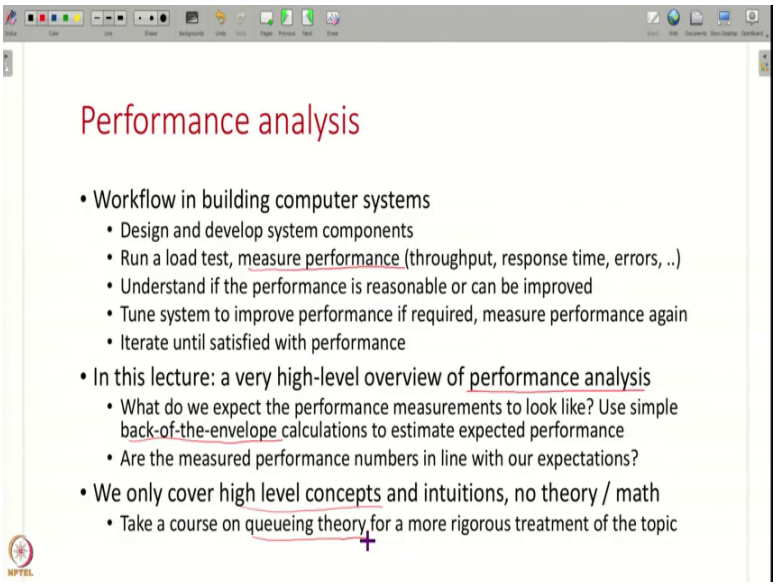
Design and Engineering of Computer Systems
Professor Mythili Vutukuru
Computer Science and Engineering
Indian Institute of Technology, Bombay
Lecture 46
Performance Analysis

Hello all Hello everyone, welcome to the 32nd lecture in the course Design and Engineering of Computer Systems. In this lecture, we are going to understand a little bit more about how to analyze the performance of your computer system.

(Refer Slide Time: 00:30)



So, what is the workflow and building computer systems that we have seen in the course so far, you will first design and develop your system and you will put all the components together, you will run a load test and you will measure performance. So, we have seen what this means in the previous lecture, what do we mean by measuring performance.

You will vary certain input parameters, vary the incoming load into your system and you will measure various metrics like throughput response time errors, utilizations and so on. Now, the next step is you want to understand if the performance is reasonable or if it can be improved. So, you have to first look at these performance numbers and analyze them and understand them to see, does this performance make sense? Is it justified?

Is it some bug in my performance measurement? Or is this how it is supposed to be? So, you have to kind of, do some back of the envelope calculations to convince yourself that this performance measurement is correct. And then once you know that, this is my performance

measurement, but I expect a lot more load into the system, then you will go ahead tune your system, and you will repeat this process again. In this lecture, we are going to see how you do performance analysis. What performance measurements do we expect from the system?

We will use simple back of the envelope calculations. What does this mean simple rough calculations that you do on rough paper to see that, okay, I am getting 100 requests per second, is it correct? Is it okay? Is it not okay? You know that kind of rough estimates, how do you do that on your performance, that is what we are going to study in this lecture.

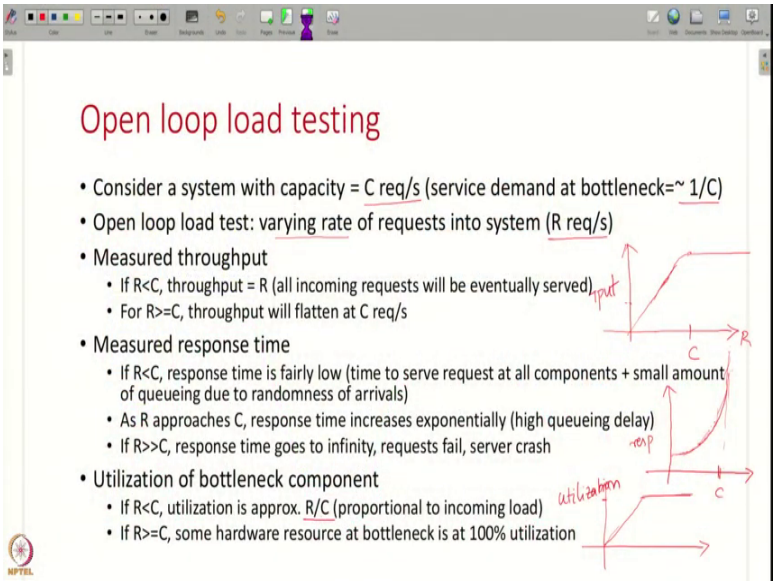
And once these estimates match what you are seeing, suppose you estimate to get a performance of 100 requests per second in your system, because each query is taking 10 millisecond, but you are seeing only a performance of 10 requests per second, a capacity of 10 requests per second then something is wrong in your measurement setup that you have to fix.

So, these back of the envelope calculations are very important to get a rough sense of, is your performance measurement correct, is it roughly in the same ballpark range that you expect and so on. So, in this lecture also, once again, I would like to clarify that I will only cover the high level concepts and I will try to give you only an intuitive feel for performance analysis.

But actually doing performance analysis involves a lot of math and notation and theory has to be introduced that we do not have the time for in this course, if you are interested in this, you should take a course on queueing theory, that is the area of computer science that actually studies how these queues build up due to performance bottlenecks, what are the throughput response times you get, all of those are studied in detail in queueing theory.

In this lecture, I will not go into a lot of math or write scary equations, I will just try to explain it in simple intuitive terms, which is enough for most cases of doing a performance analysis in real systems. So, now, let us see what performance numbers we expect in an open loop load test.

(Refer Slide Time: 03:39)



Let us begin with an open loop load test. Suppose you have a system with a capacity of C requests per second. Where is this capacity coming from? At the bottleneck component, the service demand is roughly 1/C, roughly every request is taking 10 milliseconds to process at the database. Therefore, your database is able to handle 100 requests per second. We have seen this in the previous lecture.

So, now to this system with a capacity of C requests per second you send a stream of requests at varying rates. Suppose the rate of requests into the system is R requests per second say 10 requests per second, 20 requests per second, 100 requests per second and so on. So, this is your open loop load test. Now, what is the throughput, how will the throughput look like?

So, queueing theory tells us the following, not just queueing theory even common sense will tell you this, which is this is the value of C and this is rate is varying from some small value C, beyond C and so on. Then the throughput that you will measure will look as follows. Initially the throughput will be equal to R. If your input load is 10 requests per second your throughput will also be 10, input load is 20 throughput will also be 20.

So, your throughput will initially be linear, will be equal to the offered load incoming load into the system until you reach capacity. And once you reach capacity as your incoming rate exceeds the capacity your throughput will flatten out, this is intuitive, we all understand why this happens, this is the measure throughput. What about the response time?

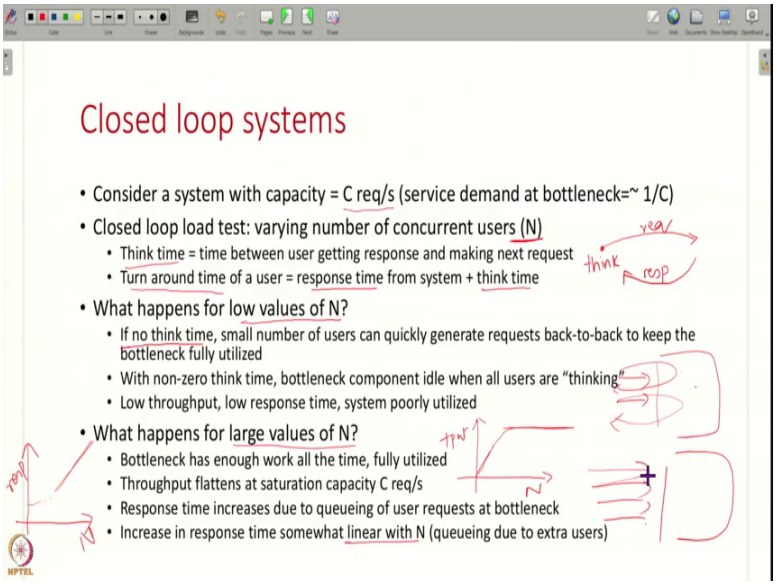
So, initially the response time as you vary the load into your system and this is your capacity initially, your response time is low. Because there is no queuing all components are free whatever request comes it is quickly processed. If R less than C the response time is low, as R approaches C, your response time increases exponentially, your graph will look something like this.

And as R is much, much greater than C your response time almost goes to infinity, if you have an infinite amount of load coming into your system, your response time drastically increases and goes to very, very high values and at some point queues build up queues overflow requests, fail, errors, crashes, all of these will happen. So, your response time starts slowly then increases increasing as you reach capacity and as you exceed capacity or response time just rapidly increases.

And what was the utilization of the bottleneck component? Utilization also initially, if you are below capacity, your utilization will be approximately the incoming load divided by capacity. Utilization if you see on a scale of 0 to 1, initially, it will be like proportional to the load. And at some point your utilization will reach 100 percent or will reach if it is fraction, it will reach a value of 1. And this will be of course, the utilization of some hardware component.

If your work is mostly on the CPU, this will be CPU utilization, if your work is mostly on the disk, this will be disk utilization, whatever is your bottleneck resource, that utilization will reach 100 percent as you reach capacity. These are all intuitive, your common sense also will tell you this, but you can use the theory in queueing theory the equations to derive similar values. So, when you run an open loop load test, this is what you should expect.

(Refer Slide Time: 07:20)



Now, what about closed loop systems, you have once again, let us consider a system with a capacity of C requests per second. And now, in a closed loop system, you are varying the number of concurrent users, this is how you vary the load, you are no longer saying send 10 requests per second into my system, you are saying have 10 concurrent users to my system.

And what are these concurrent users doing each user is sending a request, getting a response, and then thinking for some time, this is called the think time, you know this is, why this think time, this is how real users behave. If you want to measure the performance of your system, when there are 1000 users connected to it, you want to kind of be as close to real user behavior as possible.

Real users are not just doing request, request, request, they send a request, you see a webpage, then you think for some time, then you click on something else. So, this think time kind of emulates real users. So, the think time is between, you get a response, you make the next request again, that is called your think time. So, every user has a certain what is called turnaround time of the user.

That is if I am a user, I will, what is the gap between my successive requests is the time the response time have a system, I send a request, I get a response time plus my think time. So, it takes for example, 2 seconds for the system to respond. And I will think for another 8 seconds, then I have sent a request in 2 seconds response comes another 8 seconds to think so 2 plus 8, 10 seconds is my turnaround time for user, that is the time between successive requests from a user will be this turnaround time.

So, these are some of the parameters in a closed loop load test. Now, let us see what happens if the number of concurrent users is very small. Like just like for open loop load test, we have seen what happens if you vary the request rate. Here, also let us see what happens for as you vary the number of concurrent users into your system. So, suppose you have very low number of concurrent users, you have this big system and you have only one or two users concurrently connected to it, then what happens?

If there is no think time, if each user is quickly, quickly sending requests, response comes again, again, again, quickly send a request you are just bombarding with no other work in mind. You are just continuously loading a system, then even for a low number of users, you can still generate enough load to saturate your system.

For example, a response comes back in 1 millisecond, immediately next millisecond I send a request, again the response comes back, again, again. So, a small number of users can also fully load a system. But if you have nonzero think time, if you want to be more realistic, then what will happen. If you have very small number of users, then your bottleneck component will not be fully utilized, your system will not be at maximum capacity, because it could so happen that all these users are thinking.

Suppose every user once he gets a response back thinks for 10 seconds before sending the next request, then what will happen, this user has got a response, this user has got a response. And both these users are now thinking and the system has no work to do, so it can so happen that if the number of users is very low, all these users have gotten a response and are thinking have not yet come back, have not yet turned around to the system again, then your system will be underutilized, will be poorly utilized, your throughput will be low.

So, therefore, when you are running a load test, with a lot of think time between requests, and all of that, you need to have good enough value of N, you cannot use a very low value of N. Now, then what happens for large values of N? There are a large number of users into the system, then your system will be fully utilized.

Even if some fraction of those users are thinking, there will be other users who are sending requests, they will all be at different phases. So, one user was sending a request waiting for response the other user is thinking. So, once you have large enough number of users somebody or the other will not be thinking and will be making a request into the system.

Therefore, you will have enough work to fully utilize your bottleneck component and your system throughput.

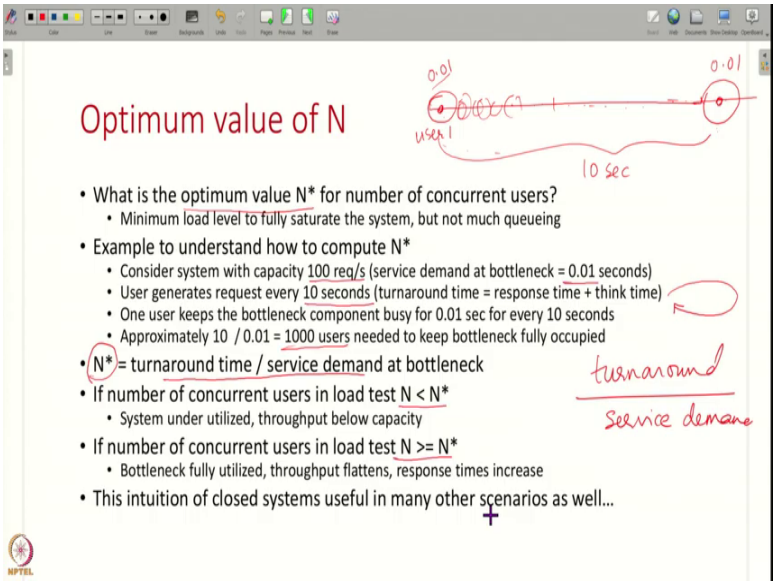
So, that is why as you increase the value of N, once again, your system throughput will initially be low for low values of number of users, but eventually it will flatten out and your bottleneck will be fully utilized. And what happens to response time? Response time also, as you increase the number of users, it will be low initially, and then it will increase, but note that the increase in response time is somewhat linear with N.

So, if you look at response time versus N, initially, it will be low, it will be low and then it will increase somewhat linearly. Why? Because what is your queuing due to, your queuing is due to the extra user, suppose your system has 10,000 users, you will have 10,000, maximum 10,000 users waiting in front of you in the queue.

And therefore your response time does not blow up the way it does with open loop systems. But it is somewhat linear in it, it increases somewhat slowly, because there are only so many users into the system, you do not have unlimited load coming into the system, you have a finite number of users who can only generate so much load into the system. So, with closed loop systems, the response time also increases.

Obviously, as you increase load response time will always increase but it will increase not so drastically, like in open loop systems. So, basically, the summary of all of this is if you have too low values of N you are not loading the system, if you have too high values of N, your throughput has flattened out and your response time will keep on increasing. Why? Because suppose you need only 10 users to keep the bottleneck busy and you have 20 users in your system, the other people are just queuing up in front of the bottleneck.

(Refer Slide Time: 13:19)



So, the question then comes up, what is the optimum number? Is there an N*? Optimum number of concurrent users, which is just enough to load the system, to let it reach maximum throughput but without too much queueing. Just enough number of users to fully load the system. What is this value of N*, you might ask. So, let us see how to intuitively compute this N*. Of course, there are a lot of math and theory behind this, but I will just try to explain it in simple terms.

So, suppose you have a system with a capacity 100 requests per second, as we have been seeing so far. What does this mean? This means that the bottleneck takes 0.01 seconds to service each request. And now suppose the turnaround time of your users in the system is 10 seconds. That is the user makes a request gets a response back, before he makes the next request this gap between request is 10 seconds.

So, then what is happening? If we look at the bottleneck, the bottleneck has handled a request for 0.01 seconds from one user. Then there is a gap of 10 seconds before this user comes once again to the bottleneck and consumes 0.01 seconds at the bottleneck. So, can you all visualize this? Here is our user 1, he came to the bottleneck, the bottleneck service his request in 0.01 seconds. Then the turnaround time is 10 seconds.

You know the response time plus think time everything is 10 seconds. After 10 seconds the user has come back again and the bottleneck is busy for 0.01 seconds again. There was only one user in the system all of this gap, this 10 second minus whatever the small value, all of this gap is idle for the bottleneck. So, how many such users do you need to keep this bottleneck fully occupied, if each user is consuming 0.01 seconds out of this 10 seconds at

the bottleneck, you will need this 10 divided by 0.01, turnaround time divided by service demand, the turnaround time divided by service demand.

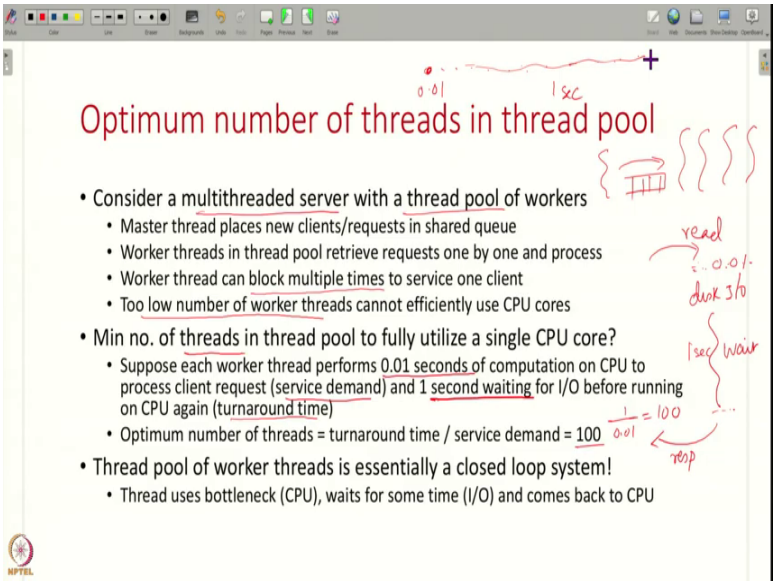
If you had that many users, then this bottleneck will be fully utilized and your system will be running at full capacity. Can you all see this, this bottleneck? Every user takes a small slice of the bottleneck, and that user comes back again after a turnaround time. So, how many such small slices do you need this turnaround time divided by the size of the service demand of each user, these many users if you have, this 10 divided by 0.01 as if you had 1000 users, then the bottleneck will be doing the work of all of these 1000 users before this first user comes back again.

This will ensure that the bottleneck is fully utilized and you are getting the maximum throughput, maximum possible throughput from your system. So, this is a rough formula to calculate the optimum value of N*. In your load test, if the number of concurrent users is less than this optimum value, your system will be underutilized, your throughput will be below the capacity.

And you are not correctly measuring the capacity of your system. If the number of users is much more than N*, of course, your throughput flattens out, but your response times will unnecessarily increase. So, if you want to measure at the optimum point, what is the throughput and response time of my system, you will basically in your load generator, you will set this value of N* according to this formula.

So, this intuition about load system is actually very powerful, seeing how to calculate the optimum number of users to keep the bottleneck busy. This intuition is actually very powerful, and it can be used in many other scenarios. I will give you a couple of examples.

(Refer Slide Time: 17:12)



So, for example, let us consider a completely different scenario. We have seen this concept of a thread pool before. There is a master thread, and it is giving work, it is putting various requests in a queue. And there are several worker threads that will take these requests from the queue and service them. We have seen this example before, this design before for multi threaded server, this multi threaded TCP server getting many requests from many new clients are connecting to it, it will keep all of these requests in a queue.

And each worker thread will take a client, handle the clients request, send a response back, and so on. So, that the main master thread is just focusing on accepting new requests. So, now this worker threads can also block multiple times, the client has requested something, the worker thread will read the, read the client request, then do some processing on it, then make some disk IO to read the file that the client has requested, it will block and then it will once again run on the CPU and send a response back.

So, your thread could be doing this, it could be waiting for a long time for the client, blocking for the client then doing some more work then maybe multiple disk IOs multiple times blocking. So, each thread is using the CPU for some time blocking, using a CPU for some time blocking.

So, if you had a very low number of worker threads then what will happen, you cannot utilize all your CPU cores, you have a 8 core machine but you have only like 2 or 3 worker threads and your CPU is laying wasted. So, then question comes up how many threads do you need in your thread pool to fully utilize your CPU? So, this is a very important question, when you

are building this multi threaded system with the thread pools you have to decide what is the size of my thread pool, how many threads do I want in my thread pool.

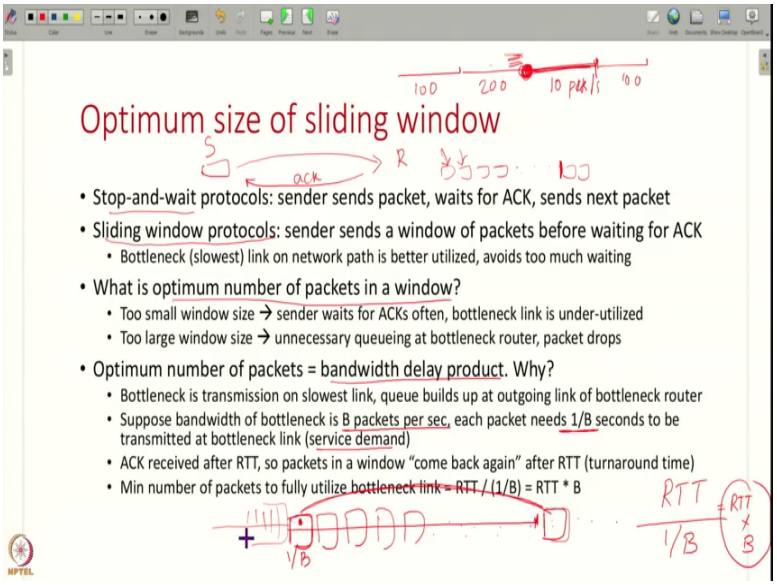
So, how do you go about deciding this? Here is the intuition. Suppose each thread performs 0.01 seconds of work on the CPU it does some computation on the CPU and then it waits for a long time, 1 second and then again for the next request it will do 0.01 seconds here and then it will wait for 1 second here. So, this is how a thread in a thread pool behaves. So, note that now can you see you can map these two actually.

This is the service demand 0.01 seconds, this is the service demand on the CPU. And this 1 second is simply the turnaround time. Thread is working at the bottleneck, at the bottleneck resource at the CPU for 0.01 seconds, then coming back again, after 1 second, 0.01 seconds come back again after 1 second. So, in this scenario, one thread of course is not enough to fully keep the CPU core busy, why?

Because the CPU core is doing some work for 0.01 second and then for the next 1 second there is nothing to do. So, how many such threads do you need to fully occupy this CPU core? The answer is once again turnaround time divided by service demand which is 100 in this case, which is 1 divided by 0.01, that is 100. So, if you had 100 threads, each thread is doing work for 0.01 seconds then blocking for 1 second and so on, then these 100 threads together will fully keep your CPU core occupied.

So, we have used the same intuition that we used for deriving the number of concurrent users in a load test, we use the same intuition here to derive how many threads do you need to have in a thread pool. This is also a closed system, the thread pool is also closed system. So, this analysis is very powerful and every time you have a thread pool, you should actually do this analysis. Each thread is doing so much work and it is waiting so much time for IO. Therefore, how many threads should I have in order to fully utilize my CPU?

(Refer Slide Time: 21:16)



So, another example, I will give you another scenario where the same thinking can be applied which is in the case of sliding window protocols. If you remember a few weeks back we have seen networking transport protocols use a sliding window of packets. So, if you just send one packet, the packet reaches the receiver, receiver will send back an acknowledgment, then you send the next packet you are wasting a lot of time waiting for acknowledgments.

So, this is your sender, this is your receiver and you send data you get an ACK back and suppose all of this takes a long time. Then your sender is just sitting idle, all your network links are idle while you wait that is not optimum. So, such protocols stop and wait protocols are not optimal. So, real life transport protocols use a sliding window of packets, that is you will send a window of packets and then you will wait.

And then when an ACK comes back for this first packet, you will send the next packet, when an ACK comes back you will send the next packet and so on. So, the question comes up what is the optimum number of packets in a window? If I have too few packets in a window what is happening I am unnecessarily waiting these packets have gone out I have no other work to do.

And I am still waiting for ACK, if I have too many packets what is happening if I just dump million packets into the network, some queue somewhere will overflow at the router at the NIC somewhere the queue will overflow, packets will get dropped and TCP will once again reduce its congestion, window slow down all sorts of bad thinks will happen. We do not want that.

So, what is the question comes up, what is the optimum number of packets in the window? So, back in the networking part of the course we have seen that the optimum number of packets is the bandwidth delay product. So, now we will try and see your reasoning for why is this the answer. So, in a computer system, what is the bottleneck, the bottleneck is the slowest link.

If you have multiple links in your computer system each with different rates, the slowest link is where the queue will build up. Suppose this link can send 100 packets per second, this can send 200 packets per second and this can send only 10 packets per second and this can send you know again 100 packets per second, then packets flow quickly, but they will build up here, at this router they will build up.

At the beginning of this slow link is where they will build up because so many packets have come in but this router is only able to send one packet every one-tenth of a second, at the rate of you know 10 packets per second you are only able to send so much, so packets will build up here. So, if your bandwidth of the bottleneck link is B packets per second each packet is taking 1/B seconds.

If your bottleneck links rate is 10 packets per second each packet is taking one-tenth of a second to be transmitted. So, in some sense, you can think of this as the service demand. So, if you look at your bottleneck link, it is taking 1/B time to send a packet, then 1/B time to send the next packet to send the next packet and so on.

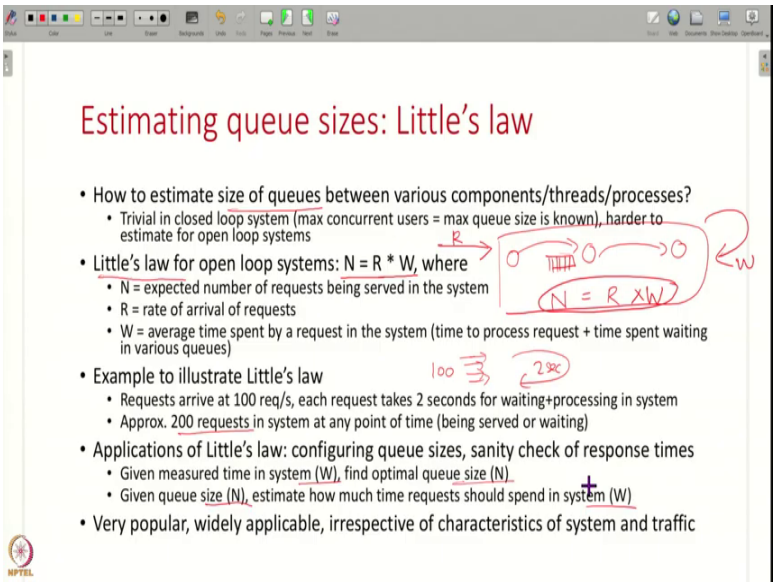
And once it transmits a packet, this packet will reach the receiver an acknowledgment will come back and the next packet from the sender will come only after RTT. So, the turnaround time is basically RTT, if the bottleneck link forwards the packet, the packet will reach the receiver, ACK will come back, the sender will send the next packet again, this entire loop to complete when will this particular you know slot in the window come back again it will come back again after RTT.

So, therefore, you have, again you can map it to the same, it is like a closed loop system. How many such packets in a window do you need to ensure that while this bottleneck link is waiting for an acknowledgment, it is fully utilized, how many such package do you need? Your turnaround time is RTT divided by your service demand is 1/B. So, this works out to RTT into B, the bandwidth delay product.

So, basically, your bandwidth delay product is nothing but the number of concurrent packets you need to have in order to ensure that your bottleneck link is fully utilized. If you have fewer packets than this, your bottleneck link will have some gaps. Why? Because all of these packets have been sent yet, the RTT time is not done.

So, the next set of packets from the window have not come, your bottleneck link is idle. If you send more packets than this what will happen? Only so much your bottleneck link can forward. The more packets than this, they will all be just piling up in the queue at the bottleneck router. So, once again, the same intuition of closed loop systems is useful to derive the size of a sliding window also.

(Refer Slide Time: 25:57)



So, now that we have seen how to estimate how many packets will be queued up and all of that let us understand a little bit more on queues also. In any system, say if you take any open loop system or anything, how do you estimate how queues build up? How many packets or how many requests are queued up at any component? How do you go about estimating that? So, there is a law in open loop systems, which is called the Little's law, which lets us estimate the size of queues between various components.

So, this component is sending some request to this component, it is sending some requests to this component. So, between two components, at what rate will this queue build up? That is specified by the Little's law. Note that for closed loop systems is easy, you only have 100 concurrent users, the maximum size of the queue can be 100, because there are only so many people who are in the system.

But with open loop systems, you do not know how many requests are there in the system. So, you use Little's law. So, Little's law tells you that N = R * W, this is a very famous law, this is very popular, widely applicable, no matter what is your traffic arrival characteristics, your service demand characteristics distributions, it always applies, it is a very beautiful law.

And what it says is the number of requests that are in your system, either being queued or being served, whatever it is, the total in your system, the total number of requests is equal to the rate at which requests arrive into your system and multiplied by the average time requests spend in the system. If every request spends and on average W amount of time in the system and the requests arrive at the rate of R into your system, then the number of requests in your system at any point of time is R * W.

This is very intuitive. For example, requests arrive at the rate of 100 requests per second, 100 requests per second are coming in. And each request is taking, 2 seconds in the system. So, over a period of 2 seconds, 200 requests have come in. And of course, by the end of 2 seconds, most of these requests start to leave. So, therefore, at any point of time, if you see there will be 200 requests, this 100 * 2, 200 requests queued up in your system.

Why? Because the requests that came in this second, request that came in the previous second, they are all still being processed. Before the request came in, they would have left anyways by now. So, therefore, this Little's law gives you an intuitive way to calculate how many requests do I expect to be lying around in my system.

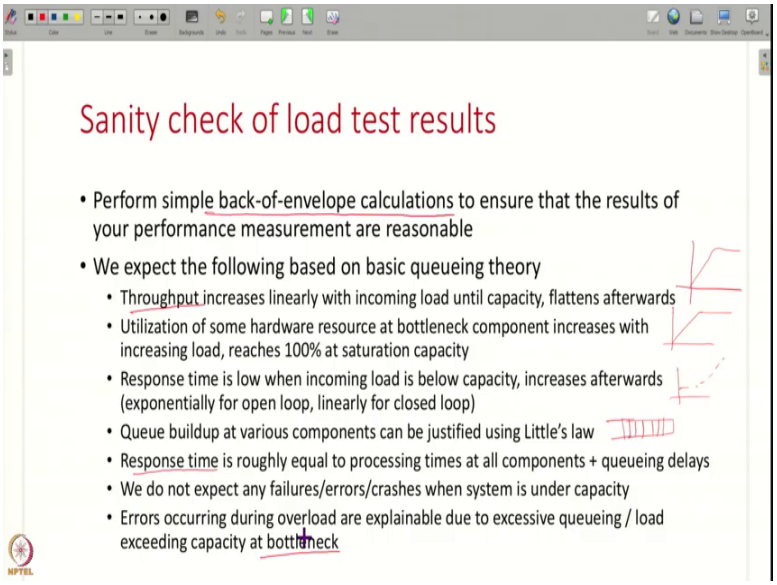
And this is used for many things, if you want to suppose you measure the amount of time your system is taking to process requests, then you can find you can calculate what should be the queue size, there is a shared buffer between different threads in a pipeline or different processes in a pipeline, there is a shared buffer what should the size of this buffer be? I will see how many requests are coming in, how long are requests pending in the system.

Therefore, I know that there are so many requests at any point of time in the system, I will make that my queue size. So, given the waiting time in the system, you can calculate queue size, given the queue size, you can also estimate the waiting time. If this is my queue size, and this is the rate of requests coming in. This is what I expect my waiting time to be and if the waiting time is more than that, then you will think why is the waiting time so high.

So, this is just a way for you to connect up many different metrics in your system and see if they all add up together or not. So, Little's law is very powerful. It is very intuitive. It is

widely applicable in real life. If you look around many places, like people waiting at a counter for tickets or in any queue, the rate at which people are coming in, how long they are spending in the queue, you multiply both of them you will roughly be able to estimate how many people are there in the queue. So, this is a very general formula. So, finally, we come to the summary of what all we have seen in this lecture.

(Refer Slide Time: 30:07)



Anytime you run a load test, you have to do some simple back of the envelope calculations to ensure that the results of your load test they are all reasonable. And from basic queueing theory, this is what you expect. Your throughput graph should basically increase linearly with increasing load and should flatten out. And the utilization of your hardware resource should increase with increasing load and then it should reach 100 percent at saturation capacity.

Your response times will be low initially, but they will start to increase as you reach capacity and go beyond capacity. This increases much more rapid for open loop systems and much more controllable for closed loop systems. And your queues will build up at various components. And anytime you see some queue building up between components, measure the queue size and see if this queue size can be justified using Little's law.

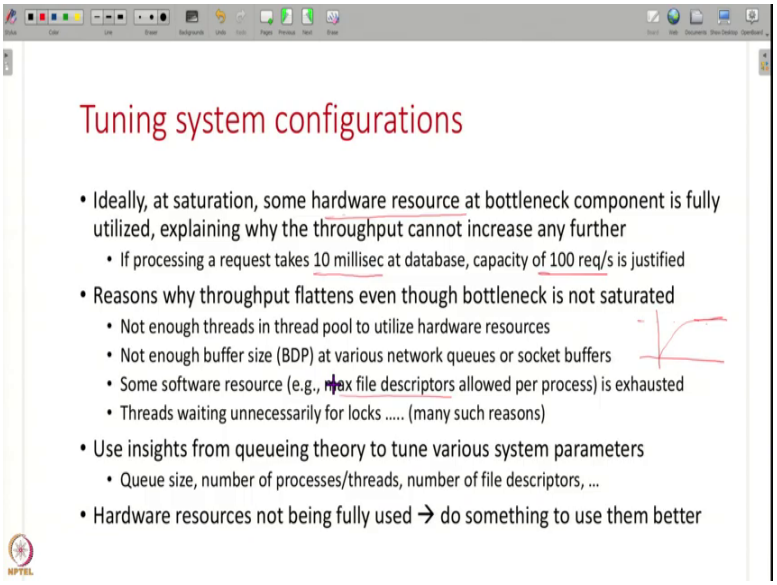
And if you do not have enough of a buffer space to handle all the expected queuing, you should increase your buffer space. So, use Little's law to check all the queue sizes. And then see if the response time is roughly equal to the processing time plus queuing delays, you know what the queuing delays should be from Little's law. And you know what the processing time should be from your application?

Is the sum of these two equal to response time? Or is my time getting wasted at some other place that I do not know about. All of these things you should check, then you should check that there are no failures when the system is under capacity. When you are operating below capacity, all requests should be satisfied. There is no reason for failures.

And after overload, after you cross the capacity of the system, you should see queue building up at the bottleneck component and errors occurring due to this queue build up. If you see errors occurring, some other component that is not fully utilized, then you cannot say that error is due to overload.

So, just because a server crashes does not mean it is due to overload you should see is the crash due to this excessive queue build up at the bottleneck component or not. So, all of these are sort of some of the sanity checks that you can do on your load test results. And once you do these sanity checks, you can also tweak your system configuration a little bit. If you see that something does not make sense.

(Refer Slide Time: 32:20)



For example, at saturation, you will expect that some hardware resource is the bottleneck, some hardware resources fully utilized and therefore, that is the reason why you cannot do any more work at your bottleneck. For example, if processing the request takes 10 millisecond, you know that your database cannot handle more than 100 requests per second. And expecting it to get a throughput of 200 requests per second is not justified.

But if your throughput is flattening your database, you measure its throughput, its throughput is only flattening at 50 requests per second, then you know that and the CPU is also not fully utilized then you know that something is wrong, this is not a correct behavior. This could be happening for many reasons, for example, you do not have enough threads in your thread pool we have seen this.

If there are no enough threads in your thread pool all threads are somehow waiting for IO your CPU is idle. In such cases, your bottleneck CPU utilization will be low your throughput will be low and that is not the correct capacity you are measuring. You should increase the number of threads. You should increase various buffer sizes. You should increase various queue sizes in accordance with Little's law, we have derived all of these things.

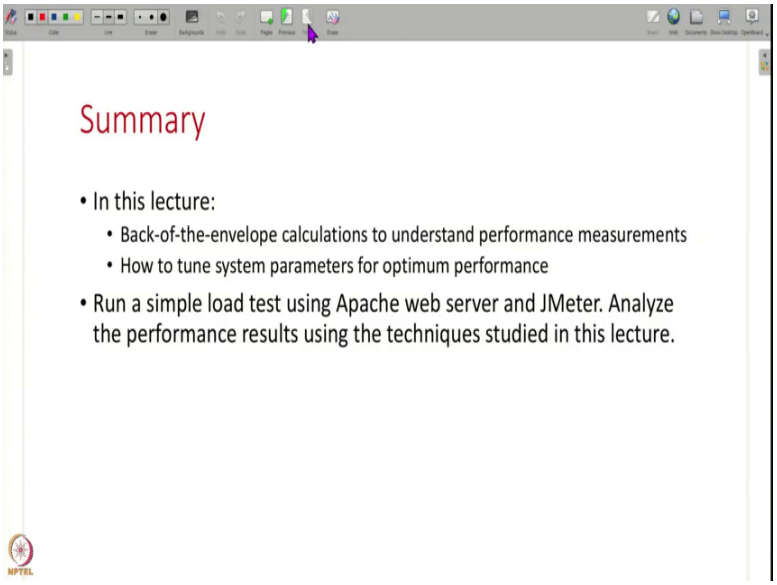
So, you should increase all of the system resources such that you are fully utilizing your hardware. If some software resource like max file descriptors or something else is exhausted increase that limit and if threads are waiting for lock C, if you can reduce unnecessary locks. So, all of these things are system level changes, software changes that you can make, so that your system performance can improve.

So, you should always make these changes first and ensure that the final bottleneck is some hardware resource. If your CPU is fully utilized or some hardware resource is fully utilised then you can say okay fine, I have extracted the maximum performance possible from my system I cannot do anything more. But if your system performance is lower due to not having enough threads or not having enough file descriptors, then these are trivial issues.

You should simply increase that software resource and then as more threads come in your CPU will be fully utilized. So, that is why use insights from queueing theory to see is my performance numbers making sense. Are my queue size is correct? Are the number of threads in my thread pool correct?

Are the number of file descriptors are there enough file descriptors to handle all the waiting requests in my queue? So, all of these you can somehow connect it up one way or the other to Little's law and to ensure that all of these thinks are sized correctly. So, that at the end, some hardware resources fully utilized and your system has reached a performance bottleneck.

(Refer Slide Time: 34:56)



So, this is all I have for this lecture. In this lecture, I have told do quite a few ways of doing a back of the envelope, simple calculations estimates to understand your performance measurements to see if they make sense or not. And some simple ways to tune some system parameters like threads in a thread pool, buffer sizes, queue sizes, in order to optimize your system performance.

So, as a practical exercise, if you have tried out, running a load test with a patchy JMeter and a web server, you should use some of the techniques we have studied in this lecture, for example, to make sense of the performance measurements that you have seen. So, thank you all. That is all I have for this lecture. We will continue on the topic of performance in the next lecture. Thanks.