Distributed Systems

How to build systems for components that fail?

COM S 352

Iowa State University

Matthew Tancreti

Distributed Systems

Central tenet of networking: communication is unreliable

- Bits get flipped during transmission
- Routers may not work correctly
- Network cables get accidentally cut
- Packets arrive too quickly at a router and overfill its buffer space
- …

Packet loss is fundamental in networking: how to deal with it?

How to build systems for components that fail?

Padlock [source]

Networks are Unreliable

Central tenet of networking: communication is unreliable

- Bits get flipped during transmission
- Routers may not work correctly
- Network cables get accidentally cut
- Packets arrive too quickly at a router and overfill its buffer space
- …

Packet loss is fundamental in networking: how to deal with it?

Option 1: Make the Application Deal with Failure

One option is to not deal with packet loss

UDP/IP networking stack only allows application to send/receive individual packets (datagrams)

Provides no guarantee to application that packets will not be lost or arrive in different order than sent

Simple UDP Library Example

```
int UDP_Write(int sd, struct sockaddr_in *addr,
  char *buffer, int n) {
  int addr_len = sizeof(struct sockaddr_in);
  return sendto(sd, buffer, n, 0, (struct sockaddr *)
                addr, addr_len);
}

int UDP_Read(int sd, struct sockaddr_in *addr,
             char *buffer, int n) {
  int len = sizeof(struct sockaddr_in);
  return recvfrom(sd, buffer, n, 0, (struct sockaddr *)
                  addr, (socklen_t *) &len);
}
```

← Send single packet

← Receive single packet

Figure 48.2: **A Simple UDP Library (udp.c)**

Option 2: Reliable Communication Layer

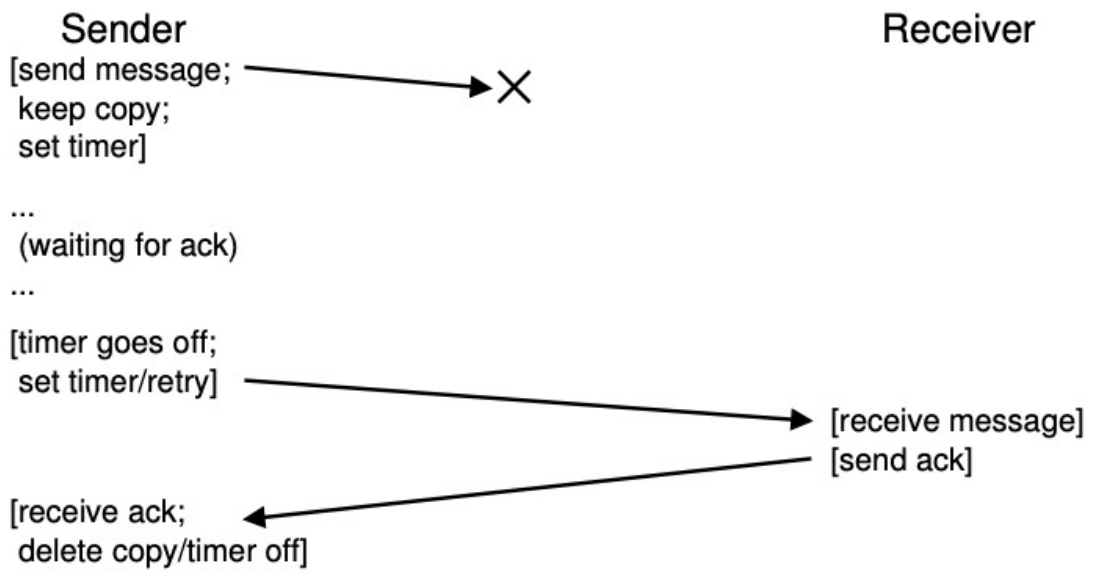
Second option is for transport layer to provide reliability

TCP/IP networking stack allows applications to send/receive full messages

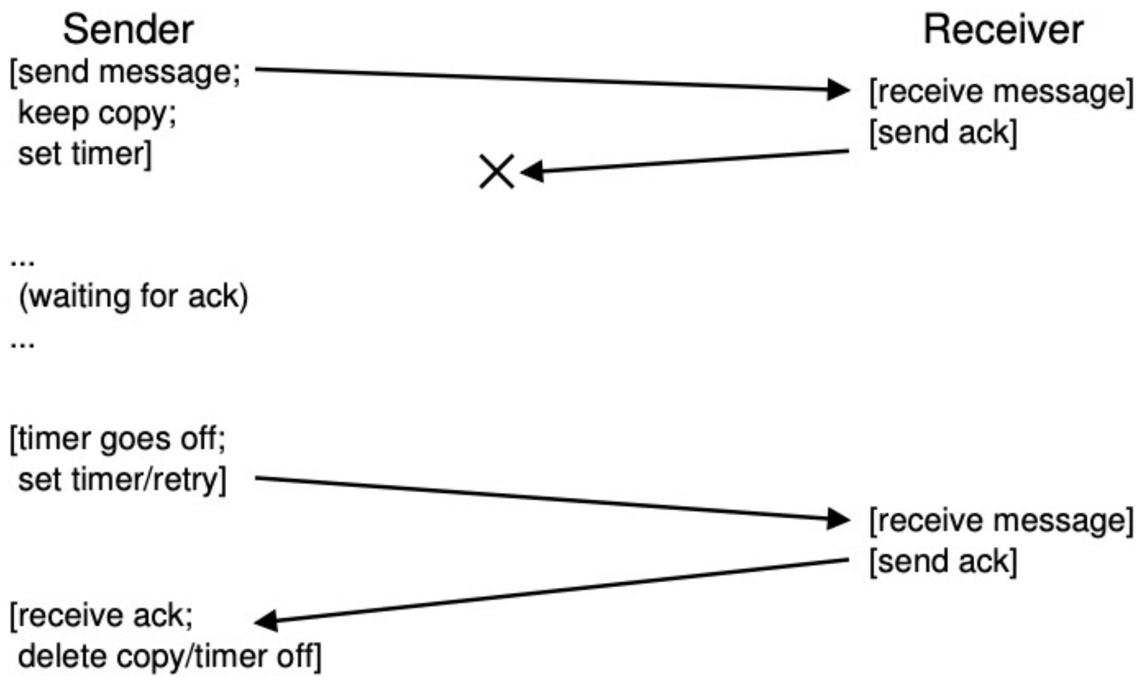
The network stack still breaks the message up into smaller packets, need to deal with packet loss and out of order arrival of packets

Acknowledgement with Timeout

Acknowledgement (ack) is technique where sender expects an acknowledgement within some amount of time, if there is a **timeout** the sender **retries** sending



Acknowledgement with Dropped Replay



Sequence Number

Sender needs to know which packet was lost

Also want to prevent packets being added to messages multiple times

Sender attaches a **sequence number** (counter) to each packet

Receiver also keep a count of received sequence numbers, sends sequence number with acknowledgement and checks to make sure packets are not provided to application twice

Remote Procedure Call

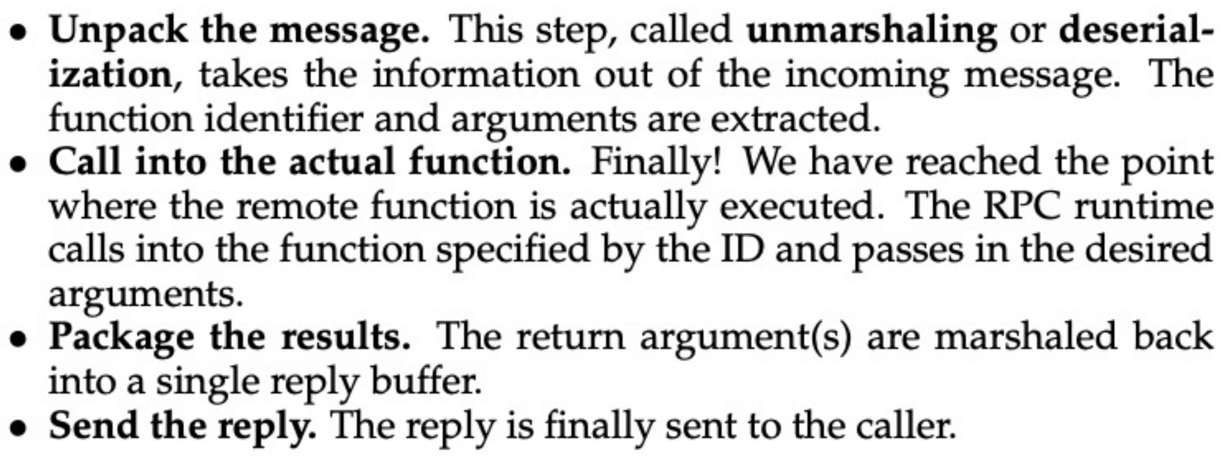
Remote Procedure Call (RPC) have goal of making the process of executing code on a remote machine as simple and straight forward as calling a local function

A stub generator is used to remove to pain of packing function arguments and results into messages

Stub Generator Actions (Client Side)

- **Create a message buffer.** A message buffer is usually just a contiguous array of bytes of some size.
- **Pack the needed information into the message buffer.** This information includes some kind of identifier for the function to be called, as well as all of the arguments that the function needs (e.g., in our example above, one integer for `func1`). The process of putting all of this information into a single contiguous buffer is sometimes referred to as the **marshaling** of arguments or the **serialization** of the message.
- **Send the message to the destination RPC server.** The communication with the RPC server, and all of the details required to make it operate correctly, are handled by the RPC run-time library, described further below.
- **Wait for the reply.** Because function calls are usually **synchronous**, the call will wait for its completion.
- **Unpack return code and other arguments.** If the function just returns a single return code, this process is straightforward; however, more complex functions might return more complex results (e.g., a list), and thus the stub might need to unpack those as well. This step is also known as **unmarshaling** or **deserialization**.
- **Return to the caller.** Finally, just return from the client stub back into the client code.

Stub Generator Actions (Sever Side)



- **Unpack the message.** This step, called **unmarshaling** or **deserialization**, takes the information out of the incoming message. The function identifier and arguments are extracted.
- **Call into the actual function.** Finally! We have reached the point where the remote function is actually executed. The RPC runtime calls into the function specified by the ID and passes in the desired arguments.
- **Package the results.** The return argument(s) are marshaled back into a single reply buffer.
- **Send the reply.** The reply is finally sent to the caller.