CS2006: Operating Systems

Module III Report - Final Project

Detailed Report on Design Choices for AirControlX System

Aabia Ali 23I-0704

Insharah Irfan 23I-0615

CS-D

Table of Contents

Introduction

The AirControlX system is a comprehensive air traffic control simulation designed to manage flight operations, monitor speed violations, issue Aviation Notices (AVNs), and process payments. The system is implemented across multiple C++ modules: atc_controller.cpp, avn_generator.cpp, airline_portal.cpp, and stripe_pay.cpp. This report outlines the key design choices made in the development of this system, explaining the rationale behind architectural decisions, data structures, inter-process communication, user interface, and synchronization mechanisms.

1. System Architecture

Modular Design

The system is divided into four distinct modules, each responsible for a specific functionality:

- ATC Controller (atc_controller.cpp): Manages flight assignment, scheduling, runway operations, and real-time monitoring of flight parameters (e.g., speed, fuel). It is the main process in our system, the control tower and central unit of this project.
- AVN Generator (avn_generator.cpp): Generates AVNs for speed violations and logs them to a file.
- Airline Portal (airline_portal.cpp): Allows airlines to query AVN details and receive payment status updates.
- StripePay (stripe_pay.cpp): Simulates a payment processing system for AVN fines.

Each module operates independently except for the AVN Generator, which has to generate AVN based on violations flagged in the ATC Controller. The separation of concerns ensures that changes in one module (e.g., payment processing logic) do not affect others (e.g., flight scheduling) and the system functions like a well-oiled machine.

Inter-Process Communication (IPC)

The system uses named pipes (FIFOs) for communication between modules:

- ATC to AVN Generator: A pipe sends violation data (flight ID, speed, etc.) to trigger AVN generation.
- AVN Generator to Airline Portal and StripePay: FIFOs (portal_fifo, stripe_fifo) transmit AVN details.
- StripePay to AVN Generator and Airline Portal: A FIFO (payment_fifo) sends payment confirmations.

Rationale: Named pipes were chosen for their simplicity and efficiency in a Unix-like environment. They provide a lightweight, file-based mechanism for asynchronous communication, suitable for the system's

need to pass structured messages (e.g., AVN details, payment confirmations). Unlike sockets, FIFOs do not require network setup, making them ideal for local inter-process communication. Non-blocking I/O was used to prevent processes from hanging when no data is available, ensuring responsiveness.

Process Forking

The AVN Generator runs as a separate child process, forked from the ATC Controller using fork() and execl().

Rationale: Forking allows the AVN Generator to operate independently, isolating its execution environment from the ATC Controller. This design prevents potential crashes in the AVN Generator from affecting the core ATC simulation. It also enables parallel processing, as the AVN Generator can handle violation data concurrently with the ATC's flight management tasks.

2. Data Structures

Flight and AVN Structures

- Aircraft Structure (atc_controller.cpp): Stores flight details such as ID, airline, type, phase, speed, fuel percentage, and AVN count.
- AVN Structure (avn_generator.cpp, airline_portal.cpp, stripe_pay.cpp): Contains AVN ID, flight ID, airline, aircraft type, speed details, issuance time, fine amount, payment status, and due date.
- Other structures like the SpeedRule or FlightData allowed for easy monitoring, input, and even graphics simulation.

Rationale: Structs were chosen for their simplicity and ability to group related attributes. The Aircraft struct encapsulates all flight-related data, making it easy to pass and update flight states. The AVN struct standardizes violation data across modules, ensuring consistency in how AVN information is stored and communicated.

Priority Queues for Runway Scheduling

The ATC Controller uses three std::priority_queue instances for runway assignments:

- Arrival Queue: For flights landing from North/South.
- Departure Queue: For flights departing to East/West.
- Cargo/Emergency Queue: For cargo or emergency flights.

Each queue uses a custom AircraftComparator to prioritize flights based on scheduled time and priority level.

Rationale: Priority queues efficiently manage flight scheduling by ensuring that flights with earlier scheduled times or higher priorities (e.g., emergencies) are processed first. The custom comparator allows

fine-grained control over scheduling logic, balancing temporal and priority-based constraints. Separate queues for different runway types prevent conflicts and streamline runway allocation.

Hash Map for AVN Storage

The AVN Generator uses an std::unordered_map to store AVNs, mapping flight IDs to AVN structs.

Rationale: The hash map provides O(1) average-case lookup time, enabling quick retrieval of AVN details when processing payment confirmations or logging. This is critical for real-time operations, as the system must handle frequent updates without performance bottlenecks.

Vector for History and Pending AVNs

- Airline Portal: Uses a std::vector to store AVN history loaded from AVNlog.txt and received via FIFO.
- StripePay: Uses a std::vector to store pending unpaid AVNs.

Rationale: Vectors were chosen for their dynamic sizing and sequential access capabilities. In the Airline Portal, the vector stores a growing history of AVNs, allowing easy iteration for display and search. In StripePay, the vector manages a small, frequently updated list of pending AVNs, where linear search is acceptable due to the typically low number of entries.

3. Flight Management and Simulation

Flight Phases and State Transitions

Flights progress through distinct phases (HOLDING, APPROACH, LANDING, TAXI, AT_GATE, TAKEOFF_ROLL, CLIMB, CRUISE), with transitions triggered by time-based conditions (e.g., 5 seconds in a phase) and runway availability.

Rationale: The phase-based model accurately simulates real-world air traffic control workflows, where flights follow a predictable sequence of states. Time-based transitions simplify the simulation while maintaining realism. Runway availability checks ensure that phase changes (e.g., from TAXI to AT_GATE) occur only when resources are free, preventing conflicts.

Runway Management

Three runways are defined:

- RWY-A: For arrivals (North/South).
- RWY-B: For departures (East/West).
- RWY-C: For cargo and emergency flights.

Each runway has a mutex, condition variable, and occupancy flag to manage access.

Rationale: Dedicated runways for different flight types reduce contention and reflect real-world airport operations, where runways are often specialized. Mutexes and condition variables ensure thread-safe access, preventing multiple flights from occupying the same runway simultaneously. The cargo/emergency runway prioritizes critical flights, enhancing safety.

Speed Monitoring and AVN Issuance

The monitorSpeed function checks flight speeds against phase-specific rules (e.g., 240-290 km/h for APPROACH). Violations trigger AVNs, sent to the AVN Generator via a pipe.

Rationale: Phase-specific speed rules mimic real-world aviation regulations, ensuring realistic violation detection. The pipe-based communication decouples speed monitoring from AVN generation, allowing the ATC Controller to focus on flight management while the AVN Generator handles violation processing.

Fuel and Fault Handling

Flights have fuel levels that decrease over time, with low fuel (<20%) triggering emergency status and reassignment to RWY-C. Ground faults are simulated with direction-specific probabilities, moving affected flights to AT_GATE.

Rationale: Fuel monitoring adds realism by simulating resource constraints, while emergency handling prioritizes safety by rerouting low-fuel flights. Faults introduce unpredictability, testing the system's ability to adapt to failures. Direction-specific fault probabilities reflect varying operational risks (e.g., higher for departures).

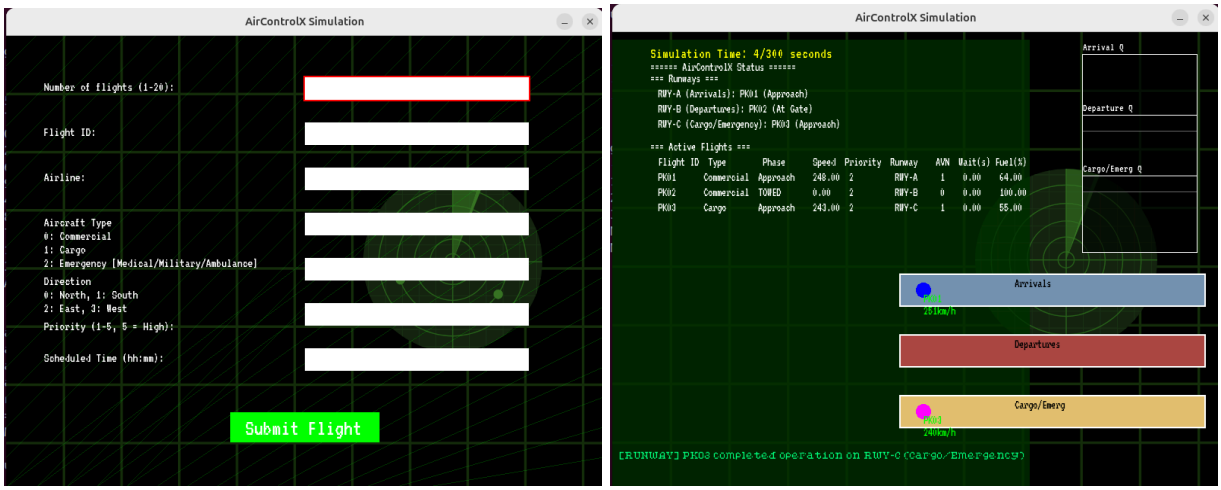
4. User Interface

SFML-Based Graphical Interface

The ATC Controller uses SFML for a graphical interface with two states:

- Input State: Users enter flight details (ID, airline, type, direction, priority, time) via text fields and a submit button.
- Simulation State: Displays runways, flight statuses, queues, and console logs in real-time.

Rationale: SFML was chosen for its lightweight, cross-platform capabilities, suitable for rendering a simple yet effective ATC interface. The input state simplifies data entry with validation, while the simulation state provides a clear visual representation of the airport's status, aiding user understanding. Color-coded runways and flights (e.g., blue for arrivals, red for emergencies) enhance readability.



Console-Based Interfaces

The Airline Portal and StripePay use console-based interfaces:

- Airline Portal: Prompts users for flight ID and issuance date, displaying matching AVNs and history.
- StripePay: Lists pending AVNs and allows payment selection by number.

Rationale: Console interfaces are simple to implement and sufficient for the modules' purposes. The Airline Portal's query-based design is intuitive for airline staff, while StripePay's numbered list simplifies payment processing. Both prioritize functionality over aesthetics, as their primary role is data interaction.

5. Synchronization and Concurrency

Multithreading in ATC Controller

The ATC Controller uses multiple threads:

- Runway Threads: One per runway, managing flight assignments.
- Radar Threads: One per flight, monitoring speed and fuel.
- Display Thread: Updates the graphical interface.

Rationale: Multithreading enables concurrent execution of independent tasks, improving performance. Runway threads ensure continuous operation of each runway, while radar threads allow real-time monitoring of all flights. The display thread keeps the UI responsive, updating every second.

Mutexes and Condition Variables

- Runway Mutexes: Protect runway access during assignment and phase transitions.
- Queue Mutexes: Synchronize access to priority queues.

- Display and Log Mutexes: Prevent race conditions in UI updates and logging.

Rationale: Mutexes ensure thread safety by preventing concurrent modifications to shared resources (e.g., runway occupancy, queue contents). Condition variables coordinate runway access, allowing flights to wait until a runway is free. This design minimizes deadlocks and ensures orderly resource allocation.

Atomic Variables

simulationTime and simulationRunning are std::atomic to ensure thread-safe updates.

Rationale: Atomic variables prevent data races in shared state variables accessed by multiple threads (e.g., simulation timer, termination flag). Their lock-free nature improves performance compared to mutex-based synchronization for simple updates.

6. Logging and Persistence

Log Files

- ATC Controller (log.txt): Records events like phase changes, AVN issuances, faults, and summaries.
- AVN Generator (AVNlog.txt): Stores AVN details in a human-readable format.

Rationale: Separate log files for ATC and AVN Generator provide clear audit trails for different system aspects. The ATC log captures operational events, aiding debugging and analysis, while the AVN log serves as a persistent record of violations, loaded by the Airline Portal at startup.

File-Based AVN History

The Airline Portal loads AVN history from AVNlog.txt and updates it with new AVNs received via FIFO.

Rationale: File-based persistence ensures that AVN data is retained across program runs, simulating a real-world database. The text-based format is simple to parse and aligns with the system's focus on functionality over complex storage solutions.

7. Error Handling and Robustness

Input Validation

- ATC Controller: Validates flight inputs (e.g., number of flights ≤ 20, valid time format) during the input state.
- AVN Generator: Checks for valid input fields (e.g., aircraft type, speed) before generating AVNs.
- StripePay: Handles invalid user choices gracefully, prompting for re-entry.

Rationale: Robust input validation prevents crashes and ensures data integrity. User-friendly error messages (e.g., "Invalid time format") guide users to correct inputs, improving usability.

Non-Blocking I/O

FIFOs are opened in non-blocking mode to prevent processes from hanging if a receiver is unavailable.

Rationale: Non-blocking I/O ensures system responsiveness, allowing processes to continue functioning even if communication channels are temporarily unavailable (e.g., StripePay not yet started). This is critical for a real-time simulation.

Fault Tolerance

The system handles faults (e.g., ground faults, low fuel) by adjusting flight states and reassigning resources (e.g., moving emergencies to RWY-C).

Rationale: Fault tolerance mimics real-world ATC systems, where unexpected events must be managed without disrupting operations. The design ensures that faults do not cascade into system-wide failures.

8. Performance Considerations

Lightweight Data Structures

Vectors and priority queues are used for their efficiency in the expected data sizes (e.g., ≤20 flights, small AVN lists).

Rationale: These structures balance performance and simplicity, avoiding the overhead of more complex containers (e.g., maps for small datasets). The small scale of the simulation makes linear searches in vectors acceptable for certain operations.

Sleep Intervals

Threads use sleep intervals (e.g., 100ms for runway controllers, 1s for radar monitors) to reduce CPU usage.

Rationale: Sleep intervals prevent busy waiting, conserving system resources while maintaining real-time responsiveness. The chosen intervals align with the simulation's time scale (e.g., 1-second updates for display)..

Conclusion

The AirControlX system's design prioritizes modularity, simplicity, and real-time performance while simulating key aspects of air traffic control. Named pipes enable efficient inter-process communication, while multithreading and synchronization mechanisms ensure robust flight management. The SFML-based GUI and console interfaces provide intuitive user interaction, and the logging system supports debugging and persistence. By balancing functionality with extensibility, the system serves as a solid foundation for further enhancements in air traffic control simulation.