CT420

Real-Time Systems

Name: Andrew Hayes
Student ID: 21321503
E-mail: a.hayes18@universityofgalway.ie

2025–03–20

Contents

1 Introduction **1**
1.1 Lecturer Contact Information . . . 1
1.2 Assessment . . . 1
1.3 Introduction to Real-Time Systems . . . 1

2 The Essence of Time: From Measurement to Navigation & Beyond **1**

3 Time Synchronisation in Distributed Systems **2**
3.1 Synchronisation in the Real World . . . 2
3.1.1 Cristian's Algorithm . . . 3
3.1.2 Berkeley Algorithm . . . 3
3.2 Logical Clocks . . . 3
3.2.1 Vector Clocks . . . 4

4 The NTP Protocol **5**
4.1 NTP . . . 7
4.1.1 NTP Protocol Header . . . 9
4.1.2 Mitigation Algorithms . . . 12

5 Soft RTS **13**
5.1 Quality of Service (QoS) . . . 13
5.1.1 QoS Metrics . . . 13
5.2 Real-Time Multimedia Technologies . . . 14
5.2.1 Transport Layer for Real-Time Media . . . 14
5.2.2 Service Requirements for Real-Time Flows (Voice / Video) . . . 14
5.2.3 RTP . . . 14
5.2.4 Mouth-to-Ear M2E Delay . . . 16
5.3 Packet Loss Strategies . . . 17
5.3.1 Network-Based Strategies . . . 17
5.4 Cloud Gaming . . . 18
5.4.1 Response Delay . . . 18
5.5 WebRTC . . . 19
5.5.1 WebRTC Working Model . . . 19
5.6 DASH . . . 20

6 Emerging Protocols **22**
6.1 How to achieve speed? . . . 22
6.2 Evolution of the Web . . . 22
6.3 HTTP . . . 22
6.3.1 HTTP/0.9 . . . 22
6.3.2 HTTP/1.0 . . . 23
6.3.3 HTTP/1.1 . . . 23
6.3.4 SPDY . . . 24
6.3.5 HTTP/2.0 . . . 25
6.4 Transport Protocols . . . 27
6.4.1 Issues with TCP . . . 28
6.5 QUIC . . . 28

7 Congestion Control in QUIC **31**
7.0.1 Deciding Sending Rate 32
7.0.2 Estimating Link Capacity 32
7.1 Congestion Control 32
7.1.1 Reno 33
7.1.2 Cubic 33

1 Introduction

1.1 Lecturer Contact Information

- Name: Dr. Michael Schukat.
- E-mail: michael.schukat@universityofgalway.ie.
- Office: CSB-3002.
- Name: Dr. Jawad Manzoor.
- E-mail: jawad.manzoor@universityofgalway.ie.
- Office: CSB-3012.

1.2 Assessment

- 2 hours of face-to-face & virtual labs per week from Week 03.
- 30% Continuous Assessment:
  - 2 assignments, 10% each.
  - 2 in-class quizzes between Week 07 & Week 12, worth 5%.

1.3 Introduction to Real-Time Systems

A system is said to be **real-time** if the total correctness of an operation depends not only upon its logical correctness but also upon the time in which it is performed. Contrast functional requirements (logical correctness) versus non-functional requirements (time constraints). There are two main categorisation factors:

- **Criticality:**
  - **Hard RTS:** deadlines (responsiveness) is critical. Failure to meet these have severe to catastrophic consequences (e.g., injury, damage, death).
  - **Soft RTS:** deadlines are less critical, in many cases significant tolerance can be permitted.
- **Speed**
  - **Fast RTS:** responses in microseconds to hundreds of microseconds.
  - **Slow RTS:** responses in the range of seconds to days.

A **safety-critical system (SCS)** or life-critical system is a system whose failure or malfunction may result in death or serious injury to people, loss of equipment / property or severe damage, & environmental harm.

2 The Essence of Time: From Measurement to Navigation & Beyond

Time is the continued sequence of existence & events that occurs in an apparently irreversible succession from past, through the present, into the future. Methods of temporal measurement, or chronometry, take two distinct forms:

- The **calendar**, a mathematical tool for organising intervals of term;
- The **clock**, a physical mechanism that counts the passage of time.

Global (maritime) exploration requires exact maritime navigation, i.e., longitude & latitude calculation. **Latitude** (north-south) orientation is straightforward; **longitude** (east-west orientation) requires a robust (maritime) clock.

Ground-based navigation systems like LORAN (LOng RAnge Navigation) were developed in the 1940s and were in use until recently, and required fixed terrestrial longwave radio transmitters, and receivers on-board of ships & planes. They are also referred to as hyperbolic navigation or multilateration. The principles of ground-based navigation systems is as follows:

1. A **master** with a known location broadcasts a radio pulse.
2. Multiple **slave** stations with a known distance from the master send their own pulse, upon receiving the master pulse.
3. A **receiver** receives master & slave pulses and measures the delay between them.
4. This allows the receiver to deduce the distance to each of the stations, providing a fix.

NEED TO FINISH

3 Time Synchronisation in Distributed Systems

A **distributed system (DS)** is a type of networked system wherein multiple computers (nodes) work together to perform a task. Such systems may or may not be connected to the Internet. Time & synchronisation are important issues here: think of error logs in distributed systems – how can error events recorded in different computers be correlated with each other if there is no common time base? The problem is that GNSS-based time synchronisation may or may not be available, as GPS signals are absorbed or weakened by building structures. There is no other time reference such systems can rely on because in such a distributed system there are just a series of imperfect computer clocks.

In distributed systems, all the different nodes are supposed to have the same notion of time, but quartz oscillators oscillate at slightly different frequencies. Hence, clocks tick at different rates (called *clock skew*), resulting in an increasing gap in perceived time. The difference between two clocks at a given pot is called *clock offset*. The **clock synchronisation problem** aims to minimise the clock skew and subsequently the offset between two or more clocks. A clock can show a positive or negative offset with regard to a reference clock (e.g., UTC), and will need to be resynchronised periodically. One cannot just set the clock to the "correct" time: jumps, particularly backwards, can confuse software and operating systems. Instead, we aim for gradual compensation by correcting the skew: if a clock runs too fast, make it run slower until correct and if a clock runs too slow, make it run faster until correct.

Synchronisation can take place in different forms:

- Based on **physical** clocks: absolute to each other by synchronising to an accurate time source (e.g., UTC), absolute to each other by synchronising to locally agreed time (i.e., no link to a global time reference), where the term *absolute* means that the differences in timestamps are proper time intervals.
- Based on **logical** clocks (i.e., clocks are more like counters): timestamps may be ordered but with no notion of measurable time intervals.

In either case, the DS endpoints synchronise using a shared network. For physical clock synchronisation, network latencies must be considered as packets traverse from a sending node to a receiving node. In a **perfect network**, messages *always* arrive, with a propagation delay of *exactly* d; the sender sends time T in a message, the receiver sets its clock to $T + d$, and synchronisation is exact.

In a **deterministic network**, messages arrive with a propagation delay $0 < d \leq D$; the sender sends time T in a message, the receiver sets its clock to $T + \frac{D}{2}$, and therefore the synchronisation error is at most $\frac{D}{2}$. **Deterministic communication** is the ability of a network to guarantee that a message will be transmitted in a specified, predictable period of time.

3.1 Synchronisation in the Real World

Most off-the-shelf networks are *asynchronous*, that is, data is transmitted intermittently on a best-effort basis. They are designed for flexibility, not determinism, and as a result, propagation delays are arbitrary and sometimes even unsymmetric (i.e., upstream & downstream latencies are different). Therefore, synchronisation algorithms are needed to accommodate these limitations.

3.1.1 Cristian's Algorithm

Cristian's algorithm attempts to compensate for symmetric network delays:

1. The client remembers the local time T_0 just before sending a request.
2. The server receives the request, determines T_S, and sends it as a reply.
3. When the client receives the reply, it notes the local arrival time T_1.
4. The correct time is then approximately $(T_S + \frac{(T_1 - T_0)}{2})$.

The algorithm assumes symmetric network latency. If the server is synced to UTC< all clients will follow UTC. Limitations of Cristian's algorithm include:

- Assumes a symmetric network latency;
- Assumes that timestamps can be taken as the packet hits the wire / arrives at the client;
- Assumes that T_S is right in the middle of the server process; for example, consider the server process being pre-empted just before it sends the response back to the client, which will corrupt the synchronisation of the client.

3.1.2 Berkeley Algorithm

In the **Berkeley algorithm**, there is no accurate time server: instead, a set of client clocks is synchronised to their average time. The assumption is that offsets / skews of all clocks follow some symmetric distribution (e.g., a normal distribution) with some clocks going faster and others slower, and therefore a mean value close to 0.

1. One node is designated to be the **master node** M.
2. The master node periodically queries all other clients for their local time.
3. Each client returns a timestamp or their clock offset to the master.
4. Cristian's algorithm is used to determine and compensate for RTTs, which can be different for each client.
5. Using these, the master computes the average time (thereby ignoring outliers), calculates the difference to all timestamps it has received, and sends an adjustment to each client. Again, each computer gradually adjusts its local clock.

Client clocks are adjusted to run faster or slower, to be synced to an overall agreed system time. The client networks is an intranet, i.e., an isolated system. Therefore, the Berkeley algorithm is an **internal clock synchronisation algorithm**. The Berkeley algorithm was implemented in the TEMPO time synchronisation protocol, which was part of the Berkelely UNIX 4.3BSD system.

3.2 Logical Clocks

Logical clocks are another concept linked to internal clock synchronisation. Logical clocks only care about their internal consistency, but not about absolute (UTC) time; subsequently, they do not need clock synchronisation and take into account the order in which events occur rather than the time at which they occurred. In practice, if clients or processes only care that event a happens before event b, but don't care about the exact time difference, they can make use of a logical clock.

We can define the **happens-before relation** $a \rightarrow b$:

- If events a and b are within the same process, then $a \rightarrow b$ if a occurs with an earlier local timestamp: **process order**.
- If a is the event of a message being sent by one process, and b is the event of the message being received by another process, then $a \rightarrow b$: **causal order**.

- We also have **transitivity:** if $a \to b$ and $b \to c$, then $a \to c$.

Note that this only provides a *partial order*: if two events a and b happen in different processes that do not exchange messages (not even indirectly), then neither $a \to b$ nor $b \to a$ is true. In this situation, we say that a and b are **concurrent** and write $a \sim b$, i.e., nothing can be said about when the events happened or which event happened first.

Happens-before can be implemented using the **Lamport scheme:**

1. Each process P_i has a logical clock L_i, where L_i can be simply an integer variable initialised to 0.
2. L_i is incremented on every local event e; we write $L_i(e)$ or $L(e)$ as the timestamp of e.
3. When P_i sends a message, it increments L_i and copies its content into the packet.
4. When P_i receives a message from P_k, it extracts L_k and sets $L := \max(L_i, L_k)$ and then increments L_i.

This guarantees that if $a \to b$, then $L_i(a) < L_k(b)$, but nothing else.

The primary limitation of Lamport clocks is that they do not capture **causality**. Lamport's logical clocks lead to a situation where all events in a distributed system are ordered, so that if an event a (linked to P_i) "happened before" event b (linked to P_k), i.e., $a \to b$, then $a$will also be positioned in that ordering before b such that $L_i(a) < L_k(b)$ or simply $L(a) < L(b)$; however, nothing can be said about the relationship between two events a & b by merely comparing their time values $L_i(a)$ and $L_k(b.)$: we can't tell if $a \to b$ or $b \to a$ or $a \sim b$ unless they occur in the same process.

3.2.1 Vector Clocks

In practice, causality is captured by means of **vector clocks**:

1. There is an ordered list of logical clocks, with one per process. Each process P_i maintains a vector $\vec{V_i}$, initially containing all zeroes. Each index k of a vector clock $\vec{V_i}[k]$ represents the number of events that process P_i knows have occurred in process P_k. $\vec{V_i}[i]$ is the count of events that have occurred locally at process P_i, while $\vec{V_i}[k]$ (for $k \neq i$) is the count of events in process P_k that P_i is aware of.

2. On a local event e, P_i increments its own clock component $\vec{V_i}[i]$. If the event is "message send", a new $\vec{V_i}$ is copied into the packet, so that on message sends the current vector state is included in the message.

3. If P_i receives a message from P_m, then each index $\vec{V_i}[k]$ where $k \neq i$ is set to $\max(\vec{V_m}[k], \vec{V_i}[k])$, and $\vec{V_i}[i]$ is incremented.

Intuitively, $\vec{V_i}[k]$is the number of events in process P_k that have been observed by P_i.

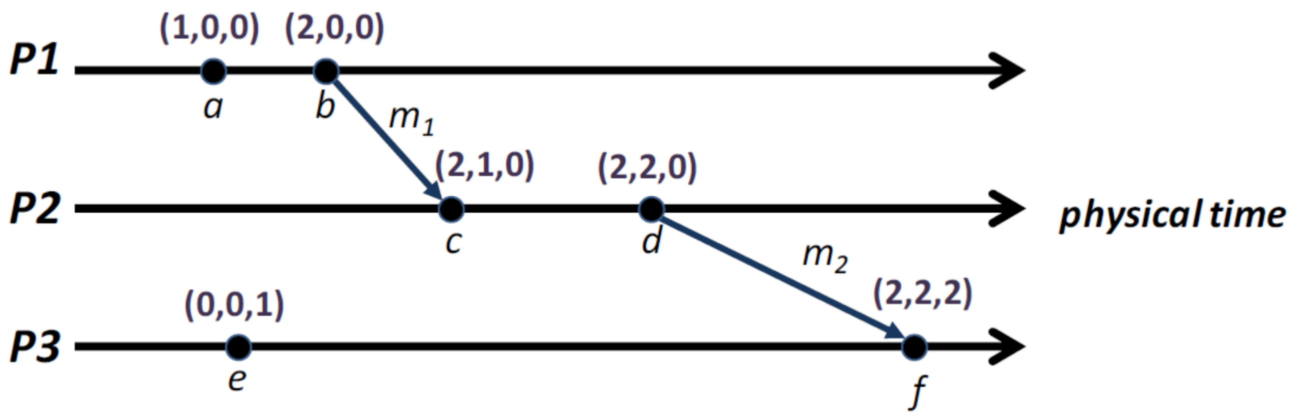

Figure 1: Vector clocks example

In the above example, when process P_2 receives message m_1, it merges the entries from P_1's clock by choosing the maximum value of each position. Similarly, when P_3 receives m_2, it merges in P_2's clock, thus incorporating the changes from P_1 that P_2 already saw. Vector clocks explicitly track the transitive causal order: f's timestamp captures the history of a, b, c, & d.

To use vector clocks for ordering, we can compare them piecewise:

- We say $\vec{V_i} = \vec{V_j}$ if and only if $\vec{V_i}[k] = \vec{V_j}[k] \forall k$.
- We say $\vec{V_i} \leq \vec{V_j}$ if and only if $\vec{V_i}[k] \leq \vec{V_j}[k] \forall k$.
- We say $\vec{V_i} < \vec{V_j}$ if and only if $\vec{V_i} \leq \vec{V_j}$ and $\vec{V_i} \neq \vec{V_j}$.
- We say $\vec{V_i} \sim \vec{V_j}$ otherwise, e.g., $\vec{V_i} = [2, 0, 0]$ and $\vec{V_j} = [0, 0, 1]$.

For any two event timestamps $T(a)$ & $T(b)$:

- if $a \to b$, then $T(a) < T(b)$; **and**
- if $T(a) < T(b)$, then $a \to b$.

Hence, we can use timestamps to determine if there is a causal ordering between any two events.

Lamport Clocks

	a	b	c	d	e	f
P_1	1	2				
P_2			3	4		
P_3					1	5

Is it e → c or e ∼ c?

Vector Clocks

	a	b	c	d	e	f
P_1	(1,0,0)	(2,0,0)				
P_2			(2,1,0)	(2,2,0)		
P_3					(2,2,1)	(2,2,2)

It is e ∼ c!

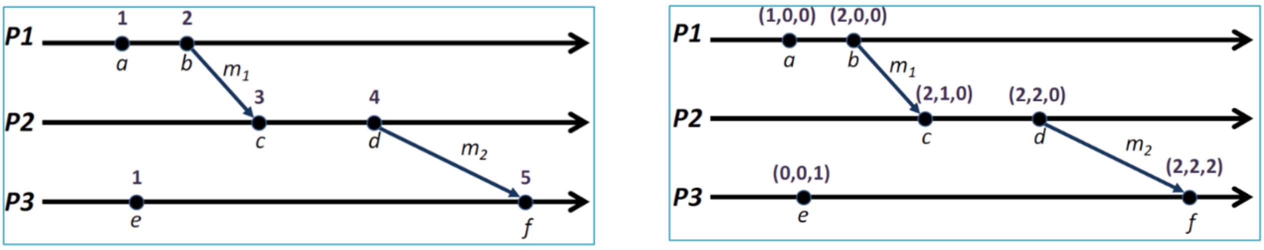

Figure 2: Lamport clocks versus vector clocks

4 The NTP Protocol

The options for computer clocks are essentially as follows:

- **Option A:** stick to crystals.
  - Costly precision manufacturing.
  - Works indoors.
  - Temperature Compensated Crystal Oscillator (TCXO).
  - Oven Controlled Crystal Oscillator (OCXO).

- **Option B:** buy an atomic clock ($50,000 – $100,000) or a GNSS receiver (based on an atomic clock, but doesn't work indoors), or a time signal radio receiver if you are based in central Europe.
- **Option C:** use software-based approaches to discipline cheap crystal clocks. Less quality, but useful for certain applications, and works indoors.

Distributed master clocks provide a time reference to hosts that are inter-connected via a network. The underlying time-synchronisation protocols combine aspects of Cristian's algorithm (i.e., RTD calculation) and Berkeley's algorithm (i.e., combining multiple reference time sources). Good time synchronisation requires:

- Good time references: easily done with GPS, atomic clocks, etc.
- Predictable / symmetric / deterministic network latencies: doable in LAN setups, but not guaranteed in Internet data communication.

There are two main distributed master clock protocols:

- **Network Time Protocol (NTP):** defined in RFC 5905, originally used in the Unix-based NTP daemon, one of the first Internet protocols that ever evolved.
- **Precision Time Protocol (PTP):** designed for managed networks, e.g., LAN.

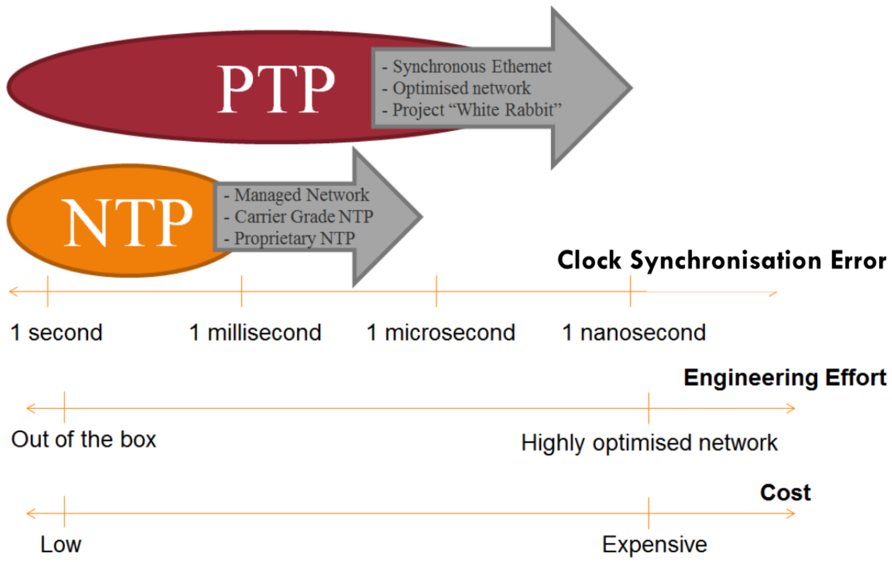

Figure 3: NTP & PTP characteristics

In **uni-directional synchronisation** a reference clock sends a timestamp to a host via a network. The host then uses the timestamp to set its local clock. Useful when message latencies are minor relative to the synchronisation levels required.

In **round-trip synchronisation (RTS)** a host sends a request message, receives a reference clock response message with known (local) submission & arrival times, allowing for the calculation of the round-trip delay (1) and the host clock error (2), i.e., the phase offset. Variations of RTS form the basis for NTP & PTP.

$$\delta = (T_{i+3} - T_i) - (T_{i+2} - T_{i+1}) \tag{1}$$

$$\theta = (T_{i+1} - T_i) + \frac{(T_{i+2} - T_{i+3})}{2} \tag{2}$$

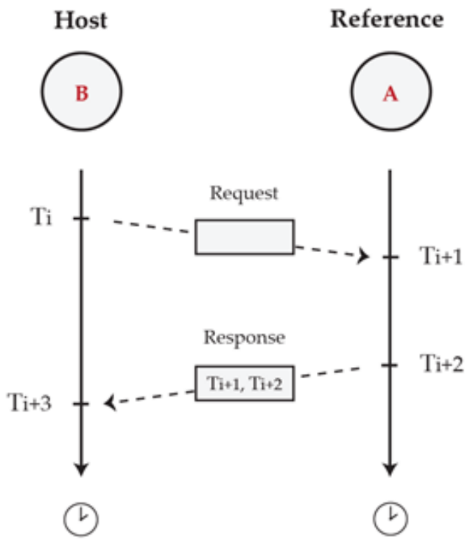

Figure 4: RTS example

4.1 NTP

The NTP architecture, protocol, & algorithms have evolved over the last 40 years to the latest NTP Version 4. NTP is the standard Internet protocol for time synchronisation & co-ordinated UTC time distribution. It is a fault-tolerant protocol, as it automatically selects the best of several available time sources to synchronise with. It is highly scalable, as the nodes form a hierarchical structure with reference clock(s) at the top:

- Stratum 0: Time Reference Source (e.g., GPS, TAI atomic clocks, DCF 77).
- Stratum 1: Primary Time Server.

NTP applies some general aforementioned principles such as avoiding setting clocks backwards and avoiding large step changes; the required change (positive or negative) is amortised over a series of short intervals (e.g., over multiple ticks).

NTP is the longest-running and continuously operating Internet protocol (since around 1979). Government agencies in many other countries and on all continents (including Antarctica) operate public NTP primary servers. National & regional service providers operate public NTP secondary servers synchronised to the primary servers. Many government agencies, private & public institutions including universities, broadcasters, financial institutions, & corporations operate their own NTP networks.

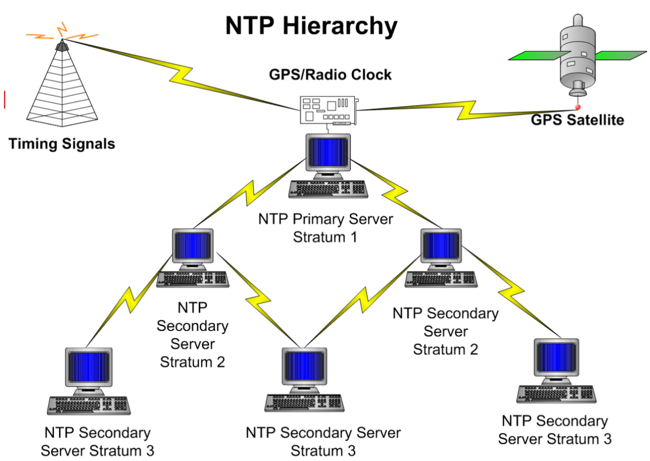

Figure 5: NTP hierarchy

In **client/server mode**, UDP is used for data transfer (no TCP), i.e., NTP over UDP on UDP port 123. There is also the optional use of broadcasting or multicasting (not covered here). Several packet exchanges between the NTP client and the Stratum server take place to determine the client offset:

1. The client sends a packet with **originate timestamp** A.
2. The server receives the packet and returns a response containing the originate timestamp A as well as the **receive timestamp** B and the **transmit timestamp** C.
3. The client receives this packet and processes A, B, C, as well as the packet arrival time D of the received packet; it then determines the offset and the round-trip delay (RTD).

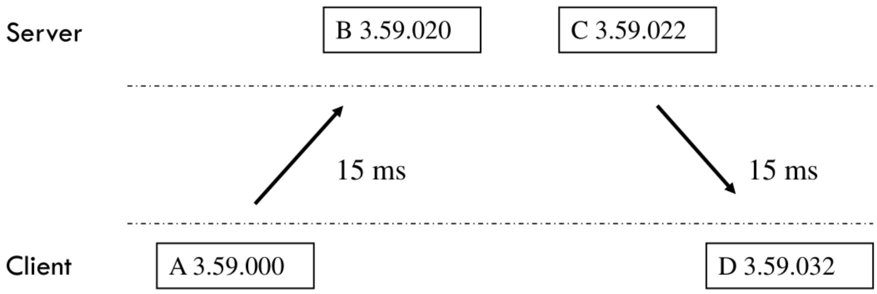

Figure 6: NTP operation

The above example is of a symmetric network with a 15ms delay each way; the client's clock lags 5ms behind the server's clock:

$$\text{RTD} = (D - A) - (C - B) = 32 - 2 = 30\text{ms}$$
$$\text{Offset} = \frac{((B - A) + (C - D))}{2} = \frac{(20 + (-10))}{2} = 5\text{ms}$$

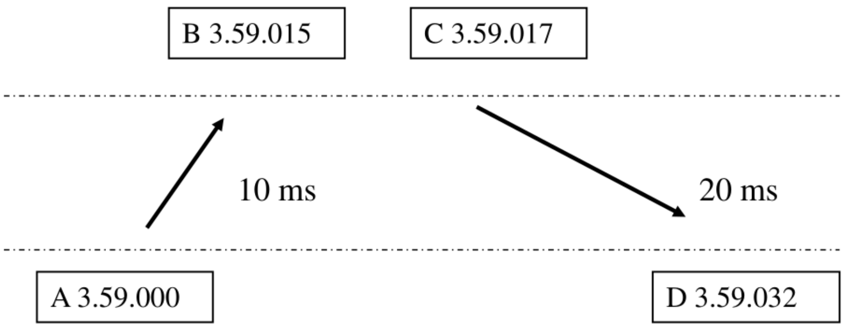

Figure 7: Network delay asymmetry

In the above example, the client's clock still lags 5ms behind the servers clock, but there is an asymmetric network latency: 10ms versus 20ms:

$$\text{RTD} = (D - A) - (C - B) = 32 - 2 = 30\text{ms}$$
$$\text{Offset} = \frac{((B - A) + (C - D))}{2} = \frac{(15 + (-15))}{2} = 0\text{ms}$$

Typical NTP performance for various set-ups is as follows:

- Small LAN: ~10 microseconds best possible case on a 2-node LAN, ~220 microseconds on a real-world small LAN.

- Typical large-building LAN: ~2ms.
- Internet with a few hops: 10–20ms.
- Long distance and/or slow or busy link: 100ms–1s.

Accuracy is further degraded on networks with asymmetric traffic delays.

The **NTP time format** has a reference scale of UTC. Time parameters are 64 bits long:

- Seconds since January 1, 1900 (32 bits, unsigned).
- Fraction of second (32 bits, unsigned).

The NTP time format has a dynamic range of 136+ years, with rollover in 2036. Its resolution is 2^{-32} seconds ~232 picoseconds.

4.1.1 NTP Protocol Header

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|LI | VN  |Mode |    Stratum     |     Poll      |  Precision   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Root Delay                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Root Dispersion                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Reference ID                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                     Reference Timestamp (64)                  +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                      Origin Timestamp (64)                    +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                      Receive Timestamp (64)                   +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                      Transmit Timestamp (64)                  +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 8: NTP protocol header

- `LI` (**Leap Indicator**) 2-bit:
  - `0`: no warning;
  - `1`: last minute of the day has 61 seconds;
  - `2`: last minute of the day has 59 seconds.
- `VN` (**Version Number**) 2-bit: currently 4.
- `Mode`: 3-bit integer, including:
  - `3`: client;
  - `4`: server.
- `Stratum`: 8-bit integer for Stratum server hierarchy level, including:
  - `1`: primary server (i.e., stratum 1);

  – `2–15`: secondary server.

- `Poll`: 8-bit signed integer representing the maximum interval between successive message, in $\log_2$ seconds. This field indicates the interval at which the client will poll the NTP server for time updates. The client dynamically adjusts this interval based on its clock's stability & the network conditions to balance accuracy & network load.
- `Precision`: 8-bit signed integer representing the resolution of the system clock (the tick increment) in $\log_2$ seconds; e.g., a value of -18 corresponds to a resolution of about one microsecond (2^{-18}) seconds.
- `Root Delay`: round-trip packet delay from a client to a stratum 1 server. It gives a crude estimate of the worst-case time transfer error between a client and a stratum 1 server due to network asymmetry, i.e., if all of the round-trip delay was in one direction and none in the other direction.

For a single client clock, the **dispersion** is a measure of how much the client's clock might drift during a synchronisation cycle:

$$\text{Dispersion} = DR \times (D - A) + TS$$

where $(D - A)$ is the duration of a synchronisation cycle, with A being the first timestamp and D being the last timestamp, DR being the local clock skew (i.e., the deviation of actual clock tick frequency to nominal clock tick frequency), and TS being the timestamping errors due to the finite resolution of the clock and delays in reading the clock when fetching a timestamp.

The **root dispersion** of a client clock is the combined dispersions of all stratum servers along the path to a Stratum 1 server.

The **root distance** is the sum of root dispersion and half the root delay. It provides a comprehensive measure of the maximum error in time synchronisation as the total worst case timing error accumulated between the Stratum 1 server and the client.

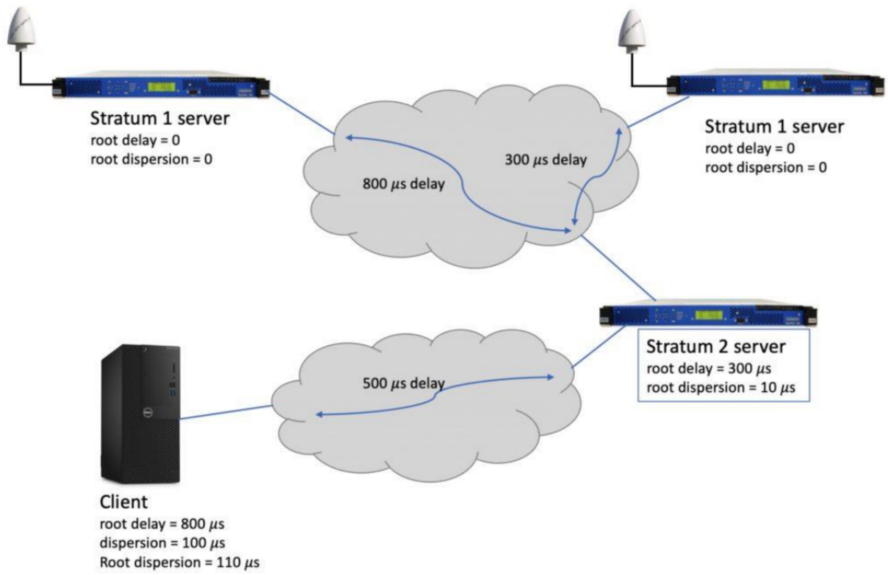

Figure 9: Example root dispersion & root delay

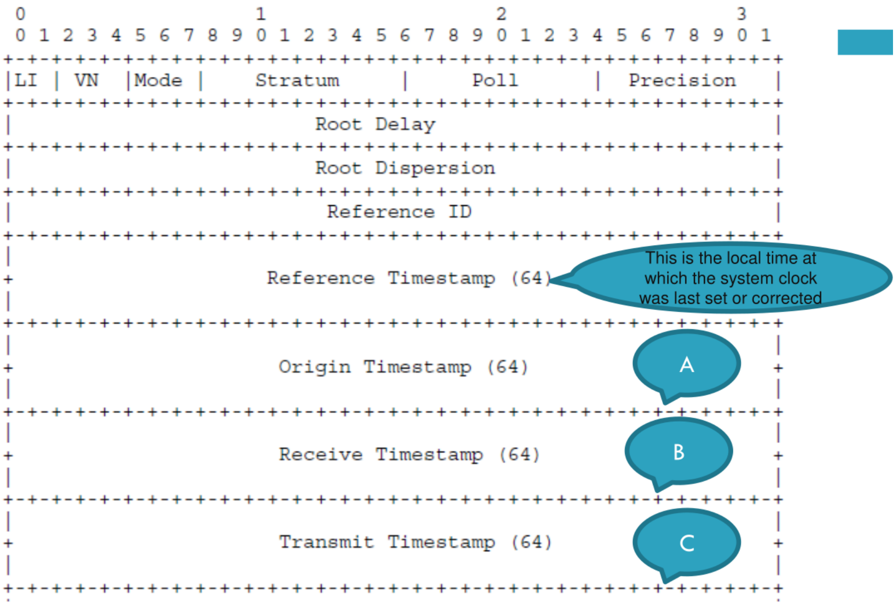

Figure 10: Annotated NTP protocol header

A **reference ID (refid)** is a 32-bit code (4 ASCII bytes) identifying the particular server or reference clock, e.g.,:

- `GPS`: Global Positioning System;
- `GAL`: Galileo Positioning System;
- `PPS`: Generic Pulse-Per-Second;
- `DCF`: LF Radio DCF77 Mainflingen, DE 77.5 kHz;
- `WWV`: HF Radio WWV Fort Collins, Colorado;
- `GOOG`: unofficial Google refid used by Google NTP servers as `time4.google.com`.

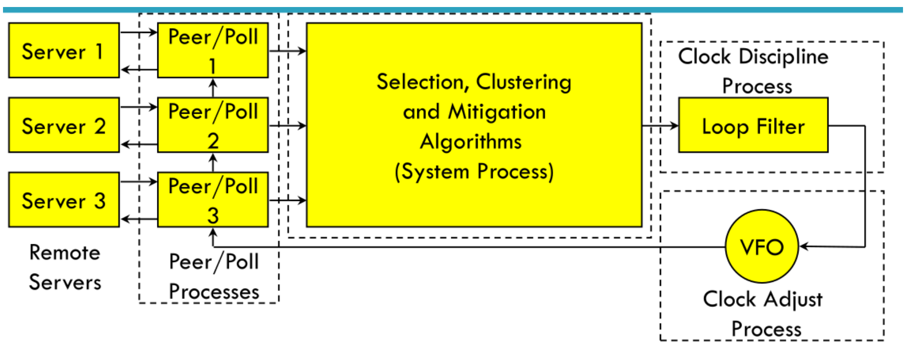

Figure 11: NTP architectural overview

An NTP client synchronises with multiple stratum servers. It uses a range of algorithms to deal with variable & asymmetric non-deterministic network delays and to determine its most likely offset, thereby running a series of processes:

- The peer process runs when a packet is received;
- The poll process sends packets at intervals determined by the clock discipline process & remote server;
- The system process runs when a new update is received;

- The clock discipline process implements clock time adjustments;
- The clock adjust process implements periodic clock frequency (VFO) adjustments.

For each stratum there is a **poll process** that sends NTP packets at intervals ranging from 8 seconds to 36 hours. The corresponding **peer processes** receive NTP packets and, after performing some packet sanity tests, $T1 - T4$ are determined / extracted. The NTP daemon calculates offset & delay as seen before. The time series of offset & delay values calculated by multiple peer processes are processed by a sequence of algorithms, thus eliminating servers with long RTD or servers that show "strange" offsets which, for example, are often the result of network asymmetries.

4.1.2 Mitigation Algorithms

The **clock filter algorithm** uses a sliding window of eight samples for each stratum server and picks out the sample with the least expected error, i.e., it chooses the sample with the minimum RTD. It is effective at removing spikes resulting from intermittent network congestions.

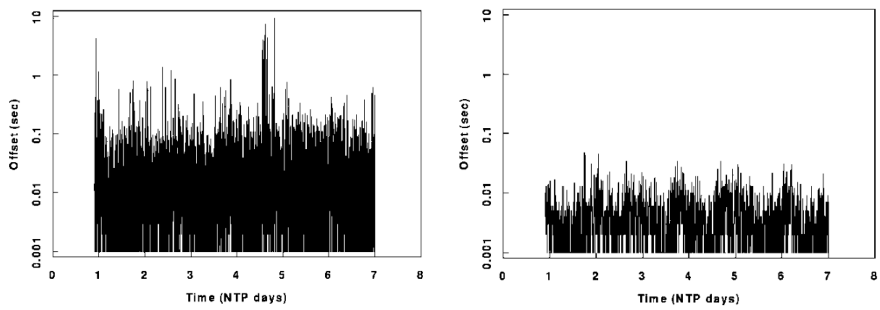

Figure 12: Clock filter algorithm before and after

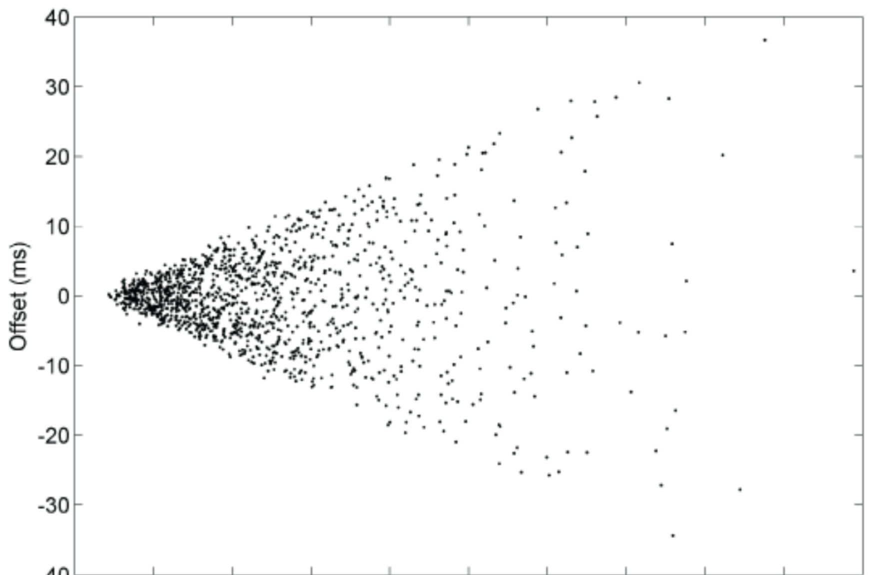

Figure 13: The wedge scattergram plots sample points of offset versus delay (RTD) collected over a 24-hour period by a client clock communicating with a single stratum server. For this experiment, the client clock is externally synced to the stratum server, so the offset should be zero; however, as the (network) round trip delay increases, the offset variability increases, resulting in increasingly larger offset errors. Therefore, the best samples are those at the lowest delay. This is taken into account by the clock filter algorithm.

The **intersection algorithm** selects a subset of peers (i.e., stratum servers) and identifies truechimers & truetickers based on the intersection of confidence (offset) intervals (i.e., min/max offsets of a clock over x readings determines its invterval).

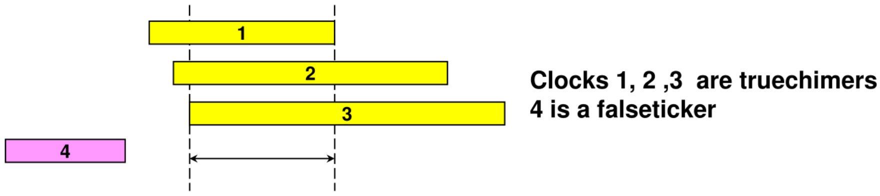

Figure 14: Plot range of offsets calculated by each peer with 1, 2, & 3 overlapping

Marzullo's algorithm is an agreement protocol for estimating accurate time from a number of noisy time sources by intersecting offset intervals; if some intervals don't intersect, it considers the intersection of the majority of intervals. It eliminates false tickers.

The **clock cluster algorithm** processes the truechimers returned by the clock intersection algorithm. It produces a list of survivors by eliminating truechimers that have a comparably large root delay & root dispersion. Finally, the **clock combining algorithm** averages the time offsets of survivors using their root dispersions as a weight, i.e., survivors with a small root dispersion have a higher weight.

The **combining algorithm** provides a final offset, so the client clock can be adjusted. The UNIX Clock Model provides the kernel variable `tickadj` which amortises the required change gradually by making adjustments every tick e.g., every 10 milliseconds.

5 Soft RTS

A **soft RTS** is a RTS in which performance is degraded but not destroyed by failure to meet response time constraints, e.g., multimedia systems.

5.1 Quality of Service (QoS)

Perceived QoS reflects the subjective evaluation of service quality from the end user's perspective and overall satisfaction. It encompasses factors like application responsiveness, ease of use, consistency, reliability, and the overall user interface delay. It is challenging to measure accurately as it involves subjective experiences & user feedback, often gathered through surveys, user studies, or feedback mechanisms.

Intrinsic QoS refers to the technical & measurable characteristics of a network or service that directly affects its performance & reliability. Intrinsic QoS parameters include bandwidth, latency, packet loss rate, jitter, & throughput. Intrinsic QoS measures are typically quantifiable and can be objectively measured using various tools & monitoring techniques.

5.1.1 QoS Metrics

Latency is the time it takes to send a packet from point A (say, the client) to point B (say, the server). It is physically limited by how fast signals can travel in wires or the open air. Latency depends on the physical, real-world distance between A and B. Typical latencies are conceptually small, between roughly 10 and 200 milliseconds. High latency can result in delays between user actions and system responses, leading to sluggish or unresponsive behaviour in real-time applications.

Jitter is the inconsistency or fluctuations in the arrival time of data packets at the receiver. It can be caused by various factors, such as network congestion, packet loss, routing changes, etc. It can have significant implications for real-time communication applications, particularly voice & video sharing. Inconsistent packet arrival times can lead to disruptions, distortion, and out-of-sync audio or video playback.

Bandwidth measures the amount of data that is able to pass through a network at a given time. It is measured in bits per second. Real-time applications with high-bandwidth requirements, such as high-definition video streaming &

VOIP may experience performance issues if the available bandwidth is insufficient to accommodate the data transmission demands.

5.2 Real-Time Multimedia Technologies

5.2.1 Transport Layer for Real-Time Media

TCP provides reliability, ordered delivery, & congestion control. Retransmissions can lead to high delay and cause delay jitter. TCP is not suitable for real-time systems and does not support mulitcast.

UDP has no built-in reliability or congestion control, and no defined technique for synchronising. It has low latency and minimal overhead (no handshake, no retransmissions). A feedback channel must be defined for quality control.

5.2.2 Service Requirements for Real-Time Flows (Voice / Video)

- **Sequencing:** the process of maintaining the correct order of data packets during transmission and ensuring that they are re-assembled correctly at the receiver's end.
- **Synchronisation:** ensures that different types of data streams (such as audio & video) are aligned in time during playback.
- **Payload identification:** different media types (MPEG1, MPEG2, H.261) may require different handling in terms of decoding or processing.
- **Frame indication:** specifying which packets belong to the same frame or video sequence and helps with decoding & rendering video frames accurately.

5.2.3 RTP

Real-time Transport Protocol (RTP) provides end-to-end transport functions suitable for real-time audio/video applications over multicast & unicast network services. It works in user space over UDP. The working model is as follows:

- The multimedia application generates multiple streams (audio, video, etc.) that are fed into the RTP library.
- The library multiplexes the streams & encodes them in RTP packets which are fed to a UDP socket.

Secure RTP (SRTP) is used by applications including WhatsApp, Zoom, Skype, etc. for transporting voice & video streams.

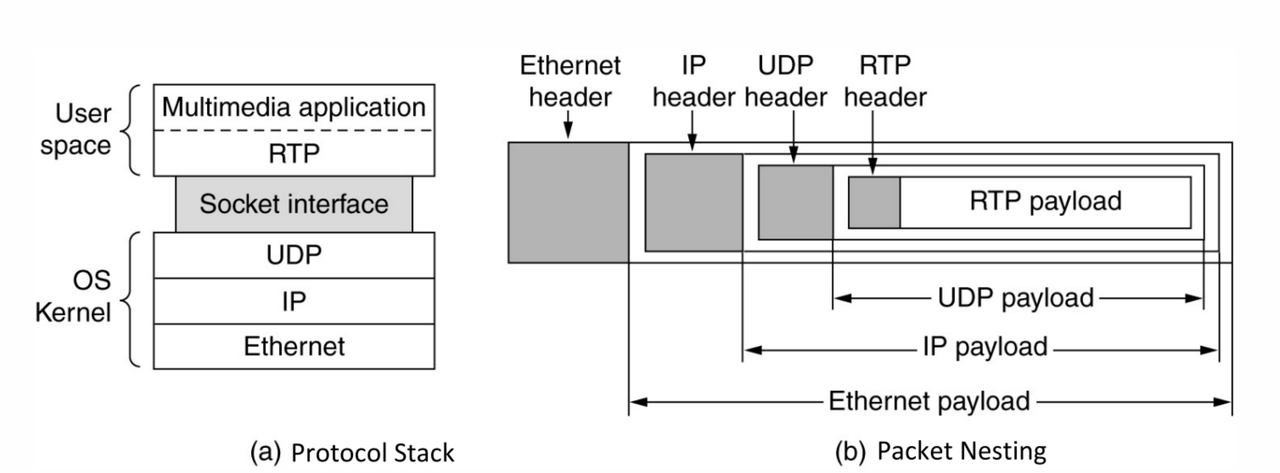

Figure 15: RTP protocol

RTP services include:

- Payload type identification: determination of media encoding.
- Synchronisation source identification: assigned to each distinct media source (such as a microphone or a camera). Enables synchronisation of multiple streams coming from the same source (e.g., lip-syncing audio & video).
- Sequence numbering: a counter is used which is incremented on each RTP packet send and is used to detect lost packets.
- Time-stamping: time monitoring, synchronisation, jitter calculation.
- RTP issues: no QoS guarantees, no guarantee of packet delivery.

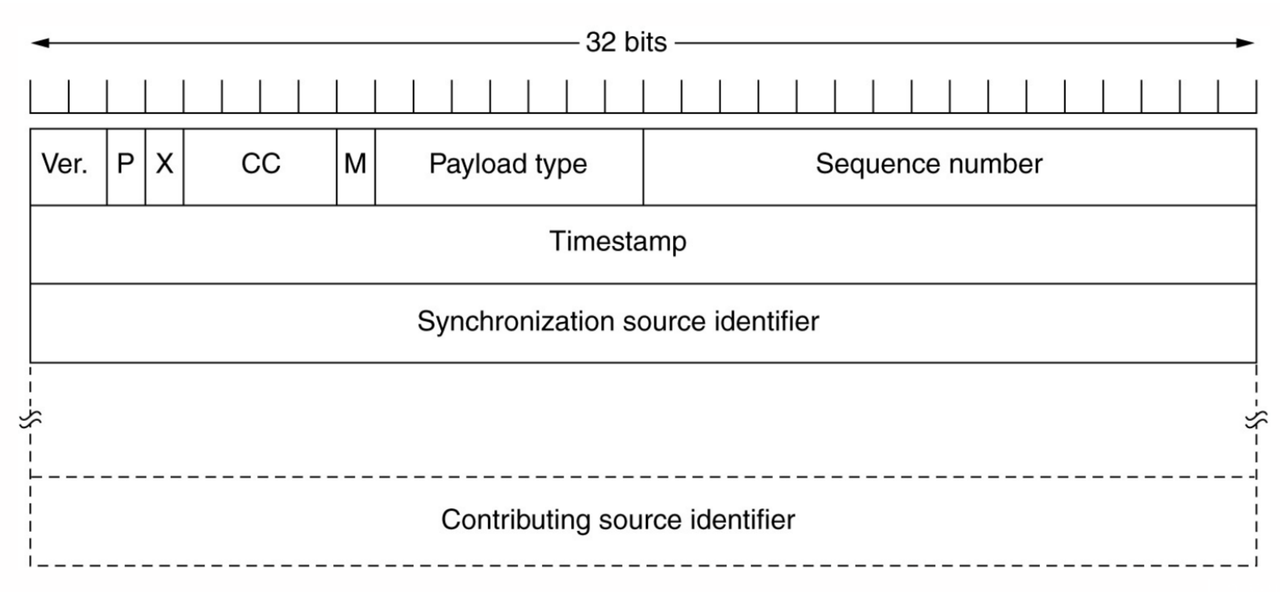

Figure 16: RTP header

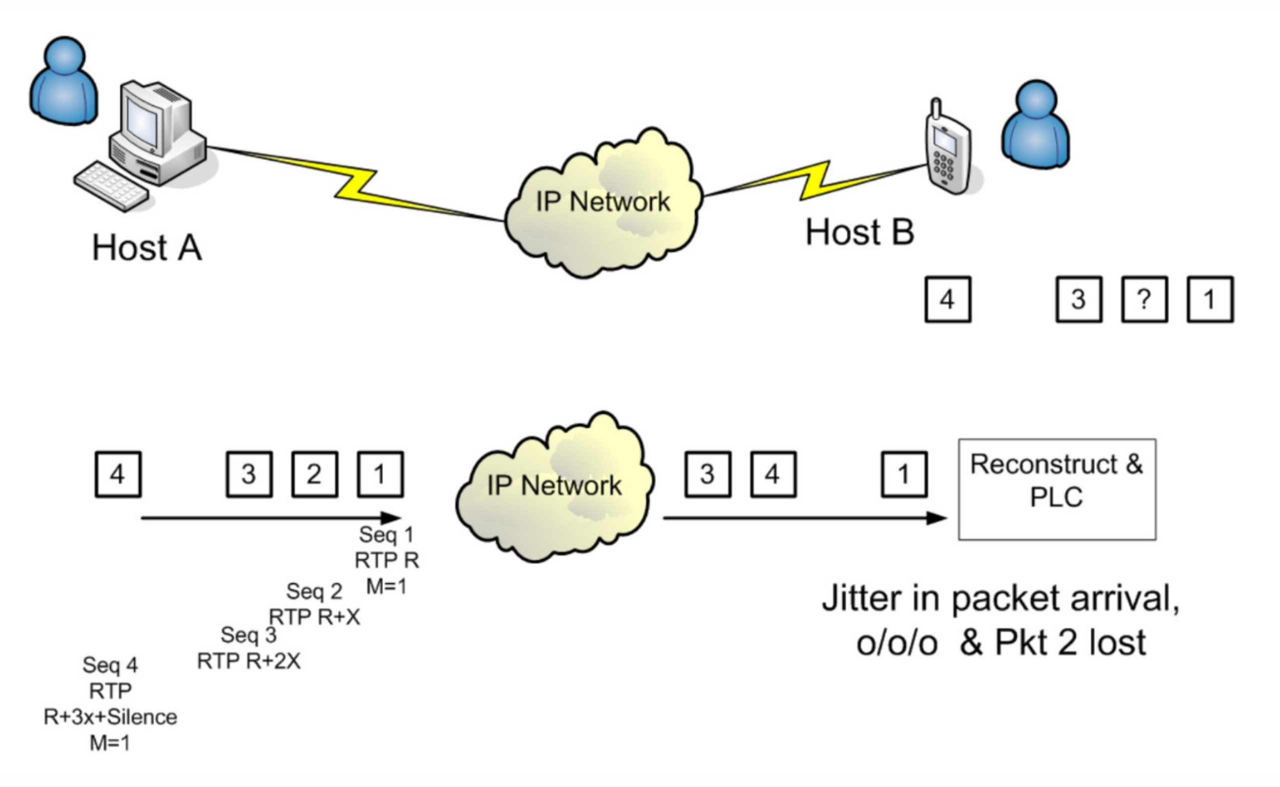

Figure 17: RTP data delivery

RTP Control Protocol (RTCP) is a companion protocol to RTP that is used periodically to transmit control packets to participants in a streaming multimedia session. It gathers statistics on the media connection, including bytes sent, packets sent, lost packets, jitter, feedback, & round-trip delay. It provides feedback on the quality of the service being provided by RTP but does not actually transport any data. The application may use this information to increase the quality of service, perhaps by limiting flow or using a different codec.

5.2.4 Mouth-to-Ear M2E Delay

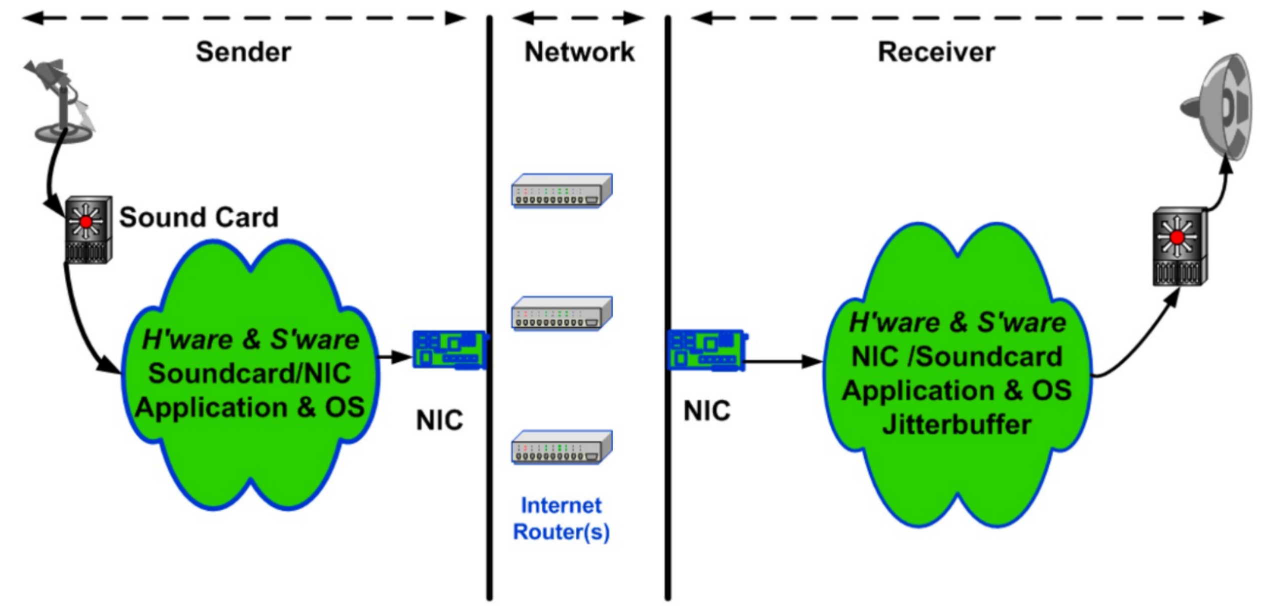

Figure 18: M2E delay

M2E delay consists of:

- Sender:
  - Packetisation delay.
  - Encoding delays.
  - OS/Application/Driver software.
  - MAC delays.
- Network:
  - Propagation delay.
  - Queuing delays.
  - Processing/serialisation delays in routers.
- Receiver:
  - NIC delays.
  - OS/Application/Driver software.
  - Jitter buffer delays.
  - Decoding delays.

VOIP QoS strategies include:

- Sender-based:

  - RTCP feedback with adaptive codeces: if loss/delay excessive, switch to lower-bandwidth code or implement Forward Error Correction (FEC) strategy.

- Network-based:
  - Prioritising delay-sensitive traffic flows.

- Receiver-based:
  - Buffering strategies: the human ear is not sensitive to short-term variations; the buffer "absorbs" variation in network queueing delays and the voice can be reconstructed using RTP timestamps, but this adds to overall M2E delay.
  - Packet Loss Concealment (PLC) and FEC.

5.3 Packet Loss Strategies

The use of UDP limits delays but can lead to packet loss; to mitigate this, compensation strategies can be employed at the sender or receiver:

- Forward Error Correction (FEC): a form of information redundancy.
- Packet Loss Concealment (PLC): silence (simplest), repetition of last packet, or interpolation.

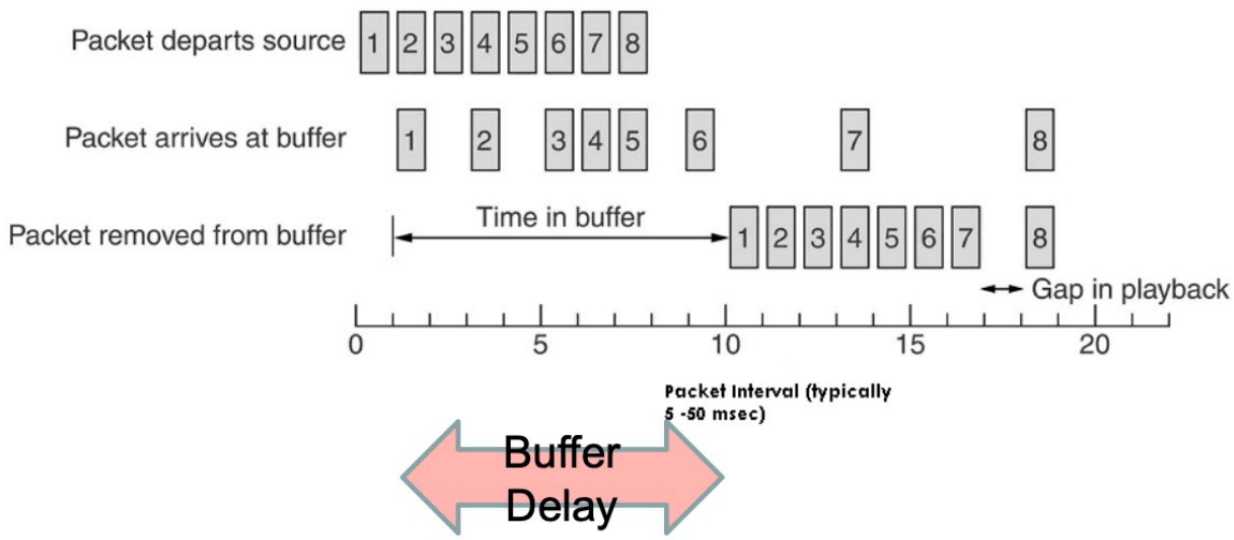

Figure 19: Receiver-based packet loss strategies: jitter buffer strategies

The buffer playout delay adds to the M2E delay. In the above figure, packet 8 arrives too late for the playout; we could drop the packet, or increase the buffer size in response to increasing delays. With a fixed buffer, the limitations are that if it is too large, it extends overall delay, and if it is too small, there will be additional late packet losses due to late arrival. An adaptive buffer size can adapt to network conditions:

- **Per Talkspurt (PT):** operates by elongating/shortening inter talkspurt silence periods; less noticeable.
- **Per Packet Scaling (PPS):** speed up/slow down speech, like in Skype or Zoom.

5.3.1 Network-Based Strategies

In a LAN environment: switched LANs are typically QoS enabled, and fast ethernet links are rarely congested. In a WAN environment, we could increase bandwidth (which is costly and mostly a temporary solution) or employ a reservation policy, traffic categorisation, & prioritisation which requires an admission control policy.

5.4 Cloud Gaming

In **cloud gaming**, the game is installed and executed on a powerful remote server located in the cloud. The game rendering (processing graphics, physics, & game logic) is done on this server. Once the game is rendered on the server, the video frames & audio are compressed and streamed to the player's device via the internet. The player's inputs, such as controller buttons, mouse movements, or keyboard presses are sent back to the could server over the Internet in real-time. It is commonly used in Massively Multiplayer Online Games (MMOG), and the server must deal with multiple players. Cloud gaming solves several issues of the traditional approach:

- Tedious installation & patching process.
- Extensive hardware & GPUs.

It is an RTC application, so we need to minimise delays (lag) and need time synchronisation to measure delays. Simultaneity is also required: all players need to see the same game representation.

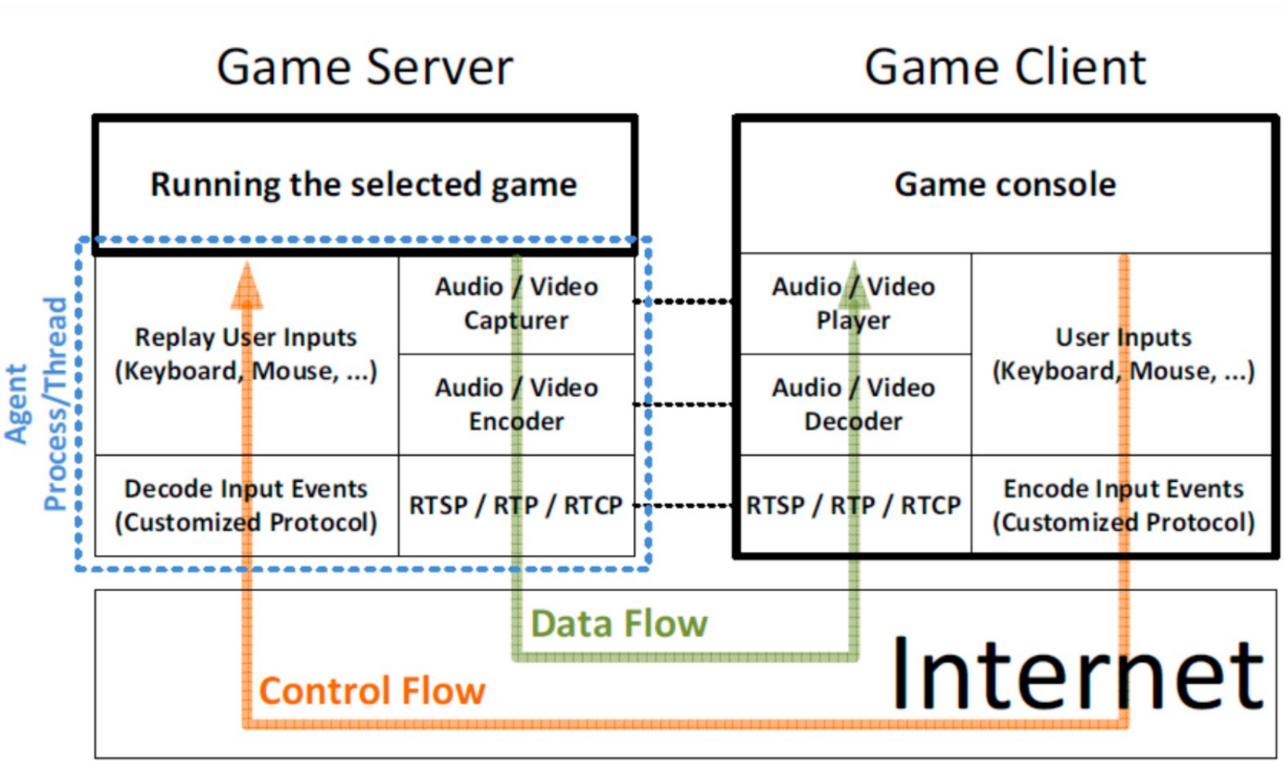

Figure 20: Client/server architecture of GamingAnywhere

5.4.1 Response Delay

- **Response Delay (RD)** is the time difference between a user submitting a command and the corresponding in-game action appearing on the screen.
- **Processing Delay (PD)** is the time required to receive / process a player's command and encode / transmit the corresponding frame.
- **Playout Delay (OD)** is the time required for the client to received, decode, and render a frame on the display.
- **Network Delay (ND)** is the round-trip delay.

$$\text{Response Delay (RD)} = \text{PD} + \text{OD} + \text{ND}$$

5.5 WebRTC

Web Real-Time Communications (WebRTC) is a free, open-source project that provides web browsers & mobile applications with real-time communication (RTC) via simple APIs. It allows audio & video communication to work inside web pages via direct peer-to-peer communication using JavaScript & HTML, that is, eliminating the need to install plugins or download native apps. It uses DTLS for encryption & SRTP for secure media streams. It works on major web browsers (Chrome, Firefox, Safari, Edge) and mobile platforms. Use cases include the web versions of Google Meet, Discord, Facebook Messenger, peer-to-peer file sharing, etc.

5.5.1 WebRTC Working Model

1. **Media capture:** accesses camera & microphone using browser APIs.

2. **Connection management:** exchanges session control messages (via WebSockets, SIP, or custom methods) to set up connections.

3. **Data transmission:** WebRTC uses the Interactive Connectivity Establishment (ICE) techniques to overcome the complexities of real-world networking like NAT:

   - STUN: finds public IP addresses.
   - TURN: relays media if direct connections fail.

   RTP/SRTP streams media, ensuring low-latency delivery.

4. **Peer discovery:** in P2P communication, the peers need to be able to identify or locate each other over the wire. Peer discovery mechanisms are not defined by WebRTC, although the process can be as simple as sharing a URL that peers can use to communicate.

5. **ICE techniques:** ICE will first try to make a connection using the host address obtained from a device's operating system. If the network is unsuccessful, ICE will obtain an external address using the STUN server. If that fails, the traffic is routed via a TURN relay server.

Session Traversal Utilities for NAT (STUN) is a protocol that is used to discover public addresses that determines any restrictions in your router that would prevent a direct connection to a peer. Clients receive their public addresses as requested from STUN servers.

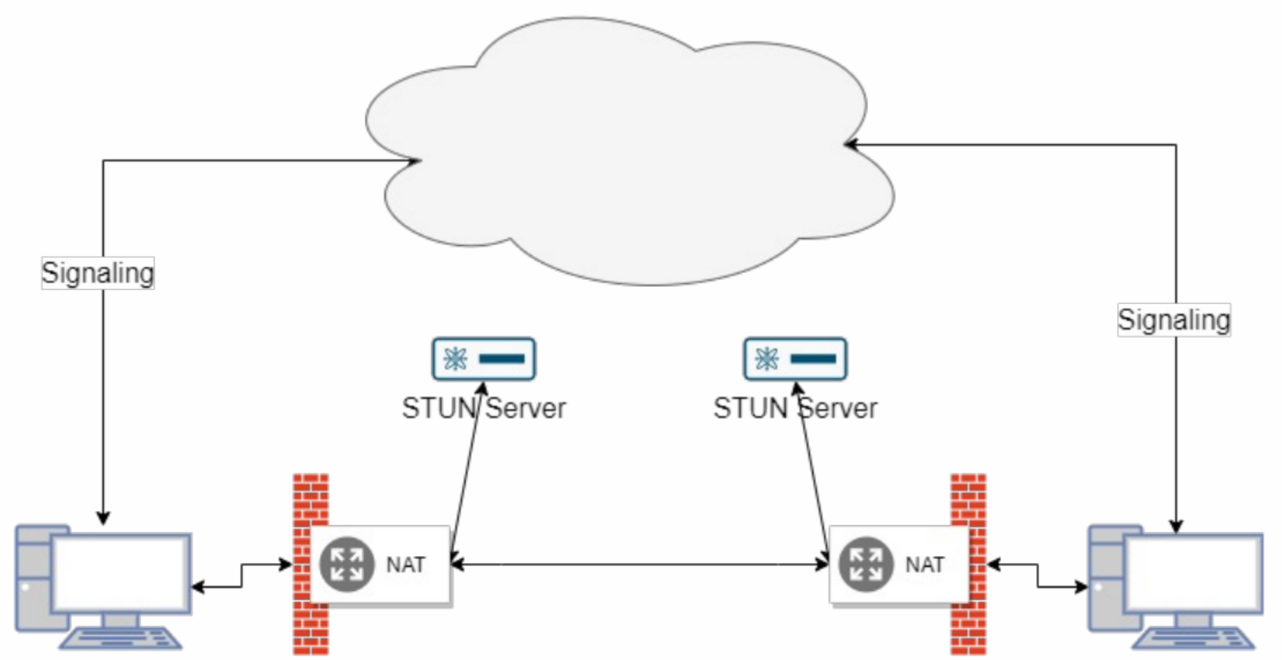

Figure 21: STUN

Traversal Using Relays around NAT (TURN) bypasses the symmetric NAT restriction by opening a connection with a TURN server and relaying all information through that server. A connection is required with a TURN server which will tell all the peers to send packets to the server which will then be forwarded to the requester.

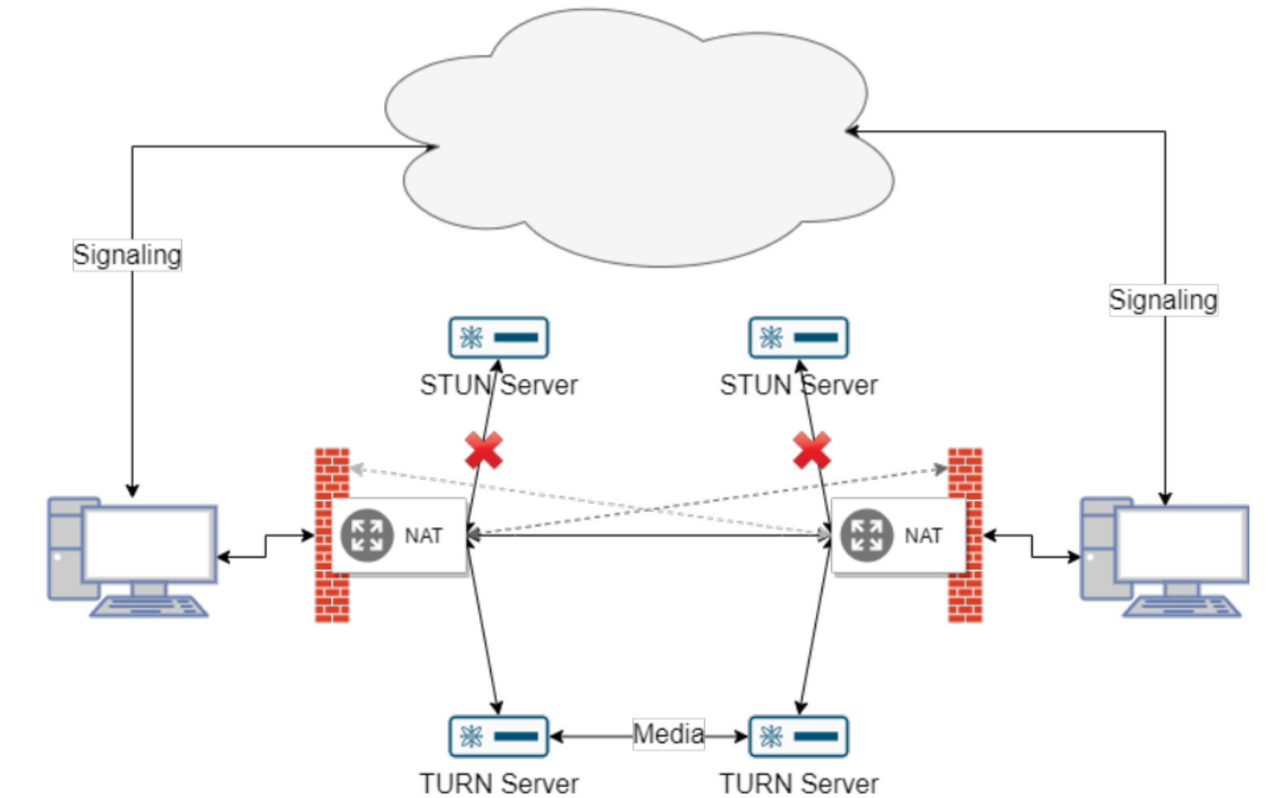

Figure 22: TURN

5.6 DASH

DASH stands for Dynamic Adaptive Streaming over HTTP:

- **Dynamic:** reacts to changing scenarios.
- **Adaptive:** has media represented in different it rates / codecs.
- **Streaming:** not strictly real-time communication, but timelines are still important.
- **HTTP:** web model, HTTP good for firewalls etc.

The DASH working model is as follows:

- **Encoding & Segmentation:** the original video/audio content is divided into small segments (usually 2-10 seconds each). Each segment is encoded at multiple bitrates & resolutions for adaptability. A manifest file is created, containing metadata about segments, their URLs, codecs, & timing.
- **Delivery:** segments and the MPD file are uploaded to HTTP servers or CDNs. The encoded video segments are pushed out to client devices over the Internet.
- **Playback:** the client downloads the MPD file to understand the available content & quality options. It chooses appropriate "representation" based on network conditions, device capabilities, & user preferences, decodes the chunks, and plays back the video.
- **Quality adjustment:** the player continuously downloads & plays segments, adjusting the quality as network conditions change.

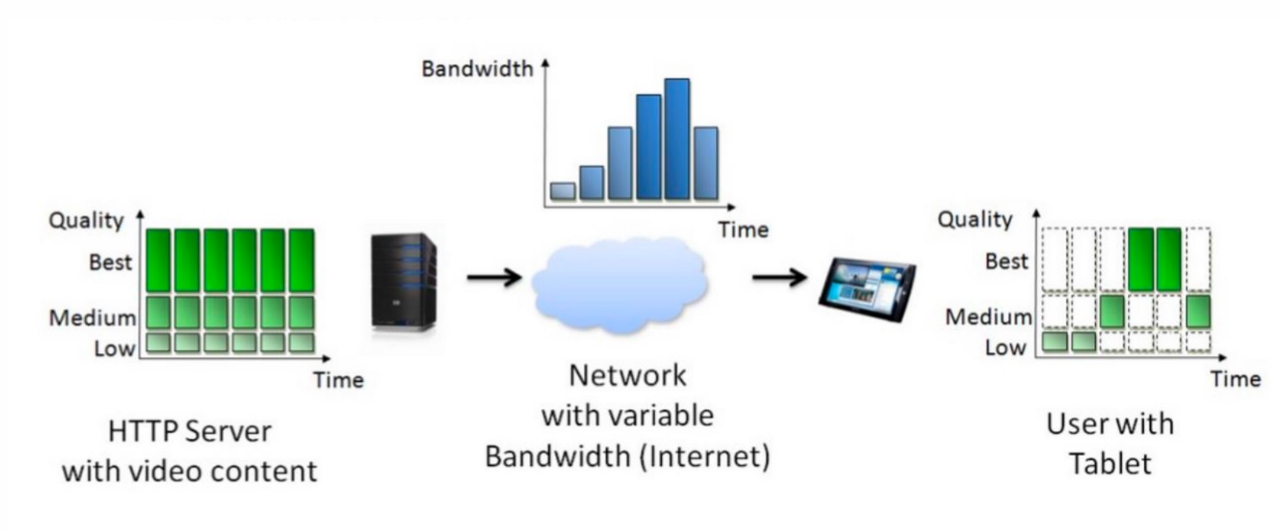

Figure 23: DASH

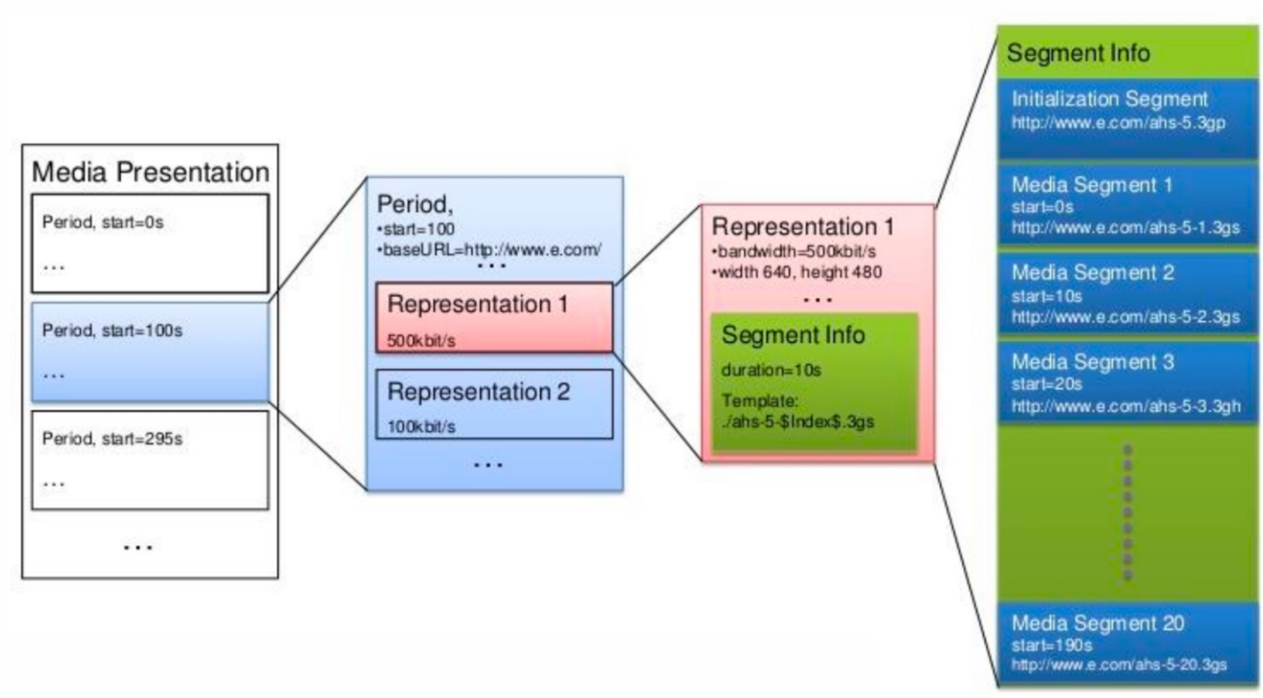

Figure 24: DASH data model

6 Emerging Protocols

QoS Attribute	**Definition**
Accuracy	The error rate produced by the service
Accessibility	The degree to which the service is capable of serving a Web service request
Capacity	The limit of concurrent requests for guaranteed performance
Response Time	The time to complete a Web service request (from the perspective of a client)
Throughput	The number of Web service requests served at a given time period
Availability	The probability that the service can respond to the consumer requests
MTTR	Mean Time To Repair
Interoperability	The ease with which a consumer application or agent interoperates with a service
Robustness	The degree to which a service can function correctly in the presence of invalid, incomplete or conflicting inputs
Authentication	A measure of how the service authenticates principals who can access the service & data
Confidentiality	A measure of how the service protects the data, so that only authorized principals can access or modify the data

Table 1: Quality of Service (QoS) Attributes and Their Definitions

Website speed, or *website performance*, refers to how quickly a browser is able to load fully-functional web pages for a given site. Site speed is important for user experience, SEO, & bounce rate. Faster sites yield more page views, higher engagement, & higher conversion rates.

6.1 How to achieve speed?

The speed limit of information is c, the speed of light. The two main ways of increasing speed are:

- Improving the communication link; &
- Improving the protocols: app protocols provide useful user-level functions, transport protocols provide guarantees to apps, network protocols provide best-effort global packet delivery, & link protocols provide best-effort local packet delivery.

6.2 Evolution of the Web

There is a paradigm shift occurring in the way in which the Internet is accessed: there is a transformation from native applications to web applications. Traditional applications are getting migrated to the cloud, while web applications & services are becoming extremely complex. QoS is very important in web applications. The complexity of web content over the years has necessitated updates to the HyperText Transfer Protocol (HTTP).

6.3 HTTP

6.3.1 HTTP/0.9

In 1991, the first documented official version of HTTP was written as a plain document, less than 700 words in length, named **HTTP/0.9** This version only supported the `GET` method, allowing clients to retrieve HTML documents from a server, but not supporting any other file formats or information upload.

6.3.2 HTTP/1.0

In May 1996, RFC 1945 was published as a final **HTTP/1.0**. In HTTP/1.0, each request/response pair requires opening a new connection. New features of HTTP/1.0 included:

- **Header:** the HTTP header was introduced, thereby allowing the transmission of metadata that made the protocol flexible & extensible;
- **Status code:** HTTP responses now contained a status code, thereby enabling the receiver to check the request processing status;
- **Content-type:** HTTP could transmit other document types other than a plain HTML file;
- **New methods:** two new methods, `POST` & `HEAD`, were added in addition to `GET`.

Establishing a TCP connection using a three-way handshake is expensive; 2 RTTs (Round-Trip Time) are required between the client and the server to establish a connection.

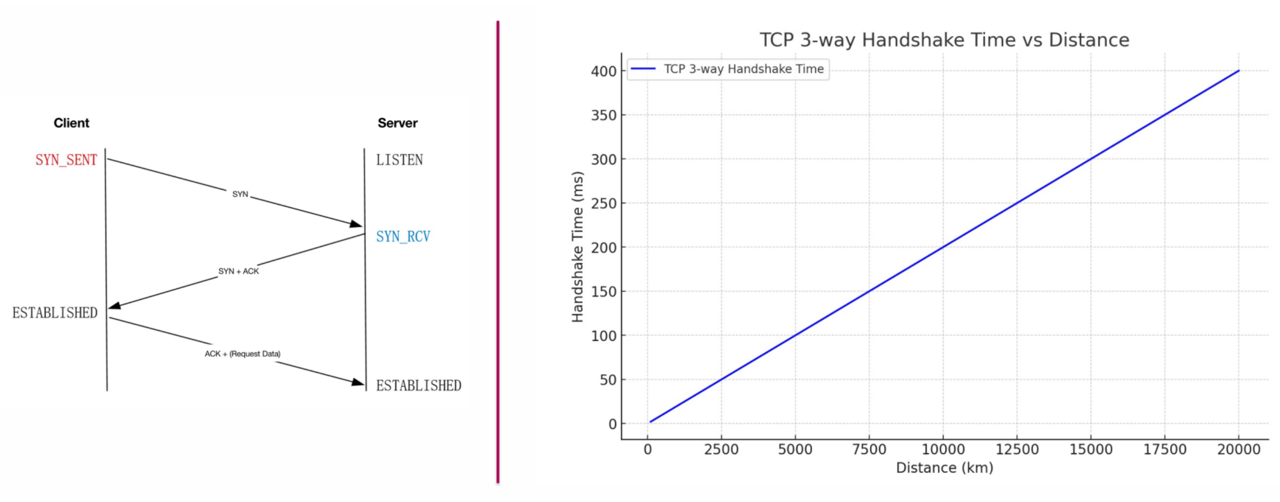

Figure 25: The TCP 3-way handshake is expensive

6.3.3 HTTP/1.1

In January 1997, RFC 2068 was officially released as **HTTP/1.1**. Its most relevant enhancements were:

- **Persistent connections:** in HTTP/1.1, it is possible to execute several requests using a single connection, and thus amortise the cost of the initial connection establishment and the cost of slow starts across multiple requests.
- **New methods:** besides the already-available methods of HTTP/1.0, the 1.1 version added six extra methods: `PUT`, `PATCH`, `DELETE`, `CONNECT`, `TRACE`, & `OPTIONS`.

With persistent connections, multiple requests could share the same connection, but they still had to be serialised one after the other; therefore, a client & a server could only execute a single request/response exchange at any given time for each connection. As the web evolved, more concurrency was required when fetching & rendering web pages with a large number of resources (such as CSS, JavaScript, images, etc.). The only way to gain concurrency at the network layer was to use multiple TCP connections to the same origin in parallel, but this has several negative effects:

- **Initial overhead:** establishing & maintaining multiple connections requires additional overhead;
- **Network congestion:** multiple parallel connections increase the amount of traffic on the network;
- **Inefficient use of resources:** every TCP connection requires resources on both the client & the server sides, such as buffers, sockets, & memory; &
- **Fairness issues:** some users or applications may monopolise bandwidth by opening many TCP connections.

6.3.4 SPDY

An unofficial HTTP protocol named **SPDY** was developed by Google in 2009 as an experimental protocol to improve web performance. SPDY opens one connection per domain; multiple data streams are multiplexed over this single TCP connection for efficiency. This reduces the amount of redundant header information sent every time a new page is requested. Higher priority resources could be transferred faster than lower priority resources. SPDY was supported by the Chrome browser and deployed in most Google services.

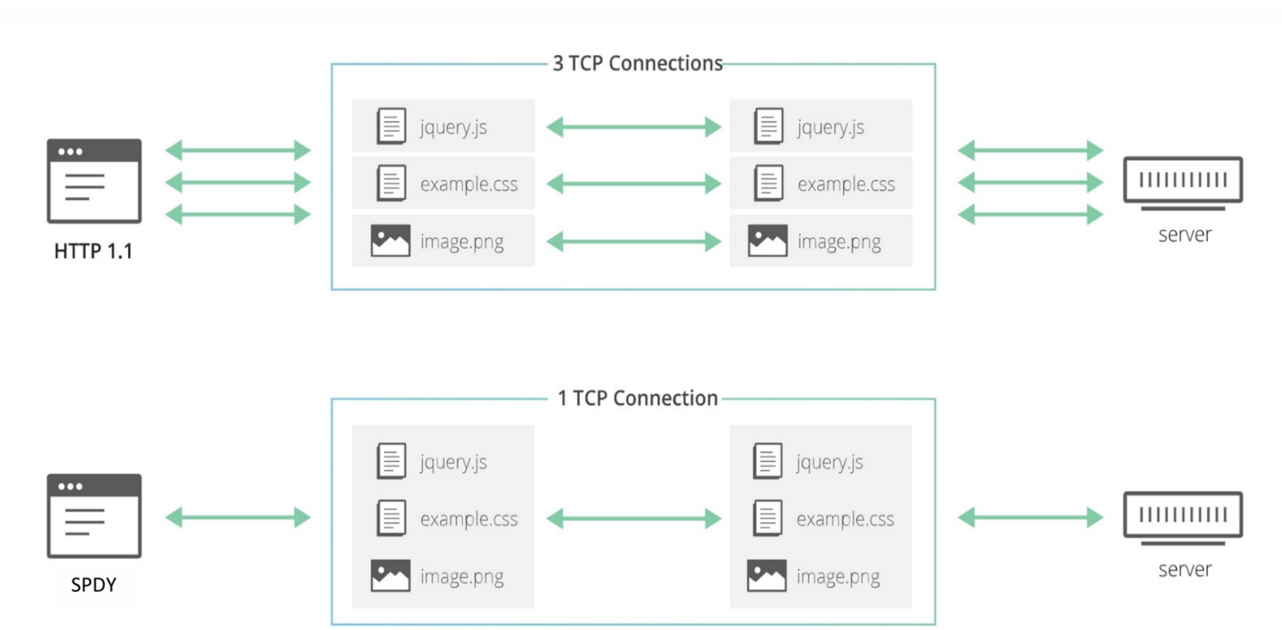

Figure 26: HTTP/1.1 versus SPDY

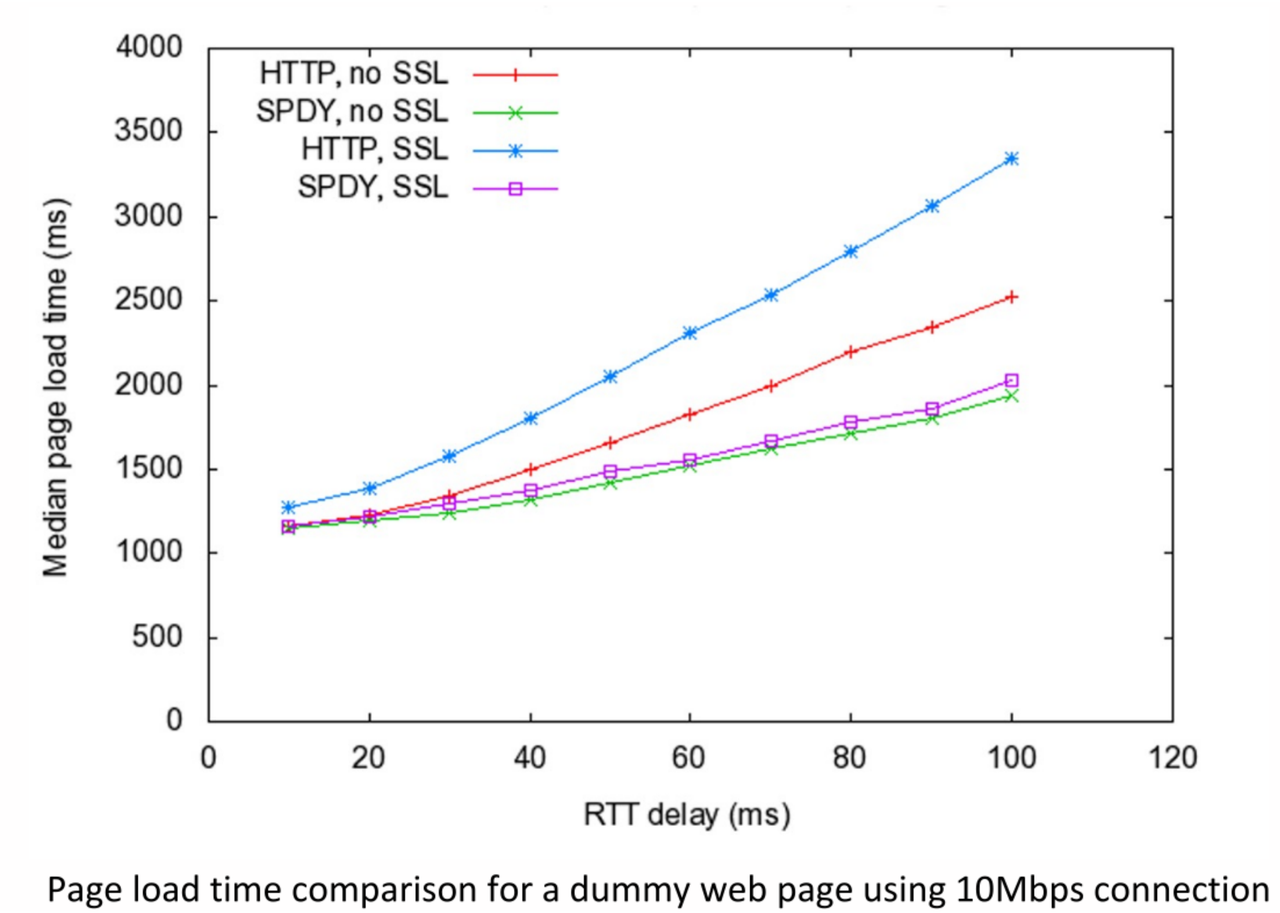

Figure 27: SPDY performance

6.3.5 HTTP/2.0

HTTP/2.0 was officially released in 2015 (RFC 7540), about 18 years after HTTP/1.1. HTTP/2.0 is largely derived from SPDY, and implements several new features to improve the protocol performance:

- **Request multiplexing:** HTTP/1.1 was a *sequential protocol*, meaning that we can only send a single request at a time; HTTP/2.0 allows us to send requests & receive responses asynchronously, thus allowing us to make multiple requests at the same time using a single connection.

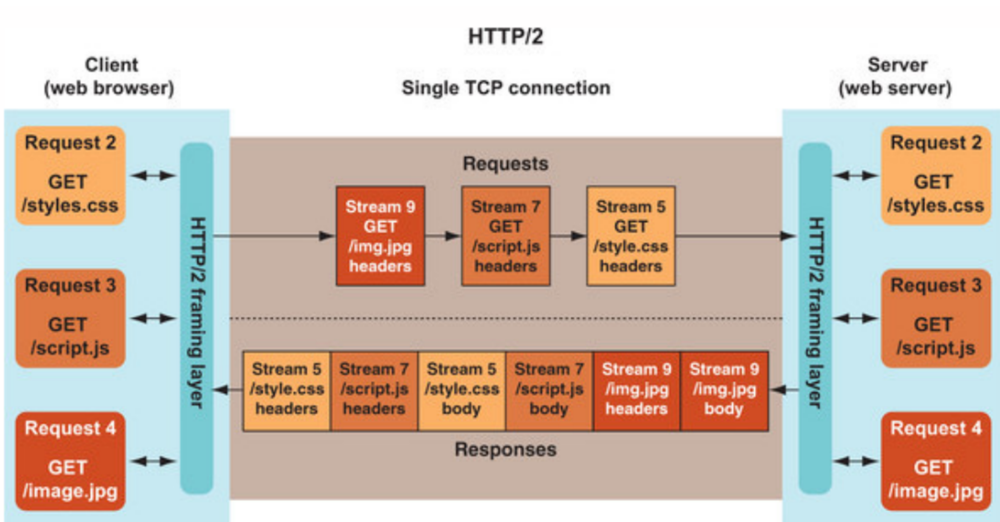

Figure 28: HTTP/2.0 in action

- **Request prioritisation:** HTTP/2.0 allows us to set a priority for each request, thus allowing us to explicit about the order in which we expect responses, such as getting a webpage's CSS before its JavaScript file.
- **Header compression:** headers are compressed with HPACK (RFC 7541), with 76% compression of the ingress header, & 69% compression of the egress header.
- **Server push:** to avoid a server receiving lots of requests, HTTP/2.0 introduced a server push functionality. With that, the server tries to predict the resources that will be requested soon. So, the server proactively pushes these resources to the client cache.

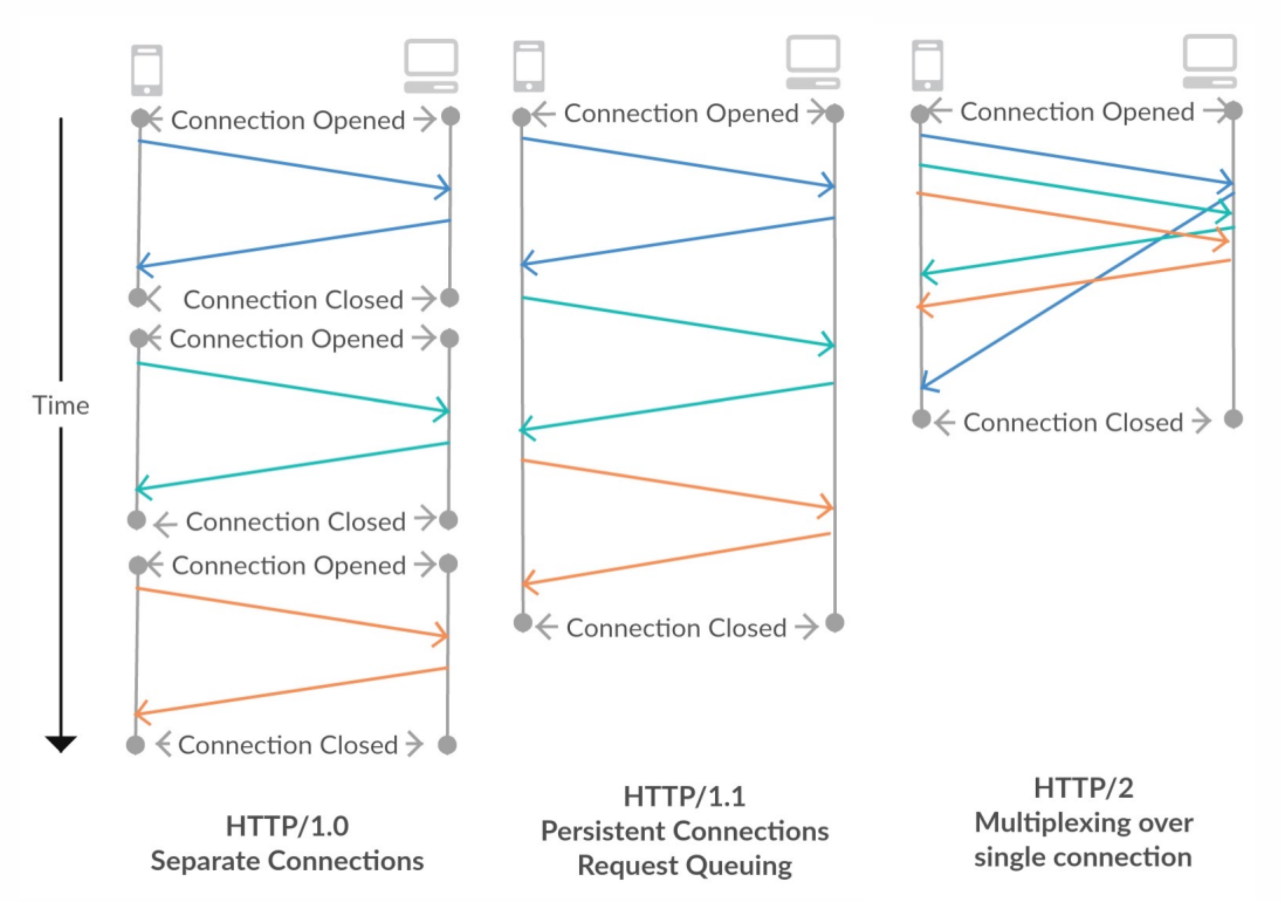

Figure 29: HTTP version comparison

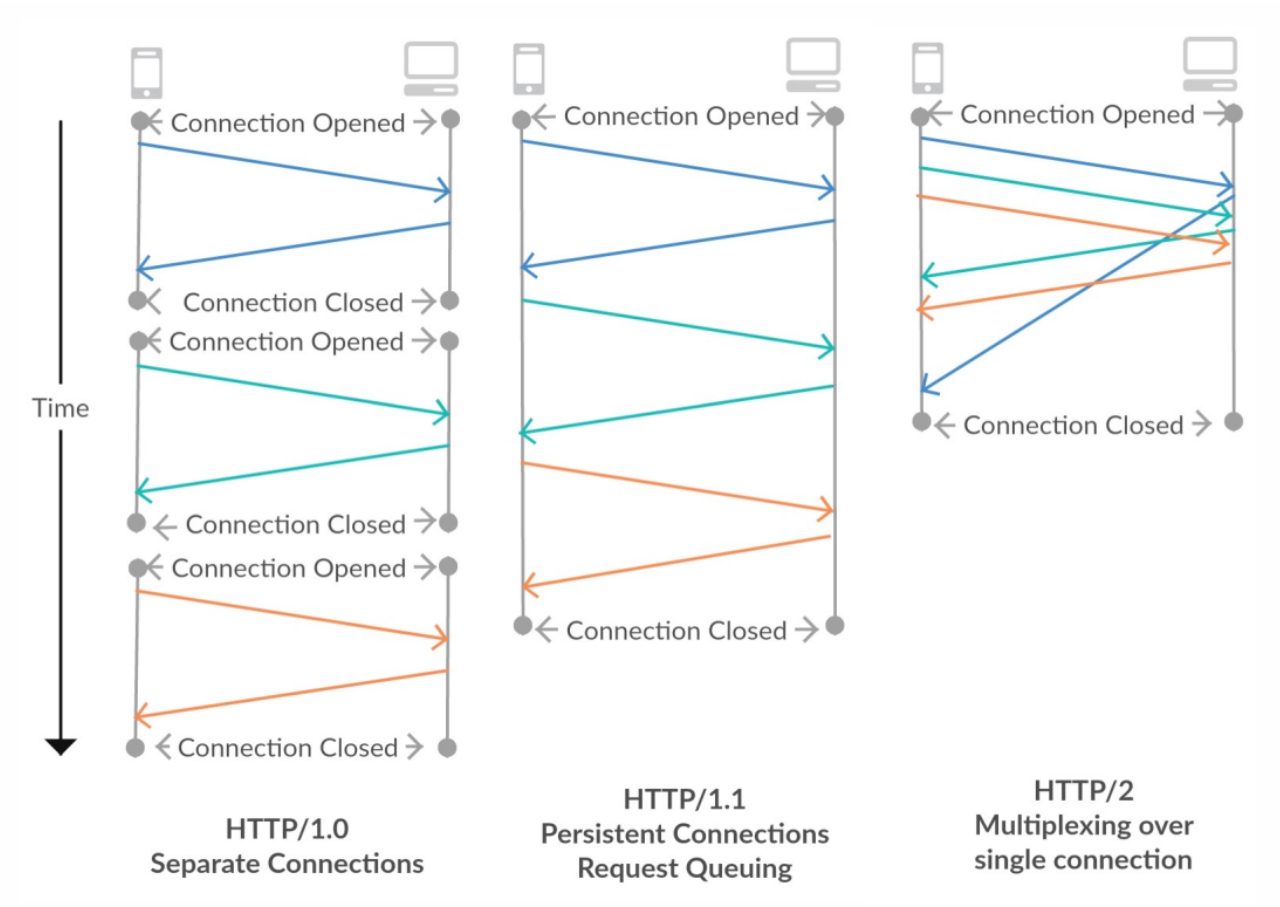

Figure 30: HTTP version comparison

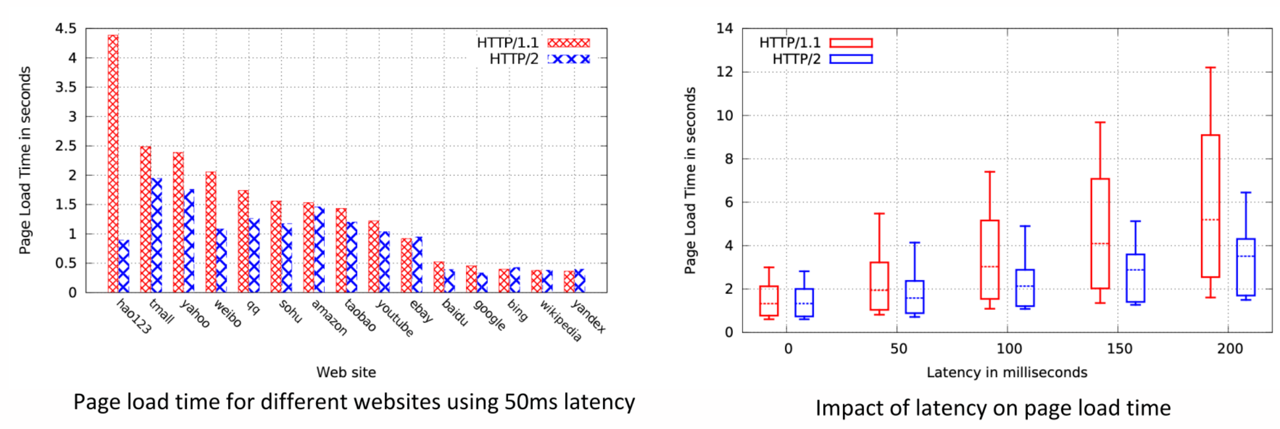

Figure 31: HTTP/1.0 versus HTTP/2.0 performance

Improvements at the application layer have been implemented in HTTP/2.0; to further improve the performance, fundamental changes to the underlying transport layer are required.

6.4 Transport Protocols

The **Transmission Control Protocol (TCP)** ensures reliability & flow control (by ensuring that data sent is delivered to the receiver application and ensuring that the receiver buffer doesn't overflow), ordered delivery (by ensuring bits pushed by the sender arrive at the receiver application in order), & congestion control (by ensuring that the data sent doesn't overwhelm network resources).

The **User Datagram Protocol (UDP)** facilitates abstraction of independent messages between endpoints. It has minimal overhead, making it the recommended transport to meet the strict latency bounds of real-time applications, but provides limited support to applications. There is no guarantee of delivery.

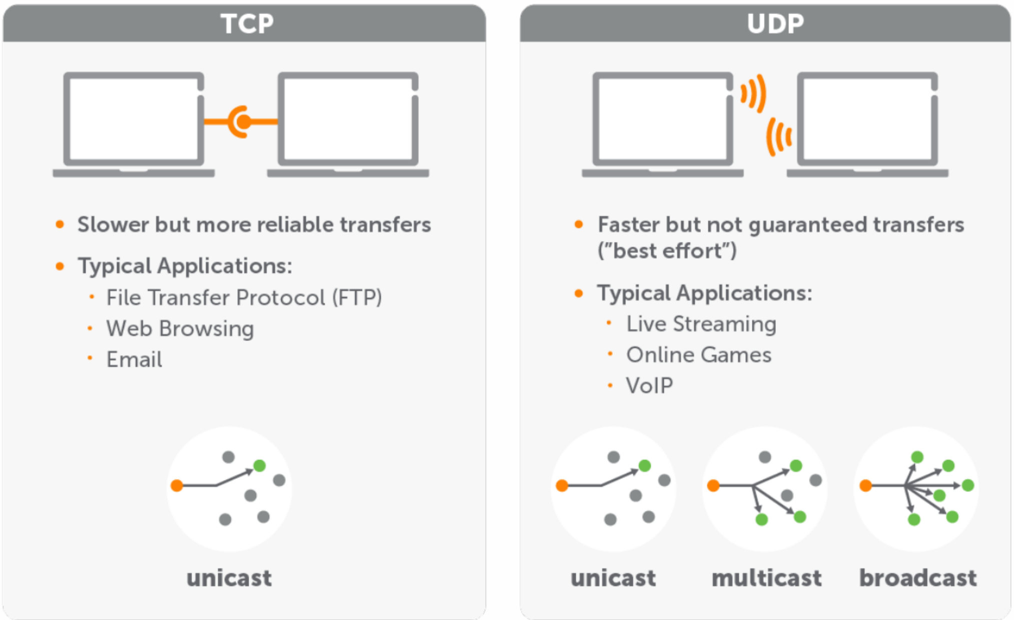

Figure 32: TCP versus UDP

6.4.1 Issues with TCP

- Slow connection establishment.
- The TCP 3-way handshake has very high latency, particularly for small web requests. This problem is made worse by adding security (HTTP/TLS) over TCP: the TLS handshake adds more round trips.
- Head-of-line blocking.

TCP is implemented in the kernel of the operating system, and to change TCP, you would likely have to change the operating system kernel on all servers & clients around the world. You might even need to change the entire network, as middleboxes may drop packets if they don't understand something on the packet.

6.5 QUIC

The **QUIC** protocol was developed to overcome the performance issues in TCP. It is a reliable transport over UDP. It was standardised in 2021, and in 2022 the IETF standardised HTTP/3 as RFC 9114 which uses QUIC by default. As of 2025, around 27-30% of all website support HTTP/3.

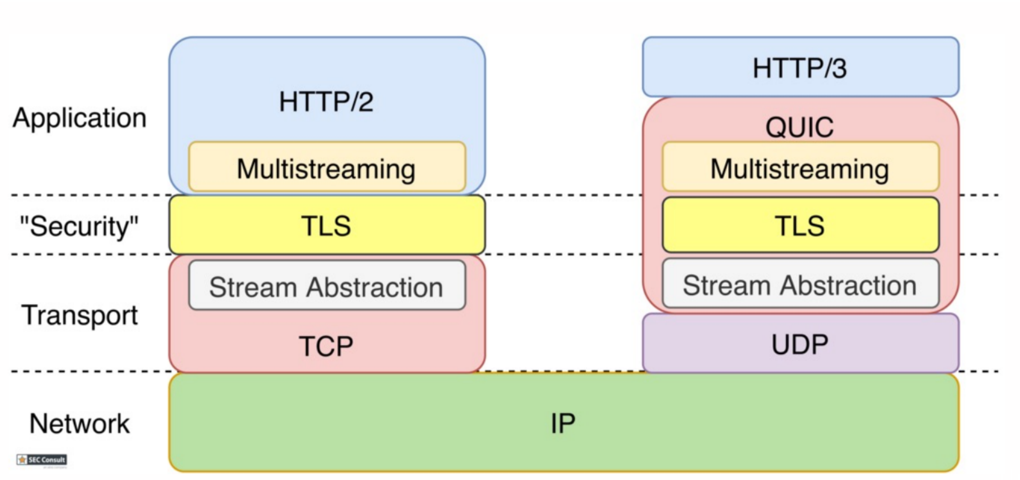

Figure 33: Where does QUIC fit?

The main difference between HTTP/2.0 and HTTP/3.0 is the underlying transport layer protocol: HTTP/2.0 uses TCP connections, while HTTP/3.0 uses QUIC (UDP-based) connections.

QUIC features include:

- **Multiplexing without head-of-line blocking:** in QUIC, each byte stream is transport independently over the network. QUIC ensures the in-order delivery of packets within the same byte stream; if a packet gets lost, only the affected stream gets blocked and waits for its re-transmission.
- **Low-latency connections:** WUIC has a faster connection setup, as it combines the transport handshake with the TLS cryptographic session establishment, which in TCP+TLS are two separate processes. To reduce the time required to establish a new connection, a client that has previously connected to a server may cache certain parameters from that connection and subsequently set up a **0-RTT connection** with the server; with the 0-RTT feature, a HTTP request can be sent, and a (partial) response can be received during the very first handshake.
- **Connection migration:** QUIC improves performance during network-switching events, like what happens when a user of a mobile device moves from a local WiFi hotspot to a mobile network. When this occurs on TCP, a lengthy process starts where every existing connection times out one-by-one and is then re-established on demand. QUIC includes a connection identifier to uniquely identify the connection to the server regardless of source; this allows the connection to be re-established simply by sending a packet, which always contains this ID, as the original connection ID will still be valid even if the user's IP address changes.
- **Linkability prevention:** to avoid privacy issues, e.g., to prevent hackers from following the physical movement of a smartphone user by tracking the unencrypted CID across networks, QUIC uses a list of connection identifiers instead of just one.
- **Encryption:** TCP + TLS only encrypts the actual HTTP data. QUIC also encrypts large parts of the protocol header. Metadata, such as packet numbers and connection-close signals, which were visible to all middleboxes (& attackers) in TCP, are now only available to the client & server in QUIC.
- **Resistance to protocol ossification:** protocol ossification is an inherent characteristic of protocols implemented in the oeprating system kernel, such as TCP; kernels are rarely updated, which applies even more to the operating systems of middleboxes, such as firewalss & load balancers. It is a problem because it makes it hard to introduce new features, as middleboxes with an older version of the protoocl don't recognise the new feature and drop the packets. QUIC aims to solve this issue: it runs in the user space instead of the kernel, so it's easier to deploy new implementations. Furthermore, QUIC is heavily encrypted, so if a middlebox can't read a piece of information, it can't make any decisions based on that information.
- **QPACK field compressoin:** QPACK is a field compression format for HTTP/3.0. Field compression eliminates redundant metadata by assigning indexes to fields that are used multiple times during the connection. The goal

of QPACK is to reduce the space taken up by the HTTP headers: smaller size translates to higher throughput and better user experience. The effectiveness of data compression is expressed as a compression ratio.

A QUIC packet is composed of a common header followed by one or more frames.

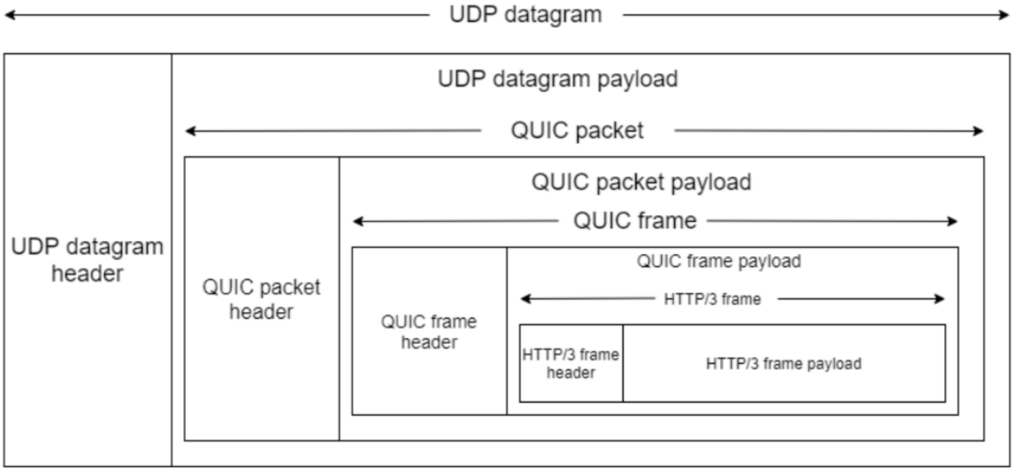

Figure 34: QUIC packet structure

The QUIC endpoints communicate by exchanging packets; the QUIC packet consists of a header & a payload. QUIC has two different types of headers: the long header is used prior to the connection establishment, and the short header is used after the first connection is established. Packets are carried in UDP datagrams.

QUIC provides different packet types:

- **Initial packet:** transports the first CRYPTO frames transmitted by the client & server during the key exchange and the ACK frames in both directions.
- **Handshake packet:** used for sending & receiving encrypted handshake messages & acknowledgements between the server & the client.
- **0-RTT packet:** sends “early” data from the client to the server before the handshake is completed.
- **1-RTT packet:** used to exchange data between the client & the server after the handshake is completed.

```
QUIC IETF
    QUIC Connection information
    [Packet Length: 1350]
    1... .... = Header Form: Long Header (1)
    .1.. .... = Fixed Bit: True
    ..00 .... = Packet Type: Initial (0)
    .... 00.. = Reserved: 0
    .... ..00 = Packet Number Length: 1 bytes (0)
    Version: draft-29 (0xff00001d)
    Destination Connection ID Length: 8
    Destination Connection ID: 45fb5955dfaa8914
    Source Connection ID Length: 0
    Token Length: 0
    Length: 1332
    Packet Number: 1
    Payload:
5a99e5b29413627619ca3b5add4cf8b6ce348355b1c1a2be9874c7961e7996a24aeec
860…
    TLSv1.3 Record Layer: Handshake Protocol: Client Hello
    PADDING Length: 997
```

Figure 35: QUIC initial packet example

A QUIC packet is composed of a common header followed by one or more frames. There are various types of frames:

- **`ACK` frame:** receivers send `ACK` frames to inform senders of packets they have received & processed. The `ACK` frame contains one or more `ACK` ranges.
- **`CRYPTO` frame:** used to transmit cryptographic handshake messages.
- **`STREAM` frame:** implicitly creates a stream and carries stream data. It contains a Stream ID, Offset, Length, & Stream Data.
- **`MAX_DATA` frame:** used in flow control to inform the peer of the maximum amount of data that can be sent on the connection as a whole.
- **`MAX_STREAM_DATA` frame:** used in flow control to inform a peer of the maximum amount of data that can be sent on a stream.
- **`MAX_STREAMS` frame:** informs the peer of the cumulative number of streams of a give type it is permitted to open.

```
TLSv1.3 Record Layer: Handshake Protocol: Client Hello
    Frame Type: CRYPTO (0x0000000000000006)
    Offset: 0
    Length: 314
    Crypto Data
    Handshake Protocol: Client Hello
```

```
TLSv1.3 Record Layer: Handshake Protocol: Server Hello
  Frame Type: CRYPTO (0x0000000000000006)
  Offset: 0
  Length: 90
  Crypto Data
  Handshake Protocol: Server Hello
    Handshake Type: Server Hello (2)
    Length: 86
    Version: TLS 1.2 (0x0303)
    Random:
0f58bdbd934450c7aa98242121447bef2fe0733aa5fc3beffab6513c7177f9a4
    Session ID Length: 0
    Cipher Suite: TLS_AES_128_GCM_SHA256 (0x1301)
    Compression Method: null (0)
    Extensions Length: 46
    Extension: key_share (len=36)
    Extension: supported_versions (len=2)
```

Figure 36: QUIC `CRYPTO` frame examples

Streams in QUIC provide a lightweight, ordered byte-stream abstraction to an application. Streams can be created by either endpoint, and can concurrently send data interleaved with other streams. Streams are identified within a connection by a numeric value, referred to as the stream ID (a 62-bit integer) that is unique for all streams on a connection. Client-initiated streams have even-numbered stream IDs and server-initiated streams have odd-numbered stream IDs.

7 Congestion Control in QUIC

Web performance is usually measured by the following metrics:

- **Latency:** we will often need quite a few round trips to load even a single file due to features such as congestion control. Even low latencies of less than 50 milliseconds can add up to considerable delays. This is one of the main reasons why CDNs exist: they place servers physically closer to the end user in order to reduce latency, and thus delay, as much as possible.

- **Bandwidth:** measures the amount of data that is able to pass through a network at a given time. It is measured in bits per second.
- **Throughput:** the number of data packets that are able to reach their destination within a specific period of time. Throughput can be affected by a number of factors, including bus or network congestion, latency, packet loss/errors, & the protocol used.

Access to higher bandwidth data rates is always good, but latency is the limiting factor for everyday web browsing.

7.0.1 Deciding Sending Rate

Reliable transports don't just start sending data at full rate, because this could end up congesting the network. Each network link has only a certain amount of data it can process every second; sending excessive data will lead to packet loss. Lost packets need to be re-transmitted and can seriously affect performance in high latency networks.

7.0.2 Estimating Link Capacity

We don't know up front what the maximum bandwidth will be; it depends on a bottleneck somewhere in the end-to-end connection, but we cannot predict or know where this will be. The Internet also does not (yet) have mechanisms to signal link capacities back to the endpoints. Even if we knew the available physical bandwidth, that wouldn't mean we could use all of it ourselves — several users are typically active on a network concurrently, each of whom need a fair share of the available bandwidth.

7.1 Congestion Control

In the 1980s when the Internet was still ran by the government and TCP was young, engineers were learning about bad TCP behaviour & networks. At the time, TCP with no congestion control would occasionally "collapse" whole segments of the Internet. Clients were programmed to respect the capacity of the machine at the other end of the connection (flow control) but not the capacity of the network. In 1986, in an especially bad incident, backbone links under incredible congestion passed only 40bps, $\frac{1}{1000}^{\text{th}}$ of their rated 32kbps capacity.

In the network area, **congestion control** is how to decide how much data the connection can send into the network. Reliable transport protocols such as TCP & QUIC constantly try to discover the available bandwidth over time by using congestion control algorithms.

- **Maximum Segment Size (MSS)** is the largest data payload that a device will accept from a network connection.
- The **Congestion Window (CWND)** is a sender-side limit on the amount of data the sender can transmit into the network before receiving an acknowledgement (ACK).

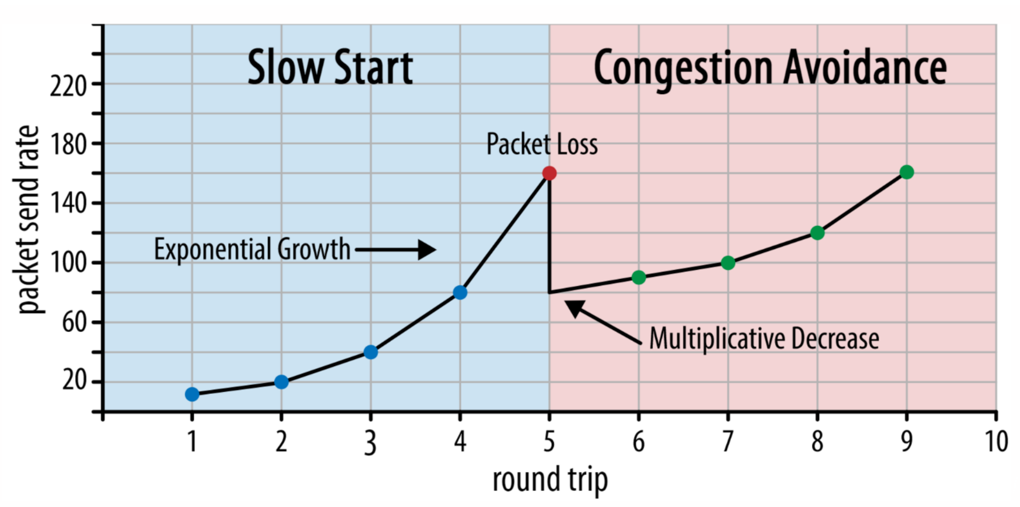

Figure 37: Congestion control

In the **slow-start phase**, a connection is started slow, and waits one round trip to receive acknowledgements of these packets. If they are acknowledges, this means that the network has capacity, and the send rate is increased every iteration (usually doubled). In the **congestion avoidance phase**, the send rate continues to grow until some packets are not acknowledged (which indicates packet loss & network congestion). On packet loss, the send rate is slashed, and afterward it is gradually increased in much smaller increments. This reduce-then-grow logic is repeated for every packet loss.

Largely, there are two categories of congestion control strategies:

- **Loss-based congestion control:** where the congestion control responds to a packet loss event, e.g., Reno & CUBIC.
- **Delay-based congestion control:** the algorithm tries to find a balance between the bandwidth & RTT increase and tune the packet send rate, e.g., Vegas & BBR.

7.1.1 Reno

Reno (often referred to as NewReno) is a standard congestion control for TCP & QUIC. Reno starts from "slow start" mode which increases the CWND (limits the amount of data a TCP connection can send) roughly 2× for every RTT until the congestion is detected. When packet loss is detected, it enters into the "recovery" mode until the packet loss is recovered. When it exits from recovery (no lost ranges) and the CWND is greater than the **SSTHRESH** (Slow Start Threshold), it enters into the "congestion avoidance" mode where the CWND grows slowly (roughly a full packet per RTT) and tries to converge on a stable CWND. A "sawtooth" pattern is observed when the CWND is graphed over time.

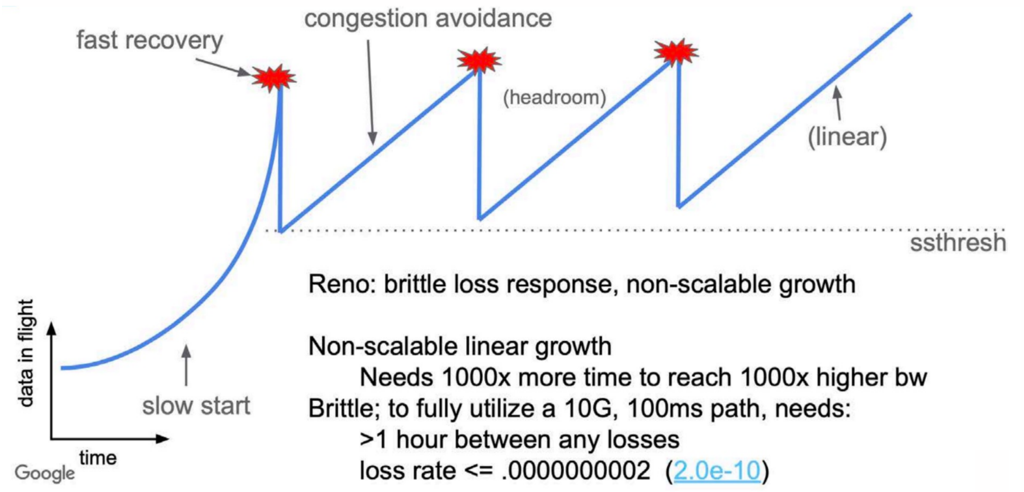

Figure 38: Reno

7.1.2 Cubic

Cubic is defined in RCF-8312 and implemented in many operating systems including Linux, BSD, & Windows. The difference between Cubic & Reno is that during congestion avoidance, the CWND growth is based on a textitcubic function instead of a linear function.

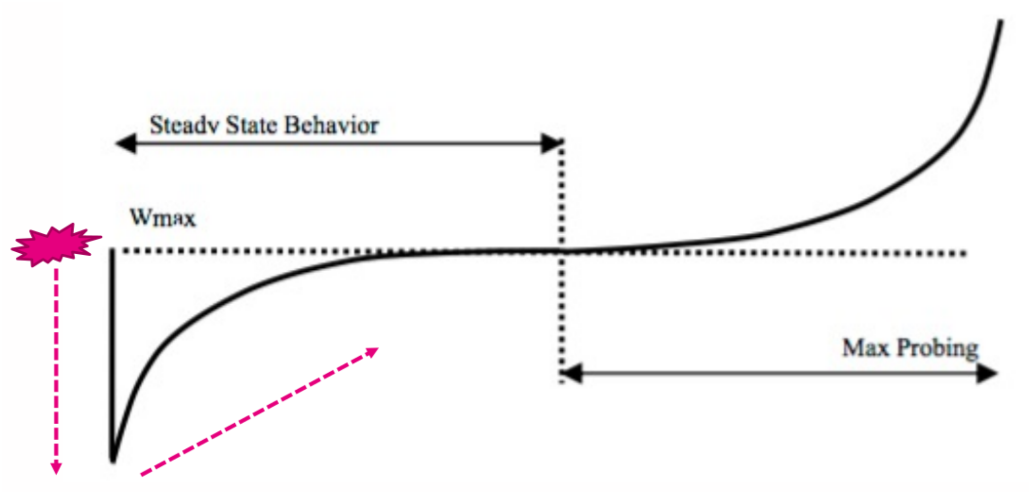

Figure 39: Cubic

W_{max} is the value of the CWND when congestion is detected. The CWND will then be reduced by 30% and will start to grow again using a cubic function as in the graph, approaching W_{max} aggressively in the beginning but slowly converging to W_{max} later. This makes sure the CWND growth approaches the previous point carefully, and once we pass W_{max}, it starts to grow aggressively again after some time to find a new CWND, called **max probing**.