ECE459: Programming for Performance Winter 2024

Lecture 26 — Load Testing

Jeff Zarnett *2024-07-20*

Load Testing

So far, we have discussed scalability at a high level. We talked about some principles (hardware, reality, and volume), and how we might narrow down which of the components is a bottleneck for the execution of your code. That's a good start, but we are going to take some time to examine more closely the idea of how to design and implement a proper load test.

Start with why. The most important question 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 previous 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.

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. Increasing the load and execution of the tests appears in this section for completeness, but we already learned the most important things about how to carry out the tests in the introduction to the scalability topic! Remember the hardware, reality, and volume principles.

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.

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. 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 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?

Finally, success in an endurance test is not just answering whether the system can handle a load of X, but is about whether that answer continues to be yes, consistently, across a long period of time. Let's get into that.

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].

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! 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, file handles, log length, accumulated errors, whatever it is. That's a little bit like fatigue.

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 though, just an endurance test, even if not a planned one. The solution just 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.

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? 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.

So you failed the load test... Like a software program, though, with some changes, I can get better. 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. That's probably true for the program – if the load test has failed, we'll have a specific scenario to improve and we can apply techniques from this course (or elsewhere) to make that part better, re-run the test, and 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. If that's the case for your software, it might not be the right software for your needs and a major redesign or replacement is needed. Or 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?

References

[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`.

[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.