New York University
Computer Science Department

G22.2620-001, Fall 07 **Problem Set 1** Solution by Hui Zhang

---

1 DNS in the real world

In this problem, you will learn more about DNS using the UNIX utility **dig**. You may also find it useful to consult RFC1035 for some of the questions.

There are two types of DNS queries, *recursive* and *iterative*. When a DNS resolver issues a recursive query to a name server, the server attempts to resolve the name completely with full answers (or an error) by following the naming hierarchy all the way to the authoritative name server. Upon receiving an iterative query, the name server can simply give a referral to another name server for the resolver to contact next. A resolver sets the RD (recursion desired) bit in DNS query packet to indicate that it would like to have the query resolved recursively. Not all servers support recursive queries from arbitrary resolvers.

1. What is **www.nyu.edu**'s canonical name? What are its authoritative name servers? Based on **dig**'s output, could you tell which DNS server answers this DNS query? Is it a recursive query?

   Answer: NYU's canonical name: WEB.nyu.edu
   NYU's authoritative name servers:
   NS1.nyu.edu, NS2.nyu.edu, NYU-NS.BERKELEY.edu, LAPIETAR.NYU.FLORENCE.IT

   The DNS server that answers this query:
   It depends on where you do the query. For example, if you do the query at home, the DNS server your ISP assigned to you will answer it.
   Is it a recursive query: Yes, it is a recursive query.

2. Instead of using your default name server, issue the query for **www.nyu.edu** to one of the root DNS servers (e.g. **a.root-servers.net**). Does this server accept recursive query from you? If not, perform iterative queries yourself using **dig** by following the chain of referrals to obtain the **www.nyu.edu**'s address. What are the sequence of name servers that you have queried? Which domain is each name server responsible for?

   Answer: No, it doesn't accept recursive query.
   A query chain example:

   ```
   a.root-servers.net (root)
   H3.NSTLD.COM (edu)
   NS2.nyu.edu (nyu.edu)
   ```

3. Use multiple recursive DNS servers located at different geographical regions[1] as well as your default name server to resolve **www.google.com**. Attach your **dig** output. What geographical regions do those IP addresses reside? How quickly do the corresponding A and NS records

[1] Here are two name servers that answer recursive queries: ns2.cna.ne.jp ns2.suomen2g.fi

expire? Why do A records expire so soon? Compare this setup using DNS with some alternative way of achieving the same goal.

Answer: Many students misunderstood this question. This question asks for the geographical locations of web servers returned by different DNS servers, not the geographical regions of DNS servers themselves.

One common way to map an IP address to its geographic location is to use "traceroute" to obtain domain names of the routers along the path to an IP address in the hope of getting some location hints based on routers' domain names. Unfortunately, traceroute did not work for Google's IP addresses as most routers along the path do not have domain names with helpful location hints.

I tried to infer an IP address's geographic location using RTT times obtained from "ping". This could be very accurate if you can perform the "ping" from multiple machines in different geographical areas. I used a couple of PlanetLab machines to do the ping.

Here is what I got:

	local DNS server	ns2.cna.ne.jp	ns2.suomen2g.fi
IP address of Google server	64.233.169.147 64.233.169.99 64.233.169.103 64.233.169.104	66.249.89.104 66.249.89.99 66.249.89.147	66.249.91.103 66.249.91.104 66.249.91.147 66.249.91.99
Location of ping source	nyu	Japan	U.K
RTT	8ms	9ms	12ms
inferred location	east coast	east Asia	west Europe

As the above table shows, queries from DNS servers at different geographical regions return Google servers at different geographical regions. This is because Google uses DNS to perform server selection so clients would be able to access a geographically close-by server with lower latency.

How quickly do the corresponding A and NS records expire? Why do A records expire so soon? Answer: TTL for A records goes up to 300 seconds and 86400 seconds for NS records. A records expire relatively quickly to allow load balancing among servers. In this case, when a cached web server encounters heavy load, the DNS server will query again after 300 seconds and hopefully obtain another lightly loaded server.

Another way to do load-balance is at the application level. A load balancer server receives all requests and redirects them to different application (e.g. Web) servers according to servers' load.

4. Alice works at a search engine startup whose main competitor is Google. She would like to crush her competitor in the "non-traditional" way by messing up with DNS servers. Recalling from her networking class that DNS servers cache A and NS records from DNS replies and referrals, Alice realizes she can configure her own DNS server to return incorrect results for arbitrary domains. If the resolver caches Alice's malicious results, it will return bad results to future DNS queries. Help Alice complete her master plan to hijack Google's domain name by writing down exactly what Alice's name server returns upon a DNS query. What must a robust DNS server implementation do to counter this attack?

Answer: Here is an attempt to hijack Google's domain name. When Alice's DNS server receives a query for $www.alicestartup.com$, it returns the following malicious results:

```
www.alicestartup.com    long-TTL    in  NS    ns1.google.com
google.com              long-TTL    in  NS    ns1.google.com
ns1.google.com          long-TTL    in  A     w.x.y.z (Alice's DNS server)
```

If a DNS server blindly caches everything, it will redirect all future queries for $www.google.com$ to Alice's nameserver (w.x.y.z).

A robust DNS server implementation should be less trustful of results returned by other DNS servers and only cache information that's directly relevant to the queried domain. In the above example, since $google.com$ is not a subdomain of $alicestartup.com$, a correct DNS server implementation should ignore all information related to $google.com$ in the results.

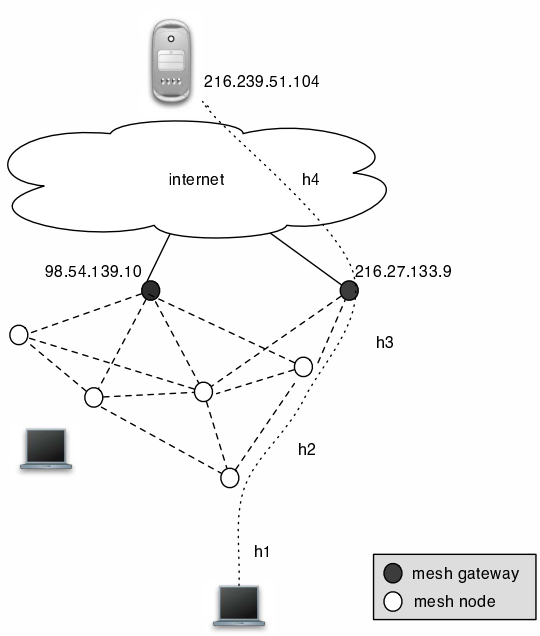

Figure 1: The architecture of Alice's mesh network. Round circles denote WiFi-based mesh nodes with dotted lines representing wireless connectivities among them. Each mesh node has a mesh interface. Each gateway mesh node, denoted by a black circle, has an additional wired interface to a DSL or cable modem link for connecting to the rest of the Internet. Each mesh node also acts as an 802.11 base station with whom a client node such as a user's laptop associates itself with. Unlike mesh nodes, Alice does not control clients' software configuration and would (ideally) not want to install any custom software on them.

2 Setting up an urban mesh network

Alice got fired from her last job at the search engine company for being evil and she has recently joined an effort to provide community supported WiFi mesh network in New York City. Figure 1 explains the basic setup of her network. In this problem, you will help Alice decide on a good addressing scheme for her mesh network.

Each mesh gateway's wired interface is already assigned an IP address (via DHCP or statically) by its ISP as shown in Figure 1. Alice's mesh software runs on each mesh node and automatically builds a routing table based on mesh nodes' IP addresses. Therefore, each mesh node knows how to route a packet to any other IP address belonging to another mesh interface. Therefore, Alice needs an addressing scheme that assigns an IP address to each mesh interface (both mesh nodes and mesh gateways have a mesh interface). Explain how you would assign an IP address to a mesh interface. Additionally, describe how to assign an IP address to a client laptop (Remember that you cannot expect to change a client's software).

Answer: A solution submitted by many students is to designate one mesh node as the DHCP server who assigns IP addresses to other mesh nodes. However, this solution does not work as mesh nodes cannot route to the DHCP server unless they already have IP addresses. This is because Alice's routing software builds routing tables based on nodes' IP addresses.

One working scheme is to manually assign IP address to mesh nodes and laptops. Obviously this is labor intensive. A better scheme is to let each mesh node randomly assign an IP address to itself. Since

the assigned IP address space (e.g. 10.*.*.*) is fairly large, it is highly unlikely that two independently assigned random IP addresses happen to be the same. The exact probability can be calculated by observing that this problem is a variant of the famous "birthday paradox" problem. Specifically, when there are much fewer than a few hundred nodes ($\ll \sqrt{2^{24}} = 2000$), the collision probability is negligible. To generate random numbers in a deterministic fashion, we can use a collision-resistant hash of a node's MAC address so a node's IP address could remain the same across reboots. A number of students suggest simply using the last 24-bits of a node's MAC address as its IP address. This is not desirable because there's no guarantee that these last 24-bits are truly random.

Once each mesh node has an IP address, it can act as a DHCP and NAT server to assign IP address for laptop clients from another private address space (e.g. $192.168.*.*$).

Suppose the client laptop sends an IP packet destined for Google (216.239.51.104) (Figure 1), describe the source and destination IP address fields of the IP packet as it traverses the sequence of path segments: h1, h2, h3, h4.

Answer: The destination field never changes in the full path, while the source field will be changed by various NAT servers along the way. Suppose the mesh node that the laptop associates with has IP address $10.1.1.23$ and it assigns the laptop a private address $192.168.1.10$. Let us assume the gateway mesh node has IP address $10.0.0.13$ and a public IP address $216.27.133.9$. Then, the source and destination IP address fields on each path segment for a packet travelling from the laptop to an Internet host ($216.239.51.104$) are:

	source IP	destination IP
h1	192.168.1.10	216.239.51.104
h2	10.0.0.13	216.239.51.104
h3	10.0.0.13	216.239.51.104
h4	216.27.133.9	216.239.51.104

The source/destination fields for a packet travelling in the reverse direction from $216.239.51.104$ to the laptop are:

	sourece IP	destination IP
h4	216.239.51.104	216.27.133.9
h3	216.239.51.104	10.0.0.13
h2	216.239.51.104	10.0.0.13
h1	216.239.51.104	192.168.1.10

Does your addressing scheme support seamless mobility? (i.e. can the client laptop keep its ongoing connections while moving its association from one mesh node to another?) If not, can you sketch a different addressing scheme that does?

Answer: The above scheme doesn't support seamless mobility. Packets intended for the laptop will be routed to the original mesh gateway ($10.0.0.13$) even after the laptop has moved to another mesh node. One way to support seamless mobility is to assign each laptop an IP address in the same subnet (10.*.*.*) as mesh nodes and rely on underlying routing protocol to automatically update routes as laptops roam. However, a big disadvantage of this scheme is that client laptops' software need to be updated to run Alice's mesh routing protocol.
Another proposal uses a scheme inspired by MobileIP. Alice could (manually) assign a mesh node (home agent) to each laptop. When a laptop associates with a new mesh node (foreign agent), the mesh node informs the laptop's home agent about the laptop's new location. Then packets from the laptop are directly routed through its foreign agent to Internet hosts. However, packets destined to a laptop is always routed to its home agent first who then forwards them to the laptop's current foreign agent.

3 TCP checksum

If you look up TCP headers carefully in any standard textbook, you will notice that TCP has a checksum field that covers parts of the IP header (source address, destination address and length fields).

1. Why does TCP checksum include part of IP header fields when IP already computes a separate checksum covering its own header?

   Answer: TCP's checksum tries to ensure integrity on the "end-to-end" basis. In contrast, IP's checksum only ensures "hop-to-hop" integrity. The IP header could be modified and a new checksum could be re-calculated along each IP forwarding hop. For example, IP routers need to decrease each packet's TTL at each hop and thus recalculates IP checksum. If IP routers happen to corrupt the source IP address during TTL processing, one will not be able to detect the corruption using IP checksum since it is recalculated. However, TCP-checksum will detect such incorrectness and discard the corrupted packets.

2. When a TCP receiver detects an incorrect checksum, it can either a) discard the segment and send a cumulative ACK for the expected in-sequence byte or b) discard the segment and do nothing else. Which action is preferable? Why?

   Answer: It is preferable to discard the segment and do nothing else.

   When a TCP receiver detects an incorrect checksum, it has no way to decide whether the source address is correct or not. Sending an error message to a wrong source makes no sense. Since senders will eventually retransmit un-acknowledged packets, the receiver can simply drop corrupted packets.

4 Designing a transport protocol for RPCs

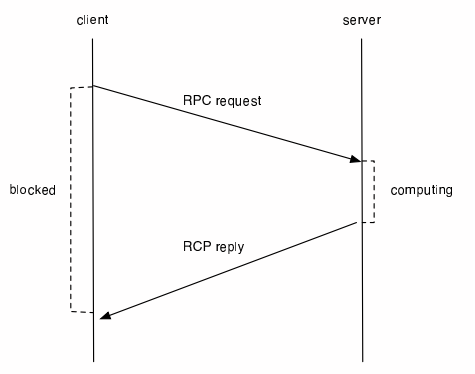

Figure 2: A timeline between RPC client and server.

```
//RPC client
...
char *buffer = "Internet rocks";
doRPC(server, writefile, "/home/jinyang/net.txt", buffer, 0, 14);
...

//RPC server
writefile(char *filename, char *buffer, int offset, int size, char[] result)
{
int f = open(filename);
seek(f, offset);
write(f, result, size);
return RPC_SUCCESS;
}
```

Remote Procedure Call (RPC) is a popular paradigm for programming distributed systems. (It is also sometimes called Remote Method Invocation as in java). RPC emulates the semantics of a local procedure call in which a caller makes a call into a procedure and blocks until the call returns. It is implemented using a request/reply message passing paradigm between an RPC client and server (see Figure 2). The pseudocode gives an example where a client reads a file from the server using the writefile RPC. In this problem, you will design a simple protocol for transporting RPC messages between the client and server using the standard UDP datagram socket interface.

We start with a design called *SimpleRPC*. In *SimpleRPC*, each UDP message consists of a RPC header with three fields: message type (indicating whether the message is a request or reply), a unique identifier (UID), procedure identifier. The RPC data contains marshalled procedure arguments or return values.

For each RPC request, the client generates a new UID (e.g. by incrementing a counter), suspends the current running thread and awaits for a corresponding reply with the same UID from the server. If a reply arrives with a UID for which a blocked thread is waiting for, it resumes the execution of the thread. (If a reply arrives with a UID that no thread is waiting for, the client simply discards it.) If no matching RPC reply arrives within 20 ms, the client retransmits the request. The RPC

server is completely stateless: it simply invokes the desired function based on procedure identifier for each received RPC request and sends back the corresponding reply.

1. Explain the significance of the fixed timeout threshold of 20 ms. Under what deployment scenarios do you expect it to work well? Or alternatively, what are the circumstances in which a 20 ms fixed timeout becomes problematic?

   Answer: The fixed timeout works well when the client-server RTT plus the server computation time is always less than 20ms. A typical such scenario would be on a LAN network ($<$ 1ms RTT) and with fast RPC procedures (e.g. read/write 32K blocks from a lightly loaded file server).

2. An RPC system possesses *at-most-once* semantics if it guarantees no procedures are executed more than once at the server as a result of the same RPC invocation. Is *SimpleRPC at-most-once*? If not, do you think it affects the correctness of *all* applications using *SimpleRPC*? Give some concrete examples. For example, does duplicate execution affect our procedure writefile in the pseudocode?

   Answer: SimpleRPC is not at-most-once. For example, if the server's RPC reply is lost, the client will re-send the same RPC request, causing the sever to execute the same request twice.

   Duplicate execution of `writefile` doesn't affect correctness. The offset argument in the pseudocode causes any duplicate execution to overwrite the same range of a file with the same contents. However, duplicate execution of other types of procedures will cause correctness problems, such as a procedure that counts numbers or a procedure that appends to a file.

   Someone suggests you add a small amount buffer at the server to remember the UID and corresponding results of *recently* executed RPCs. If an RPC request arrives with a UID already present in this buffer, the server simply replies with the corresponding saved result. Does it solve the problem of potential duplicate execution of RPCs? If not, give a design that guarantees at-most-once execution of RPCs. What about a design guaranteeing *exactly-once*? (In addition to message losses, you also need to consider untimely server or client crash.)

   Answer: Caching some UIDs does not solve the problem of duplicate RPC execution because the server can only have a finite buffer size and thus cannot remember all previously executed RPC requests. A simple design guaranteeing *at-most-once* is one that does not retransmit any RPC requests. Obviously, that is not very interesting as what many applications really need is *exactly-once* execution. One could imagine some fixes to SimpleRPC for *exactly-once* execution: for example, we can change SimpleRPC to require each client to send a RPC reply-ack message upon receiving an RPC reply, thus allowing the server to remove acknowledged UIDs from the cache. Together with some other basic flow control mechanisms, we can ensure clients do not overwhelm the server's finite-sized buffer. Unfortunately, if the server or client crash unexpectedly, they will forget previously executed RPC requests. Thus, unless one logs each RPC execution persistently to disk, there is no way to guarantee *exactly-once* execution across machine failures.

   As you can see, guaranteeing *exactly-once* execution could be quite expensive and not all applications need such semantics from the RPC system. Hence, it may be better to leave such strong semantics to individual applications themselves.

3. The RPC arguments and results might be arbitrarily large. In order to avoid extra design and implementation effort, you propose to rely on IP fragmentation to deal with big RPC requests/replies. Why might this not be a good idea?

   Answer: Here is a concrete example why IP fragmentation is bad. If a large message is fragmented into 100 IP packets, then the loss of any one packet will cause the sender to retransmit the entire

set of 100 IP fragments. If each fragment is lost with probability 1%, then the retransmission probability will reach as high as $1 - (1 - 1\%)^{100} = 63.4\%$.

4. You start to get ambitious and want to use *SimpleRPC* for the wide area network. Apart from having to change the timeout threshold, you also realize that you will have to incorporate congestion control. Why is it okay to overlook the issue of congestion control on the local area network?

   Answer: A typical local area network only has a few machines each with 10/100Mbps network interface card and a switching capacity of 100/1000Mbps. Furthermore, the RPC system can probably handle less than 10Mbps traffic (depending on how efficient your implementation is), therefore, you probably won't see much congestion on local area network. (Of course, we are just talking about typical LAN setup here, if you have a huge number of machines, that's a different story.)

   Things are very different with WAN. Without congestion control, your client can push out 10/100Mbps from its network interface card. Majority of this will be immediately dropped at your access link (a T1 link has only 1.5Mbps) before even traveling across the wide area Internet and encountering congested links with even less available bandwidth.

5. You think it is too much hassle to implement TCP-like functionalities such as careful RTO calculation and congestion control in *SimpleRPC* and decide to implement a RPC system on top of TCP instead of UDP. Why might TCP-based RPC not perform well?

   Answer: TCP imposes in-order delivery. Therefore, if one RPC request is lost, TCP will not deliver to the server any of the rest received RPC requests until a retransmission of the lost RPC request is received later. This is not desirable as the server wastes idle CPU/disk resource while it could be processing other received requests. This situation is referred to as the head-of-line blocking.

Summary

A Statistics of the Students' Grades

Grades	Student Num
30	1
25∼29	6
20∼24	7
<20	3