Design and Engineering of Computer Systems

Lecture 32: Performance analysis

Mythili Vutukuru

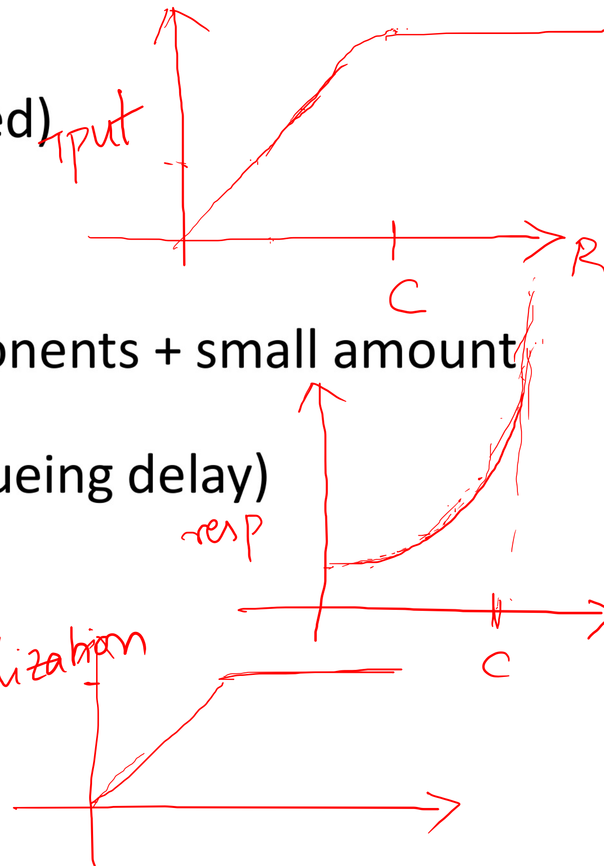
IIT Bombay

Performance analysis

- Workflow in building computer systems
  - Design and develop system components
  - Run a load test, measure performance (throughput, response time, errors, ..)
  - Understand if the performance is reasonable or can be improved
  - Tune system to improve performance if required, measure performance again
  - Iterate until satisfied with performance
- In this lecture: a very high-level overview of performance analysis
  - What do we expect the performance measurements to look like? Use simple back-of-the-envelope calculations to estimate expected performance
  - Are the measured performance numbers in line with our expectations?
- We only cover high level concepts and intuitions, no theory / math
  - Take a course on queueing theory for a more rigorous treatment of the topic

Open loop load testing

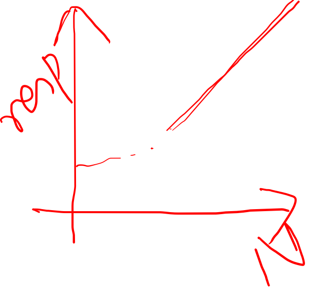
- Consider a system with capacity = C req/s (service demand at bottleneck=~ 1/C)
- Open loop load test: varying rate of requests into system (R req/s)
- Measured throughput
  - If R<C, throughput = R (all incoming requests will be eventually served)
  - For R>=C, throughput will flatten at C req/s
- Measured response time
  - If R<C, response time is fairly low (time to serve request at all components + small amount of queueing due to randomness of arrivals)
  - As R approaches C, response time increases exponentially (high queueing delay)
  - If R>>C, response time goes to infinity, requests fail, server crash
- Utilization of bottleneck component
  - If R<C, utilization is approx. R/C (proportional to incoming load)
  - If R>=C, some hardware resource at bottleneck is at 100% utilization

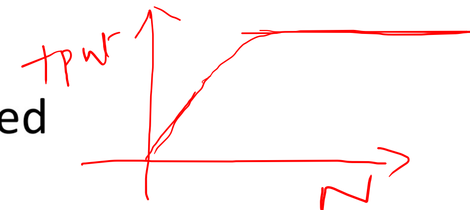


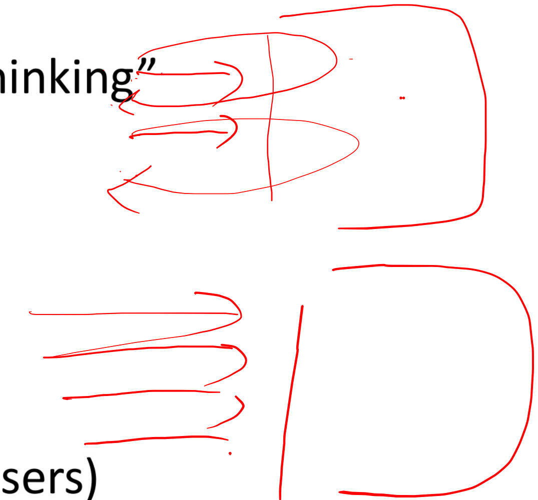
Closed loop systems

- Consider a system with capacity = C req/s (service demand at bottleneck=~ 1/C)
- Closed loop load test: varying number of concurrent users (N)
  - Think time = time between user getting response and making next request
  - Turn around time of a user = response time from system + think time

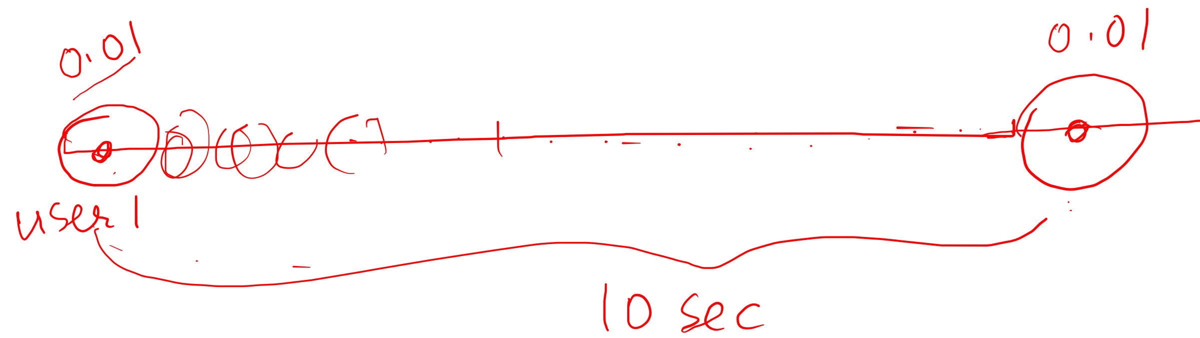
- What happens for low values of N?
  - If no think time, small number of users can quickly generate requests back-to-back to keep the bottleneck fully utilized
  - With non-zero think time, bottleneck component idle when all users are "thinking"
  - Low throughput, low response time, system poorly utilized
- What happens for large values of N?
  - Bottleneck has enough work all the time, fully utilized
  - Throughput flattens at saturation capacity C req/s
  - Response time increases due to queueing of user requests at bottleneck
  - Increase in response time somewhat linear with N (queueing due to extra users)







Optimum value of N

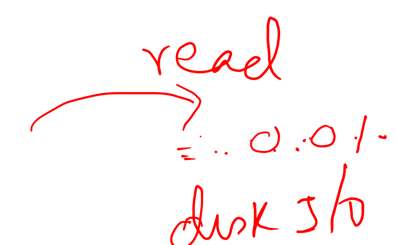


- What is the optimum value N* for number of concurrent users?
  - Minimum load level to fully saturate the system, but not much queueing
- Example to understand how to compute N*
  - Consider system with capacity 100 req/s (service demand at bottleneck = 0.01 seconds)
  - User generates request every 10 seconds (turnaround time = response time + think time)
  - One user keeps the bottleneck component busy for 0.01 sec for every 10 seconds
  - Approximately 10 / 0.01 = 1000 users needed to keep bottleneck fully occupied
- N* = turnaround time / service demand at bottleneck
- If number of concurrent users in load test N < N*
  - System under utilized, throughput below capacity
- If number of concurrent users in load test N >= N*
  - Bottleneck fully utilized, throughput flattens, response times increase
- This intuition of closed systems useful in many other scenarios as well…

turnaround
service demand

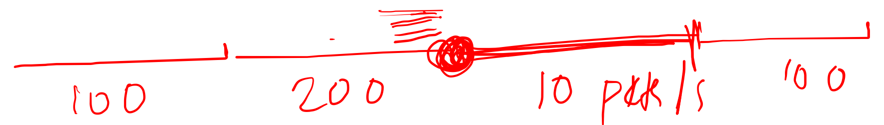
Optimum number of threads in thread pool

- Consider a multithreaded server with a thread pool of workers
  - Master thread places new clients/requests in shared queue
  - Worker threads in thread pool retrieve requests one by one and process
  - Worker thread can block multiple times to service one client
  - Too low number of worker threads cannot efficiently use CPU cores



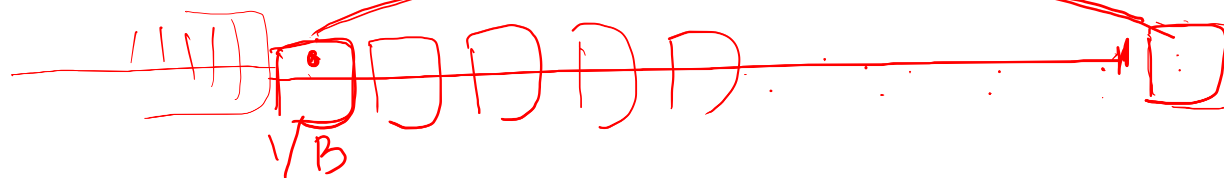
- Min no. of threads in thread pool to fully utilize a single CPU core?
  - Suppose each worker thread performs 0.01 seconds of computation on CPU to process client request (service demand) and 1 second waiting for I/O before running on CPU again (turnaround time)
  - Optimum number of threads = turnaround time / service demand = 100

- Thread pool of worker threads is essentially a closed loop system!
  - Thread uses bottleneck (CPU), waits for some time (I/O) and comes back to CPU



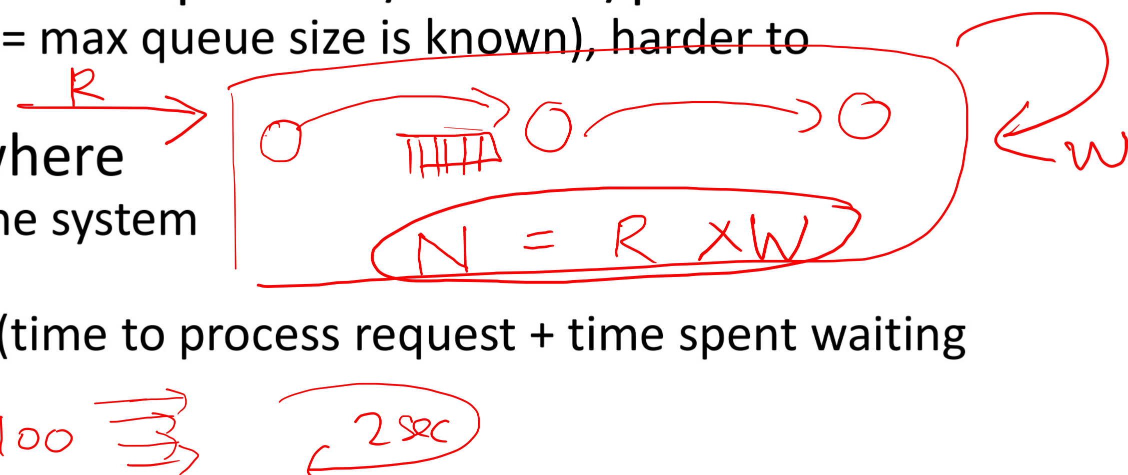
Optimum size of sliding window

- Stop-and-wait protocols: sender sends packet, waits for ACK, sends next packet
- Sliding window protocols: sender sends a window of packets before waiting for ACK
  - Bottleneck (slowest) link on network path is better utilized, avoids too much waiting
- What is optimum number of packets in a window?
  - Too small window size → sender waits for ACKs often, bottleneck link is under-utilized
  - Too large window size → unnecessary queueing at bottleneck router, packet drops
- Optimum number of packets = bandwidth delay product. Why?
  - Bottleneck is transmission on slowest link, queue builds up at outgoing link of bottleneck router
  - Suppose bandwidth of bottleneck is B packets per sec, each packet needs 1/B seconds to be transmitted at bottleneck link (service demand)
  - ACK received after RTT, so packets in a window "come back again" after RTT (turnaround time)
  - Min number of packets to fully utilize bottleneck link = RTT / (1/B) = RTT * B



Estimating queue sizes: Little's law

- How to estimate size of queues between various components/threads/processes?
  - Trivial in closed loop system (max concurrent users = max queue size is known), harder to estimate for open loop systems
- Little's law for open loop systems: $N = R * W$, where
  - N = expected number of requests being served in the system
  - R = rate of arrival of requests
  - W = average time spent by a request in the system (time to process request + time spent waiting in various queues)



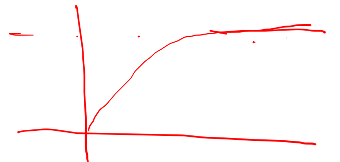
- Example to illustrate Little's law
  - Requests arrive at 100 req/s, each request takes 2 seconds for waiting+processing in system
  - Approx. 200 requests in system at any point of time (being served or waiting)
- Applications of Little's law: configuring queue sizes, sanity check of response times
  - Given measured time in system (W), find optimal queue size (N)
  - Given queue size (N), estimate how much time requests should spend in system (W)
- Very popular, widely applicable, irrespective of characteristics of system and traffic

Sanity check of load test results

- Perform simple back-of-envelope calculations to ensure that the results of your performance measurement are reasonable
- We expect the following based on basic queueing theory
  - Throughput increases linearly with incoming load until capacity, flattens afterwards
  - Utilization of some hardware resource at bottleneck component increases with increasing load, reaches 100% at saturation capacity
  - Response time is low when incoming load is below capacity, increases afterwards (exponentially for open loop, linearly for closed loop)
  - Queue buildup at various components can be justified using Little's law
  - Response time is roughly equal to processing times at all components + queueing delays
  - We do not expect any failures/errors/crashes when system is under capacity
  - Errors occurring during overload are explainable due to excessive queueing / load exceeding capacity at bottleneck

Tuning system configurations

- Ideally, at saturation, some hardware resource at bottleneck component is fully utilized, explaining why the throughput cannot increase any further
  - If processing a request takes 10 millisec at database, capacity of 100 req/s is justified
- Reasons why throughput flattens even though bottleneck is not saturated
  - Not enough threads in thread pool to utilize hardware resources
  - Not enough buffer size (BDP) at various network queues or socket buffers
  - Some software resource (e.g., max file descriptors allowed per process) is exhausted
  - Threads waiting unnecessarily for locks ….. (many such reasons)



- Use insights from queueing theory to tune various system parameters
  - Queue size, number of processes/threads, number of file descriptors, …
- Hardware resources not being fully used → do something to use them better

Summary

- In this lecture:
  - Back-of-the-envelope calculations to understand performance measurements
  - How to tune system parameters for optimum performance
- Run a simple load test using Apache web server and JMeter. Analyze the performance results using the techniques studied in this lecture.