Section 12: Networking, 2PC, RPC

CS 162

November 20, 2020

Contents

1 Vocabulary

- **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).
- **Primary/Secondary** or **Coordinator/Worker**. A scheme for separation of responsibilities. In this scheme, there is typically a single active primary and one or more secondaries. The primary is typically responsible for being the source of truth and directing operations towards the secondaries.
- **Failover** A fault tolerance procedure invoked when a component fails. Typically this involves switching over to, or promoting a secondary.
- **2PC** - Two Phase Commit (2PC) is an algorithm that coordinates transactions between one coordinator and many workers. Transactions that change the state of the worker are considered 2PC transactions and must be logged and tracked according to the 2PC algorithm. 2PC ensures atomicity and durability by ensuring that a write happens across ALL replicas or NONE of them. The replication factor indicates how many different workers a particular entry is copied among. The sequence of message passing is as follows:

```
for every worker replica and an ACTION from the coordinator,
origin [MESSAGE] -> dest :
---
COORDINATOR [VOTE-REQUEST(ACTION)] -> WORKER
WORKER [VOTE-ABORT/COMMIT] -> COORDINATOR
COORDINATOR [GLOBAL-COMMIT/ABORT] -> WORKER
WORKER [ACK] -> COORDINATOR
```

  If at least one worker votes to abort, the coordinator sends a GLOBAL-ABORT. If all worker vote to commit, the coordinator sends GLOBAL-COMMIT. Whenever a coordinator receives a response from a worker, it may assume that the previous request has been recognized and committed to log and is therefore fault tolerant. (If the coordinator receives a VOTE, the coordinator can assume that the worker has logged the action it is voting on. If the coordinator receives an ACK for a GLOBAL-COMMIT, it can assume that action has been executed, saved, and logged such that it will remain consistent even if the worker dies and rebuilds.)

- **Endianness** - The order in which the bytes are stored for integers that are larger than a byte. The two variants are big-endian where byte 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

2 Networking

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

False, it has 32 bits per IP address

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

False, they are in the transport layer. (TCP/UDP).

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

False, this is a part of the TCP protocol. In UDP there is no notion of a sequence number.

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

True.

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

True.

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?

The seqno field.

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

3 way handshake. Client sends a random sequence number (x) in a syn packet. Server sees this and sends another random seqeuence number back (y) in addition to acknowledging the sequence number that it received from the client (sends back x+1) in a syn-ack packet. Client acknowledges this sequence number in an ack packet by sending back y+1.

It is important to randomize the sequence number so an off path attacker cannot guess it and send spurious packets to you.

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

The acknowledgement field says that the receiver has received all bytes up until that number (x). The window field says how many additional bytes past x the receiver is ready to receive.

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?

1) Physical Layer: the physical layer is responsible for the delivery of raw bits from one endpoint to another. It includes ethernet, fiber, and other mediums of data transmission.
2) Datalink Layer: this layer adds the packet abstraction and is responsible for the local delivery of packets between devices within the same network (usually a LAN but can also include WANs). Devices here include switches, bridges, and network interface cards (NICs).
3) Network Layer: this layer is the skinny waist of the internet and routing algorithms here only speak IP. The network layer is responsible for global delivery of packets across one or more

(f) note:
Sender side: Each byte of data is assigned a sequence number. The TCP header includes the sequence number of the first byte in the segment.
Receiver side: The receiver uses the sequence numbers to place incoming data in the correct order in a reassembly buffer. If a segment arrives out of order, it is held until all earlier data arrives.
Acknowledgments: The receiver sends ACKs (using the Acknowledgment Number field) to confirm receipt of consecutive data up to a certain point, ensuring reliability and enabling retransmission if needed.

networks.

4) Transport Layer: this layer provides host to host or end to end communication. Because processes operate in terms of streams of data rather than individual packets, the transport layer handles the multiplexing of packets into streams, end to end reliable delivery, and introduces the concept of flows. This layer lives in the operating system, and TCP and UDP are examples of popular transport layer protocols.

5) Application Layer: the application layer provides network support for applications. The protocols that live here include: HTTP, FTP, DNS, SSH, TLS, etc.

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

NO. It is not sufficient to implement reliability in the network. The argument here is that a network element can misbehave (i.e. forwards a packet and then forget about it, thus not making sure if the packet was received on the other side). Thus the end hosts still need to implement reliability, so it is not sufficient to just have it in the network.

ii) Only If Necessary

NO. Reliability can be implemented fully in the end hosts, so it is not necessary to have to implement it in the network.

iii) Only If Useful

Sometimes. Under circumstances like extremely lossy links, it may be beneficial to implement it in the network. Lets say a packet crosses 5 links and each link has a 50% chance of losing the packet. Each link takes 1 ms to cross and there is an magic oracle tells the sender the packet was lost. The probability that a packet will successfully cross all 5 links in one go is $(1/2)^5 = 3.125\%$. This means the end hosts need to try 32 times before it expects to see the packet make it through, taking up to # of tries $\times$ max # of links per try $= 32 \times 5 =$ 160 ms. Likewise at each hop, if the router itself is responsible for making sure the packet made it to the next router, each router would know if the packet was dropped on the link to the next router. Thus each router only has to send the packet until it reaches the next router, which will be twice on average. So to send this packet, it will take on average # of tries per link # number of links $= 2 \times 5 = 10$ ms. This is a huge boost in performance, which makes it useful to implement reliability in the network under some cases.

3 Two-Phase Commit

Quorum consensus is able to provide weak consistency. For this section, assume the DHT backs each node with multiple replicas. Further, assume that these machines use a two phase commit protocol to commit information in a strongly consistent manner.

1. 2PC requires a single coordinator and at least one worker. By default, all replicas can be workers. How should the coordinator be picked?

   The general problem here is leader election - the bootstrap process of choosing and agreeing a leader in distributed systems.

   Naively, we could pick a single machine to be the coordinator for all transactions and hard code that machine as the leader. If the leader crashes, then we must wait for that leader to restart before the system can make any forward progress.

   Modern leader election algorithms do not rely on a hard coded leader but rather allow the workers to choose who they want as their leader. In the event that the leader fails (as detected by some form of health checking), the subset of the machines that are still connected would rerun the algorithm and find a new leader.

   This provides better availability because a new leader can be choose on the fly, but it also introduces further complexities. For example what if there was a network partition and a single cluster of works split into two where each one elects their own leader. Then the partition heals how do you deal with the fact that there are now two leaders.

   Additionally in any situation where we choose a single leader, we would be bottle necked by its upload bandwidth, since it would have to send the request to all replicas. To mitigate this problem, we could pick a leader at random for each request. This would spread resource usage for the dominant resource (upload bandwidth).

   Note we typically don't consider consider GFS style chaining. It doesn't fit in the byzantine generals model, and may not be fast if we can't do a shortest hop tour (in the event of a failed machine).

2. Briefly describe the messages that the **coordinator** will send and receive in response to a PUT. Also describe when and what would be logged.

   - RECEIVE (from client): A PUT query
   - LOG (to journal): PREPARE query
   - SEND (to workers): N PREPARE PUT messages
   - RECEIVE (from workers): COMMIT or ABORT per worker
   - LOG (to journal): GLOBAL-COMMIT or GLOBAL-ABORT
   - SEND (to workers): GLOBAL-COMMIT if it receives COMMIT from all workers, otherwise GLOBAL-ABORT
   - RECEIVE (from workers): ACK
   - SEND (to client): SUCCESS if it sent GLOBAL-COMMIT, otherwise FAIL.

3. Briefly describe the messages that a **worker** will send and receive.

   - RECEIVE (from coordinator): A PREPARE PUT
   - LOG (to journal): PREPARE query

- SEND (to coordinator): COMMIT or ABORT
- RECEIVE (from coordinator): GLOBAL-COMMIT or GLOBAL-ABORT
- LOG (to journal): GLOBAL-COMMIT or GLOBAL-ABORT
- SEND (to coordinator): ACK

4. Under the current model, multiple PUT queries could be sent to separate coordinator machines. Propose set of worker routines which can handle this. In particular, consider the case in which 2 coordinators receive PUT queries for **the same key but different values**. The state of the system should remain consistent.

Upon receiving a PREPARE PUT request, `try_acquire` a lock on the key.

If acquire fails, ABORT, otherwise COMMIT.

Upon receiving a GLOBAL-COMMIT, set the value then release the lock. Release the lock without setting the value on a GLOBAL-ABORT.

The state of the system will always be consistent now. Note that a potentially unintuitive outcome of 2PC here is that both queries could fail here, leaving the key empty despite a PUT request.

One potential solution to this problem (which is out of scope) while maintaining strong consistency could include using PAXOS to determine an order of these queries, and accept only the first.

Note: Many popular DHTs in the real world take different approaches to handling this edge case. Most of these DHTs are eventually consistent.

Phase 1: Preparation Stage (PREPARE)
When a coordinator receives a PUT (key, value) request:
1. Send PREPARE PUT (key, value, transaction ID) to all relevant nodes.
2. Upon receiving PREPARE, work nodes:
   -Try to acquire the lock for the key (try_acquire)
   -If successful, return COMMIT (indicating proceed)
   -If failed, return ABORT (indicating conflict and abort).

Phase 2: Global Decision Stage
The coordinator collects responses from all work nodes:
-If all nodes return COMMIT: Send GLOBAL-COMMIT
-If any node returns ABORT: Send GLOBAL-ABORT

Work nodes receive final instructions:
-Upon receiving GLOBAL-COMMIT: Set value + release lock
-Upon receiving GLOBAL-ABORT: Directly release lock (no value set)

4 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.

4.1 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.

We need to provide a unique id for each function and transmit it in at the start of our RPC message. There are many ways to do this, for example if these two functions were a comprehensive list of the rpc calls we needed to support we could provide an enum or we could use defines with unique values for example.

```
enum RPC_ID {
    PRIME_ID = 1,
    COPRIME_ID = 2
};
```

4.2 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?

The read is trying to return 8 bytes from the socket or indicate there is no more data. However, when the socket is empty the read will only return due to failure (or if the socket is closed). So long as the connection stays open the reads will never fail and will block instead.

Read() behavior in blocking mode:
1. When the socket contains no data, read() does not return 0 unless the connection is closed.
2. It will remain blocked until the data arrives.
3. This is the default behavior for blocking I/O

4.3 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 ((curr_read = read (sock_fd, ((char *)&id) + bytes_read,      Read function ID
            sizeof (uint32_t) - bytes_read)) > 0) {
        bytes_read += curr_read;
    }
    id = ntohl (id_data);      id = ntohl (id); // Convert the network byte order (big-endian) ID
                                to host byte order
    get_sizes (id, &args, &arg_bytes, &rets, &ret_bytes);
    bytes_read = 0;
    while ((curr_read = read (sock_fd, &args[bytes_read], arg_bytes - bytes_read)) > 0) {
        bytes_read += curr_read;                               Read the function arguments
    }
    call_server_stub (id, args, arg_bytes, rets, ret_bytes)
    while ((curr_read = write (sock_fd, &rets[bytes_written],
            arg_bytes - bytes_written)) > 0) {       Write the return values
        bytes_written += curr_write;
    }
    close (sock_fd);
}
```

4.4 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:

```
// addrs is used for setting up our socket connection
uint32_t ith_prime_cstub (struct addrinfo *addrs, uint32_t i) {
    uint32_t prime_val;
    i = htonl(i);
    call_rpc (addrs, RPC_PRIME, (char *)&i, 1, (char *) &prime_val, 1);
    return ntohl(prime_val);
}
uint32_t is_coprime_cstub (struct addrinfo *addrs, uint32_t x, uint32_t y) {
    uint32_t coprime_val;
    uint32_t args[2] = {htonl(x), htonl(y)};
    call_rpc (addrs, 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.

```
void ith_prime_sstub (char *args, char *rets) {
    uint32_t* args_ptr = (uint32_t *) args;
    uint32_t i = ntohl (args_ptr[0]);
    uint32_t result = ith_prime (i);
    result = htonl (result);
    memcpy (rets, &result, sizeof (uint32_t));
}
void is_coprime_sstub (char *args, char *rets) {
    uint32_t* args_ptr = (uint32_t *) args;
    uint32_t x = ntohl (args_ptr[0]);
    uint32_t y = ntohl (args_ptr[1]);
    uint32_t result = is_coprime (x, y);
    result = htonl (result);
    memcpy (rets, &result, sizeof (uint32_t));
}
```

4.5 Handling failure

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

1. Transport failure (i.e. the client can fail to connect to the server, or fail to send data to the server). This can be handled via retrying (the case of dropped packets is handled by reliable transport).

2. The server could crash/error before completing the procedure. *This means that all the client*

sees is a severed connection. In this case an ACID compliant transaction will never complete, and it's safe to retry the request.

3. The server could crash have a transport error after completing the procedure, but before the client receives the results. *This means that all the client sees is a severed connection.* If we retry now, we have done the transaction twice! This is a good case for using idempotent operations.

4. (Alternative) The server could send the RPC result back to the client before committing the procedure. The client would have to tell the server when it has received the results. What if the server doesn't receive that message? This is the simple case of the General's Paradox.

   A distributed design note: Notice that the complicated error handling here comes from stateful modifications. This is why, when designing a distributed system, idempotent operations are preferable when modifying state, and read-only operations are preferable to operations which modify state. Stateless operations (pure functions) are typically preferable to read-only operations (but this is more typically for performance and consistency reasons).