

Section 12: Distributed Systems and Networking

April 24, 2020

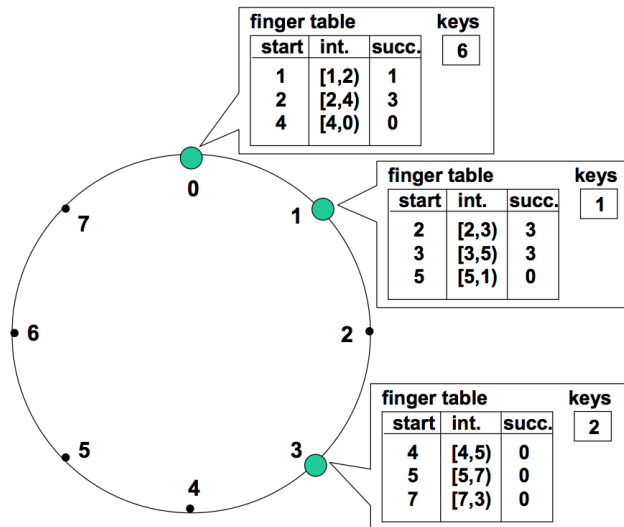
Contents

1	Vocabulary	2
2	RPC	4
2.1	Selecting Functions	4
2.2	Reading Arguments	4
2.3	Handling RPC	5
2.4	Call Stubs	6
2.5	Handling failure	7
3	Distributed File Systems	8
4	Distributed Key-Value Stores	10
5	Networking	12

1 Vocabulary

- **Eventual Consistency** - A weaker form of a consistency guarantee. If a system is eventually consistent, it will converge to a consistent state over time.
- **Network File System (NFS)** - A distributed file system written by Sun. NFS is based on a stateless RPC protocol. Buffers are **write behind**. Few strong consistency guarantees on parallel writes. NFS is **eventually consistent**.
- **Andrew File System (AFS)** - A distributed file system written at CMU. Full files are buffered locally upon **open**. Buffers are **write back** and only flushed on **close**. File contents follow "last write wins" semantics.
- **Endianness** - The order in which the bytes are stored for integers that are larger than a byte. The two variants are big-endian where the most significant byte is at the lowest address and the least significant byte is at the highest address and little-endian where the least significant byte is at the lowest address and the most significant byte is at the highest address. The network is defined to be big-endian whereas most of your machines are likely little-endian.
 - `uint32_t htonl(uint32_t hostlong)` - Function to abstract away endianness by converting from the host endianness to the network endianness.
 - `uint32_t ntohl(uint32_t netlong)` - Function to abstract away endianness by converting from the network endianness to the host endianness.
- **RPC** - Remote procedures calls are a technique for distributed computation through a client server model. This process effectively calls a procedure on a possibly remote server from a client by wrapping communication over the network with wrapper stub functions. This six steps are:
 1. The client calls the client stub. The call is a local procedure call, with parameters pushed on to the stack in the normal way.
 2. The client stub packs the parameters into a message and makes a system call to send the message. Packing the parameters is called marshaling.
 3. The client's local operating system sends the message from the client machine to the server machine.
 4. The local operating system on the server machine passes the incoming packets to the server stub.
 5. The server stub unpacks the parameters from the message. Unpacking the parameters is called unmarshaling.
 6. Finally, the server stub calls the server procedure. The reply traces the same steps in the reverse direction
- **TCP** - Transmission Control Protocol (TCP) is a common L4 (transport layer) protocol that guarantees reliable in-order delivery. In-order delivery is accomplished through the use of sequence numbers attached to every data packet, and reliable delivery is accomplished through the use of ACKs (acknowledgements).
- **Consistent Hashing** - A technique for assigning a K/V Pair to a node. With consistent hashing, a new node can be added to a DHT while only moving a fraction (K/N) of the total keys. With consistent hashing Nodes are placed in the key space. A node is responsible for all the keys less than it, but greater than its predecessor. When a new node joins, it copies its necessarily data from its successor.

- **Chord** - Chord is a distributed lookup protocol for efficiently resolving the node corresponding to a key in a DHT. Chord uses a finger table which contains pointers to exponentially further nodes provide $\log(N)$ lookup time. It periodically updates the fingertable to provide for eventual consistency.



2 RPC

To explore the process for performing RPC we will consider implement the server end for two procedures. We want to implement the server side of an RPC version of the following code

```
// Returns the ith prime number (0 indexed)
uint32_t ith_prime (uint32_t i);

// Returns 1 if x and y are coprime, otherwise 0.
uint32_t is_coprime(uint32_t x, uint32_t y);
```

Assume the server has already implemented `ith_prime` and `is_coprime` locally.

2.1 Selecting Functions

As a first step we receive data from the client. How do we decide which procedure we are executing? Provide a sample header file addition that could be used to indicate this.

2.2 Reading Arguments

When examining the arguments for your two functions you notice that the arguments require either 8 or 4 bytes, so you believe you can handle either case by attempting to read 8 bytes using the code below.

```
// Assume dest has enough space allocated
void read_args (int sock_fd, char *dest) {
    int byte_len = 0;
    int read_bytes = 0;
    while ((read_bytes = read (fd, dest, 8 - byte_len)) > 0) {
        byte_len += read_bytes
    }
}
```

However when you implement it you notice that for some inputs your server appears to be stuck? Why might this be happening and for which inputs could this happen?

2.3 Handling RPC

Realizing your previous solution was insufficient you decide to implement a slightly more complicated protocol. You settle on the following steps for the client:

1. The client sends an identifier of the function it wants as an integer (0 for `ith_prime`, 1 for `is_coprime`).
2. The client sends all the bytes for all the arguments.

The server then takes the following steps:

1. The server reads the identifier.
2. The server uses the identifier to allocate memory and set the read size.
3. The server reads the remaining arguments.

Complete the following function to implement the server side of handling data. You may find `ntohl` useful.

```
// Function to implement the server side of the protocol.
// Returns whether or not it was successful and closes the socket when finished.
// Assume get_sizes loads all our sizes based on id and call_server_stub selects
// our host function
void receive_rpc (int sock_fd) {
    uint32_t id;
    char *args;
    size_t arg_bytes;
    char *rets;
    size_t ret_bytes;
    int bytes_read = 0;
    int bytes_written = 0;
    int curr_read = 0;
    int curr_write = 0;
    while ((_____
        _____) > 0) {
        bytes_read += curr_read;
    }
    _____;
    // Allocates sizes based on our id (not a real library function)
    get_sizes (id, &args, &arg_bytes, &rets, &ret_bytes);
    bytes_read = 0;
    while ((_____
        _____) > 0) {
        bytes_read += curr_read;
    }
}
```

```

    }
    // Calls the appropriate server stub based on id (not a real library function)
    call_server_stub (id, args, arg_bytes, rets, ret_bytes)
    while ((-----) > 0) {
        bytes_written += curr_write;
    }
    -----;
}

```

2.4 Call Stubs

Finally we want to implement the call stubs, which are function wrappers between each individual function we support and the generic RPC library. For example these are the client stubs for our functions:

```

// addr is used for setting up our socket connection
uint32_t ith_prime_cstub (struct addrinfo *addr, uint32_t i) {
    uint32_t prime_val;
    i = htonl(i);
    call_rpc (addr, RPC_PRIME, (char *)&i, 1, (char *) &prime_val, 1);
    return ntohl(prime_val);
}

uint32_t is_coprime_cstub (struct addrinfo *addr, uint32_t x, uint32_t y) {
    uint32_t coprime_val;
    uint32_t args[2] = {htonl(x), htonl(y)};
    call_rpc (addr, RPC_COPRIME, (char *) args, 2, (char *) &coprime_val, 1);
    return ntohl(coprime_val);
}

```

`call_rpc` is our generic rpc handler that our stubs provide an abstraction around. Notice that we marshal arguments with `htonl()` and unmarshal them with `ntohl()`.

After the raw data is processed we need to perform the actual computation and return the result to the client. To do this we introduce a stub procedure which unpacks our arguments, calls the procedure to execute on the server, and finally packs the return results to give to the transport layer.

Implement `ith_prime_sstub` and `is_coprime_sstub` which should unmarshal the arguments, call the implementation functions, and finally marshal the return data.

```

// Both functions take in args in network order and should place
// return value(s) in rets in network order
void ith_prime_sstub (char *args, char *rets) {
    -----
    -----
    -----
    -----
    -----
}

void is_coprime_sstub (char *args, char *rets) {
    -----
    -----
    -----
    -----
}

```

```
}  
  
-----  
-----  
-----  
-----
```

2.5 Handling failure

What are some ways a remote procedure call can fail? How can we deal with these failures?

3 Distributed File Systems

Distributed file systems must provide the same access API as standard file systems. That access API quickly becomes non standard in the face of concurrent operations.

Compare the performance of AFS, NFS, EXT4, and EXT4 with the streaming api. Assume processes run on separate machines wherever it is applicable.

1. **open**

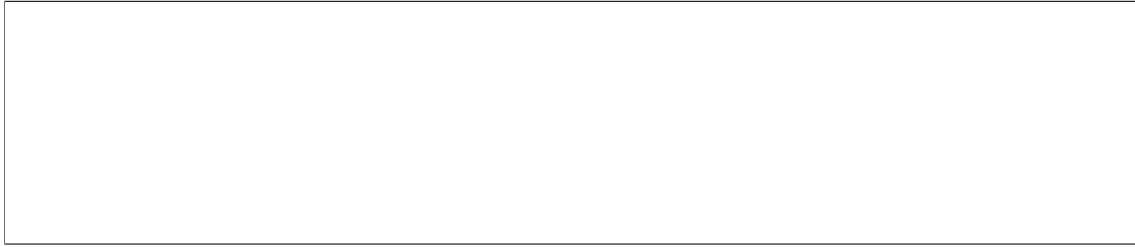
2. **write**

3. **read**

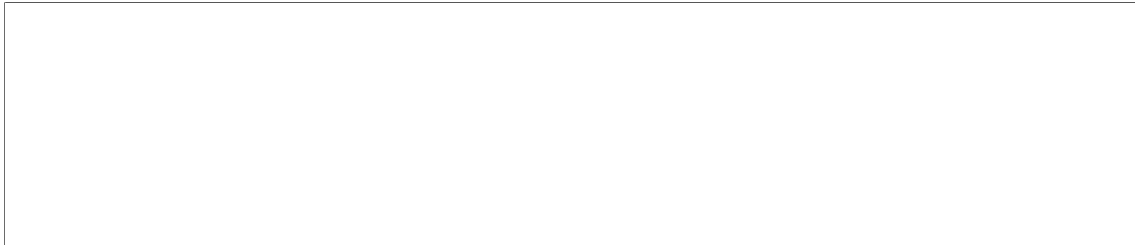
4. **close**

Now compare the behavior of AFS, NFS, EXT4, and EXT4 with the streaming api (with a large buffer). Assume processes run on separate machines wherever it is applicable and that buffers are only flushed when filled and upon close.

1. Process A and Process B write to a file simultaneously, then Process A closes the file, then Process B closes the file.



2. Process A increments an integer (using read and write), 1 second later, Process B increments the integer too. Both processes close the file.



3. Process A increments an integer (using read and write), 1 minute later, Process B increments the integer too. Both processes close the file.



4. Process A writes a large amount of data, then Process B writes a large amount of data. Then Process B closes the file, then Process A closes the file.



4 Distributed Key-Value Stores

- a) Consider a distributed key-value store using a directory-based architecture.

Keys are 256 bytes, values are 128 MiB, each machine in the cluster has a 8 GiB/s network connection, and the client has a unlimited amount of bandwidth. The RTT between the directory and data machines is 2ms and the RTT between the client and directory/data nodes is 128ms.

- i) How long would it take to execute a single GET request using a recursive query?

- ii) How long would it take to execute 2048 GET requests using recursive queries?

- iii) How long would it take to execute a single GET request using an iterative query?

- iv) How long would it take to execute 2048 GET requests using an iterative query?

- v) Now imagine our client is located in the same datacenter, and the RTT between all components is the same (this is a common assumption when modeling datacenter topology).

Briefly describe how your results would change.

- vi) What are some advantages and disadvantages to using a recursive query system?

vii) What are some advantages and disadvantages to using an iterative query system?

- b) In a distributed key-value store, we need some way of hashing our keys in order to roughly evenly distribute them across our servers. A simple way to do this is to assign key K to server i such that $i = \text{hash}(K) \bmod N$, where N is the number of servers we have. However, this scheme runs into an issue when N changes — for example, when expanding our cluster or when machines go down. We would have to re-shuffle all the objects in our system to new servers, flooding all of our servers with a massive amount of requests and causing disastrous slowdown. Propose a hashing scheme (just an idea is fine) that minimizes this problem.

- c) Consider a distributed key value store, in which for each KV pair, that pair is stored on a single node machine, and we use iterative querying.

(a) Describe some limitations of this system. In particular, focus on bandwidth and durability.

(b) Propose some strategies for overcoming these limitations.

5 Networking

- a) (True/False) IPv4 can support up to 2^{64} different hosts.

- b) (True/False) Port numbers are in the IP header.

- c) (True/False) UDP has a built in abstraction for sending packets in an in order fashion.

- d) (True/False) TCP provide a reliable and ordered byte stream abstraction to networking.

- e) (True/False) TCP attempts to solve the congestion control problem by adjusting the sending window when packets are dropped.

- f) In TCP, how do we achieve logically ordered packets despite the out of order delivery of the physical reality? What field of the TCP packet is used for this?

- g) Describe how a client opens a TCP connection with the server. Elaborate on how the sequence number is initially chosen.

- h) Describe the semantics of the acknowledgement field and also the window field in a TCP ack.

- i) List the 5 layers specified in the TCP/IP model. Layering adds modularity to the internet and allows innovation to happen at all layers largely in parallel. What is the function of each layer?

j) The end to end principle is one of the most famed design principles in all of engineering. It argues that functionality should **only** be placed in the network if certain conditions are met. Otherwise, they should be implemented in the end hosts. These conditions are:

- Only If Sufficient: Don't implement a function in the network unless it can be completely implemented at this level.
- Only If Necessary: Don't implement anything in the network that can be implemented correctly by the hosts.
- Only If Useful: If hosts can implement functionality correctly, implement it in the network only as a performance enhancement.

Take for example the concept of reliability: making all efforts to ensure that a packet sent is not lost or corrupted and is indeed received by the other end. Using each of the three criteria, argue if reliability should be implemented in the network.

i) Only If Sufficient

ii) Only If Necessary

iii) Only If Useful