CS1632, LECTURE 14: PERFORMANCE TESTING

Wonsun Ahn

What do we mean by Performance?

- *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
    - What do we mean by well?
    - What do we mean by effectiveness?
    - What do we mean by speed?

- One thing is for certain: performance is a **non-functional** requirement

Performance can mean different things

- Let's say we are using speed as the measure of performance
- Even speed can mean different things, depending on context
- Speed for a *web browser*
  - How quickly a page loads on the screen
  - How fast the web page responds to a button press
- Speed for a *web server*
  - How quickly a page is serviced to a web browser is a part of it, yes.
  - More importantly, page services per second (a.k.a. **throughput**)
  - As long as page service time is less than a threshold (say, < 100 ms), how quickly a page is serviced on average is less important

Performance can mean different things

- Speed for a *video game*
  - Real-time **responsiveness** is paramount
  - A button press must immediately translate to action
    (Even when game AI was in the middle of thinking up some grand strategy)
- Speed for a *weather forecasting program*
  - Responsiveness is unimportant
  - As long as program gives an answer within a reasonable timeframe
    (As in, still useful for the purposes of forecasting)
  - More important: amount of CPU time required for an accurate answer
    (a.k.a. **utilization**)

Performance indicators

- Quantitative measures of the performance of a system under test
- Examples:
    - How long does it take to respond to a button press?
    - How many users can access the system at one time?
    - How long can the system go without a failure?
    - How much CPU does a standard query on the database take up?
    - How much memory does the program use in megabytes?
    - How much energy does a program use per second in watts?

Two categories of performance indicators

- Service-Oriented
  - Measures how well a system is providing a service to the users
  - Measured from an end-user's point of view
    - Only cares about what’s visible to the user (similarly to blackbox testing)
- Efficiency-Oriented
  - Measures how well system makes use of computational resources
  - Measured from a more “system” perspective
    - Cares about how efficient the system is behind the scenes
    - E.g. how many servers do I have to install to provide that user experience

Service-Oriented indicators have two subcategories

- Availability
  - How available is the system to the user?
  - What percentage of the time can they access it?
- Response Time
  - How quickly does the system respond to user input?
- *Note: Both are very visible to the user, hence service-oriented*

Efficiency-Oriented indicators have two subcategories

- Throughput
  - How many events can be processed in a given amount of time?
- Utilization
  - What percentage or absolute amount of compute resources are used?
- *Note:*
  - *Describes an aspect of system efficiency not directly visible to end-user*
  - *But bad efficiency is often the cause of bad service*
  - *E.g. bad throughput is often the cause of bad availability*
  - *E.g. bad utilization is often the cause of bad response time*

Testing performance

- First, you must decide on Key Performance Indicators (KPIs)
  - KPIs: Subset of performance indicators that are important to you
- Second, you must decide on performance targets
  - Targets: Quantitative values that the KPIs should reach ideally
- Third, (optionally) you can decide on performance thresholds
  - Thresholds: bare minimum to be considered production-ready
  - Typically more lax compared to targets

KPI / Performance Target / Performance Threshold

- Let’s say you are developing a web application.
- Here is an example KPI / Performance Target / Performance Threshold
  - A key performance indicator (KPI) might be response time
  - A performance target may be 1 second
  - A performance threshold may be 3 seconds
- Another example KPI / Performance Target / Performance Threshold
  - A key performance indicator (KPI) might be throughput
  - A performance target may be 20 user requests / second
  - A performance threshold may be 10 user requests / second

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

☛ 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
  - Kill all processes in the machine other than the one you are testing
  - Remove all periodic jobs from your cron tab
  - Fill memory / caches by doing several warm up runs of app before measuring
  - Make sure you are running on the same hardware configuration
  - Make sure you have identical Library / OS / device driver versions

- Even after doing all of this, there is still going to be variability
  - Try multiple times to get a statistically significant result

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
- system time: Amount of time kernel code executes
- total time : user time + system time

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 care about total, user, and system time
  - Measure of CPU utilization in user / kernel code

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 (and not go re-load Reddit)

  - 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?

- Often difficult – most managers won’t let you run a few “test years” before deploying it for real

- Modeling system and estimating uptime is the only feasible approach

Determine values for model with load testing

- Load testing:
  - How many concurrent users can system handle and for how long?
- Kinds of load testing:
  - Baseline Test - A bare minimum amount of use, to provide a base
  - Soak / Stability Test - Leave it running for an extended period of time, usually at low levels of usage
  - Stress Test – High levels of activity typically in short bursts
- Estimate availability based on test results and historical load data

Reality

- For true availability numbers, also need to determine:
  - Likelihood of hardware failure
  - Likelihood of program bugs leading to crashes
  - 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!
  - What things matter to me, speedwise?
  - How fast do I expect this to be?
  - Are large variances in response time allowed?
  - How often do I expect this to be available?

Determine kpis, targets, and thresholds

- Example:
  - Average page load time – Target: less than two seconds, Threshold: less than five seconds
  - Max page load time – Target: less than five seconds, Threshold: less than ten seconds
  - 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 do efficiency-oriented testing?

1. More granular than service-oriented testing
2. Easier to pin down bottlenecks
3. Possible to determine if problem can be solved by hardware modification / scaling / upgrading / etc.
4. Talk in a language developers can understand

Example

- Rent-A-Cat added a Web API showing which cats are available to rent
- Service-oriented testing shows that it takes on average five 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. Lots of new object creation – tune garbage collector, reduce new objects
3. Just need faster hardware – time to migrate away from the Commodore 64
4. Everything running on single machine - Spread work to other cores/processors

- 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” – Donal 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

- What is throughput testing?
- Measuring the maximum number of events the system can handle in a given timeframe.

Examples

- You have a router, and you would like to know how many packets it can handle in one second.
- You have a web server, you'd like to know how many static pages of a given size it can serve in one minute.
- You are running a video game server, you’d like to know how many users can play simultaneously.

How’s that different from service-oriented testing?

1. A given user doesn't care about the number of users who can access a system, just about what it means for them
   - As in, do I notice any disruption in service?
2. More granular
   - Describes a specific system behavior not visible to the user (e.g. packets serviced / sec, pages served / sec, 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.
- Perspective is of system, not the user

Load testing

- Load testing can also be used for subsystems
    - How many simultaneous requests on DB before requests start being queued?
    - How many requests on webserver before response time starts to rise?
    - How many simultaneous images can be written to disk per second?
    - How many videos can be compressed per second if 100 CPUs are allocated?
    - Etc.

Testing resource 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
- 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?

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

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. Determine if it is a performance problem
   - If it's not, let sleeping dogs lie!
2. Track down from top-level to low-level
3. Keep track of performance throughout versions

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