COSC 3P95 Assignment 2 – Report

Question 1

Implementation Details

For my implementation, I decided to use Python's Flask library as it has the benefit of easily producing a testable client/server system, while also supporting OpenTelemetry Auto Instrumentation – something that simple socket programming, which I regularly use for school assignments, does not have. With Flask, I simply set up a root domain route which aims to execute the following workflow:

- With rate-limiting logic added, checks to see if the rate-limit has been exceeded; if not, we continue, but if so, we wait until it refreshes. This could in theory return a 429, and it did in prior experience, but I decided on this approach as we're really only concerned with file fetching.
- Calls to a threaded file fetching function:
    - Generates a random integer between 1 and 20, the bounds of the folder size for this assignment.
    - Picks a random file based on this integer.
    - If deliberate delay is enabled, if the file is number 7, we wait 3 seconds.
    - If data integrity is enabled, we calculate the checksum and store it for later.
- Joins the thread and receives the file from a global context.
- Returns the file through Flask's send_file method.

The client is a relatively simple piece of software – it can be run in parallel to the server and makes a request every 2 seconds per completed request to grab another random file. This allows for a probabilistic overview of the system running time. If data integrity is enabled, it also verifies the checksum and informs the user of this output.

For my UI, I use a Docker build of Jaeger (running instructions attached in the README). We only use this to visualize traces and spans, but as a consequence of its UI, we see total execution time which will be an important metric in the later sections.

Screenshots of the UI are attached in the MEDIA folder under this question's folder.

The provided files for data transfer are simple Lorem Ipsum files of a specific length to ensure some differences.

Impacts of Advanced Features

Through testing, impacts are mostly negligible. Over a small sample size, as attached under MEDIA, there appears to be no immediate impact whether all features are on or off. This makes sense:

1. **Error-Retry Logic** – only impacts runtime if errors are found.
2. **Rate-Limiting Logic** – only impacts runtime if rate-limit is exceeded.
3. **Checksum Logic** – a fairly robust and quick bytewise solution that shows no immediate slowdown.

However, if some conditions, such as the repeated exception or rate-limit rules are violated, we may see increases in runtime. I do note that through these implementations, though, there appears to be no immediate performance impacts.

Difficulties and Challenges

With respect to challenges in implementing the required deliverables, there appears to be very few:

1. **OpenTelemetry/Jaeger Implementation** – this was a fair bit more difficult because it required reading through a lot of documentation as the primary, and often *only*, source of information for setting up my monitoring. As a result, a lot of time was spent grappling with both frameworks, which could have been bettered applied down the road.
2. **Choosing a Client/Server Architecture** – using Flask felt a bit less authentic, with less fine-grained control, but I had to make that sacrifice to ensure that I could use auto instrumentation out of the box.
3. **Control over Advanced Logic** – because these features aren't exactly plug and play, analyzing with and without them was rather tedious. However, as I discussed, these features had little impact anyways, so this was a very short-term problem.

Final Performance

I find that performance is relatively quick with respect to input size – of course, it will be slower to process a 45MB file than a 5KB file. The application is simple and easy to use & evaluate. There could likely be some code cleanup done with more abstraction, but this is beyond my limited scope of Python compared to other languages I attempted to implement this in (namely, TypeScript).

Question 2

Defining Bug & Predicate

The bug we have introduced in this case is a deliberate sleep when the file number is greater than or equal to 15. In the real world this could be a corrupted or malformed file causing issues, but here we just sleep on any file above or at 15 being selected. The predicate, therefore, in this case would be rand >= 15, with an opposite predicate of rand < 15.

Statistics

I ran 80 tests on random files, with no rate-limiting to prevent false positives (the logic remained implemented, but I set the tokens to 60, equivalent to one every one second which is double what we need). I then stored the output results in a CSV file (attached alongside the JSON dump of the spans from this run).

To help our data stand out, I isolated runs that took more than a few seconds, as my typical runtime environment operates in under half a second. You can observe this behaviour from the MEDIA folder where I illustrate the Jaeger graph of the spans.

The following was observed:

Predicate	Failing / Total Runs	Fail Rate
Rand >= 15	22 / 22	100%
Rand < 15	0 / 58	0%

For the predicate where Rand >= 15, we see a **Failure of 1.0, a Context of 1.0, and therefore an Increase of 0.** This tells us that our slowdown bug is isolated **purely** to the sleep line.

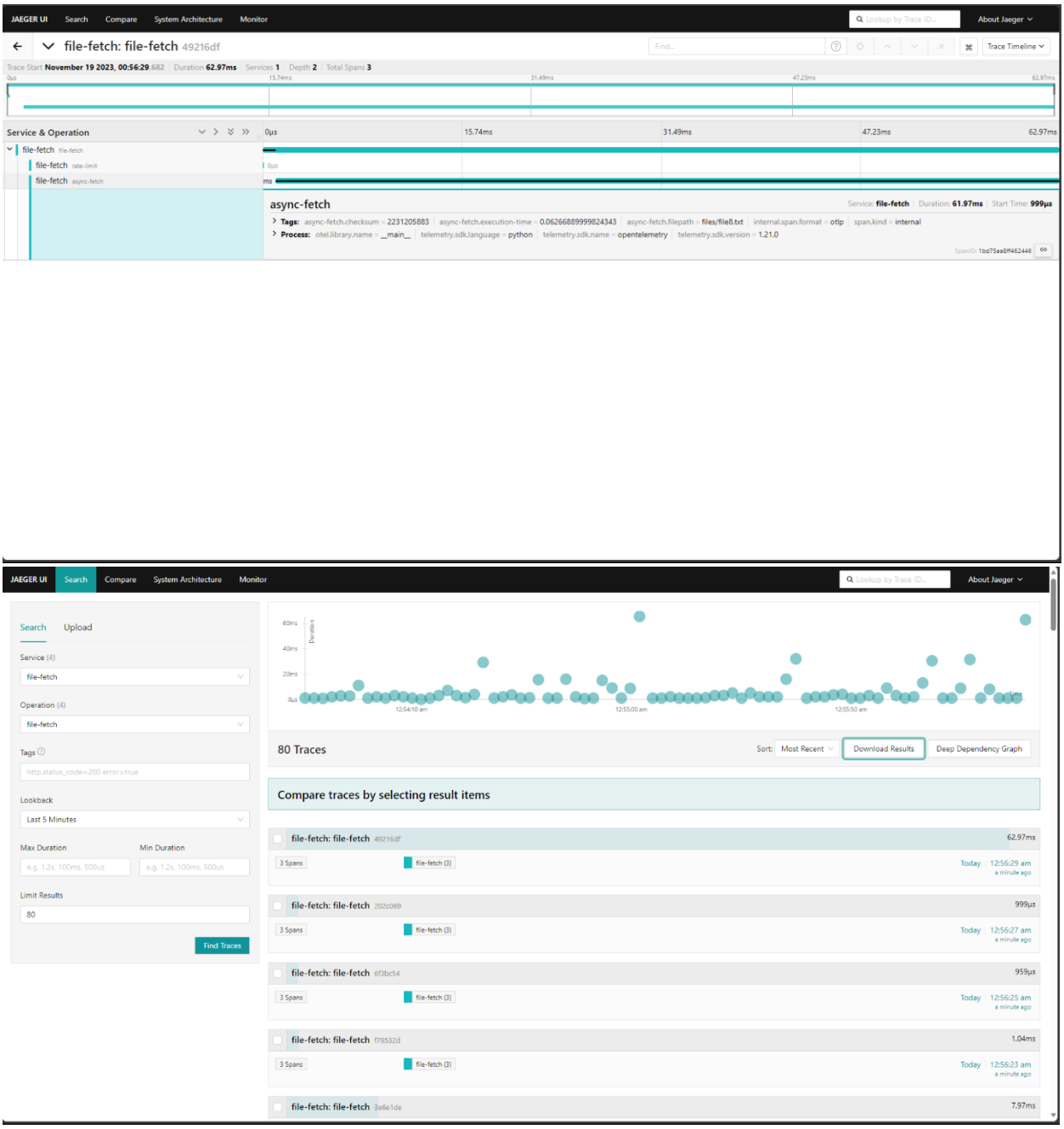
Therefore, we can conclude reasonably that the branch of code responsible for our slowdown is:

```
if (rand >= 15):
        time.sleep(3)
```

OPTIONAL MARK: we are able to comment out this section to improve performance immediately. I ran 80 more runs to illustrate this, also attached as a json trace file. As observed, it does not exceed 60ms, compared to the ~3s before.

No configurations were changed. The branch above was the only codebase change introduced.

Media



Jaeger Visualizations – dashboard and trace analysis.