**3P95 Assign2 Report**

**Basim Ahmad**

**Fouzan Abdullah**

**Question 1:**

This file transfer programme uses Python sockets to make it easier to move files from a client to a server. Both client-side and server-side code are used in the implementation. Prometheus is used as the tracing exporter, and OpenTelemetry has also been integrated onto both sides for distributed tracing.

**Server-Side**

Implementation:

The server initializes OpenTelemetry in the first stage, configuration, and initialization, to enable distributed tracing. The programme adds a PrometheusSpanExporter and sets up a TracerProvider to export trace data to allow for operation monitoring. The server configuration comes next, where you may set up the host, port, buffer size, and output folder to receive files, among other important server configuration parameters. Next is file handling, in which the server handles incoming files from clients by using the receive\_file function. It begins writing to the file and repeatedly receives file portions until the entire file has been sent. The client connection is then managed by the handle\_client function. It transfers the file transfer process to the receive\_file function after receiving the file information (filename and size) from the client. The server then uses a ThreadPoolExecutor to implement concurrent programming to manage several client connections at once. The client handling logic is executed in parallel via the fork\_parallel\_pattern function. Then, using the produce random file content and generate random files functionalities, the server can create random files with random content. Before launching the server, these functions are used to generate test files. Lastly, the server-side implementation enters through the main function. It configures the server socket, waits for connections to come in, and uses the ThreadPoolExecutor to handle each one in turn.

Performance Analysis:

The first is Thread Pooling and Concurrency, where the server uses a ThreadPoolExecutor with a maximum of 10 worker threads to achieve concurrency. Because of its design, the server can manage several client connections at once, improving throughput and responsiveness. The second is file reception, where the receive\_file method effectively uses less memory by receiving files in pieces. By ensuring that data is processed in reasonable pieces, the usage of a buffer size optimises file transfer performance. Robust error handling is built into the handle\_client method to handle any problems that may arise during file transmission. Using the generate random files feature, the server creates 20 random files of different sizes. The size of the created files determines how file production affects performance. Concurrent programming may result in costs even while it improves server responsiveness. The ideal number of worker threads (10 in this example) might need to be changed depending on the workload and system resources.

Findings:

Some key points are as follows:

1. File transfer functionality:

Ensuring that the files generated by the client are successfully transferred over to the server. We can do this by checking the file size of each file on both ends and making sure that they match.

1. Error handling:

A small breakdown of the error handling function is a simple implementation of a ‘try’, ‘except’, and ‘finally’ block. The ‘try’ block attempts to perform the file transfer process by first receiving the file information from the client, then calls the receive\_file function to receive the actual data within the file. The ‘except’ block captures an exception to get any information that we can about said exception and stores it in a variable e. The ‘finally’ block contains code that is executed regardless of whether or not the exception case is true. In this case, it closes the client socket to prevent any memory leaks, and the code just prints a message saying that the client is closed.

1. OpenTelemetry integration:

The server side and client-side code were tested vigorously using different parameters to ensure the best results for the metrics that were visualized using Prometheus. The following graphs display the results for the initial implementation with no advanced features initiated:

A screen shot of a graph

Description automatically generated

A graph on a computer screen

Description automatically generated

This graph shows a metric that represents the total number of http requests processed by a web server. As you can see, once we run the code initially it provides a total number of 0 Prometheus requests which over the course of 30 minutes, which is the testing time used, the number of total requests increases to around 119-120. This shows that once we start running the code without any implementation of any advanced feature, it starts from 0 and makes it way up.

A screenshot of a computer

Description automatically generated

On the other hand, when we implement the error handling feature, we can visually see and capture specific error responses. As we can see in the graph the successful request and failed request are split which is shown by the empty line in the middle. So, what error handling does, is that it enables the separation of successful and failed attempts by categorizing them based on their status code. A more advanced visualization would be to create different Prometheus queries for failed attempts and successful attempts.

Challenges faced:

Several issues and concerns were considered when implementing the file transfer application in the server-side code. Here are a few noteworthy difficulties:

1. Management of Concurrency:

  Managing several client connections at once adds complexity to resource sharing and thread safety.

Solution: The code optimises the amount of worker threads and handles any race conditions by using a ThreadPoolExecutor to regulate concurrency.

2. Buffer Size Calculation:

It is not easy to figure out what size buffer is best for receiving file chunks. A buffer that is too large can cause excessive memory utilisation, while a buffer that is too tiny might lead to frequent network requests.

Solution: The code sets the buffer size to 1500. Depending on the size of the files being transferred and the state of the network, this decision may have an effect.

3. File Generation and Random Content:

Determining the file size and randomization of content present difficulties when creating random files with random material.

Solution: The file size is arbitrarily chosen between 5 KB and 100 MB, and the algorithm generates random content using the os.urandom function. Variability is now introduced for testing.

4. Monitoring and Tracing:

Including features for monitoring and tracing brings additional work that is associated with third-party integrations (like Prometheus and OpenTelemetry).

Solution: Prometheus and OpenTelemetry are used by the code to provide distributed tracing and monitoring. Still, constant observation and fine-tuning are necessary for the best performance.

**Client-Side**

Implementation

Initialization and configuration are the first steps, where The client initialises OpenTelemetry to enable distributed tracing in a manner like the server side. To export trace data, it creates a PrometheusSpanExporter and configures a TracerProvider. The configuration of the client comes next. Here, it is set up with parameters like the address and port of the server, the size of the buffer, and the input folder that has the files that need to be delivered to the server. File sending comes next, where Sending a file to the server is handled by the send\_file function. The file's actual content is transferred in small, manageable portions once the filename and size have been sent. Finally, the client-side implementation's entry point is the main function. After establishing a connection with the server and creating a client socket, iterates through the files in the input folder, sending each file to the server via the send\_file function.

Performance analysis

First off, files are effectively sent to the server in little portions via the send\_file method. By ensuring that data is transferred in manageable chunks, the buffer size helps to avoid using too much RAM. The main function then handles connections by iterating through the files in the input folder and creating a connection for each item. Distributed tracing and monitoring are possible with OpenTelemetry integration on the client, just like it is on the server side. Next, Performance is impacted by file size, particularly in terms of bandwidth utilisation and network delay. Larger files could use more resources and cause longer transfer times. To handle possible problems during file transfer, the client contains basic error handling. Logging of exceptions yields information useful for debugging and troubleshooting.

Challenges Faced:

Several issues and concerns arose throughout the file transfer application's implementation in the client-side code. These are a few noteworthy obstacles:

1. Bandwidth and Latency in the Network:

The client-side code might encounter issues with available bandwidth and network latency, which would slow down file transfers.

Solution: These issues can be lessened by carefully managing network circumstances and optimising the buffer size. The influence of network properties is still considered, though.

2. Impact of File Size:

The performance of the file transfer might be affected by the size of the files being transferred from the client to the server.

Solution: To control the transmission of file chunks, the algorithm makes use of a buffer size. Nonetheless, while testing, the buffer size selection, and the effect of different file sizes on performance must be considered.

3. Sequential Connection Handling:

Each file requires a new connection, which the client manages sequentially. Performance could be impacted by this, particularly when handling numerous files.

Solution: Concurrent file uploads or connection pooling could be implemented to improve efficiency, depending on the use case.

4. File Transfer Effectiveness:

Sending files to the server efficiently means taking into account things like chunk size, transmission speed, and how to handle possible network outages.

Solution: To control file transfer, the algorithm makes use of a buffer size. The outcomes of performance testing may lead to further optimisations being investigated.

**Conclusion**

In summary, this file transfer programme makes use of socket communication in Python to offer a reliable method for file transfers between a client and a server. Performance monitoring and improved observability of the application are made possible by the integration of OpenTelemetry for distributed tracing. Several factors affect the file transfer application's performance, such as network circumstances, file handling efficiency, and concurrency management. Continual performance evaluation and iterative optimisation initiatives will help to keep the file transfer system reliable and effective. The server-side concurrent programming guarantees effective management of several client connections, resulting in a scalable and responsive application. The overall design offers flexibility, concurrent processing, and monitoring features, demonstrating a well-rounded approach to file transmission.

**Question 2:**

The structure for the server-side and client-side code is the exact same as Question 1. The only difference is that a deliberate delay was introduced in part of the server program that only occurs in some cases.

The bug that was introduced involves the server-side code for receiving files. This intentional delay is introduced under certain conditions and is not consistently applied, leading to unexpected behaviour during file reception.

The SD techniques applied for debugging include:

1. Logging: Print statements were added at key points in the code to identify where the delay should occur.
2. Probability sampling: A probability-based delay mechanism is introduced to simulate scenarios where delays may occur. In this case it was set to a rate of 40%.

Analysis of the bug:

1. Predicates: The bug was identified by observing that the delay is not consistently applied during file reception. It only has a chance which is 40% to occur.
2. Metrics: Probability\_sampling was utilized to control the likelihood of applying a delay. This probability is adjusted during testing to observe the impact of different probabilities on file reception.

Other findings:

1. OpenTelemetry: was integrated for distributed tracing. This provides comprehensive tracing capabilities for debugging and performance analysis.
2. Exception handling: was implemented to ensure effective termination of the client socket. This in turn improves robustness of the code.

How was the issue fixed:

1. Identifying the issue: During file receiving, it was found that the probability-based delay was not always used.
2. Updating the code: The handle\_client function was changed to guarantee that the deliberate delay is applied uniformly upon receiving each file chunk.

Validating the fix:

1. Testing: Different file sizes and probability\_sampling rates were tested to ensure that the intentional delay is applied consistently whenever the condition is met. Prometheus is used to visualize the data as shown in question 1 with the different parameters. The following graph shows output for when the server-side code is ran using the deliberate delay bug:

A screenshot of a computer

Description automatically generated

Similarly, to when error handling was introduced, the graph shows the specific error responses. The failed requests are separated by the successful requests which is very similar to the output for the error handling feature. This is testing it using a probability rate of 40%. If I increase the probability rate to 100%, the graph is as follows:

A screen shot of a graph

Description automatically generated

In this graph, we can see that the total number of http requests stays successful throughout the run which shows that there is little to no failure caused by the bug. Since the probability rate is 100%, the bug will never have a failed attempt thus having a solid line with no gaps as shown by the graph.

OpenTelemetry also utilizes performance monitoring to trace the performance of the delay which ensures that it aligns with the expected outcome.

Conclusion:

By making sure that deliberate delays are applied consistently during file reception, the bug was found and fixed effectively. The debugging process was aided by the integration of logging, probability-based sampling, and systematic testing, which produced a more dependable and sturdy file reception mechanism. The integration of OpenTelemetry improves the general debugging and performance analysis capabilities of the code as per requirements.