# CS1632, LECTURE 13: PERFORMANCE TESTING

Wonsun Ahn

# What do we mean by Performance?

- If you look it up in a dictionary …
  - *Merriam-Webster*: the ability to perform
    - Dictionaries can be self-referential like this ☹
  - *Cambridge*: how **well** a person or machine does a piece of work
  - *Macmillan*: the **speed** and **effectiveness** of a machine or vehicle

- In software QA: it is a **non-functional** requirement (**quality attribute**)
  - Narrow sense: **speed** of a program
  - Broad sense: **effectiveness** of a program
  - In this chapter, we will refer to performance in the broad sense

# But Even Speed is Hard to Define

- Even performance in the narrow sense (speed) is hard to define
- Speed for a *web browser*
  - How quickly a website responds to user interactions
    (Page loads, button clicks, dragging, typing …)
  - Responsiveness is measured in average **response time**
- Speed for a *web server*
  - How quickly a server responds to a page request is a part of it, yes.
  - More importantly, pages served per second (a.k.a. **throughput**)
  - As long as response time is less than a threshold (say, < 100 ms),
    web server performance is measured by throughput, not response time
- We need more than one metric to quantify performance

# Performance Indicators

- Quantitative measures of the performance of a system under test
- Examples (in the narrow sense, speed):
  - How long does it take to respond to a button press? (*response time*)
  - How many users can the system handle at one time? (*throughput*)
- Examples (in the broad sense)
  - How long can the system go without a failure? (*availability*)
  - How much CPU does a standard query on the database take up? (*utilization*)
  - How much memory does the program use in megabytes? (*utilization*)
  - How much energy does a program use per second in watts? (*utilization*)

# Key Performance Indicators (KPIs)

- **KPI**: a performance indicator important to the user
- Select only a few KPIs that are really important
  - Those that are indicative of success or failure of your software
  - e.g. miles-per-gallon *should* be a KPI for a hybrid-electric car
  - e.g. miles-per-gallon *should not* be a KPI for a formula-1 race car
  - Being indiscriminate means important performance goals will suffer

- **Performance target**: quantitative measure that KPI should reach ideally
- **Performance threshold**: bare minimum a KPI should reach
  - Bare minimum to be considered production-ready
  - Typically more lax compared to performance target

# KPI / Performance Target / Performance Threshold

- Let’s say you are developing requirements for a web application
- Here is an example KPI / Performance Target / Performance Threshold
  - KPI: response time
  - Performance target: 100 milliseconds
  - Performance threshold: 500 milliseconds
- Another example KPI / Performance Target / Performance Threshold
  - KPI: throughput
  - Performance target: 100 user requests / second
  - Performance threshold: 10 user requests / second

# Performance Indicators: Categories

- There are largely two categories of performance indicators
- Service-Oriented
- Efficiency-Oriented

# Service-Oriented Performance Indicators

- Measures how well a system is providing a service to the users
  - Measures end-user experience
  - Oblivious to the internals (like blackbox testing)
- Two subcategories:
  - Response Time
    - How quickly does the system respond to a user request?
  - Availability
    - What percentage of time can a user access the services of the system?

# Efficiency-Oriented Performance Indicators

- Measures how well a system makes use of computational resources
  - Measures the “internals” of a system
  - May indicate *why* an end-user is having a good / bad experience
- Two subcategories:
  - Utilization
    - How much compute resources does the system use?
  - Throughput
    - How many requests can be processed in a given amount of time?

# Service-Oriented vs. Efficiency-Oriented

- In the end, user experience is what's important (service-oriented)
- But efficiency-oriented indicators directly impact user experience
  - App *utilizes* too much CPU time → *response time* suffers
  - App *utilizes* too much memory and you run out → *availability* suffers
  - App *throughput* cannot handle a surge of requests → *response time* suffers
  - App *throughput* is chronically low → *availability* suffers
- Service-oriented indicators show *whether* users may be dissatisfied
- Efficiency-oriented indicators show *why* users may be dissatisfied
  - And point to a solution to the problem (e.g. install extra memory)

# Testing Service-Oriented Performance Indicators

Response Time / Availability

# Testing Response Time

- Easy to do!
  - Do something
  - Click “start” on stopwatch
  - Wait for response
  - Click “stop” on stopwatch
  - Write down number on stopwatch!
- Any problems with this approach?

# Problems with Response Time Manual Testing

1. Impossible to measure sub-second response times
2. Impossible to measure responses not visible to end-user
3. Human error
4. Time-consuming
5. Probably the most boring thing a person can do

☛ Performance testing relies heavily on automation and statistics.

# Statistics? Why?

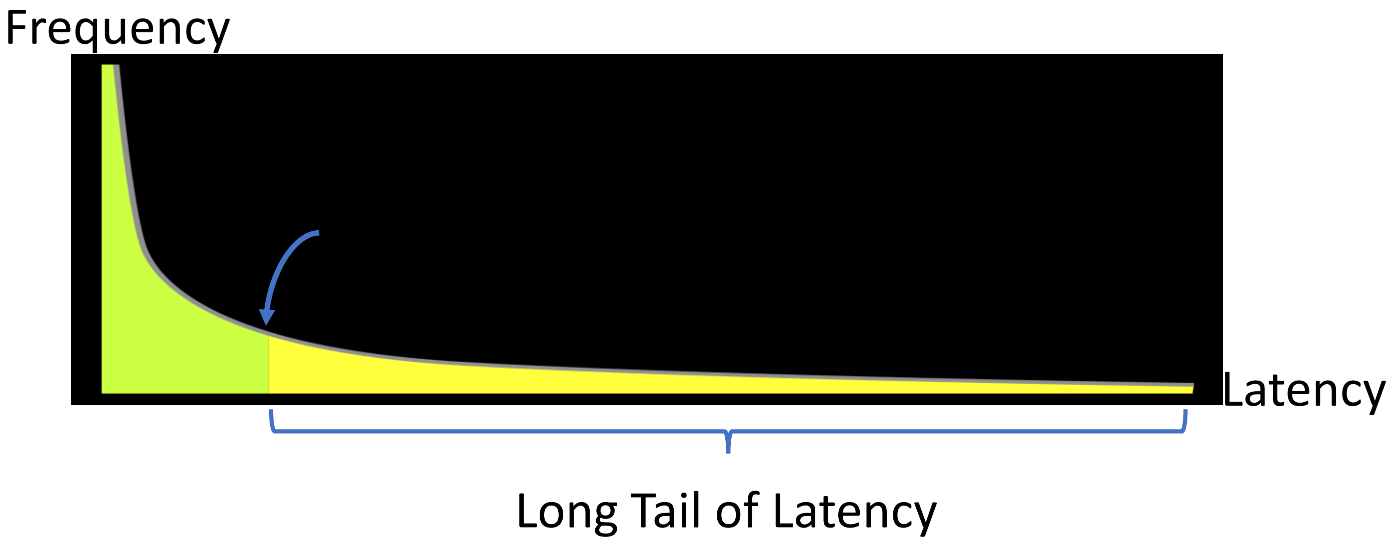
- You should never trust a single result in performance testing
  - Always try multiple times to get the average value
  - Also look at min/max values to check for large variances
- Why? So many things can go wrong in a single test run:
  - Other processes taking up CPU time
  - Having to swap in memory pages from hard disk
  - Network bandwidth occupied by some other machine
- A single test run is almost worthless.

# Performance testing is a science

- Eliminate all variables OTHER THAN THE CODE UNDER TEST
  - Make sure you are running on the same hardware configuration
  - Make sure you have identical Library / OS / device driver versions
  - Kill all processes in the machine other than the one you are testing
  - Remove all periodic scheduled jobs (e.g. anti-virus that runs every 2 hours)
  - Fill memory / caches by doing several warm up runs of app before measuring
- Even after doing all of this, there is still going to be variability
  - Try multiple times to get a statistically significant result

# The Dreaded Long Tail of Latency

- Typically, this is the type of latency distribution you will get




- Often the “long tail” is more important than average latency
  - Could be caused by external variable you failed to control (e.g. periodic job)
  - Could be caused in a distributed system with asynchronous components
- Many runs are required not only to accurately measure the average, but also to detect the length and height of the “long tail”

# Kinds of events to test for response time

- Time for calculation to take place
- Time for character to appear on screen
- Time for image to appear
- Time to download
- Time for server response
- Time for page to load

# What kind of time should we measure?

- real time: “Actual” amount of time taken (wall clock time)
- user time: Amount of time user code executes on CPU
- system time: Amount of time kernel (OS) code executes on CPU
- total time : user time + system time

- real time ≠ total time
  - real time = total time + idle time
  - idle time: time app is idling (not executing on CPU) waiting for some event (where event can be an I/O event, synchronization event, interrupt event, …)

# Example

- time command in Unix
  - time java Foo
  - time curl http://www.example.com
  - time ls –l
- Windows PowerShell has something similar
  - Measure-Command { java Foo –wait }

# What kind of time do we care about?

- For service-oriented testing
  - Users almost always care about real ("wall clock") time
  - Measure of how long user has to wait to get a response

- For efficiency-oriented testing
  - Developers also care about total, user, and system time
  - Total time: total CPU utilization
  - User time: CPU utilization directly caused by app code
    - May indicate a need to optimize algorithm or use efficient data structure
  - Kernel time: CPU utilization in OS as a result of app system calls or interrupt handling (e.g. page faults)
    - May indicate a need to optimize system calls or tune the OS

# Rough response time performance targets

< 0.1 S : Response time required to feel that system is instantaneous

< 1 S : Response time required for flow of thought not to be interrupted

< 10 S : Response time required for user to stay focused on the application

- Taken from “Usability Engineering” by Jakob Nielsen, 1993

*Things haven't changed much since then!*

# Testing availability

- Availability - often referred to as uptime
  - What percentage time is the system accessible to the user?

- Often guaranteed in a SLA (service-level agreement)
  - “I am a web host.  I guarantee you that you and your users will be able to access your service 99% of the time in a given month.”

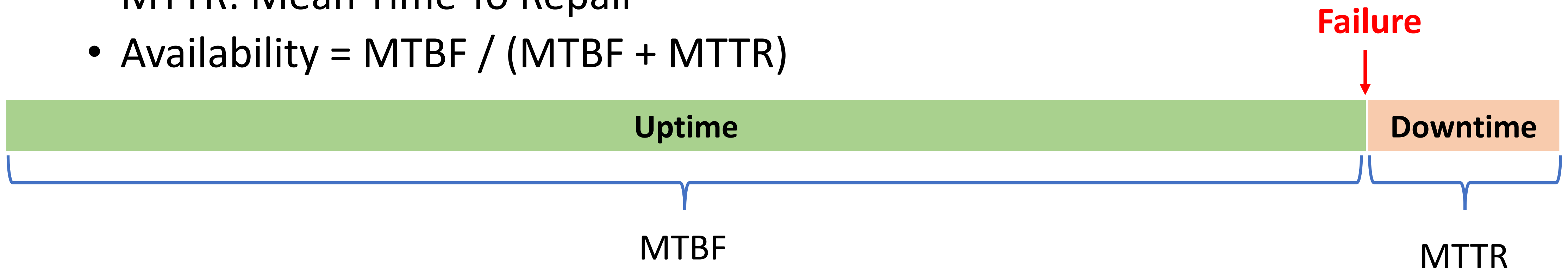
# Nines

- Uptime is often expressed in an abbreviated form as 9’s (e.g. 3 nines, 5 nines etc)
- Refers to how many 9’s start out the percentage of time available
  - 1 nine: 90% available (36.5 days of downtime per year)
  - 2 nines: 99% available (3.65 days of downtime per year)
  - 3 nines: 99.9% available (8.76 hours of downtime per year)
  - 4 nines: 99.99% available (52.56 minutes of downtime per year)
  - 5 nines: 99.999% available (5.26 minutes of downtime per year)
  - 6 nines: 99.9999% available (31.5 seconds of downtime per year)
  - 9 nines: 99.9999999% available (31.5 ms of downtime per year)

# How to test?

- Difficult – not feasible to run a few "test years" before deploying
- Modeling system and estimating uptime is the only feasible approach

- Metrics to model
  - MTBF: Mean Time Between Failures
  - MTTR: Mean Time To Repair
  - Availability = MTBF / (MTBF + MTTR)



# Measuring MTTR and MTBF

- Measuring MTTR is easy
  - Average time to reboot a machine
  - Average time to replace a hard disk

- Measuring MTBF is hard
  - Depends on how much the system is stressed
  - Depends on the usage scenario
  - Measure MTBF for different usage scenarios
    → Calculate a (weighted) average of MTBF for those scenarios

# Measuring MTBF with Load Testing

- Load testing:
  - Given a load, how long can a system run without failing?
  - Load is expressed in terms of concurrent requests / users
- Kinds of load testing:
  - Baseline Test - A bare minimum amount of use, to provide a base
  - Soak / Stability Test – Typical usage for extended periods of time
  - Stress Test – High levels of activity typically in short bursts
- Estimate MTBF based on test results and historical load data
  - E.g. if 90% of time is typical usage, 10% of time is peak usage,
    MTBF = Soak Test MTBF * 0.9 + Stress Test MTBF * 0.1

# MTBF is Not Only about Your Software

- For true availability numbers, also need to determine:
  - Likelihood of hardware failure
  - Likelihood of OS crashes
  - Likelihood of data center cooling system failures
  - Planned maintenance
  - etc.

# Things can still go wrong

- Even with all this work, things go wrong
- Many major service providers “breach” their SLAs in a given month
  - Including Microsoft Azure and Amazon Web Services
  - Usually, money is refunded automatically

# Developing a Service-Oriented Test Plan

- Think from a user's perspective!
  - How fast do I expect this to be on average?
  - Are large variances in response time allowed?
  - How often do I expect this to be available?

# Determine KPIs, Targets, and Thresholds

- Example:
  - KPI: Average page load time
    Target: less than two seconds
    Threshold: less than five seconds
  - KPI: Max page load time
    Target: less than five seconds
    Threshold: less than ten seconds
  - KPI: Availability of system
    Target: greater than 99.9%
    Threshold: greater than 99%

# Think about contingency plans!

- What if performance requirements aren't met?
- What if they can't be?
- What if they can be, but at a high cost in time/resources?
- etc.

# Testing Efficiency-Oriented Performance Indicators

Throughput / Resource Utilization

# Why use Efficiency-Oriented Indicators?

1. More granular than service-oriented indicators
   - Easier to pin down where exactly the performance problem lies
2. Possible to determine how problem can be solved:
   - Software modification (better algorithm, better data structures, etc.)
   - OS tuning (upgrade OS, tune paging algorithm, tune I/O device driver, etc.)
   - Hardware modification (scaling hardware, upgrading hardware, etc.)
3. It’s the language developers understand

# Example

- Rent-A-Cat has a Web API showing which cats are available to rent
- Service-oriented testing shows that it takes on average 5 seconds to respond to list-sorted-cats, violating performance target of 1 second
- After efficiency-oriented testing, you see that on list-cats request …
  - Network bandwidth usage is 1%
  - Disk bandwidth usage is 3%
  - Memory usage is steady before / after
  - But the CPU is pegged at 99% for five seconds
- Where would you look for solutions to this issue?

# Possible issues / Ameliorations

- If CPU utilization is the issue:

1. Cats sorted with insertion sort – Use better sorting algorithm
2. A lot of time used in garbage collection – Tune garbage collector
3. Everything running on one CPU - Spread work to other CPUs
4. If none of the above – Maybe it's time you upgrade your server hardware

- If network utilization is the issue:

1. Lengthy HTML / JavaScript – Minify source code to reduce network bandwidth
2. Request just too popular - Cache sorted listings in a proxy server

# “Premature optimization is the root of all evil” – Donald Knuth

- Do service-oriented testing first
  - If key performance indicators hit targets, why bother?
  - Only drill down with efficiency-oriented tests if otherwise

# Testing Throughput

- Throughput
  - Number of events a system can handle in a given timeframe
- Examples:
  - Packets per second (that can be handled by a router)
  - Pages per minute (that can be served by a web server)
  - Number of concurrent users (that a game server can handle)

# How’s that different from service-oriented testing?

1. A given user doesn't care about the number of users who are on the system, just about what it means for them
   - As in, do I notice any degradation in response time?
   - As in, do I notice any disruption in service?

2. More granular
   - Describes a specific system behavior not visible to user (e.g. pages served / sec, database requests / sec, etc.)
   - Can measure performance of each subcomponent of a system (web server, database server, router, etc.)

# Load testing

- Variations on load testing can be used to test throughput
  - Increase number of events until system crashes
  - Increase number of events until response time falls below threshold
  - Etc.

- Basically measure the maximal load that system can handle
  - Without degrading Quality of Service (QoA)

# Testing Utilization

- *You need tools for this*
  - Unless you can tell by the sound of your fan exactly how many operations your program is running on the CPU

# Tools

- General purpose
  - Windows Systems – Task Manager, perfmon
  - OS X - Activity Monitor or Instruments, top
  - Unix systems - top, iostat, sar

- Program-Specific Tools

# Resources watched by general purpose tools

- CPU Usage
- Threads
- Physical Memory
- Virtual Memory
- Disk I/O
- Network I/O

# You can get more specific

- Disk cache misses – may be the reason high disk I/O utilization
- CPU cache misses – may be the reason for high memory bandwidth
- File flushes – frequent forced flushes may contribute to high disk I/O
- Outbound Network Packets discarded – network bandwidth issue?
- IPv6 Fragments Received/Sec
- ACK msgs received by Distributed Routing Table

# General purpose tools only give general info

- Lots of memory being taken up…
  - …but by what objects / classes / data?
- Lots of CPU being taken up…
  - …but by what methods / functions?
- Lots of packets sent…
  - …but why?  And what's in them?

# Tools

- General purpose
  - Windows Systems – Task Manager, perfmon
  - OS X - Activity Monitor or Instruments, top
  - Unix systems - top, iostat, sar

- Program-Specific Tools

# Program-Specific Tools

- Protocol analyzers
  - e.g., Wireshark or tcpdump
  - See exactly what packets are being sent/received
- Profilers
  - e.g. JProfiler, VisualVM, gprof, and many, many more
  - See exactly what is in memory
  - What methods are being called and how often
  - What objects/classes have been loaded

# To Wrap it Up …

# From service-oriented test to solution

- Response test: “Our app is slower that we would like”
- Utilization test:
  - “CPU utilization is high”
  - “Memory utilization is high”
  - “I/O utilization is high due to swapping between memory and hard disk”
- Profiling:
  - “The garbage collector is running way too often, taking up CPU time”
  - “Memory is filled with ConnectionCounter objects”
- Solution:
  - “Remove memory leak on ConnectionCounter objects”

# Fixing performance issues

1. Service-Oriented Testing:
   - Determine if performance is a problem
   - If it’s not, let sleeping dogs lie!
2. Efficiency-Oriented Testing:
   - Track down from top-level to low-level
   - Start from general purpose on to program specific tools
3. *Keep track of performance throughout versions*
   - Performance testing should be part of your regression suite!

# Food for thought

- Q: Aren't throughput and utilization similar concepts?
  - If you have low utilization, don't you automatically have high throughput?
  - If you use resources efficiently, you will process jobs faster, no?
- A: Not necessarily! You can have low utilization *and* low throughput
  - When you have high utilization in another resource, which is the bottleneck
    e.g. you may have low CPU utilization but I/O bandwidth is the bottleneck
  - When you process jobs sequentially, and not in parallel
    e.g. if CPU utilization is at 25%, you have room to process 4 jobs in parallel!
  - When your app is latency-limited rather than bandwidth-limited
    e.g. you have a long latency network and you often sit idle waiting for a packet

# Now Please Read Textbook Chapter 19