Software Design Specification

David E Smith • Mike Kasper • Ryan Raub

2008/02/18 Rev 1.1

Table of Contents

Introduction

Synthetic Aperture Radar (SAR) is a modern radar imaging technique. The key constraint of this technology is that it can only be used from moving platforms while aimed at stationary targets. Fortunately, these instruments are commonly employed in this manner. However, issues arise during their development, as a moving platform is also required during testing. A common solution is to simulate movement by sending the SAR instrument inertial navigation data. Current simulations are hard-coded to devices that can send this data to the SAR instrument reliably at a constant desired rate. Unfortunately, the required rate and format of input vary among SAR instruments, and these devices only work for a single variation. This project aims to develop an application that provides a more flexible solution, where simulated aircraft movement can be sent to a wider range of SAR instruments.

Problem Statement

Testing of Synthetic Aperture Radar (SAR) is currently done manually using Inertial Navigation Systems (INS), which can currently play back pre-recorded flight data or simulate actual flight data to the SAR for testing. There are several different protocols for data transfer for each SAR interface; each requiring a specific INS that is not interchangeable or configurable. This makes the testing environment very rigid and time-consuming to set up and run. Recorded Flight data can only be played back over the same protocol upon which it was collected, which results in an even narrower selection of test cases available. In summary, the current testing environment is rigid and could benefit greatly from a more flexible solution.

Solution Statement

An INS simulator will feed the SAR simulated data calculated from a flight path defined by the user; which allows for an improvement to there testing environment for their SAR product. The INS simulator will have two main components; the User Interface and the Navigation Simulation System. The User Interface will allow the user to interactively manipulate a flight plan that the Navigation Simulator will run. The output from the Navigation Simulation can be in several forms, which are defined by a modular output. The final module will control the synchronization of data processing in the core module, which allows for a dynamic timing interface. This software application will take the place of several hardware INS systems currently in use, and provide a more convenient and flexible interface for testing.

Architecture Overview

There will be two main packages for this simulation application, and a TCP Socket connection between them. Each Package is a standalone application that can run independently of the other; however, they must establish a TCP network socket connection between one another to be useful.

The first package is the Simulator, which is where all of the processing is done. All bounds checking and error correction of data will be handled within this package. The Simulator package is responsible for reporting errors to the client and preventing data inconsistencies. The Simulator will have no direct user interface.

The second package is the Client, which is responsible for interfacing between the user via graphical interface and the Simulator via network socket. The Client will have a copy of the current Simulator state, and provide updates to the Simulator as the parameters are altered by the user. The goal with this package is to have the Client be as simple as possible, while relying on actual application functionality provided by the Simulator.

Module Descriptions

Core Module

The Core Module is the central hub for the entire simulation; it is responsible for communication between all other modules in the Simulator package. The Core is responsible for instantiating a sync module, physics module, and an output module for its operation. Having all data flowing through a central module limits the need for interconnection between modules, which increases the flexibility of the system. For example, if we wanted to change the rate at which to process the data; a new sync module would have to be instantiated in the core, and no other changes would be required.

Physics Module

The Physics Module is where all the actual simulation computation will be done. All that this module needs to know is the state of the plane, a list of waypoints, and what time the flight started. The list of waypoints can be altered at any time from the core module to reflect changes by the user, and the calculations will be based off of the current list of waypoints.

Output Module

The Output Module will have a very simple interface that takes in a state to output. Keeping this module independent greatly simplifies our design, and allows for many different types of output modules to be used interchangeably. The only communication back to the core module that the output will need is in the case of an error; and this will be handled by throwing exceptions back to the core.

Sync Module

The Sync Module is responsible for calling the core module with the correct timestamp at proper intervals. This module will run on its own thread, allowing for simple timing

issues to be managed. The module will aim for approximate real-time calling with time tags, in an effort to create a more accurate simulation. Potentially, this module could be replaced with one that uses an external timing source.

Server Module

The Server Module is responsible for managing connection(s) between the client and the core module. This provides abstraction of the client-server interaction from the basic functionality of the core, thus allowing other server modules to be used with no change to the core classes. Also, this allows client-server interactions to be handled by an independent thread, isolating network problems from affecting the simulation. Another useful service that the server module would provide is a list of possible sync and output modules to the client. This would allow for seamless management of modules the core uses in the simulation.

Client Module

The Client Module is only concerned with sending and receiving messages to and from the server module. In addition to message handling, the client is also responsible for initiating the connection to the server. The client provides a nice interface between the GUI and the server; as well as a clean separation from the display and server communication.

GUI Module

The GUI provides the user with continuous feedback regarding the simulation. It is closely linked to the client module, as the client module is what links the GUI to the server. User input is directed to the server module through the client module, and feedback about the simulation status is directed through the client module to the GUI for display to the user. All user controls are presented via the GUI.

The Frame will run as the main class. It will extend the JFrame object on hold all other objects. Because this class will instantiate and hold references to the CoreClient object, this class will administrate all communication between the CoreClient and the components implementing the CoreListener and CoreController.

The CoreClient class managers the communication between the remote core server and the GUI client. It will use object input and output streams so that more complicated objects may be exchanged with the server. It will receive updates of the plans current state, as well as send commands, such as starting/stopping a flight or changing waypoints. All components will need to communicate via this object to manipulate the server side behavior.

Any Component object that generates its content based on the state of the simulation will implement the CoreListener interface. It will receive the PlaneState object containing all of the most current simulation information, from which it may select specific data to use for its display and/or functionality.

The CoreController interface allows for commands to be passed to the server, via the CoreClient object. Each component that implements this interface is passed the CoreClient object reference so that it may employ its methods.

The GUI Component objects may implement one, both, or neither of these two interfaces, depending on its functionality. Implementing these interfaces allows for all of them to be managed easily by both the Frame and CoreClient objects.

Data Types

There will be seven unique data classes both packages will need to use in order to simplify communication and data storage. Each of the classes in this package is self-explanatory, from the UML diagram. Their use provides a uniform and standardized format amongst the classes, which results a cleaner code base to manage.

Client-Server Communication

Java object I/O streams will be employed for communication between client and server. These will be acquired from each connecting client socket. Because we are implementing both the client and server sides in Java, that this will be a simpler and more flexible solution. Method calls (from both the client and server) will take the following form:

method_name, [param1], [param2]…

where each element is a distinct Serializable object. The method_name will be a String of the exact name of the method on the opposing side. Because of this convection, no methods will be overloaded. Java reflection will provide the means to determine the actual method to invoke. This will also allow for lists of supported methods to be easily extended. For each call to the server, a response/confirmation of the return type, a confirmation message, or an Exception if the called failed, will be sent back to the client.

Module Implementation Specifications

Core Module

- Constructed with references to a sync, physics, output, and server class.
- Acts as a central hub for all module communication

public void update_state(State)
Action: Calls Output.write_state(State), and Server.update_state(State)

public void calc_state(Time)
Action: Calls Physics.calc_state(Time) and then calls update_state(State) with the returned State object

public void set(Sync)
Action: Sets the Sync module reference

public void set(Output)

Action: Sets the Output module reference

public void update_waypoints(ArrayList<<WayPoint>>)
Action: Calls Physics.update_waypoints(ArrayList<<WayPoint>>)

public void start()
Action: Calls start() on both the sync and output modules, as well as calls Physics.set_start(Time) with the current system time

public void stop()
Action: Calls stop() on both the sync and output modules

Physics Module

- Constructed with a reference to the core module, for call back functionality
- Stores only an ArrayList of waypoints, and a starting time

public State calc_state(Time)
Action: Based off of the difference between the start_time and the Time parameter a current state for is calculated and returned based off of the list of waypoints

public void update_waypoints(ArrayList<<WayPoint>>)
Action: Changes the waypoint list reference to the new list from the parameter

public void set_start(Time)
Action: Sets the start_time class variable to the value of the parameter

Output Module

- Constructed with no parameters
- This class functions as a data formatter, as it gets passed in state data it formats it and outputs it in another format

public void write_state(State) throws IOException
Action: Writes the state parameter to an output, and throws an exception if there is an error
Requirements: Must be called after start() and before stop()

public void start()
Action: Initializes internal constructs to get ready for the write_state(State) method to be called.
Requirements: Must be called before write_state(State)

public void end()
Action: Finalizes the output, when no more writing is possible
Requirements: This method must be called after start()

Sync Module

- Constructed with a reference to the core module, for callback functionality
- Controls the rate and time tags used in its invocations

public void start()
Action: Starts the internal thread

public void stop()
Action: Stops the internal thread

protected void run()
Action: Calls Core.calc_state(Time) with the correct time stamp

Server Module

- Constructed with a reference to the core module and lists of available sync and output modules
- Manages connections between clients and the core module

public void start()
Action: Starts the server, listening for clients

public void stop()
Action: Stops the server, connections are dropped

public void process_message(Message)
Action: Translates the Message from the client into the correct method call on the core, and executes the call

public void update_state(State)
Action: Sends State object to all clients

public void accept_connect()
Action: Spawns a new server thread for a client

public void drop_connect()
Action: calls stop() on the thread currently dealing with the client and drops the connection

GUI Module

Frame

- Constructed with no parameters
- Initializes the CoreClient object and all components
- CoreListeners that created will be added to the CoreClient object's list of listeners
- CoreController created will be passed reference to the CoreClient

CoreClient

- Constructed with on parameters
- Manages an ArrayList of CoreListeners and sends updates to each upon receiving a new PlaneState object from the server

public void connect(String address, int port)
Parameters:
Takes the location of the host (CoreServer) as a String
Takes the port number the CoreServer is running at the specified address
Action:
Connects the client to the specified ServerSocket
Upon establishing a connection, an input and output stream will be made
Creates object streams from the servers input and output stream

public void disconnect()
Action: Closes an existing connection

public void start()
Action: Communicates to the server that the simulation should begin

public void stop()
Action: Communicates to the server that the simulation should be halted

public void reset()
Action: Communicates to the server that the simulation should be restarted

public void udateWaypoints(Waypoint[] waypoints)
Parameters: An array of all the waypoints to be used in the simulation
Action: Send a news list for the server to user for the simulation; these waypoints could be a small alteration to the current set or a completely new set

public void setSync(int id)
Parameters: The sync module id, used by the server
Action: Communicates to the server what sync module to use

public void setOuput(int id)
Parameters: The output module id, used by the server
Action: Communicates to the server what output module to use

public void addCoreListener(CoreListener listener)
Parameters: The new listener for simulation updates to add
Actions:
Adds the listener to an ArrayList of CoreListeners
This list will be used to send updates to all components that need it

CoreListener
- Implemented by components requiring plane state information

public void update(PlaneState state)
Parameters: The current update of the simulated plane state
Action: This method is to be implemented by each individual component where specific data may be used for display or functionality purposes

CoreController
- Implemented by components requiring control over the core simulation

public void setCoreClient(CoreClient client)
Parameters: The client connected to the core server
Action: This method gives components access to the client, allowing it to make calls to the server when needed

FlightInput
- Implemented by classes providing the ability to retrieve stored waypoints
- Parameters for specifying the file name/location will be passed to the implementing constructor. This allows for the implemented method to be the same, regardless of the type of input stream.

public Waypoint[] read()
Action: Returns an array of waypoints for use in the simulation

FlightOutput
- Implemented by classes providing the ability to retrieve stored waypoints
- Parameters for specifying the storage location will be passed to the implementing constructor. Allowing for the implemented method to be the same, regardless of the type of output stream.

public write(Waypoint[] waypoints)
Parameters: The list of flight waypoints
Action: Stores the list of waypoints for later retrieval

Implementation Plan

Toward the end of February, we will be finishing our architecture and GUI design. This will involve constructing UML diagrams and GUI sketches, to be exchanged and reviewed by our client. All team members will participate in the applications design, and once completed, implementation will begin.

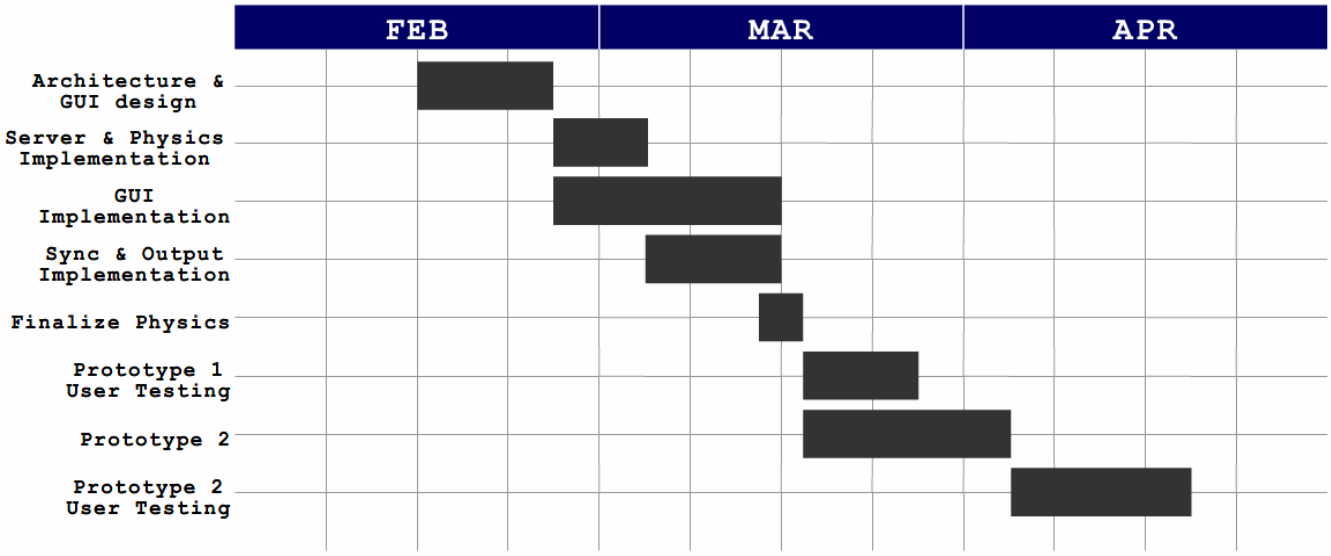
Throughout the end of February and early March, we will begin implementing our design. First, the core module and physics module, as well as the GUI, will be constructed in parallel. David and Ryan will work on the core as Mike will begin work on the GUI. The GUI is expected to take much longer then the core and physics modules, so David and Ryan will begin working on the sync and output modules as Mike finishes work on the GUI.

At this time, we expect to receive the full set of physics equations from the clients for calculating the plane status information. This will require a quick exchange (if needed) of our previously implemented equations. Once completed, modules will be fully integrated and the first prototype will be fully functional.

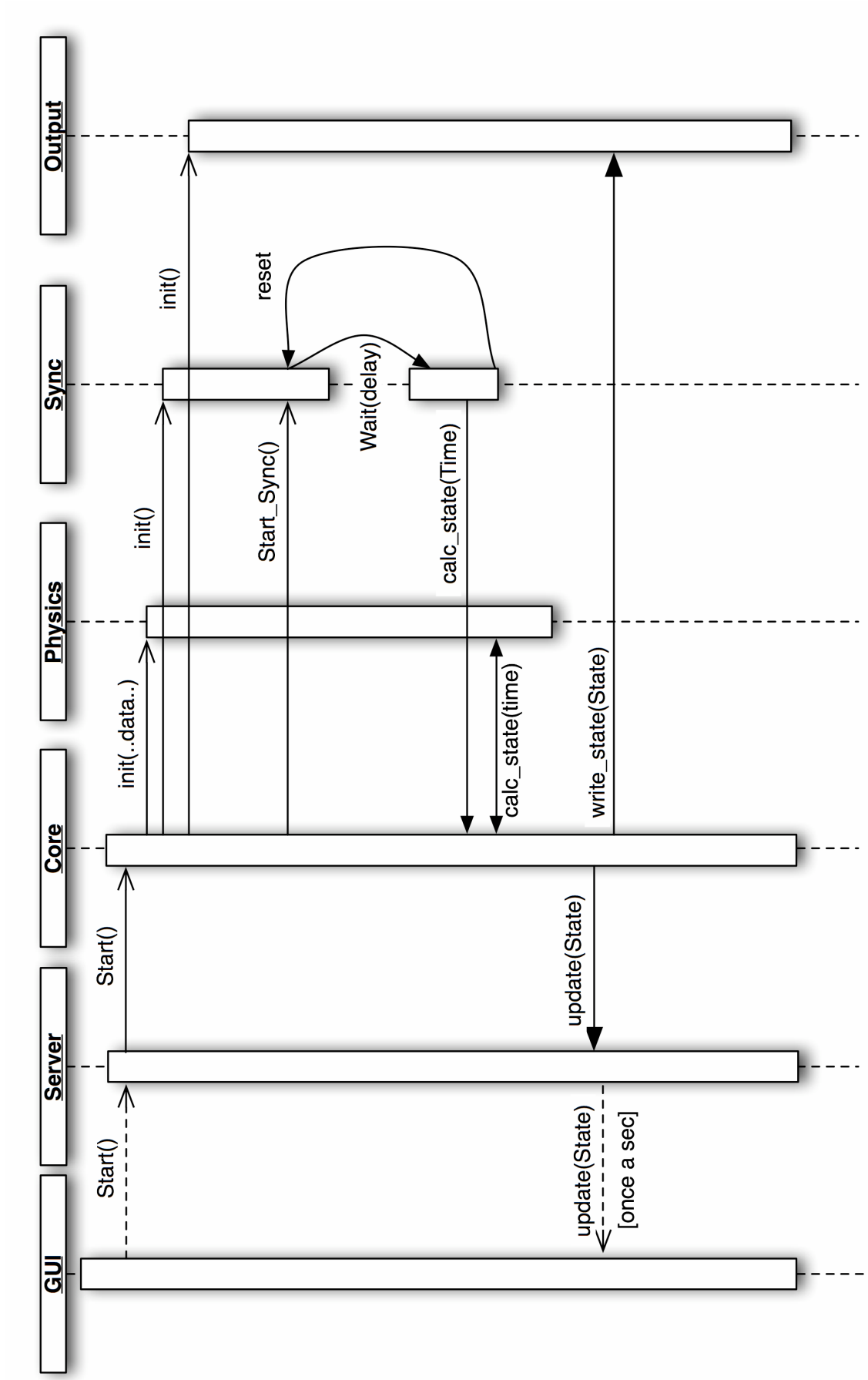
After completion of the first prototype, the majority of our focus will be on user testing. This first series of user tests will most likely not be done with the sponsor in person. This is due to our distance from the sponsor. We do not believe this to be very detrimental, because we expect these initial tests to find major problems and identify desired functionality. Throughout user testing, changes will be made to the existing prototype and additional functionality will be added, resulting in the second prototype.

Finally, we will conduct the second series of user tests will our new prototype. We do plan on traveling to meet with our clients to help find more obscure problems and final changes they wish to make with the GUI client.

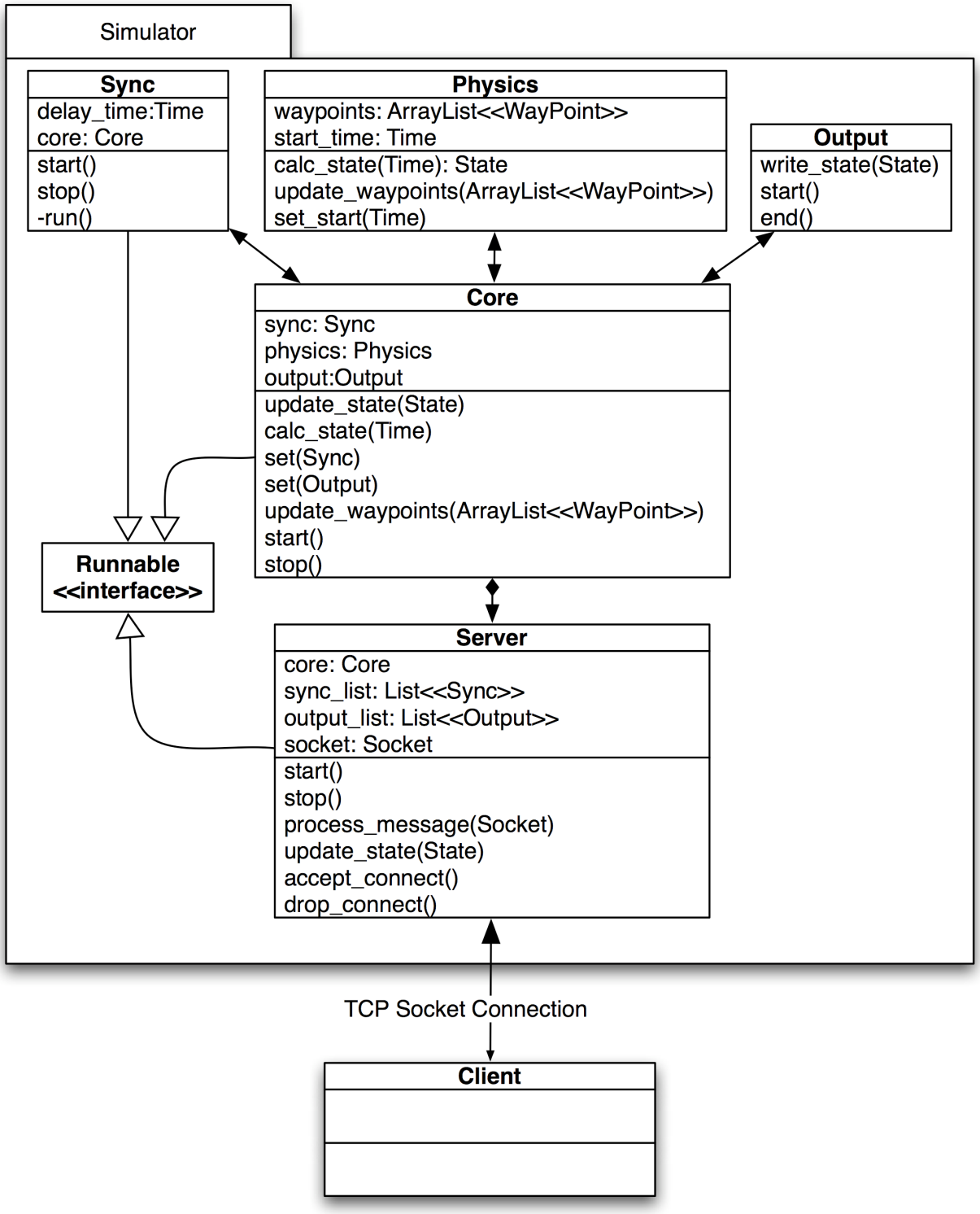
Timeline

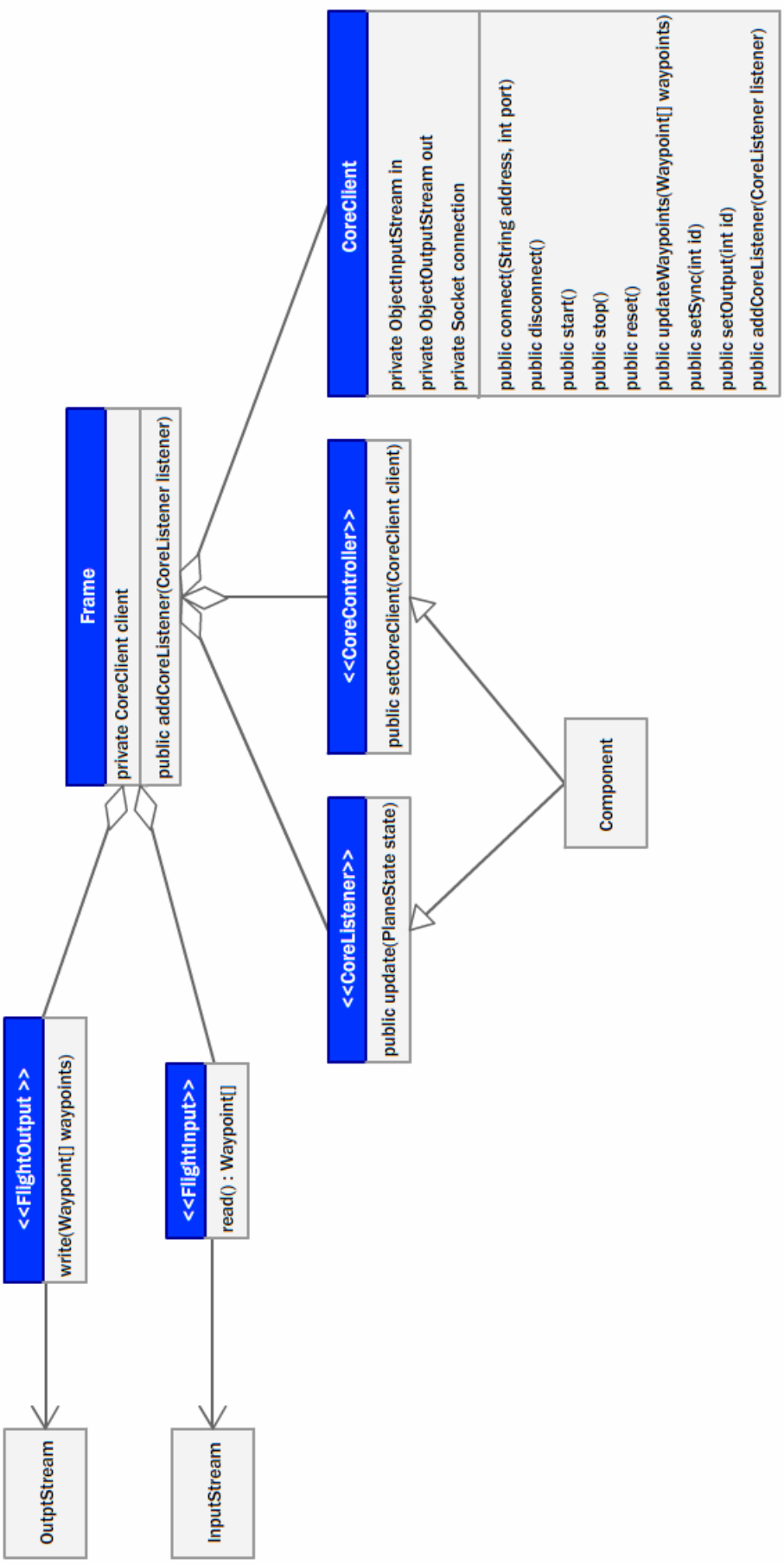


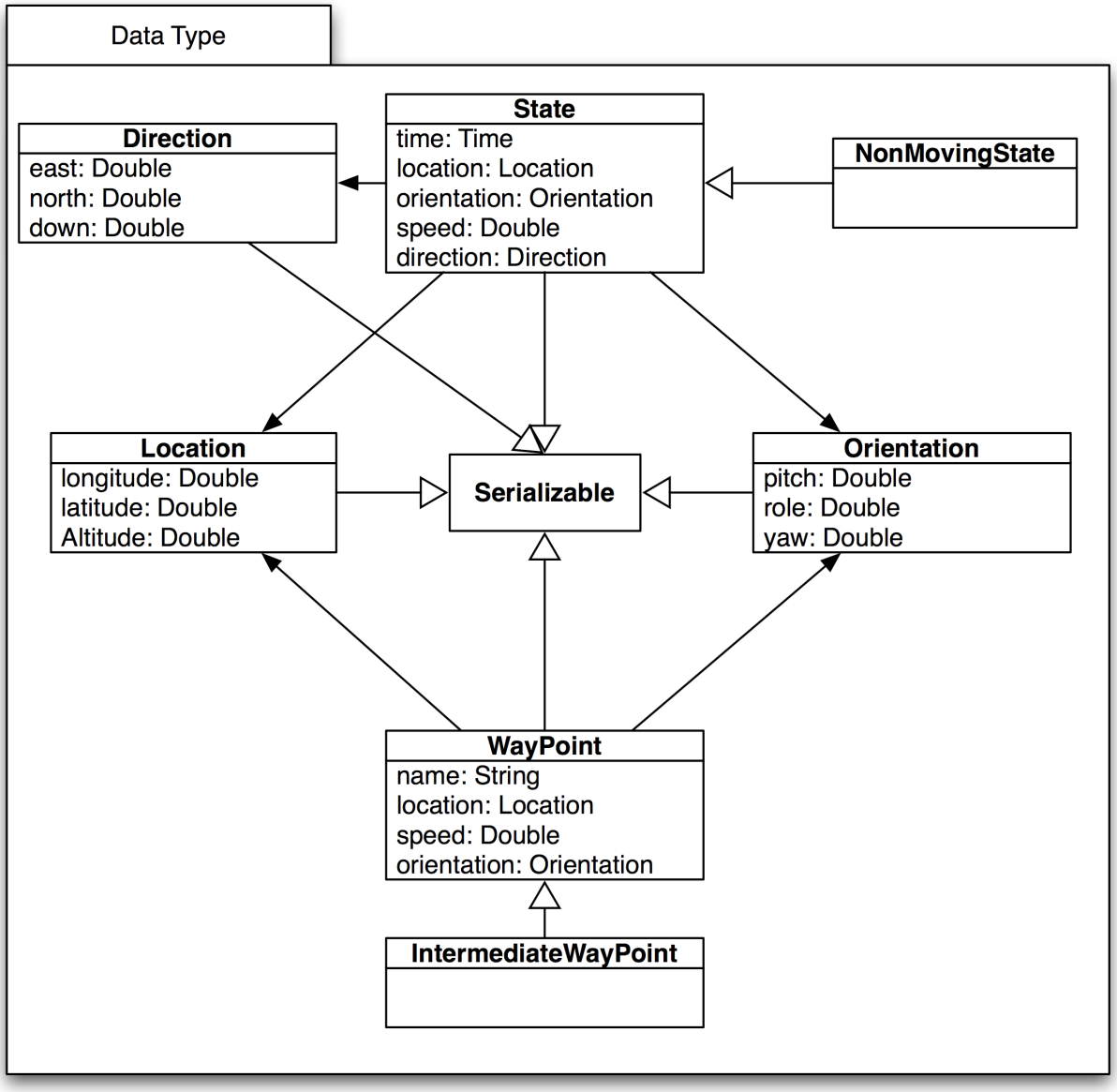
Appendix A: Sequence Diagram



Appendix B: UML Diagrams




CoreClient
private ObjectInputStream in
private ObjectOutputStream out
private Socket connection
public connect(String address, int port)
public disconnect()
public start()
public stop()
public reset()
public updateWaypoints(Waypoint[] waypoints)
public setSync(int id)
public setOutput(int id)
public addCoreListener(CoreListener listener)
Frame
private CoreClient client
public addCoreListener(CoreListener listener)
<<CoreController>>
public setCoreClient(CoreClient client)
<<CoreListener>>
public update(PlaneState state)
Component
<<FlightOutput >>
write(Waypoint[] waypoints)
<<FlightInput>>
read() : Waypoint[]
OutptStream
InputStream


Data Type
State
time: Time
location: Location
orientation: Orientation
speed: Double
direction: Direction
Direction
east: Double
north: Double
down: Double
NonMovingState
Location
longitude: Double
latitude: Double
Altitude: Double
Serializable
Orientation
pitch: Double
role: Double
yaw: Double
WayPoint
name: String
location: Location
speed: Double
orientation: Orientation
IntermediateWayPoint