1 HTTP protocol

This protocol is linked to the application layer, it stands for **H**yper**t**ext **T**ransfer **P**rotocol and is the web's application layer protocol. It is a *client/server* model:

- *client:* browser that requests/receives and displays a web object.
- *server:* the web server sends the object as a response to the request.

As the requests and responses can be understood no matter the web browser, it must be **standardized**.

It uses *TCP* because it requires **reliability**. For example, imagine a scenario where the request does not arrive and is lost, the server thinks it was delivered but the client not and redoes the request and so on.. Also, the response can be a big file that is sent as a series of chunks of data and if a piece is missing, it must be detected -> reliability.

The client initiates a *TCP* connection to the server on port 80. The server accepts this connection from the client and then *HTTP* messages are exchanged between both. When it is finished, the *TCP* connection is closed.

HTTP is said to be **stateless**, which means that it maintains no information about the past requests from the client.

There exists two types of connections:

- **non-persistent HTTP:** at most one object is sent over the *TCP* connection and it is closed right after (means that downloading several objects requires multiple connections).
  A typical exchange between server and client looks like Figure 1.
  Let the *RTT*(Round Trip Time) be the time to send a small packet to travel from client to server and back, then the *HTTP* response time is given by one *RTT* to initiate the connection, another one from the request to the first byte of the answer and then the time needed to transmit the file -> $2RTT+$file transmission time.
  The file transmission time is related to the average throughput and depends on the congestion and on eventual packet losses that require re-transmission, it is calculated by the file size over the average throughput of the connection.
- **persistent HTTP:** multiple objects can be sent over the same *TCP* connection: the server leaves the connection open after sending the response. If the requests are pipelined, it takes about one *RTT* for all the objects.
  This second one seems to be better but for the server prospective, maintaining the connection while not being sure that the client will request new things can be useless and lead to saturation at server side.

There are two types of *HTTP* messages: the request and the response. A message consists of a header and a payload, the end of the header is announced by a carriage return. For persistent type of connection, the field connection contains the message "Keep-alive". The header also contains the host even if the *TCP* connection is already established. The purpose of that is for caching, for example when we are talking to the proxy's cache and not the server itself.

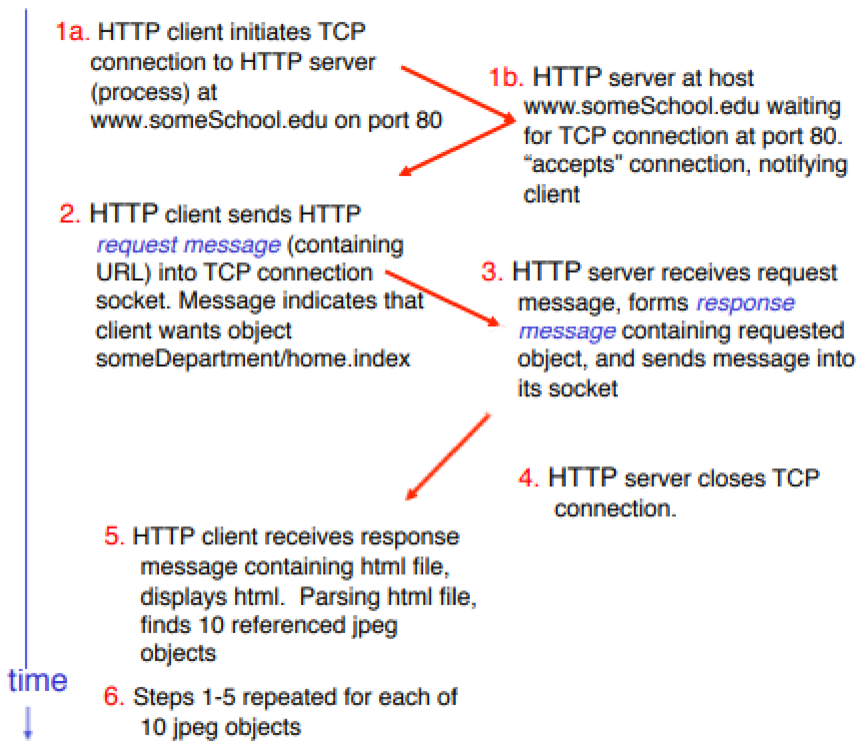

1a. HTTP client initiates TCP connection to HTTP server (process) at www.someSchool.edu on port 80
1b. HTTP server at host www.someSchool.edu waiting for TCP connection at port 80. "accepts" connection, notifying client
2. HTTP client sends HTTP request message (containing URL) into TCP connection socket. Message indicates that client wants object someDepartment/home.index
3. HTTP server receives request message, forms response message containing requested object, and sends message into its socket
4. HTTP server closes TCP connection.
5. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects
6. Steps 1-5 repeated for each of 10 jpeg objects
time

Figure 1: Typical exchange for several objects with non-persistant *HTTP*.

The request is either a *GET*, *POST* or *HEAD* and is written in *ASCII* format. Usually, *GET* is used. *POST* is used when we need to send data to a given URL instead of receiving from it and a *HEAD* is a sort of *GET* without content just to check if the given object exists.
In version *HTTP1.1*, two other exist: *PUT* which adds new files/objects to a server at a given URL and *DELETE* that deletes a file or object at a given URL.

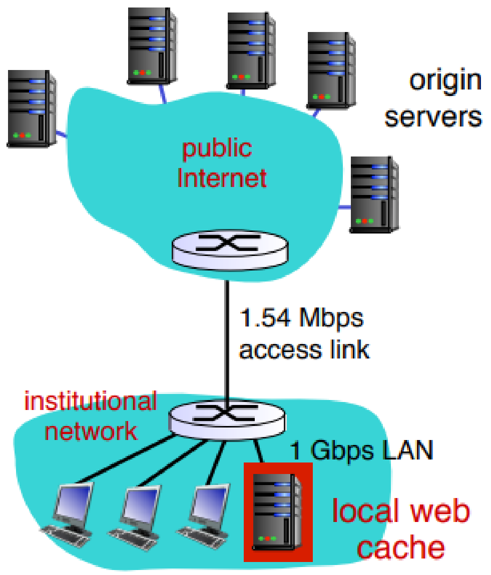
The response also contains a header and a body separated by a carriage return. The first line always tells the version as well as code which is 200 for OK, 300 or 303 for redirection and above 400 for indicating an error. The field "Last modified" gives information about how likely it is that it will change in a close future.

One header message for the request is "cookie" and for the response "set-cookie", it deals with **cookies** which allow to keep state. A cookie file is kept on the user's host and is managed by the user's browser and at the web server, there is a back-end database that understand these cookies. Such a cookie file is given to us the first time we visit a website and allows later on to faster retrieve to data. It can actually be used for authorization, remembering shopping carts, provide personal recommendations and to remember the user session. One must pay attention that there are security issues associated, if one can get our cookies, it can act as if it is us.

Let's finally talk about **web caches** or **proxy servers**. The idea is that the web browser sends all our request to cache and if we ask for an object, if the proxy servers has it, it is returned from the cache, and only if it hasn't the object in cache, it will ask the origin server. That is why the "host" field is always present even if the connection to the server is done, it is present to explicitly tell the proxy server the original source of the object. It can be useful for the three following reasons (Win-Win-Win situation):

- It reduces response time for the client.
- It reduces the traffic on an institution's access link
- It reduces the load on the original server

A proxy server acts both as a server and a client and enables "poor" content providers to effectively deliver content. Let's consider an architecture as shown in the following figure in which the bottleneck is the outgoing link, then even if the internal links are very fast, request are slowed down by this slow link, leading to delays. A first solution would be to increase the speed of the link, but it is very costly. A second one is to use a proxy server in the institutional network which would decrease the delay by decreasing the amount of people (i.e. the Utilization) of the low speed link. It is less expensive as it is programmed in software.



In case one would make sure the version of the object in the cache is not out-dated, one can use a CONDITIONAL GET which sends the object only if it has changed since a given date. It thus either returns a 304 code (Not modified) or a 200 code if it has changed and then the payload contains the object. Using the "Last modified" field, one can also have a guess about how likely it is that the content changed.

2 Principle of DNS

It stands for **D**omain **N**ame **S**ystem. It is basically the answer to the question "*How to map between IP address and name and vice versa ?*". It consists of a distributed database. The host of the application layer communicates with these servers to have a address/name translation.

It can translate a name to an *IP* address but also a name to another name if there exist several aliases. For example, whatever the *Amazon* server you need to talk to, writing "*www.amazon.com*" will work. Indeed, load balancing is more and more popular and the idea is that there are several machines (set of servers) and we are mapped such that there is no overload and without being mapped to a server located too far away. Similarly, for mail server aliasing where "*@uliege.be*" will redirect the Uliege email server.

It is not centralized because it would lead to a total fail if there is a single point of failure or in case of the maintenance, it would be hard to handle all the traffic.

It is a distributed and hierarchical database as shown in Figure 2. Let's assume one wants the *IP* address for "*www.amazon.com*", the client first queries the root server to find the *com* server. It will query that server the *amazon.com* to get to the *amazon.com DNS* server and will then get the *IP* address. There are over 400 Root servers scattered all over the world.

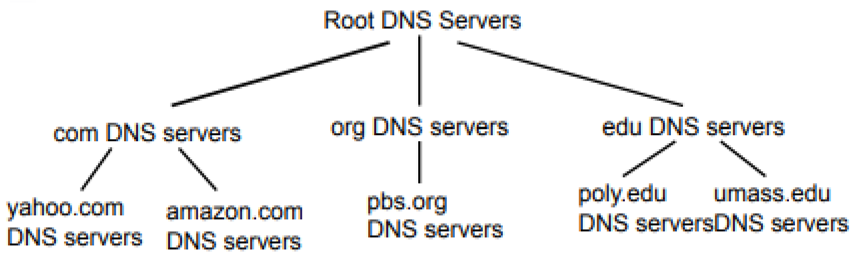


Figure 2: *DNS* hierarchy.

There are two types of *DNS* servers:

- **T**op-**L**evel **D**omain (TLD) servers: these are responsible for the *.com*, *.org*, *.net*, ... as well as all top-level country domains such as *.fr*, *.be*, ...
- Authoritative *DNS* servers: personal *DNS* of some organization's.

The local *DNS* name server (also called the default name server) does not strictly belong to the hierarchy. Each Internet Service Provider (ISP) has one. When a host makes a *DNS* query, that query is sent to its local *DNS* server. It has a local cache of recent name-to-address translation pairs and it acts as a proxy and forwards the query into the hierarchy. Two examples of resolution are shown below:

- Iterated Query: the contacted server replies with the next server to contact "*I don't know this name, but ask this server*". With caching, it is possible to skip some steps and directly go to the last step next time the same query is performed. (Mostly used)

- Recursive query: puts burden of name resolution on contacted name server. It is linked to high traffic at the upper levels of the hierarchy but it allows all the levels to cache because the response flows back over the same way that the query moved. The major drawback is that all entities stay in the loop for long time and memory is needed to keeps the state to be able to respond, which can overload the server.

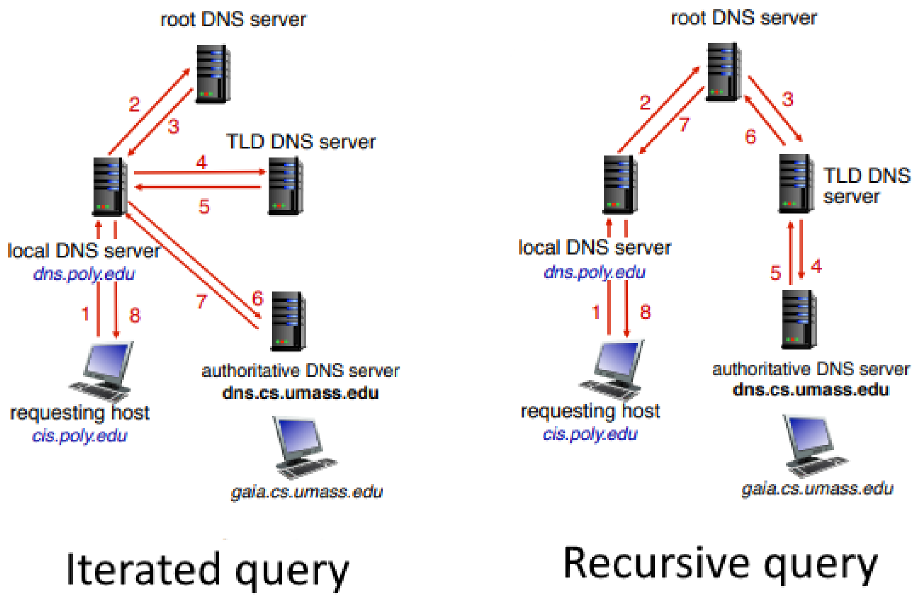


Figure 3: Two type of *DNS* resolutions.

More about **caching**: after some Time To Live (TTL), the entries disappear. The TLD servers typically cached in the local name servers meaning that the root servers are not often visited. The cache entries may be out of date for example if the host corresponding to a given name changes its *IP* address, it may not be known to the world until the TTL expires.

A *DNS* is a distributed database that stores **R**esource **R**ecords (RR) with the following format:

$$RR\ \ format: \quad (name, value, type, TTL)$$

Here are the different types:

- Type=A: the name is the hostname and the value is the *IPv4* or *IPv6* address (mostly used)
- Type=NS: the name is the domain, the *DNS* zone (sub-tree in the hierarchy) (for example "*Uliege.be*") and the value is the hostname of the authoritative name server of this domain.
- Type=CNAME: the name is an alias name for some "canonical" real name and the value is the "canonical" name. For example, "*www.ibm.com*" is actually "*servereast.backup2.ibm.com*" (the machine that actually runs the server).
- Type = MX: the value is the name of the mailserver associated with the domain name.

The *DNS* protocol usually runs over a *UDP* connection because the request is a small packet and the response as well, if the response gets lost, we can just resend it as it only constitutes a few bytes. It would be overkill to use a reliable channel. Here is a typical message: (notice that a question can have several answers)

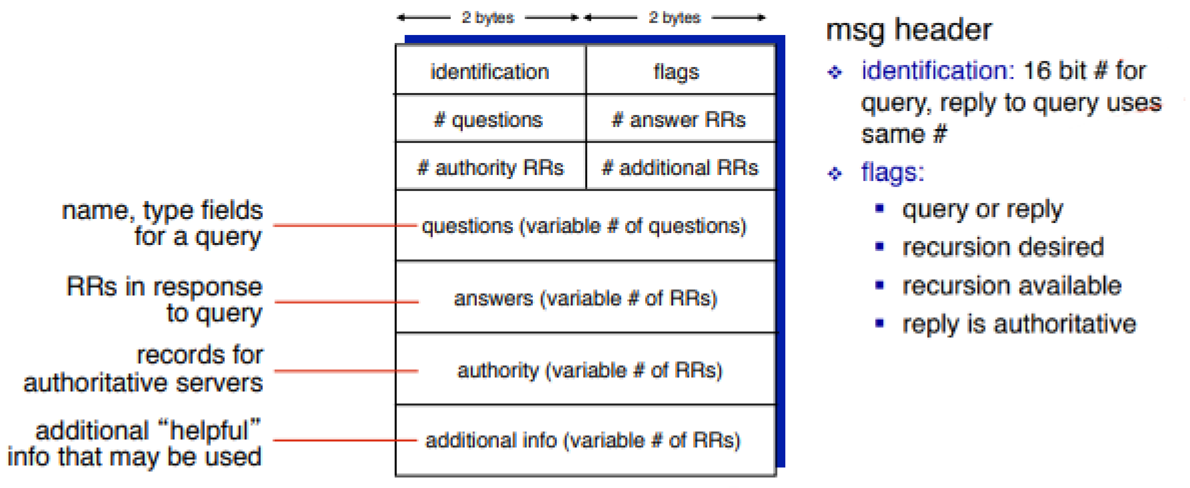

2 bytes
2 bytes
identification
flags
# questions
# answer RRs
# authority RRs
# additional RRs
questions (variable # of questions)
answers (variable # of RRs)
authority (variable # of RRs)
additional info (variable # of RRs)
name, type fields for a query
RRs in response to query
records for authoritative servers
additional "helpful" info that may be used
msg header
identification: 16 bit # for query, reply to query uses same #
flags:
query or reply
recursion desired
recursion available
reply is authoritative

Figure 4: Structure of a *DNS* protocol message.

In order to insert a new node in the tree (=inserting a record into a *DNS*), one needs to provide names, *IP* addresses of authoritative name server. The registrar needs to insert two RRs into the *TLD* server: example for a given URL:

$$(networkutopia.com,\ dns1.networkutopia.com,\ NS)$$
$$(dns1.networkutopia.com, 212.212.212.1,\ A)$$

where $dns1$ is the authoritative server of this address. The idea is domain$\rightarrow$ map to server $\rightarrow$ map to *IP*address.

3 Socket programming (TCP and UDP), and addressing and (de)multiplexing in the transport layer

The two socket types for two transport services are:

- UDP: stands for **U**ser **D**atagram **P**rotocol and is **unreliable**.
- TCP: stands for **T**ransmission **C**ontrol **P**rotocol and is a **reliable** and **byte stream oriented** protocol.

Socket programming with UDP

There is no connection between the client and the server: no handshaking before sending the data, the sender explicitly attaches *IP* destination to each data unit and the receiver extracts the sender's *IP* address and port from each received data unit.
The transmitted data may be lost or received out-of-order.

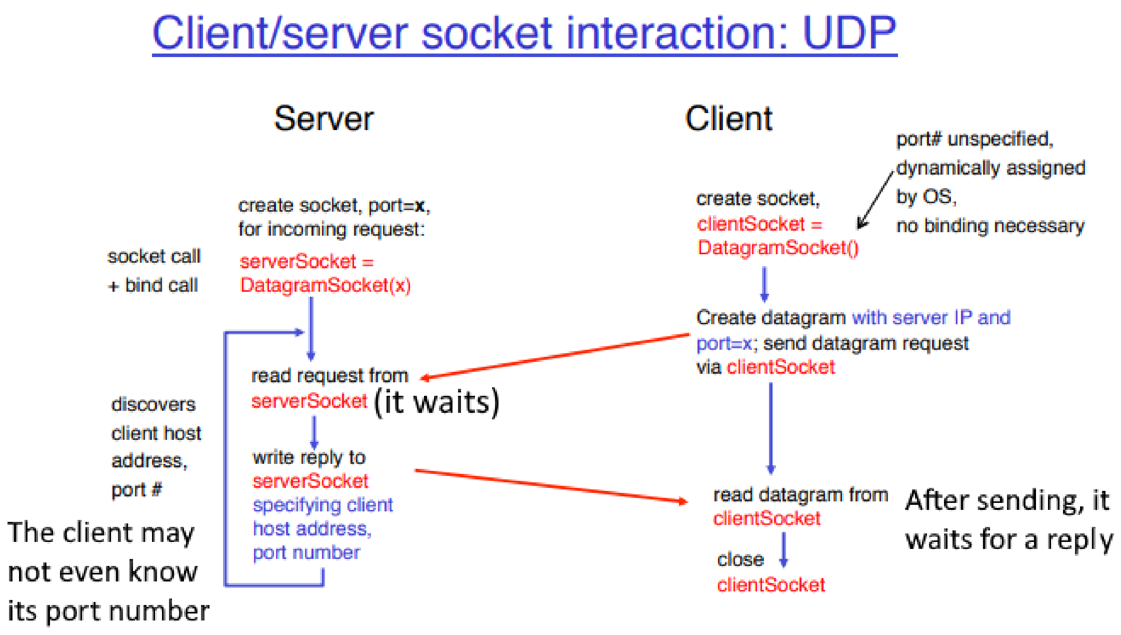

Client/server socket interaction: UDP
Server
Client
port# unspecified, dynamically assigned by OS, no binding necessary
create socket, port=x, for incoming request:
serverSocket = DatagramSocket(x)
socket call + bind call
create socket, clientSocket = DatagramSocket()
Create datagram with server IP and port=x; send datagram request via clientSocket
read request from serverSocket (it waits)
discovers client host address, port #
write reply to serverSocket specifying client host address, port number
read datagram from clientSocket
After sending, it waits for a reply
close clientSocket
The client may not even know its port number

Figure 5: Typical exchange with *UDP*.

Both the client and the server use DatagramSocket and the destination *IP* and port are explicitly attached to app data unit. It requires one single socket at the server side and only one port no matter the number of clients.

Socket programming with TCP

The client must contact the server: the server process must be running meaning that the server must have created a socket that welcomes client's contact. The client contacts the server by creating a *TCP* socket specifying the *IP* address and port number of the server process. When contacted by the client, the server *TCP* creates a new socket for the server process to communicate with that particular client. This way, the server can talk with several clients and the *IP* as well as the port are used to distinguish the clients. A lot of

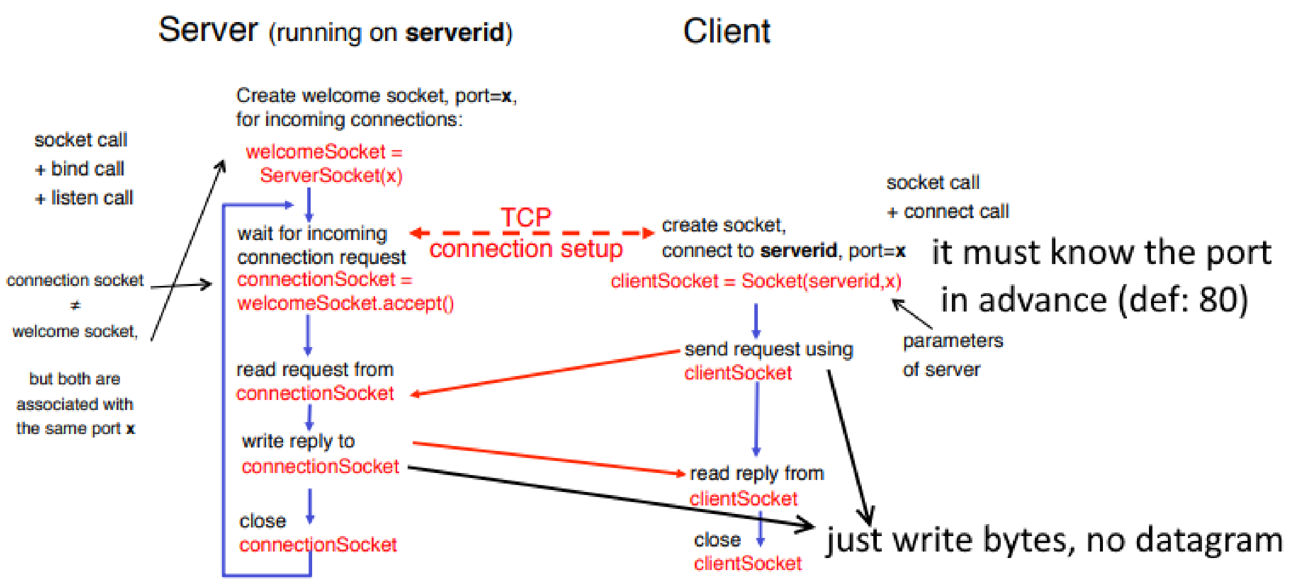

Server (running on serverid)
Client
Create welcome socket, port=x, for incoming connections:
welcomeSocket = ServerSocket(x)
socket call + bind call + listen call
connection socket ≠ welcome socket, but both are associated with the same port x
wait for incoming connection request
connectionSocket = welcomeSocket.accept()
TCP connection setup
create socket, connect to serverid, port=x
clientSocket = Socket(serverid,x)
socket call + connect call
it must know the port in advance (def: 80)
parameters of server
read request from connectionSocket
send request using clientSocket
write reply to connectionSocket
read reply from clientSocket
close connectionSocket
close clientSocket
just write bytes, no datagram

Figure 6: Typical exchange with *TCP*.

stuff happens in the transport layer to ensure that the bytes arrive and that it is reliable (which is not the case with *UDP*).

It can welcome several clients as well by queuing or with // programming. It requires one welcoming socket + 1 socket per client and it only uses port x.

Multiplexing at sender means that data is handled from multiple sockets and a transport header is added and **demultiplexing** at receiver means that thanks to the header, info is delivered to the correct socket. It is illustrated below:

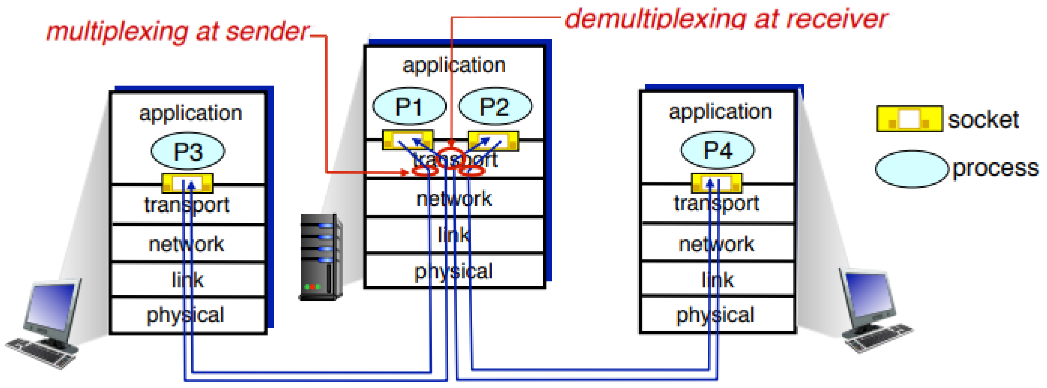


Figure 7: (De)Multiplexing processes.

With *UDP*, in the application layer: a socket is created and has a host-local port and the datagram contains the *IP* and port number. In the transport layer, when *UDP* receives a segment from below, it checks the destination port number in the segments and directs the *UDP* segment to the socket with that port number. Note that *IP* datagrams with same destination port number but different *IP* source addresses and/or port number are redirected to the same socket at destination.

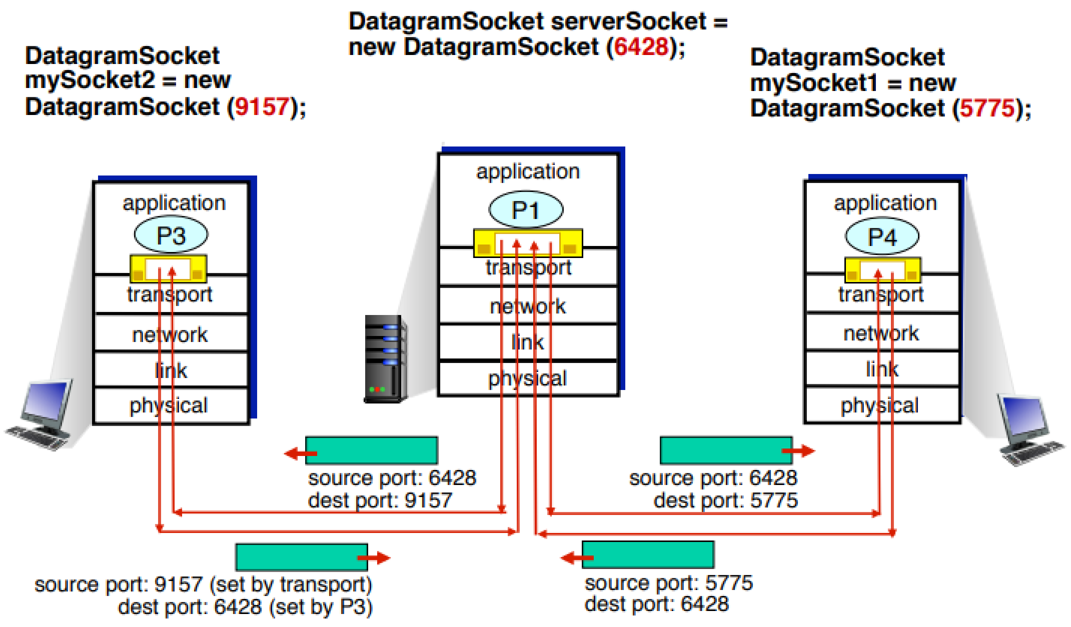

DatagramSocket serverSocket = new DatagramSocket (6428);
DatagramSocket mySocket2 = new DatagramSocket (9157);
DatagramSocket mySocket1 = new DatagramSocket (5775);
application
P3
transport
network
link
physical
P1
P4
source port: 6428
dest port: 9157
source port: 6428
dest port: 5775
source port: 9157 (set by transport)
dest port: 6428 (set by P3)
source port: 5775
dest port: 6428

Figure 8: Example of (De)multiplexing with *UDP*.

With *TCP*, each client has a different socket and with non-persistent *HTTP*, we even have different sockets for each request. These many simultaneous *TCP* sockets are all associated with the same server port number but it is not enough to direct segment to the appropriate socket. Each socket is thus identified by a 4-tuple: source *IP* address and port number and destination *IP* and port number. The demultiplexing uses all four values to direct a segment to the appropriate socket.

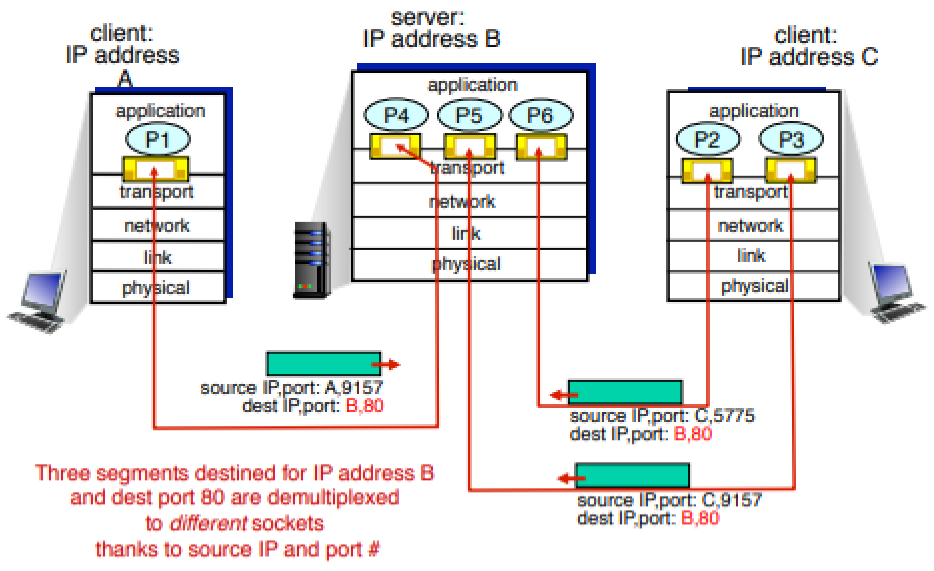


Figure 9: Example of (De)multiplexing with *TCP* without multi-threading.

4 Reliable data transfer: design steps of the Alternating-Bit protocol, its efficiency, and error detection techniques (checksum and CRC)

A reliable data transfer basically consists of a reliable channel which means that there is a service provided such that the data put in the channel is ensured to arrive in the right order at the receiver side.

To see how we can obtain a reliable data transfer, we are going to incrementally develop sender and receiver side of a Reliable Data Transfer (RDT) protocol.

RDT1.0: reliable transfer over a reliable channel

Here we assume that the channel is perfectly reliable: no bit errors and no loss of packets. The Finite State Machine (FSM) is the following:

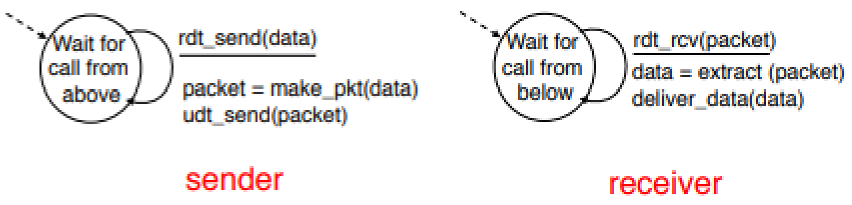


Figure 10: FSM of RDT1.0

where make_pkt consist of creating a header and putting the payload and extract(pkt) is the symmetric action of make_pkt.

RDT2.0: channel with bit errors

We now assume that the underlying channel may flip bits in packet and the question is "*how to recover from errors?*":

- Acknowledgements (ACKs): receiver explicitly tells sender that pkt received is OK.
- Negative acknowledgements (NAKs): receiver explicitly tells sender that the pkt had errors.
- sender re-transmits pkt on receipt of a NAK.

So the additional features are error detection and feedback via control messages.

The problem is that the protocol as such is incomplete: the ACK and NAK messages should also have a form of checksum: can we assume a positive feedback was sent if a positive feedback is received ? Can something wrong happen with the feedback message ? There is a **fatal flow**.

If a garbled ACK/NAK is detected and that it is discarded, the sender doesn't know what happened at receiver side and can only retransmit: risk of duplicates. We need a mechanism to detect duplicates: add a number to each packet.

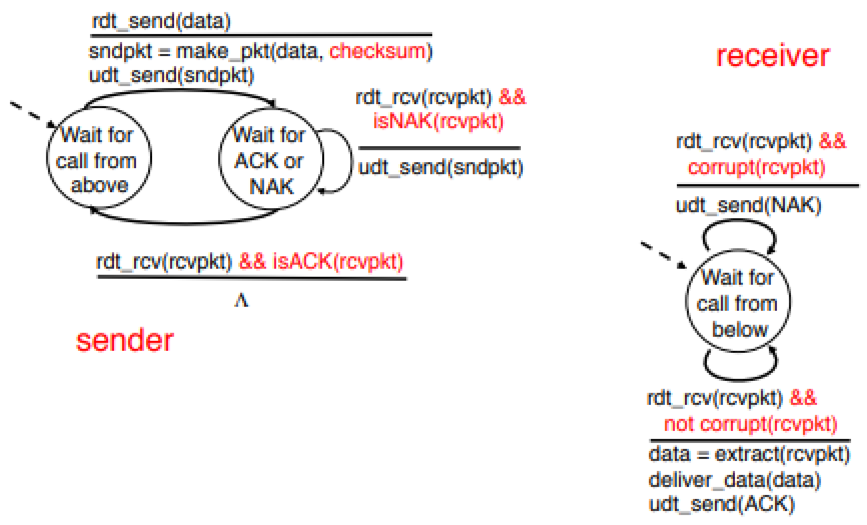


Figure 11: FSM of RDT2.0

RDT2.1: handles garbled ACK/NAKs

For the sender, a sequence number is added to each pkt. Two number are sufficient because we can just alternate and check if we receive the right one. It must now also check if the ACK/NAK are corrupted leading to twice as many states in the FSM. The receiver on the other hand must check if the pkt is a duplicate or not. The FSM becomes more complex, it is shown below.

The answer to the question in the blue square is: the ACK was corrupted so he sent it again but we send a positive ACK to allow the sender to move on: the fact that he sent a duplicate told us that he did not receive the positive ACK. Note that, as a duplicate is detected, it is not send to the socket because it was already done before.

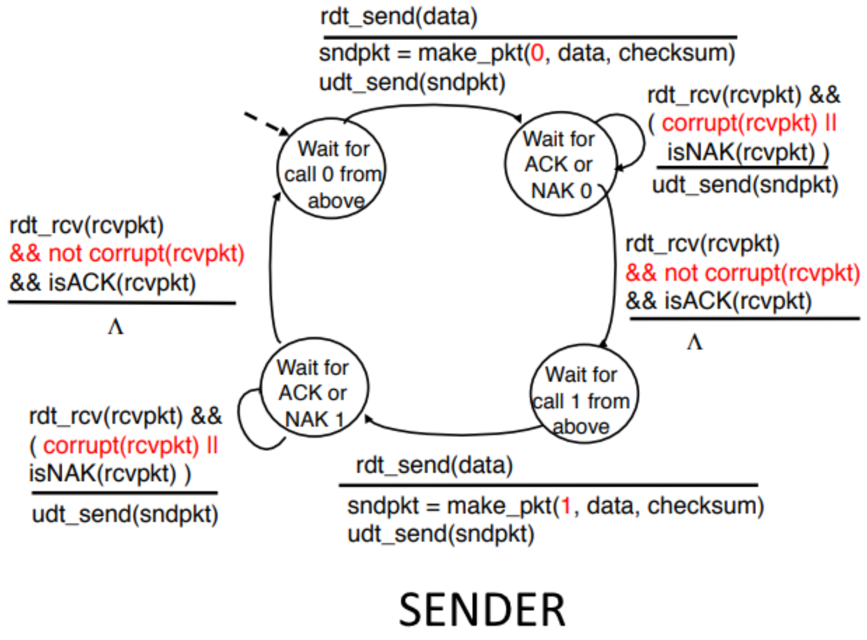

rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
Wait for call 0 from above
Wait for ACK or NAK 0
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))
udt_send(sndpkt)
rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && isACK(rcvpkt)
Λ
rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && isACK(rcvpkt)
Λ
Wait for ACK or NAK 1
Wait for call 1 from above
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))
udt_send(sndpkt)
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
SENDER

Figure 12: FSM of RDT2.2

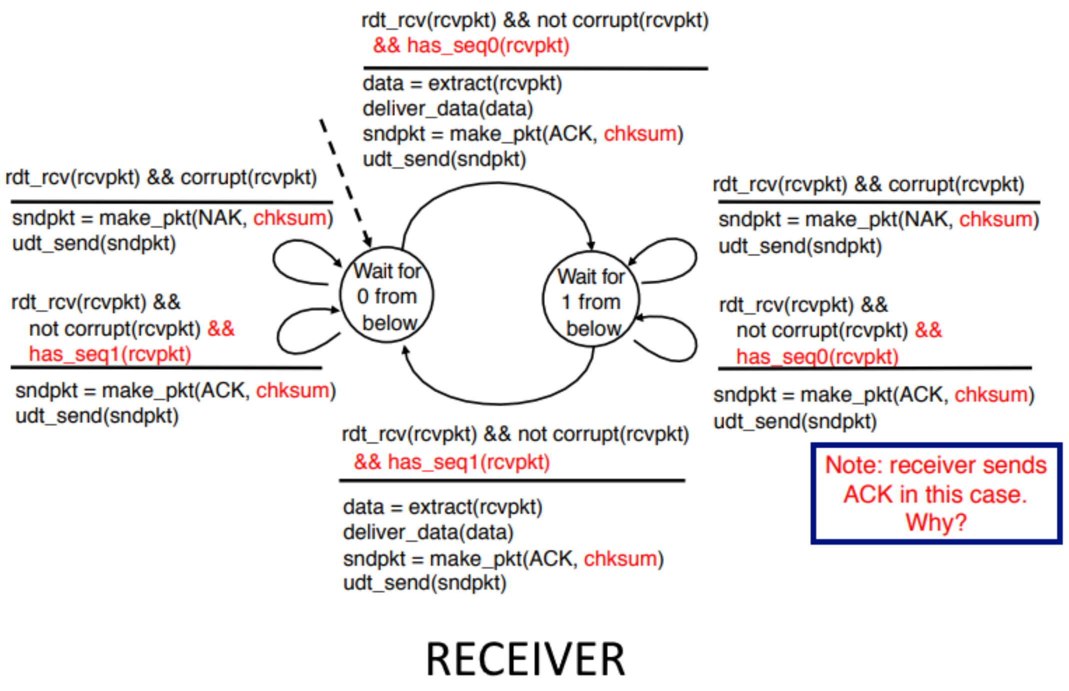
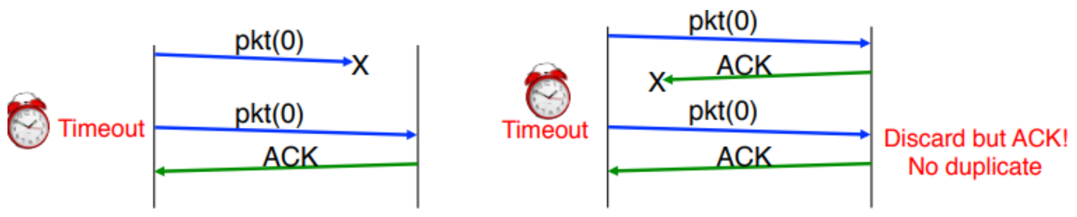

rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && has_seq0(rcvpkt)
data = extract(rcvpkt)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && has_seq1(rcvpkt)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
Wait for 0 from below
Wait for 1 from below
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && has_seq0(rcvpkt)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && has_seq1(rcvpkt)
data = extract(rcvpkt)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
Note: receiver sends ACK in this case. Why?
RECEIVER

Figure 13: FSM of RDT2.2

RDT3.0: channels with errors and loss

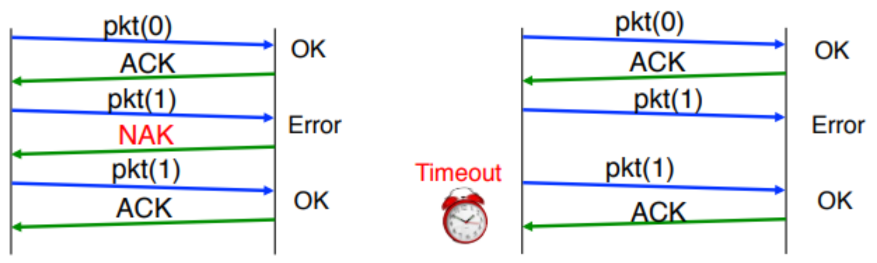
The assumption again change and the underlying channel can now also lose packets (data or ACK/NAKs). The approach is to wait for a reasonable amount of time for ACK and retransmit if no ACK/NAK received in this time.
The timer must be wisely calibrated (performance: not too long / RTT are not over: not too short). The principle is the following:



RDT3.1: we actually don't need NAKs

Notice that it is actually possible to just not send ACK when we want to send a NAK. A NAK might make it faster but here we just want something as simple as possible.

If a packet is delayed and not lost, the re-transmission will be duplicate but the sequence number will handle the detection of such cases. But, **race conditions** are possible between the received ACK and the retransmitted packet (if the channel is full duplex: 2 directions of transfer are independent):



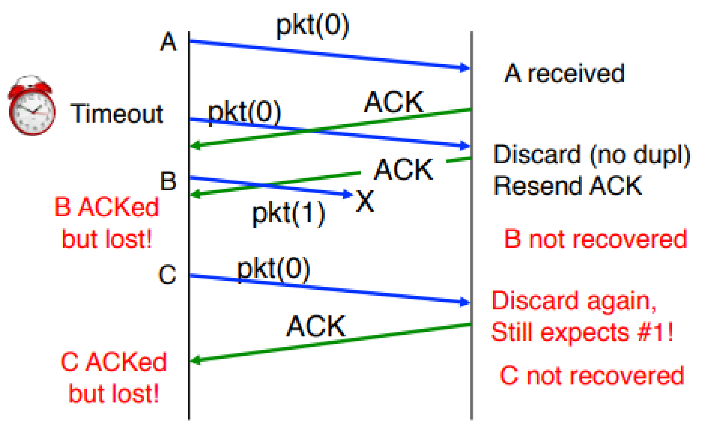
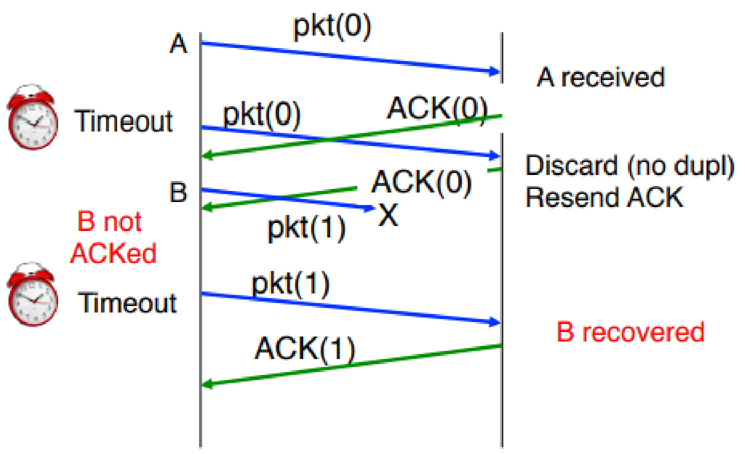


Figure 14: Race condition leading to errors.

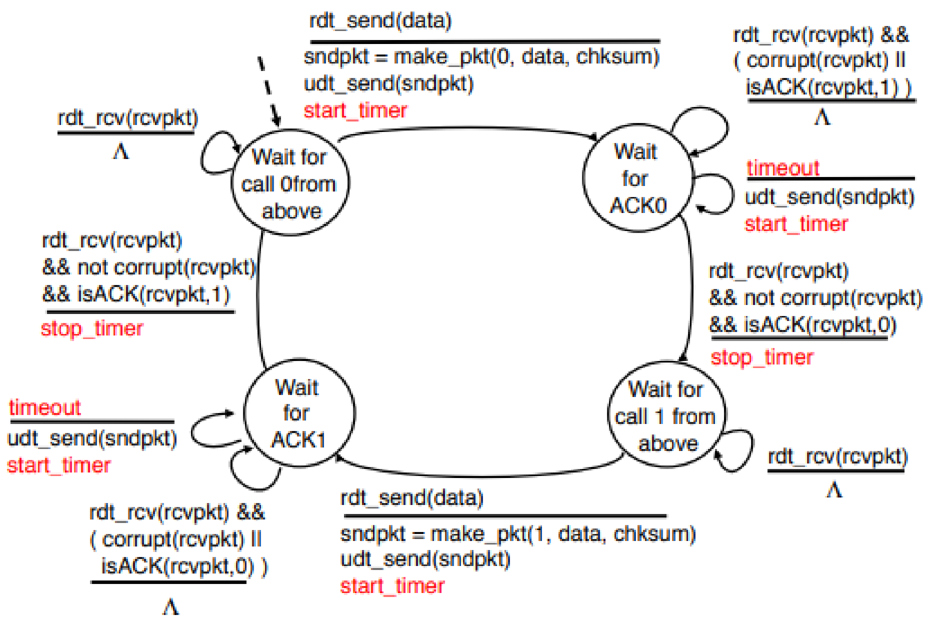
RDT3.2: adding sequence number in ACKs

We add a number that is linked to the packet that was sent so that sender knows which pkt was acknowledged:



Figures 16 shows the alternating bit protocol in action under several scenarios. Figure 15 shows the limitation of the protocol. A first solution to this scenario is to choose a large timeout so that the sender is sure that the previous copy of this pkt as well as its duplicate are lost in network. A second, better, solution is to use more than just 2 sequence numbers, so that it is re-used after such a long time that it is impossible that the pkt is still somewhere in the network.

The final FSM of the sender is shown in the following Figure:


rdt_send(data)
sndpkt = make_pkt(0, data, chksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) \|\| isACK(rcvpkt,1))
Λ
rdt_rcv(rcvpkt)
Λ
Wait for call 0from above
Wait for ACK0
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && isACK(rcvpkt,1)
stop_timer
rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && isACK(rcvpkt,0)
stop_timer
timeout
udt_send(sndpkt)
start_timer
Wait for ACK1
Wait for call 1 from above
rdt_rcv(rcvpkt)
Λ
rdt_send(data)
sndpkt = make_pkt(1, data, chksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) \|\| isACK(rcvpkt,0))
Λ

SENDER

In terms of efficiency, one can say that it is a **stop-and-wait** operation mode, meaning that it is most of the time idle. The number of bits on the channel (that are known by sender but no yet by the receiver) can be expressed by $R \cdot RTT/2$ (*e.g.* a rate of 6 bits per second for 30 seconds leads to 180 bits on the channel). The last bit is send at time $t = L/R$ and it takes one RTT for the last ACK to arrive after the last pkt is sent. Let $\mathcal{U}$ be the utilisation: fraction of time the sender is busy sending:

$$\mathcal{U}_{sender} = \frac{\frac{L}{R}}{RTT + \frac{L}{R}} = \frac{1}{1 + \frac{R \cdot RTT}{L}}$$

One can see that the utilisation is bad when $\frac{R \cdot RTT}{L}$ is big, which happens when one has a good data rate and/or small packets to send. So, it was OK over local low speed network as there were at the time, but with nowadays technology, let's see: 1 Gbps link, 15 ms RTT and 1Kb packet leads to a transmission time L/R of 8μs and a utilisation of 0.027 %.

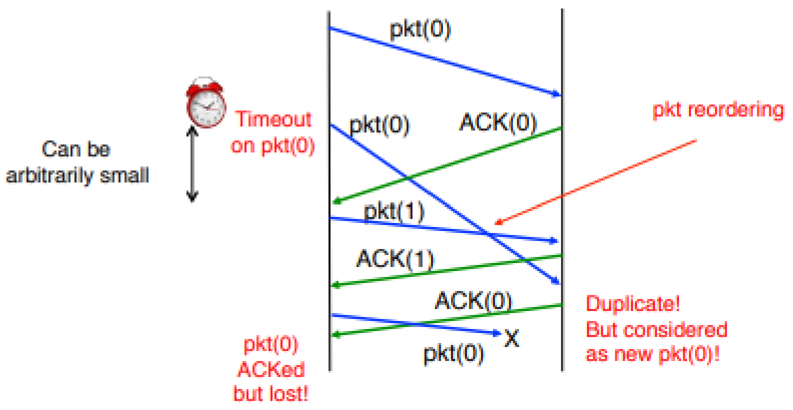


Figure 15: Alternating bit protocol failure when pkt reordering.

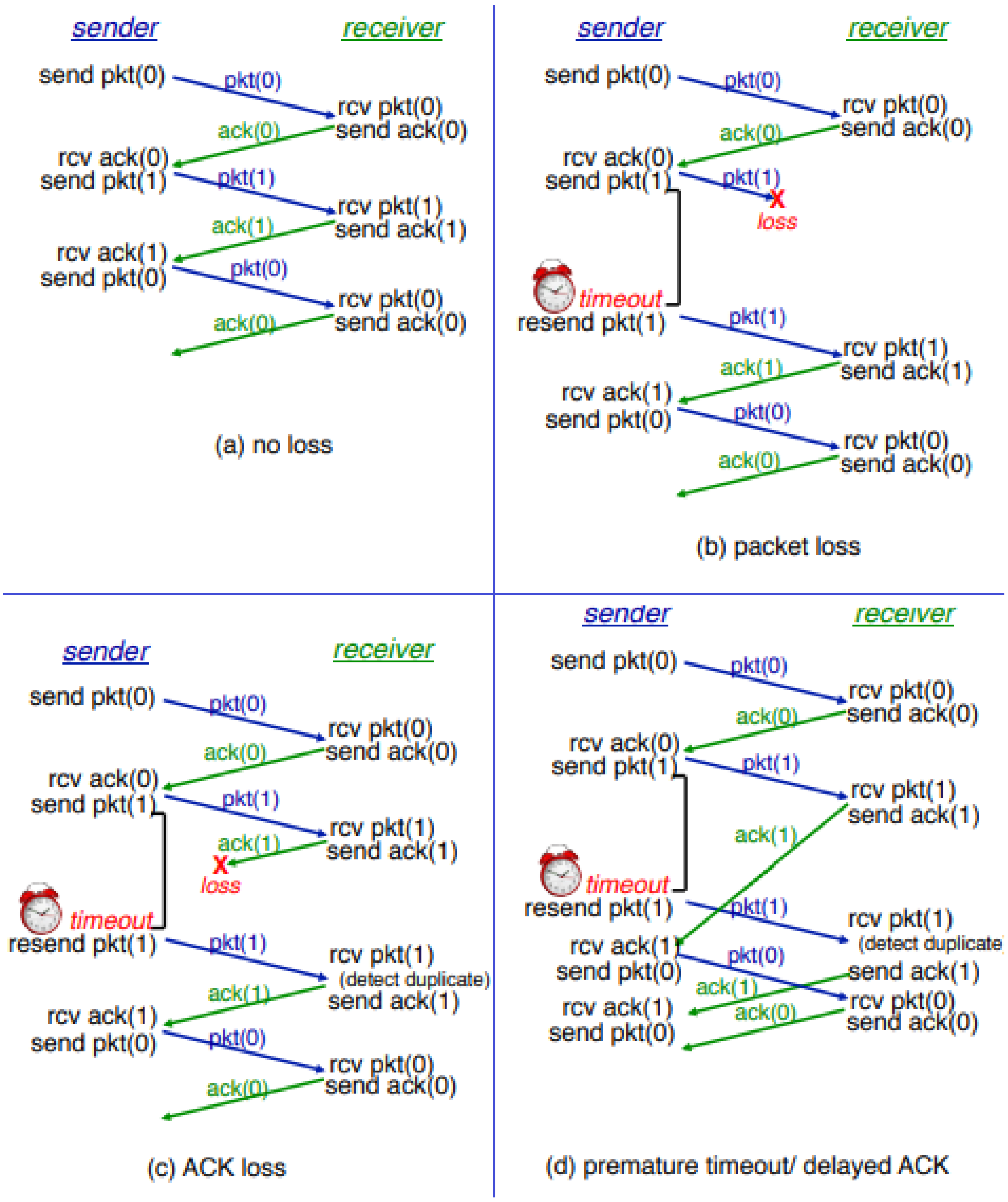


Figure 16: Alternating bit protocol in action

Error detection

1. Checksum: the checksum is a mechanism to detect errors in a transmitted segment. The idea is that the sender treats the segment content (including the header) as a sequence of 16-bit integers. The checksum is the 1's complement addition of the content (perform the sum, if carry at last bit: add 1 to the result). The receiver computes the checksum of the received segment and checks if the computed one corresponds to the one that was sent. If "NO" there was an error and if "YES" there were maybe errors but fingers crossed.
   This method is used on a *UDP* link which is unreliable by nature anyway. There are better methods on other layers anyway, not a problem if this one is not perfect.

2. **C**yclic **R**edundancy **C**heck (CRC): this method is more powerful than the previous one. Consider a number $< D, R >$ as d bits of data to be sent and r bits for the CRC code, then the equivalent number can be read as $D \cdot 2^r xor R$ (in base 2 arithmetic: +,- and xor are equivalent operations). The idea is to chose the r CRC bits such that $< D, R >$ is exactly divisible by G, a $r + 1$ bit pattern called Generator and known by sender and receiver. The receiver then divides the number $< D, R >$ by this value G and if the rest of the division is not zero, an error was detected. For it to work, one needs:

   Desired: $D \cdot 2^r \; xor \; R = n \cdot G$
   equivalent to: $D \cdot 2^r = n \cdot G \; xor \; R$ (+,- and xor are equivalent)
   equivalent to: $D \cdot 2^r = n \cdot G + R$
   equivalent to dividing $D \cdot 2^r$ by G to obtain the remainder: $R = D \cdot 2^r \mod G$

   An alternative view of this method is to consider a polynomial $D(x)$ whose maximum exponent is $D - 1$ and where each coefficient is zero or one corresponding to the bits of D, then $R(x) = D(x) \cdot x^r \mod G(x)$ and the transmitted frame is $T(x) = D(x) \cdot x^r + R(x)$ which is divisible by $G(x)$ by construction. If some errors occur, they can be modelled by $E(x)$ where each non-zero coefficient identifies a position of error. The erroneous received messages is thus $T(x) + E(x)$ and the receiver will calculated $(T(x) + E(x)) \mod G(x)$ and as long as $E(x)$ is not divisible by $G(x)$, errors will be detected.

   A famous example is $G(x) = x^{16} + x^{12} + x^5 + 1$ which has the following properties:

   - detects every error of an odd number of error bits.
   - detects every 2-bit error (at any place in the frame).
   - Detects every single error burst of length $l \leq 16$. Actually, $G(x)$ of degree r detects any error burst of length $l \leq r$.

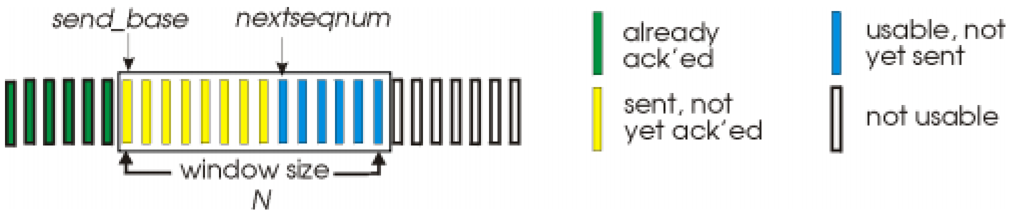
   It is not robust for $E(x)$ that are multiples of $G(x)$ as explained before.

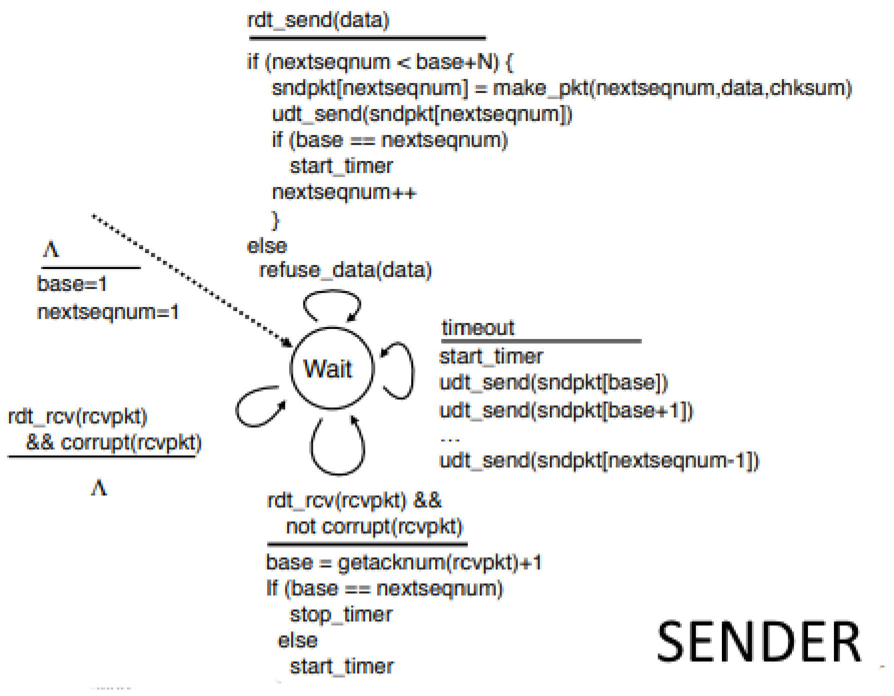
5 Reliable data transfer: GBN and SR sliding window protocols, and TCP error control improvements

It was seen in the course that a sending mechanism that operates in a "stop and wait" manner leads to low utilisations, such as with the alternating bit protocol. For this reason, people prefer to use pipelined protocols in which the sender allows multiple "in-flight" packets that are not yet acknowledged by the receiver. This allows to multiply by n (number of in-flight pkts) the utilization until the pipeline is entirely filled. **G**o **B**ack **N** (GBN) and **S**elective **R**epeat are two generic forms of pipelined protocols.

Go-Back-N protocol

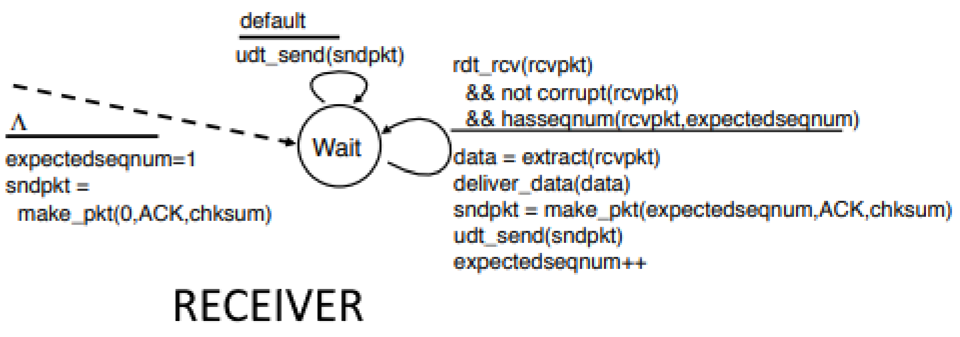
In this protocol, each pkt has a k-bit sequence number and there is a window of up to N consecutive unacked pkts. The acknowledgment $ACK(n)$ acks all the pkts up to and including pkt n. Notice that if ACK(n) is lost but ACK(n+1) arrives, pkt n will be considered as acknowledged because of the cumulative ACK principle. It works with a single timer that is conceptually for the oldest in-flight pkt. Timeout(n) forces to resend pkt n as well as all pkt having a higher sequence number and that are in the window.



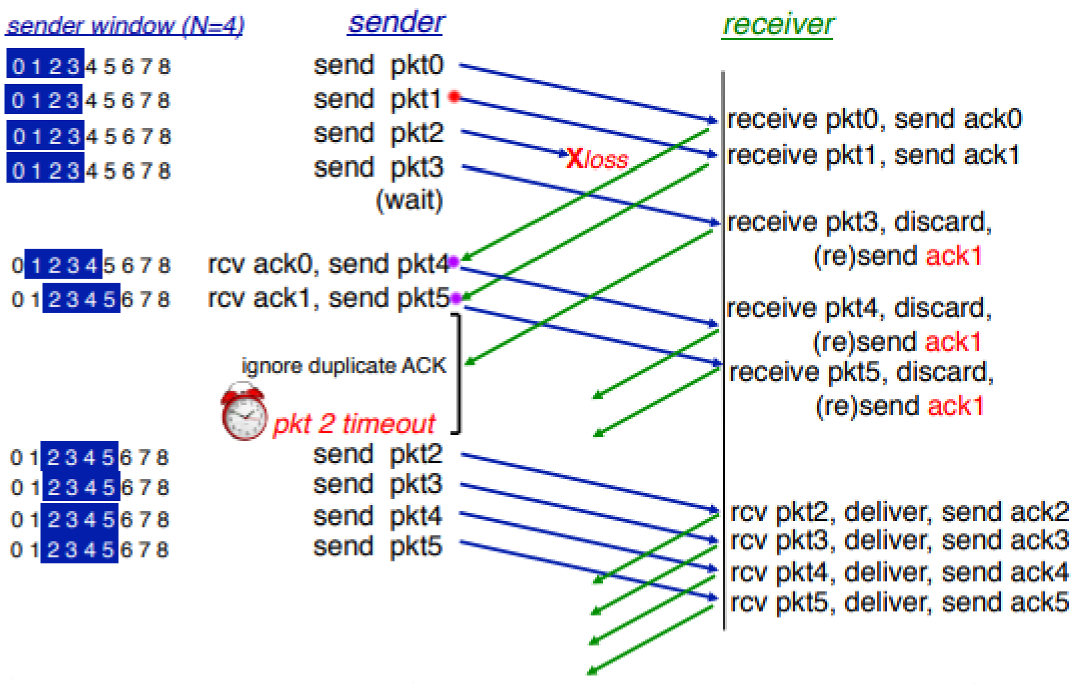


```
Λ
base=1
nextseqnum=1

rdt_send(data)
if (nextseqnum < base+N) {
    sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
    udt_send(sndpkt[nextseqnum])
    if (base == nextseqnum)
      start_timer
    nextseqnum++
    }
else
  refuse_data(data)

timeout
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
...
udt_send(sndpkt[nextseqnum-1])

rdt_rcv(rcvpkt)
  && corrupt(rcvpkt)
Λ

rdt_rcv(rcvpkt) &&
  not corrupt(rcvpkt)
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
   stop_timer
 else
   start_timer
```

ACK($n+1$) is not sent if pkt(n) was not yet ack'ed, so if we receive pkt($n+1$) while waiting for pkt(n), we resent the last ACK which was $n-1$. There is no buffer at the



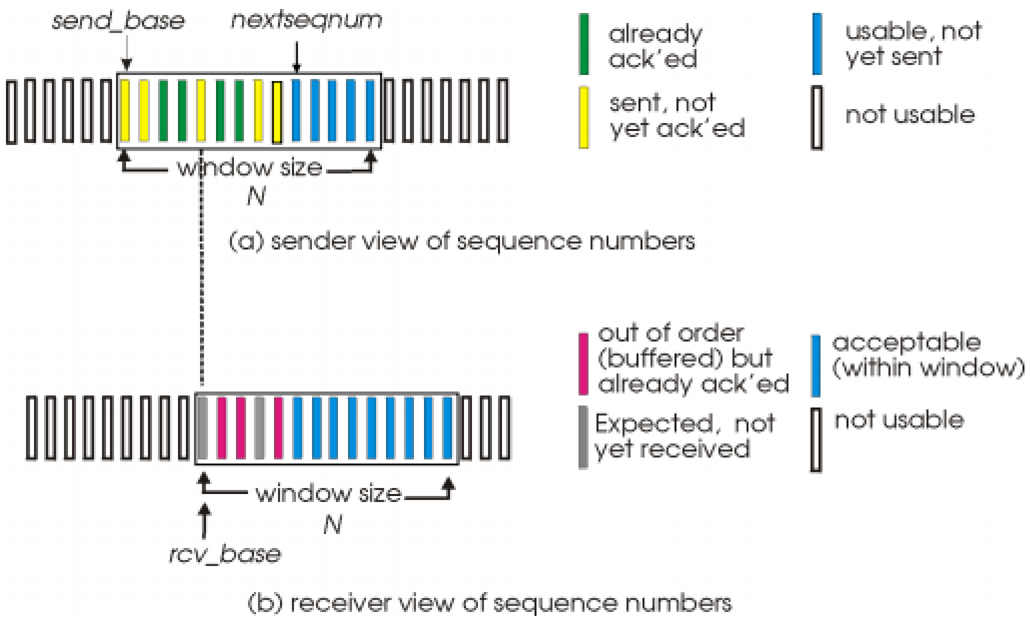
receiver side, an out-of-order pkt is thus discarded and the packet with highest sequence number that is in order is re-acked.
In the following Figure, one can see it in actions. The two purple dots represent moments where the timer is reset, delaying the detection of the lost pkt.



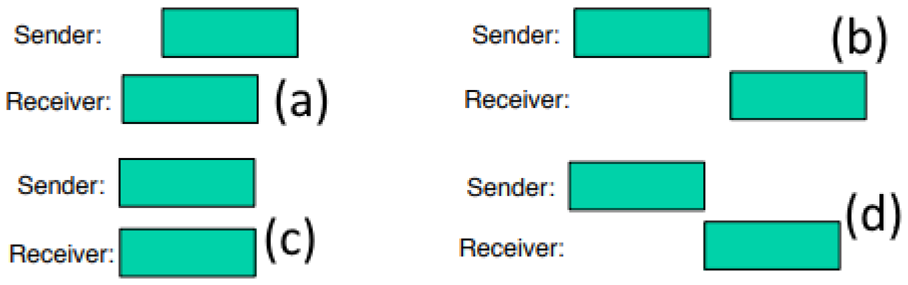
Let's now discuss the window size linked to this protocol. Let N the window size and K the number of different sequence numbers. N can't exceed K because it would lead to several in-flight pkts with the same number which would make it impossible. If it is equal, there are also failure scenarios, for example if all ACKs are lost, the receiver will resend everything starting from the first number and the receiver will re-accept them all but they are actually duplicates: $N \leq K - 1$ (if no reordering).

Selective Repeat protocol

With this protocol, the receiver individually acknowledges all correctly received pkts and buffers them if needed (out-of-order for example). The sender then only re-sends packets for which it has not yet received an ACK. There is thus a notion of sliding window and the position of that window can differ from sender to receiver:


send_base
nextseqnum
window size
N
already ack'ed
sent, not yet ack'ed
usable, not yet sent
not usable
(a) sender view of sequence numbers
out of order (buffered) but already ack'ed
Expected, not yet received
acceptable (within window)
not usable
window size
N
rcv_base
(b) receiver view of sequence numbers

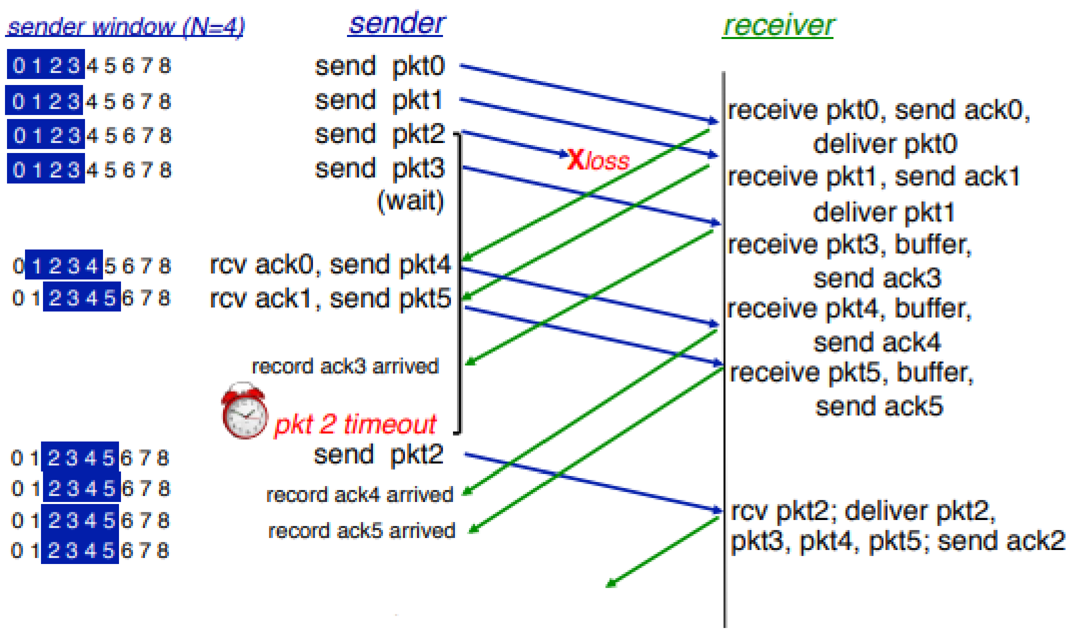
The sender side waits for the two first yellow one to slide its window and the receiver waits for the first grey one to slide. The following figure shows several window scenarios:



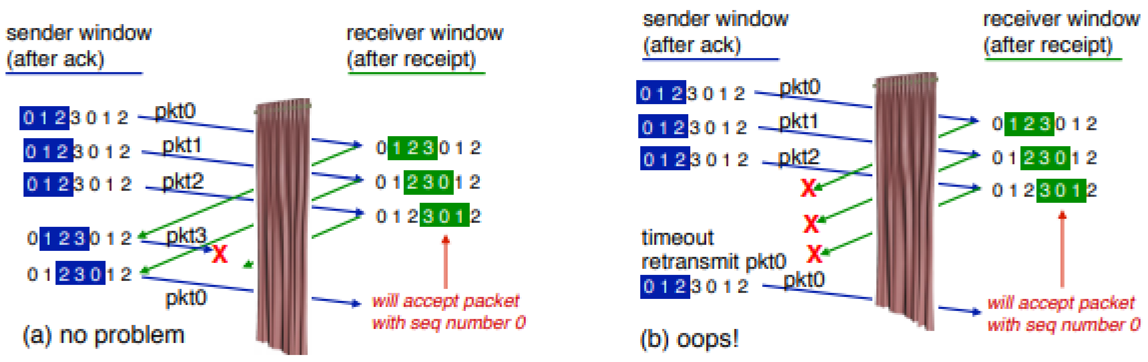
(a) is not consistent because the sender is ahead which would mean that is has received an ACK for a pkt that has not yet been received by the receiver.
(b) is not consistent because the receiver has received a given packet and acknowledged it but it was not yet sent by the sender.
(c) and (d) are the two limit cases: in (c), nothing has yet been sent nor received and (d) is the limit case in which all pkts are received but the sender has not yet received any ACK corresponding to the pkts that it has sent.

To sum up: for the sender, if it receives data from above, if the next available sequence number is in the window, send the pkt. If there is a timeout for pkt n, resend that pkt and reset the timer(n). If it receives ACK(n), mark the pkt as received and if n is the smallest unacked pkt, slide the window to the next unacked sequence number. For the receiver, if it receives a packet that is in its window, send the corresponding ACK. If it is in-order, deliver it above and slide the window to the next not yet received pkt. If it is out-of-order, buffer it. If the sequence number is lower than the base of the window, it means that the ACK was lost and we just re-send it. If n is higher than the last sequence number of the window, just ignore that pkt.

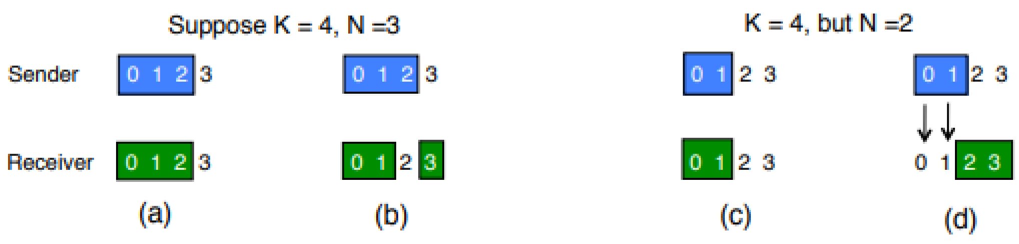
The next Figure shows it in action.


sender window (N=4)
sender
receiver
0 1 2 3 4 5 6 7 8
send pkt0
send pkt1
send pkt2
send pkt3
(wait)
X loss
receive pkt0, send ack0, deliver pkt0
receive pkt1, send ack1 deliver pkt1
receive pkt3, buffer, send ack3
receive pkt4, buffer, send ack4
receive pkt5, buffer, send ack5
rcv ack0, send pkt4
rcv ack1, send pkt5
record ack3 arrived
pkt 2 timeout
send pkt2
record ack4 arrived
record ack5 arrived
rcv pkt2; deliver pkt2, pkt3, pkt4, pkt5; send ack2

As for GBN, let's discuss the window size of this protocol. First note that $N \leq K-1$ is not OK for this pipeline because of the following example in which the receiver sees twice the same situation and can't discriminate:


sender window (after ack)
receiver window (after receipt)
0 1 2 3 0 1 2
pkt0
pkt1
pkt2
pkt3
X
will accept packet with seq number 0
(a) no problem
timeout retransmit pkt0
(b) oops!

The maximum window size is now: $N \leq K/2$ as illustrated by the following Figure:



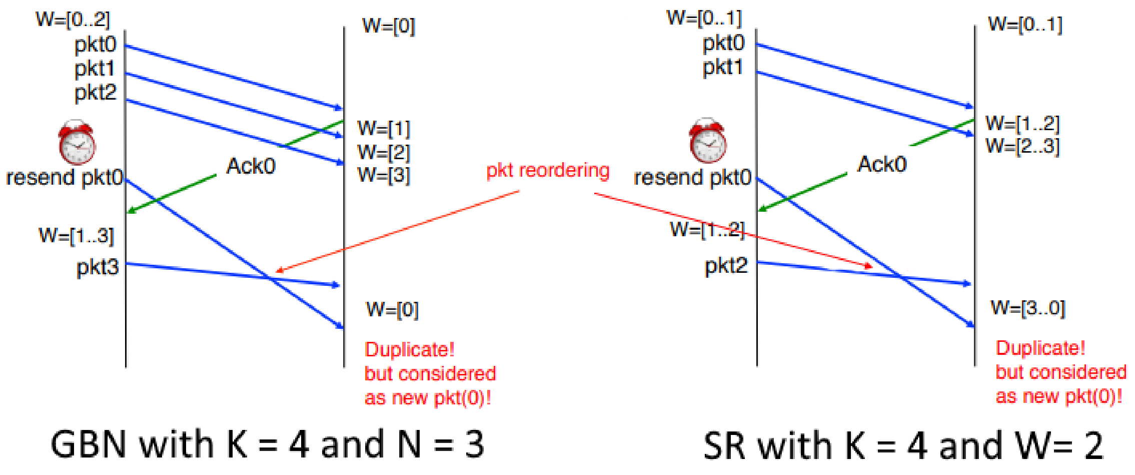
(a) Initial situation with a window of size N=3.
(b) After 3 frames have been sent and received but not ACK'ed.
The upper side of the rec window falls in the sending window
(c) Initial situation with a window of size N=2.
(d) After 2 frames sent and received but not ACK'ed.
The upper side of the rec window does not fall in the sending window

This can be summarized with the following general rule:

- sequence number works in modullo K
- sender window: N_S and receiver window N_R

- no reordering: $N_S + N_R \leq K$
- specific cases: GNR: $N_R = 1$ and SR: $N_R = N_S$

When there is re-ordering, these window sizes do not work as illustrated by the flowing Figure:



What about TCP ?

In *TCP*, sending ACK(n) means that we are expecting the n^{th} byte next. Losing an ACK is thus not a problem, if ACK(100) and ACK(120) are sent, if ACK(100) is lost, it will still be considered as ACKed.
There is also what is called **TCP fast retransmit** which is a way to discover lost packets faster. If a segment a lost, it is likely that several duplicate ACKs will be sent, the idea is to say that is the sender receives 4 ACKs for the same data (so a triple duplicate), it will resent that packet because it is probably lost. Why do we wait for 3 duplicates ? because if there is re-ordering, we might receive packet duplicates when the packet is not really lost, it is a surety level.

TCP also uses **delayed ACKs**, this allows to send only one ACK out of two if everything goes well and allows to decrease the load on the network. If we receive an out-of-order packet, we don't wait and directly send the corresponding ACK. (it is possible to buffer them, then one needs to set SACK which a selective ACK that allows to tell the sender which are already stored). If there is a segment that fills a gap, *TCP* also directly sends the corresponding ACK.

TCP mixes the good ideas of both protocols: from GBN it takes the cumulative ACKs and the single timer and from SR it takes the buffer at receiver, only retransmit the oldest pkt and SACK. Additionally, it adds the fast retransmit mechanism and the delayed ACK.

6 TCP timer, flow control, connection establishment and closure

TCP is a point-to-point pipelined reliable data transfer protocol. It is full-duplex and connection oriented.

TCP timer

TCP uses a single timer, as is done in the pipelined protocol GBN. When data is received from above at sender side, a segment is created with the correct sequence number. This number corresponds to the number of the first data byte in the corresponding segment. If not already running, the timer is set. It corresponds to the **oldest unacked segment** and expires after the *TimeOutInterval*. When there is a timeout, the corresponding segment (and only that one) is retransmitted and the timer is reset. When an ACK is received, update what is known to be acked and slide the window, if there are still unacked segments, reset the timer.

Flow control

By definition, flow control is the fact that the receiver controls the sender so that the sender does not overflow the receiver's buffer by transmitting too much too fast. Indeed, this would create a load on the network that is not even used by the receiver and this situation has to be avoided. In *TCP* it is done as follows: the receiver advertises the free space in its buffer by including *rwnd* value in the *TCP* header. The sender limits the amount of unacked ("in-flight") pkts to that value *rwnd* to ensure that the buffer does not overflow.

How to chose the *TimeOutInterval* ? It should be longer than the RTT of course, but the RTT is variable. If it is chosen too short, we will have premature timeouts and unnecessary re-transmissions but if it is too long, it will create delay and too slow reaction to losses. One must fin a way to estimate the probability density of the RTT, but over a network it is known to have to have a big variance. The way it is estimated is as follows, once in a while, a RTT is sampled (time between transmission and received ACK) and based on this, an exponential update rule is applied:

$$Estimated\_RTT = (1-\alpha)\ Estimated\_RTT + \alpha\ Sample\_RTT$$

this way, the impact of a sampled decreases over time and a bigger weight is given to recent ones. Typically, $\alpha = 0.125$.
The deviation around the estimation is estimated as follows:

$$Dev\_RTT = (1-\beta)\ Dev\_RTT + \beta\ |Sample\_RTT - Estimated\_RTT|$$

Typically, $\beta = 0.25$.

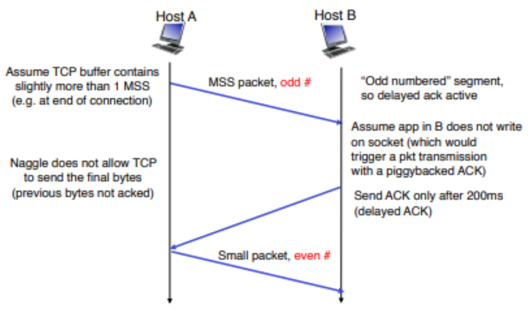
To have a sufficient margin, the timer is chosen as:

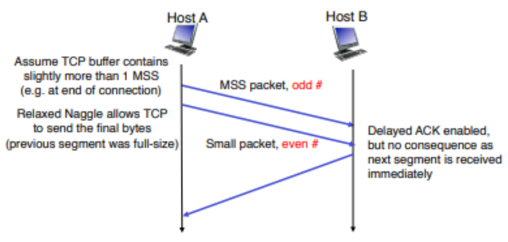
$$TimeOutInterval = \underbrace{Estimated\_RTT}_{\text{estimated average RTT}} + \underbrace{4\,Dev\_RTT}_{\text{safety margin}}$$

There are some sources of improvement:

1. Use larger windows: throughput limited by $rwnd/RTT$. There are 16 bits in the header to encode that, so the max value is 2^{16} bytes. This limitates the throughput to 64 kbytes per RTT. There is an option that allows to scale that factor by a factor $k \leq 14$ so the maximum size is actually 2^{30} bytes.

2. Naggle: to protect from poor programming at the application layer. When *TCP* receives data from the socket in units much smaller than the Maximum Segment Size (MSS): it first sends a small packet immediately and buffers the rest until the following ACK. The other segments are sent only when the equivalent of a MSS is buffered or when the ACK is received. /!\ can interfere with delayed ACKs, solution: relaxed Naggle: allows to send a packet smaller than the MSS if the previous one was one MSS long:


Host A
Host B
Assume TCP buffer contains slightly more than 1 MSS (e.g. at end of connection)
MSS packet, odd #
"Odd numbered" segment, so delayed ack active
Assume app in B does not write on socket (which would trigger a pkt transmission with a piggybacked ACK)
Naggle does not allow TCP to send the final bytes (previous bytes not acked)
Send ACK only after 200ms (delayed ACK)
Small packet, even #


Host A
Host B
Assume TCP buffer contains slightly more than 1 MSS (e.g. at end of connection)
MSS packet, odd #
Relaxed Naggle allows TCP to send the final bytes (previous segment was full-size)
Small packet, even #
Delayed ACK enabled, but no consequence as next segment is received immediately

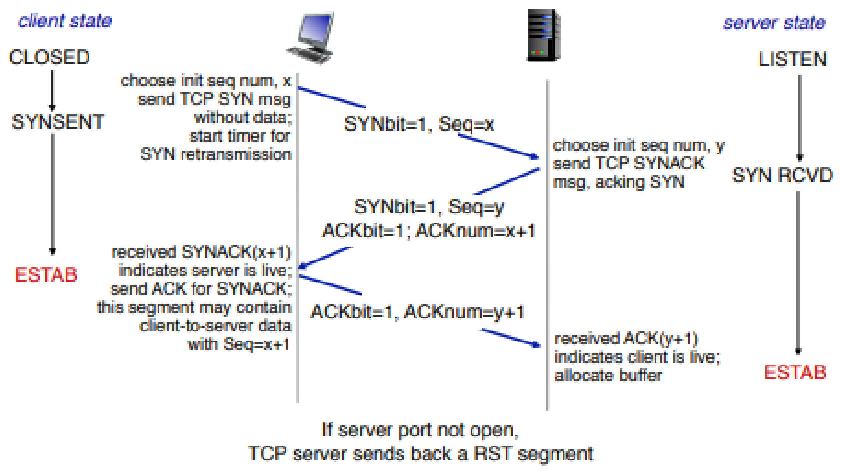
3. Silly window: to protect from poor programming at the application layer at receiver side. If we read byte by byte from the socket, a single byte will always be advertised but it is stupid to advertise such a bad *rwnd*. Clark's solution: receiver sends a window update if and only if the buffer is half empty or if a full segment can be received.

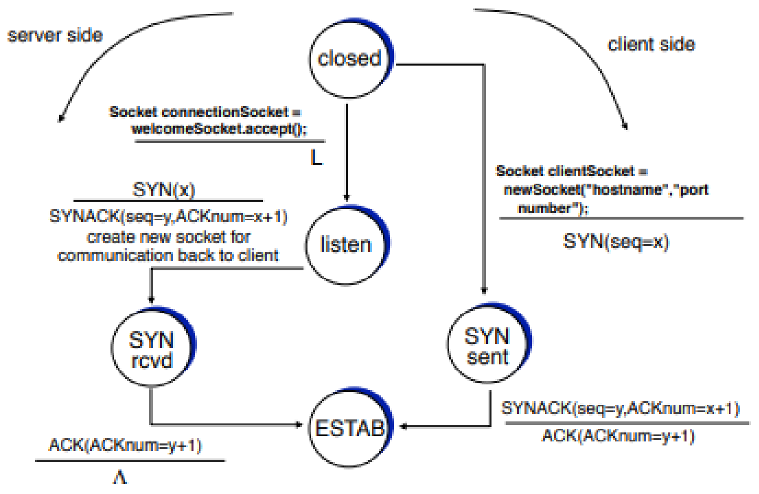
Connection management

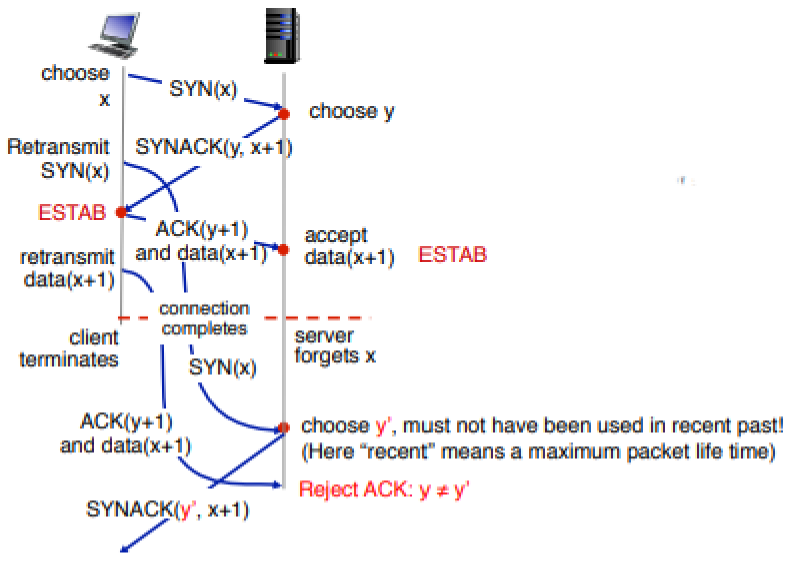
Before the data exchange, both sides agree to establish a connection (each knows that the other wants to establish a connection) and agree on the connection parameters (MSS, options: first seq nb, ..). It is done with a 3-way handshake because it is more robust than a 2-way handshake (because: i) multiple connection demands: DDOS or because of timeout ii) data sent to a ghost connection and the data is sent twice: what if transaction ? Might pay twice: Problem).
The following figure illustrates a 3-way handshake. There is a x and a y because it is a bi-directional communication and they both need to know where it starts. These numbers should not have been used recently in a formet connection between this client and this server, otherwise a still-alive connection could be confused with this connection. In practice, it is picked at random: $1/2^{32}$ small chance that it was used recently and random is good for security.
Only when both have "ESTAB" state memory is allocated to the buffers.

The following scenario shows that it is more robust than a 2-way handshake. As $y \neq y'$, a malicious person can't receive this because it does not know y'.

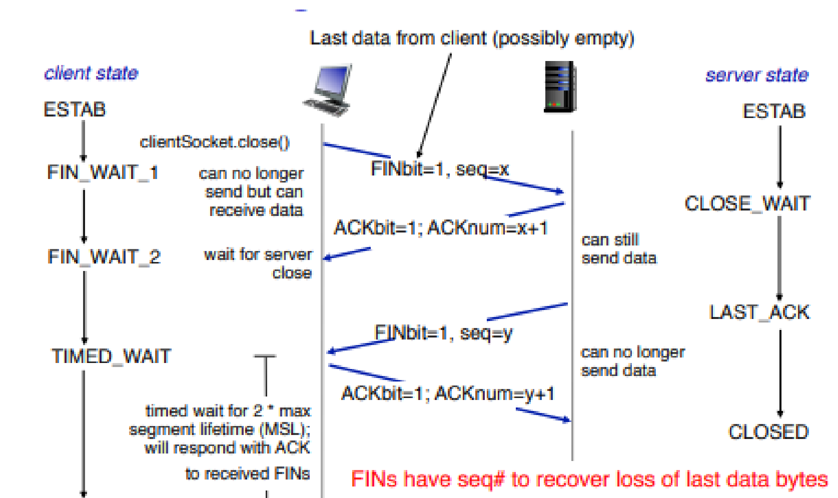
In order to close the connection, client and server close their "sending" side of connection. The last segment will have the bit FIN set. When responding to a received FIN, the other side can set FIN as well in the ACK segment. It is done as follows: in this scenario,



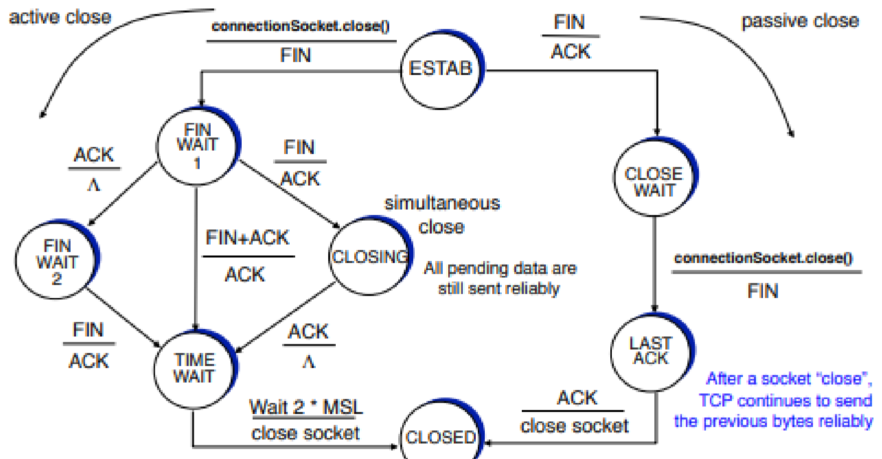




the client has nothing more to send and stops at sequence x, but the server maybe still has data to send so its closing is separate. Potential problem: client sends FIN → disappears → server sends FIN → never ACKed → resend fin → .... To solve it, the client must stay at least 1 MSL after closing its connection to ACK the FIN of the other side.



FINs have seq# to recover loss of last data bytes
ACKs are sent when all data received. So there are also retransmissions when FINs are not acked



Side that actively closes socket waits a double MSL (safety margin). Therefore same TCP 4-tuple cannot be reused sooner (e.g. with same client port#). It ensures that no duplicate segment from an earlier connection with the same 4-tuple can jump in the current connection. But this may lead to some starvation of resources.

A few questions:

- What if last FIN and retransmissions are lost ? Need to disconnect after k attempts.
- But what about other side? add a hear beat mechanism to detect that the other side has disconnected (send something even if idle, if not sent: means it is closed).
- What if heartbeat msg are lost ? no perfect solution, it can continue over and over like this.

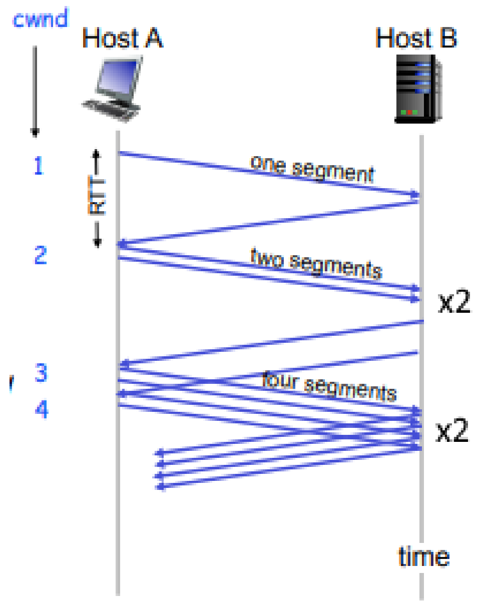
7 TCP congestion control and its derived properties

Unlike flow control which aims at handling the receiver side, congestion control deals with moments where the sender wants to send faster than the throughput which might overflow the sender's buffer. It is manifested by long delays and lost packets.

The goal is to say that *TCP* sender should transmit as fast as possible but without congesting the network. It is decentralized meaning that each *TCP* segment sets its own rate based on an implicit feedback which consists of the ACKs (positive) and lost segments.

The system is the following: **additive increase multiplicative decrease** (AIMD). The additive increase means that the congestion window *cwnd* is increased by 1MSS every RTT until a loss is detected and this value is cut in half after a loss. The idea is to be more aggressive when reducing the window size to avoid instability.
One must notice the difference between *cwnd* and *rwnd* where the latter is the free buffer size at receiver. When taking congestion into account, the sender is limited to the smallest of both values.

At the beginning of the connection, there is the **slow start** in which the rate is increased exponentially until the first loss as shown in the following Figure:



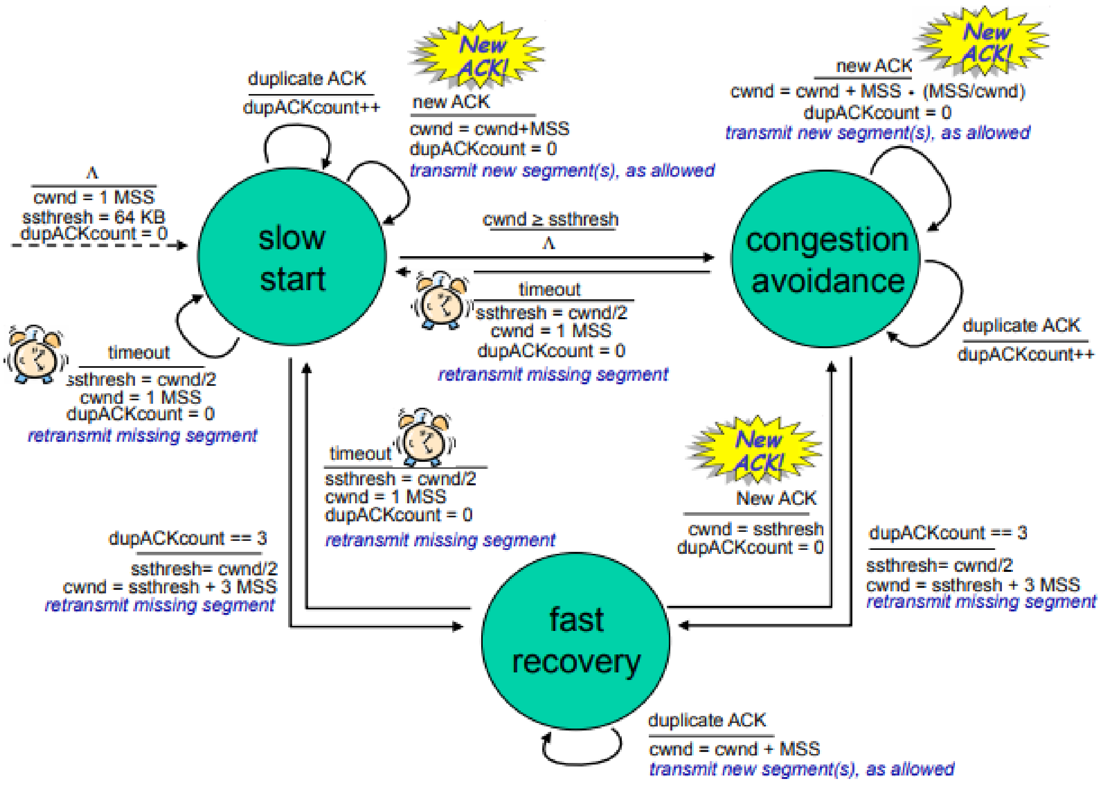
What happens when there is a loss is the following in *TCP RENO*:

- indicated by triple ACK: the facts that ACKs passed on network means that it is able to deliver segments: it is not that bad. Then *cwnd* is cut in half and grow linearly after.
- indicated by timeout: *cwnd* is reset to one and the slow start is relaunched until the threshold, then it grow linearly. What is this threshold ? half of the size where the problems happened.

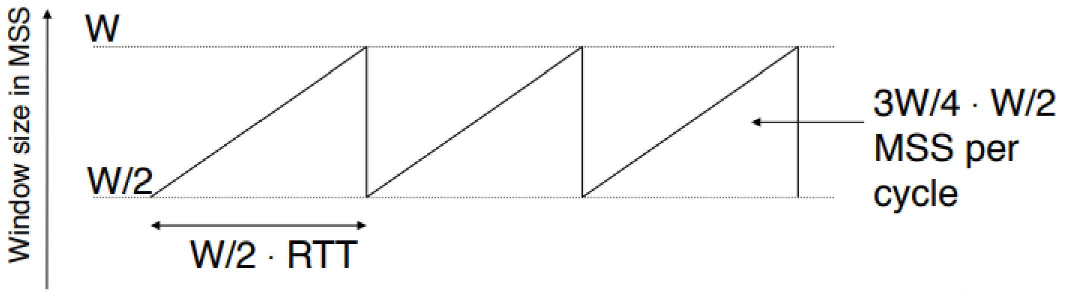
The earlier version of *TCP* (*Tahoe*) sets it to 1 for each case.

The linear growth is implemented as follows: $cwnd \to cwnd + \frac{MSS}{cwnd} MSS$ for each ACK. This is equivalent to increase it by 1 at each RTT, but the fraction of each segment depends on the current window size.

This can be summarized as a Finite State Machine:



What happens at steady state ? Saw-tooth behaviour as shown in the following figure. The average window size is thus $\frac{3W}{4}$. One can compute a steady-state "goodput"



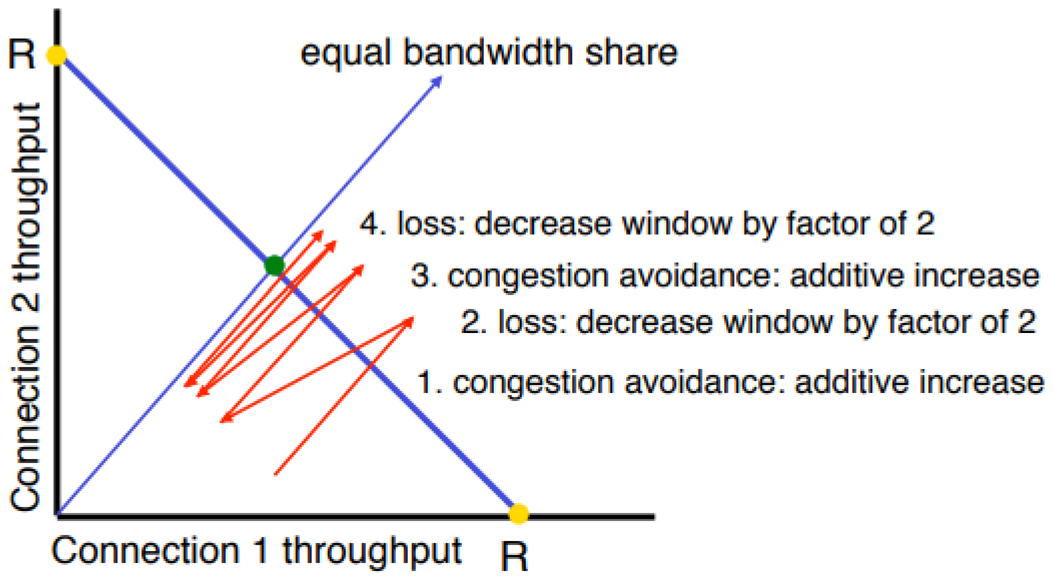
(throughput without headers and retransmissions). The number of MSS per cycle is $\frac{3W}{4} \cdot \frac{W}{2} = \frac{3W^2}{8} = \frac{1}{p}$. (A cycle lasts $\frac{W}{2} RTT$ because it increases by one at each RTT) where p is the packet loss ratio. Indeed, assume one pkt is lost at the end of the ramp only, it is $\frac{1}{nb\_MSS}$. So, we have: $W = \sqrt{\frac{8}{3p}}(*)$.
The average "goodput" (in MSS/sec) is the average window divided by one RTT:

$$Avg\_goodput = \frac{3W}{4RTT} = \sqrt{\frac{3}{2}} \frac{1}{RTT\sqrt{p}},$$

by inserting the formula for W(*). It can be equivalently expressed in bps by multiplying it by the MSS expressed in bits.

Conclusion: i) Larger segments: better throughput ii) lower p better goodput (logical) iii) long distance: RTT increases: lower goodput (expected as well).
It is a first approximation but allows some good analysis.

Let's now discuss **TCP fairness**: if k TCP sessions share the same bottleneck in a link of bandwidth R, each should have an average rate of R/k. Why is it fair ?


R
equal bandwidth share
Connection 2 throughput
4. loss: decrease window by factor of 2
3. congestion avoidance: additive increase
2. loss: decrease window by factor of 2
1. congestion avoidance: additive increase
Connection 1 throughput
R

The yellow points represents points where one of the sessions gets everything and the green dot is where the share is equal. If we go below the feasible region (inside the triangle), queue will grow at the router and congestion appears. Here, they both increase at the same time because they see no congestion. Then congestion appears and both have pkt losses so they both divide by 2: it is fair.

This example is when both have the same RTT, otherwise the bandwidth share is inversely proportional to the RTT because the ramping slopes are not the same.

Note that fairness is not enough to achieve equity because an app can open several // connections between two hosts which would increase its share of the bandwidth.

8 Network layer: Data plane versus control plane, IP addressing and forwarding, router architecture

The network has two key functions: **forwarding** which consists in moving a packet from router input to appropriate router's output and **routing** which is the determination the route that the packets have to take to go from a destination to a source. If we do an analogy with a holiday trip, forwarding is saying "Now i'm at the airport, which gate should I take?" and routing is "I first take the bus, then the train, then the airport, ...". Well similarly, one can see it as the data plane versus the control plane:

1. Data plane: it is local and per router. It determines how a datagram arriving on the router input port is forwarded to the router output port, done via a **forwarding table**.
2. Control plane: it is a network wide logic. It determines how a datagram is routed among all the routers from the source to the destination. This is either done via traditional **routing tables** or via software-defined networking but this was not studied in the course. The control plane can be "per-router" meaning that all routers communicate and compute their own tables or it can be "logically centralized" in which there is a remote controller that interacts with all the routers and this controller interacts with local control agents (CAs). This is often done is smaller networks (like a data center for example).

An *IP* address is a 32-bit identifier for a host or a router interface. What is called an interface is a connection between the device and a physical link. Routers have several interfaces while hosts usually have only one or two. There is one such address per interface. A subnet is a part of the network in which devices can interacts without intervening routers. The high order bits of the *IP* address correspond to the subnet and the host is defined by the low order bits.

Historically, there were **classful addresses**, the network portions of the *IP* address were constrained to be either 8, 16 or 24 bits which were known as class A, B and C. The longer the network part, the smaller the number of bits for the host and thus also the smaller the number of hosts that could be in the network. Now it is classless, called **C**lassless **I**nter**D**omain **R**outing (CIDR) and the subnet part can be arbitrary long. At the end of the address, the number of bits corresponding to the subnet is given, for example an address could be `223.1.1.0/24` meaning that the 24 first bits are used to identify the subnet.

Now, how is **forwarding** actually implemented. If we assume that it is only done based on the destination address, there are 4 billion possibilities (2^{32}) different addresses and the forwarding tables would just be gigantic. All the addresses in the same subnet can be aggregated into a single entry in the table which allows to save space in memory and to make the look-up easier. But then, one must handle the overlaps that can occur in the forwarding table and that is done with **longest prefix matching**: when looking for forwarding table entry for a given destination address, use the longest address prefix that matches the destination address. The default address is thus only chosen if the mask has a size zero otherwise there is always a longer prefix.

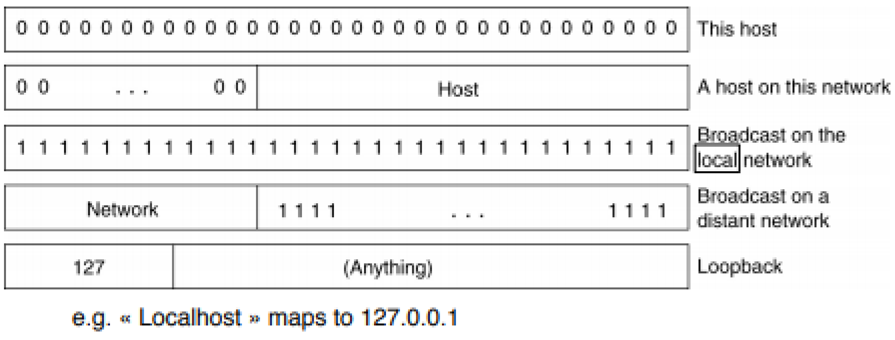
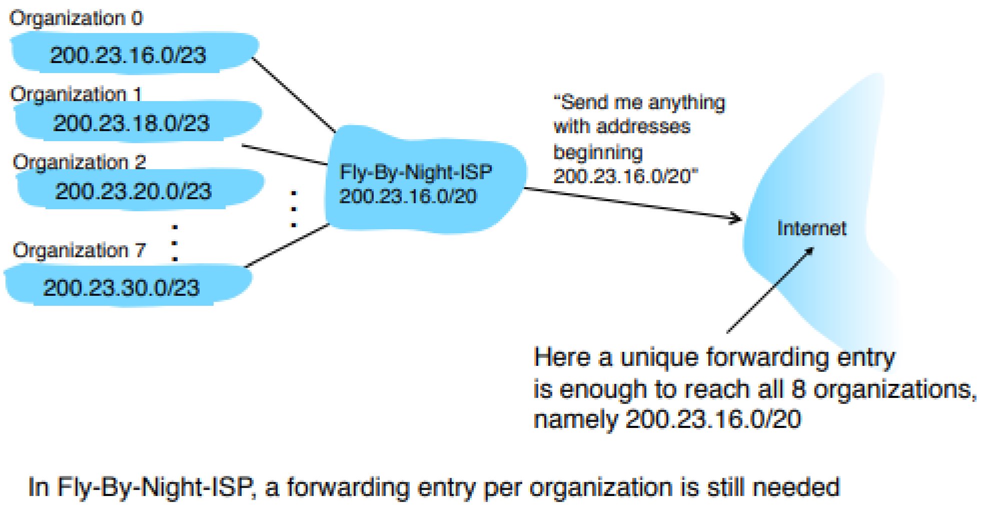


Figure 17: List of reserved *IP* addresses.



Here a unique forwarding entry is enough to reach all 8 organizations, namely 200.23.16.0/20

In Fly-By-Night-ISP, a forwarding entry per organization is still needed

Figure 18: Regrouping the subnet in a single entry.

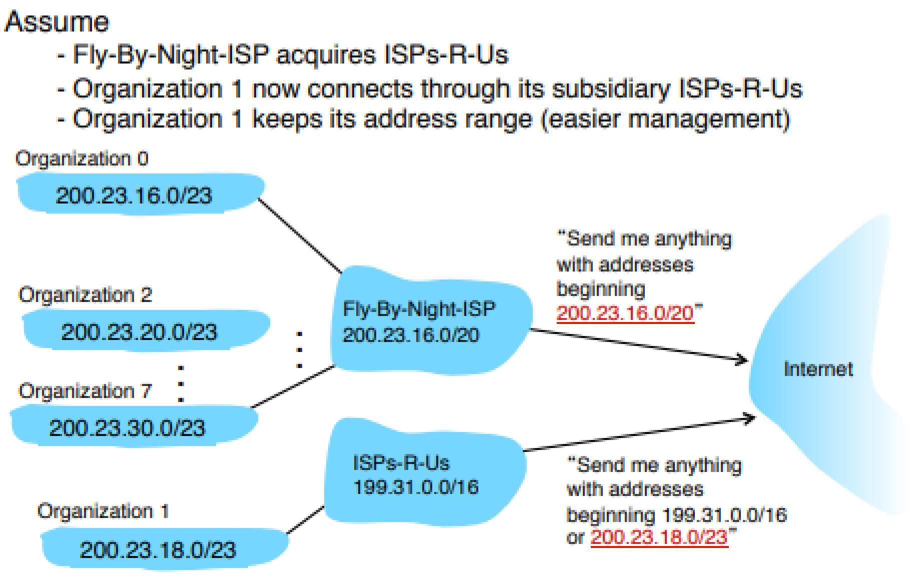
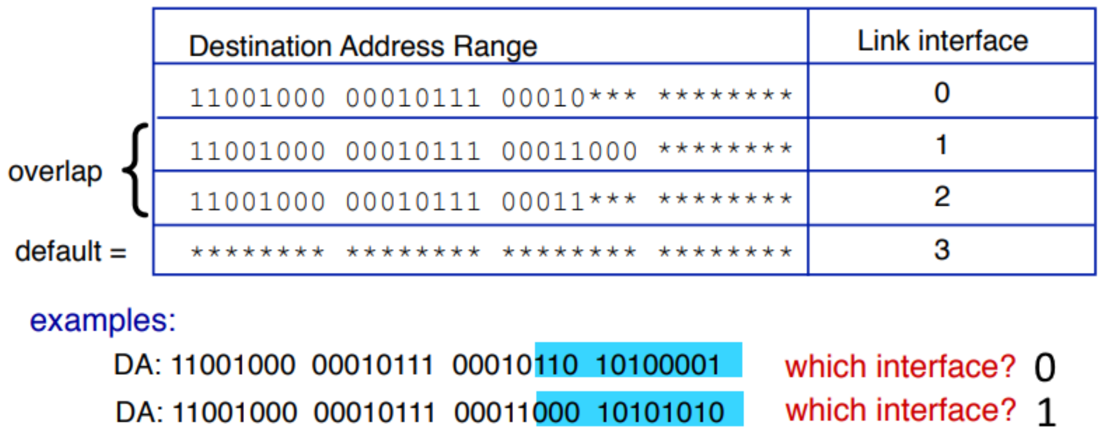

Assume
- Fly-By-Night-ISP acquires ISPs-R-Us
- Organization 1 now connects through its subsidiary ISPs-R-Us
- Organization 1 keeps its address range (easier management)

Figure 19: What if a company moves but keep the same addresses for simplicity.



	Destination Address Range	Link interface
	11001000 00010111 00010*** ********	0
overlap	11001000 00010111 00011000 ********	1
overlap	11001000 00010111 00011*** ********	2
default =	******** ******** ******** ********	3

examples:

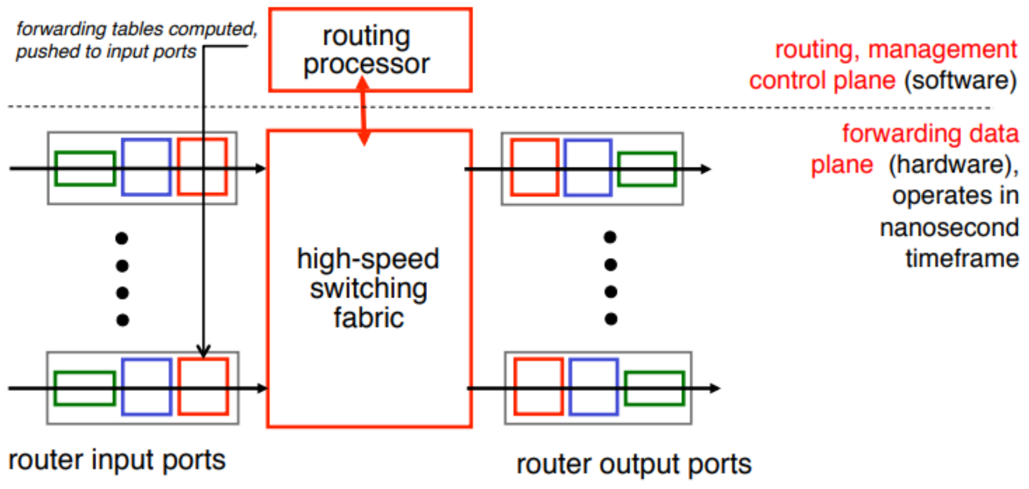
DA: 11001000 00010111 00010110 10100001 which interface? 0

DA: 11001000 00010111 00011000 10101010 which interface? 1

Figure 20: Longest prefix matching.

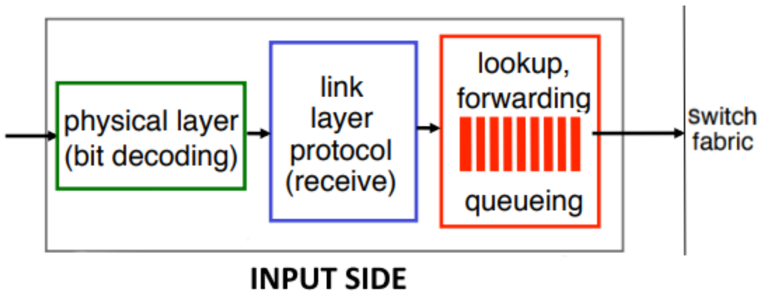
Doing this makes sense if entries with a same prefix are in the same area because it then allows to reduce the size of the tables. But it can lead to performance issues as it is done by adding bit by bit and and then going deeper in the addresses.

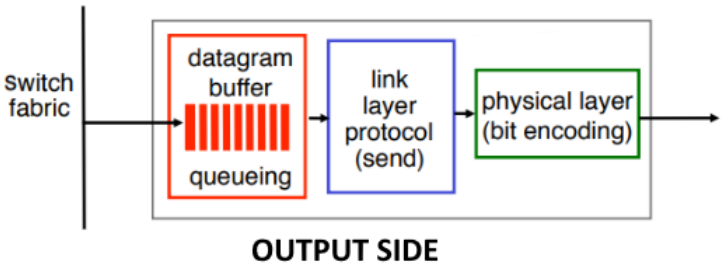
Let's now finally see what is inside a router. The following pictures shows a general view of it, each port is of course in and out at the same time.

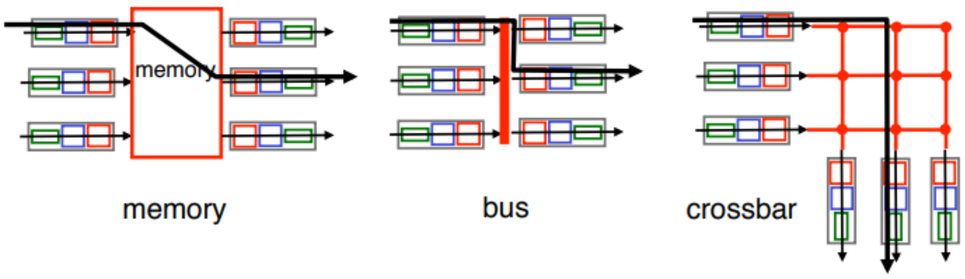

forwarding tables computed, pushed to input ports
routing processor
routing, management control plane (software)
forwarding data plane (hardware), operates in nanosecond timeframe
high-speed switching fabric
router input ports
router output ports

- Physical layer: bit level reception
- Link layer protocol: interprete it, for example if it is sent over *Ethernet*.
- The third block is a decentralized switching. Based on the *IP* header fields values, look for the right output. It should be done at line speed for performance and this also updates the TTL, the packet count etc..
- buffering is required at the output when datagrams arrive from switching fabric faster than the transmission. There can be fragmenting and there must be a scheduling displine that is encoded (FIFO for example).

The switching fabric transfers the packet from input buffer to appropriate output buffer. There are three types, either **memory** but this is bad for high speed, via a **bus** on which the speed depends on the bandwidth of the bus or with a **interconnection network** which overcomes the bus bandwidth limitations.







9 Network layer: DHCP, NAT, IP fragmentation, IPv6, ICMP

DHCP

It stands for **D**ynamic **H**ost **C**onfiguration **P**rotocol and aims at answering the question "How does a host get an *IP* address ?". A wide overview is given here:

- Host broadcasts "*DHCP* discover" msg
- DHCP server responds with "*DHCP* offer" msg
- Host requests an IP address "*DHCP* request" msg
- DHCP server sends address "*DHCP* ack" msg

The two first messages are not mandatory because it can be that we just want to extend the live time of our address, not need for discovery.

Note that it musts use *UDP* for two reasons. First, it does a broadcast for the discover msg but *TCP* is one to one. Secondly, to create a *TCP* connection, a socket is created, but on which *IP* address can we do it if we have none ? The broadcast is done in the subnet, so all the *DHCP* server of the subnet will respond. Note that *DHCP* can also give the address of the first-hop router for a client, the name and *IP* address of the default *DNS* server, the network mask,.. It is not only there for getting an *IP* address.

NAT

It stands for **N**etwork **A**ddress **T**ranslation. It allows to reduce the size and computation linked to the forwarding tables. The motivation behind it is to give only a single address to a local network but to have several addresses inside the network itself.

- Range of addresses not needed from the ISP, just do it locally as you want to.
- One can change the private addresses of devices in local network without needing to notify the external world
- Keep the same private addresses even if changing from ISP
- Devices in the network are not directly accessible: additional layer of security (possibility to restrict incoming traffic)

A *NAT* router must:

- Replace (source *IP* address, port nb) of outgoing datagrams to (*NET IP* address, new port nb) so that the client can respond to the address of the *NAT*.
- Remember every translation from (source *IP* address, port nb) to (*NET IP* address, new port nb)
- replace again for incoming datagrams.

There is one problem linked to *NAT*, that is the traversal problem. When a client wants to connect to a server that is inside a NATed network. The server usually answers, so it has not sent anything to the router so far and there is no entry in the *NAT* for the server yet, which will result in the packet being dropped. Several proposed solutions:

- Statically configure the *NAT* to forward incoming connection request at a given port
- Universal Plug and Play Internet Gateway Device Protocol which allows the *NAT*ed host to learn the public address and to add or remove the port mappings. It is an automation of the previous solution.
- Relaying (what Skype does). The server establishes a connection to a relay and the external clients connects to the relay as well. the relay then bridges packets.

It is a bit controversial, because it violates the different layering principle. Indeed, if a port number is encrypted, how can the *NAT* understand it or how can it modify it as it does not know the key. That is why in principle, each layer should only work on its layer and a router should thus only operate at the network level. The address shortage should thus rather be solved by *IPv6* which has much more available addresses. Furthermore, with *NAT*, the source and destination change over time which is also not liked by purists.

IP fragmentation

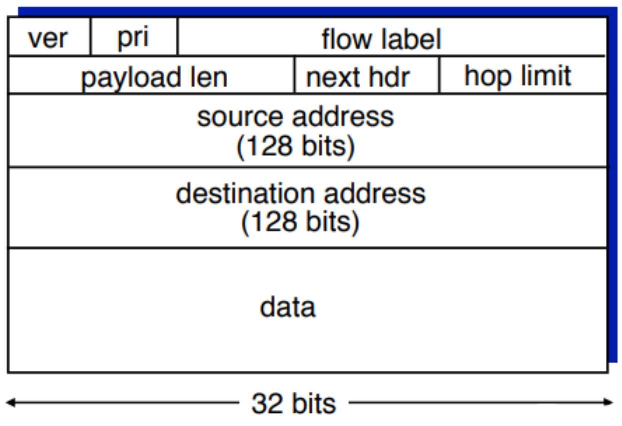
The network links have a max transfer units (MTU) which is the largest possible frame that can be carried. So, large *IP* datagrams are fragmented and all these separated frames are re-assembled at the end (to not repeat the process several time for different links and to reduce the number of overheads and byte that are carried in the network, also it would take time to wait for all fragments at every node). A fragmented datagram can be re-fragmented if needed.

In practice, it is done thanks to 2 variables in the header: *fragflag* which is 0 only if it is the last segment (so it is 0 if there is no fragmentation) and *offset* which gives the position of that fragment in the re-construction. When *offset* = 0, it means that it is the first fragment. Having them both to zero means that the first fragment is also the last one, so that there was no fragmentation.

To avoid fragmentation, the source must send small enough datagrams but it must then know the minimal MTU of the path. discovering the MTU along a path is done with trial and error by sending packet with "Don't Fragment flag" that is set, if the source received an *ICMP* error message it tries again with the MTU indicated in the ICMP packet.
Yet this is not perfect as it relies on routers properly returning the *ICMP* messages, congestion problem could discard the *ICMP* messages or maybe the route may change.

IPv6

Motivation: 32-bit address space soon completely allocated. Also, new header forward helps speed processing and forwarding. The change should not affect other layers. The header is 40 bytes long and no fragmentation is allowed at routers (only at end points).



Flow label allows to identify datagrams from a similar flow, but the notion of flow is not well defined. Next header allows to chain several headers if ones would like to extend the header, it could be used to identify upper layer protocols. The checksum is removed to reduce the processing time at each hop. Options is still allowed, but not in the header (via the next header field) and finally, it comes with new *ICMP* messages.

The addresses are 128 bits long written as: x:x:x:x:x:x:x:x where each x denoted a hexadecimal number. Leading zeros in a field are optional and a series of zeros can be encoded as :: (only once per address to remove ambiguity when reconstructing). A prefix of 64 bits is used to identify a site and a suffix of 64 bits is used to identify an interface on that site.

To perform the transition of one protocol to the new one, a solution must be found when a routers is not yet updated to *IPv6*. This can be done by **tunneling**: the *IPv6* is carried in the payload of a *IPv4* datagram.

ICMP

It stands for **I**nternet **C**ontrol **M**essage **P**rotocol. It is used by routers and hosts to communicate network level information. It can be of the type of error reporting (unreachable host, network, port, protocol,..) or it can be a request/reply (ping). It contains a Type, a Code and a description.

Such messages can be used to perform a traceroute by sending series of *UDP* segments with bigger and bigger TTL. When the n^{th} datagram arrives at the n^{th} router, it will be discarded and the source is sent an *ICMP* message explaining that the TTL expired. It stops either when the destination is reached (if the source receives an *ICMP* message telling that the port was not reachable) or when the client stops.

10 Routing metrics, and Link-State routing (and OSPF)

Routing metrics allows to set the cost of this optimization problem. Link costs can be engineered to optimize the network to some extend but this usually requires to know the traffic matrix TM (for a pair of nodes i, j it gives the amount of traffic entering at i and exiting at j). According to which routing metric is chosen, several objectives can be achieved, for example to achieve minimum hop routing: just set the costs of each link to 1. This also minimizes the average link load but not the delay nor the congestion:

$$score = Avg\_link\_load = \frac{\sum_{i \in links} load_i}{N}$$

minimizing the average load is equivalent to minimizing the sum of all the link loads. So, remove denominator N from score. Routing a new flow of rate R will increase the load as:

$$score\_increase = \sum_{i \in Path} R = R \cdot nb\_hops(Path)$$

because each link on the new path will see its rate increase by R. One directly sees that to minimize it, the idea is to minimize the number of hops.

Another example of routing metric is the InvCap which minimizes the link utilisation:

$$score = Avg\_link\_util = \frac{\sum_{i \in links} util_i}{N} = \frac{\sum_{i \in links} \frac{load_i}{capacity_i}}{N}$$

$$score\_increase = \sum_{i \in Path} \frac{R}{C_i} = R \cdot \sum_{i \in Path} \frac{1}{C_i}$$

and therefore the cost of each node is the inverse of the capacity. This actually makes more sense than the previous one: higher capacity, less cost so that traffic is attracted to paths with a good capacity (to avoid overloading).

Some other metrics are:

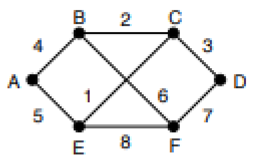
- Link delay metric: minimizes the delay but the delay can come from several causes: propagation, transmission (pkt size / capacity $\rightarrow$ back to invcap), queuing delay
- Administrative link cost

It can basically be any summable quantity.

Link state routing

The principle is to represent the network as a graph and then the least cost path is computed. Each node computes a least cost path to all other nodes thanks to the Djikstra's algorithm and this gives the forwarding table for that node.

A link state packet is composed of a source node, a sequence number and an age. These packets are flooded selectively (not forwarded on the link from which they arrive and duplicates/older packets are detected by sequence number and not forwarded). The packets are then ACKed. The flooding should not be done naively otherwise some packets may end up travelling forever. For example, in the following topology, this is an example for node B. In that topology, D will send to C and F which will both send to B, B will keep only one as they share the same sequence number. Also, routers do not forward directly but put them in a buffer (holding area) for a short delay, this way they can receive the other messages (like from D via C after having received from D via F which are equivalent) and see the direction of the flood and to follow it.



Packet received from D via C and F

The packet buffer for router *B*

			Send flags			ACK flags			
Source	Seq.	Age	A	C	F	A	C	F	Data
A	21	60	0	1	1	1	0	0	
F	21	60	1	1	0	0	0	1	
E	21	59	0	1	0	1	0	1	
C	20	60	1	0	1	0	1	0	
D	21	59	1	0	0	0	1	1	

Clearly, routers do not forward the received Link State packets immediately but put them for a short while in a **packet buffer (holding area).** Why?

A few problems and corresponding solutions are now listed:

- What if the sequence number wraps around? choose a 32-bit sequence (137 years to wrap around: OK)
- What if router crashes ? restarts with sequence number 0 and all its pkts are ignored until the sequence nb reaches previous value.. Solution: age field decremented by one every second and the entry is removed every time it reaches zero (as if we had never seen it $\rightarrow$ re-accept its data).
- Sequence number is corrupted ? same consequence, same solution. (example: receive seq nb 50 but we were at 20, it will ignore 21,22,... up to 50. With ageMax = 60 sec, we re-accept after 1 minute: OK.

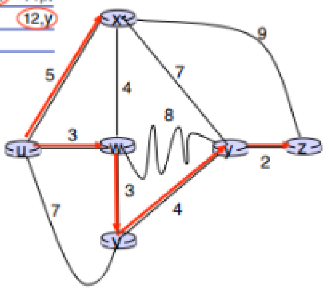
Djikstra's algorithm is explained and illustrated below:

```
Initialization:
 N' = {u}
 for all nodes v
   if v adjacent to u
      then D(v) = c(u,v)
   else D(v) = ∞

Loop
  find w not in N' such that D(w) is a minimum
  add w to N'
  update D(v) for all v adjacent to w and not in N' :
     D(v) = min( D(v), D(w) + c(w,v) )
  /* new cost to v is either old cost to v or known
   shortest path cost to w plus cost from w to v */
until all nodes in N'
```

Step	N'	D(v) p(v)	D(w) p(w)	D(x) p(x)	D(y) p(y)	D(z) p(z)
0	u	7,u	3,u	5,u	∞	∞
1	uw	6,w		5,u	11,w	∞
2	uwx	6,w			11,w	14,x
3	uwxv				10,v	14,x
4	uwxvy					12,y
5	uwxvyz					

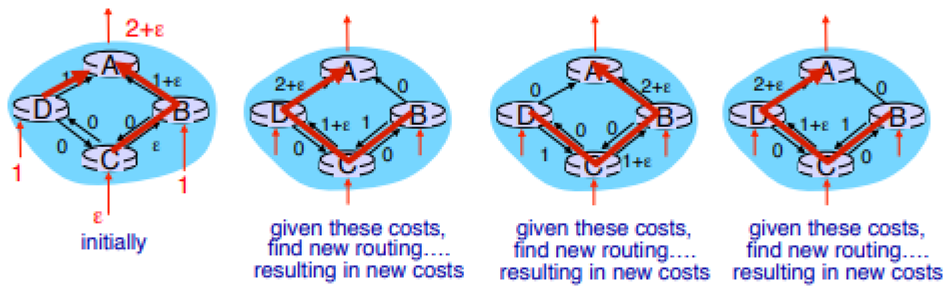
notes:

- construct shortest path tree by tracing predecessor nodes
- ties can exist (can be broken arbitrarily)



If a link is down, the tree is updated and another path is found. With n nodes, the complexity is $\mathcal{O}(n^2)$ because there are n steps and each steps requires comparisons with n then $n-1$ then $n-2$... other nodes. It can nevertheless be implemented in $\mathcal{O}(n \log n)$.

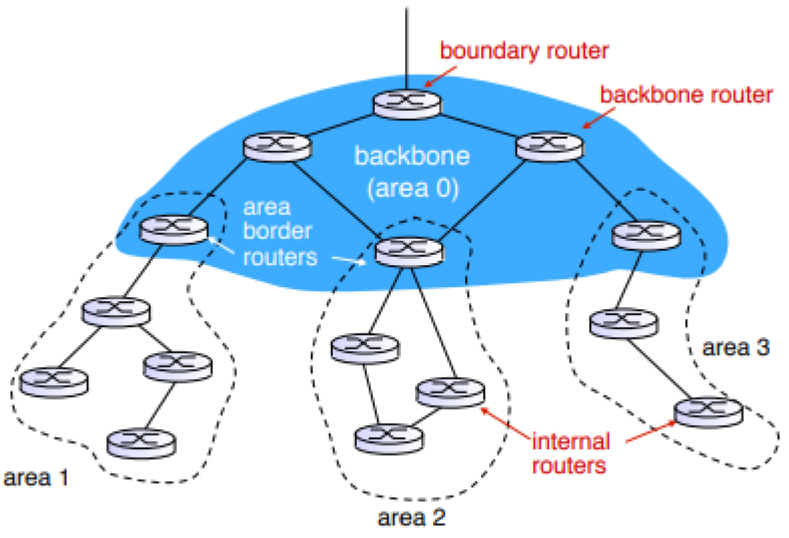
Calculating the path like this sometimes generates oscillations as shown in the following figure:


2+ε
A
B
C
D
1+ε
1
ε
initially
given these costs, find new routing.... resulting in new costs
given these costs, find new routing.... resulting in new costs
given these costs, find new routing.... resulting in new costs

Open Shortest Path First (OSPF)

This protocol uses link states algorithms. The messages that are sent are called LSAs: Link-State Advertisements and are carried directly over *IP*. There are some advanced features:

- There is a **designated router**: when there are several routers connected to the same subnet, one of them can be designated to exchange LSAs with the others (allows to reduce traffic). When there are N nodes at the side of a subnet, each of them sends N-1 messages to all the others while with designated routers, the M-1 send only to the designated router.
- LSAs can be authenticated for security.
- Multiple same cost path allowed, this allows to balance the load. But all the packets of a given flow have to follow the same path because the paths could actually be different when considering another metric.
- In large domain, hierarchical OSPF can be used. The big domain is divided in smaller ones and a backbone is used to connect the different areas. This is good because flooding on a small area produces less traffic than in a huge one and compute n small Djikstra's is faster than one big.



11 Distance Vector routing (and RIP)

This protocol is another manner to obtain a routing table and is based on dynamic programming. It uses the Bellman-Ford equation which states that if $d_x(y)$ is the least cost path from x to y, then:

$$d_x(y) = \min_v \{c(x,v) + d_v(y)\}$$

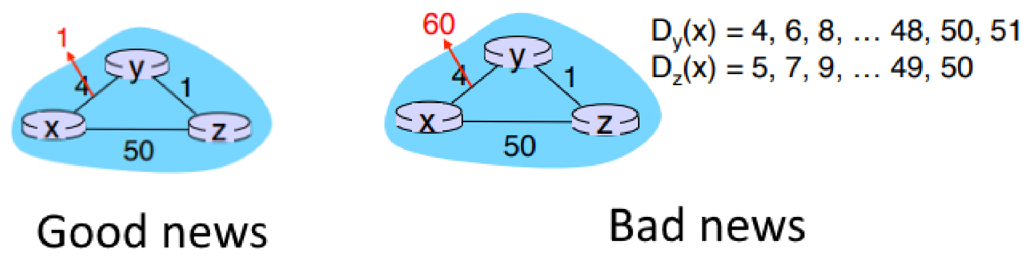
The node that achieves this min is the next hop in the path. In this protocol, each node maintains a distance vector notated D_x. A node x knows the cost to each neighbor $c(x,v)$ and it also maintains its neighbors' distance vector: x maintains $D_v = [D_v(y) : y \in N]$. The key idea is to send our own distance vector estimate to our neighbors from time to time and when a node receives a new DV from a neighbour, our own DV are updated using the Bellman ford equation:

$$D_x(y) \leftarrow \min_v \{c(x,v) + D_v(y)\}$$

After some time, the estimated distance vector will converge towards a SS which is the actual least cost path, as long as all the costs are positive. It is iterative and asynchronous: each node notifies neighbors only when its DV change and then, if needed, the neighbors will also notify their neighbors and so on. For robustness, even after convergence we sometimes re-send our data to tell that we are still present.

Let's now study the effect of change in the costs.

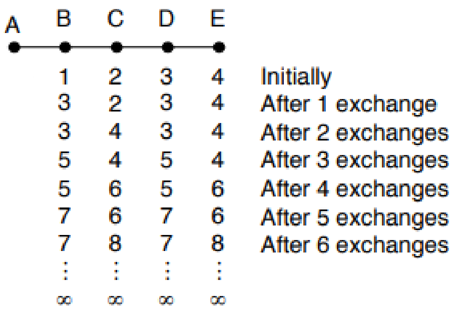
1. Good news travels fast: a node detects a local link cost change, it updates its routing information and recalculates its DV. If it changes, it will notify its neighbors. The following Figure illustrates the principle: at time t_0, y detects a change and $D_y(x) \leftarrow 1$, info that is sent to its neighbors. At time t_1, z receives the updates from y and computes a new cost $D_z(x) \leftarrow 2$ which is sent to the neighbors of z. At time t_2, y receives the updates and itself updates its table, but nothing changes and the convergence is obtained.



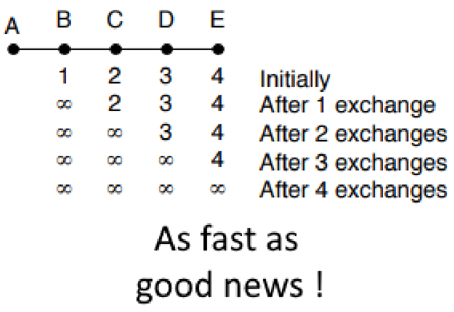
2. Bad news travels slowly: consider first a case where one link goes down. For example, imagine 5 hops organised as A-B-C-D-E in a line. If link A-B is down, $c(B,A) = +\infty$. For B to reach A, it must now compare $+\infty$ to $2+1 = 3$ in the Bellman Ford update rule and it will chose 3. This update is sent to C, which will also recompute. To go to A, it can either go via B with a cost $1 + 3$ or it can go via D with a cost $1+3$ as well so it is updated to 4 and that is sent again to the neighbors. Then, B will have to choose between $+\infty$ and $1 + 4 = 5$ and so on until all nodes reach $+\infty$. this is called a count to infinity problem, it can also happen

in a cost simply increases as shown in the illustration as well.

A way to solve this issue is called **poisoned reverse**, it basically consist in saying that "if C routes through B to get to A, then C lies to B: C tells B its distance to A is infinite". This way, it is impossible to chose a path that would make the packet bounce back to its sender and the info propagates much faster, as illustrated below as well. This time, C tells B its distance towards A is infinite, so it can chose between $+\infty$ and $+\infty$ so it takes $+\infty$. Then, C can choose between $+\infty$ and $+\infty$ as well and so on.

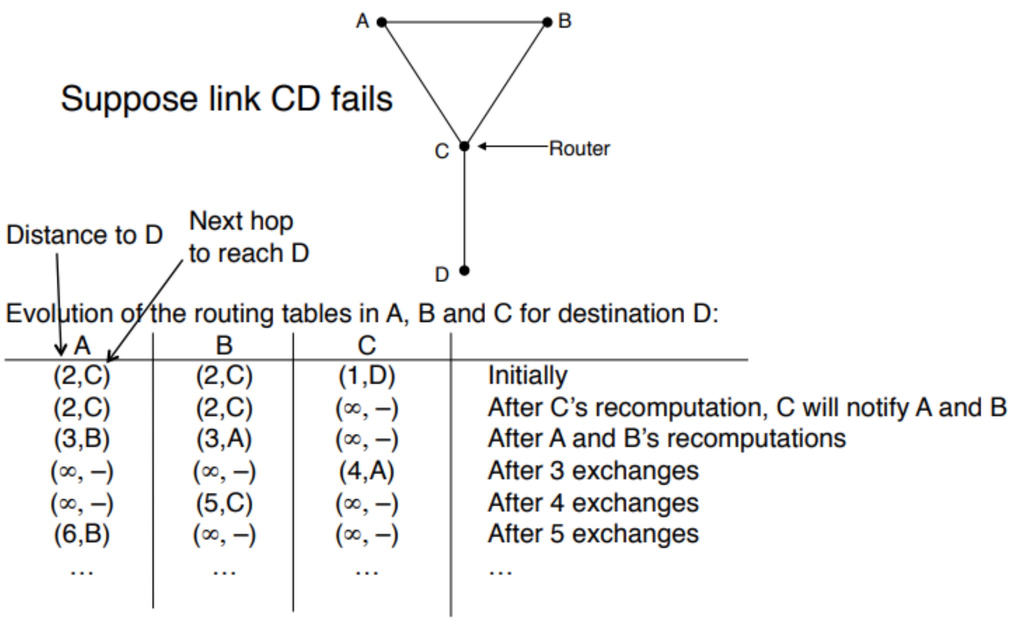


B	C	D	E	
1	2	3	4	Initially
3	2	3	4	After 1 exchange
3	4	3	4	After 2 exchanges
5	4	5	4	After 3 exchanges
5	6	5	6	After 4 exchanges
7	6	7	6	After 5 exchanges
7	8	7	8	After 6 exchanges
⋮	⋮	⋮	⋮	
∞	∞	∞	∞	



B	C	D	E	
1	2	3	4	Initially
∞	2	3	4	After 1 exchange
∞	∞	3	4	After 2 exchanges
∞	∞	∞	4	After 3 exchanges
∞	∞	∞	∞	After 4 exchanges

Note that it is not an ultimate solution, there still remain configurations that can fail as shown at the end. In this configuration, there is intermediate node in between which will prevent node C from lying even if it should actually do it. As the cost in C increases, A will route via B and B via A but the path still contains C. when A and B update their paths, A will tell $+\infty$ to B and vice versa, leading to their costs being infinite in the next iteration, but now, C can find a path via A or B and will remove its infinite value and so on.



Evolution of the routing tables in A, B and C for destination D:

A	B	C	
(2,C)	(2,C)	(1,D)	Initially
(2,C)	(2,C)	(∞, –)	After C's recomputation, C will notify A and B
(3,B)	(3,A)	(∞, –)	After A and B's recomputations
(∞, –)	(∞, –)	(4,A)	After 3 exchanges
(∞, –)	(5,C)	(∞, –)	After 4 exchanges
(6,B)	(∞, –)	(∞, –)	After 5 exchanges
…	…	…	…

Routing Information Protocol (RIP)

It is a distance vector algorithm where each link as a cost of one. As we are now in a real network, we don't want to reach a node but a subnet. It consists of a table with three entries: the destination subnet, the next router and the number of hops to reach the destination. When next router is empty, it means that the current router is attached to the corresponding subnet (cost of 1 hop).
The poisoned reverse is not implemented exactly as before, here the idea is to say that when a distance vector is received from a neighbour, if the next hop is ourself, we should remove that entry as it can cause a count to infinity problem.

Comparison of Link state versus Distance vectors

- Message complexity: LS with n nodes and E links: $\mathcal{O}(nE)$ messages are sent for the flooding. DV only exchanges with neighbours, but it is a bit more complex to define as it depends on the number of iterations.
- Speed of convergence: LS uses Djikstra's algorithm: $\mathcal{O}(n \log n)$. With DV it can be much slower as there is a risk of count to infinity.

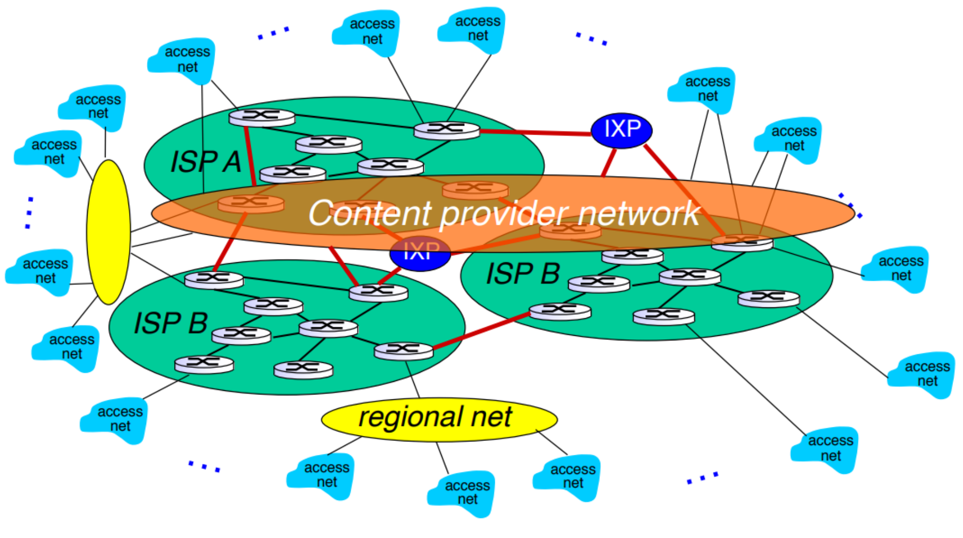
12 Internet structure, interdomain routing, and BGP

The routing study with distance vectors and link state protocols is an idealization but in practice, it is a gigantic network, it is actually even worse: it is a network of networks. The end systems connect to Internet via acces **I**nternet **S**ervice **P**roviders. Those ISPs must also be inter-connected so that any two hosts can send pkts to each other.

First question: *"How to connect these millions ISPs together ?"* connecting them all to all the other ones would result in $\mathcal{O}(n^2)$ connections which is not scalable. Instead, connect them all to a **global transit ISP**. Of course, if it is economically interesting to have such a global ISP, there will be competitors and in term, several global ISPs. These different global ISPs are themselves inter-connected to be present in any big city via **Internet exchange points** (like a big switch on which several ISPs connect). There can also be **peering links** (less often) that just connect them without need to pay, just to exchange traffic (no notion of provider, they are equal and just exchange traffic). What is so-called a "Tier-1 ISP" is a player big enough to be connected to all the others (it is nothing official, just a way to see it).

Then, **regional networks** or regional ISP may arise to connect access nets to those global ISPs. A bit later in time, **content providers** appeared. This new type of players may run their own network to prevent services or content to all end users (such as *Google* for example). The often bypass the tier-1 ISP and users can directly connect to them because they own their own network linking their data centers. they try to have presence in as much location as possible (all big ISPs, some regional ISPs, etc..) so that when you make a *Youtube* request, you are quickly served. This leads to the final complex architecture shown in the following figure.

In a world-map of the internet, one can see point where ISPs are present, those are actually Points Of Presence (POP) which provide connections to peers, connections to other POPs or other networks as well as several connection to regional ISPs, providing a lot of redundancy to avoid single points of failure.



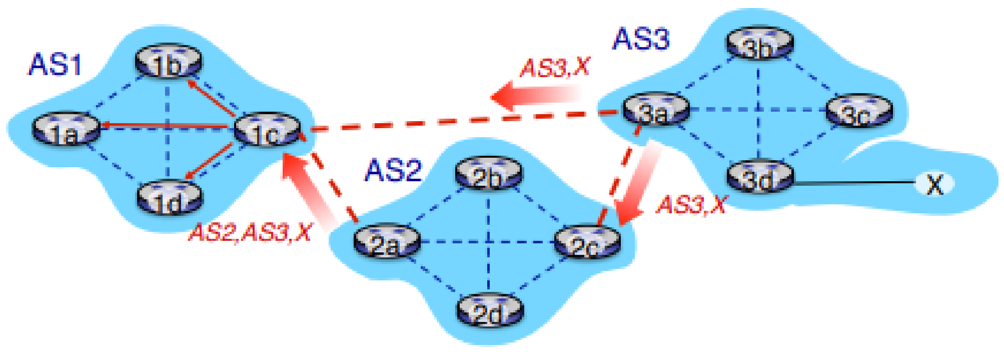
Even though it is so complex, one still needs a scalable routing method, and that can be done by aggregating routers into regions as **Autonomous Systems** (ASes), also called domains. This separates the routing task into two categories: the intra-AS routing which is the routing among hosts and the inter-AS routing which is the routing between the different ASes. This is done by the gateway routers which are at the edge of their own ASes and that have links to other ASes.

The inter AS-task consists in learning which destinations are reachable though which AS and then to propagate that info to the other routers of the AS. **B**order **G**ateway **P**rotocol (BGP) is the "de facto" interdomain routing protocol. It is divided in eBGP (external) to obtain subnet reachability information from neighboring ASes and also iBGP (internal) to propagate reachability information to all AS-internal routers. The gateway routers run both eBGP and iBGP.

The idea is that two BGP routers exchange BGP messages over a semi permanent *TCP* connection (semi because it can fail etc..). In these messages, they advertise paths to different destination network prefixes, a prefix is a *IP* address prefix, thus a range of *IP* addresses. An advertised prefix includes BGP attributes, there are two important ones: AS-PATH (list of ASes through which the advertisement has passsed) and NEXT-HOP (*IP* address of the internal-AS gateway router leading to the first AS in AS-PATH).

The routing can be policy based. That means that the gateway receiving route advertisements uses import policy to accept/decline a path. The policies also determine to which neighboring ASes we send the advertisements.

The following figures shows an illustration of path advertisements propagation between ASes. A gateway router may learn about multiple paths, based on policy it will chose a given path.



Over the *TCP* connection, several messages can be exchanged: OPEN (opens the connection between the *BGP* peers), UPDATE (advertise a new path), KEEPALIVE (keeps connection alive in absence of updates, this also ACKs the OPEN request) and NOTIFICATION (report errors in previous msg, or to close the connection).

The selection principle when several paths are available is done according to the following order of importance:

1. Local preference: policy decision. The idea is that going via a customer is good (money in), going through a peer is fine (no money in nor out) but going via a

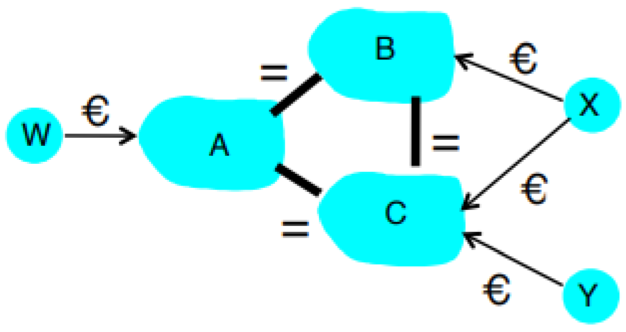
provider is bad (money out).

2. Shortest AS-PATH
3. Closest gateway router (NEXT-HOP)
4. Additional criteria

It can be summarized as economical → inter-domain performances → intra-domain performances. A internal router can learn several paths to reach different gateways, in such a case **hot potato routing** can be performed, that is chosen the local gateway router with least intra-domain cost (without regarding the inter-domain cost).

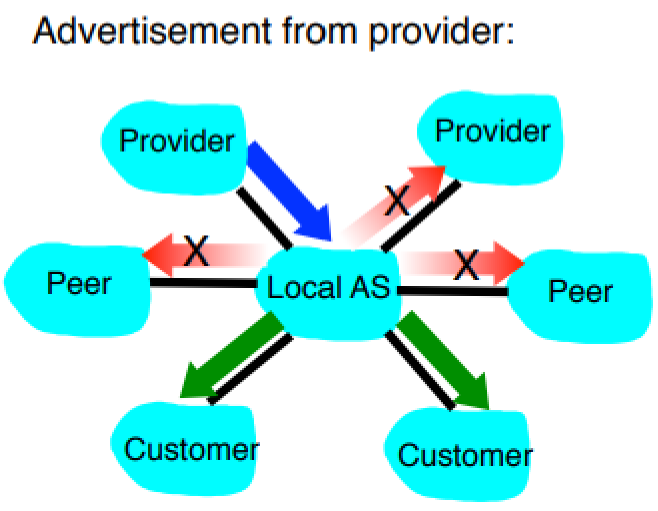
Let's now see how to avoid routing loops. There is a mechanism similar to the poisoned reverse of RIP: if an AS sees itself in the AS-PATH advertised by a neighbor AS, it discards it because it means that a loop would be created. It is actually stronger than the Poisoned reverse mechanism because the routers have access to the entire path and not only the next hop.

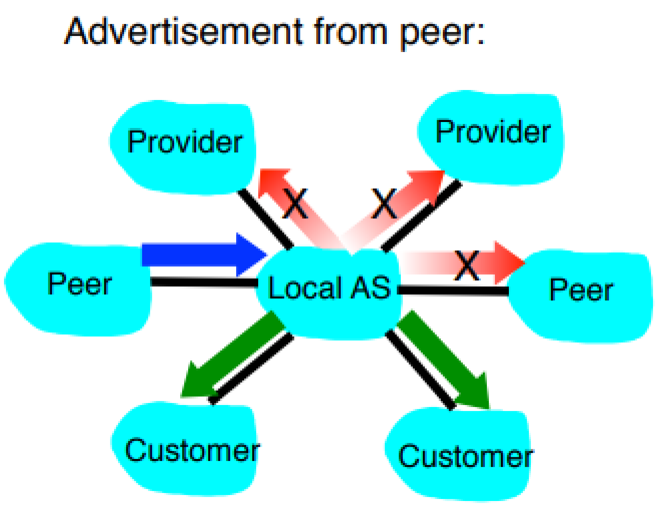
Let's finally dig a bit in the policies. The following image is a network example in which A,B and C are providers and X,Y and W and clients. X is "dual-homed" meaning

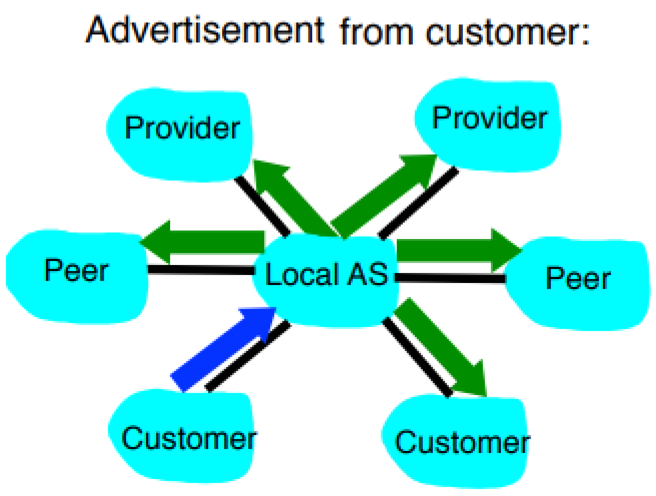


that it is attached to two networks to provide it a better connection, but to go from B to C, X does not want a path via itself. It will thus not advertise to B a route through C. Also, A advertises AW to B and to C and B advertises BAW to X. But B will choose not to advertise BAW to C because it will not gain anything, because in CBAW there is no client for B.

These examples can be generalized in the **Valley-Free** (VF) routing. A promise is propagated to customers (as they pay us) but not to peers (no money is won) and not to providers (cost money to use that path). A customer will need to pay anyway so it can propagate to everyone.


Advertisement from provider:
Provider
Provider
X
X
X
Peer
Local AS
Peer
Customer
Customer


Advertisement from peer:
Provider
Provider
X
X
X
Peer
Local AS
Peer
Customer
Customer


Advertisement from customer:
Provider
Provider
Peer
Local AS
Peer
Customer
Customer

13 Multiple access protocols in LANs: Aloha, CSMA, CSMA/CD, Ethernet

A multiple access protocol is a distributed algorithm that determines how nodes share a channel. There is no out-of-band channel coordination, which means that the control messages themselves can collide. They are so called MAC (medium access control) protocols. The ideal protocol consists in:

1. If one node wants to send, it can do at full rate R
2. When M nodes want to transmit, each of them gets a rate R/M (fairness)
3. Fully decentralized: no special node for coordination and no synchronization (no slots/ clocks)
4. Simple protocol

These protocols can be divided into categories: Channel partitioning (slots, frequency,..), random access (which allows collisions) or taking turn (take the best out of both previous ones).

Aloha

It is a random access MAC protocol. It can be of two types, slotted or unslotted. Let's start by analyzing the slotted Aloha. We first add a few assumptions to ease the analysis: all frames have the same size, the time is divided into equal slots, nodes transmit at the beginning of the slot, nodes are synchronized and if 2 or more transit, they all detect the collision. The operation principle is the following: when a node obtains a fresh frame, it can transmit in the next slot. If there is no collision detected, the node can send a new frame in the next slot. If there is a collision, the node re-transmits the frame in each subsequent slot with probability p until success.
The idea is to add some randomness into the system to avoid having all nodes re-sending at the same time and having the same problem again.

Let's discuss the pros and cons:

- PROS:
  - Single active can send at rate R
  - Highly decentralized (only slots need to be sync)
  - Simple
- CONS:
  - Collisions $\rightarrow$ waste of slots
  - Idle slots
  - Nodes may be able to detect collision in less than time to transmit (detected by a loss and timeout, but could be faster)
  - Requires a clock synchronization

What is the efficiency of this protocol? We will calculate it only is a situation in which it should be evaluated (when problems occur) otherwise all protocols would be good. The efficiency is here defined as the long-run fraction of successful slots when there are many nodes, each with many frames to send.

Suppose N nodes with many frames and they can send with probability p (not really Aloha, but simpler for calculation). It follows a binomial probabilistic distribution:

$$P(success, 1) = p(1-p)^{N-1} \text{ AND } P(success, all) = N\,p(success, 1) = Np(1-p)^{N-1}$$

Then, one can find the best probability by canceling the derivative of $p(success, all)$ with respect to p, this gives $p^* = 1/N$. Let's now take the limit as the number of nodes tends to infinity of the proba with that optimal p^*:

$$\lim_{N\to+\infty} Np^*(1-p^*)^{N-1} = \lim_{N\to+\infty} \left(1 - \frac{1}{N}\right)^{N-1} = e^{-1} = 37\%$$

This result was obtained using the formula $\lim_{N\to+\infty} \left(1 - \frac{G}{N}\right)^N = e^{-G}$.

Let's now see what pure (unslotted) Aloha looks like. Here the idea is that are no slots and no time synchronisation, meaning that the probability of collision is increased. Let's see the efficiency of this version:

$$\begin{aligned} P(success, 1) &= P(node\ transmits) \\ &\quad \times P(no\ other\ transmits\ in\ [t_0 - 1, t_0]) \\ &\quad \times P(no\ other\ transmitsin\ [t_0, t_0 + 1]) \\ \Leftrightarrow P(success, 1) &= p(1-p)^{N-1}(1-p)^{N-1} = p(1-p)^{2(N-1)} \\ \Leftrightarrow P(success, all) &= Np(1-p)^{2(N-1)} \end{aligned}$$

which, this times, gives en efficiency of 18%, which is worse than before.

Let's modify the way we define the efficiency and let's do it with respect to the **average aggregated traffic load per frame time**, $G = pN$. This basically represents the number of transmission attempts in a time frame, because N stations are sending with probability p at each frame. Let's see how this changed the efficiencies:

- Slotted: $Np(1-p)^{N-1} = G(1-G/N)^{N-1}$ which gives Ge^{-G} when N tends to infinity.
- unslotted: $Np(1-p)^{2(N-1)} = G(1-G/N)^{2(N-1)}$ which gives Ge^{-2G} when N tends to infinity.

Notice that when $G << 1$, both are almost linear, which is perfect because it means that there is no collision.

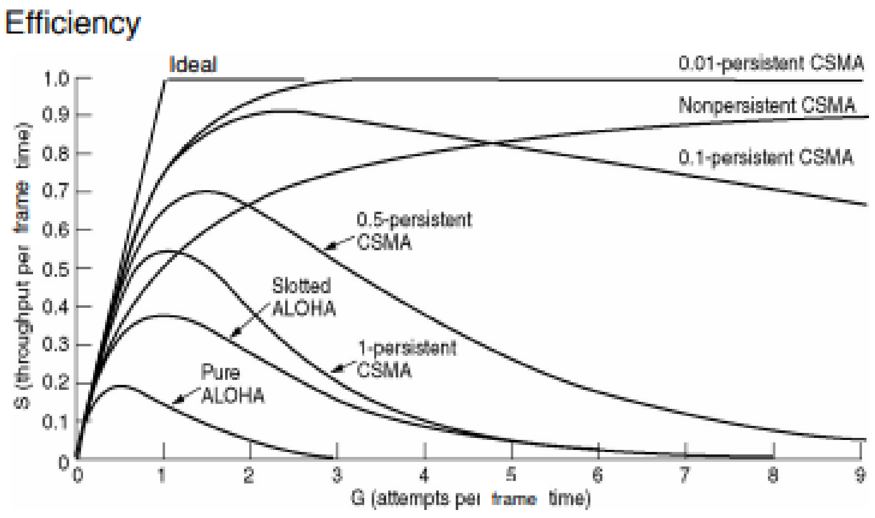
Carrier Sense Multiple Access (CSMA)

It is an improved version of pure Aloha with carrier sensing. The idea is to listen before transmitting and to send only if nothing is sensed. Note that collision can still occur when doing this because when the bit propagate, there is a propagation delay. This means that if a node A sends, node B might sense right after and not see anything and also decide to transmit, but this will lead to a collision later on.
Having this in mind, we can move to p-persistent CSMA, which consists in:

- While(1) do
- if(channel is free) do
  - with proba p: send immediately
  - with proba $(1-p)$: stay idle during at least propagation time τ
- else do listen until channel is freed

This is a mix between persistent (busy: listen until freed) and non-persistent (if busy, send later).

Let's compare it to Aloha:



Let B the channel data rate, F the frame size, L the channel length and c the propagation speed ($2/3c_{light}$ usually). Then $\tau = L/c$ is the propagation delay and $T = F/B$ is the transmission delay. One can defined $a = \tau/T = (BL)/(cF)$. The smaller a, the lower the risks. Then, one can dimension such a network by finding the frame size F for a given a.

CSMA/CD: Collision Detection

With this version, collision are detected and the transmission is aborted: it reduces the channel wastage. Collision detection is easy in wired LANs but is difficult for wireless LANs as the signal strongly attenuates over distance. With pure $CSMA$, there is no aboard so the loss time is the entire transmission delay T. With this update, the lost time is the time needed for both sender to notice the interference, that is 2τ in the worst case (as τ is the time for all of them to detect it, which is equivalent to a scenario where the two extremity nodes send at the same time).

To make sure the collision is detected, the sender must still be sending when the collision occurs and propagates back to it, that is $T > 2\tau$, or equivalently $F/B > 2L/c$, which leads to $F_{min} \simeq BL \cdot 10^{-8}$ bits.

Ethernet

Ethernet is based on *CSMA/CD*. It consists in the following:

1. NIC receives datagram from network layer: creates datagram.
2. if NIC senses that the channel is idle, it starts the transmission (1-persistent). If it is busy, it waits until it is idle and then transmits.
3. if NIC transmits the entire frame without error, NIC is done with that frame.
4. if NIC detects another transmission, it sends jam (to reinforce the collision to make sure everyone senses it).
5. After aborting, NIC enters in **binary exponential backoff**. After the m^{th} collision ($m < 10$) for a given frame, NIC chooses K at random from $\{0, 1, 2, \ldots, 2^m - 1\}$ and then waits $K \cdot 512$ bit times and returns to step 2.
   So, the bigger the number of collisions, the longer the backoff interval. This also means that after a collision, it moves from 1-persistent to p-persistent with a proba p that evolves.

In other words, it can be called a p-persistent adaptive *CSMA/CD*. Ethernet works with a frame size of 64 bytes.

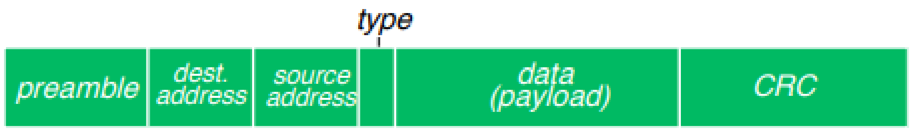
One can see that the efficiency increases if the frame size increases, indeed when dividing the size by 2, we multiply by 2 the risk of collision. Let α the number of slots 2τ before any successful transmission, then:

$$\eta = \frac{T}{T + \alpha 2\tau} = \frac{1}{1 + \alpha\frac{2\tau}{T}} = \frac{1}{1 + 2\,\alpha\frac{BL}{cF}}$$

For large values of N, α converges towards e, meaning that the efficiency can't be better than:

$$\frac{1}{1 + e\frac{2\tau}{T}} = \frac{1}{1 + 5.4\frac{\tau}{T}}$$

More about it: *Ethernet* was the first widely used LAN technology, it is simple and can work at different speeds from 10 Mbps to 10 Gbps. The *IP* datagram is encapsulated in an *Ethernet frame* which is constructed as in the image below. The preamble is a 7 bytes pattern with 1010101...1011 (finishes with 2 ones). this way, even if the receiver does not receive everything, not a problem as the pattern is recognized. Also, by reading this pattern, it can calculate the clock frequency of the sender and lock to that frequency. The addresses are 6 bytes source and destination *MAC* addresses and the Type field indicates higher level protocols (*IPv4*, *IPv6*, *ARP*,...). The CRC field is used for error detection.



Ethernet is connectionless (no handshaking) and unreliable (receiving NIC doesn't sent ACKs or NACKs to the sending NIC). There exist many different *Ethernet* standards, but they share a common MAC protocol and frame format, but vary in the speed and the physical media used to perform the transmission.

14 Link-Layer: addressing, ARP, switching, self-learning and spanning tree, differences between hubs, switches and routers

Addressing and ARP

A *MAC* (or LAN or physical or *Ethernet*) address is used to locally get a frame from one interface to another one that is physically connected. For most LANs, it is a 48-bit address. Unlike *IP*, it is not supposed to change and is set in the ROM of the device. To perform an analogy with human being, the *IP* address is the postal address while the MAC address is more like a social security number that never changed. It is said to be **portable** as it remains unchanged even if we go from one LAN to another. It is standardized by *IEEE*: the prefix corresponds to the manufacturer and the suffix is the address itself.

ARP stands for **A**ddress **R**esolution **P**rotocol. It aims to answer the question "How to determine the *MAC* address of an interface knowing its *IP* address ?". It is done with an *ARP* table that maps *IP* to *MAC* addresses. It is often refreshed and there is a TTL after which the entries are discarded.
It works as follows: imagine that A wants to send a datagram to B but B's *MAC* address is not in A's *ARP* table. A broadcast query packet is sent containing B's *IP* address, this is done by using the FF-FF-FF-FF-FF-FF *MAC* address. As it is a broadcast, it will be received by B which will answer to A with its *MAC* address. With this broadcast, all other stations have also cached A's *IP*-to-*MAC* pair. [nice example at slide 6-57 to 6-62]

Switches: self learning and spanning tree

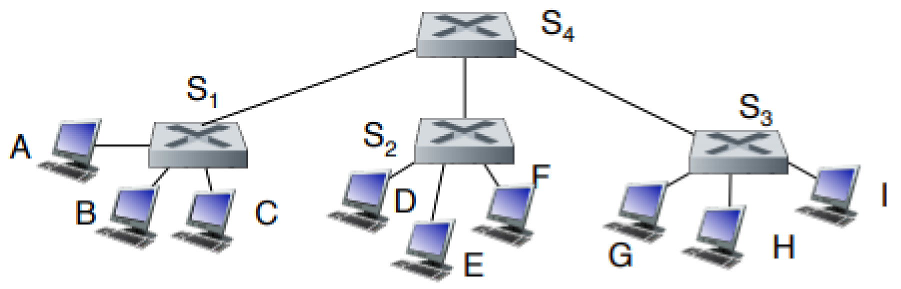
Switches take an active role in the transmission of frames. They store and forward *Ethernet* frames. They examine the incoming frame's *MAC* address and selectively forward that frame to one or more outgoing links. It uses *CSMA/CD* to access segment.
Switching is **transparent** meaning that the hosts are unaware of their presence and it is **plug-and-play / self learning** meaning that switches do not need to be configured.

Switch **learns** which host/routers can be reached through which interface. When it receives a frame, it learns the location of the sender. It happens as follows:

1. record incoming link and the *MAC* address of sending host/router
2. index switch table using the *MAC* destination address
3. if (entry found for destination) do:
   - if (dest==source) drop the frame
   - else forward frame on interface indicated by entry
4. else flood (forward on all interfaces but that on which it arrives)

Now, it can happen that several switches are themselves interconnected. then, self learning also applies and the frame is flooded from switch to switch and then further into each subnetwork within the LAN.

For example, consider the following LAN: If A wants to reach F, it will go to S1 which



will flood it. B and C will discard the frame and S4 will also flood it. D,E will discard it and F will keep it. G, H and I also discard the frame. In this process, S1, S2, S3 and S4 learned about A so if F replies, the path to go to A will be known. As far as F is concerned, S2, S4 and S1 will learn it with the reply but not S3 because nothing is flooded this time.

If there are cycles in the topology, the flooding can be endless and this needs to be solved. The solution is to build a logical spanning tree over the real topology and to flood only on that tree. This can be done with a three steps process:

1. Determine the root switch: the switches regularly send **B**ridge **P**rotocol **D**ata **U**nits (BPDUs) containing their root as assumed, the distance to the root and the source switch id. Each switch has a unique name and there is a lexicographical way to determine which is the root. So at first, they all think they are the root and send (i,0,i) where i is the id of the switch itself. Then, they receive the BPDUs from their neighbors and adapt their knowledge about the root.

2. Build the tree: by continuously receiving BPDUs, each switch can determine its distance to the root and which port leads to the root with shortest distance. This port is called r as "root port". It is similar to a distance vector routing protocol but with a single destination which is the root switch. It case of ties, it is done as follows: (rootX, distX, sourceX) is better than (rootY, distY, sourceY) if:

   - rootX < rootY
   - rootX = rootY AND distX<distY
   - rootX = rootY AND distX=distY AND sourceX<sourceY

3. Determine whether non-root ports should be forwarding or blocking: a port is forwarding f on a given segment if and only if the BPDUs that this switch sends on this segment are smaller than those other switches would send, otherwise it is blocking. The root switch as only forwarding ports, all other switches have only one root port.

In the final tree, some switches may not be part of it, if they only have blocking (+1 root) ports. If a switch fails, the tree is updated and some blocking ports will become forwarding, this means that switches must keep listening to BPDUs all the time to detect failures.
It is different than router's spanning tree because here there is only one single tree for the entire architecture while in the network layer, there is one tree per node and the tree is used for all the forwarding while here, it is only to perform the flooding.

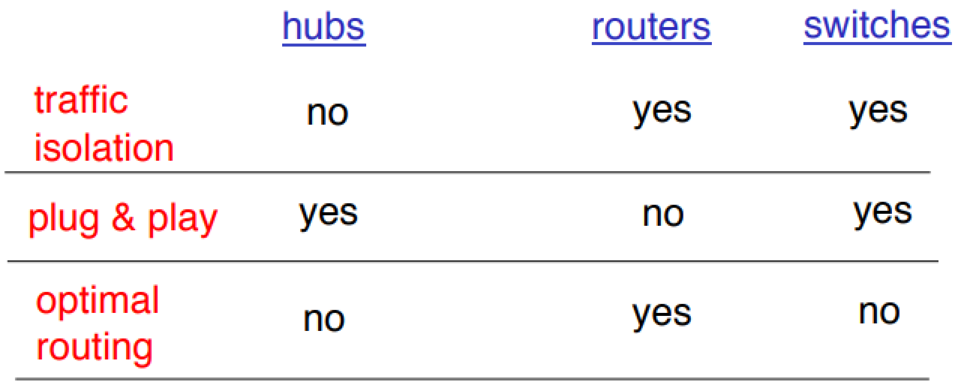
Routers VS Switches VS Hubs

Routers and Switches are both store-and-forward systems but routers operate in the network layer while switches operates in the link layer. They both contain forwarding tables, one with *IP* addresses and the other with *MAC* addresses. In routers, it is obtained with routing algorithms while a switch is self learning based on the messages it receives.

A hub is also something operating in the physical layer, but they are just "dumb" repeaters. The incoming bits on a port are repeated on all the other links at the same rate. All nodes that are connected to a hub can collide with one another, there is no buffering at all. There is also no collision detection mechanism, it is the hosts NIC that are in charge of detecting collisions.

There are several topologies for both switches and hubs, they can be in a bus configuration (but more chances of collision) or in a star configuration which is more practical and prevails nowadays. Indeed, with a bus, if there is a defect at one place, it is unusable anymore, so it is harder to manage.

The following image compares them all for different criterion:



	hubs	routers	switches
traffic isolation	no	yes	yes
plug & play	yes	no	yes
optimal routing	no	yes	no