Computer Networks

2018A7PS0193P

February 18, 2021

1 Networks

Definition 1.1. A **network** is a shared infrastructure that allows users to communicate with each other.

The basic building blocks of a network are **nodes** and **links**. Nodes may be hosts or forwarding nodes. The end hosts communicate with one another through the **core network**, consisting of forwarding nodes. These nodes are connected via links called **network edges**.

End hosts are physically connected to the core network via the **access network**. This consists of many parts, such as the ethernet switch, router, etc.

1.1 Communication Models

Networks could have multiple communication models:

- **Client-Server model**: The client host requests, and receives the service from an always-on server. This server is also an end host, but has some special privileges.
- **Peer-peer model**: Here, there is minimal or no use of dedicates servers, as in Bit-Torrent. The clients directly communicate with one another.

1.2 Core Network Models

1.2.1 Circuit Switching

How is our core network made? One way to do this is via **circuit switching**. There are end-to-end resources reserved for a "call", like on a telephone network. There is no sharing of resources. Call setup needs to be done as a preparatory step. Circuit switching generally can be implemented by two different methods:

- **FDM**, which stands for Frequency Domain Multiplexing. The total frequency bandwidth is divided among the users, allowing them to send data simultaneously.
- **TDM**, which stands for Time Domain Multiplexing. Here the time is divided among the users (perhaps in a round robin fashion). As such, one user gets access to the entire bandwidth of the circuit, but only for a period in time.

1.2.2 Packet Switching

Another way is through **packet switching**, where data is sent over the net in discrete "chunks", called packets. This is how it is done on the Internet. The host takes the application message, and breaks into packets of length L bits. It then transmits packets into the access networks at transmission rate R, also called the bandwidth. Of course, this means that each packet faces a transmission delay of L/R.

Packet switching uses **store and forward**. The packets are stored at intermediate nodes before sending to the next node. The intermediate node checks for errors in the packets before transmitting, assuring the integrity of the packets.

When using packet switching, there may be four sources of packet delay:

- **Nodal processing** : The node performs error checking and checks the header for the destination of the packet.
- **Queueing** : When the arrival rate of the packets is faster than the sending rate, the node will keep the packets in a buffer queue. As such, there is a delay when the packet wait in the node queue.
- **Propagation** : This is the delay from propagation of the packets from a node to the next node, i.e., the outgoing delay. This depends on the medium of the wire, and is given by d/s, where d is the length of the connection and s is the speed.
- **Transmission** : This is the delay from transmission of packets into the node, i.e. the incoming delay.

1.3 Performance of a Network

The performance of a network can be measured by the following parameters:

- Delay

- Packet loss : This is the number of packets lost when transmitting. Some applications, like streaming, might not care too much about this.
- Throughput : This is the amount of bits transferred in unit time. This is important in some applications, such as for file transfer.

2 The Internet

The Internet is, in fact, a network of networks. The networks must be able to communicate despite using different applications running on different devices - i.e. it is heterogeneous.

As such, the Internet is full of different access ISP networks. How do end hosts on different access ISPs communicate with one another. Of course, if we directly connect them all, it would not be scalable as it would need $O(N^2)$ connections. We also cannot use a single global hub, since it would be difficult to find a single place to put it and connect the entire world.

Since a single global ISP cannot scale to connect the entire world, we use multiple global ISPs. These must be interconnected themselves. One way to do this is using **peering links**, which directly link two global ISPs. Another is to use **Internet Exchange Points**, called IXPs, to which multiple global ISPs can connect.

The Internet uses this system in a tiered manner - end hosts might connect to a regional ISP, which may then connect to a higher level country ISP, and so on.

Some corporations, like Google, have their own Content Distribution Networks (CDNs), and have their own network to bring services and content closer to users.

2.1 Layered Network Model

The Internet is based on a Layered Network Model known as **OSI**. Any device under OSI can have the following layers:

1. Physical
2. Data Link
3. Network
4. Transport
5. Session

6. Presentation

7. Application

Each of these layers depend on the one below (lower number) and export their services to the ones above (higher number).

The end hosts implement all 7 layers of the model. Those which implement the first 3 layers are called **routers** or Layer 3 devices. Routers are used to connect two different networks. Those which implement the first 2 layers are called **switches** or Layer 2 devices. They connect devices within a network, i.e., in Local Area Networks.

The Internet stack does not actually use all 7 layers - in fact it uses only 5. It removes the Presentation layer, which allows applications to interpret the meaning of data. It also removes the Session layer, which is used for synchronization, check pointing and recovery of data exchange. These functions are generally performed by the Application layer and/or the Transport layer. This Internet stack is known as **TCP/IP model**.

In the context of the Internet, these layers perform the following functions:

1. **Physical:** This layer delivers bits between the two endpoints of a link, e.g. copper, fiber, wireless, etc.

2. **Data Link:** This layer delivers packets between two hosts in a local area network. These are bridges and switches.

3. **Network:** This layer connects multiple networks, e.g. routers. This uses the Internet Protocol (IP).

4. **Transport:** This layer does process-process data transfer. It may use a multitude of protocols, including TCP, UDP, etc.

5. **Application:** This layer supports network applications. It may use FTP, SMTP, HTTP, etc.

2.2 IP Hourglass Architecture

One way to imagine the Internet architecture is as a hourglass. The IP interconnects multiple existing networks, and hides the underlying technology from applications. This provides minimal functionality (has a "narrow waist"). The trade-off of this approach is that there are no assumptions being made, and as such no guarantee that something works.

3 The Application Layer

3.1 Network Applications

A **Network application** is a program that runs on different end systems and communicates over a network. These are run only on the end hosts - core network devices do not run user application code.

The application architecture can run different application architectures:

- **Client-Server**: The server is an "always on" host, which has a permanent IP address. To be able to scale, there are generally large data centers acting as a virtual server. The clients communicate with the server. Unlike the server, they may be intermittently connected, and may have a changing dynamic IP address. These clients never directly communicate with one another.
- **Peer to peer** : Here, there is not always on server. Instead the end hosts directly communicate with one another. The peers are connected and may change IP addresses dynamically.
- **Hybrid of client server and peer to peer** : This is the case in Instant Messaging and Skype. In instant messaging, the chatting between two users is P2P, but to get the IP addresses of a user's friends, a central server is needed.

Processes communicate within the same host using interprocess communication, but they must communicate with different hosts by exchanging messages.

Definition 3.1. A **socket** is the interface between the application later and the transport layer within the host.

A process (a network application) send and receives messages using it's socket. This is a software entity, not a physical one.

To receive messages, each process must have some identifier. The IP address is not enough since it will only uniquely identify the host, but not the individual processes running on the host. So, we also use the port number to identify a process. For instance, an HTTP server would use port number 80, and a mail server would use port number 25.

3.2 Application Transport Services

What transport services does an application need? This changes from application to application. Here are some examples:

- **Data Loss** : Depending on the application, it might be necessary that data transfer is 100% reliable. For instance, in the case of file transfer, there must be no data loss, but some loss can be tolerated in the case of streaming.
- **Bandwidth** : Applications may also have bandwidth requirements. Streaming needs some minimum amount of bandwidth to work, but elastic applications like email and file transfer will make do with whatever bandwidth is available.
- **Timing** : Another parameter to consider is timing. Some applications, like games, require low delay, but for others this is unnecessary.

3.3 HTTP

The application layer for web pages is HTTP. A web page consists of objects, like HTML files, JPEG images, etc. Each of these objects is addressable by a URL. HTTP applications use TCP as it's transfer protocol.

1. Client initiates the TCP connection on port 80. The time taken to establish this is called the **Round Trip Time** or RTT.
2. Client sends a request to the server
3. Server receives message through it's socket and sends the response.
4. Server attempts to close the TCP connection. The client may refuse to do so if the client has not received the message.
5. The client receives the message and closes the connection.

It is important to remember that the time taken to send a file would be $2RTT$ + File Transmission Time - one RTT to establish connection, one RTT to send the first message and the time taken to send the file. Let us say the received file is an HTML file, which as 10 images embedded. Then, the client will request these images as well, by reopening the TCP connection and sending requests.

If at most one object is sent over a TCP connection, it is called **Non-persistent HTTP**. This was the case in HTTP version 1.0. In HTTP version 1.1, **Persistent HTTP** was introduced, which allows multiple objects to be sent over a single TCP connection between client and server. This could be done with or without a pipeline. If there is no pipeline, the client requests for each object, waits for the response, requests the next, and so on. To speed this up, we can use a pipeline and send requests for multiple objects one after another, while

waiting for the response. The responses are always received in the same order in which they were sent.

3.4 HTTP Requests

A HTTP request message consists of:

- A request line. It contains the method (POST,GET,HEAD), the URL being requested, and the HTTP version being used.
- Header lines. Each header line has the header field name and a value. An example of a header line can be `User-Agent: Firefox/3.6.10` or `Keep-Alive:115`.
- The body, which contains all the data of an entity

Each line is delimited by a carriage return character \r and a line feed character `\n`.
The request methods could have the following meanings:

- `GET` : This method is used to transfer from the server to the client
- `POST` : This is used to upload something to the server
- `HEAD` : This asks the server to leave requested object out of response
- `PUT` : This is present in HTTP version 1.1. It uploads a file in entity body to the path specified in the URL field.
- `DELETE` : This is present in HTTP version 1.1. It deletes the file specified in the URL field.

3.5 HTTP Responses

A HTTP response message consists of:

- A status line, which consists of the protocol, a status code and a status phrase. An example is `HTTP/1.1 200 OK`.
- Header lines, which are in the same format as in the request message. It could contain the time, the OS, etc.
- The body, which contains the requested data.

The HTTP response status codes are:

- `200 OK` : Request succeeded, requested object later in this message.
- `301 Moved Permanently` : Requested object moved, new location specified in the Location header line.
- `400 Bad Request` : Request message not understood by the server.
- `404 Not Found` : Requested document not found on this server.
- `505 HTTP Version not supported` : The HTTP version used in the request is not supported by the server.

3.6 States and HTTP

HTTP is a stateless protocol - it does not save anything from it's previous actions. To save the state in HTTP, we use **cookies**. When a client sends a http request to the server, it will create an ID for the user and create an entry in the backend database. Now, when the server sends it's response, it will include a header line called `set-cookie`. From then onwards, the usual HTTP request message will send the cookie ID through a header line called `cookie`.

Hence, using cookies, we can save state with HTTP. This can be used to save user information and track them on other sites.

3.7 Proxy Servers

Proxy servers are also called web caches. Clients send their requests to a proxy server, which would forward it to the destination. The incoming responses are also forwarded to their respective clients. The proxy server can cache information, and hence answer requests itself to save time. This is especially useful in institutions when the content that users access has a large overlap.

One issue with this is that the cached content might turn stale. To fix this we use conditional GETs. When the client sends a request, it adds a header line called `if-modified-since` which gives the last date at which an access was made. If the object has not been modified since that date, the server will give a cached response along with the status code `304 Not Modified`. If it has, then the proxy server will query the updated information and return the status code `200 OK`.

3.8 HTTP/2 Protocol

Definition 3.2. A stream is a bidirectional sequence of text format frames sent over the HTTP/2 protocol exchanged between the server and client.

HTTP/1 was capable of transmitting only one stream at a time. This made receiving large amount of media content inefficient and time consuming. HTTP/2 allows transmission of parallel multiplexed requests and responses. A binary framing layer is created, which allows the client and sever to disintegrate the HTTP payload into small independent and manageable interleaved sequence of frames. This information is then reassembled at the other end.

HTTP/2 also allows the server to send additional cacheable information to the client that isn't requested but is anticipated to be needed in future requests. This mechanism saves a RTT and reduces network latency. This is called **Server PUSH**.

3.9 Domain Name System

When a client requests the URL, it must obtain the IP address of the destination host to send the request to. The **Domain Name System** maps the name people use to locate a website to the IP address that a computer uses to locate a website.

The DNS is formed by a distributed hierarchical database. The 13 **root DNS** servers contain reference to the **top level domain** servers, like the .com servers, the .org servers, etc. These then contain references to the website servers, like the yahoo.com servers, amazon.com servers, etc.

One issue we would face is that since we would need to traverse the hierarchy recursively to query a host, it can take a long time to process a DNS query. In practice, instead of traversing the tree on a direct path, the local DNS server would first get information about the TLD DNS server from the root server, and then contact the TLD server directly, instead of letting the root DNS server do this. This is done recursively, contacting lower and lower levels in the hierarchy directly from the local DNS server and getting the IP of the next level.

DNS responses are generally cached to improve the delay performance and to reduce the number of DNS messages.

DNS provides the following services:

- Host name to IP address mapping
- Host aliasing

- Mail server aliasing
- Load distribution

The DNS servers store the necessary information in the form of resource records(RR). They are in the format `(name, value, type, ttl)`. Here `ttl` stands for "time to live", which tells the DNS resolver how long to cache a query before requesting a new one. The values of the other fields depends on the type of the resource record. There are four types of resource records, depending on the `type`:

- `type=A` : `name` refers to the host name, and `value` refers to the IP address.
- `type=NS` : `name` is the domain name, and `value` is the host name of authoritative name server for this domain.
- `type=CNAME` : `name` is the alias name for some canonical name, `value` is the canonical name.
- `type=MX` : `value` is the name of the mail server associated with the `name`.

To insert a record into DNS, a newly created domain name should be first registered at a registrar.

3.10 File Transfer Protocol

This protocol is used to transfer file from client to client. It uses the TCP protocol, since it must be reliable and error free.

In a typical session, FTP will use two connections - a control connection and a data connections. The control connection, which is generally established on port 21, is the primary connection and is used to send commands back and forth between the client and the server. The data connection, established on port 20, is used solely to transfer the requested data. It stays open until the transfer is complete.

Some FTP commands are:

- `USER username`
- `PASS password`
- `LIST` returns list of file in current directory
- `RETR filename` retrieves file

- `STOR filename` stores file onto remote host

These commands can give return codes and phrases as in HTTP:

- `331 Username OK, password required`
- `125 data connection already open; transfer starting`
- `425 can't open data connection`
- `452 Error writing to file`

3.11 eMail

e-Mail consists of three major components

- User agents, e.g. Outlook, mutt
- Mail servers, Contains incoming messages for user
- Simple Mail Transfer Protocol (STMP)

A user creates a mail and sends it to their mail server using the user agent. The sender's mail server will forward this mail to the recipient's mail server over a TCP connection on port 25. The recipient's user agent will then try to access this mail from the server. The protocol used in this case is called **Mail Access Protocol**, like IMAP or POP3. Everywhere else in this process, we use SMTP. We cannot use SMTP for the recipient to access the mail, since SMTP is a push-based protocol, not a pull based protocol.

POP3 is the Post Office Protocol. It allows the user to download and keep the mails. The user can create folders and move the messages into them locally. It is stateless across sessions.

A more feature-rich protocol is IMAP or the Internet Mail Access Protocol. It allows the user to create remote folders and maintains user state information across sessions, It also permits a user agent to obtain components of messages, which is ideal for low bandwidth connections.

The user agent could be web-based, like in Hotmail or Gmail. Now, the transfer of the message from the sender to the mail server happens over HTTP. The access of mails by the receiver is also done over HTTP.

3.12 Peer To Peer Architecture

As mentioned before, there is no always-on server. The end hosts directly communicate, and the peers are intermittently connected.

Let us say we want to send N file copies of size F. In a client-server system, ignoring the delays, the time taken to send to a server would be NF/u_s, where u_s is the upload rate. Each client must download a file copy, and it the slowest download speed is d_{min}, then the slowest download time would be F/d_{min}. So, the minimum time to distribute to the N clients would be:

$$D_{c-s} \geq \max(NF/u_s, F/d_{min})$$

In the case of P2P system, at least one copy must be uploaded, taking time F/u_s. Each client must download a file copy, with the slowest download time once again being F/d_{min}. Unlike in the client-server scheme, as the file download gets completed on peers, the number of possible "providers" increases. As this happens the maximum upload rate increases, up till $u_s + \sum u_i$. So, the time needed is:

$$D_{P2P} \geq \max(F/u_s, F/d_{min}, NF/(u_s + \sum u_i))$$

In BitTorrent, the file is divided into chunks, typically 256 KB in size. There are trackers that track peers participating in a torrent. A new peer joins torrent and registers with the tracker to get a list of peers, and connects to some subset of them. While torrenting, a group of peers exchange chunks of a particular file. At any given time, each peer will have a subset of the chunks, and asks its neighbours for a list of which chunks they have. The peer will then take a call on which chunks it should request from the neighbour, and to which of its neighbours it should send requested chunks. Ideally, the peer would request and send the most scarce chunks.

One P2P protocol is Napster. Here, we have a central database where information is available, and the peers contact it for the necessary information. However, if the database is down then the peer-to-peer application cannot run.

Another P2P protocol is Gnutella. Unlike Napster, it is a completely decentralized protocol. On startup, an end host finds at least one other node to connect to. It then queries this node for more nodes to connect to, until it reaches some quota. Due to the completely decentralized network, this protocol is unscalable - searching takes time exponential in the number of nodes, and often end hosts are only connected intermittently.

Yet another P2P protocol is Kazaa, which is the basis for Skype. Kazaa is also a decentralized system. The users are divided into two groups - supernodes and ordinary nodes. Supernodes are powerful computers that act like traffic hubs, processing data requests from ordinary nodes.

3.13 Distributed Databases

Distributed databases are a common application of the P2P framework. In a distributed database, each peer holds a small subset of the total (key,value) pairs. Any peer can query the distributed database with a particular key. The distributed DB locates the peers that have the corresponding (key,value) pairs and return it to the querying peer. Any peer can insert a new (key,value) pair into the database.

A non-scalable way of doing this is as a distributed hash table. The key,value pairs are randomly scattered across all the peers. They maintain a list of the IP addresses of all the peers, and send their queries to all these peers. Those who contain the required (key,value) pairs respond with the matching pairs. This is, as mentioned before, not scalable since all peers must be queried.

Instead we may implement this with the **Circular DHT** protocol. A hash function assigns each node and key an m bit identifier using a base hash function such as SHA-1. The node's ID could be a hash of the port and the IP address, while the key's ID can be a hash of the original key. Since there is a finite number of hashes, these ID values will lie on an imaginary circle, ranging values from 0 to 2^{m-1}. We now assign (key,value) pairs to the peer that has the closest node ID to the pair's key ID.

One protocol that uses this concept is the **Chord protocol**. Every node keeps track of 2 pointers - the predecessor, a pointer to the previous node on the ID circle, and the successor, a pointer to the next node on the ID circle. When doing a lookup operation, a node will check whether the key's ID is between the ID of the node and it's successor. In this case, it will know that the key is present in the successor. If not, the query is forwarded to the successor, and the same operation is done recursively. The number of messages is $O(n)$, where n is the number of nodes. So, this is not scalable.

A scalable way to do this is that each node n contains a routing table with up to m entries (m is the number of bits), known as a **finger table**. The i^{th} entry in the table at node n contains the first node s that succeeds n by at least 2^{i-1}, i.e. $s = succ(n + 2^{i-1})$. s is called the i^{th} **finger** of node n. With this information, we can jump to the furthest successor that precedes the key ID.

4 Socket Programming

To a kernel, a socket is an endpoint of communication. To an application, a socket is a file descriptor that lets the application read or write to and from the network.

Sockets can be of two types depending on the transport service - UDP or TCP. UDP is

unreliable, while TCP is reliable and byte-stream oriented. By reliable, it means that all the packets sent will be received, in the correct order.

In UDP, there is no "connection" between client and server - there is no handshaking before sending data. The sender attaches the destination address and the port number to each packet, which the receiver extracts upon receiving. The transmitted data may be lost or received out of order.

A socket structure looks like this:

```
struct sockaddr {
  unsigned short int sa_family; // Address family
  unsigned in_addr sin_addr // Internet address
}
```

TODO: complete this.

5 Transport Layer

5.1 Introduction

The transport layer is responsible for logical communication between the hosts, unlike the network layer which is responsible for logical communication between processes. Since the transport layer works on top of the network layer, it is constrained by the underlying network layer protocol. However, in some cases it can offer services even when the network layer doesn't offer it.

The transport layer provides two protocols - UDP, an unreliable connectionless service, and TCP, a reliable, connection oriented service. The transport layer works with transport layer packets called **segments**.

5.2 Multiplexing and Other Functions

The transport layer multiplexes messages at sending time by handling data from multiple sockets and adding a transport header for each message. It is also responsible for demultiplexing the received messages by using the header information to deliver the received messages to the correct sockets. These are the only two services UDP provides, but TCP provides some additional services. As mentioned before, it also proved reliable data transfer. Besides this, it provides what is called congestion control. TCP congestion control prevents any one TCP connection from swamping links and routers between hosts with excess traffic, and tries to give every connection an equal share of traffic.

In **connectionless multiplexing**, as seen in UDP, the sockets are identified by a port number. So to perform multiplexing, the segments must contain information of a source port number and a destination port number.This information is passed from the sender's transport layer to it's network layer, which encloses it within a **datagram** (network layer packet). This datagram will now also have information about the sender's IP address and the destination IP address. Finally, when this message reaches the destination, that host checks the destination port number and redirects the message to that port. The source port number seems unnecessary, but acts as a "return address" for the application to use to communicate.

As such, a UDP segment has the following content:

- 16 bit Source port number
- 16 bit Destination port number
- 16 bit length
- 16 bit checksum, which is used to detect errors. This is calculated by treating the segment contents as a sequence of 16-bit integers, and summing them all. The one's complement of the sum is put in this field.
- The rest is the application data

TCP uses **connection oriented multiplexing**. Unlike a UDP socket, a TCP socket is identified by a 4-tuple of (source IP address, source port number, destination IP address, destination port number). Thus, when a TCP segment arrives from the network to a host, the host uses all four values to demultiplex the segment to the appropriate socket. In contrast with UDP, two arriving TCP segments with different source IP addresses or source port numbers will be directed to two different sockets.

Knowing that UDP is unreliable, one might wonder why we use UDP at all. Some reasons are as follows:

- No connection establishment, which would add delay
- It is simple, since there is no connection state to maintain at the sender and the receiver end
- It has a small header size
- It has no congestion control. While this is undesirable in some cases, it means that UDP can run as fast as desired.

5.3 Designing a Reliable Data Transfer

Let us assume that the underlying channel is perfectly reliable. Then, let us design a protocol called `rdt1.0`. We diagrammatically show this as two Finite State Machines - one for the sender, and another for the receiver.

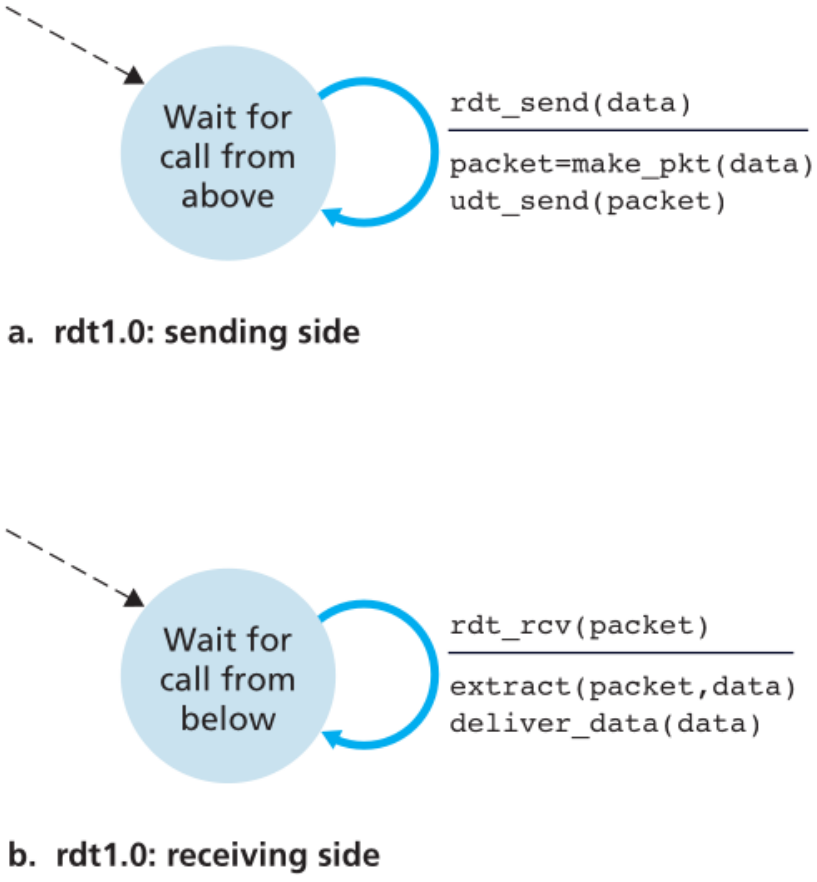
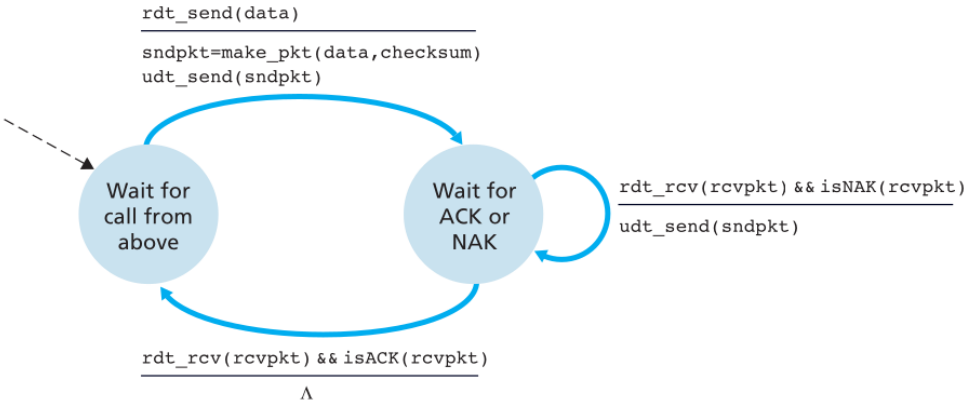


Figure 1: FSM for rdt1.0

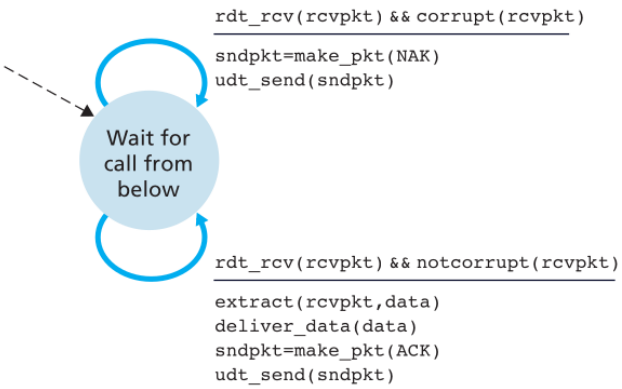
As we can see in Fig 1, sender simply sends data into the underlying channel, and the receiver simply reads from the underlying channel.

What if our underlying channel has some bit errors? Then the packet may have some flipped bits! For this we design our second protocol, `rdt2.0`. For this, we can use the checksum to detect the bit errors. If there were no bit errors, the receiver will return an **acknowledgement** (ACK), explicitly telling the sender that the packet is OK. If not, the receiver will return a **negative acknowledgement** (NAK), saying that the packet had errors. In this case, the sender will retransmit packets. The FSM for the same is in Fig 2.

However, this has a fatal flaw. What if ACK/NAK is corrupted? One way to handle this is to be safe, and send the packet again if the ACK/NAK is corrupted. But this creates



a. rdt2.0: sending side



b. rdt2.0: receiving side

Figure 2: FSM for rdt2.0

a new problem of handling duplicate packets, where the receiver is unsure if this is new information of old data. To fix this, the sender adds a sequence number to each packet. The receiver checks this to decide which packet to discard. In this mechanism (called a stop and wait mechanism), we need only 1 bit for the sequence numbers.

This creates a new protocol - `rdt2.1`. The FSMs are as in Fig 3 and 4.

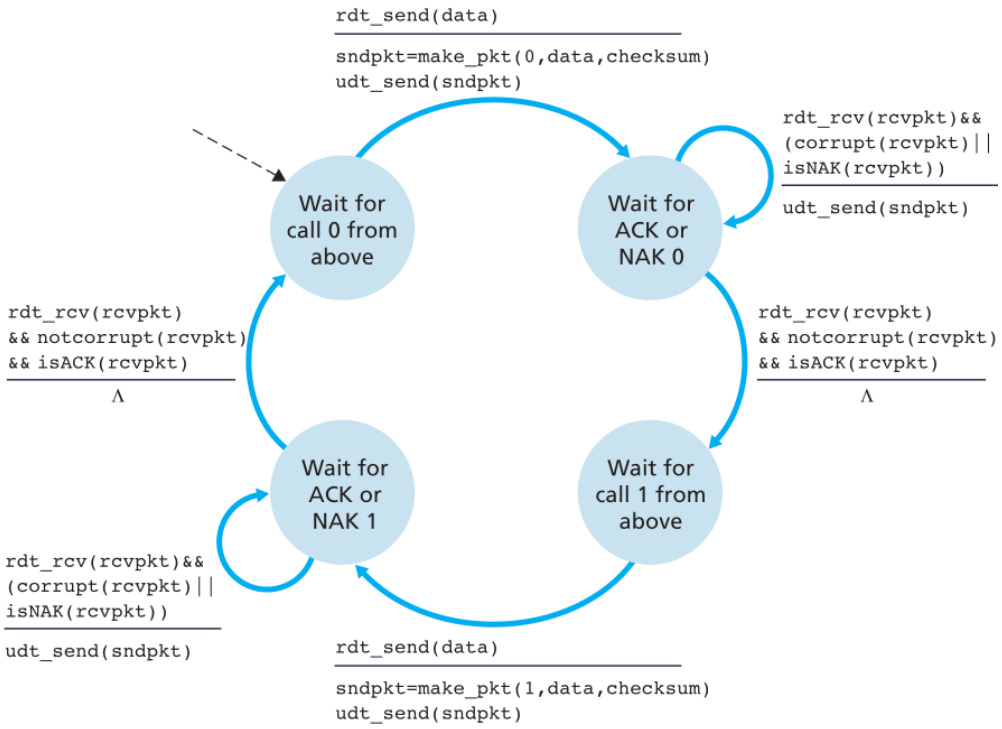


Figure 3: rdt2.1 Sender

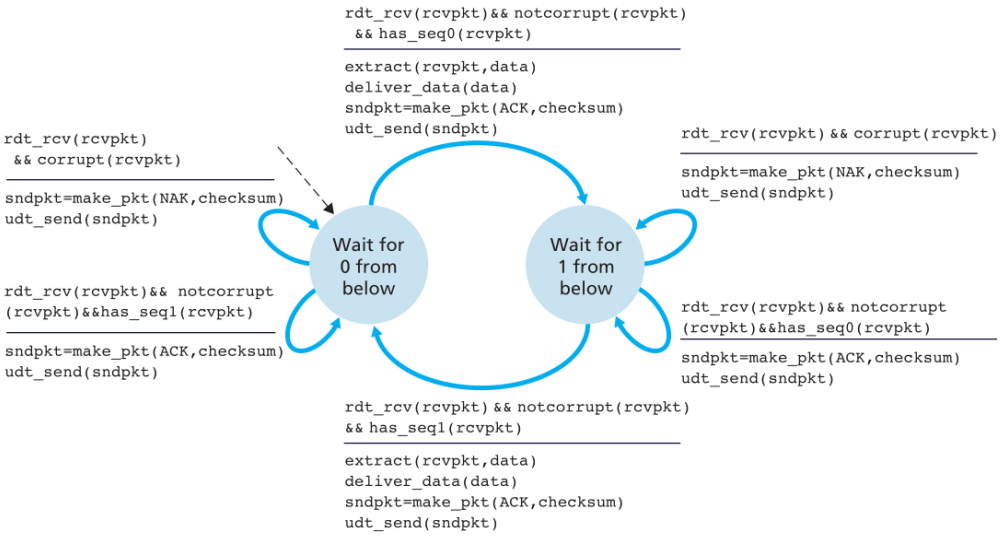


Figure 4: rdt2.1 Receiver