Firefighting Drone Swarm

SYSC 3303 A
Lab Section: A3
Group: L3

Due: April 8, 2025

Team Members:
Brendan Bower: 101220993
Abolarinwa Elegbede: 101188618
Zeena Ford: 101229954
Jaydon Haghighi Saed: 101206884
Raiqah Islam: 101226820
Leen Youssef: 101264371

Table of Contents

1.0 Breakdown of Team Member Responsibilities

1.1 Iteration 1:

Brendan: Implemented the Scheduler class
Abolarinwa: Wrote test cases for Scheduler class and documentation
Zeena: Wrote test cases for FireIncidentSubsystem class and designed sequence diagrams
Jaydon: Implemented DroneSubsystem and FireEvent class, designed UML class diagram
Raiqah: Wrote test cases for FireEvent and DroneSubsystem classes
Leen: Implemented FireIncidentSubsystem class

1.2 Iteration 2:

Brendan: Designed sequence diagrams and state machine diagrams for drone and scheduler
Abolarinwa: Implemented core scheduling logic to decide which drone handles a new fire
Zeena: Implemented Drone State transitions
Jaydon: Updated UML class diagram and ReadMe file, reviewed and documented final code
Raiqah: Implemented Drone State transitions
Leen: Conducted testing on entire system

1.3 Iteration 3:

Brendan: Implemented UDP into the system
Abolarinwa: Implement core scheduling logic to coordinate movement of drones
Zeena: Updated documentation and ReadMe file
Jaydon: Implement core scheduling logic to coordinate movement of drones
Raiqah: Updated UML class diagram with UDP attributes and methods
Leen: Conducted testing on entire system

1.4 Iteration 4:

Brendan: Implemented fault injections and updated UML class diagram
Abolarinwa: Implemented fault handling for nozzle-jam
Zeena: Implemented fault handling for drone stuck in mid-air
Jaydon: Created timing diagram, fixed bugs, and performed overall code review
Raiqah: Implemented fault handling for drone stuck in mid-air and updated ReadMe file
Leen: Implemented fault handling for packet-loss

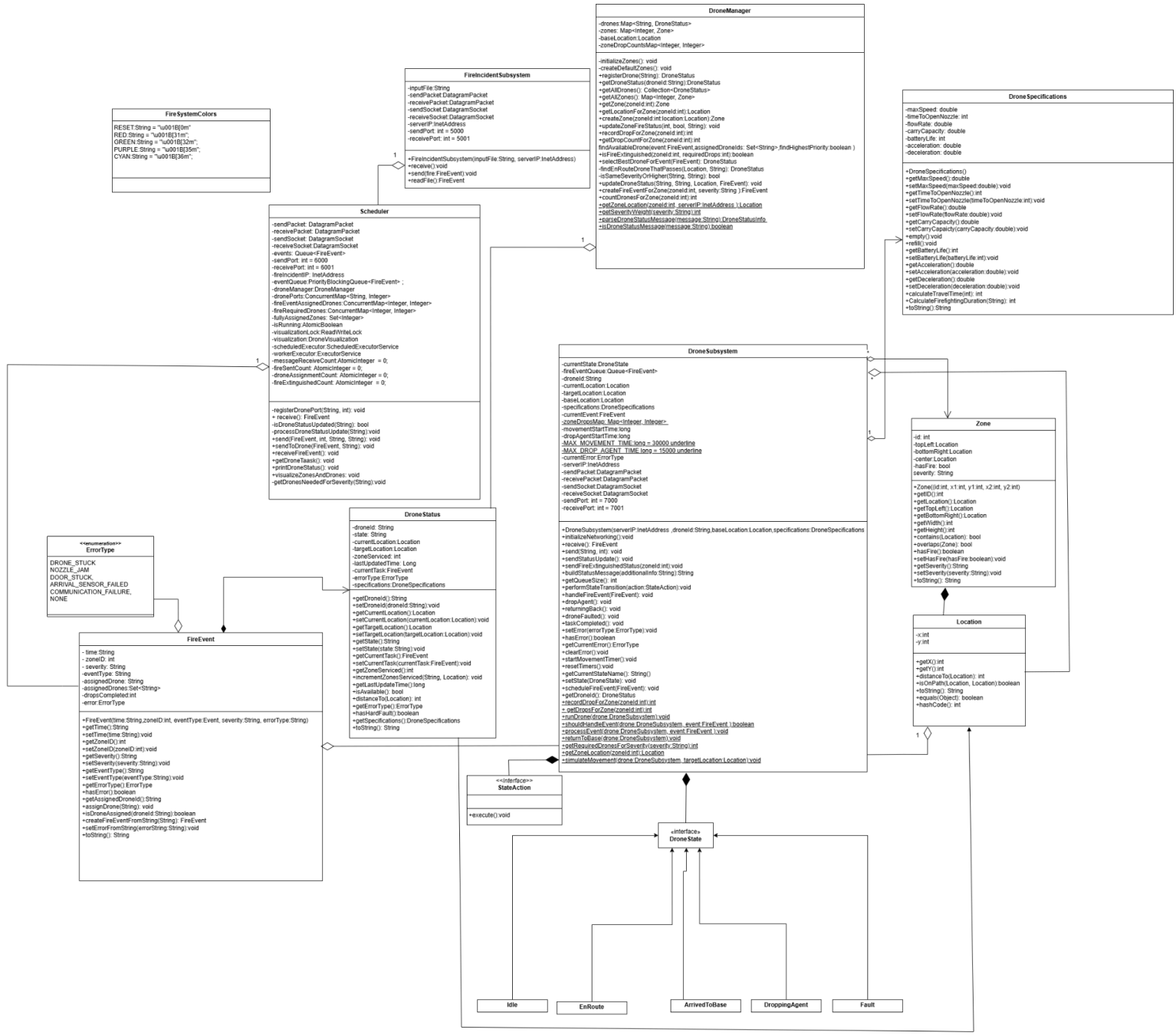
1.5 Iteration 5:

Brendan: Updated State Machine diagram, Re-implemented fault injection
Abolarinwa: Helped implement UI and metrics
Zeena: Updated Sequence Diagram
Jaydon: Implemented user interface and metrics
Raiqah: Updated UML Class Diagram

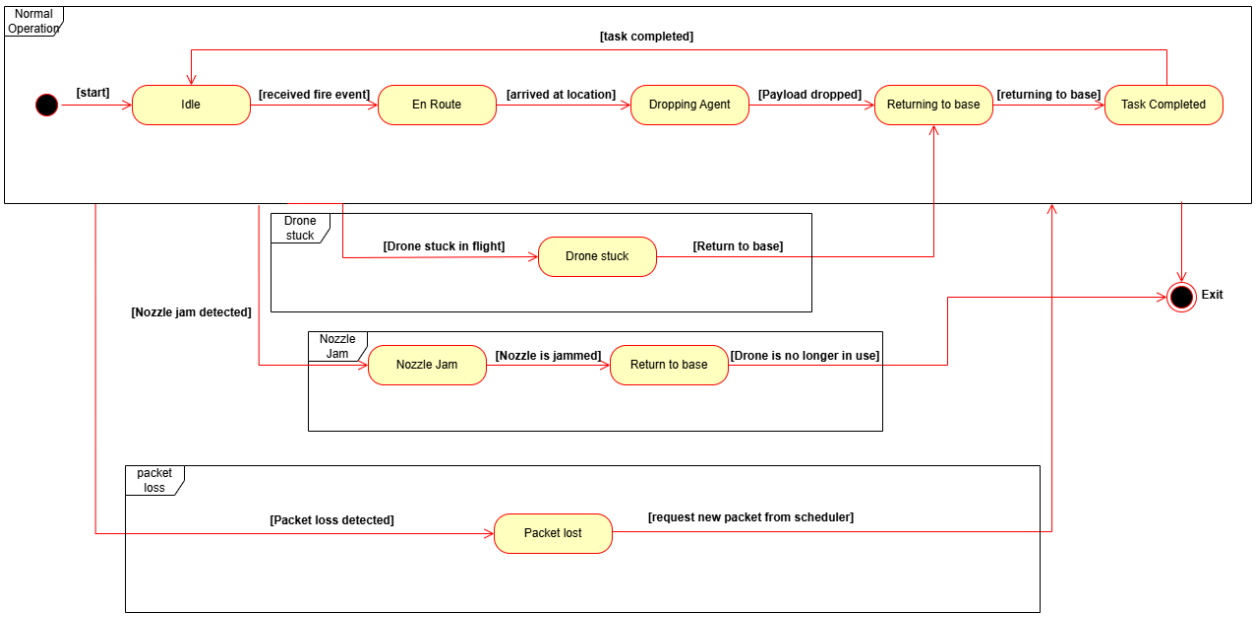
Leen: Updated Sequence Diagram

2.0 Diagrams

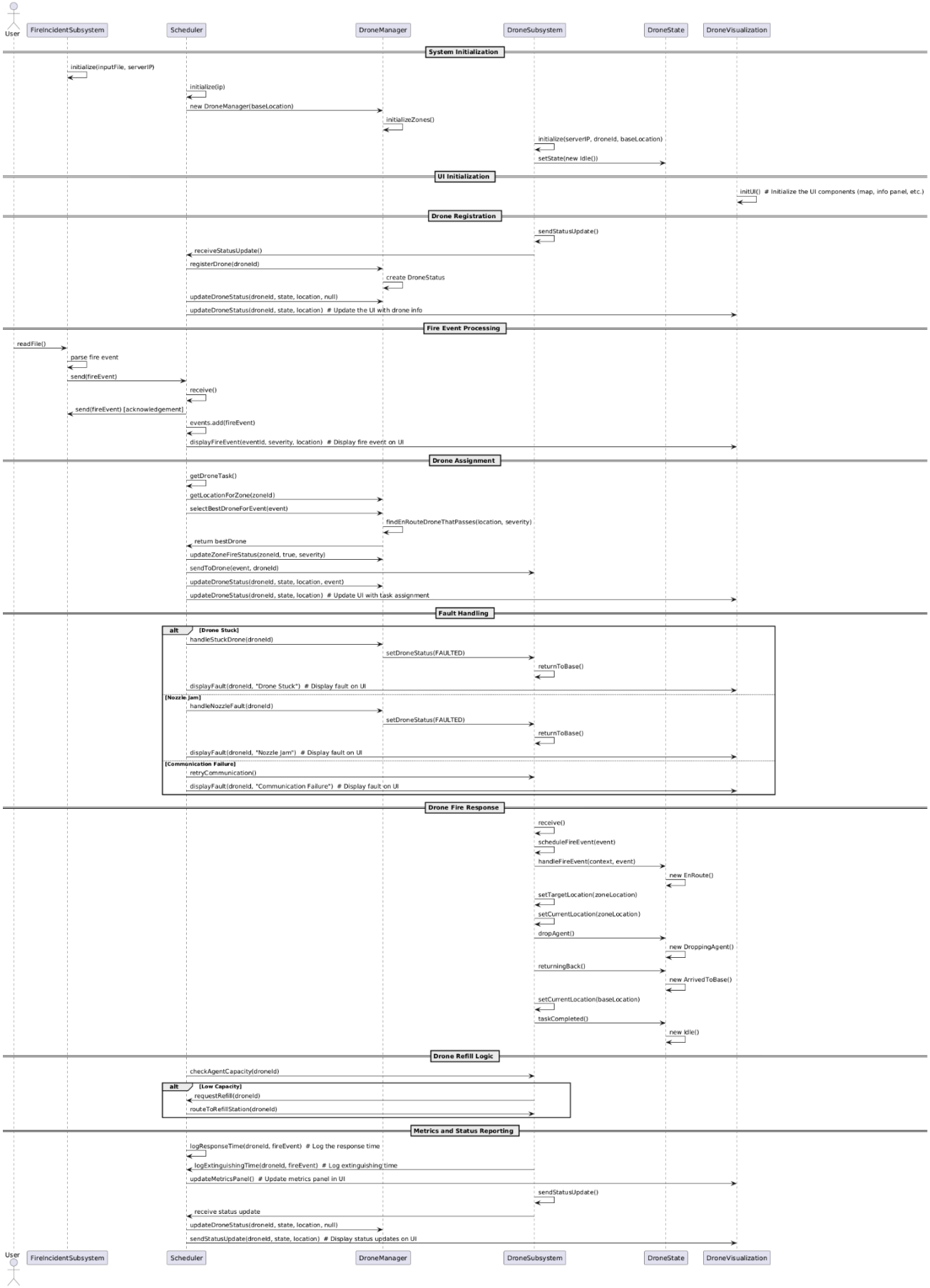
2.1 UML Class Diagram for the three components



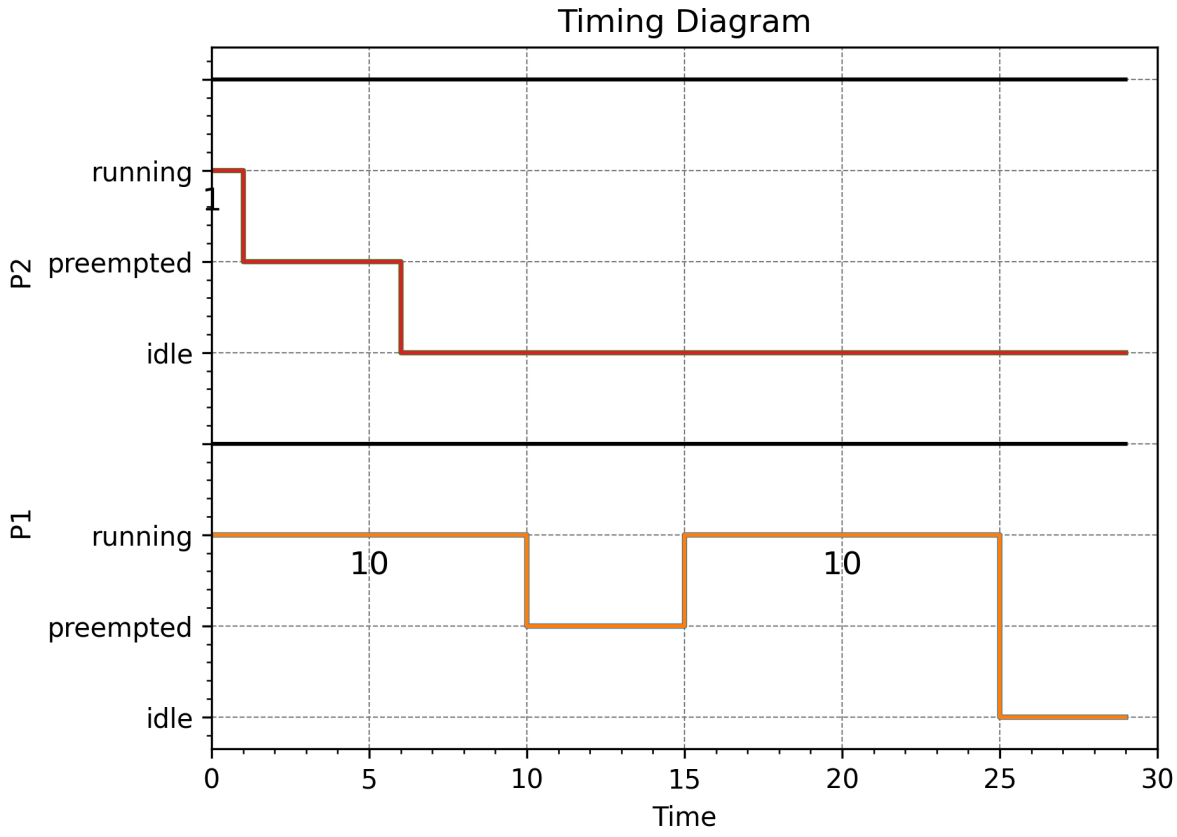
2.2 A State Machine Diagram for the drones



2.3 Sequence Diagrams showing all the error scenarios


User
FireIncidentSubsystem
Scheduler
DroneManager
DroneSubsystem
DroneState
DroneVisualization
System Initialization
initialize(inputFile, serverIP)
initialize(ip)
new DroneManager(baseLocation)
initializeZones()
initialize(serverIP, droneId, baseLocation)
setState(new Idle())
UI Initialization
initUI() # Initialize the UI components (map, info panel, etc.)
Drone Registration
sendStatusUpdate()
receiveStatusUpdate()
registerDrone(droneId)
create DroneStatus
updateDroneStatus(droneId, state, location, null)
updateDroneStatus(droneId, state, location) # Update the UI with drone info
Fire Event Processing
readFile()
parse fire event
send(fireEvent)
receive()
send(fireEvent) [acknowledgement]
events.add(fireEvent)
displayFireEvent(eventId, severity, location) # Display fire event on UI
Drone Assignment
getDroneTask()
getLocationForZone(zoneId)
selectBestDroneForEvent(event)
findEnRouteDroneThatPasses(location, severity)
return bestDrone
updateZoneFireStatus(zoneId, true, severity)
sendToDrone(event, droneId)
updateDroneStatus(droneId, state, location, event)
updateDroneStatus(droneId, state, location) # Update UI with task assignment
Fault Handling
alt
[Drone Stuck]
handleStuckDrone(droneId)
setDroneStatus(FAULTED)
returnToBase()
displayFault(droneId, "Drone Stuck") # Display fault on UI
[Nozzle Jam]
handleNozzleFault(droneId)
setDroneStatus(FAULTED)
returnToBase()
displayFault(droneId, "Nozzle Jam") # Display fault on UI
[Communication Failure]
retryCommunication()
displayFault(droneId, "Communication Failure") # Display fault on UI
Drone Fire Response
receive()
scheduleFireEvent(event)
handleFireEvent(context, event)
new EnRoute()
setTargetLocation(zoneLocation)
setCurrentLocation(zoneLocation)
dropAgent()
new DroppingAgent()
returningBack()
new ArrivedToBase()
setCurrentLocation(baseLocation)
taskCompleted()
new Idle()
Drone Refill Logic
checkAgentCapacity(droneId)
alt
[Low Capacity]
requestRefill(droneId)
routeToRefillStation(droneId)
Metrics and Status Reporting
logResponseTime(droneId, fireEvent) # Log the response time
logExtinguishingTime(droneId, fireEvent) # Log extinguishing time
updateMetricsPanel() # Update metrics panel in UI
sendStatusUpdate()
receive status update
updateDroneStatus(droneId, state, location, null)
sendStatusUpdate(droneId, state, location) # Display status updates on UI
User
FireIncidentSubsystem
Scheduler
DroneManager
DroneSubsystem
DroneState
DroneVisualization

2.4 Timing diagrams for the Scheduler



3.0 Set Up and Test Instructions

3.1 Set Up

Ensure that all installation requirements are met, to avoid any errors during compilation. You must be using the IntelliJ IDEA to run the application. The project must first be downloaded and opened into IntelliJ IDEA with a properly configured JDK.

In order to run the program, first navigate to the Scheduler.java file and click run. Next, to start multiple drone instances, run the DroneSubsystem.java file and wait until all drones are initialized. Finally, run the FireIncidentSubsystem.java file.

3.2 Test Instructions

The following steps outline how to verify the core functionality of the system as implemented in iteration 5. These tests should be run after the system has been properly set up as described in section 3.1.

Test 1: fire event transmission and scheduling
Steps:

1. Run the system in the following order: Scheduler, one or more DroneSubsystem instances (each with a unique drone ID & initial coordinates), then FireIncidentSubsystem.
2. Observe the FireIncidentSubsystem output in the console.

The expected result:
Each fire event from fire_events.txt is sent to the Scheduler via UDP. The Scheduler logs the reception of each fire event and adds it to the event queue. After receiving a response, the FireIncidentSubsystem prints an acknowledgement message confirming that the event has been successfully received and queued by the Scheduler.

Test 2: multi-drone assignment based on severity
Steps:

1. Run the system in the following order: Scheduler, one or more DroneSubsystem instances (each with a unique drone ID & initial coordinates), then FireIncidentSubsystem to begin processing fire events.
2. Monitor the console output of the Scheduler and all the running DroneSubsystem instances.

The expected result:
When a High severity fire event is triggered the Scheduler assigns multiple drones to the same fire zone. The number of drones assigned is dynamically determined by the Scheduler based on the severity and availability of drones.Then drones transition through the following states:
IDLE → ENROUTE → DROPPING_AGENT → ARRIVED_TO_BASE → IDLE
DroneVisualization shows the drones moving to the fire zone, performing the agent drop and returning to base.

Test 3: fire event completion and task reassignment
Steps:

1. Run the system in the following order: Scheduler, one or more DroneSubsystem instances (each with a unique drone ID & initial coordinates), then FireIncidentSubsystem .
2. Monitor the fire event handling by checking the task completion in the Scheduler console logs.

The expected result:
After enough agent drops are completed (based on fire's severity) the fire is marked as extinguished. The Scheduler logs the fire as resolved and drones are returned to base. Drones are marked IDLE and become available for new tasks once their current mission is completed.

Test 4: real time visualization of drones and fire zones
Steps:

1. Run the system in the following order: Scheduler, one or more DroneSubsystem instances (each with a unique drone ID & initial coordinates), then FireIncidentSubsystem
2. Observe the DroneVisualization window for real time updates.

The expected result:
Drone movements should be displayed as they move toward fire zones. Fire zones should be shown with color coding indicating fire severity: Red for High, Yellow for Moderate, Green for Low. Drone states are also color coded and include: Green for IDLE, Yellow for ENROUTE, Blue for DROPPING_AGENT, Red for FAULT. Drone tasks and errors should be reflected in real time within the GUI.

4.0 Measurement Results

In our testing we used **12 zones**, with a size of **300 x 350 metres**. Our system utilized **10 drones**, with specifications taken from the data gathered in iteration 0. Each drone had the following characteristics:

- Maximum speed: 56 km/h
- Water capacity: 10 Litres
- Flow rate: 500 L/s

This configuration allows each drone to release its entire payload in under one second, enabling continuous movement and ensuring the system remains fast and efficient.

Test Scenario:
The following fire events were used during testing:

14:03:33 12 FIRE_DETECTED High
14:15:00 3 FIRE_DETECTED Low
14:14:03 2 FIRE_DETECTED Moderate
14:03:06 1 FIRE_DETECTED High
14:03:09 8 FIRE_DETECTED High
14:03:15 5 FIRE_DETECTED Low
14:03:18 7 FIRE_DETECTED High
14:03:21 11 FIRE_DETECTED Moderate
14:03:24 10 FIRE_DETECTED Low
14:03:27 6 FIRE_DETECTED High
14:03:30 4 FIRE_DETECTED Moderate

Results:

- Total Runtime (Real time): 12 minutes, 18 seconds
- Average Response Time: 3 minutes, 32 seconds
- Average Time to Extinguish a Fire: 6 minutes, 39 seconds

5.0 Reflection

Overall, we are happy with the design and implementation of our Firefighting Drone System. From the beginning, the system was designed with scalability in mind, using multiple programs to allow the Scheduler to send and receive messages to both the drones and the Fire Incident System in parallel. The user can simulate the system entirely on one device or distribute it across multiple devices and IP addresses for more realistic use cases. Additionally, the design is flexible. Users can configure the number and size of zones, as well as the number of drones in the simulation.

We are especially pleased with our use of priority queues, which allow the system to service higher-severity fire events first. Zones can dynamically update their fire severity if a fire grows or a new one is detected in the same location. We also implemented scheduled checks to ensure the system is self-correcting in case anything is missed or delayed during processing.

That said, our system does have some limitations. A major challenge we faced was dealing with concurrency issues, particularly after introducing the user interface. During early development, all testing was done via the terminal, and no concurrency issues were apparent. However, once we integrated the UI, we began encountering several race conditions and timing-related bugs that were hard to detect in a terminal-only environment. This required substantial refactoring and debugging to ensure the UI interacted correctly with the rest of the system, and was displaying information correctly.

If we were to build the system again, we would likely create a simple graphical interface early in the project. Doing so would have helped us detect concurrency problems sooner and allowed us to progressively enhance the interface in later stages of development. We also would have liked to begin implementing more complex aspects of the design earlier in the development cycle. Since some iterations introduced more requirements than others, there were weeks where we had to rapidly integrate several complex features at once, while other weeks had simpler, more incremental changes. Some of the concurrency issues we encountered may have been better managed if we had started with a stronger emphasis on future-proofing our initial architecture to better accommodate upcoming features.