New York University
Computer Science Department

G22.2620-001, Fall 07 (Li) **Problem Set 1** September 10, 2007

Answer each question as clearly and concisely as possible. This is an individual assignment. You are allowed to discuss questions and ideas 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 authorative 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 DNS servers that you have queried and which domain is each name responsible for?

3. Use multiple recursive DNS servers located at different geographical regions[1](one of which is your default name server) to resolve the query for **www.google.com**. What are the characteristics of the IP addresses being returned? (e.g. Are all servers close to you? How quickly do the A and NS records expire?) Explain the benefits of this setup and compare it to some alternative ways 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, Alice realizes she can configure her own DNS server to return incorrect results for arbitrary domains as referrals. Help Alice complete her master plan to hijack Google's domain name. What must the DNS server implementation do to counter this attack?

---

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

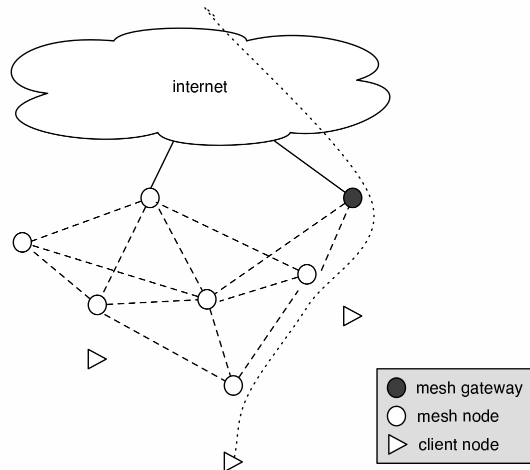

Figure 1: The architecture of Alice's mesh network. Round circles denote WiFi-based mesh nodes deployed by Alice for which she controls all of the node's software. The dotted lines denote wireless connectivities among mesh nodes. The black circles represent gateway nodes which have both a wireless interface and a wired DSL/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 an user's laptop associates itself with. Client nodes are denoted by triangles. Unlike mesh nodes, Alice does not control clients' software configuration and would (ideally) not want them to install any custom software.

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 already comes with a DHCP-assigned (or statically assigned) IP address from an ISP that will be used when communicating with the rest of the Internet. Each client node (represented as a triangle in Figure 1) must have an IP address to enable IP-based communication. Come up with at least two addressing schemes (a) (b) for Alice's mesh network. Scheme (a) assigns an IP address to each mesh node and scheme (b) does not require an IP address for each mesh node. How do clients communicate with the rest of the Internet with each scheme? (e.g. what do routing table entries at each mesh node look like? How addresses are translated at different places?) Discuss the pros and cons of each addressing scheme (e.g. in terms of manageability and mobility).

(If you can not come up with a *working* scheme for either (a) or (b), discuss why it is infeasible or too complicated to do so.)

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

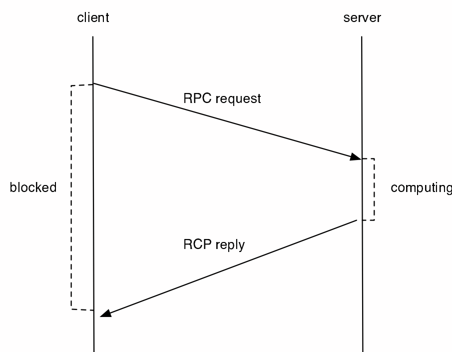

Figure 2: A timeline between RPC client and server.

```
//RPC client
...
char buffer[64000];
doRPC(server, readfile, 0, 64000, "/home/jinyang/net.txt", buffer);
...

//RPC server
int
readfile(char *filename, int offset, int size, char[] result)
{
int f = open(filename);
seek(f, offset);
read(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 readfile 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) and awaits for a corresponding reply from the server. If no such reply arrives within 20 ms, it 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 your RPCs? Give concrete examples. For example, does duplicate execution affect our pseudocode readfile RPC?

   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 discards it. Does it completely 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 some form of 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. How do you expect your new TCP-based RPC to perform?