Discussion 10: Distributed Systems

April 22, 2022

Contents

1 Distributed Systems

1.1 Concept Check

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

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

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.

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

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?

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?

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?

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. */
  ________________;
}
```

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) {
  ______________________________;
  ______________________________;
  ______________________________;
  ______________________________;
  ______________________________;
}
```