Madison Threadgill **Distributed and Fault Tolerant Systems** April 24, 2023

## Goals

The original aim of this lab was to create and implement a distributed client-server system using two microcontrollers and a central server. However, as this lab progressed it was observed that the provided ESP 8266 Wi-Fi module was unreliable, failing to hold a connection throughout the entire execution of the program. Additionally, if the ESP 8255 Wi-Fi module succeeded in staying connected throughout the execution of the program, the program on the microcontroller clients would hard fault around 75% of the time, preventing successful execution of the distributed workload. Therefore, it was determined that instead of distributing tasks over multiple microcontrollers, a fault-tolerant system needed to be developed instead in order to avoid the loss of tasks in the system. Moreover, this lab aims to create and implement a client-server fault-tolerant system using two microcontrollers and a central server.

## Background

Fault tolerance within embedded and distributed systems is highly prominent in literature [2, 3, 1]. The survey in [2] details basic fault tolerance concepts and implementation examples for distributed systems. Using the definition of a distributed system from [2], we will describe a distributed system as a group of independent processing elements that appear to the user as a single system. Additionally, distributed systems have the same goal for the tasks they are processing. In this lab, the single goal for the distributed system will be executing multiple matrix multiplication tasks. The reason for choosing matrix multiplication is that it simulates a machine-learning workload. By observing how the system executes matrix multiplication tasks, it can provide insight into how machine learning algorithms can be mapped to the same system in the future. The work in [2] also describes two types of fault tolerance: reactive and proactive fault tolerance. Reactive fault tolerance simply recovers the state of the program before the fault occurred whereas proactive fault tolerance predicts faults and replaces faulty components with properly working components. In this lab, we will focus on reactive fault tolerance. More specifically, we will be using the fault tolerant technique called retry. When a failure occurs, the retry technique simply tries execution of the program again, beginning where the fault occurred. In our case, the program will be retried on a separate microcontroller.

## Procedure

### Overview

First, a simple client-server system was created. This system can be seen in Figure 1. The server contains a task queue that contains the matrices to be multiplied together. Upon the start of the system, the server waits for a connection from a client. When this connection is received, the client then sends a signal that it is ready for a task. The server then checks the task queue, and if the task queue contains tasks to be processed, the task is then sent to the client. If the tasks queue is empty, the client shuts down and the program ends execution. Once the client receives the task, the task is processed and the result (multiplied matrices) are sent back to the server. The server then adds this task to the result queue. This process continues until there are no more tasks left in the task queue. Upon the end of the program, in order to verify the correctness of results as well as the order that the results were sent back to the server, the matrix multiplication for every original task is calculated on the server and added to a compare queue. Finally, the compare queue is compared to the results queue within the server. If the compare queue contents are equal to the result queue contents, we know that the program was completed deterministically and correctly.

With more than three tasks, the simple client-server system would not complete execution around 75% of the time. This problem arose due to the client hard faulting or the ESP 8266 Wi-Fi module failing to receive or send data to/from the central server. Therefore, creating a system with fault tolerance was deemed necessary. The fault-tolerance implementation simply includes another microcontroller that servers as a backup client. Additionally, the main client would now send a message to the central server at a specified interval to let the server know the main client is still active and ready to receive and process tasks. Once the main client fails to send a message to the server, the backup client is activated and the unfinished task sent to the main client was recovered. The backup client then continues execution similar to the main client.

The system with fault-tolerance added is shown in Figure 2. At least every second, the main client sends a heartbeat message to the central server. Once this message is received, the server updates the time it last heard from the main client. If the server does not hear from the client in more than two seconds, the server

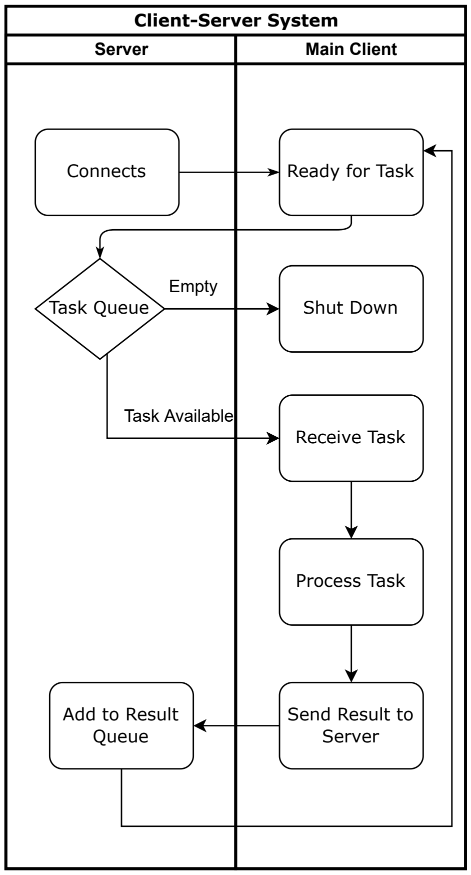

Figure 1: Client-Server System Without Fault-Tolerance

assumes that the client has faulted. The tasks most recently sent to the client that was unfinished is recovered from the client. The backup client is then activated. Once the backup client received the message from the server that it has been activated, it tells the server that it is ready for a task. The backup client then acts just as the main client, continuously processing and requesting tasks until the task queue is empty. Finally, the verification process begins similarly to the main client. Moreover, if the compare queue is equal to the results queue, we know that the fault-tolerant system executed correctly.

## Specification Details

### Server Side

The server was created using Python and included the socket, threading, queue, time, numpy, and select libraries. Three queues were using on the server side including the task, done, and compare queue. The task queue held tasks that needed to be sent and processed by the clients. The done queue held tasks that have

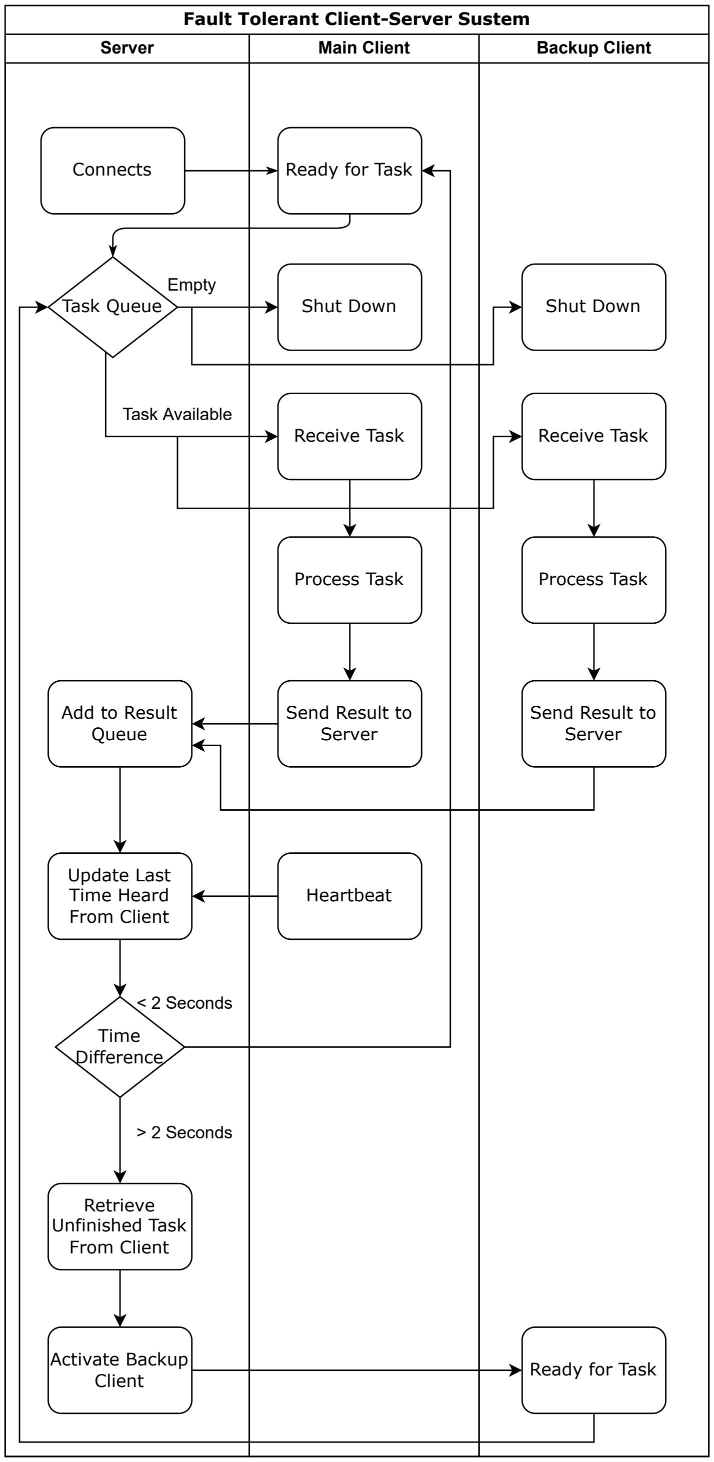

Figure 2: Fault Tolerant Client-Server System

been completed by the clients and were sent back to the server. The compare queue held the results from the matrix multiplications that were calculated on the server. This queue was used for validation purposes. At the end of the program execution, the compare queue and the done queue were compared and if the queues were equivalent, we know the program executed successfully.

The socket library was used in order to create a socket to listen for incoming connections from clients. The server was hosted locally on a laptop using port 6000. Therefore, any device on the same WiFi as the laptop could connect to the server using the generated host name. The socket was set to listen to at most two connections from clients, one for the main client and one for the backup client.

The numpy library was simply used for matrix multiplication calculations and matrix creation on the server side. A task sent to the clients was composed of the dimensions of each matrix as well as the contents of each matrix. This format was selected due to the fact that the clients are running c code, so in order to loop through the matrices sent over WiFi the size of the matrices being sent must be known beforehand.

The threading library was used as each client created in the server program was its own thread. Therefore, the main client and the backup client can execute tasks simultaneously if necessary. This is useful upon the startup of the program. Each client can connect to the server and say they are ready at the same time. The threading environment is also useful for future work if more main clients are added. When this, parallel execution of tasks is possible so work can be distributed over multiple clients, increasing the throughput of the program.

The time and select libraries were used to check if the main client stopped execution. Every time the heartbeat message is sent to the server, the last time the client was active is updated. The select library was used as a way to avoid deadlocks while waiting for the client to respond to the server. When the server listens for a response from the main client, if the response takes longer than two seconds, we assume that the main client has faulted and activated the backup client. This is used in conjunction with the heartbeat message to ensure that the backup client is activated when the main client fails.

In order to recover the lost task that was sent to the main client if the main client fails, a variable holds the most recent task taken out of the queue and sent to the client. This way, if the client fails, when the backup client begins, instead of taking the first task from the queue, the first task the backup client executes is in the temporary variable. After the task is recovered and processed by the backup client, the backup client then takes tasks from the task queue.

### Client Side

For both the main client and the backup client, the real-time operating system created throughout the class was used. The only additions were a new thread that connects to the server and begins executing tasks. Within this thread, the ESP 8266 driver was used to send messages and receive tasks from the server. A task parser function was created that reads in the task and converts the matrices to integers. Another function was created to multiply the matrices together. Finally, a function was created that converts the final product matrix into a character format that can be received by the server. The only difference between the main client and the backup client threads is the fact that the backup client does not start processing tasks until an activate message is sent by the server. The main client begins executing tasks sent by the server immediately after the connection is established.

## Results

In order to verify the results of the program, program output was displayed showing the current status of what is happening within the program. The test program contained 5 matrix multiplication tasks that needed to be completed by the clients. The output of the program is shown below.

```
Client 10.159.64.223 requested a task
Sent task to client
Message from client 10.159.64.223: heartbeat
Client 10.159.64.223 finished a task
Put task into done queue
Size of done queue: 1
Client 10.159.64.223 requested a task
Sent task to client
Message from client 10.159.64.223: heartbeat
Client 10.159.64.223 finished a task
Put task into done queue
Size of done queue: 2
Client 10.159.64.223 requested a task
Sent task to client
Message from client 10.159.64.223: heartbeat
Client 10.159.64.223 finished a task
Put task into done queue
Size of done queue: 3
Client 10.159.64.223 requested a task
Sent task to client
Message from client 10.159.64.223: heartbeat
```

```
22 Client failed to send message, calling for backup
23 Backup client activated
24 Client disconnected
25 Backup client 10.159.64.223 requested a task
26 Backup client received current task
27 Sent task to backup client
28 Message from backup client 10.159.64.223: heartbeat
29 Backup client 10.159.64.223 finished a task
30 Put task into done queue
31 Size of done queue: 4
32 Backup client 10.159.64.223 requested a task
33 Backup client received current task
34 Sent task to backup client
35 Message from backup client 10.159.64.223: heartbeat
36 Backup client 10.159.64.223 finished a task
37 Put task into done queue
38 Size of done queue: 5
39 Backup client 10.159.64.223 requested a task
40 No tasks available
41 Backup disconnected
42 Comparing results...
43 Results match!
```

The program output consists of communication to and from the clients as well as an update on the size of the queue that contains the completed tasks. As can be seen by the output, the main client failed at line 22. This is because it took longer than five seconds to receive data from the client. The backup client was activated in line 23 with the main client being disconnected at line 24. As can be seen in line 38, the size of the done queue was equivalent to the original number of tasks that the client received. Therefore, when the backup client requested another task, the server sent a message in line 40 saying there are no more tasks available. Subsequently, in line 41 the backup client was disconnected and the matrix multiplication results received from the client were compared to the matrix multiplication results calculated on the server. In line 43, the server stated that the results match, meaning that the program executed deterministically and successfully.

## Conclusion

In this lab, a fault tolerant system involving two microcontrollers and a central server was created. The main workload of the system was matrix multiplication, simulating what would be seen in a distributed machine-learning framework. The fault-tolerant system ensured that when one client failed, the task provided to the client was restored and sent to the backup client where regular program execution continued. The main problems in this lab consisted of implementing the ESP 8266 WiFi modules into the system as these modules are unreliable. Additionally, dealing with a system's scheduling and race conditions proved challenging as an ideal system should be deterministic. Since the complexity of this lab was to be equivalent to one of the labs in the course, implementing the fault-tolerant system was the ultimate end goal of the lab. Moreover, future work of this lab would be adding task distribution by adding more than one main client in the system. Additionally, instead of matrix multiplication simulating machine learning workloads, perhaps exploring partitioning a machine learning network over multiple clients would be interesting due to the low memory capacity of the microcontrollers.

## References

[1] Falk Salewski and Adam Taylor. "Fault Handling in FPGAs and Microcontrollers in Safety-Critical Embedded Applications: A Comparative Survey". In: *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*. Aug. 2007, pp. 124–131. DOI: `10.1109/DSD.2007.4341459`.

[2] Abdeldjalil Ledmi, Hakim Bendjenna, and Sofiane Mounine Hemam. "Fault Tolerance in Distributed Systems: A Survey". In: *2018 3rd International Conference on Pattern Analysis and Intelligent Systems (PAIS)*. Oct. 2018, pp. 1–5. DOI: `10.1109/PAIS.2018.8598484`.

[3] M Shyama and Anju S Pillai. "Fault Tolerance strategies for Wireless Sensor Networks – A Comprehensive Survey". In: *2018 3rd International Conference on Inventive Computation Technologies (ICICT)*. Nov. 2018, pp. 707–711. DOI: `10.1109/ICICT43934.2018.9034298`.