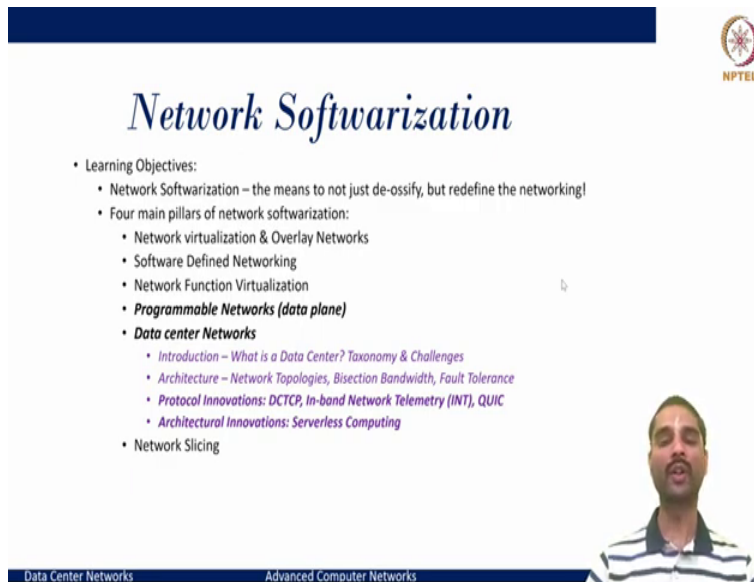



Advanced Computer Networks
Instructor Doctor Sameer Kulkarni
Department of Computer Science Engineering
Indian Institute of Technology, Gandhinagar
Lecture 59
QUIC


(Refer Slide Time: 00:17)





Network Softwarization

- Learning Objectives:
 - Network Softwarization – the means to not just de-ossify, but redefine the networking!
 - Four main pillars of network softwarization:
 - Network virtualization & Overlay Networks
 - Software Defined Networking
 - Network Function Virtualization
 - **Programmable Networks (data plane)**
 - **Data center Networks**
 - *Introduction – What is a Data Center? Taxonomy & Challenges*
 - *Architecture – Network Topologies, Bisection Bandwidth, Fault Tolerance*
 - *Protocol Innovations: DCTCP, In-band Network Telemetry (INT), QUIC*
 - *Architectural Innovations: Serverless Computing*
 - Network Slicing



Data Center Networks Advanced Computer Networks

Today, we will look into another of the protocol innovations that came into light because of the internet or HTTP workloads called QUIC, QUIC also pronounced QUIC, is a general purpose Transport Layer Network protocol that was initially designed by Jim Roskind at Google. And it started to take shape and got implemented around 2012 and deployed in the early versions of the Chrome.

And as it grew, the experimentation broadened. And many of the web browsers, including the MS Edge started to adopt it. And now, we see it supported by almost all of the browsers that we use today. So, it is important to understand a bit of background on why there was a need for a newer protocol called QUIC. And what are the key things that it tries to address, at least in brief?

(Refer Slide Time: 01:08)

EARLY INTERNET AND THE EVOLUTION: TWO MAIN TRANSPORT LAYERS

- Transmission Control Protocol (TCP)
 - Provides support for a **stream of bytes** abstraction
 - A Reliable transport over unreliable (best-effort) internetwork
- User Datagram Protocol (UDP)
 - Abstraction of **independent messages** between endpoints
 - Just provides demultiplexing and error detection
 - Low overhead, good for query/response and multimedia

The diagram is an hourglass shape representing the layers of the network stack. From top to bottom, the layers are: Web, VoIP, P2P; Email, RTSP; TCP, UDP; ICMP; IP; Ethernet; Sonet, ATM; Cat5, Wi-Fi, 3G; and Coax, Fibre.

Data Center Networks
Advanced Computer Networks

So, to understand that we need to know like what was the early internet like and what were the key protocols that existed. And when we look at the transport layer, it is primarily the two of the key protocols that shared the majority of the traffic. And when there was an analysis done in the early 2000s, about what was the network traffic look like.

And it was observed that more than 90% of the traffic was TCP, while rest of the 10% was shared amongst UDP and other kinds of protocols. So, TCP being a prominent chunk in the internet at the transport layer. And the popularity was so because it tried to provide reliable communication over the best-effort internetwork or the IP layer. And it essentially built the congestion control and flow control, which were quintessential to ensure that multiple of the end users can reliably connect and access the services over the internet.

And the other variant that was prominently used for majority of the services was the UDP, which basically was meant for very short, independent message kind of communications where you do not care as much about the loss, but you would be happy to retry and get the service done. So, wherever you saw the potentials for low overhead, and pattern of query response, would directly gel in this kind of a pattern.

And that is where we saw the use of DNS, DHCP that tried to adapt UDP. While most of the Internet services were packet loss, and congestion were a common phenomena they adopted for TCP to ensure reliable data transfer. And what eventually happened was this ossification of the

protocol around a transport layer where all the applications that we can think of be it the web services email or have them try to rely on one or the other variants of TCP.

And this is where another way to look at it was a protocol ossification that happened around TCP. And it was not that TCP was the best fit. But it was amongst all the services that you could rely on for reliable transmissions, it served the key purposes.

(Refer Slide Time: 03:11)

TRANSMISSION CONTROL PROTOCOL (TCP)

- Multiplexing/demultiplexing
 - Determine which conversation a given packet belongs to
 - All transports need to do this
- Reliability and flow control
 - Ensure that data sent is delivered to the receiver application
 - Ensure that receiver buffer doesn't overflow
- Ordered delivery
 - Ensure bits pushed by sender arrive at receiver app in order
- Congestion control
 - Ensure that data sent doesn't overwhelm network resources

Data Center Networks Advanced Computer Networks

Hence, if we see TCP, the key characteristics that it provided was to ensure that it would provide for multiplexing and demultiplexing of the services. That means at the end hosts, we could run multiple of TCP connections and ensure that different applications use the same TCP stack but with different port numbers so that you are able to get the services done for different applications.

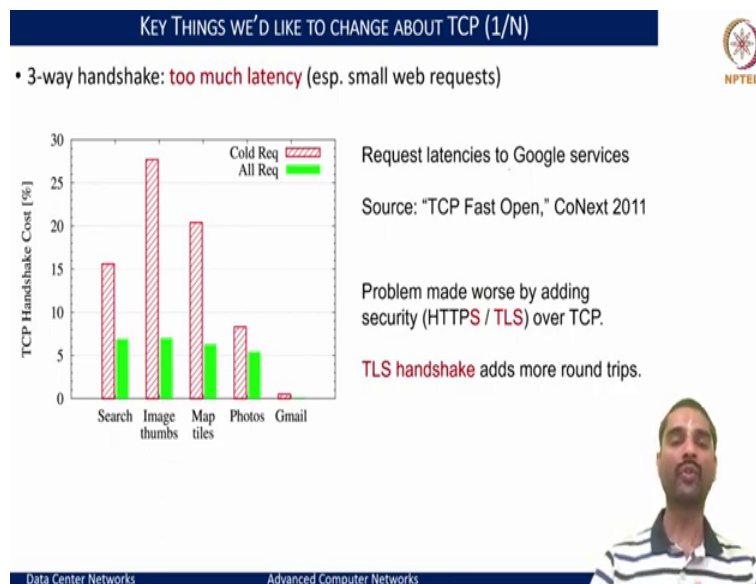
So, I could run the mail service, I could run the web browser, I could run the FTP service, all of those could underlying use the same transport stack, and you would essentially multiplex and demultiplex the streams when you process it through the IP and get it to the other end. And most importantly, as we see that IP layer is best-effort service wherein there is no guarantee that the packets that you transmit would eventually make it to the other end. So, you need at the controls of reliability, how would you ensure that the other end is existing, listening for the connection and that is where the 3-way handshake protocol of TCP played a major role.

And then you may have diversity in terms of devices, which may have different rates at which they would serve. So, you will need a flow control that ensures that the data that is delivered does not overflow the receiver and ensure that both are in sync with respect to the data exchange without trying to lots unnecessarily over-transmit and full the buffers.

And also the ordering of the delivery which ensures that the stream of bytes are correctly received at the receiver regardless of whatever the packet loss happened then receiver is able to get back the ordering which the packets are intended to work; made it all essential for TCP to be serving for majority of the applications.

And likewise, in a network, when we speak of conditions that can happen sporadically, you also wanted the mechanism to react to those conditions which were incorporated in the TCP. And this made a very strong case as for why TCP dawned majority of the Internet services to be the underlying transport protocol.

(Refer Slide Time: 05:07)



However, at the TCP, it has its own set of challenges. So, if we think of when we are transacting for a very small connection, and we see that in order to make the transaction, we need the TCP handshake to go through that means the 3-way handshake of the TCP that is SYN, SYN-ACK, and ACK packet exchange between the sender and the receiver is a mandatory precursor to ensure that we can even start before we start exchanging the data.

And this in many times, was a major issue because if you see that I am trying to exchange information on a one-gigabit link for just half around few kilobytes, then I will be done in few milliseconds. But this TCP handshake through which the service that I am trying to connect is around 10 to 20 milliseconds apart, then I am spending at least almost one full round trip to get this initiation done.

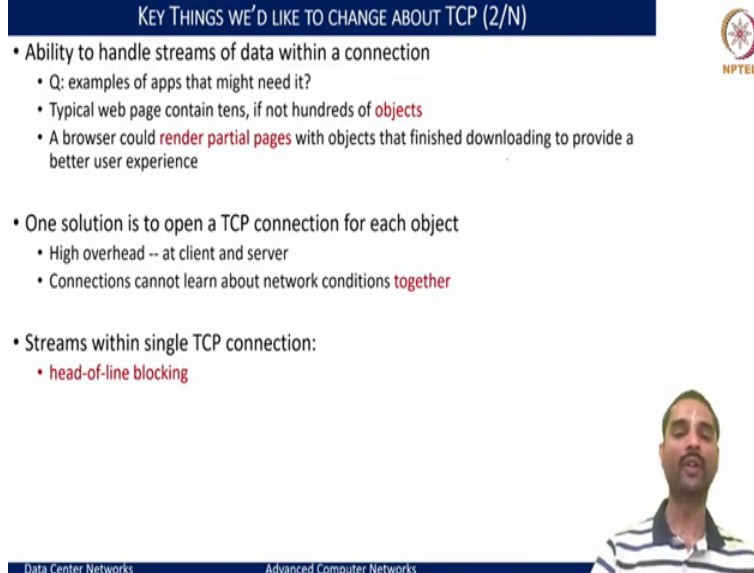
So, I am adding a lot more penalty in getting the service done. While the UDP on the other hand, we saw that it would just transmit the packets without caring whether the service is up or not. So, you will see that you do not have any overhead in handshake, but you do not guarantee that the service on the other side is there to even process your request. That was the other side of the challenge.

So, there were many works that came in the light of how to address this 3-way handshake approach. And one of the nice works is in CoNext 2011, that was TCP Fast Open. And it tried to analyze and say what is the real overhead that we really see with TCP. And it was shown in this plot here saying you are doing a simple Search kind of a query, you are paying around 15 to 20 percent of the overheads that are coming for the initial handshake while the Image Thumbs that are going to be put or the Map Titles that you are getting your photos, in most of the requests the code request includes the 3-way handshake overhead while the other requests where you avoid the handshake overhead, you will see that it reduces by more than 60 - 70 percent overheads. And this meant that whenever you are transacting for small queries over internet, you are paying a very high penalty.

Besides when we add the overheads that are necessary for ensuring that you have an encrypted channel that is built by the use of TLS, you need additional handshakes to happen between the client and server to negotiate the cryptographic parameters and exchange of the certificate so that the two parties can trust and negotiate the keys through which they can send the encrypted data.

And this further worsened the overheads that you would see that would add up or pile up for having this connection setup. And with TLS handshake, we are typically talking about almost 2 to 3 additional RTTs before we even could send or make a request. Hence, this was some way to be looked at to see how we can minimize these overheads.


(Refer Slide Time: 07:53)



KEY THINGS WE'D LIKE TO CHANGE ABOUT TCP (2/N)

- Ability to handle streams of data within a connection
 - Q: examples of apps that might need it?
 - Typical web page contain tens, if not hundreds of **objects**
 - A browser could **render partial pages** with objects that finished downloading to provide a better user experience
- One solution is to open a TCP connection for each object
 - High overhead -- at client and server
 - Connections cannot learn about network conditions **together**
- Streams within single TCP connection:
 - **head-of-line blocking**

Data Center Networks Advanced Computer Networks



And the other aspect to think of when we see of TCP is if I am currently connecting to any of the websites, typically we expect multiple of the different kinds of data that are being glued would be downloaded even if I am going for a simple web page, you have the HTTP connection, HTTPS connection. But that would essentially require you to get the HTML page, the CSS styling page that is associated scripts, and then any of the image contents or data that you would additionally have.

So typical web connection would end up having multiple of the kinds of data that you would have them downloaded before you render such a page. So, there were essentially the aspects of how would you handle downloading of multiple of these data, whether to make independent TCP connections, or have it as one TCP connection on which the entire data is going to be transacted.

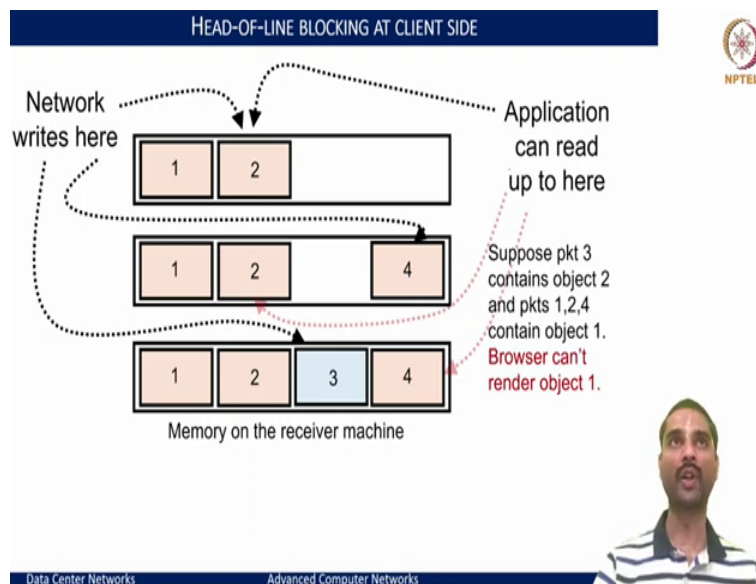
And this was studied much nearly HTTP 2 times. And it was that okay, every data that you want is very fluid. And you want them to be treated as streams of data that could be either independent connections or could be served on the same connection, like in HTTP 1.2 onwards, and you will see an HTTP 2, you could basically have a stream of connections that could be run over different TCP connections.

But if you open multiple of the TCP connections, you would end up having a lot more overhead at the client as well as the server because now server and client have to maintain the state for all

of these connections. And that would not may not be as much a problem on the client. But on the server, which is going to serve many of the other clients. This having multiple connections per client could become a bottleneck.

And hence it was also thought about to say how you can basically multiplex a same TCP connection to carry different streams of data that was streams within a single TCP connection. And this was, in essence to say that I could have a stream of bytes that would say, what is the CSS content or styler page or I could have this other stream that would say what is the HTML content and start to get them downloaded within a single TCP connection. Rather than having a connection per object, you could basically have a single connection multiplexed for multiple of the objects that you would want to download as streams. But this introduced another critical aspect of what we call as head-of-the-line blocking. This, again, is a characteristic of TCP that we need to understand what this is and where the problems could lie.

(Refer Slide Time: 10:25)



So, if we try to look into the head of the line blocking, let us consider just a very simple page that has two different kinds of objects that we want to download. And from the server side, it may be able to put the contents of one type and put the contents of other type of the object both in a same TCP connection. And what that means is network may write first the packet 1 and packet 2 which belong to a particular object, and then may try to write other data.

But when it writes packet 1 and packet 2, on the application side of the client, you will basically be able to receive and read up till packet 1 and packet 2, because this is a sequence of stream of bytes that make up the TCP packet to be read. But now, if suddenly that the network wrote packet 4, while skipping the packet 3, but consider packet 1 2 4 all belong to the same object.

At this point, because TCP is byte ordered reception, it would not be able to read beyond packet 2 because it requires packet 3 to even start to look at packet 4. And this is what we term as head of the line blocking. This is a genuine case if we need or if we have a scenario where packet 3 also belongs to the same object.

But what if the packet 3 belongs to another kind of an object, while packet 1, 2 and 4 are actually for one kind of an object and they are in sequence. Now, TCP cannot distinguish between the two and it would just block even if packet 4 were the next sequence that you need, after packet 2 for getting or fetching the object 1.

So, what this essentially means is, we are blocked now because of the head packet that is packet 3 not being there, to make the sequence complete. And what this would mean from the application point of view is that browser is not able to render the object 1, which consists of packet 1, 2 and 4, which had already received but because of the missing packet 3, which was for another object, but it is still not able to render the object 1.

And when we see nowadays, with the websites, typically we are having around 15 to 20 different kinds of objects that are going to be delivered. And if all of these are going to be done in a TCP sequence, and if any one of them in between changes or have a loss or we are not able to have a sequence, we are blocked for majority of the contents, even though we might have the full content that we may have for a given object.

And this is a major problem that was to be worked out and see what options or what mechanisms could be build. And we need to think now in terms of what are all the ways that this could have been addressed. And several works stemed like we said about the TCP Fast Open in 2011, that tried to focus primarily on how you can abridge the RTT connection or the round trip time.

And likewise, the attempts were made, especially from the web community in terms of what other aspects could be worked out. And this is where basically the emergence of QUIC started.

(Refer Slide Time: 13:37)

WHAT'S SO HARD ABOUT CHANGING TCP? PROTOCOL OSSIFICATION!!

- You may need to change the operating system kernel
- You may need to change the operating system kernel on all servers and clients (ex: all your laptops and phones!)
- You may need to change the entire network!
- Middleboxes sitting in the network may change, drop, or add info on packets
- Middleboxes may drop packets if they don't understand something on the packet

Data Center Networks Advanced Computer Networks

The slide features a dark blue header with white text. Below the header is a bulleted list of five points. The word 'Middleboxes' is highlighted in red in the third and fourth points. To the right of the text is the NPTEL logo. At the bottom of the slide, there is a video inset showing a man in a striped shirt speaking, and a footer with the text 'Data Center Networks' and 'Advanced Computer Networks'.

And if not like what were all the other options that we had were also the considerations to think. And here you may see that if we have to change or tweak anything with TCP, you have to change the operating system kernel because the entire network stack is where the kernel hosts this TCP stack. And you would have to change this kernel.

And that also means then all the end hosts. And you have lots of diversity on operating systems, kernels that reside on both the server side and client side, including variety of form factors, including the laptops, phones, all had to be changed. And that is overwhelming task. And if anything in the TCP headers were to be changed nowadays, we said the network is full of middleboxes.

And if the middleboxes are not able to understand the packets at the transport layer, if they are operating at layer 4, there is no way that the packets would eventually run to their destiny. Hence the updates on the middleboxes, also to ensure that any changes on such protocols at the network stack in the TCP layer at the transport means you have to also upgrade all of these essential middlebox devices.

And if the middleboxes, by default rule, when especially when your security primary is to say that if you are not able to recognize the kind of a packet, you simply drop it. What that meant is you run a situation where any tampering with the transport layer headers could essentially mean that packets get dropped silently somewhere in the network.

Hence, a better alternative to say how we could work around and bring the concept to weave within the realms of the TCP and UDP to facilitate or overcome these challenges were thought about.

(Refer Slide Time: 15:19)

The slide is titled "QUIC" in a dark blue header. Below the title, there are three main bullet points, each with sub-bullets. The first bullet point is "Designed over UDP with fresh packet formats at app layer", with a sub-bullet "Initially QUIC – an acronym for *Quick UDP Internet Connection*". The second bullet point is "Issue #1: Better handshake procedure", with sub-bullets: "Designed with security & encryption in mind from the beginning", "Almost everything is encrypted", "If middlebox can't read a piece of info, it can't make any decisions based on that info", and "In particular, can't change packet without endpoints noticing". The third bullet point is "Issue #2: Support lightweight streams natively", with sub-bullets: "Avoid Head-of-the-line Blocking." and "Enable better priority management for streams within a connection." To the right of the text is the NPTEL logo. At the bottom right, there is a small video inset of a man with a beard and glasses, wearing a striped shirt, speaking. At the bottom of the slide, there is a dark blue footer with the text "Data Center Networks" and "Advanced Computer Networks".

And now, what the engineers at Google tried to realize is to say that why not just rely on UDP for the internet connections so that you get rid of the 3-way handshake, but try to build a packet format on top of UDP at the application layer.

So, what it meant is, like the key characteristics that TCP handles in terms of flow control, condition control, and reliable byte stream in an ordered fashion of packet delivery, all of this if it can be pushed to the application layer, and keep the transport layer as simple as what the UDP does just that is to exchange your message between the two ends.

And this was the start of what we call as a QUIC or which initially stood for an acronym QUIC UDP internet connections. And this work started in, like I said, around 2012 and 2013. And

Google Chrome was the first to start this kind of service. And what it really tried to address are the two critical issues that we said, and there are many more other issues. Because of the other constraints, we are limited to just these two issues to understand QUIC in this prospect. So, first, was to address the handshake problem. So, if handshakes meant that there is a round trip time overhead, there is a means that we want to cut down on that. And this means were supposed to be defined as a part of a newer protocol. And we know that UDP has no such overhead.

So, if we try to bring that on top of it, and use the application layer, we would try to build the intelligence to say how we can build the reliability over the UDP in one way, and how we can build the order delivery of packets over UDP. And the third part is how we could also build multiple streams of data over a stream UDP connection.

And all of this in a way that because UDP is an established network protocol, and middleboxes would treat them as just the typical UDP connections, you will not have any issues or concerns of packet drops, that will happen either at the middleboxes in anywhere in the network. So, ticking all of these aspects.

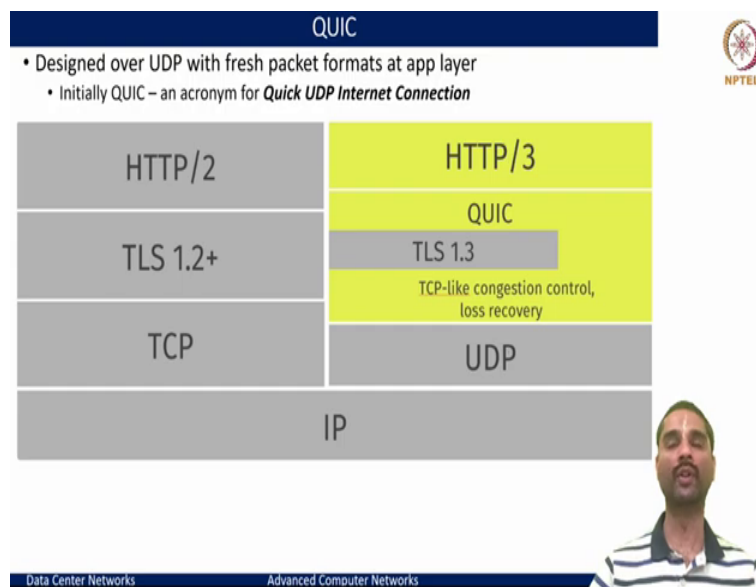
Now the question, in essence, was how to build the handshake procedure so that you are able to ensure that you are able to negotiate and ensure the client-server side of the window and parameters in terms of what data rate that you would want to send. And the second issue, like we just said, if there is a head of line blocking because of the ordering of the packets and byte streams within the TCP connection, how this can be basically decoupled so that you are able to ensure that ordering of the bytes for a particular object is maintained, but not for the entire connection, wherein if my connection is serving to send different objects, I want to ensure just the ordering of bytes for each of those objects, not necessarily that when object 2 is sent, I want to look for any information that is tied with object 1 or object 3, likewise.

So, we also want it to have better priority management in terms of if I have the streams. And if I have like different kinds of data, which data I want to render first so that I could prioritize sending that information over the same UDP channel to ensure that you get the data correctly in the order that you would want the browser to render.

So, all of these were the key design characteristics that led to the development of QUIC. And one more important aspect that we also want to bring with respect to QUIC is this primarily was thought in the light of the HTTP workloads where you had a browser as a client side and you are typically interacting or transacting with HTTP server on the other side.

And whenever we had this connections, we always thought of having a secure connection. And that is where the TLS also came in to provide the secure or encrypted connections.

(Refer Slide Time: 19:19)



And now, if we saw the earlier stack, what we will see is you have the TLS on top of TCP on the left-hand side here. And if we now want to support the newer framework or a newer protocol, we also had to ensure that we support the TLS. But this when we see the TLS has a handshake, which is done only after the TCP handshake is done. And this TLS handshake in a sense ensures that there is a server that is listening for your connection and you could establish a connection.

So, you can see that the TCP initial handshake, in essence, becomes redundant, if we are able to ensure the same with a TLS handshake. And likewise, the only things that now we need to consider if we do this a TLS handshake is what are the initial negotiation parameters that TCP exchanges need to be done alongside with the cryptographic handshake.

And this is where QUIC tried to incorporate both the TLS part of what handshake needs to be done and the transport side of the handshake to negotiate the parameters on the congestion control, the features that you would want, how do you want to ensure loss recovery all of these parameters together with the encryption based handshakes that you would want to do.

Hence, the way the QUIC stack was developed was to replace the transport layer TCP with UDP and replace the earlier versions of TLS with a newer version of TLS 1.3, which also supported for less RTTs to negotiate the cryptographic parameters. In fact, 0 RTT in terms of exchange of the secure keys if you already have the pre-shared keys you could directly use, and these made TLS 1.3, a better fit for QUIC.

And what QUIC, readily tried to absorb was to use the TLS 1.3 as the encryption model and bend the application layer protocol around TLS 1.3 and build it on the UDP transport and the services that it provides again, leads to the other application layer on the above that is called the HTTP 3.

And in May 2021, all of these efforts got standardized, and we have a series of RFCs starting from 8999 to almost 9012 or 9013, detailing about how the QUIC operation should be and what are the ways that QUIC will work with HTTP 3. And this also meant there was a need to change the ecosystem around it.

That means if I am trying to go check and get ready DNS parameters for a given connection, I should be able to get the DNS connection whether the server supports HTTP 3, that is QUIC. And if it supports then client and server could ensure that they can communicate directly over QUIC. And that also meant that there is a need for a DNS change.

And there were all services that were being updated to say that whether a server supports HTTP 3 or not. And if not you will fall back to using HTTP 2 and use the standard TCP, TLS model that you are seeing here on the left-hand side. And this is how the QUIC brought in significant change in terms of how we could ensure the connections to come up

(Refer Slide Time: 22:31)

#1 LOWER-LATENCY HANDSHAKE



- "Common case" 0-RTT handshake
- Otherwise, 1-RTT or (more rarely) 2-RTT handshake
- Merge cryptographic information within the first packet
- Use cached credentials to make subsequent handshakes faster
 - Optimistically assume handshake succeeds
 - Server can reject or drop request without doing too much work

Source: [APNIC](#) and [Cloudflare](#)



Data Center Networks Advanced Computer Networks

And what it essentially tried to address in case of when we consider the first issue of the TCP handshake and the latency overheads, it tried to bring what we call as the most common case have a 0 RTT handshake. And that meant you had no overheads to pay. And you could start exchanging the information right away without any overheads.

And if that was in special cases, that if it were not to be possible, then it made possible of 1 RTT handshake where you will have to establish the connection with 1 RTT and then start exchanging the parameters. And this is much smaller than typical 3 RTTs that you have with the TCP/TLS stack.

And only in the cases very rarely, where there is no support for the QUIC, and if client had started with a QUIC, you would have to fall back to the TCP connection. That meant you would go up to 2 RTT handshakes, when you know that there is no support for QUIC, and this is what the most common case that we see with that a traditional TCP, and it is still the same.

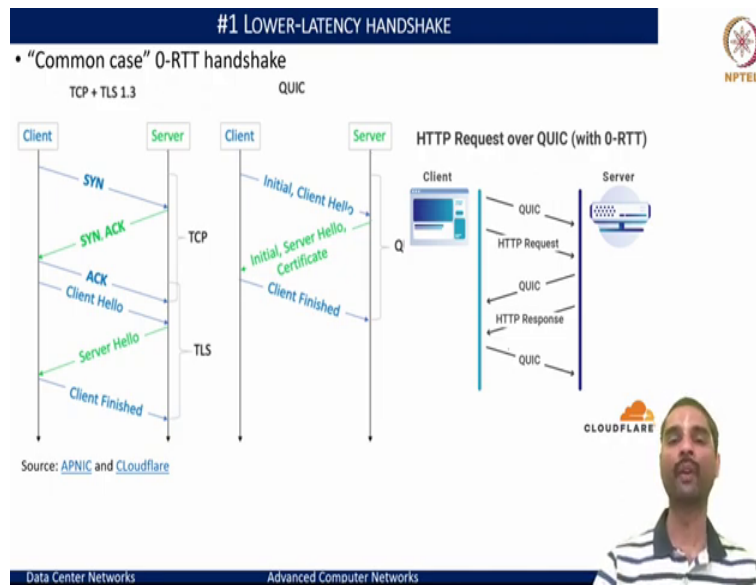
And what other important aspect the QUIC also tried to address it trying to do it is because now you are trying to merge the cryptographic information within the first packet, you avoided the overheads of additional RTTs to negotiate the cryptographic parameters, especially the symmetric key that you would want to use for encrypting the session details.

And with the TLS 1.3, it also facilitated to say that if I have a cached credential, I could directly start to assume that the server would work with the credentials that I earlier shared in one of the

connections and start using it. Because most often, we would have the connection parameters we are connecting to a particular site, we repeatedly connect over time.

And if I can ensure that the keys are recycled over a large duration, let us say in an hour or so then you could reuse the same keys to establish subsequent connections. And this is where the low latency handshake parts were realized, through QUIC.

(Refer Slide Time: 24:30)



So, to quickly look up what it meant in the TCP and TLS 1.3 version. First, you would have a SYN, SYN-ACK, and ACK, basically the TCP 3-way handshake. And once TCP handshake was done, you would have the TCP connection parameters that were exchanged.

And then the TLS connection handshake would start with a typical Client Hello, Server Hello, and followed up by the exchange of the server certificate, which the client would authenticate, and client would verify that he is connecting to the right server and then start the client side of the TLS negotiation and eventually derive key through which the entire session could be encrypted and then start sending the packet request.

And this is where we can see that it at least involves 2 RTTs and typically 3 RTTs to ensure that you have the TLS setup and TCP setup before we start and send the application data. But with QUIC, because it operates at the application layer, and relies on the UDP transport, we could

merge basically the Client Hello with the initial parameters for a QUIC communication to exchange the application layer of the transport characteristics that we want at this point.

And what this meant is you typically need 1 RTT to exchange the Client Hello, Server Hello, all the certificate negotiation wherein encryption is done right from the end of the client finished. So, you would have the entire encrypted session that is starting right after the client finished, which is where we can see that this is just 1 RTT overhead.

And if we already have this connection that were being sent, or these certificates that were being exchanged with the server, you would have those cached within the client side. And for the next subsequent connection, you could reuse the same and start directly with the HTTP request using the same earlier cached with parameters in just communicating that we are trying to establish a connection with these parameters with the server and if all sessions parameters seem fine, you could essentially have the server respond back and this is where the 0 RTT overhead communications happen with QUIC.

(Refer Slide Time: 26:40)


#2: STREAMS WITHOUT HOLB

- QUIC supports packet format with frame-level data

```
-----  
| Flags (8) | Connection ID (64) (optional) | ->  
-----  
-----  
| Version (32) (client-only, optional) | Diversification Nonce (256) | ->  
-----  
-----  
| Packet Number (8 - 48) |  
-----  
-----  
| Frame 1 | Frame 2 | ... | Frame N |  
-----  
-----  
Stream frame  
-----  
| Type (8) | Stream ID (8 - 32) | Offset (0 - 64) |  
-----  
-----  
| Data length (0 or 16) | Stream Data (data length) |  
-----
```

The diagram illustrates the structure of a QUIC packet. It starts with a header containing flags and an optional connection ID. This is followed by a version field and a diversification nonce. The packet number is highlighted with a red circle. The packet contains multiple frames, with the first frame being a stream frame. The stream frame header includes the type, stream ID, and offset, with the stream ID field also highlighted with a red circle. The stream frame contains data of a specific length.

Data Center Networks Advanced Computer Networks



And the second important aspect that QUIC also tried to address is the head of the line blocking. And if we consider the reason why that happened with TCP is independent of the kind of an object that you transmit, every information that you pass in a TCP packet is sequenced as per the numbers that you would put that means every byte belong to just one stream or one stream of

byte that you would exchange for TCP. And these bytes could refer to different objects. And that was for the user and to say what objects that this byte meant for.

So, now if we decouple this information and say, if we build the protocol with the packet data structure, where we isolate the packet number and this packet number essentially refers to the stream of connections that are happening over the QUIC. And within this, I may support multiple of the frames. And what this means is I may send multiple of the packets 1 to n. And within each packet, I may have packet 1 correspond to one of the frame's data, packet 2 correspond to one of the frame's data and likewise, and within each of the frame, I want the sequence numbering that means the offset information that is highlighted here would correspond to the actual sequence of bytes that you would need for a kind of an object.

So, if I say frame 1, it belongs to a particular object, I would want to ensure that this offset information is this sequence of streams that I want to build within that frame 1, while for frame 2, I could have a different packet number immediate next packet number sent out packet frame 2. So, then it does not matter whether the packets were really in sync or not.

But what matters is whatever the packets that you receive, do they have the frames that are in sync or not? And this is a simple data structure that was being pulled to say that you isolate the streams from the packets that are going to be exchanged to basically decouple this information. And now we can ensure that applications look for and match on the offsets to build a stream of sequence bytes for a given frame. So, that way, we can multiplex now multiple of the objects within the same connection.

(Refer Slide Time: 28:48)

#2: STREAMS WITHOUT HOLB



- Stream-level flow control
 - Separate advertised window per stream and for connection
 - Window update frames per stream
- Stream-level sending rate mechanisms (“priorities”)
- QUIC receiver can deliver packets to app as long as stream packets received in order (even if connection’s packets are not)
- Ideas are known from prior work on “Structured Streaming”

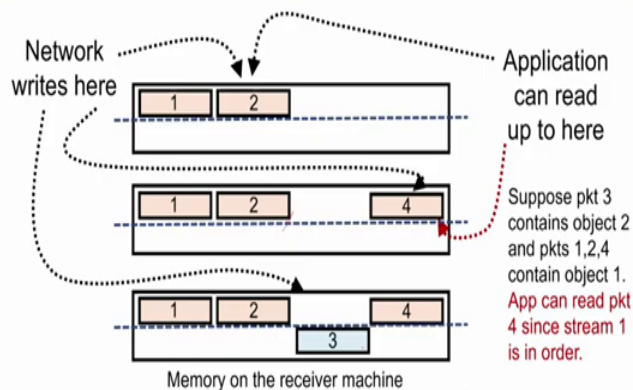


And this means that now, streams will not encounter any of the head of the line blocking as we have separate advertised window per stream for each of the connections because the number that we are seeing for the frame offset is the sequence of data that is for each stream. And also we could now say which frames have different priorities and embed the packets to ensure that these frames are going to be sent with different priorities.

And this structured streaming mechanism at the QUIC enables us to overcome head-of-the-line blocking.

(Refer Slide Time: 29:25)

#2 NO HOLB ACROSS STREAMS



So, let us try to revisit the same example that we spoke about earlier and see how this would operate in the view of QUIC. So, if we had the packet 1, packet 2 that were sent from the application, you could read out packet 1, packet 2, and you are going to read packet 1, packet 2 also as a contiguous bytes for stream 1, which refers to a particular orange object here to say that data is prepared.

And then when the packet 4 is sent, although its packet identifier is not in sequence, but within this packet, when you see the stream which belongs to the same stream as the orange object and the byte offset aligns with whatever the data that you had at this end, now, you are able to basically take the entire orange stream that is 1, 2 and 4 to view it as a single frame.

And this ensures now that the browser can basically work out the 1, 2, and 4 together and render the object as you receive the packet 4. And eventually, when the packet 3 is received, and you will see that this is for another object, it could also take and build the sequence of the stream. So, if that is contained object, it could also render.

So, essentially, now, this helps decouple of the stream sequence of bytes versus the packet sequence of bytes, ensuring that application can now be free of the head of the line blocking, utilize the same connection, you need not how to build multiple connections either because each UDP connection can now be viewed as different streams of connections that are operating in parallel. In essence, the streaming of the data is done per stream within the same QUIC or UDP connection.

(Refer Slide Time: 31:10)

LESSONS FROM QUIC

- Layering enables modularity but can hurt performance (TCP+TLS)



- Benchmark application metrics (macrobenchmarks) and protocol metrics (microbenchmarks) to ensure
 - That features are working correctly
 - That features are useful to applications
- User space networking has performance caveats
 - High CPU usage
 - Harder to write applications (PACKET_RX_RING + mmap)



Data Center Networks

Advanced Computer Networks

And, in fact, there are several other additional things that the QUIC brought in, in terms of saying how you could do better RTT management, how do you want to ensure when there is a packet loss, in TCP you have to discard retransmissions from accounting for RTT estimation, but with QUIC, it also meant that they came up with the mechanisms where you could try to say how you could still be able to do better RTT estimation even in the event of packet loss and it also introduced a stream bit, which was a very interesting aspect but debated heavily as well in terms of having just one bit to ensure that you can do the RTT estimations without having to do much of the computation overhead because at one point when I send a window, I can set a bit to 0 for one window, 1 RTT. And in the next RTT, I can spin the bit back from 1 to 0.

So, there were a lot of optimizations that were brought in from QUIC. And the deployment although it started in nearly 2014-15, now we see most of the internet traffic when we are especially transacting with any of the Google sites or any of the Chrome Website, it tries to use QUIC to ensure that you are able to get better and faster connection. And what we can learn from the QUIC in nutshell, is that layering is a good aspect in terms of modularity, but it can hurt performance because of some redundancies that creep in.

Because TCP, TLS both had to do the handshakes, but TLS stacked on top of TCP could not proceed until the TCP handshake is done. And this also meant now during the TLS handshake engendered the TCP handshake to be, in essence, redundant. And although there were some key

parameters that you will want to exchange for TCP, which could then be coupled with the cryptographic handshake.

And this is where the QUIC tries to leverage the two and tries to bring a handshake that essentially does both the aspects in one go. And it is also important to see when we layer and build, what are the side effects or the impacts that they bring in terms of the application metrics of latency throughput at a macro level, as well as a micro level.

Because if we had not done the metrics to see what is the overhead that is being added by TCP, how does it start in the current world because earlier the entire bandwidth, the connections were slow. But now if you are having one gigabit connection all the way up your home, you are able to transact a lot more data than what you would otherwise send in a slower rate.

And this is where the overheads of communications in terms of RTT tries to weigh in as a higher cost. And now we have to reconsider how we would want to use them or minimize these overheads. Nonetheless, when you used to try to do the UDP and brought in QUIC, it also has its own share of challenges, especially when we say QUIC is no more network stack that is coming with the kernel, but it is an application layer stack. That means, on one hand, you have given the flexibility to build variants of QUIC protocols at the application layer and then adapt it for different applications, two: the essence of trying to do things lot more, like if I have multiple connections that I open, I would be doing the same aspect of the congestion control, flow control in each of the process independently.

And this is where the overheads also start to creep in. And essentially, there will also be measurements that were done where it was shown that because QUIC uses UDP and UDP use less of a utilized service, the stack in the network is not optimized for UDP at all. And that meant you would spend a lot more overheads in processing the UDP packet. So, all of these essentially means that QUIC started to have a high CPU usage.

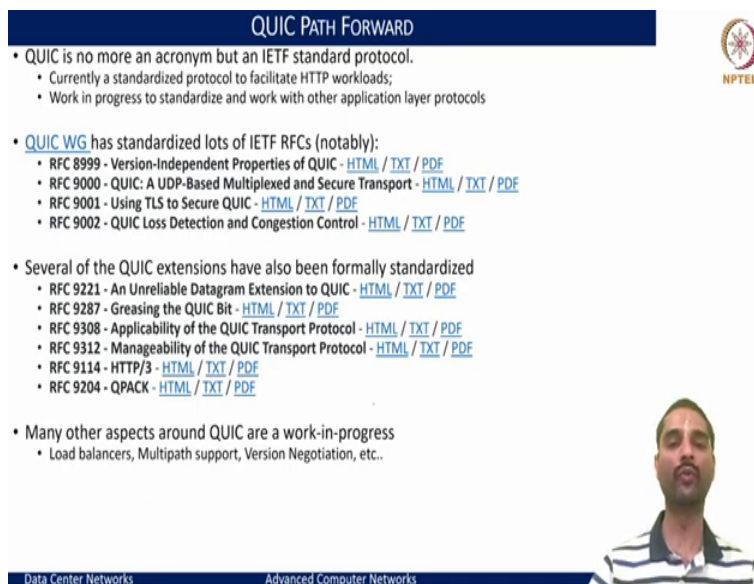
And also we spoke earlier about the offloads that were possible to the network card like if I want to do the encryption offload, checksum offload, all of those were supported by NIC, which were done by the TCP, and the kernel stack would then use the offloads of the hardware. But now with

QUIC, now you are in the user space. So, it becomes a lot more challenging to say how we can really utilize these offloads.

And this also means that now you are doing all of this computation on the CPU, which ends up having high CPU usage. So, if you have noticed, when you use Chrome, most often you will end up having Chrome utilize a large portion of the CPU, that is because the variant that underlying if it is having a lot more QUIC connections, it will start to see higher CPU usage.

And when you want the data to be written to the user space, there are also the optimizations that were brought in to see how you can directly access the data right at the application layer. So, we spoke earlier about the DPDK and Mem map mechanism. So, the same now hold fit for QUIC, and in essence, if you see the other side of the opportunity, we see that QUIC makes gluing this as a transport and application layer protocol for any of the applications that we want to build with our DPDK kind of a framework. It makes it very ready fit.

(Refer Slide Time: 36:14)



The slide is titled "QUIC PATH FORWARD" and features a list of RFCs and a video feed of a speaker. The list includes:

- QUIC is no more an acronym but an IETF standard protocol.
 - Currently a standardized protocol to facilitate HTTP workloads;
 - Work in progress to standardize and work with other application layer protocols
- QUIC WG has standardized lots of IETF RFCs (notably):
 - RFC 8999 - Version-Independent Properties of QUIC - [HTML](#) / [TXT](#) / [PDF](#)
 - RFC 9000 - QUIC: A UDP-Based Multiplexed and Secure Transport - [HTML](#) / [TXT](#) / [PDF](#)
 - RFC 9001 - Using TLS to Secure QUIC - [HTML](#) / [TXT](#) / [PDF](#)
 - RFC 9002 - QUIC Loss Detection and Congestion Control - [HTML](#) / [TXT](#) / [PDF](#)
- Several of the QUIC extensions have also been formally standardized
 - RFC 9221 - An Unreliable Datagram Extension to QUIC - [HTML](#) / [TXT](#) / [PDF](#)
 - RFC 9287 - Greasing the QUIC Bit - [HTML](#) / [TXT](#) / [PDF](#)
 - RFC 9308 - Applicability of the QUIC Transport Protocol - [HTML](#) / [TXT](#) / [PDF](#)
 - RFC 9312 - Manageability of the QUIC Transport Protocol - [HTML](#) / [TXT](#) / [PDF](#)
 - RFC 9114 - HTTP/3 - [HTML](#) / [TXT](#) / [PDF](#)
 - RFC 9204 - QPACK - [HTML](#) / [TXT](#) / [PDF](#)
- Many other aspects around QUIC are a work-in-progress
 - Load balancers, Multipath support, Version Negotiation, etc..

The slide also includes the NPTEL logo in the top right corner and a video feed of a speaker in the bottom right corner. The footer contains the text "Data Center Networks" and "Advanced Computer Networks".

And QUIC has evolved a lot more than what it started in the early 2015 to where we stand into 2023. And QUIC now is no more an acronym, but an IETF standard. And it was standardized in May of 2021 where there are series of RFCs put forth by the QUIC working Group resulted in like I said RFC 8999 for vendor-independent properties of QUIC, RFC 9000 which essentially describes a UDP-based multiplexed secure transport and how it is, what is the protocol aspects,

how it is going to be built, and 9001 detailing about the TLS to secure QUIC and why TLS 1.3 for QUIC and 9002 for QUIC loss detection and congestion control schemes. And several of the additional extensions were brought in terms of how the HTTP workloads would often use in terms of leveraging the header compressions like QPACK that you will use with for the HTTP and HTTP 3 in itself, a new variant that you will want to support for the faster HTTP connections.

So, all of these have been standardized in a very recent times. And QUIC has a very strong way forward also in terms of looking at the newer aspects that we want to build, especially like the load balancer, multi-path support, version negotiation, and many more aspects are currently still being discussed actively and are on the path to get standardized.

And we have tried to basically summarize QUIC in a very nutshell, but there is a lot more beyond what is being covered in this. I encourage you to go look at the RFC 9000 at the least to get a good understanding and grip on QUIC.