

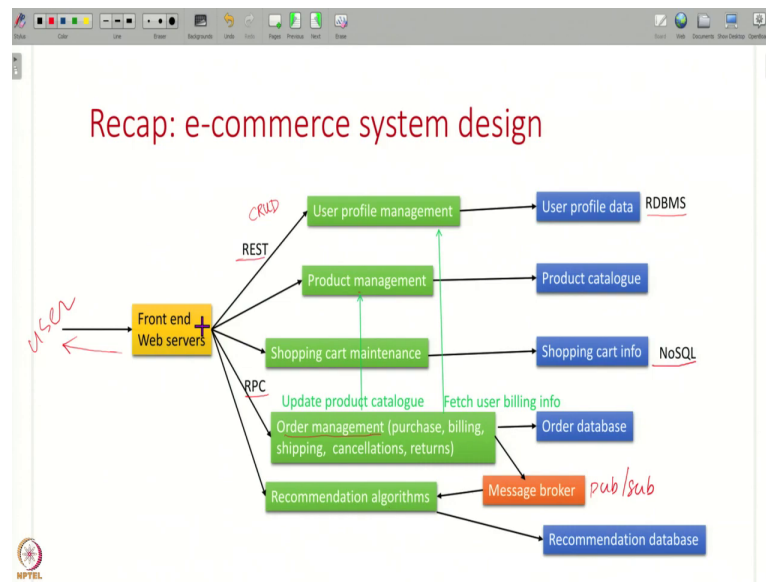
**Design and Engineering of Computer Systems**  
**Professor Mythili Vutukuru**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Bombay**  
**Lecture - 41**  
**Examples of End-to-End Systems Design**

Hello everyone, welcome to the 29th lecture in the course Design and Engineering of Computer Systems. So, in the previous lecture, we have started our discussion on end to end system design we have seen if you have a large computer system that has many different requirements it has to satisfy, then we have seen how you take those functional requirements and how you arrive at a modular design of the computer system, how you can split the computer system into multiple components.

And then we have also seen in each component when you have to store data, what are the various choices available to you, and if these components have to interact with each other then what are the various ways in which you can design these API's. So, we have seen all of these concepts, and we have used an e-commerce application as sort of a running example to explain all of these concepts.

In this lecture, what we are going to do is we are going to take many more examples, like e-commerce and many other real systems that you would have seen in your day to day life and try to do this entire process, work out an end to end design for many other systems, so, that the concepts of the previous lecture become clear. So, let us get started.

(Refer Slide Time: 01:31)



So, here is a recap of the e-commerce system design that we have briefly touched upon in the previous lecture. But I want to once again, recap it and show it in a more concrete format, so that you can fully understand the end to end picture. So, you might have multiple front-end web servers that are receiving traffic from users who want to interact with this e-commerce system.

And, if the, there are many other application servers for different functionalities inside the system, and based on what request the user has sent, the front end web server will redirect the request will talk to many other application servers. For example, if the user is creating an account, you might talk to the user profile management server.

If the user is searching for products, you might talk to the product management application server and so on. So, depending on what is the user's request, this front end will talk to many different components, assemble an HTTP response back and send it back to the user. Now, what are all the different application servers, you might have one to manage user data, one to manage product information, one to maintain your shopping cart, one to do order management the actual purchase, and one to give you recommendations.

And all of these application servers will be contacted by the front-end server in order to display the web page for the user. Now, the front end can be interacting with these various servers using either REST API's or RPC based API's. For example, if it is just managing user profile adding

users creating, reading, updating, deleting. The crud model that we have seen for rest API's, that simple interface may be enough to manage user information.

But when interacting with this order management server, the front end might need to do more complex actions like purchase, cancel orders. And such actions, it might be hard to do it using REST API. So, you might have a custom RPC interface defined here where this order server exposes a bunch of functions.

And this front end can remotely invoke those functions. So, these interactions between components can happen either by a REST or RPC. And then when it comes to the data stored by each component, the user profile information can be probably stored in a relational database because it has structured data and you need to store it consistently.

Whereas things like shopping cart of a user maybe use a NOSQL database that does not guarantee perfect consistency, but it gives you very high performance. Similarly, for other components, also the product catalogue the order database, you might have many different design options available for how you store the data.

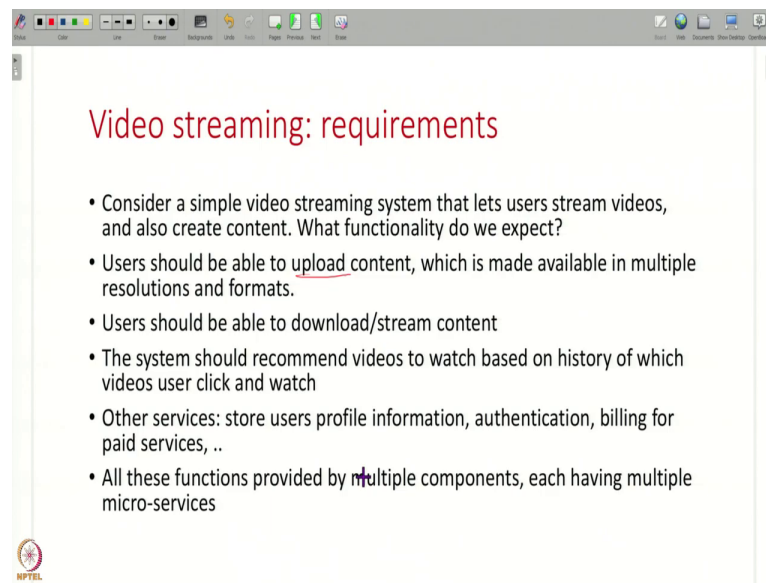
And note that the components can also interact amongst each other. For example, this order management component, when user purchases an item, it might have to talk to this product management component and say remove that item from your catalogue it is sold or this component may have to fetch user billing info from this other user profile component.

So, this front end interacts with the application servers and these application servers may also be interacting with each other, using various REST or RPC or such API's. Then the third type of API is the message broker or an asynchronous a Pub Sub API that we have seen. For example, whenever this order server gets some orders, it will just publish those orders into a message broker or a task queue, these are also called message queues.

There are many names for these message brokers. So, the orders that arrive can be just published into this message broker and other servers like servers that are running these recommendation algorithms can subscribe to those notifications. And whenever some order information comes in this recommendation guy will see that order information constructors his recommendations, saying oh, the other users like this user have purchased these other items. So, maybe I should recommend those items to the user and store all of that information in a database.

So, the next time the user logs in the front end will get this recommendation information and displayed to the user. So, this is sort of an example of how you do an end to end design of an e-commerce system. Of course, this is just one sample depending on you can make it simpler, more complex, add more functionality, and so on. And you can also modularize it in different way, you can use different kinds of interfaces, different databases, there are many options available, but this is one sample example that I am showing you.

(Refer Slide Time: 06:45)

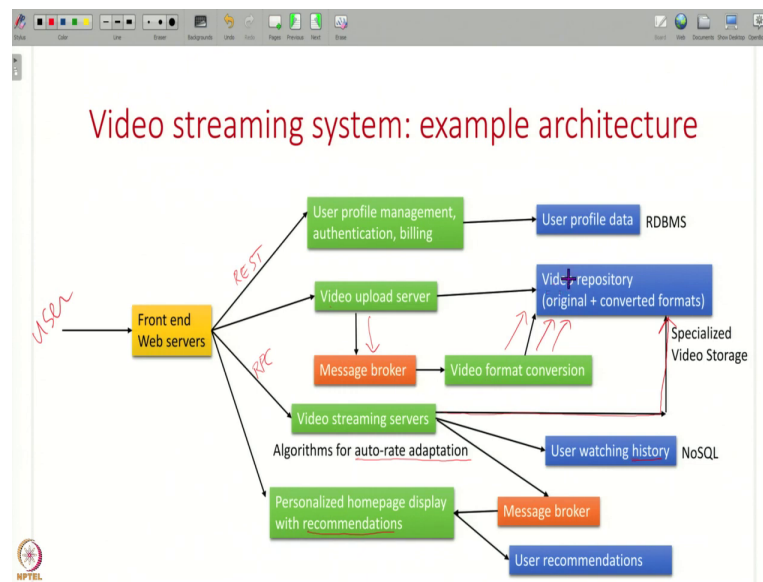


So, now, let us do the same exercise for various other kinds of systems also. For example, let us consider a video streaming system like the YouTube platform on which you might be watching these videos. So, if you have to design such a system, and how would we go about doing this. So, what functionality do we expect from such a system? First user should be able to upload content, whoever creates video should be able to upload content and this content should be available in multiple resolutions multiple formats to be later consumed by users later on.

And of course, some users upload content a large number of users also will be able to download or stream content, either download the entire video or stream it, which is not download the entire thing but get small parts of it and display. Then the system should also be able to recommend videos based on the history. For example, if you are watching some NPTEL courses on system design, YouTube might recommend some other videos related to system design to you that is a useful feature for the system to have.

And of course, you can have other services like users will create accounts log into accounts, they have some paid services billing, all of these other functionalities also you can expect. And all of these functionalities will be provided not just in one monolithic component, but will be split across multiple different components.

(Refer Slide Time: 08:02)



So, let us see an example of if you have to design such a system, how would you go about doing this. So, this is one example architecture that I have worked out. But you can also do different architectures. For example, users will as usual, you will interact with a front-end web page where all the information is displayed.

And to display the information on the webpage, the front-end server might contact many other application servers. For example, you can have an application server that is managing all the user profile information, authentication, billing, and storing all that information in some database. Then you can have a server that is handling video uploads, so whenever the user uploads a video, the front end will send that HTTP POST request or whatever, to a video upload server, which will store that video in some video repository.

Now, this could be a specialized database, it need not be a traditional relational database with tables and all it can be a specialized database that is optimized for video storage. And when a user uploads a video, this server can simply publish that information to a message broker and another component another server can read, can fetch these videos that have been published and

convert them into multiple formats and one video is published, multiple other videos will be uploaded into the video repository. Because you might want to have a high definition version, lower resolution version for users with low bandwidth, network bandwidth and so on.

So, this video upload server need not do all this job, it can just publish the video and in a message broker or a task queue and another component can do this format conversion later on asynchronously. So, if you have ever uploaded videos on YouTube, you will see that initial one version will be available quickly but other resolutions will take some time. They will YouTube will process them later on and upload them later.

Then, of course, when users want to watch videos, those requests will go to another server that probably specializes in streaming content, these servers, that stream videos will do some rate adaptation, they will send you some video to you. But if your network connection is slow, then they will, the next chunk of the video will be sent in lower resolution, they will automatically adapt the rate to match your network bandwidth.

If your TCP is giving you some throughput, then they will try to match that throughput. And if your throughput is high send you high resolution, if it is low, send your lower resolution. So, you have some specialized video streaming servers that do that. So, this front end might actually be fetching data to send in an HTTP response from this specialized video streaming server.

Then these servers will, of course, look up this video repository to fetch the video and stream it to the user. And whenever a user watches these videos at the servers, the servers will also maintain a history of all the videos watched by the user. So, this could be like a simple, this user this history, like it could be an unstructured, NOSQL key value store or something.

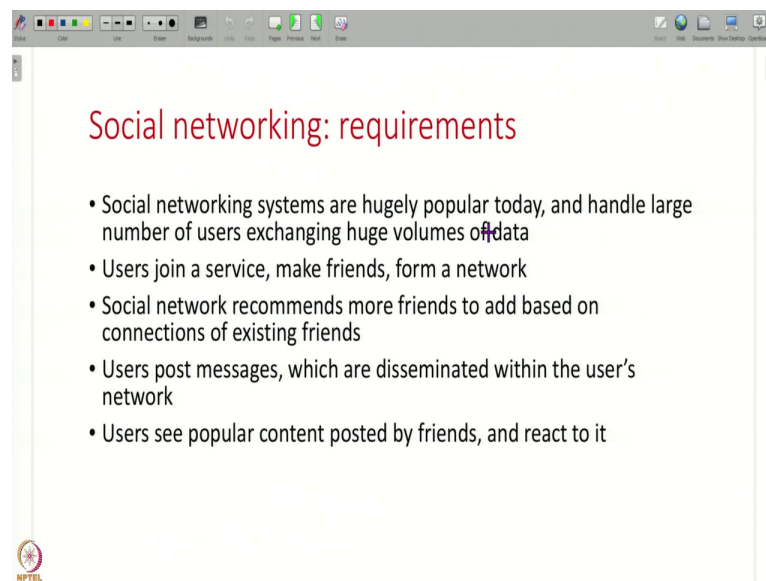
And of course, this information can also be published in a message queue or a task queue or a message broker for other components, the fact that you have watched some videos that will be published, and other components will try to run some analytics on it and try to recommend videos for you in the future.

In your personalized homepage, you might get some recommendations. So, these recommendation servers will subscribe to these notifications of what all activity a user is doing, and then run some analytics on it and then come up with a model saying this user likes to watch

this type of videos and store those recommendations, and give them out, give out a personalized homepage to the front end.

In this way, this is an example of how you will put together an end to end system. Where you can see that different components are doing different types of work. They are storing their data in different types of databases or data stores and they are communicating using different types of API's. It can either be, for example, this can be like a REST API, streaming videos is probably more complex functionality. It can be an RPC, uploading videos and converting into different format, these components interact via Pub Sub interface. You have different kinds of interfaces between different components.

(Refer Slide Time: 12:37)



### Social networking: requirements

- Social networking systems are hugely popular today, and handle large number of users exchanging huge volumes of data
- Users join a service, make friends, form a network
- Social network recommends more friends to add based on connections of existing friends
- Users post messages, which are disseminated within the user's network
- Users see popular content posted by friends, and react to it

So, now let us move on to another example. Suppose you have a social networking website. So, all of you would have interacted with social media in today's digital age, everybody uses social media. So, if you have a social networking system, what are the requirements? You have users, they will register, they will add some friends, they will form a network and the social media website also will recommend more friends to you if you add some 5 friends that will tell you oh, add these other people also, maybe you know them.

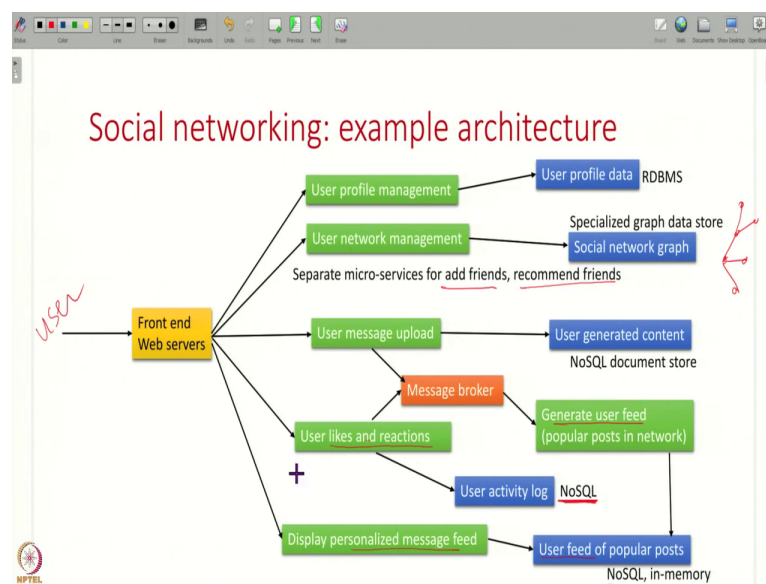
It will recommend more friends to you. Then once you have this network of friends, all the users will post messages and these messages are viewed by all the other users within the network. So,

when you post a message, it is disseminated to all the other users. And when you log into a social media website, you will see all the content posted by your friends.

And of course, there could be a lot of content. So, these social media websites will actually run some algorithms to decide which content to show you which content is more important, more popular, they will try to show that to you and you can also react to that content. If your friend posts any message, you can like it, you can comment on it, there are all of these reactions also.

So, this is an example of the requirements that we have from a social networking system. And these systems also handle large amounts of data later on in the course when we study about how to do performance, engineering and reliability engineering. We will also touch upon these topics, there are billions of users on some social media websites, and they process millions of requests per second. So, how do you guarantee that level of performance is something that we will study later in the course. For now, let us see how you put together an end to end architecture.

(Refer Slide Time: 14:24)



So, again, this is an example of an end to end architecture where the users are interacting with front end web server, and doing many different things. For example, you could have a component that is managing the user's profile storing all the user data in a database, you could have another component that is managing the user's network. Whenever the user adds friends or removes friends, all of those requests, go to this user network management component, and this component can provide multiple micro services or API's. For adding friends can be managed by

one micro service one process, recommending friends can be managed by another process within the same component and the users network information can be stored in a specialized graph database, not a regular traditional database, but something that is optimized to store a graph.

And then you run algorithms on this graph to recommend friends, if the user has these friends, this is the user's network so far, maybe I can recommend the friends of this user also, something like that, you can run processing algorithms on this graph and recommend more friends for the user. And these different functionalities of adding friends, recommending friends can be done by separate sub components within this main component. And now when a user posts a message, this message all the user content can be stored in some NOSQL document store, because this is not very structured data, you just have some timestamp and some blob of message posted by the user.

Then, and this messages that the users upload are also given to another component that generates user feed. Now, this component that generates this user feed will see all the messages uploaded by all the friends of a user will see also all the likes and reactions to the message. If your friend has posted a message that a lot of your other friends have liked.

So, this component will subscribe to all of this information, user likes, details of message uploaded and put all of these together to generate what should be the feed that you see. So, if your friend has posted a very unpopular message, it may not appear in your feed. But if your friend has posted something that lot of your other friends have liked, it may appear high up in your feed. So, this component will subscribe to all these streams of information digest it and come up with your user feed. So, that when you make a request to see your user feed, this personalized message feed is fetched and displayed.

And of course, you can have various databases, all your likes and reactions can be stored in one NOSQL database. The user feed can be temporarily stored in another NOSQL database until it is displayed. This might be a persistent database, NOSQL but stored on disk. This might be NOSQL, but just temporarily stored in memory because it is constantly getting updated, as people are posting messages as people are liking messages, this feed is constantly getting updated. So, you may not want to store it on disk, you might just want to store it in memory. So, you can see there are different types of databases and different types of interfaces between components and a modular architecture.

(Refer Slide Time: 17:50)

**Instant messaging: requirements**

- Users connect to server via persistent connections from mobile apps
- If two users are online and wish to communicate, messages exchanged between such users are delivered via the messaging service immediately
- If the recipient of a message is offline, the service buffers messages for later delivery. User's client checks for such messages periodically.
- Users form groups of other users and post messages in groups, which are broadcast to all the users in the group.

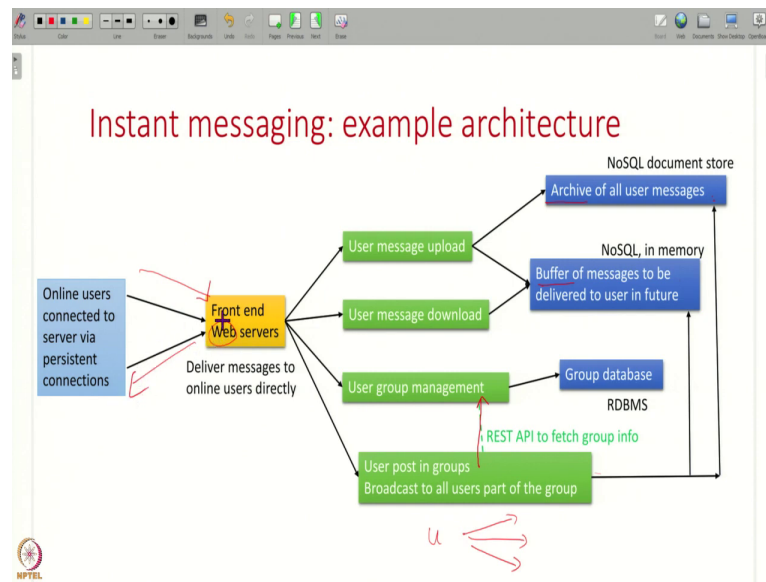
NPTEL

So, next another example is let us consider an instant messaging system like WhatsApp for example. Users connect to WhatsApp, and they can send and receive messages to each other. So, how do you design such a system? So, such systems normally, if there is an application, a mobile app on your phone, you connect to the server via persistent connections, your mobile app will maintain a persistent connection or transport layers over some transport protocol connection to a server to send and receive messages.

And, what are the requirements? What should the server do? What are all the functional requirements on the server? Users who are online should be able to exchange messages. If two users user one and user two are both connected to the server, if user one sends a message to user to user two should immediately see it.

But if your recipient is offline is not available is not connected to the server right now, then the server should buffer the message later on. And when the user logs in later on, this message should be delivered. Then, all of these messaging apps also have the concept of groups, you can form a group of users and when you post a message in a group, all the users in the group should be able to get the message. So, these are all some of the common features you see in instant messaging applications.

(Refer Slide Time: 19:12)



And here is an example architecture of how you would realize these features. Users are connected to the server, there are frameworks to establish persistent transport layer connections and keep them alive, maintain a persistent connection to the front-end web servers of the system. It can be web, it can be HTTP it can be any other application layer protocol also.

And once a user posts a message, this message of course, you want an archive of all the user messages, you might store that in some NOSQL documents store on disk somewhere. So, whenever the user posts a message, you might archive it. Then you will also identify the recipient of the message.

So, if the recipient is directly also available online, then you will directly whatever message has come in, you will store a copy and also send it back to the recipient on this other persistent connection if the user is online, but if the user is offline, then you will buffer the message, all the list of messages that are buffered for each user that is stored and probably this is stored in an in memory database, because you expect this to be temporary storage or if you think you have to store it for a long time, you can store it in disk, it is up to the system to decide how long messages will be buffered for.

Now, when you as a user, you log in, and you check your messages periodically, when you connect to the server, it will fetch all the messages buffered for you and it will deliver them on

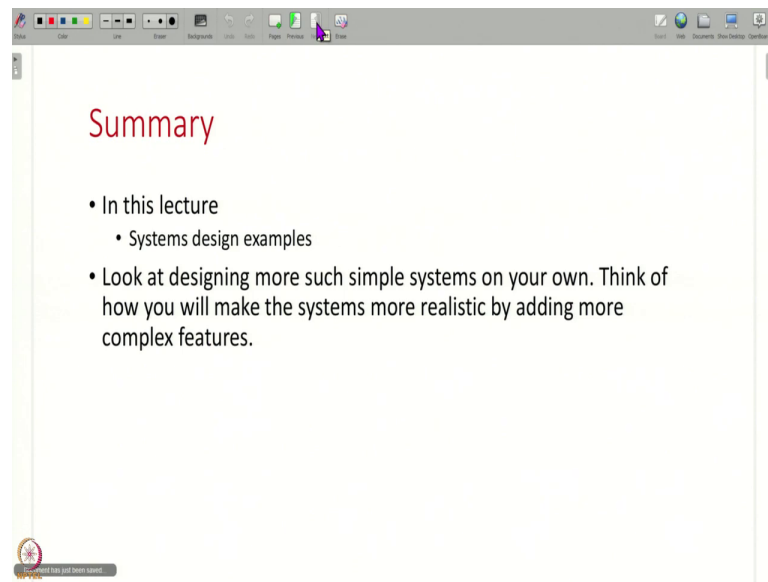
your connection. So, there can be separate components to handle message upload, separate components to handle message download, and so on.

And of course, there might be separate components to handle groups, when you create groups, all these, each group might have some unique identifier and the information might be stored in a database and this can be a relational database, a structured schema or what you can do is you can also store this in some kind of a graph database kind of thing. It depends on the system designer. Then when a user posts a message into a group, then this component will actually talk to this component fetch information about all the users in this group and then deliver the message broadcast the message to all the users.

If one user has posted a message to a group, all the users in that group have to get this message. You will once again go back populate these other databases that store the messages of the user, if the user is online, then you will archive that message received message for the user, deliver it directly if the user is offline, you will buffer the message and deliver it later on. But whatever the message that is posted it will be distributed to all the users in the group.

So, in this way, as you can see, there are multiple different components and different functionalities, there are different types of databases and different types of interfaces between components and here between the client and the system also you can have a persistent connection or many different options can be present.

(Refer Slide Time: 22:20)



So, that is all I have for this lecture. We have seen many different examples of how you design common computer systems that you interact with in your daily life, we have seen examples of end to end design. So, I request you to do more such exercises on your own look at other computer systems that you interact with, and try to come up with functional requirements and try to come up with such an end to end design yourself. And you can also enhance the designs that we have seen just now to add more features to make them more realistic closer to what you see in real life. So, that is all I have in this lecture. Thank you all and we will continue this discussion in the next lecture. Thank you.