Discussion 10: Distributed Systems

April 22, 2022

Contents

1 Distributed Systems **2**
1.1 Concept Check 2
1.2 Two Phase Commit 3
1.3 Primes 3

1 Distributed Systems

1.1 Concept Check

1. The vanilla implementation of 2PC logs all decisions. How could 2PC be optimized to reduce logging?

   Abort decisions can be ignored and not logged with the idea of presumed abort. On recovery, if there is nothing logged, then we can simply assume that the decision made was to abort.

2. 2PC exhibits blocking behavior where a worker can be stalled until the coordinator recovers. Why is this undesirable?

   If a worker is blocked on this coordinator, then it may be holding resources that other transactions may need.

3. An interpretation of the End to End Principle argues that functionality should only be placed in the network if certain conditions are met.

   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.

   Consider the example of the reliable packet transfer: 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.

   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.

   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.

   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.

4. Why would you ever want to use UDP over TCP?

UDP is used in applications that prioritize speed and low overhead, and either don't care about being "lossy" or implements their own protocol for reliability. For example, in streaming audio or video, it doesn't matter if some packets are lost or corrupted, as long as most of the packets are sent, the user will still be able to hear/see the data well enough. Another example would be any application where real time data is very important (e.g. real time news, weather, stock price tracking, etc), and we don't have time for anything beyond best effort delivery.

1.2 Two Phase Commit

Consider a system with one coordinator (C) and three workers (W_1, W_2, W_3). The following latencies are given for each worker.

Worker	Send/Receive (each direction)	Log
W_1	400 ms	10 ms
W_2	300 ms	20 ms
W_3	200 ms	30 ms

You may assume all other latencies not given are negligible. C has a timeout of 3 s, log latency of 5 ms, and can communicate with all workers in parallel.

1. What is the minimum amount of time needed for 2PC to complete successfully?

For each phase, the worker needs to receive the message, log a result, and send back a message. Since each worker can operate in parallel, only the longest latency matters. Using the latencies given, W_1 has the longest round trip time latency with $400 + 10 + 400 = 810$ ms. Therefore, the two phases will require 1620 ms. However, we also need to add in the time that it takes for the coordinator to log the global decision, so the minimum amount of time is 1625 ms. The reason this is the minimum amount of time is because this assumes no failures.

2. Consider that all three workers vote to commit during the preparation phase. The coordinator broadcasts a commit decision to all the workers. However, W_2 crashes and does not recover until immediately after the coordinator's timeout phase. Does this transaction commit or abort? What is the latency of this transaction, assuming no further failures?

The transaction still commits since a commit decision was made by the coordinator. The preparation phase will take 810 ms as calculated before, and the commit decision needs to be logged which takes 5 ms. The first try of the commit phase will take 3000 ms (i.e. the timeout). The second try will only take $300 + 20 + 300 = 620$ ms because W_2 is the only worker that needs to commit; all other workers succeeded on the first try. In total, the latency of this transaction is $810 + 5 + 3000 + 620 = 4435$ ms.

1.3 Primes

Edward wants to build a RPC system for the following two procedures.

```
/* Returns the ith prime number (0-indexed). */
```

```
uint32_t ith_prime(uint32_t i);

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

In particular, Edward is looking to setup the server side. You may assume the actual logic for both procedures have already been implemented.

1. Edward notices that the arguments required are either 4 or 8 bytes. As a result, he believes he can handle either case by attempting to read 8 bytes with 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(sock_fd, dest, 8 - byte_len)) > 0) {
       byte_len += read_bytes;
     }
   }
   ```

   However when Edward implements it, he notices that for some inputs the 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, the socket will not return 0 unless the socket is closed. As long as the connection stays open, the `read` will block instead.

2. Edward realizes his previous solution was insufficient, so he decide to implement a slightly more complicated protocol.

   The client will perform the following.

   (a) Send an identifier for the function it wants as an integer (0 for `ith_prime`, 1 for `is_coprime`).

   (b) Send all bytes for all the arguments.

   The server will then perform the following.

   (a) Read identifier.

   (b) Use identifier to allocate memory and set read size.

   (c) Read arguments.

   ```
   /* Converts NETLONG from network byte order to host byte order. */
   uint32_t ntohl(uint32_t netlong);

   /* Converts NETLONG from host byte order to network byte order. */
   uint32_t htonl(uint32_t hostlong);

   void receive_rpc (int sock_fd) {
     /* Read in procedure identifier. */
     uint32_t id;
     int bytes_read = 0, cur_read = 0;
     while (_____________) {
       bytes_read += cur_read;
     }
     id = _____________;
   ```

```
  /* Get sizes and allocate space for arguments and return values. */
  char *args, *rets;
  size_t arg_bytes, ret_bytes;
  get_sizes(id, &args, &arg_bytes, &rets, &ret_bytes);

  /* Read in arguments. */
  bytes_read = 0;
  while (_________________) {
    bytes_read += cur_read;
  }

  /* Call appropriate server stub stub function based on id. */
  call_server_stub(id, args, arg_bytes, rets, ret_bytes);

  /* Write return values. */
  int bytes_written = 0, cur_written = 0;
  while (_________________) {
      bytes_written += cur_written;
  }

  /* Clean up socket. */
  _________________;
}
```

```
void receive_rpc (int sock_fd) {
  /* Read in procedure identifier. */
  uint32_t id;
  int bytes_read = 0, cur_read = 0;
  while ((cur_read = read(sock_fd, ((char *) &id) + bytes_read,
                          sizeof(uint32_t) - bytes_read)) > 0) {
    bytes_read += cur_read;
  }
  id = ntohl(id);

  /* Get sizes and allocate space for arguments and return values. */
  char *args, *rets;
  size_t arg_bytes, ret_bytes;
  get_sizes(id, &args, &arg_bytes, &rets, &ret_bytes);

  /* Read in arguments. */
  bytes_read = 0;
  while ((cur_read = read(sock_fd, &args[bytes_read],
                          arg_bytes - bytes_read)) > 0) {
    bytes_read += cur_read;
  }

  /* Call appropriate server stub stub function based on id. */
  call_server_stub(id, args, arg_bytes, rets, ret_bytes);

  /* Write return values. */
  int bytes_written = 0, cur_written = 0;
  while ((cur_written = write(sock_fd, &rets[bytes_written],
                            ret_bytes - bytes_written)) > 0) {
```

```
        bytes_written += cur_written;
    }

    /* Clean up socket. */
    close(socket_fd);
  }
```

3. Finally, Edward want to implement the call stubs, which are function wrappers between each individual function we support and the generic RPC library. The following are the *client* stubs for our procedures.

```
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);
}
```

Implement the server stubs for our procedures.

```
void ith_prime_sstub (char *args, char *rets) {
  ______________________________;
  ______________________________;
  ______________________________;
  ______________________________;
}
void is_coprime_sstub (char *args, char *rets) {
  ______________________________;
  ______________________________;
  ______________________________;
  ______________________________;
  ______________________________;
}
```

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