ECE459: Programming for Performance Winter 2024

Lecture 25 — Load Testing

Jeff Zarnett *2024-07-21*

Load Testing

We've had a long look at the subject of observation – identifying areas in the execution of our program that are a potential issue. Now, we will also take some time to relate performance to scalability. Very early on in the course we mentioned the idea that what we want in scalability is to take our software from 1 user to 100 to 10 million. To scale up, we probably need to do some profiling to find out what's slow and make a decision about what to change. It's also useful in terms of making an estimate of our maximum number of users or transactions could be.

That's not a hypothetical scenario either; I [JZ] was asked by a C-Level (company executive) whether a particular system could handle 10× as many users – and that answer had to be supported with some numbers. How would I make that determination? Analysis and testing, of course.

Start with why. The most important question when we want to start doing some load testing is: why are we doing this? A few possible answers [Mel21]:

- A new system is being developed, and management wants to understand the limits and the risks.
- The workload is expected to increase, and we (the CFO) want(s) to be sure the system can handle it.
- A sudden spike in workload is expected, like tax season, and we want to know we can handle it.
- The system has high uptime requirements, like 99.99%, and must work over an extended period of time.
- The project plan has a checkbox "performance testing", even though nobody has a real idea of what it means.

Leaving aside the last "reason", each of the actual reasons why implies a different kind of testing is required. If it's a new system, maybe we just need to understand the average workload (plus, perhaps, a buffer for safety) and make a plan for that. If the workload is expected to increase, like having ten times the number of users, we just need to establish our bottlenecks as in the next topic to identify what we think the limit is. If it's the spike situation, we need to find a way to generate that spike behaviour to see how the system responds, and the hardest part might actually be in how to create the test. If our reason is uptime requirements, then in addition to putting a lot of load on the system, we have to maintain it for an extended period to make sure there's no performance degradation – a test of endurance.

Stress Tests? Load testing is not the same thing as *stress testing*. Load testing involves having a specific target and the goal is to demonstrate that the application can handle that amount. Stress testing is about turning up the pressure (load) until things break, or at least stop working well. We are not going to go into stress testing in this topic, though it should be possible to take the load testing lessons and repurpose them by simply turning the setting up to even-bigger numbers.

Plans Are Useless, Planning Is Essential

The previously-cited book suggests making a plan that answers the questions of *who, what, where, when, & why*. We just covered the "why" question, and understood how it points us towards the answer of "what" – what kind of load testing are we intending to do here. That leaves "who" and "when". In a company (or FYDP) situation,

there might be reason for debate or discussion around who should do the tests and when they should take place. For our purposes the answers are: we are going to do them, and we're going to do them now.

How detailed the plan needs to be is going to depend on your organization, and the same applies for how many sign-offs (if any!) you need on the plan. Some companies have a high need for justification of the time invested in this, after all, because time is money (developer salary) and there are opportunity costs (what work are we not doing in favour of this). There's lots of literature out there about how to justify technical investments to senior management and let's not get sidetracked.

The load testing plan needs at least a definition of what will be tested, how it will be tested, and how we'll know if the test passes or fails. A simple scenario could look something like: we will test saving things to the database by generating 10 000 update-account requests, submit them all at once, and the the total time to complete all transactions should not exceed 15 seconds. That's all made up and probably even a little bit silly, but it meets the important criteria of explaining what to test, how to test it, and how to evaluate success. Let's expand on this.

What Workflows to Test?

While we might have a clear direction about the kind of test we want from why, the question still remains about what workflows are going to be tested. We cannot test everything – aiming for 100% coverage is unnecessary, but load testing should be reserved for only where it's truly needed because of its high cost and effort requirements.

If we already know which ones are slow (rate-limiting) or on the critical path, then we can certainly start there. Ideally, the observability/monitoring (you have it, right?) gives us the guidance needed here. If monitoring doesn't exist, that might need to get addressed first.

If it's a new (as-yet-unimplemented) system, which are the critical workflows for this application? Critical is determined by the nature of the product and what you want it to do. If something is computationally intensive, like compiling analytical data, then that workflow is important. In other scenarios, user experience determines what's critical for load testing: if we have information that says users quit the signup process if the signup page takes more than two seconds to load, we care about making sure the signup time doesn't exceed that limit even when the system is busy. There can also be external requirements that require load testing: if you are processing a transaction and you must return a yes or no response within 1 second, anything that is part of coming up with this answer is critical and worth testing.

If your current utilization is low, you might not know what the rate-limiting steps are at a glance. You can take a guess, but be prepared to revise those guesses partway through the process as you ramp up the load. Actually, you might need to do that even if utilization isn't low; you may find new things along the way that turn out to be the real limiting factors.

In the event of the uptime requirements, the tests likely look the same as the increased-workload situation – we just run them for longer. Endurance tests have significant overlap with load tests, but are not exactly the same. We'll come back to figuring out how long an endurance test should be shortly.

How to Test Them

Given a workflow and the kind of test that we want to do, then we just need to think about how to test it. Carrying out these tests might require provisioning additional (virtual) hardware, particularly if they are long-running. The typical test harness tools used for unit testing aren't always intended for this purpose, so you may need different tools, or just write a custom script to execute the tests.

If we want to run these tests to evaluate or experiment, or if we have an application developed and we want to estimate what its performance and limiting factors would be, there are some things we need to consider to make sure that the results we get are meaningful. So we should respect the following principles, as outlined in [Liu09].

Hardware Principle. Scalability testing is very different from QA testing (you test your code, right?) in that you will do development and QA on your local computer and all you really care about is whether the program produces the correct output "fast enough". That's fine, but it's no way to test if it scales. If you actually want to test for scalability and real world performance, you should be doing it on the machines that are going to run the program in the live/production environment. Why? Well, low-end systems have very different limiting factors. You might be limited by the 16GB of RAM in your laptop and that would go away in the 64GB of RAM server you're using. So you might spend a great deal of time worrying about RAM usage when it turns out it doesn't matter.

Reality Principle. It would be a good idea to use a "real" workload, as much as one can possibly simulate. Legal reasons might prevent you from using actual customer data, but you should be trying to use the best approximation of it that you have.

If you only generate some test data, it might not be representative of the actual data: you might say 1% of your customers are on the annual subscription but if it's really 10% that might make a difference in how long it takes to run an analysis. On the other hand, your test data might be much larger than in production because your tests create (and don't delete) entities in the test system database every time you run.

This isn't theoretical: I [JZ] have been the incident responder on an incident where an app doesn't start up because the database migration takes too long and gets killed. This issue wasn't caught in the test environment because the size of data there was much smaller than the live database. So of course the migration was finished well under the time limit during testing... and failed when going live. Oops.

Related to that, user behaviour is hard to simulate, too. If you are making threads that pretend to be users to simulate concurrent usage of the system, the scripts those threads run may not accurately represent the way the users really behave. Your test example scenario may assume that company managers want to run the report once a month... your users might run them every hour. That one really happened to me [JZ] as well. The manager was using the report to keep an eye on what his team members were doing all day!

Volume Principle. "More is the new more." It's okay to use lighter workloads for regression testing and things like that, but if you actually want to see how your system performs under pressure, you actually need to put it under pressure! You can simulate pressure by limiting RAM or running something very CPU-intensive concurrently, but it's not quite the same as having an actually-high workload.

Why not? The issue isn't so much the test as it is what you're not seeing. If I want to test the app on my laptop under CPU pressure I can just run the app while I repeatedly convert a lecture video just to consume CPU cycles. That does have the effect of limited CPU time available and we can see how the app performs when CPU time is maxed out. But that's not the same as what happens if you have 500 users concurrently, because you might not get to full CPU usage with 500 users due to other reasons, such as lock contention.

These tests, incidentally, are of great interest to the customers and C-Levels, who would like to know that you can deliver promised levels of performance or throughput, or what the current maximum is.

Reproducibility and Regression Testing. Your results will need to be reproducible. That is, two different runs of the load testing on the same code should produce results that are very similar. Unlike unit tests where we expect results to be the same every time, load testing often has a larger degree of randomness and variance in execution because of the nature of the workload (e.g., randomly generate 10 000 customers) and the normal random factors caused by the operating system, scheduler, luck, and so on.

Endurance Tests: How Long?

Chances are you're familiar with the idea of endurance being different from peak workload. Can I [JZ] run at 10 km/h for 1 minute? Sure – I can run much faster than that for 1 minute! Can I run 30 minutes (5 km) at 10 km/h? Yes. 60 minutes (10 km)? Yes, but with difficulty. Four hours (40 km)? No, I'll get tired and slow down, stop, or maybe hurt myself. Okay, so we've found a limit here – I can endure an hour at this workload but things

break down after that. If we only studied my running for a time period that was less than half an hour, we might conclude that I could run at 10 km/h forever (or at least indefinitely), even though that's absolutely not true[1]. The problem is that if our sample period is 15 minutes, that is not long enough to reflect the cumulative negative effects that contribute to my eventual slowing down and being forced to stop.

Is this analogy suitable for software, though? CPUs don't get "tired", nor do their parts accumulate fatigue at anywhere near the same rate as a runner's muscles. Yes, it's still valid! A process that has a data hoarding problem is slowly accumulating memory usage and when its memory space is exhausted, we might encounter a hard stop (e.g., encountering the error `java.lang.OutOfMemoryError: Java heap space`) or just a degradation of performance based on the increasing amount of swapping memory to disk that is required. Same for filling up disk, exhausting file handles, log length, accumulated errors, whatever it is. That's a little bit like fatigue for the executing application, whether it's building in the program or its environment: it builds over time, and eventually the effects of it force a slowdown or stoppage of execution.

Accumulated "fatigue" for an application is not just a hypothetical, either. I [JZ] have personally seen services that encountered a problem due to running out of some internal resources as a result of a code freeze over the holiday season. That wasn't a load test in the sense of applying a higher load to validate execution rate – it was just an endurance test, even though it was not planned as one. The solution to getting the error rate down involved restarting the instances one-by-one and things got back on track. This example calls back what I said about how endurance tests are not exactly the same as load tests, in that we can have an endurance test with low load and it's still valid.

With that said, how do we identify what is the relevant period for the endurance test – or in the running analogy, how do I know if 30 minutes is the right running length to get a real idea? Should it be 3 hours? Once again, our first guide might be the product requirements for what we're building (testing) might give an idea. If it's an e-commerce platform then the endurance test might be something like five days to cover the period of US Thanksgiving, Black Friday, the weekend, and then Cyber Monday.

Other ideas for choosing endurance targets might consider the maintenance windows for the platform. Suppose you have a scheduled maintenance window that takes place on Sundays from 02:00 – 03:00 and there can be downtime during that period, according to the contracts (service level agreements) the company has signed. In that case, you want to validate that the system will work correctly and consistently long enough that it can be restarted (or updated) only during that maintenance window.

Unfortunately, there are no universal rules we can offer that give you the exact length of time to evaluate. You'll have to consider the requirements, the likely scenarios, and use your judgement.

How to Evaluate Success

There are two kinds of answers that load testing can give you (and they're related). The first is "Yes or no, can the system handle a load of X?"; the second is "What is the maximum load Y our current system can handle?". If we know Y then we can easily answer whether X is higher or lower. Between the two of them, this suggests we might prefer to find Y rather than answer the first question. But it might be hard to find the maximum limit. The difficulty of generating test data or load might increase much faster than the rate of load added to the system, and we might be crossing over into stress testing. Sometimes answering the first question is all that is necessary.

The value of Y may have some nuance above. The maximum rate we can handle may imply a hard stop – if the load exceeds Y then we crash, run out of memory, reject requests, or something else. It may also be a degradation of service: this is the point at which performance degrades below our target or minimum.

Observability has come up previously and it might have been present to help decide what's important. We might

[1] That's a common fallacy in the media, too – you may notice, if you're looking for it, a plethora of journalists writing opinion pieces that say "the current situation is the new normal and will go on forever" – even though that's almost certainly not true. Things change all the time – political parties that win an election are often voted out again after a few years; rough job markets improve and good ones get tighter; high interest rates come down or low rates increase, etc. New and unprecedented things happen a lot, and change is constant.

also need to add some monitoring or logging that tracks when events start and end, so we can gather that data needed to make the overall evaluation. Examples that we are looking for are things like the following. Is the total work completed within the total time limit? Did individual items get completed within the item time limit 99% of the time or more?

As you might expect, the raw load test results are not always sufficient to make the call as to whether a test has passed or succeeded. Then there is post-processing to aggregate and analyze the data. Some manual work might also be necessary to correlate the data with other known factors, particularly if it had not been possible to test on separate hardware or separate instances [Mel21]. At the end of this process, hopefully you can look at the outcomes and conclude whether a given scenario is passed or failed.

Finally, success in an endurance test is not just answering whether the system was able handle a load of X, but is about whether that answer continued to be yes, consistently, across a long period of time. In an endurance test especially, looking at the trend may be more instructive as to how well things are working.

So You Failed the Load Test...

Like a software program, when it comes to running, I [JZ] can get better – I just need to make some changes. To get better at endurance running, chances are I mostly just need to increase (slowly) the running distances and practice more and I'll get better. Or as the internet might say, git gud (get good).

The idea works for the programs too – if the load test has failed, we'll have one or more specific scenarios to improve and we can apply techniques from this course (or elsewhere) to make that part better. Then re-run the test and re-evaluate. If all is well, done; otherwise, repeat as necessary until we've passed the scenario.

And also like a software program, there is a point at which I can make no more improvements. At the time of writing, Google says that the world record for marathon running is 2:01:09, held by Eliud Kipchoge, or an average speed of 21.02 km/h. There is absolutely no chance I can get to that level, no matter how hard I train. So I am not getting selected for the Canadian Olympic team.

If we've reached the limits of what we can do in the software, it might not be the right tool for your needs and a major redesign or replacement is needed. Designing the system with higher load in mind is sometimes possible, though even in situations where it's possible the cost might be prohibitive. Are we stuck?

No! You can have different expectations. A four-hour marathon seems achievable for me if I worked on it. If it's unrealistic to bill all your customers on the same day, why not convince the company to let billing be spread across the whole month? Think about the problem outside the constraints that are currently given.

Constant Vigilance

You may have heard the saying "constant vigilance" around the topic of defending against the dark arts. Load testing at a given moment in time captures the state of things at that time only. The load testing procedure needs to be repeated regularly to catch performance degradations that would make your software fail to meet its targets or design load. These are rarely intentional, but the tendency of software is to grow in complexity and in functionality, both of which are likely to make it slower. Improved hardware does, over time, offset some of the slowdown of added complexity. However, at the time of writing, this is an era of small, incremental improvements year-to-year, not big leaps and bounds forward [2]. So, for now, it is still a priority to repeat the load tests often enough to catch major regressions before they become a problem.

Real-life example: `https://arewefastyet.com` tracks regressions/improvements in Firefox performance.

[2] Considering the previous footnote, it would be thoroughly foolish to now fall into the trap of saying "the current situation will go on forever" and say that there will never be revolutionary change in execution hardware that offsets the increases in complexity. But that's not where we are right now.

References

[Liu09] Henry H. Liu. *Software Performance and Scalability: A Quantitative Approach*. John Wiley & Sons, 2009.

[Mel21] Leandro Melendez. *The HitchHiking Guide to Load Testing Projects: A Fun, Step-By-Step Walk-Through Guide*. Journeyman Publishing LLC, 2021. URL: `https://books.google.ca/books?id=_eWAzgEACAAJ`.