Socket - Tri-Code Interoperability

Enhancing Java for High-Performance Server-Client Communication

Team

Utkarsh Satishkumar Shah
Sagar Alpeshkumar Patel

Introduction

The report is for the first mini-project for CMPE-275, and the goal of this report file is to discuss socket programming and to create client-server nodes using C++, Python, and Java languages. Along with that, we are also sharing findings and improvements that we have applied to the provided seed code.

Interoperability Matrix

Client/Server	Java	Python3	C/C++
Java	Y	Y	Y
Python3	Y	Y	Y
C/C++	Y	Y	Y

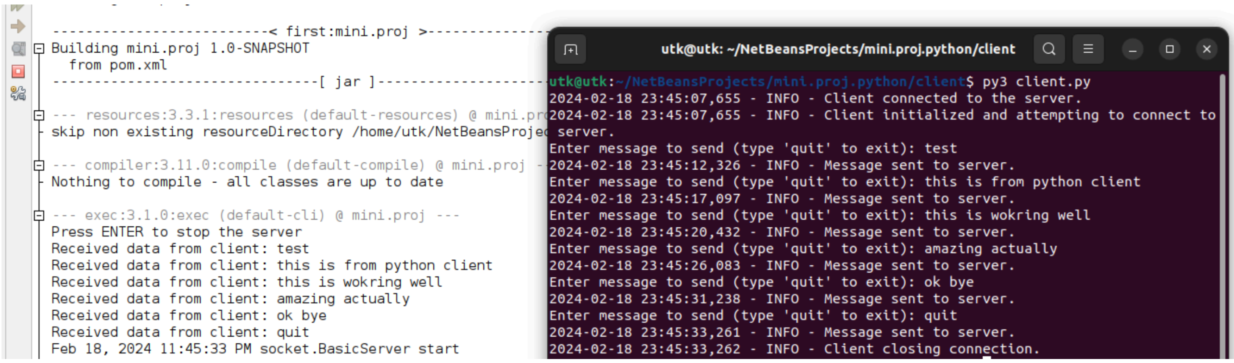
Snapshots for interoperability

Java Server and Java Client

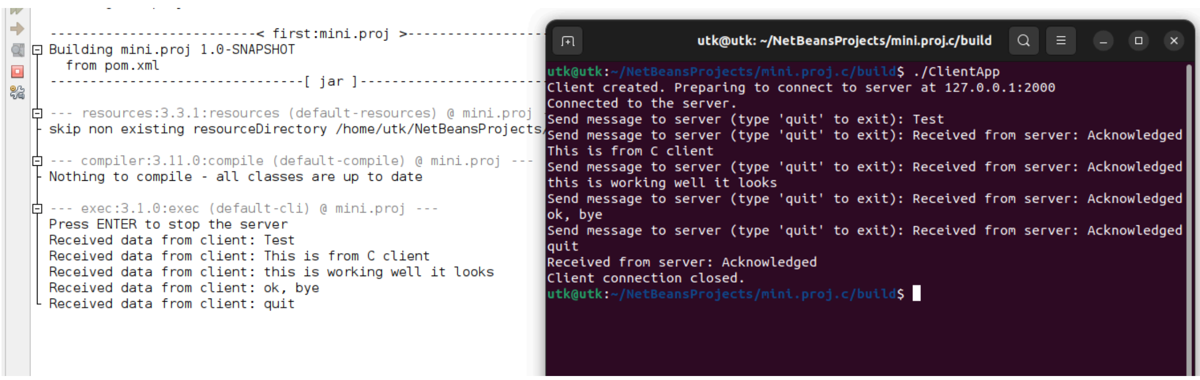
```
 --------------------------< first:mini.proj >---------------------------
] Building mini.proj 1.0-SNAPSHOT
    from pom.xml
 --------------------------------[ jar ]---------------------------------

] --- resources:3.3.1:resources (default-resources) @ mini.proj ---
- skip non existing resourceDirectory /home/utk/NetBeansProjects/mini.proj/src/main/resources

] --- compiler:3.11.0:compile (default-compile) @ mini.proj ---
- Nothing to compile - all classes are up to date

] --- exec:3.1.0:exec (default-cli) @ mini.proj ---
  Feb 18, 2024 11:07:55 PM socket.BasicClient connect
  INFO: Connected to 127.0.0.1

  enter message ('exit' to quit): I am a client in Java
  Received acknowledgement from server: Acknowledged

  enter message ('exit' to quit): Hola
  Received acknowledgement from server: Acknowledged

  enter message ('exit' to quit): Good Bye
  Received acknowledgement from server: Acknowledged

- enter message ('exit' to quit): exit
 ------------------------------------------------------------------------
  BUILD SUCCESS
 ------------------------------------------------------------------------
```

```
 --------------------------< first:mini.proj >---------------------------
 Building mini.proj 1.0-SNAPSHOT
    from pom.xml
 --------------------------------[ jar ]---------------------------------

 --- resources:3.3.1:resources (default-resources) @ mini.proj ---
 skip non existing resourceDirectory /home/utk/NetBeansProjects/mini.proj/src/main/resources

 --- compiler:3.11.0:compile (default-compile) @ mini.proj ---
 Nothing to compile - all classes are up to date

 --- exec:3.1.0:exec (default-cli) @ mini.proj ---
 Press ENTER to stop the server
 Received data from client: group_chat,app,I am a client in Java
 Received data from client: group_chat,app,Hola
 Received data from client: group_chat,app,Good Bye
```

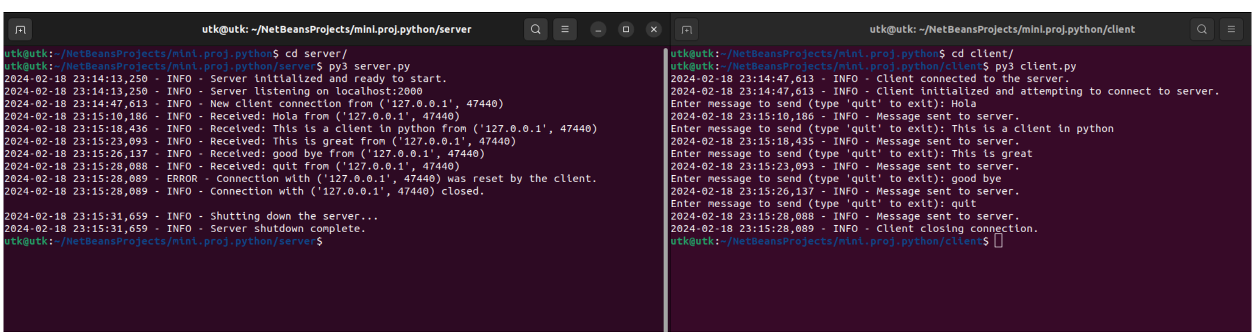
Java Server and Python Client



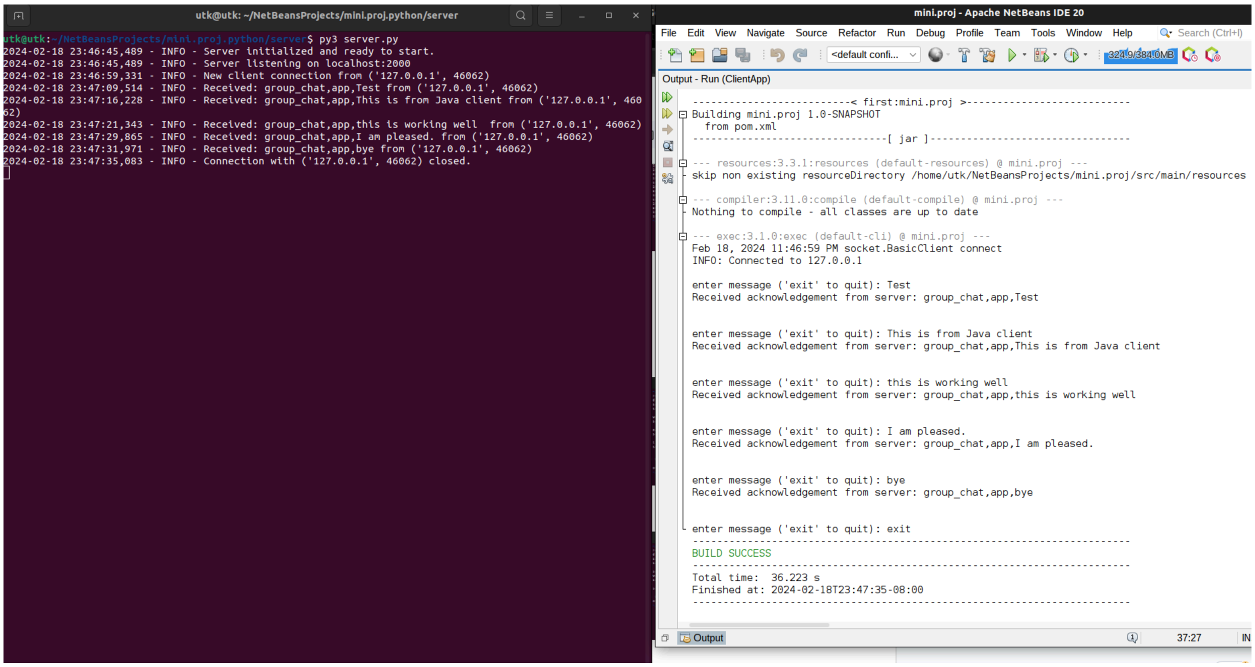
Java Server and C/C++ Client



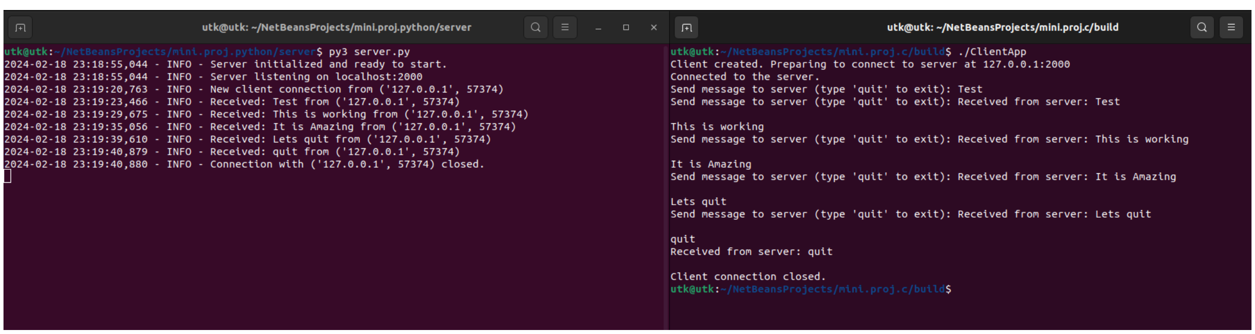
Python Server and Python Client



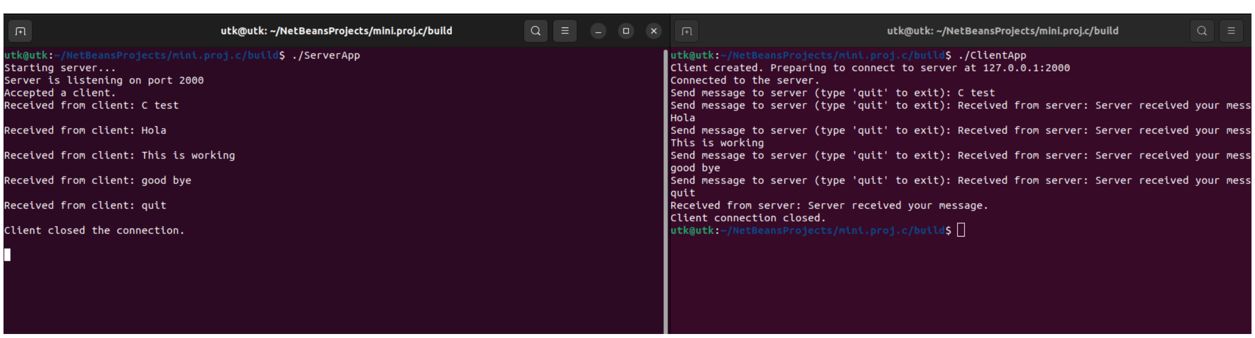
Python Server and Java Client



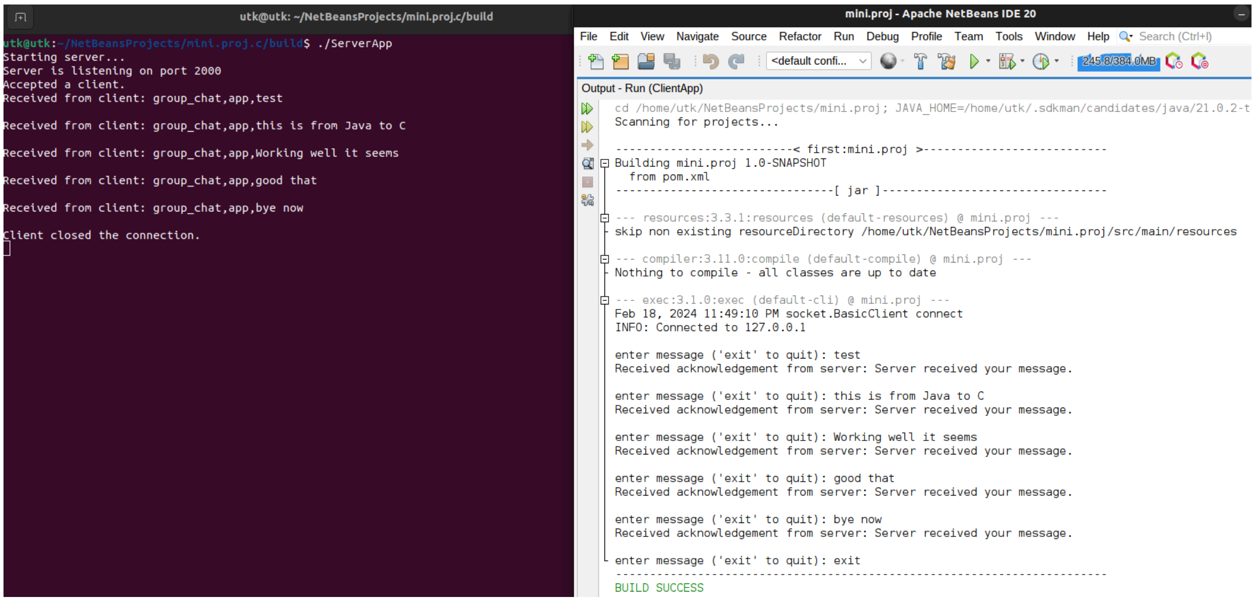
Python Server and C/C++ Client



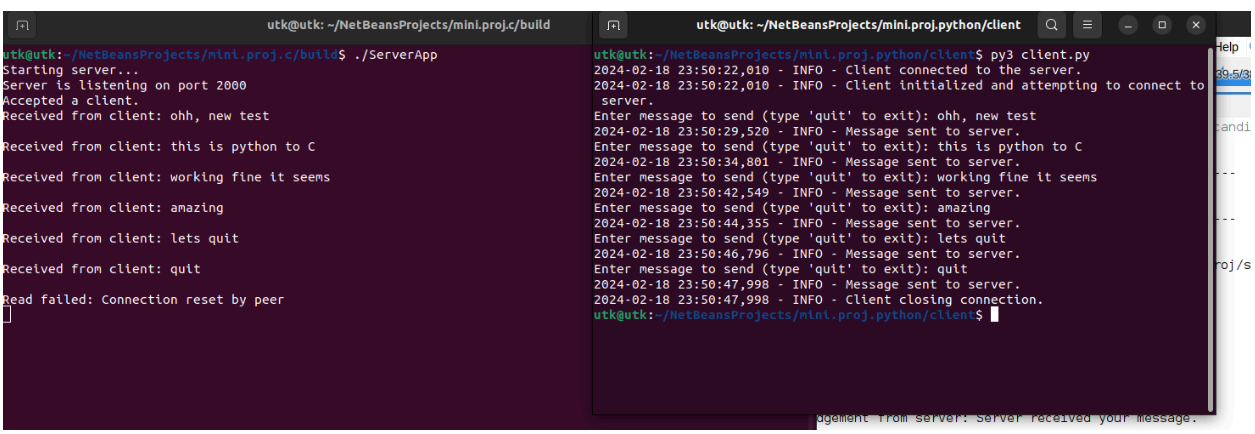
C/C++ Server and C/C++ Client



C/C++ Server and Java Client



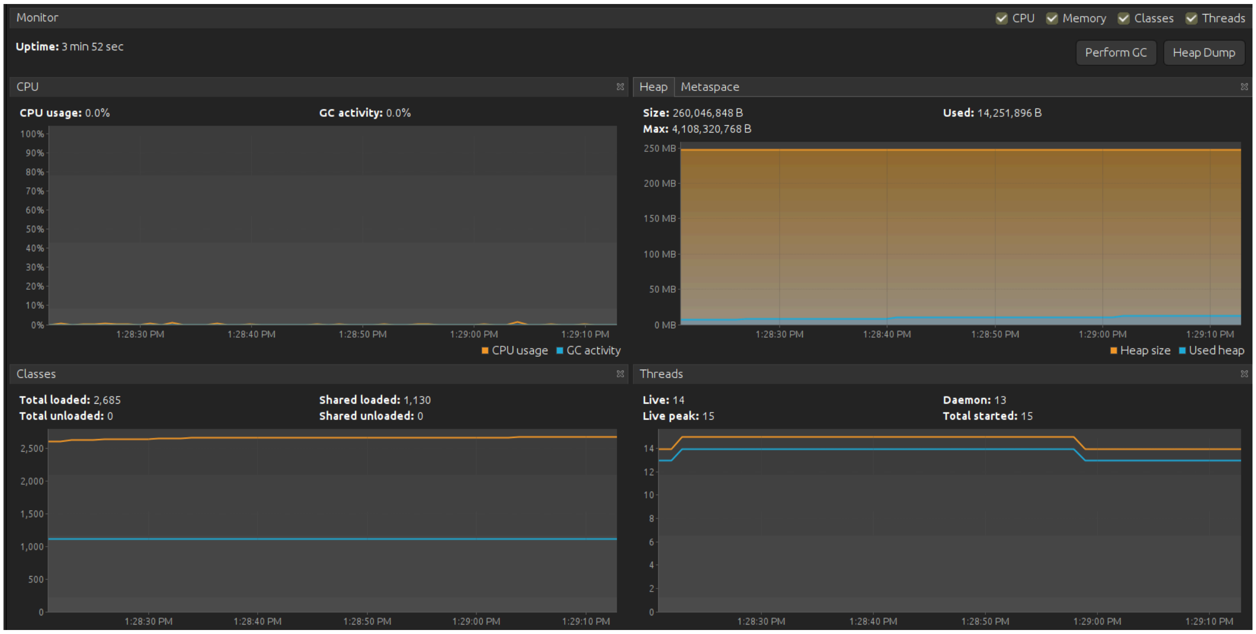
C/C++ Server and Python Client



Java Findings and Optimization

First Attempt

One Server and One Client Communication
VisualVM Profiling for CPU utilization, Heapspace, and GC activity



Server output and timetaken

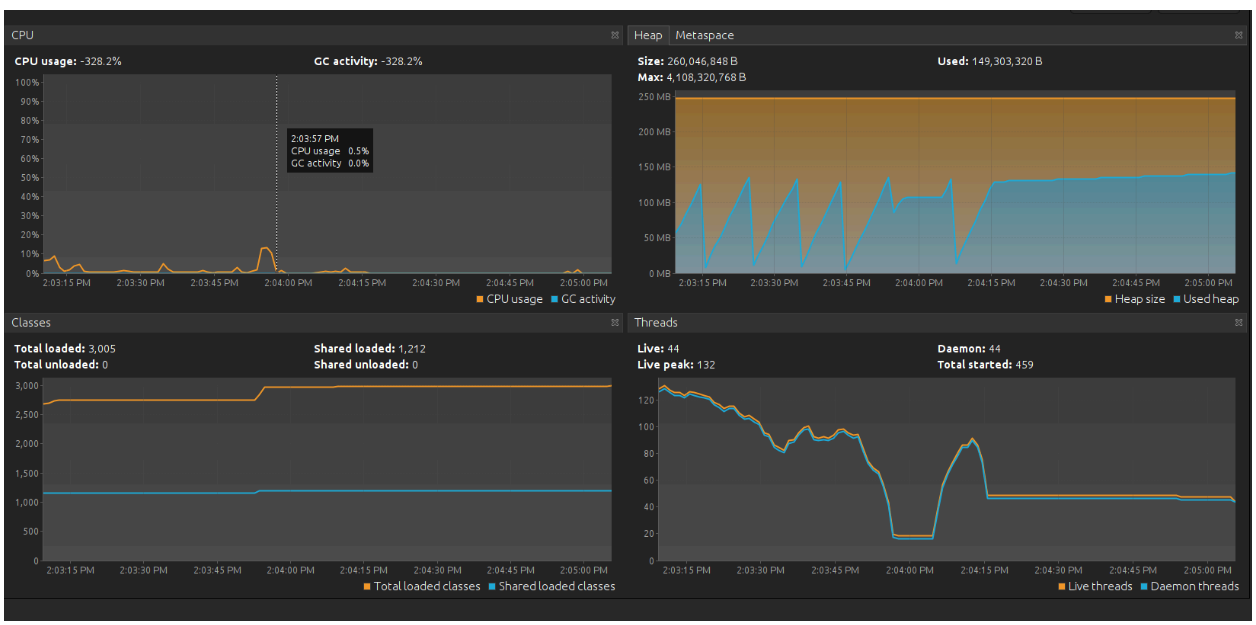
```
--- exec:3.1.0:exec (default-cli) @ mini.proj ---
Server Host: 0.0.0.0
Press ENTER to stop the server
--> server got a client connection
Session 21 started
from app, to group: pets/dogs, text: Hi there
from app, to group: pets/dogs, text: Hope it is all going well
from app, to group: pets/dogs, text: it is good to see you again
from app, to group: pets/dogs, text: Gotta go, see you soon.
from app, to group: pets/dogs, text: Good bye
Session 21 ending

java.net.SocketException: Socket closed
        at java.base/sun.nio.ch.NioSocketImpl.endAccept(NioSocketImpl.java:682)
        at java.base/sun.nio.ch.NioSocketImpl.accept(NioSocketImpl.java:755)
        at java.base/java.net.ServerSocket.implAccept(ServerSocket.java:698)
        at java.base/java.net.ServerSocket.platformImplAccept(ServerSocket.java:663)
        at java.base/java.net.ServerSocket.implAccept(ServerSocket.java:639)
        at java.base/java.net.ServerSocket.implAccept(ServerSocket.java:585)
        at java.base/java.net.ServerSocket.accept(ServerSocket.java:543)
        at socket.BasicServer.start(BasicServer.java:33)
        at java.base/java.lang.Thread.run(Thread.java:1583)
Total requests: 1
Average processing time: 28627725314 ns
------------------------------------------------------------------------
```

Findings and Improvements

First effort is to add some kind of testing code so that we can measure the performance as we move on to improve the code quality and remove bottlenecks. It is applied on the provided raw seedcode which is suboptimal to handle sessions and socket communication.

Running into issues while executing the test
Client - 1000
Messages per client - 10000

```
] java.net.SocketException: Broken pipe
        at java.base/sun.nio.ch.SocketDispatcher.write0(Native Method)
        at java.base/sun.nio.ch.SocketDispatcher.write(SocketDispatcher.java:62)
        at java.base/sun.nio.ch.NioSocketImpl.tryWrite(NioSocketImpl.java:394)
        at java.base/sun.nio.ch.NioSocketImpl.implWrite(NioSocketImpl.java:410)
        at java.base/sun.nio.ch.NioSocketImpl.write(NioSocketImpl.java:440)
        at java.base/sun.nio.ch.NioSocketImpl$2.write(NioSocketImpl.java:819)
        at java.base/java.net.Socket$SocketOutputStream.write(Socket.java:1195)
        at java.base/java.io.OutputStream.write(OutputStream.java:124)
        at socket.BasicClient.sendMessage(BasicClient.java:70)
        at app.TestClient.run(TestClient.java:19)
-       at java.base/java.lang.Thread.run(Thread.java:1583)
] java.net.SocketException: Broken pipe
        at java.base/sun.nio.ch.SocketDispatcher.write0(Native Method)
        at java.base/sun.nio.ch.SocketDispatcher.write(SocketDispatcher.java:62)
        at java.base/sun.nio.ch.NioSocketImpl.tryWrite(NioSocketImpl.java:394)
        at java.base/sun.nio.ch.NioSocketImpl.implWrite(NioSocketImpl.java:410)
        at java.base/sun.nio.ch.NioSocketImpl.write(NioSocketImpl.java:440)
        at java.base/sun.nio.ch.NioSocketImpl$2.write(NioSocketImpl.java:819)
        at java.base/java.net.Socket$SocketOutputStream.write(Socket.java:1195)
        at java.base/java.io.OutputStream.write(OutputStream.java:124)
        at socket.BasicClient.sendMessage(BasicClient.java:70)
        at app.TestClient.run(TestClient.java:19)
-       at java.base/java.lang.Thread.run(Thread.java:1583)
```



Encounted error "Broken pipe"
java.net.SocketException: Broken pipe at java.base/sun.nio.ch.SocketDispatcher.write0(Native Method) at java.base/sun.nio.ch.SocketDispatcher.write(SocketDispatcher.java:62) at java.base/sun.nio.ch.NioSocketImpl.tryWrite(NioSocketImpl.java:394) at

java.base/sun.nio.ch.NioSocketImpl.implWrite(NioSocketImpl.java:410) at java.base/sun.nio.ch.NioSocketImpl.write(NioSocketImpl.java:440) at java.base/sun.nio.ch.NioSocketImpl$2.write(NioSocketImpl.java:819) at java.base/java.net.Socket$SocketOutputStream.write(Socket.java:1195) at java.base/java.io.OutputStream.write(OutputStream.java:124) at socket.BasicClient.sendMessage(BasicClient.java:70) at app.TestClient.run(TestClient.java:19) at java.base/java.lang.Thread.run(Thread.java:1583)

Better exception handling and logging

We believe the error occurred on the client side since the server is getting overwhelmed with the number of connections or the rate of incoming data, and is closing connections to cope. If this is the case, we might need to optimize the code for session handling.

Utilize ThreadPool in Java

In the current implementation, a new thread is created for each client connection. This can be inefficient and resource-intensive if there are many client connections, as creating and destroying threads can be expensive.

A thread pool can help mitigate this issue. With a thread pool, a fixed number of threads are created and kept alive. When a task is submitted to the thread pool, it is executed by one of the existing threads. This can be more efficient than creating a new thread for each task.

Sending 10M Messages

Even after using ThreadPool, the program was again and again running into the BrokenPipe issue. We believe it is because the server is getting overwhelmed with the number of clients trying to communicate and the amount of messages received.

Switching to Non-blocking I/O (NIO)

Also, with non-blocking I/O, a single thread can handle multiple client connections. This is because when the thread performs an I/O operation, it doesn't block if the operation can't be completed immediately. Instead, it can move on to another task and come back to the operation later when it can be completed. This means you can handle multiple client connections with a single thread, eliminating the need for a thread pool.

The selector in NIO is used to monitor both the server socket channel and all connected client socket channels. When a client connects, the server accepts the connection, sets the client socket to non-blocking mode, and registers it to the selector for read events. When data is ready to be read from a client socket, the server reads the data and processes it. This allows the server to handle multiple client connections using a single thread.

Even after switching to NIO, we still received a BrokenPipe error on the client side while sending 10M messages with 1000 client numbers.

While fixing this we found that we need to implement some form of flow control in our application. One simple way to do this is to have the client wait for an acknowledgment from the server after sending a certain amount of data. Once the acknowledgment is received, the client can continue sending more data. This ensures that the server has a chance to process the incoming data and prevents it from getting overwhelmed.

This solution finally worked and we were able to send 10M requests from multiple clients to the server.

Bottleneck with 10000 threads

While increasing the number of threads, we again reached another bottleneck, and the communication was interrupted with an exception stating the socket is closed.

```
Feb 19, 2024 11:54:41 AM socket.BasicClient sendMessage
SEVERE: An error occurred while sending message
java.net.SocketException: Socket is closed
        at java.base/java.net.Socket.getOutputStream(Socket.java:1157)
        at socket.BasicClient.sendMessage(BasicClient.java:72)
        at app.TestClient.run(TestClient.java:24)
        at java.base/java.lang.Thread.run(Thread.java:1583)

Feb 19, 2024 11:54:41 AM socket.BasicClient sendMessage
SEVERE: An error occurred while sending message
java.net.SocketException: Socket is closed
        at java.base/java.net.Socket.getOutputStream(Socket.java:1157)
        at socket.BasicClient.sendMessage(BasicClient.java:72)
        at app.TestClient.run(TestClient.java:24)
        at java.base/java.lang.Thread.run(Thread.java:1583)
```

Test observations for the Java application

100 Threads, 1,000 messages
INFO: Processed 100,000 requests
INFO: Total duration time: 1,779,891,827 ns
INFO: Average request processing time: 17,798 ns

100 Threads, 10,000 messages
INFO: Processed 1,000,000 requests
INFO: Total duration time: 14,257,202,868 ns
INFO: Average request processing time: 14,257 ns

100 Threads, 100,000 messages
INFO: Processed 10,000,000 requests

INFO: Total duration time: 165,155,258,498 ns
INFO: Average request processing time: 16,515 ns

100 Threads, 100 messages
INFO: Processed 10,000 requests
INFO: Total duration time: 258,295,330 ns
INFO: Average request processing time: 25,829 ns

1000 Threads, 100 messages
INFO: Processed 100,000 requests
INFO: Total duration time: 1,642,501,560 ns
INFO: Average request processing time: 16,425 ns

10000 Threads, 100 messages
INFO: Processed 923,000 requests
INFO: Total duration time: 13,957,230,646 ns
INFO: Average request processing time: 15,121 ns

Citations

https://www.baeldung.com/a-guide-to-java-sockets
https://mailinator.blogspot.com/2008/02/kill-myth-please-nio-is-not-faster-than.html
https://stackoverflow.com/questions/7611152/nio-performance-improvement-compared-to-traditional-io-in-java
https://medium.com/@tesla8877/java-nio-vs-java-socket-exploring-advantages-and-disadvantages-in-network-communication-65565dd5d2c5
https://www.baeldung.com/java-io-vs-nio
https://s3-eu-central-1.amazonaws.com/ucu.edu.ua/wp-content/uploads/sites/8/2019/12/Petro-Karabyn.pdf

Python Findings and Optimization

Summary of Server, Client

1. Server.py: This script sets up a server that listens for incoming client connections. It uses a thread pool to handle multiple clients concurrently. Each client is handled in a separate thread where the server receives messages from the client, logs them, and sends an acknowledgment back to the client. The server keeps track of all active sessions and their messages.

2. Client.py: This script sets up a client that connects to the server. It sends messages input by the user to the server and logs the server's responses. The client can send messages continuously until the user types 'quit'.

Features, Tools, and Libraries

Import Statements: The import keyword is used to include Python's built-in libraries such as logging, socket, concurrent.futures, and threading.

Logging: The logging module is used to record events happening in the server and client. It's configured with basicConfig to set the level and format of the logs.

Socket Programming: The socket module is used to create a TCP/IP socket, bind it to an address, listen for incoming connections, accept a connection, send data to the client, and receive data from the client.

Exception Handling: try/except/finally blocks are used to catch and handle potential runtime errors, improving the robustness of the code.

String Formatting: The f-string syntax is used to insert variables into strings. This is a convenient way to create complex strings.

Multithreading: The concurrent.futures.ThreadPoolExecutor is used to create a pool of worker threads. The threading.Lock is used to ensure thread-safe access to shared resources.

String Encoding/Decoding: The encode and decode methods are used to convert strings to bytes and vice versa. These are built-in methods provided by Python's str and bytes classes.

Essential Parts

Socket Programming: The creation of a socket, binding it to an address, listening for connections, accepting a connection, and sending and receiving data are all essential parts of a server and client.

Multithreading: The use of a thread pool to handle multiple clients concurrently is essential for a scalable server.

Exception Handling: Proper handling of exceptions is essential to prevent the server and client from crashing when errors occur.

Session Management: The server maintains a dictionary of sessions, which is essential for tracking client interactions.

Locks: The use of a lock to ensure thread-safe access to shared resources is essential in a multithreaded environment

Citations

https://realpython.com/python-sockets
https://realpython.com/intro-to-python-threading/#using-a-threadpoolexecutor/
https://docs.python.org/3/tutorial/errors.html
https://docs.python.org/3/library/threading.html#lock-objects
https://docs.python.org/3/howto/sockets.html
https://codefellows.github.io/sea-python-401d4/lectures/async.html
https://www.datacamp.com/tutorial/a-complete-guide-to-socket-programming-in-python

C++ Findings and Optimization

Summary of the Server, Client, and Session Handler

1. Server:
The Server file is responsible for setting up the server, accepting incoming client connections, and managing these connections. It uses the POSIX socket API for networking, and it creates a new thread for each client to handle communication concurrently.

Key Features:

- Socket Programming: The server uses sockets for communication. It sets up a socket, binds it to a specific IP address and port, and listens for incoming connections.
- Multithreading: For each client that connects, the server creates a new thread to handle the client's requests. This allows the server to handle multiple clients concurrently.
- Session Management: The server uses the Session Handler to manage sessions for each client.

2. Client:
The Client file is responsible for setting up the client and communicating with the server. It also uses the POSIX socket API for networking.

Key Features:

- Socket Programming: The client sets up a socket and connects to the server using the server's IP address and port.
- Communication: The client sends requests to the server and receives responses. This communication is done using the send and read functions.

3. Session Handler:
The Session Handler file is responsible for managing sessions. A session starts when a client connects to the server and ends when the client disconnects. The Session Handler keeps track of all active sessions.

Features

- Session Management: The Session Handler can create, end, and print all active sessions. It also generates a unique session ID for each session.
- Thread-Safety: The Session Handler uses a mutex to ensure that updates to the sessions are thread-safe. This is important because multiple threads (each handling a different client) may try to update the sessions concurrently.

Libraries

1. <iostream>: Used for input/output operations.
2. ‹string>: Provides the std:: string class.
3. ‹vector>: Provides the std:: vector container.

4. <thread>: Used for working with threads.
5. <mutex>: Provides the std: :mutex and std:: lock_guard classes for synchronizing data access among threads.
6. ‹chrono>: Provides utilities to deal with time.
7. <ctime>: Provides functions to get and manipulate date and time information.
8. ‹ma>: Provides the std:: map container.
9. ‹netinet/in.h>: Contains constants and structures needed for internet domain addresses.
10. <unistd.h>: Provides access to the POSIX operating system API.
11. ‹cstring>: Provides functions to manipulate C strings and arrays.
12. <cerrno>: Provides the macro errno, which is used to report error conditions.
13. ‹atomic>: Provides components for fine-grained atomic operations.

Linker

The linker is a tool that combines multiple object files (produced by the compiler) into a single executable or library. It resolves references between object files, such as function calls and global variables, ensuring that all code dependencies are correctly mapped.
Once source files are compiled into object files, the linker takes these object files as input. It then performs symbol resolution (matching function calls with their definitions) and relocation (adjusting code and data references to fit the executable's memory layout). The output is an executable file or a library that can be run on a system.
Key Commands executed:

- clang++ -c -Iinclude src/Server.cpp src/SessionHandler.cpp src/main_server.cpp for compiling into object files.
- clang++ Server.o SessionHandler.o main_server.o -o server_program for linking object files into an executable.

Output: Executable files (server_program, client_program) that can be run to perform the tasks implemented in the source code.

Static Analyzers

Static analyzers examine source code for potential errors, bugs, or vulnerabilities without executing the code. They use various analysis techniques to detect issues such as memory leaks, buffer overflows, and logical errors, helping developers to fix problems early in the development process.
The Clang static analyzer, for example, parses the source code, builds an abstract syntax tree (AST), and performs checks against a set of rules to identify potential issues. The analysis can be customized with different checkers to focus on specific types of problems.
Key Commands Used :

- clang++ --analyze -Xanalyzer -analyzer-output=html -Iinclude src/Server.cpp src/SessionHandler.cpp to analyze source files and generate reports.

Output: Reports in various formats (e.g., HTML, plist) detailing potential issues found in the code. These reports include information about the nature of the issues, their locations, and sometimes suggestions for fixes.

Sanitizers

Sanitizers are dynamic testing tools that detect various types of errors such as memory corruption, undefined behavior, and data races during the execution of a program. They are instrumental in identifying issues that might not be caught by static analysis or are difficult to reproduce in a debugging environment.

When you compile and link your program with a sanitizer enabled (using specific compiler flags), the compiler inserts additional checks into the generated code. These checks are designed to detect and report runtime errors related to memory, behavior, or threading issues.

Key Commands:

- Compilation with sanitizers: clang++ -fsanitize=address -Iinclude src/Server.cpp src/SessionHandler.cpp -o server_program for AddressSanitizer.
- Execution of instrumented executables to detect runtime issues.

Output: Detailed error reports printed to the terminal or log files when the instrumented program encounters an error during its execution. These reports include information about the type of error, the location where it occurred, and sometimes a stack trace.

Incorporating linkers, static analyzers, and sanitizers into our project has significantly enhanced its quality. These tools helped us stitch together our code into working software, spot potential issues before they became problems, and catch errors that only show up when the software is running.

Potential Improvements

1. Concurrency Management: Use a thread pool to handle clients and consider asynchronous I/O operations.
2. Logging: Record more detailed logs, use a logging library for better log management, and consider a centralized logging system for easier analysis.

Citations

https://www.geeksforgeeks.org/socket-programming-cc/
https://www.geeksforgeeks.org/socket-programming-in-cc-handling-multiple-clients-on-server-without-multi-threading/
https://rvarago.medium.com/introduction-to-cmake-for-cpp-4c464272a239
Chat GPT - Bug fixing, Understanding and how to usa (sanitizer, linker, analyzer), other key aspects of the socket programming

Contribution

For our group project, Sagar and Utkarsh, who are roommates, worked together for the most part. We spent a lot of time working side-by-side project, using what each of us was good at to make the project better.

Utkarsh was good with Java. He focused a lot on learning all the necessary parts and used that knowledge to make sure those parts of our project worked well.

Sagar took charge of the Python socket programming part, having knowledge of networks. He used his knowledge and implemented a client-server connection to exchange data smoothly.

When it came to C++, we both worked together. We knew that combining our knowledge would help us deal with the tricky bits of C++. Working together on C++ helped us come up with better ideas and solutions.

For the project's presentation, Sagar made the poster, making sure it showed what our project was about and what we achieved. Utkarsh wrote the project report, carefully explaining everything we did, what we found out, and our final thoughts. We both checked to make sure that the parts of our project using Java, Python, and C++ worked well together.

Working together this way helped us learn a lot about different programming languages and made our working relationship stronger. It all helped make our project successful.