New York University
Computer Science Department

G22.2620-001, Fall 07 (Li) **Problem Set 1** Due: September 24, 2007

Give *short* and *precise* answers to each question. This is an individual assignment. You are allowed to discuss with your classmates, but you must write down your own solution independently. Do not look at anyone else's solutions or copy them from external sources. You can read more about NYU academic integrity guidelines from class home page.

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?

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?

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

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

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

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?

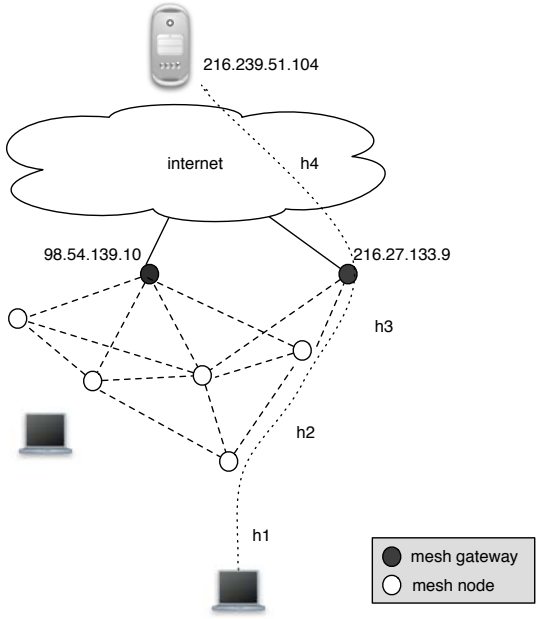

Figure 1: The architecture of Alice's mesh network. Round circles denote WiFi-based mesh nodes with dotted lines representing wireless connectivities among them. Each mesh node has a mesh interface. Each gateway mesh node, denoted by a black circle, has an additional wired interface to a DSL or cable 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).

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.

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?

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?
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?

4 Designing a transport protocol for RPCs

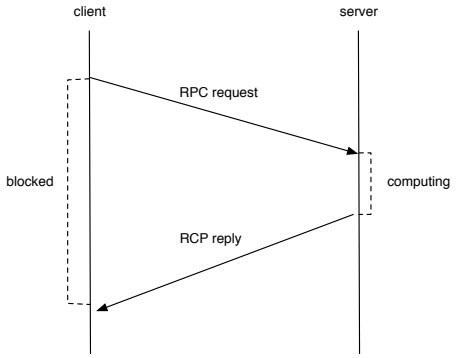

Figure 2: A timeline between RPC client and server.

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

//RPC server
int
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?

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?

   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.)

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?

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?

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?