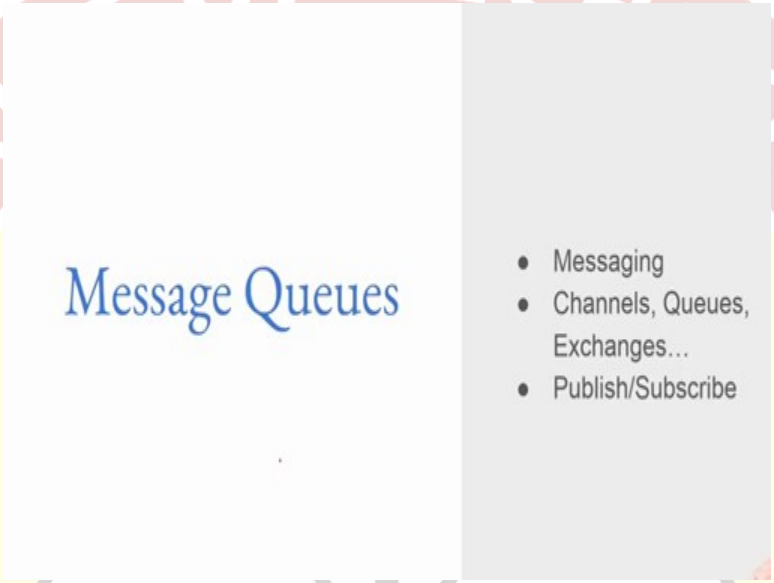
IIT Madras

ONLINE DEGREE

Modern Application Development-II
Professor. Nitin Chandrachoodan
Department of Electrical Engineering
Indian Institute of Technology, Madras
Message Queues

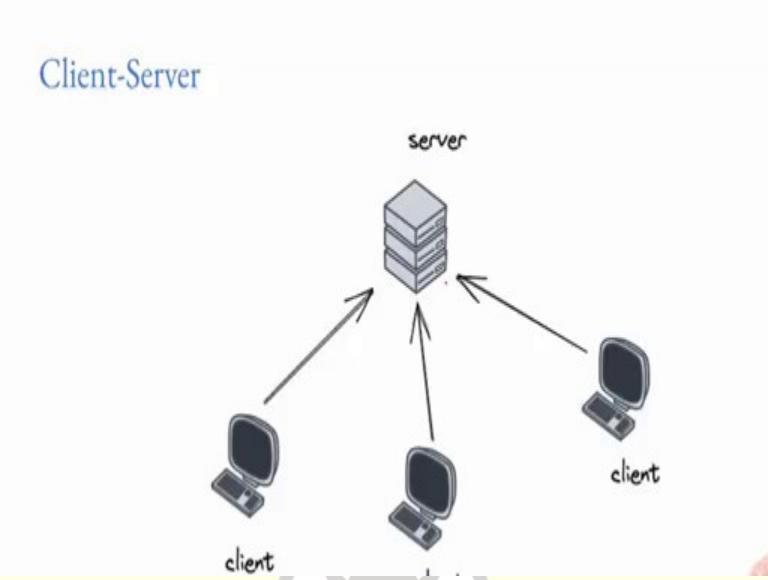
Hello, everyone, welcome to Modern Application Development part 2.

(Refer Slide Time: 00:14)



Let us look at the notion of a Message Queues. We have already discussed that when you have multiple servers that need to communicate with each other to handle the coordination of task execution, there is some kind of messaging and communication that needs to happen between them. So, what are some terms associated with this? How do we set such things up? We will try and understand those now in a little bit more detail.

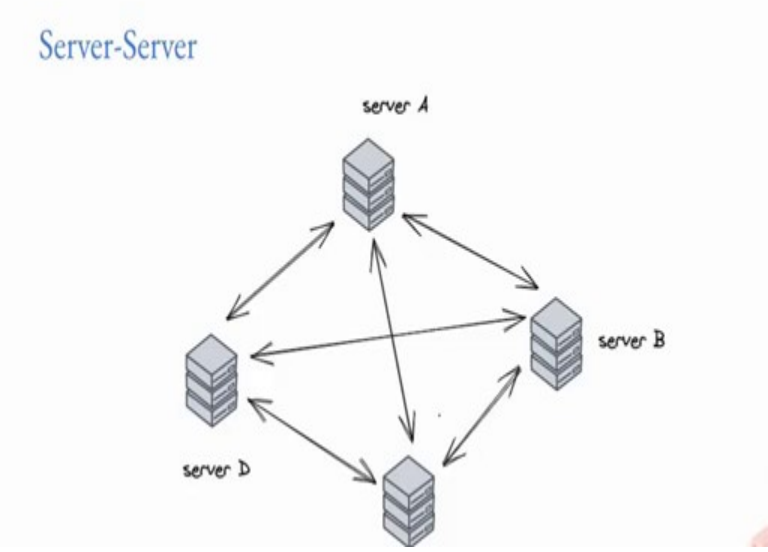
(Refer Slide Time: 00:38)



So, this picture over here is a very simple Client-Server model. Effectively, what we are saying over here is there is some kind of a single centralized server. And there are multiple clients, each of those clients connects to the server sends a request and gets back a response from the server. Now, this is essentially what a web server in fact looks like.

Because after all, there is one domain name, which most likely corresponds to one or at most a few IP addresses, that receives the request from multiple clients and sends back responses. So, this is sort of the traditional HTTP client server model that we are familiar with.

(Refer Slide Time: 01:23)

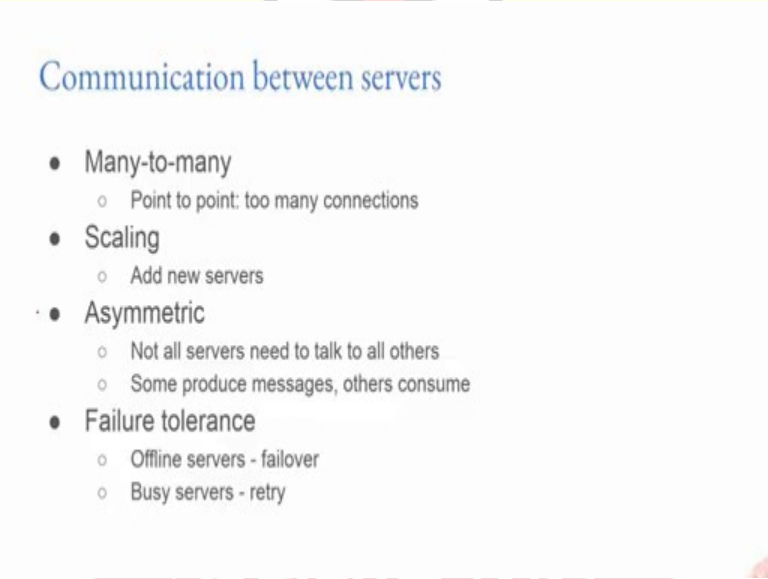


Now, on the other hand, a truly distributed system could potentially have what is called server to server communication. So, I have multiple servers ABCD as I marked here, and for various reasons, each one of them might need to talk to each other server. So, this is one way by which you could do it, you could essentially set up point to point links between each pair of such servers.

Maybe one of them has a web server front end, one of them is a database one of them is handling image recognition, the other one is taking care of email transfers. Do they all really need to talk to each other? Do I need to set up point to point communications in every direction? Does each server need to know how many other servers are present? What if I need to add a new database server? Or what happens if I need to change my email server to another system?

Effectively, what is happening is I do need to communicate between these servers. But should that necessarily be done in this way, by having point to point links between each pair of servers is a tricky question to answer.

(Refer Slide Time: 02:31)



So, this kind of communication between servers is something that you need to take care of. You could have sort of many to many communications. Many servers are there, and many of them, or they might, at some point, all need to talk to each other. Now, one way to do it, as was shown in the previous picture, is to have point to point connections. But that does not scale well.

Essentially, if you have n-servers, you are going to have n-square such connections. On the order of n squared such connections.

Now, if you add more servers, more databases in order to handle bigger load or more front ends in order to handle more requests coming in, or more image processing, workload servers, because more photos are being piled on. How do you handle this? Now, suddenly, you have a scenario where this number of point to point connections is just increasing too much.

It is also Asymmetric, because not all servers need to talk to each other. So, it is unlikely that the database server needs to directly send out emails. Or that the email server needs to send information to the server that is doing image recognition. So, you could have a scenario where some of these servers are producing messages, whereas others are consuming them, meanings that they get messages from the other servers do something and then they generate a response.

And finally, one of the things that you would probably want in this entire scenario is some kind of failure tolerance. What happens is one of the servers is offline, it crashed, or it is rebooting or something else happened to it? Should I automatically go over to the next server of the same type? Or should I wait until this comes back up? Should I keep on retrying until I get a response from a given servers.

(Refer Slide Time: 04:18)

So, the bottom line over here is we need some form of messaging that allows us to communicate between these different systems. And this communication usually happens through a mechanism

known as message queues or some kind of brokers that are able to take messages and make sure that those messages are made available. Or even explicitly delivered to the right clients, clients meaning the clients themselves could be other servers that are waiting for such messages.

Once again, what is the core concept over here separation of concerns. We want the web server front end to handle serving web requests. We want the image processing server to handle image recognition tasks. And we want the mail server to handle email. So, we want to decouple the sending of a message from the execution of the message.

I should be able to just send out a message saying I want an email to be sent, without having to worry about whether the email actually got sent. I send out a message, somebody is responsible for taking care of it and saying, there was a message to send out an email. I have the information I need, I will take care of sending it out.

The communication necessarily needs to be Asynchronous, I do not want to wait until the email has actually gone out, before I return. Otherwise, the whole purpose of having a distributed system and splitting it up among multiple servers is lost. The interesting thing is this brings in a notion of something called data flow processing. Now, data flow is a concept that is been studied very extensively in the context of signal processing systems.

Effectively, what is happening out there is you say that whenever data is present, the computing system responds to the presence of data, it in other words, over here, each of these servers will respond or react to the presence of messages in the queue. If there is a message saying send out an email, then the email server needs to respond to it. The rest of the time, the email server could probably just sleep, or potentially even shut down and not just come back up when required.

So, how does that exactly work, basically, what would happen is that the messages would be in the queue and there is some way by which the presence of a message in the queue can alert the appropriate kind of server. This data flow processing is very powerful, because what it says is, the rate at which these different servers are reacting, can automatically adjust to the amount of activity in the system.

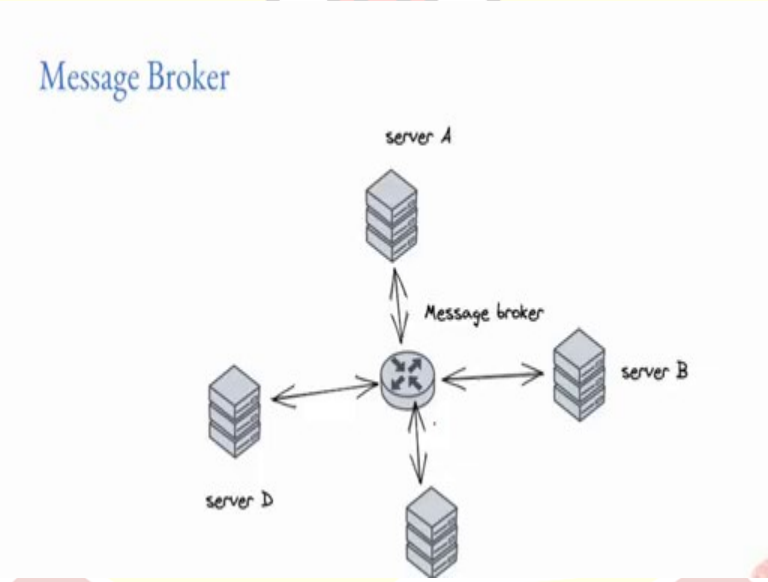
And in fact, it also becomes a lot easier to scale meaning that if I find that there are messages building up of a certain type, I can probably add another server to reduce the load on that kind of message, or process those messages faster. And one final important factor that we would like to

have with regard to these messages is, can we have ordered transactions? In other words, it should be First In First Out.

Let us say that multiple people are trying to buy something on a E-commerce site. So, rather than sort of waiting until you know which of their threads responded, first, I just put in the messages saying, so and so wanted to buy, somebody else wanted to buy, somebody else wanted to buy, I put all those messages into a queue.

And, since those messages are then going to be processed in an ordered manner, because they were in a queue a first in first out manner, it pretty much will take care of ensuring that the first person to put in the request is also the first person to actually get the product. So, all of these are useful features to keep in mind when we are designing such a message system.

(Refer Slide Time: 07:57)



Which means that one of the techniques that is used is to have some kind of a system called a Message Broker. And what we have done is we took out that point to point communication between each of the different servers and have replaced it with one centralized router that takes care of taking in messages from various different places and then shipping them out to the right targets.

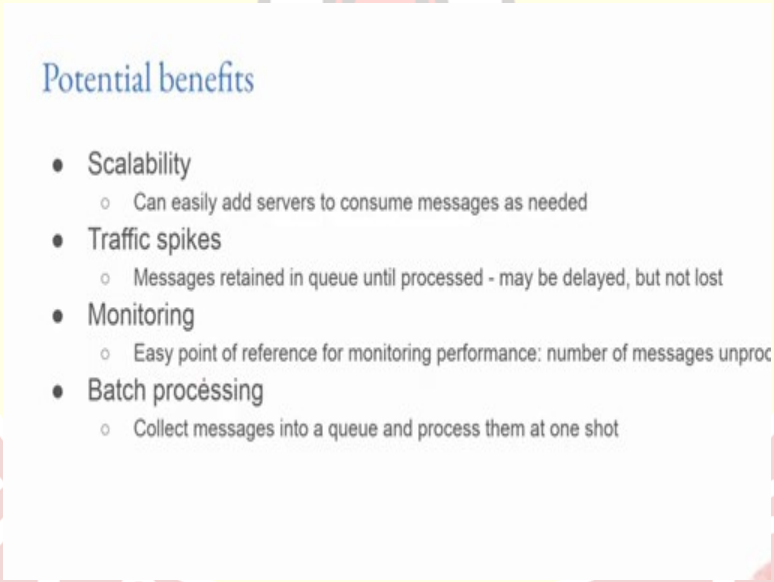
Keep in mind that this Message Broker could potentially have some degree of intelligence, it might actually be able to say, look, I do not really need to send out all these messages to all the other servers, all I need to do is collect information from these servers. And, if necessary, wake

up the servers who need to respond, or alert them in some way that there is a message waiting for them.

So, that amount of minimal intelligence can be baked into the Message Broker, but it does not by itself need to do anything very dramatic. The interesting thing is this scales a lot better, instead of n-square order of n-square links that need to be established to pretty much only need order of n. There is a catch, what is the catch? This Message Broker now is potentially a single point of failure.

What does a single point of failure mean? It means that if the Message Broker stops working, the entire system goes down. So, it does not matter that you had all these different servers, the fact that messages are trying to be sent but are not able to go anywhere, essentially means that your entire system has broken down.

(Refer Slide Time: 09:25)



So, what are the potential benefits of using some kind of messaging Message Broker or a messaging queue system? Scalability, I can easily add servers because after all, what I am looking for is are there messages of a certain kind? So, I can add on more servers that just look at the queue and say, if there is a message saying send out an email, and the other server that is taking care of email is already busy, I will go fetch this and send it out.

It becomes relatively easy to handle small spikes in traffic. What I mean by a spike in traffic is let us say that you are receiving on average, let us say, 2 requests per second to your web server.

But for a short duration, let us say about 10 seconds or so you suddenly receive something like 200 requests. So, that is like 10 requests per second. But only for a short duration, just about in 200, requests over a period of about 10 seconds or so it is suddenly gone up to around 20 requests per second.

Now, that is a traffic spike, it does not actually mean that you need to add a whole lot of different servers, it might have been some coincidence, or it might have been something where, there was a brief flurry of activity, some bunch of people suddenly came together at the same time, and hit the server. Now, with a normal web server based approach, what would happen is that all of those requests would go, suddenly, a large number of threads would get created, and each one of them would be taking up memory resources.

On the other hand, if you have a Queuing system, each one of those requests essentially put something into a queue can accuse be processed quickly, maybe not, things might slow down a little bit. So, for that duration of about 10 seconds or so people might find the system slowed down a little bit. But after that time, once again, things sort of taper back and become normal. So, it is able to sort of smooth out the sudden spikes, because as long as the messages go into the queue and do not get dropped.

I still see that I get a response, I do not find no website not responding, or some kind of a message like that, which is very upsetting to people when they are trying to connect to something. It also helps to monitor, and this is very useful, because what happens is that let us say I suddenly find that I am getting a large number of database requests or a large number of requests for processing certain kinds of images.

And that is not really getting processed fast enough, the queue is building up. I just have to monitor the queue, I just have to see, how big is this queue? How much is the backlog that is building up. And if I find that there is a backlog, I can very easily say, look, maybe I need another server out here to handle the images.

And I can sort of selectively decide to scale only the parts that are required in order to handle the queue that needs to be processed quickly. One other thing is that you can actually do what is called batch processing. So, maybe there are scenarios where I do not really want to process each and every message as soon as it comes, the messages themselves might be relatively small.

It might just be an update of, what are the scores from different people playing angry birds or something like that each one of them. As soon as I get a message from someone, I do not need to go and immediately update the leader boards and the highest score, I can do it probably after a gap of like a few seconds. So, once in every few seconds, I wait, collect all the information received in a batch, process it and say, this is the new leader board that I have. So, all of these are potential benefits of using Queuing systems.

(Refer Slide Time: 12:57)

Now, there are many variants to how this can be implemented? The message queue is what I have sort of been focusing on for the most part. It is sort of a point to point link in the sense that there is a producer, but instead of the producer directly sending a message all the way to the consumer.

So, an example of this would be the producer is the web front end that accepts the image to be processed. And the consumer is the server that is actually going to do the image recognition. Instead of directly connecting to the server that does image recognition, the producer just puts the message the image onto a queue. And the consumer then pulls it off the queue and starts processing. That is one of the most basic forms of such distributed system communication.

Now, there is another interesting variant, which is called a publish subscribe or a Pub Sub model. What this allows is to have some kind of fan out. And what we mean by fan out is that a producer can publish information without really caring about who is going to read it. And you

could have multiple subscribers that have connected to the queue and have informed a Message Broker saying, look, anytime there is an update on this, let me know.

An example of this could be something like, let us say there is an update on stock exchange prices. The producer is the one that is getting an update on the stock exchange price. And the consumers are all the different trading managers, the fund managers and so on who are interested in getting information about what happened. So, if the stock exchange needs to individually go out and send information to each one of them, that is going to be an impossible task. It does not scale.

Instead, what would normally be done is they would probably in a old time stock exchange would probably write it up and stick it on a wall somewhere. Every few seconds or whatever it is, whenever the stock exchange gets an update, they would write up some information and stick it somewhere and everyone then goes and looks at the wall.

Same way, you could have subscribers that look at the message queue and say there was an update on stock prices, fine. I am interested, I want to subscribe to this, I want this information. So, the Message Broker then stores that information about who are all the subscriber, the one that is generating the information does not really care about who is consuming the information which means that fan out becomes very easy in this model.

There are many variants; many of these variants have been designed more in the context of distributed systems and not necessarily the web. So, I am just mentioning them for completeness. And, primarily, because some of these ideas can be used in the context of the web, there is something called a message bus. A bus is an idea that comes more from electrical hardware, it is sort of a set of wires that carry related information.

So, those of you who have looked into the innards of computer or write or how a computer works, might have heard the terms address bus and data bus. So, the address bus is something that carries information about which memory address your processor is trying to get information from. And a data bus is something that actually carries the information from the memory to the CPU and back.

So, in the same way, a message bus in a distributed system is something that could potentially allow for some kind of addressable communication between different entities. So, once again, it

is similar to this message queue. But the difference is that I might have different servers sitting over there.

And let us say server A wants to talk to server B, without having to establish a direct point to point connection between A and B, A can then sort of communicate with the bus and say, I want to get a message through to B, and the bus takes care of routing the message appropriately. There are so called API's and web services, which any web server could expose a web service or an API, saying that, you want to communicate with me directly use my API. You just make a request to this endpoint, I will receive the information.

Now, that is an interesting point, what it is saying is, look, the server says, anybody wants to communicate with me directly use my API, why are you sending something to another intermediate Message Broker, you want to send me a message fine, just, make a connection to this endpoint, send me the data that I need to accept.

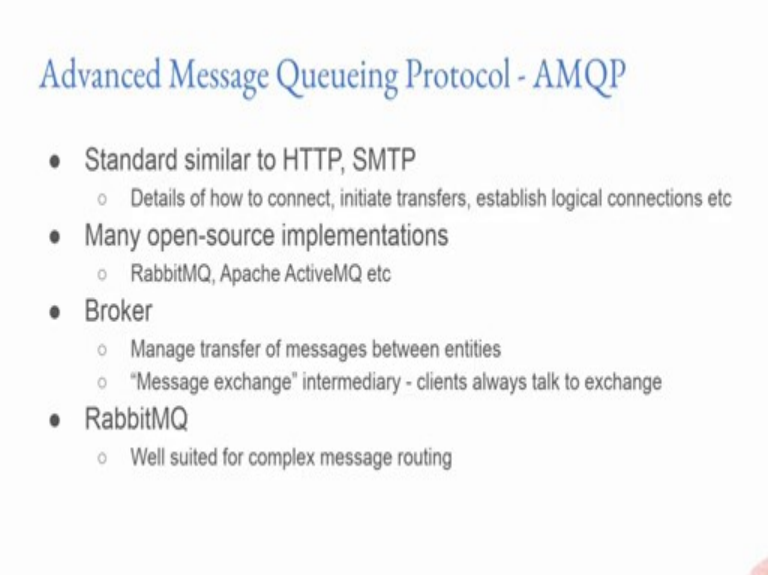
And this is, in fact, a very common approach that is used in a lot of systems. Why? Because it is simpler to implement, you do not have this additional overhead of the message queue. There are problems with it, though one of them is that it is not really resilient, you do not have sort of storage of the messages, what happens if the server that is handling these API requests, crashes, or it is overloaded.

There is no sort of temporary storage and making sure you can retry or retransmit or, just ensure that the processing gets completed properly, none of that happens in a systematic manner. And finally, you could also think of just having a database as a way of as a means of communicating between different systems. After all, a message is a piece of information databases are good at storing information.

But the problem is, this is not really the scenario that you are creating databases for databases are designed for something where I want to store structured information, usually and large amounts of information. Whereas over here, I am more concerned with how fast does the information travel between different entities?

And how quickly can it be processed? So, there are several different variants or different techniques that you can use in order to do this messaging. And of course, at the end of the day, you need to pick the one that is most useful for the problem that you are trying to solve.

(Refer Slide Time: 18:53)



So, there are a couple of different examples that I am just going to mention briefly or here. There are many other techniques, as you can imagine? One of them is what is called the Advanced Message Queuing Protocol or AMQP. It is a protocol. It is a standard which is similar to HTTP SMTP.

Meaning that the details of how to connect to an AMQP server how to initiate the transfer, what is a logical connection? What is the kind of header information? What is the kind of request information? How should responses look, all of that is specified as part of the protocol? There are several open source implementations of AMQP, one of the most well known as what is called RabbitMQ?

There are others Apache ActiveMQ, there are some proprietary implementations as well. It effectively functions as something called a broker, which means that it manages the transfer of information between different entities that are interested and it primarily is working as a message exchange. So, it brings in this notion of an exchange.

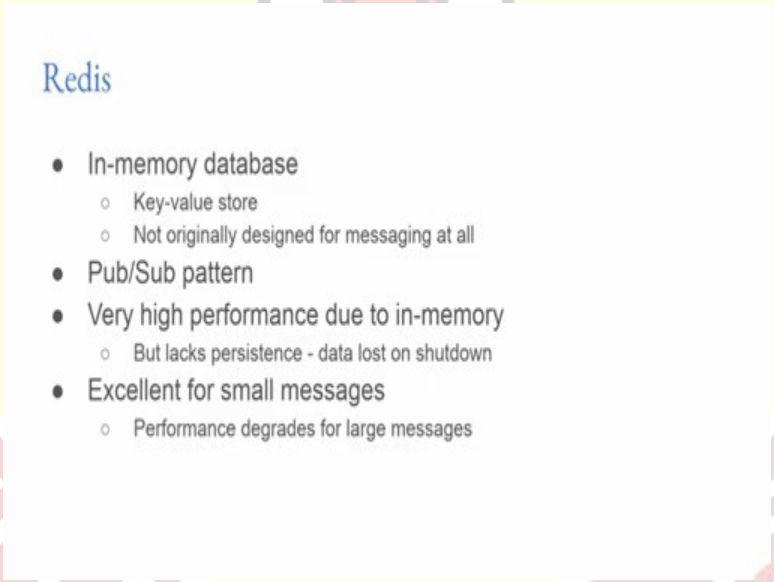
None of the servers or clients rather directly talk to the Message Broker or to other clients or to other servers, they talk to the exchange and give information there. They exchange then takes care of delivering it to whoever else is required. You might have noticed that over here, I am going on jumping between the terms client and server. And the reason is because in a distributed system of this sort, pretty much every system can be thought of as either a client or a server.

Every one of these servers is a server in the sense that it is actually receiving certain tasks and performing them. But it is also a client of the message exchange. It is relying on that message exchange to accept it is messages, and also to deliver to it messages that it needs. So, distributed systems in general, it is not useful to think of them explicitly as client server systems.

But the notion of something acting as a client meaning, that it is trying to request some kind of data or as a server meaning that it accepts a task and returns a response. It is still useful to find out how the functionalities actually progress. So, RabbitMQ, in other words, along with other servers in this class is very well suited for complex message routing.

Meaning that if you have a fairly complicated system with a large number of different types of servers, and you want to have some complex routing between the different kinds of servers, a protocol like AMQP, with a service like RabbitMQ would be a good solution to have.

(Refer Slide Time: 21:35)



Now, Redis on the other hand, is another application that very often comes up in this context of message queuing. The thing is Redis, by itself is actually not a message queue. It is meant to be an in memory database. And what an in memory database means is that it does not store anything to disk by default, it you can sort of do persistence, which means that you can actually dump the data out of Redis onto a disk server.

But while regular operation is happening, everything is supposed to run in memory. And Redis, by itself was not designed for messaging. All that it does is it allows you to define keys and store

a value corresponding to each key. So, it is a key value store. A key value store pretty much just means a dictionary in Python, or an object in JavaScript. So, you can have keys, and you can have corresponding values for those.

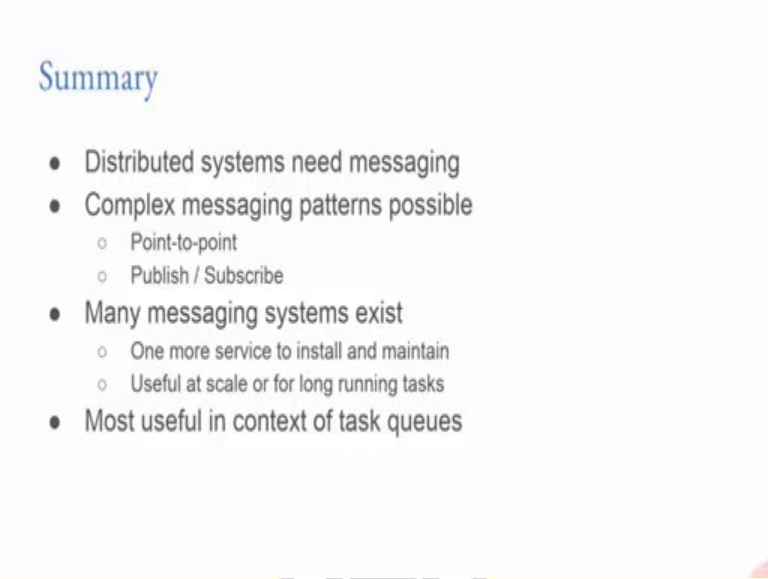
Now, what makes something like Redis powerful in the context of a messaging system, it is because along with having this key value store, they were also able to add on a couple of other things, which essentially allowed it to implement the publish subscribe pattern which means that I can have some entities that write to the Redis database, they publish information out there. And there could be other entities that are doing sort of blocking reads on certain channels that are present in the Redis database.

So, all they do is they look for information to be present on a given channel or a given database. And what the Redis server says, as soon as information is present on those, it will immediately alert anybody who is trying to read on that channel, and give them the information that they want. Automatically, that implements the publish subscribe pattern, which in turn means that arbitrary messaging can actually be implemented using this.

So, even though Redis was designed as a key value store, and not primarily as a messaging system, it can perform the same tasks. Now, why is it useful? It is because of the extremely high performance. It is implemented completely in memory, which means that everything, access to all the data is very fast, it is the fastest that you can imagine in a system, because in memory is always faster than anything that needs to go out to disk.

The problem is it lacks persistence, what if the server crashes; everything that was in the database is lost. So, the idea is that you should be designing your system in such a way that it can recover from such crashes in a relatively easy manner. In addition to that, Redis is also excellent for small messages. But the performance can actually degrade quite badly if you are trying to send large amounts of information through the Redis database.

(Refer Slide Time: 24:25)



So, to summarize, distributed systems need messaging, and you could have complex messaging patterns. Some examples are just the point to point communication and of course, the publish subscribe pattern. But as I mentioned, you could have point to multipoint or you could have something where a particular server needs to talk to some other server that is coming connected through another distributed system.

So, it is possible that you need to have some kind of relatively complex patterns to be implemented. There are many messaging systems that exist to solve problems in an each of them sort of specialize in a slightly different way. And one of the biggest issues, though, with this is that it is one more service that needs to be installed and maintained.

So, none of this sort of comes for free. Effectively, what you are saying is, if you are scaling out, you have long running tasks, and you have something that is going to be computationally intensive take a long time, you almost certainly need some kind of a messaging system. But if you are writing a relatively small app, then a messaging system might be overkill, it might be too much work to implement.

So, at the end of the day, do you really need something of this sort? Are there other ways by which you could have handled it is still something that should be decided with care, not just put in a messaging system because it sounds like a cool thing to do. So, this is most useful, these messaging systems, of course, are primarily useful in the context of handling task queues,

because the messages that you are trying to transfer between these different entities is usually something that is related to performance of some functionality. So, what exactly are these task queues and how do we make use of them is something that we also need to understand in a little bit more detail.