it is the threads turn with the mutex, it converts the packet from network to host byte order by calling packet_to_host, then extracts the info by calling extract_clientpacket. It then calls operate() on the extracted arguments. After the operation is complete, a response packet is allocated and formatted using format_response. Once complete the packet is converted to network byte order by calling packet_to_network. The mutex is left and the packet is sent back to the client. This is repeated until the connection is lost.

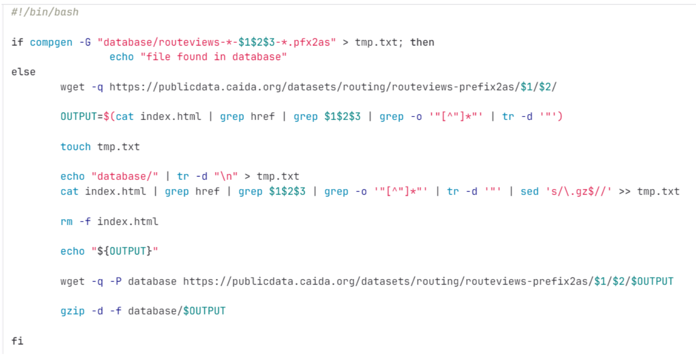
3.8.8 iterate.c. iterate.c contains the bulk of the servers operation, it contains the function operate, which when called, completes the IP2ORG or ORG2IP request given the required arguments. operate works by abusing bash scripts, first it checks the request type and based on the request it proceeds with different logic. For IP2ORG it starts by calling ip_to_asn.sh with the required args, then it opens the tmp.txt file created by the script. This file contains the file name of the main dataset that was just downloaded. Using that file name it opens the file using fopen(). With the file opened, the program iterates through the dataset searching for the IP address that was given in the request. Once the IP is found it stores the associated ASN. With this ASN it calls the next bash script, asn_to_org.sh, with the required args. For ORG2IP the process is similar. It starts by getting the ASN of the ORG by calling org_to_asn.sh with the required args, then the program opens tmp1.txt which contains the ASN that was found using the bash script. With the ASN stored ip_to_org.sh is called again to get the required database in the same way it was called for IP2ORG. The program iterates through the dataset and looks for the found ASN of the ORG, once found it stores the IP and prefix in output.txt.

3.8.9 ip_to_asn.sh. ip_to_asn.sh is a custom bash script we made to download the required dataset given the date (year month day). The script as seen in Figure 6 starts be checking if the requested dataset is already downloaded. If it is then it just stores its name in tmp.txt, if not then we move on. First wget is called on the API link given the year and month, this downloads the index.html which



```
while(1){
  /*recv msg request from client*/
  memset(addr,0,20);
  memset(buf,0,32);
  r = recv(c,buf,32,0);
  if ( r > 0){
    /*wait for turn with DB*/
    pthread_mutex_lock( &mutex1 );
    fprintf(stdout,"server recieved request\n");

    /*network to host conversion*/
    packet_to_host(buf,16);

    /*extract info from request*/
    extract_clientpacket(buf,&reqtype,addr,year,month,day);

    /*preform request*/
    operate(reqtype,addr,year,month,day);

    response = (uint16_t*) malloc(sizeof(uint16_t)*32);

    format_response(response);

    packet_to_network(response,32);

    pthread_mutex_unlock( &mutex1 );

    /*send response to client*/
    s = send(c,response,64,0);
    fprintf(stdout,"server sent response\n");
    if (s != -1){
      free(response);
```

Figure 5: server_ops.c (connection())

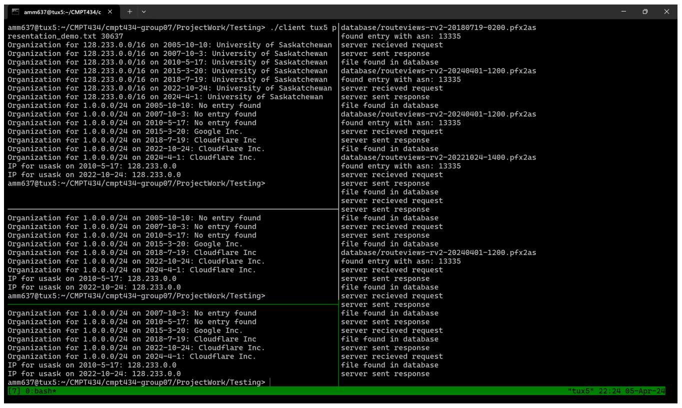


```
#!/bin/bash

if compgen -G "database/routeviews-*-$1$2$3-*.pfx2as" > tmp.txt; then
            echo "file found in database"
else
        wget -q https://publicdata.caida.org/datasets/routing/routeviews-prefix2as/$1/$2/

        OUTPUT=$(cat index.html | grep href | grep $1$2$3 | grep -o '"[^"]*"' | tr -d '"')

        touch tmp.txt

        echo "database/" | tr -d "\n" > tmp.txt
        cat index.html | grep href | grep $1$2$3 | grep -o '"[^"]*"' | tr -d '"' | sed 's/\.gz$//' >> tmp.txt

        rm -f index.html

        echo "${OUTPUT}"

        wget -q -P database https://publicdata.caida.org/datasets/routing/routeviews-prefix2as/$1/$2/$OUTPUT

        gzip -d -f database/$OUTPUT

fi
```

Figure 6: ip_to_asn.sh

contains the file names of all the datasets for that month. Using a long one liner consisting of greping, piping, and others, we get the file name of the dataset we want and store it into tmp.txt. We then append that filename to the previously used wget command to get the dataset we want, then call gzip to unzip the set. This script gets us the needed dataset and the name of the dataset file so that the C program can open the file.

3.8.10 asn_to_org.sh. asn_to_org.sh is a script that consists of a really long one liner that gets the ORG name of the ASN using the API and a bunch of piping to get the ORG name out of the



```
amm637@tux5:~/CMPT434/cmpt434-group07/ProjectWork/Testing> ./client tux5 p
resentation_demo.txt 30637
Organization for 128.233.0.0/16 on 2005-10-10: University of Saskatchewan
Organization for 128.233.0.0/16 on 2007-10-3: University of Saskatchewan
Organization for 128.233.0.0/16 on 2010-5-17: University of Saskatchewan
Organization for 128.233.0.0/16 on 2015-3-20: University of Saskatchewan
Organization for 128.233.0.0/16 on 2018-7-19: University of Saskatchewan
Organization for 128.233.0.0/16 on 2022-10-24: University of Saskatchewan
Organization for 128.233.0.0/16 on 2024-4-1: University of Saskatchewan
Organization for 1.0.0.0/24 on 2005-10-10: No entry found
Organization for 1.0.0.0/24 on 2007-10-3: No entry found
Organization for 1.0.0.0/24 on 2010-5-17: No entry found
Organization for 1.0.0.0/24 on 2015-3-20: Google Inc.
Organization for 1.0.0.0/24 on 2018-7-19: Cloudflare Inc
Organization for 1.0.0.0/24 on 2022-10-24: Cloudflare Inc.
Organization for 1.0.0.0/24 on 2024-4-1: Cloudflare Inc.
IP for usask on 2010-5-17: 128.233.0.0
IP for usask on 2022-10-24: 128.233.0.0
amm637@tux5:~/CMPT434/cmpt434-group07/ProjectWork/Testing>

Organization for 1.0.0.0/24 on 2005-10-10: No entry found
Organization for 1.0.0.0/24 on 2007-10-3: No entry found
Organization for 1.0.0.0/24 on 2010-5-17: No entry found
Organization for 1.0.0.0/24 on 2015-3-20: Google Inc.
Organization for 1.0.0.0/24 on 2018-7-19: Cloudflare Inc
Organization for 1.0.0.0/24 on 2022-10-24: Cloudflare Inc.
Organization for 1.0.0.0/24 on 2024-4-1: Cloudflare Inc.
IP for usask on 2010-5-17: 128.233.0.0
IP for usask on 2022-10-24: 128.233.0.0
amm637@tux5:~/CMPT434/cmpt434-group07/ProjectWork/Testing>

Organization for 1.0.0.0/24 on 2007-10-3: No entry found
Organization for 1.0.0.0/24 on 2010-5-17: No entry found
Organization for 1.0.0.0/24 on 2015-3-20: Google Inc.
Organization for 1.0.0.0/24 on 2018-7-19: Cloudflare Inc
Organization for 1.0.0.0/24 on 2022-10-24: Cloudflare Inc.
Organization for 1.0.0.0/24 on 2024-4-1: Cloudflare Inc.
IP for usask on 2010-5-17: 128.233.0.0
IP for usask on 2022-10-24: 128.233.0.0
amm637@tux5:~/CMPT434/cmpt434-group07/ProjectWork/Testing>
[7] 0:bash*                                                "tux5" 22:24 05-Apr-24
```

```
database/routeviews-rv2-20180719-0200.pfx2as
found entry with asn: 13335
server recieved request
server sent response
file found in database
database/routeviews-rv2-20240401-1200.pfx2as
found entry with asn: 13335
server recieved request
server sent response
file found in database
database/routeviews-rv2-20240401-1200.pfx2as
found entry with asn: 13335
server recieved request
server sent response
file found in database
database/routeviews-rv2-20221024-1400.pfx2as
found entry with asn: 13335
server recieved request
server sent response
file found in database
server recieved request
server sent response
file found in database
server recieved request
server sent response
file found in database
database/routeviews-rv2-20240401-1200.pfx2as
found entry with asn: 13335
server recieved request
server sent response
file found in database
server recieved request
server sent response
file found in database
server sent response
server recieved request
file found in database
server sent response
server recieved request
file found in database
server sent response
```

Figure 7: Example Server and Client Output

JSON data and into the format we want. the ORG name is stored in output.txt

3.8.11 org_to_asn.sh. org_to_asn.sh is a script that consist of a really long one liner that gets the asn name of the ORG using the same API as asn_to_org.sh and a bunch of piping to get the asn out of the json data and into the format we want, the asn is stored in tmp1.txt

4 IMPLEMENTATION RESULTS

To wrap up the report, we will cover some of the results of our implementation. This includes some of our testing results, the limitations of the current version of our implementation, potential areas where we could optimize the implementation.

4.1 Testing

This section will be fairly brief, as substantial testing against pre-established resolution services was not completed in the allotted time for this project. From more general qualitative testing, it is clear that our implementation does not get overly close in matching the performance of either the Team Cymru or BTTF implementation. From our understanding this primarily stems from our lack of a local database, which will be further covered in 4.2.1. In order to showcase some sort of results for the program, Figure 7 highlights an example output of our server and 3 clients using the service, primarily for IP to Org attribution.

4.2 Limitations

From the above testing, although our implementation works decently well, there are still some glaring limitations preventing our implementation from matching the quality of the implementations researched. This was expected due to various reasons, specifically the man power available, the time available to work through the project, as well as general competency levels that we lack due to inexperience. With each of our limitations, we will also highlight some potential solutions that could be used if were to continue on with the implementation.

4.2.1 Lack of Local Database. In terms of the impact on performance of the implementation, the lack of a local database was the biggest limitation seen with our implementation. With the sheer amount of data present throughout the datasets we used, we felt creating a local data structure to handle all this data was not feasible. Developing efficient data structures in order to maximize the querying speed of an attribution service could be it's own project in itself, and we felt worrying about that would prevent us from exploring our main goal of understanding how IP attribution services generally work.

The solution to this is pretty obvious, create a local data structure that holds all the information present in the dataset. This is definitely easier said than done, but with enough time, could be done. The data structure utilized in the BTTF paper was an IP Trie, which is a tree based structure used to store strings alphabetically, and allow efficient retrieval. The Trie BTTF used was in python, but with reasonable time, a C implementation of this Trie could be done.

4.2.2 *Dataset Timelines.* A limitation that impacts the accuracy of our program is simply the timelines of the available data. This includes the infrequent resolutions of the ASN2ORG datasets, as well as the inconsistent start dates of our two datasets. Mentioned briefly in 2.4.2, as the ASN2ORG dataset only adds entries on a quarterly basis, some changes will not be present in the dataset. The first scenario is an ASN changing organization information within a quarterly period, then changing again before the resolution occurs. Although this could happen, it is not overly likely. A more likely scenario that would be seen is an ASN changing organization information, and this change not being updated before the next quarterly resolution.

In terms of the inconsistent start dates of our datasets, this simply just forces our earliest date of data available to be the latest date, which in this case, would be May 5, 2005. This does limit the scope of the implementation, however, only missing out on an additional year of data is not the end of the world.

To get around this, we would have to utilize data from more sources, as well as actually aggregating all the data ourselves. Although it would be difficult to get access to all of this historic data, if this implementation was done on a higher scale, doing all the aggregation, as well as using data directly from RIR's would give us access to data with more frequent resolutions, and a longer history.

4.2.3 *No IPv6 Support.* Another hole in the implementation was the lack of IPv6 support. As the vast majority of the IP addresses assigned to ASN's are IPv4, which is seen through the file sizes (3.3 MB for IPv4, compared to 719 kB for IPv6) we made our main focus for this service IPv4 address attribution. With this being our main focus, we unfortunately reached a point where the lack of time prevented us from pursuing the IPv6 attribution.

The solution for this problem would be fairly trivial at this point in our implementation. With what we have learned with the IPv4 attribution, we are confident that adding IPv6 support could be done with few complications, and would be entirely feasible with additional time.

4.2.4 *Organization to IP Attribution.* The most disappointing limitation that we found in the implementation was unfortunately the limited capability of the Organization to IP attribution. The ORG to IP service works when an ASN only has a singular advertised IP prefix, however, this is rarely the case. When beginning the implementation plan, we gave ourselves a false impression that going from ORG to IP would simply be a reversed version of going from IP to ORG. When making our plan, we let the obvious fact that AS's, particularly the large ones, advertise large amounts of IP prefixes. As a result, the datasets that we were using were not suitable for an ORG to IP resolution service. Although we were still able to implement a solid IP to Org resolution, we would have liked to see this service come to fruition, but we still believe the knowledge gained from this failure was beneficial.

When thinking about this limitation, we came up with two possible solutions. The first would be to simply return the entire range of IP addresses that the requested ASN has advertised. Acquiring the IP addresses would be trivial, as we would simply need to iterate through all entries in the prefix2asn data, however, actually sending all this data back to the client would come with it's own complications. The second possible solution would be the solution presented in 4.2.2, utilizing more data sources.

4.3 Optimizations

Outside of large solutions to our bigger limitations, we also thought of some smaller optimizations we could include. These are as follows:

4.3.1 *Caching Database Tables.* To avoid time wasted retrieving the database tables from the prefix2as dataset, we could set up a simple caching system on our server. This would use least recently used cache replacement to keep frequently used database tables stored locally on our server.

4.3.2 *Binary Search.* To help increase the time complexity of our IP prefix searching, we could implement a binary search on the file. This could reduce the time complexity of our prefix search from O(n) to O(log(n)).

4.3.3 *Packet Space Optimization.* In order to save space in our packets, and allow for more complex messages to be returned to the client, we could write characters to the high and low bits of our packet entries. As packets are stored as an array of 16-bit integers, we could very easily write the characters in this manner to halve the number of array entries used for characters.

4.4 Current issues

The main current issue with our implementation we would like to highlight is the following; during a whitebox code review a Remote Code Execution (RCE) vulnerability was found. This allows malicious users to run their own bash commands on the server. This happens when a users does an ORG2IP request and the ORG name they put is a piped bash command such as "|ping tux7". This happens due to how we call the org_to_asn.sh script. Tt takes one argument, the ORG name, which is placed into the command string using sprintf("org_to_asn %s"). Ff the previously stated example is used this results in the command org_to_asn |ping tux7 to be run on the server, Which essentially means an attacker can run whatever commands they want on the server.

5 CONCLUSION

This paper has introduced our version of a historic bulk IP attribution service. Using the methodologies outlined in our chosen paper by BTTF [5, 11], we attempted to create a similar implementation, while including ORG to IP functionality. Although our implementation was not as high quality as we had initially hoped, the knowledge learned throughout the development of our implementation has more than made up for the shortcomings.

If we were to move forward with this implementation further, we would like to address a few of the limitations from 4.2, in particular, 4.2.1 and 4.2.4. Having a better local data structure would considerably improve the performance of our program, and would be essential for a service like this to actually be useful to the public. Implementing the ORG to IP is also a high priority as it would help expand on the use cases for our service, and having ORG to IP would be necessary for this implementation to be comparable in quality to some of the existing solutions.

To wrap things up, we believe the lessons learned throughout this project outweigh some of the limitations of our implementation and hope that the knowledge acquired will be beneficial going forward in our future careers.

REFERENCES

[1] CAIDA. 2019. *Inferred AS to Organization Mapping Dataset.* Retrieved April 4, 2024 from https://www.caida.org/catalog/datasets/as-organizations/

[2] CAIDA. 2019. *RouteViews Prefix to AS Mappings Dataset for IPv4 and IPv6.* Retrieved April 4, 2024 from https://www.caida.org/catalog/datasets/routeviews-prefix2as/

[3] Team Cymru. [n. d.]. *IP to ASN Mapping Service.* Retrieved April 4, 2024 from https://team-cymru.com/community-services/ip-asn-mapping/

[4] L. Daigle. 2004. *WHOIS Protocol Specification. RFC 3912. IETF.* Retrieved April 4, 2024 from https://tools.ietf.org/rfc/rfc3912.txt

[5] Tobias Fiebig. 2024. *bttf-whois.as59645.* https://bttf-whois.as59645.net/

[6] J. Hawkinson and T. Bates. 1996. *Guidelines for creation, selection, and registration of an Autonomous System (AS). RFC 1930. IETF.* Retrieved April 4, 2024 from https://datatracker.ietf.org/doc/html/rfc1930

[7] R. Housley, J. Curran, G. Huston, and D. Conrad. 2013. *The Internet Numbers Registry System. RFC 7020. IETF.* Retrieved April 4, 2024 from https://tools.ietf.org/rfc/rfc7020.txt

[8] Internet Assigned Numbers Authority (IANA). 2024. *Autonomous System (AS) Numbers.* Retrieved April 4 2024 from https://www.iana.org/assignments/as-numbers/as-numbers.xhtml

[9] Number Resource Organization (NRO). 2024. *Regional Internet Registries.* Retrieved April 4, 2024 from https://www.nro.net/about/rirs/

[10] RouteViews. 2024. *University of Oregon RouteViews Project.* Retrieved April 4, 2024 from https://www.routeviews.org/routeviews/

[11] Florian Streibelt, Martina Lindorfer, Seda Gurses, Carlos H. Ganan, and Tobias Fiebig. 2023. Back-to-the-Future whois: An IP Address Attribution Service for Working with Historic Datasets. In *2023 International Conference on Passive and Active Network Measurement (PAM 2023).* Springer International Publishing, 209–226. https://doi.org/10.1007/978-3-031-28486-1_10 https://link.springer.com/chapter/10.1007/978-3-031-28486-1_10

[12] Andrew Tanenbaum and David Wetherall. 2010. *Computer Networks (5th Edition).* Pearson.

[13] L. Zhou, N. Kong, S. Shen, S. Sheng, and A. Servin. 2015. *Inventory and Analysis of WHOIS Registration Objects. RFC 7485. IETF.* Retrieved April 4, 2024 from https://www.rfc-editor.org/rfc/rfc7485.txt

A CONTRIBUTION BREAKDOWN

A.1 Addison

- Bulk of research
- Abstract
- 1 Introduction Section
- 2 Background Section
- 4 Implementation Results Section (except for 4.4 Current Issues)
- 5 Conclusion
- Entry collection troubleshooting

A.2 Marwan

- Bulk of implementation
- 3 Implementation Methodology section
- 4.4 Current issues