University of Glasgow | School of Computing Science

Assessed Coursework

Course Name	**Networked Systems (H)**			
Coursework Number	**Summative exercise 2**			
Deadline	**Time:**	**4:30pm**	**Date:**	**11 March 2016**
% Contribution to final course mark	**10%**			
Solo or Group ✓	**Solo**	✓	**Group**	
Anticipated Hours	**10**			
Submission Instructions	**Submit via Moodle, following instructions in the lab 3 handout.**			
Please Note: This Coursework cannot be Re-Assessed				

Code of Assessment Rules for Coursework Submission

Deadlines for the submission of coursework which is to be formally assessed will be published in course documentation, and work which is submitted later than the deadline will be subject to penalty as set out below.

The primary grade and secondary band awarded for coursework which is submitted after the published deadline will be calculated as follows:

(i) in respect of work submitted not more than five working days after the deadline
   a. the work will be assessed in the usual way;
   b. the primary grade and secondary band so determined will then be reduced by two secondary bands for each working day (or part of a working day) the work was submitted late.

(ii) work submitted more than five working days after the deadline will be awarded Grade H.

Penalties for late submission of coursework will not be imposed if good cause is established for the late submission. You should submit documents supporting good cause via MyCampus.

Penalty for non-adherence to Submission Instructions is 2 bands

You must complete an "Own Work" form via https://studentltc.dcs.gla.ac.uk/ for all coursework

Networked Systems (H) lab 3: A map of (part of) the Internet

Dr Colin Perkins
School of Computing Science
University of Glasgow
https://csperkins.org/teaching/2015-2016/networked-systems/

9/10 February 2016

Introduction

The Internet is a network of networks. Each network acts as an autonomous system (AS). That is, the networks that form the Internet are independently operated, and provided they implement the Internet Protocol (IP), can make their own technology choices, and set their own routing policies. In this third laboratory exercise you will explore the topology of the Internet, studying both the router-level topology of the Internet, to map out the devices that form the network, and the AS-level topology of the network, to understand how the different organisations that run the network interconnect.

This is a summative exercise, that is worth 10% of the marks for this course.

IP Addresses of Popular Websites

The Internet separates naming from addressing. The network operates using IP addresses, that denote locations in the network, while users prefer to use readable domain names. The domain name system (DNS) is an application that runs of the network, and translates names into addresses.

Popular websites, and other networked services, use multiple servers, hosted at different locations in the network. This increases robustness of the service to failures, and also allows for load balancing across different data centres and network connections. To support this, domain names often correspond to more than one IP addresses, in different locations in the network, or perhaps connected using different protocols, such as IPv4 and IPv6.

Write a program, `dnslookup`, that takes a list of domain names on the command line, and performs a DNS lookup for each name (hint: use `getaddrinfo()` as described in lab 1, but without trying to connect to each address, then `inet_ntop()`). After each DNS lookup, print out the list of IPv4 addresses returned. For example, your program might print the following, if asked for the addresses of `www.google.com`:

```
$ ./dnslookup www.google.com
  64.233.184.103
  64.233.184.147
  64.233.184.99
  64.233.184.104
  64.233.184.105
  64.233.184.106
$
```

Use your `dnslookup` tool to find the IPv4 addresses for a range of websites. These should be a mix of academic, commercial, social network, news, and personal websites, from a wide range of different countries.

The Router-level Network Topology

The `traceroute` tool can discover the network path used to reach a particular destination. It works by sending UDP probe packets towards the destination with a lower than usual time-to-live (TTL) value specified in the IP header. The first probe sent has TTL = 1, the second has TTL = 2, and so on. Each router that the probe traverses in the network decreases the TTL by one, until the TTL reaches zero. When the TTL reaches zero, the router discards the probe packet and sends back a TTL exceeded message to the sender of the probe. The `traceroute` tool listens for the TTL exceeded messages, and prints out the address the TTL exceeded message came from, and the time taken (or a $\star$ if it receives no response). By using increasing TTL values, `traceroute` forces a response from the first router on the path, then from the second router on the path, and so on, until it reaches the destination. For example, a traceroute to `www.google.com` might show:

```
-->traceroute -n -q 1 www.google.com
traceroute to www.google.com (64.15.119.108), 30 hops max, 60 byte packets
 1  130.209.240.48  0.320 ms
 2  130.209.2.17    0.297 ms
 3  130.209.2.122   1.078 ms
 4  194.81.62.153   1.050 ms
 5  146.97.41.9     1.248 ms
 6  146.97.33.53    5.537 ms
 7  146.97.35.50    5.426 ms
 8  193.62.157.198  5.828 ms
 9  *
10  *
11  *
...
```

The `-q 1` option means only probe each hop once, the `-n` option means don't try to look-up the domain names for the routers. Each line gives the distance in hops, the IP address of the router, and the time taken for the response. The first router (130.209.240.48 in this example) is the device directly connected to the sender. That connects, in turn, to a router with IP address 130.209.2.17, which connects to 130.209.2.122, and so on. A long sequence of $\star$ responses, as seen at the end of the example trace, indicates a firewall that is blocking traceroute requests. In this example, the firewall is between hops 8 and 9 on the path, and prevents traceroute to any later hop.

Run `traceroute` to each of the IP addresses you discovered using your `dnslookup` tool, redirecting the output into files. You might find it helpful to write a script to do this.

Write a program, in the language of your choice, that reads the saved traceroute files, strips out any hops that didn't respond to traceroute (lines with $\star$ responses), and reformats them as a list of pairs of adjacent router addresses, in quotes, each pair separated by `--`. For example, if run on the traceroute output shown above, it would display:

```
"130.209.240.48" -- "130.209.2.17"
"130.209.2.17"   -- "130.209.2.122"
"130.209.2.122"  -- "194.81.62.153"
"194.81.62.153"  -- "146.97.41.9"
"146.97.41.9"    -- "146.97.33.53"
"146.97.33.53"   -- "146.97.35.50"
"146.97.35.50"   -- "193.62.157.198"
```

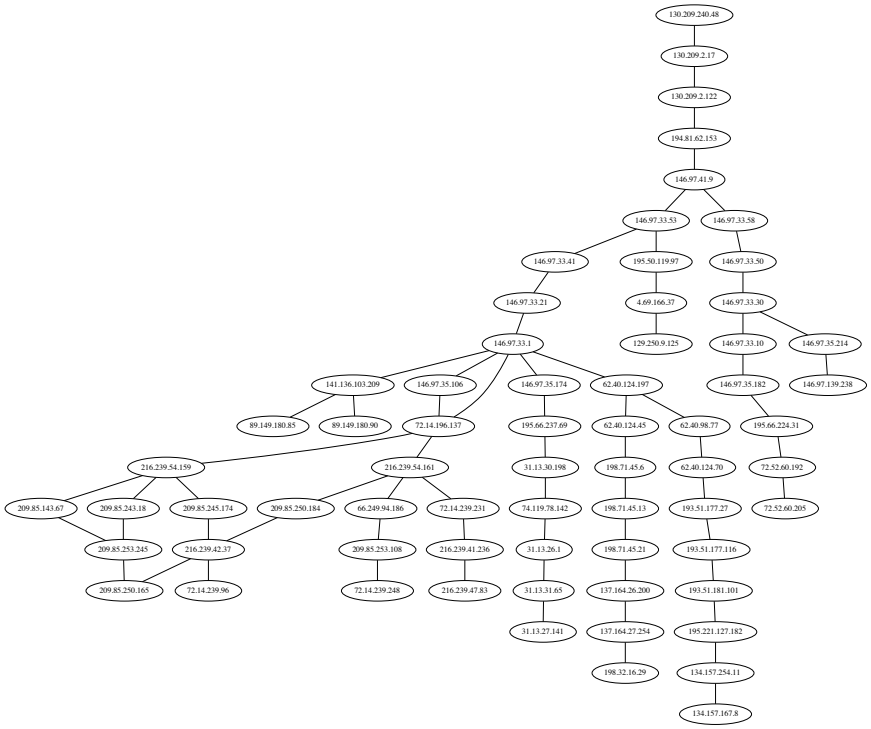

Figure 1: Sample network topology visualisation from a single site

Run your program on each of the saved traceroute files in turn, and save the output from each in a separate file (a "processed traceroute file"). Then, concatenate the contents of all your processed traceroute files into a single file called `router-topology.dot`, sorting them into order, removing duplicate lines (hint: `cat ... | sort | uniq > ...`), and adding a line:

```
graph routertopology {
```

to the beginning, and a line containing just:

```
}
```

at the end. The resulting `router-topology.dot` file is a description of the network topology, and should be a valid file in the *dot* graph description language.[1]

Use the `dot` program, installed on the Linux machines in the lab and part of the Graphviz suite of tools (or some other graph plotting program of your choice) to generate a router-level map of the network topology, as a file called `router-topology.pdf`. Figure 1 shows an example topology map, of the sort you are to produce.

To improve your understanding of the network, review your network topology map. Think about what is the longest path you can find in the network? Are paths from different locations to the same destination disjoint? Are there multiple routes to some destinations? Can you infer organisational boundaries (e.g., which parts of the network below to different ISPs) from the IP address changes? You should not submit answers to these questions, but you might want to discuss them with the lab demonstrators, to check your understanding of the network.

[1] `https://en.wikipedia.org/wiki/DOT_(graph_description_language)`

The AS-level Network Topology

The routers you discovered in the previous part of this exercise are operated by different organisations (that is, by different *autonomous systems*, in Internet routing terminology). Each autonomous system (AS) is uniquely identified by an AS number, and owns one, or more, blocks of IP addresses, known as IP address prefixes. The Border Gateway Protocol (BGP) is an inter-domain routing protocol, that distributes the information (the *routing table*) about which AS owns which prefix, and what route to take to reach that AS. The *AS-level network topology* is a map of the network that shows not the connections between individual routers, but rather than connections between the organisations that own the routers. The aim of this part of the exercise is to find which AS owns each of the routers you found in the previous part, and to use that information to draw a map of the AS-level network topology.

To do this, you need a copy of the BGP routing table, and a program to parse the contents of the routing table. The RouteViews project archives copies of the BGP routing table. You can fetch a recent archive by running the following command on the Linux machines on the lab, or by opening the link in a web browser:

```
curl -O# http://routeviews.org/bgpdata/2016.01/RIBS/rib.20160101.0000.bz2
```

This is a compressed archive of the routing table, collected from a router at the University of Oregon. Do not decompress it.

Next, download, compile, and install the `bgpdump` tool by running the following sequence of commands in the terminal of the Linux machines in the lab:

```
curl -O# http://www.ris.ripe.net/source/bgpdump/libbgpdump-1.4.99.15.tgz
tar zxvf libbgpdump-1.4.99.15.tgz
cd libbgpdump-1.4.99.15
./configure --prefix=$HOME
make && make install
```

These will install a binary called `bgpdump` in your `$HOME/bin` directory.

The `bgpdump` tool can read the compressed archive of the BGP routing table, and display it in an easy to parse format. To view the contents of the routing table, run the command:

```
bgpdump -Mv rib.20160101.0000.bz2
```

The complete output is around 24 million lines long, so you might want to pipe it into a pager such as `less` rather than have the entire file display in the terminal. Each line looks something like the following:

```
TABLE_DUMP2|01/01/16 00:00:01|B|129.250.0.11|2914|1.0.0.0/24|2914 6453 15169|IGP
```

There are eight fields on each line, separated by `|` characters. For this exercise, the important fields are the *prefix* in field 6 (`1.0.0.0/24` in this example) and the *AS path* in field 7 (`2914 6453 15169`). The last entry in the AS path field is the AS number of the network that owns the range of IP addresses denoted by the prefix. The other entries in the AS path field list the autonomous systems to traverse in order to reach the destination. In the example, AS 15169 owns IP addresses in the prefix 1.0.0.0/24 (i.e., IP addresses whose first 24 bits match the first 24 bits of 1.0.0.0). AS 15169 is reachable, from the RouteViews collector, by sending the data to AS 2914, which will send it to AS 6453, which will finally send it to AS 15169 (i.e., via each AS listed in the AS path field in turn). The routing table will usually contain more than one entry for each prefix, since there are different routes from the collector to the AS owning the prefix.

For a given IP address, you can find the AS number that owns the address by using a longest prefix match. For example, the routing table contains the prefixes 32.0.0.0/8 owned by AS 2686, and 32.2.128.0/18 owned by AS 8030. The IP address 32.2.128.1 would match both prefixes, since its first 8 bits match those of 32.0.0.0, and its first 18 bits match those of 32.2.128.0, but since 18 is larger than 8, AS 8030 owns the address.

The file `http://www.cidr-report.org/as2.0/autnums.html` contains a list of all AS numbers, along with the name corresponding organisation. For example it tells us that AS 8030 is AT&T WorldNet. Download and save a copy of this file.

Write a C program called `aslookup` that takes as command line parameters the name of a RIB file, and a list of IP addresses, and prints out the AS number and name corresponding to each of the IP addresses. For example, if you run:

```
aslookup rib.20160101.0000.bz2 130.209.240.1 128.9.176.20 32.2.128.1
```

it would display:

```
130.209.240.1  786  JANET Jisc Services Limited,GB
 128.9.176.20    4  ISI-AS - University of Southern California,US
   32.2.128.1 8030  WORLDNET5-10 - AT&T WorldNet,US
```

The `aslookup` program will look up the IP addresses in the RIB file specified, using a longest prefix match to get the AS number, then look up the AS number in a local copy of the `autnums.html` file. If the IP address does not have a corresponding AS number, your `aslookup` tool should print an AS number of – and name `unknown`, for example:

```
10.0.0.1         -  unknown
```

If more than one AS claims the IP address with the same prefix length, your `aslookup` tool should print an AS number of – with the name `multiple`.

You should assume the RIB file specified on the command line, the `autnums.html` file, and the `bgpdump` binary are available in the same directory as `aslookup`. Your code must be a single C file, `aslookup.c`, that compiles without error on the Linux machines in the lab, using the command:

```
clang -W -Wall -Werror aslookup.c -o aslookup
```

There are some hints on how to write `aslookup.c` in the Appendix. Some up-front design work is beneficial here – don't just start hacking code without planning.

Use the `aslookup` program to find the AS number corresponding to each of the IP addresses in the router-level topology you generated earlier. Generate a new DOT file, called `as-topology.dot` derived from `router-topology.dot` but with each IP address replaced by the corresponding AS number and the first word of the AS name, and with any duplicate lines, or lines showing a connection between an AS and itself, removed. If you can't find an AS number for some IP addresses, leave those IP addresses in the final output.

Redraw the topology map using `as-topology.dot` as input, to produce a new AS-level topology map, in a PDF file call `as-topology.pdf`. Format the resulting PDF file to display on a single sheet of A4 paper. The text should be large enough to be legible when printed. Figure 2 is an example of the type of AS topology map expected.

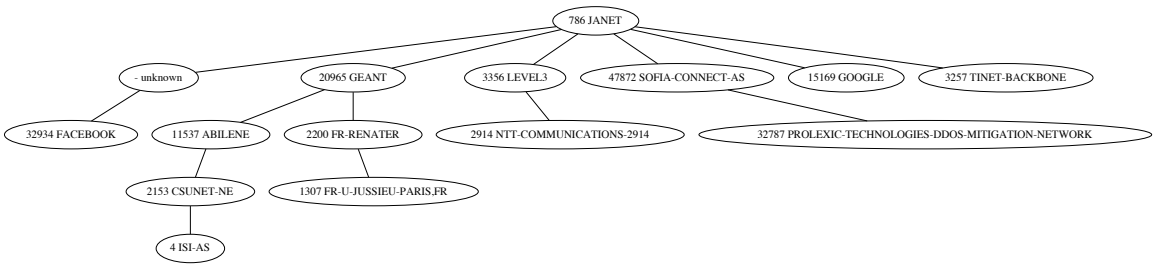

Figure 2: Sample AS-level topology visualisation from a single site

Submission Instructions

Create a zip archive, called `lab03-`*`matric`*`.zip`, replacing *`matric`* with your 7-digit matriculation number, that extracts into a subdirectory named `lab03-`*`matric`*. Once extracted, the subdirectory must contain the following files: `dnslookup.c`, `aslookup.c`, `router-topology.pdf`, and `as-topology.pdf`. Do not include any other files. Submit the zip archive via Moodle. For example, if your matriculation number is 1234567, create an archive called `lab03-1234567.zip` that expands to create the directory `lab03-1234567` containing the specified files.

Check carefully that your zip archive extracts into the appropriate subdirectory, and contains only the requested files.

This is an assessed exercise, worth 10% of the marks for this course. The deadline for submissions is 4:30pm on Friday 11 March 2016. The Code of Assessment allows late submission up to 5 working days beyond this deadline, subject to a penalty of 2 bands for each working day, or part thereof, the submission is late. Submissions received more than 5 working days after the due date will receive an H (band value of 0).

Submissions that are not made via Moodle, that have the wrong filename, that have a zip archive that extracts into the wrong directory or that contains the wrong files, or that otherwise do not follow the submission instructions will be subject to a 2 band penalty. This penalty is in addition to any late submission penalty.

Marking Scheme

We will mark submissions considering the following:

- Correct operation of the `dnslookup` program. We will compile your `dnslookup` program using the command `clang -W -Wall -Werror dnslookup.c -o dnslookup` run on the Linux machines in the lab. If your `dnslookup` program compiles cleanly, we will use it to look-up a list of domain names, to find their IP addresses. We will the compare the IP addresses your program finds for these names with the IP addresses found by a copy of the `dnslookup` program we have prepared, assigning marks for correct matches. If your `dnslookup` tool does not compile cleanly, no marks will be assigned for this section. We will return the mark, along with the set of names tested, and the expected output. This task is worth 2% of the marks for this course.
- Correct operation of the `aslookup` program. We will compile your `aslookup` program using the command `clang -W -Wall -Werror aslookup.c -o aslookup` run on the Linux machines in the lab. If your `aslookup` program compiles cleanly, we will test it using a RIB file you have not previously seen, to find the AS numbers for a set of IP addresses of our choice. Marks

will be assigned for correct AS number and AS name lookups. If your `aslookup` tool does not compile cleanly, no marks will be assigned for this section. We will return the mark, along with the name of the RIB file tested against, the IP addresses tested, and the expected output. This task is worth 6% of the marks for this course.

- Quality of your router- and AS-level topology maps. Marks will be assigned for including a range of destinations in different locations, correctly handling routers that don't respond to traceroute, and for showing the interconnection of autonomous systems in the network, and for producing a visualisation that is clear and easy to read. We will return a mark, along with a brief written justification of the mark. This task is worth 2% of the marks for this course.

In total, this exercise is worth 10% of the marks for this course.

Appendix: Hints for the `aslookup` program

The pseudo-code for the `aslookup.c` file might look something like the following rough outline:

```
FILE *rib_file = popen("bgpdump -Mv filename", "r");

foreach line in rib_file {
  extract prefix and AS number
  if prefix != previous prefix {
    save prefix and AS number in hash table, indexed by first octet of prefix
  }
}

load AS number to AS name mapping file (autnums.html)

foreach address {
  foreach prefix in appropriate row of hash table {
    if address matches prefix {
      if (first match for this prefix) or (longer prefix than previous match) {
        store corresponding AS number and AS name
      }
    }
  }

  print address, and matching AS number and AS name
    or print error if not found
    or print error if found multiple times
}

pclose(rib_file);
```

The `popen()` function allows a C program to run another program, such as `bgpdump`, and incrementally read its output as it runs concurrently to the original program. You will need to parse the output of `bgpdump` to extract the prefix and AS path. The standard C library provides functions like `sscanf()` and `strrchr()` that can help do this.

A simple chained hash table, using 256 buckets based on the first octet of the prefix, is easy to implement and performs well enough for storing the prefix-to-AS number matching table, but you can use any form of hash table, or other data structure, you think appropriate.

The following code might be helpful to check if an address matches a prefix:

```
static void
bytes_to_bitmap(int byte, char *bitmap)
{
  int offset = 0;

  assert(byte >= 0);
  assert(byte <= 255);

  for (int i = 7; i >= 0; i--) {
    if ((byte & (1 << i)) != 0) {
      bitmap[offset++] = '1';
    } else {
      bitmap[offset++] = '0';
    }
  }
}

static int
addr_matches_prefix(char *addr, char *prefix) {
  // Check if an IP prefix covers an IP address. The addr field is an IPv4
  // address in textual form (e.g., "130.209.240.1"); the prefix field has
  // its prefix length specified in the usual address/length format (e.g.,
  // "130.209.0.0/16"). Return 1 if the prefix covers the address, or zero
  // otherwise. This is not an efficient implementation, but is simple to
  // debug, since the bitmaps it uses for comparison are printable text.

  // Parse the address:
  int  addr_bytes[4];
  char addr_bitmap[32+1];

  sprintf(addr_bitmap, "                                ");

  if (sscanf(addr, "%d.%d.%d.%d",
        &addr_bytes[0], &addr_bytes[1], &addr_bytes[2], &addr_bytes[3]) != 4) {
    printf("cannot parse addr\n");
    return 0;
  }

  for (int i = 0; i < 4; i++) {
    bytes_to_bitmap(addr_bytes[i], &addr_bitmap[i * 8]);
  }

  // Parse the prefix:
  int  prefix_bytes[4];
  int  prefix_len;
  char prefix_bitmap[32+1];

  sprintf(prefix_bitmap, "                                  ");

  if (sscanf(prefix, "%d.%d.%d.%d/%d",
        &prefix_bytes[0], &prefix_bytes[1], &prefix_bytes[2], &prefix_bytes[3],
        &prefix_len) != 5) {
    printf("cannot parse prefix\n");
    return 0;
  }

  for (int i = 0; i < 4; i++) {
    bytes_to_bitmap(prefix_bytes[i], &prefix_bitmap[i * 8]);
  }
```

```
  for (int i = prefix_len + 1; i < 33; i++) {
    prefix_bitmap[i-1] = '?';
  }

  // Check if address matches prefix:
  for (int i = 0; i < 32; i++) {
    if ((addr_bitmap[i] != prefix_bitmap[i]) && (prefix_bitmap[i] != '?')) {
      return 0;
    }
  }
  return 1;
}
```

The following code might be helpful to load the `autnums.html` file:

```
struct as {
  int        num;
  char      *name;
  struct as *next;
};

static struct as *
load_autnums(void)
{
  struct as *head = NULL;
  FILE      *inf  = fopen("autnums.html", "r");

  if (inf != NULL) {
    int  buflen = 1024;
    char buffer[buflen+1];

    while (!feof(inf)) {
      memset(buffer, 0, buflen+1);
      fgets(buffer, buflen, inf);
      if (strncmp(buffer, "<a href=\"/cgi-bin/as-report=AS", 8) == 0) {
        int   asnum;
        char  asname[1024];
        char *p = "<a href=\"/cgi-bin/as-report?as=AS%*d&view=2.0\">AS%d </a> %[^\r\n]";
        if (sscanf(buffer, p, &asnum, asname) == 2) {
          struct as *curr = malloc(sizeof(struct as));
          curr->num  = asnum;
          curr->name = strdup(asname);
          curr->next = head;
          head = curr;
        }
      }
    }

    fclose(inf);
  }
  return head;
}
```

You can freely use either of the above code snippets, if they are useful, but do not have to do so. When working on your implementation, you're advised to focus on correctness, and ease of writing and debugging the code, rather than on performance. It's acceptable if your implementation of `aslookup` takes a some time to run, provided it displays something to indicate progress while it runs (e.g., print a dot every 100,000 lines read from `bgpdump` – don't forget to `fflush(stdout)`).

If you don't understand the above code, or any of the instructions provided in the handout, then please ask for help.