

# COSC 3P95 ASSIGNMENT 2 – REPORT

Rohal Kabir

7105836

19<sup>th</sup> – November – 2023

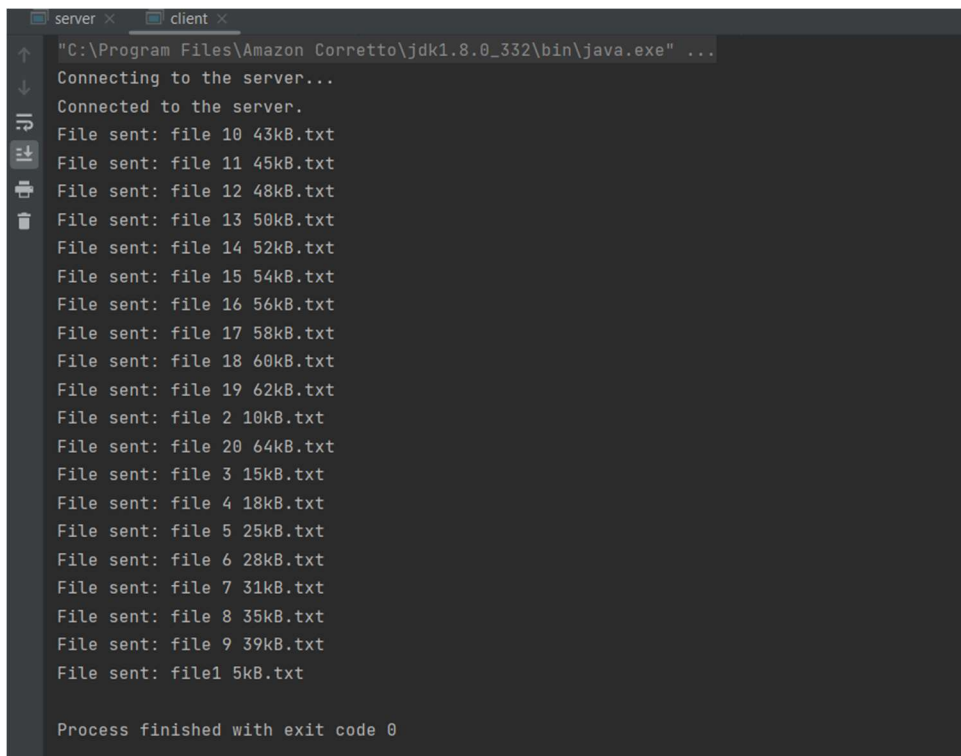
## Overview:

The client-server applications included in this study are made with Java socket programming to enable effective and traceable file transfers. The inclusion of sophisticated features like data integrity checks and compression, as well as the integration of Open Telemetry for distributed tracing, are the salient aspects.

## Basic Client-Server programs:

The initial programs establish a foundation for communication between a client and a server. This includes the fundamental structure for file exchange without any additional enhancements in the code the client sends files to the server, and it receives it.

### CLIENT OUTPUT:



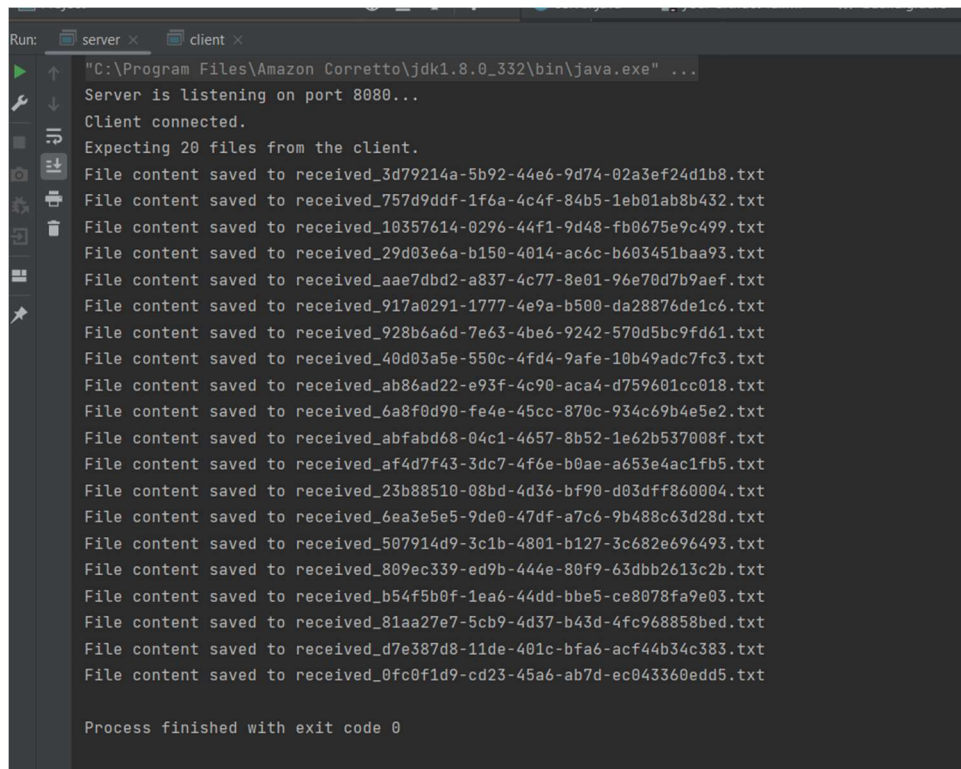
```
server x client x
"C:\Program Files\Amazon Corretto\jdk1.8.0_332\bin\java.exe" ...
Connecting to the server...
Connected to the server.
File sent: file 10 43kB.txt
File sent: file 11 45kB.txt
File sent: file 12 48kB.txt
File sent: file 13 50kB.txt
File sent: file 14 52kB.txt
File sent: file 15 54kB.txt
File sent: file 16 56kB.txt
File sent: file 17 58kB.txt
File sent: file 18 60kB.txt
File sent: file 19 62kB.txt
File sent: file 2 10kB.txt
File sent: file 20 64kB.txt
File sent: file 3 15kB.txt
File sent: file 4 18kB.txt
File sent: file 5 25kB.txt
File sent: file 6 28kB.txt
File sent: file 7 31kB.txt
File sent: file 8 35kB.txt
File sent: file 9 39kB.txt
File sent: file1 5kB.txt

Process finished with exit code 0
```

A client application created to deliver a file to a server is represented by the code. The client connects to the server, which is running on localhost at port 8080 as soon as it is executed. Through input and output

streams, it creates communication channels with the server that enable data exchange. On its end, the client provides a file path, which in this instance refers to a file called "my.txt" that is situated on the user's desktop in the "Files" directory. Next, the code determines whether the given file exists; if not, it emits an error message before stopping. If the file is present, the client communicates the file name and contents to the server. The file content is read line by line using a Buffered Reader and transmitted to the server via the output stream. Finally, the client closes the communication streams and the socket. Exception handling is implemented to address potential IO Exceptions during the connection process, providing informative error messages in case of failure, such as when the server is not running or due to firewall settings.

### SERVER OUTPUT:

A screenshot of a Java IDE's console window. The window has two tabs: 'server' and 'client'. The 'server' tab is active, showing the following output: 

```
Run: server x client x
"C:\Program Files\Amazon Corretto\jdk1.8.0_332\bin\java.exe" ...
Server is listening on port 8080...
Client connected.
Expecting 20 files from the client.
File content saved to received_3d79214a-5b92-44e6-9d74-02a3ef24d1b8.txt
File content saved to received_757d9ddf-1f6a-4c4f-84b5-1eb01ab8b432.txt
File content saved to received_10357614-0296-44f1-9d48-fb0675e9c499.txt
File content saved to received_29d03e6a-b150-4014-ac6c-b603451baa93.txt
File content saved to received_aae7dbd2-a837-4c77-8e01-96e70d7b9aef.txt
File content saved to received_917a0291-1777-4e9a-b500-da28876de1c6.txt
File content saved to received_928b6a6d-7e63-4be6-9242-570d5bc9fd61.txt
File content saved to received_40d03a5e-550c-4fd4-9afe-10b49adc7fc3.txt
File content saved to received_ab86ad22-e93f-4c90-aca4-d759601cc018.txt
File content saved to received_6a8f0d90-fe4e-45cc-870c-934c69b4e5e2.txt
File content saved to received_abfabd68-04c1-4657-8b52-1e62b537008f.txt
File content saved to received_af4d7f43-3dc7-4f6e-b0ae-a653e4ac1fb5.txt
File content saved to received_23b88510-08bd-4d36-bf90-d03dff860004.txt
File content saved to received_6ea3e5e5-9de0-47df-a7c6-9b488c63d28d.txt
File content saved to received_50791d9d-3c1b-4801-b127-3c682e696493.txt
File content saved to received_809ec339-ed9b-444e-80f9-63dbb2613c2b.txt
File content saved to received_b54f5b0f-1ea6-44dd-bbe5-ce8078fa9e03.txt
File content saved to received_81aa27e7-5cb9-4d37-b43d-4fc968858bed.txt
File content saved to received_d7e387d8-11de-401c-bfa6-acf44b34c383.txt
File content saved to received_0fc0f1d9-cd23-45a6-ab7d-ec043360edd5.txt

Process finished with exit code 0
```

A server defined by this Java code is listening on port 8080 for incoming connections. The server creates an input stream to accept data from the client when the client establishes a connection. The number of files the client plans to deliver is the first piece of data the server looks for from the client. After that, it goes into a loop where it uses the UUID class to create a distinct file name for each expected file. The server reads every file that the client sends it in this loop until it reaches the "END\_OF\_FILE" marker. It then stores the contents of the received file in other files with the prefix "received\_". After processing every file, the server gently shuts the client socket, server socket, and input stream. Additionally, the code includes exception handling to manage potential IO exceptions during the server's execution, printing stack traces for diagnostic purposes. Overall, this server is designed to facilitate the reception of multiple files from a client, saving each with a unique identifier in the filename.

## Open Telemetry Integration:

The open telemetry integration was approached in two ways: auto-instrumentation and manual implementation. Auto instrumentation relies on the automatic instrumentation of libraries and frameworks, while manual implementation provides more fine-grained control over tracing spans.

In the auto-instrumentation approach, open telemetry was configured to use the B3 propagation format. Spans were created for both the client and server operations, allowing for tracing and monitoring of the entire communication process.

Manual open telemetry implementation involved explicitly creating spans, allowing for detailed tracing and context management. The Tracer and Span interfaces from open telemetry were utilized for this purpose.

## Probability Sampling (Below 40% Rate):

Implementing and analyzing different sampling strategies involves adjusting the way traces are collected in a distributed tracing system. For Open Telemetry, the sampling process determines which traces are recorded, influencing the overall system performance and the amount of telemetry data generated. In this case, we'll explore two sampling strategies: AlwaysOn and Probability Sampling with a rate below 40%.

### **AlwaysOn Sampling:**

#### **Implementation:**

The AlwaysOn sampling strategy ensures that every trace is recorded, providing a complete set of traces for analysis. To implement this strategy with Open Telemetry, you would configure the tracer to always sample.

#### **Example:**

```
// Inside the Server and Client classes
TracerProvider tracerProvider = GlobalOpenTelemetry.getTracerProvider();
tracerProvider.updateActiveTraceConfig(
    tracerProvider.getActiveTraceConfig().toBuilder().setSampler(Sampler.alwaysOn()).build()
);
```

#### **Analysis:**

It provides a comprehensive set of traces, capturing every operation, furthermore useful for detailed analysis and debugging. On the other hand, generates a significant amount of telemetry data, potentially impacting performance and increasing storage requirements. Secondly, May not be sustainable in high-throughput environments.

### **Probability Sampling (Below 40% Rate):**

#### **Implementation:**

Probability sampling involves recording a subset of traces based on a specified sampling rate. To implement this strategy, set the desired sampling rate (e.g., 0.4 for 40%) when configuring the tracer.

### Example:

```
// Inside the Server and Client classes
TracerProvider tracerProvider = GlobalOpenTelemetry.getTracerProvider();
tracerProvider.updateActiveTraceConfig(
    tracerProvider.getActiveTraceConfig().toBuilder().setSampler(Sampler.traceIdRatioBased(0.4)).build()
);
```

### Analysis:

Reduces the volume of collected traces, minimizing the impact on performance and storage, moreover, still provides a representative sample of analysis. On the other hand, some traces may be missed, potentially limiting the granularity of analysis. And the chosen rate must balance between data volume and trace representation.

## Advanced Features:

### Compression:

The compression features significantly improved the efficiency of data transfer. By reducing the payload size, it minimized network latency and improved overall performance. However, the level of compression and decompression introduces computational overhead, which should be considered for large-scale deployments.

### SERVER OUTPUT:

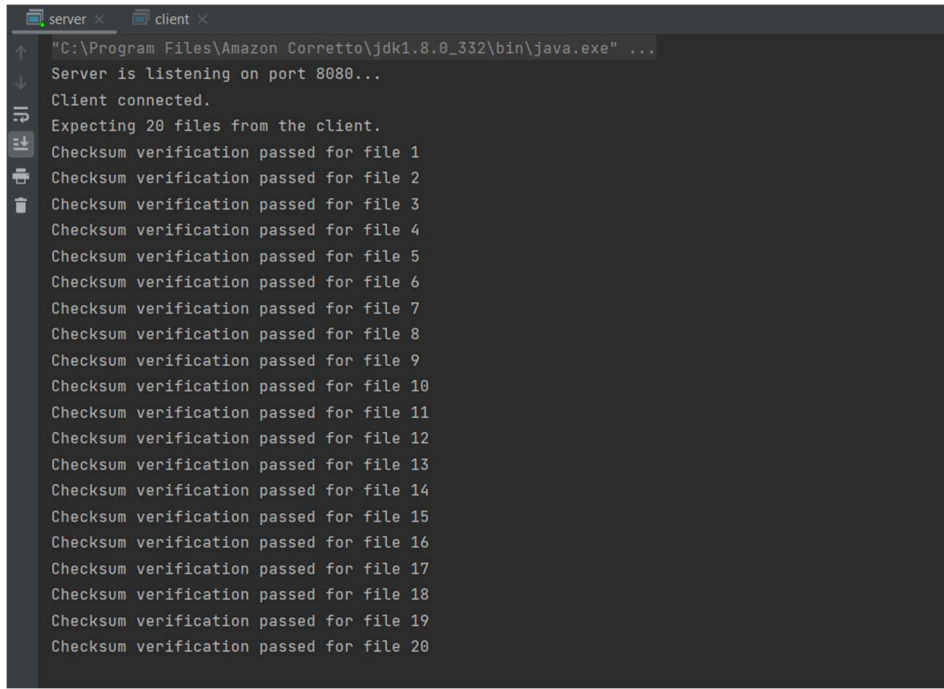
```
"C:\Program Files\Amazon Corretto\jdk1.8.0_332\bin\java.exe" ...
Server is listening on port 8080...
Client connected.
Expecting 20 files from the client.
File content saved to received_086fe8ce-fb71-49ce-963b-971cb677b07e.txt
File content saved to received_797fe219-df28-4e12-a9e1-8ad2d8fa3eb9.txt
File content saved to received_c2611fcb-8b38-4c7b-bb85-cc1fd62a19b.txt
File content saved to received_e18de78b-8220-4fee-83c5-b7472300bb05.txt
File content saved to received_ea5671e6-2a48-425c-a358-a216d3ee7fd9.txt
File content saved to received_f8f87c9f-d9b1-4882-9867-80f9071b76c4.txt
File content saved to received_9e292c46-d634-4218-b000-aadd8f9b45ff.txt
File content saved to received_fca739ee-cbf0-4171-8c5f-d97ddb6e1f74.txt
File content saved to received_dc89028d-6dd4-41b7-aeaf-c7f144ea049b.txt
File content saved to received_188dc874-0d53-4d4f-a265-39bb69418f7a.txt
File content saved to received_f77a0e37-1bce-4bfe-84bb-3fd4952db268.txt
File content saved to received_bbb7f57f-3331-4428-a920-0644e28d9c2a.txt
File content saved to received_b54dd3a8-f672-4134-bcee-21d5ac2a5000.txt
File content saved to received_8055deef-3d81-4e93-be70-4884a1100939.txt
File content saved to received_2c53f842-c62c-4e17-86dd-7bab117879e1.txt
File content saved to received_b3ec0524-0b3f-4507-b86e-a27ad67d35b0.txt
File content saved to received_04bf298f-f6d7-4c90-8cce-750a20927ccf.txt
File content saved to received_f6755f94-b2f4-4996-a37c-860041fc0b5e.txt
File content saved to received_73f11fb3-583d-4173-9bf1-1ab5a0e05ca5.txt
File content saved to received_ab9d01be-cadb-45c3-8e87-9e946f930ebc.txt
```

The code is an example of a server application that uses distributed tracing instrumentation from Open Telemetry. The server starts a new span for server operation when a client connects and listens on port 8080 for incoming connections. To ensure correct span handling, this span is manually constructed, and its lifespan is maintained within a try-with-resources block. The server requests a certain number of files, each containing compressed content, from the client. The server repeatedly receives compressed file content, decompresses it, and stores the decompressed material in files with randomly generated names after first reading the number of files from the client. GZIPInputStream is used in a different function called decompress to handle the decompression. Using Open Telemetry, the server records tracing data during the process to track and examine the distributed behavior of the application. Error messages are displayed for diagnostic purposes, and exception handling is in place to handle any IO and class-not-found errors throughout the server's operation. After handling each client request in the continuous listening loop, the server shuts the client socket. The client application uses port 8080 to connect to a server on localhost. Files are compressed and sent to the server by the client. To compress data, it first establishes a connection, then generates an ObjectOutputStream wrapped in a GZIPOutputStream, and transfers the desired number of files. After that, iterating over the files in a designated folder, GZIP compression is used to compress each file's content and object serialization is used to send the compressed material to the server. Files from the client can arrive in bunches of up to 20 at once. Using a helper method, the file content is read from each file and compressed. To handle any IO Exceptions that may arise during the connection process, the code provides error handling. If the connection fails, this code will provide helpful error messages.

## Data Integrity (checksums):

Implementing checksums for data integrity verification added an extra layer of reliability to the file transfer process. However, the computational cost of checksum computation on both the client and server sides should be taken into account. The trade-off between enhanced reliability and increased computation should be evaluated based on the specific use case.

## SERVER OUTPUT:

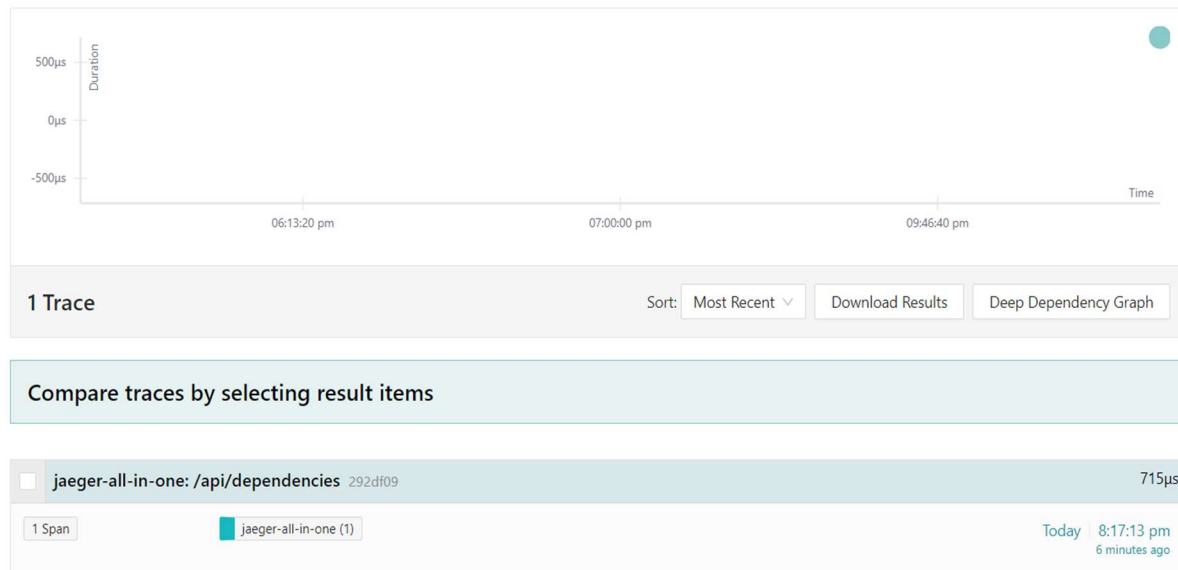
A screenshot of a terminal window with two tabs: 'server' and 'client'. The 'server' tab is active, showing the following output: 

```
"C:\Program Files\Amazon Corretto\jdk1.8.0_332\bin\java.exe" ...  
Server is listening on port 8080...  
Client connected.  
Expecting 20 files from the client.  
Checksum verification passed for file 1  
Checksum verification passed for file 2  
Checksum verification passed for file 3  
Checksum verification passed for file 4  
Checksum verification passed for file 5  
Checksum verification passed for file 6  
Checksum verification passed for file 7  
Checksum verification passed for file 8  
Checksum verification passed for file 9  
Checksum verification passed for file 10  
Checksum verification passed for file 11  
Checksum verification passed for file 12  
Checksum verification passed for file 13  
Checksum verification passed for file 14  
Checksum verification passed for file 15  
Checksum verification passed for file 16  
Checksum verification passed for file 17  
Checksum verification passed for file 18  
Checksum verification passed for file 19  
Checksum verification passed for file 20
```

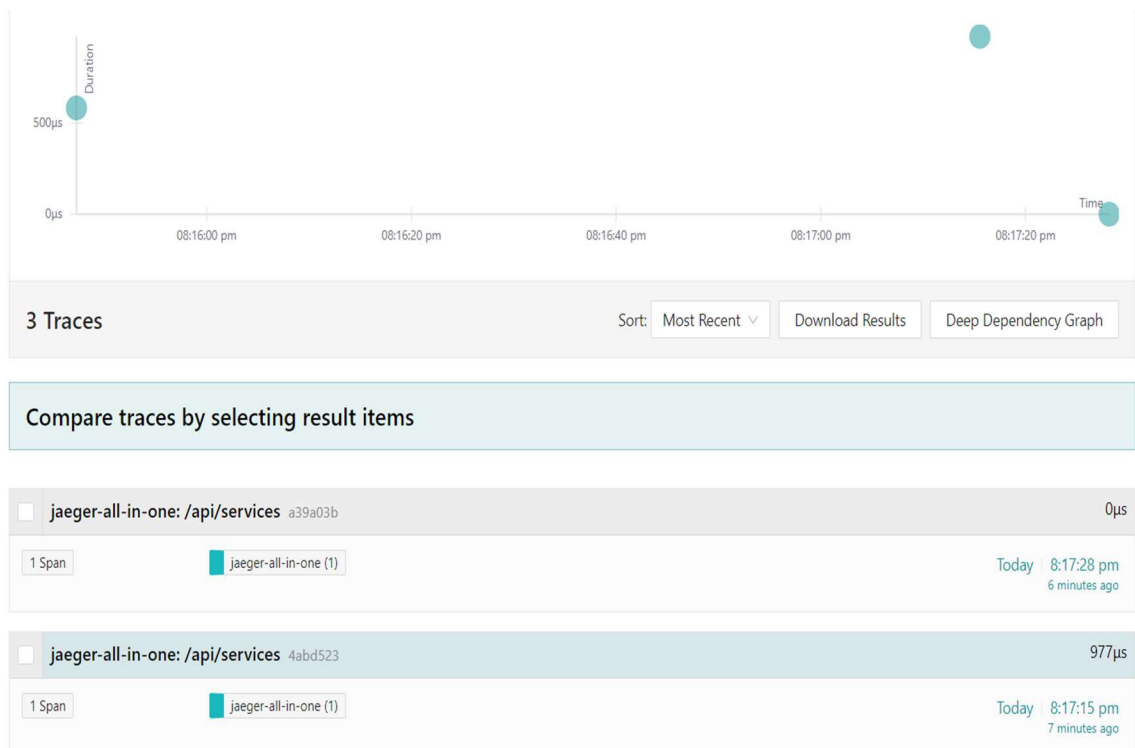
The transmission of compressed data with checksum verification is made easier by the integrated client-server architecture. Before delivering files in batches of up to 20 at a time, the client connects to the server, computes checksums for each file, and compresses the information using GZIP. Using Open Telemetry for distributed tracing, the server accepts connections, starts a new span for every action, and reads checksums and compressed file content from the client. The material is decompressed, checksums are verified for data integrity, and the checksum verification results are printed. Robust error handling is included in both the client and the server to handle any exceptions that may arise during connection setup and operation. This system offers distributed tracing to track and examine its behavior in addition to facilitating safe and effective file transfer.

## Jaeger Examples:

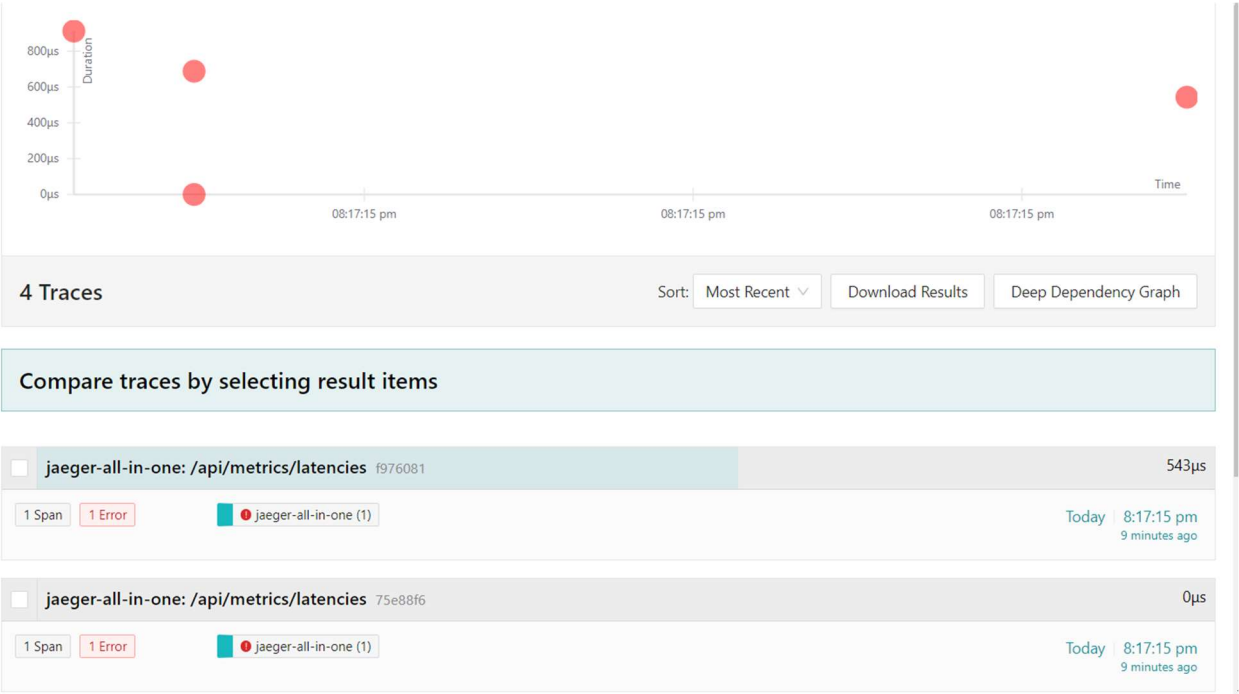
### Dependencies:



### Services:



Latencies:



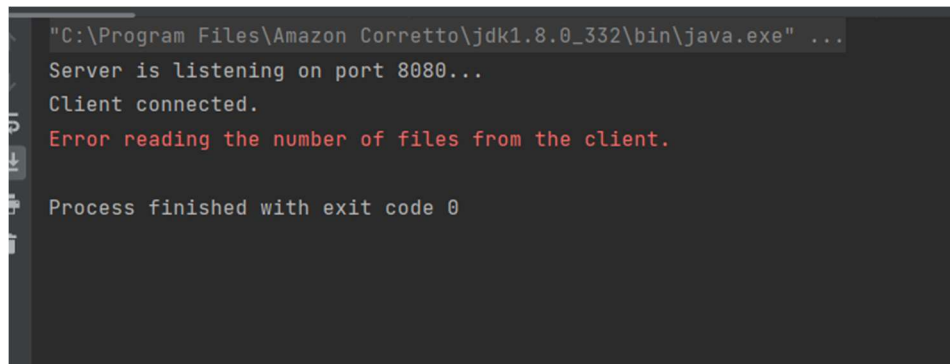


## **Question 2:**

### **Including the bug:**

In the code, the corrupt Compression method introduces a bug by writing "Corrupted data" instead of the actual compressed content. This simulates a scenario where the compression algorithm doesn't work correctly, leading to data corruption.

### **Example:**



```
"C:\Program Files\Amazon Corretto\jdk1.8.0_332\bin\java.exe" ...  
Server is listening on port 8080...  
Client connected.  
Error reading the number of files from the client.  
  
Process finished with exit code 0
```

### **How Statistical Debugging help?**

Using Statistical Debugging (SD) methods with Open Telemetry instrumentation, the intentional flaw in the corruptCompression function was carefully found. The intentional flaw entailed altering the compression algorithm to mimic a situation of data corruption. The technique was changed to output the text "Corrupted data" instead of publishing the actual compressed material. Utilizing Open Telemetry instrumentation while running the server application was essential to gather a wide range of data, such as distributed traces, metrics, and logs. With Open Telemetry set up to record these insights as the program ran, a wealth of data was available for further examination.

The next stage of the procedure involved analyzing the metrics and traces that had been gathered. Patterns and abnormalities connected to the intended defect were found by carefully analyzing the data. Through statistical analysis, metrics that showed failures, errors, or unusual behavior were identified, assisting in identifying potential problem locations where the introduced bug may have arisen. Open Telemetry's distributed traces and logs were useful resources for tracking down the execution flow. This thorough analysis of the application's runtime behavior allowed for the identification of any deviations from the predicted behavior.

The corrupt compression function was found to be the definitive source of the problem. The approach was improperly writing "Corrupted data" instead of the desired compressed content, as shown by the statistical analysis of the acquired data. The identification process was followed by the resolution stage. To ensure that the compression was handled correctly and that the compressed material was written as intended, the corrupt compression method was fixed.

Rerunning the application with Open Telemetry instrumentation allowed us to confirm that the deliberate bug had been fixed. This all-encompassing strategy, which combined statistical insights with Open Telemetry's distributed tracing capabilities, expedited the debugging process and made it easier to resolve the new issue.