# Introduction

In terms of core framework architecture, the application can be subdivided into services layer, packet handling layer, and also the underlying peer representation layer. Within the services layer there are three key services that are running within separate threads – beacon service, data service, and geo device service. Packet handling layer is responsible for packing, unpacking, and also validating various types of packets such as common-header packet, beacon packet, and also data packets. Finally, the peer representation structure is an overarching system that encapsulates all the information pertaining to a single peer operating in a specific configuration. Below the core framework structure are various types of network related and byte manipulation utilities that allow communication with the MAC layer and low-level operations. For gathering and analyzing information pertaining to performance and efficiency of the KimonoNet protocol, the complete framework attaches to a statistics monitoring component built with the slave-master architecture in order to allow analysis of both simulation and production level testing.

KimonoNet implementation introduced several interesting concepts that harvested Java language’s advantages over other low-level native programming languages such as C or C++. Although with a cost of highly optimized code, the developers found the object oriented features as well as Java’s advanced supported for threaded applications very useful. The below sections will delve deeper into specific services, architectures, and also Peer-to-Peer application implementation innovations.

# Services Layer

Each service is defined by a Service interface that mandates various thread management methods such functionality for gracefully starting and stopping threads. Given that most likely each service is going to be run within a separate thread, management of shared memory access is controlled by synchronized or blocking structures. For example, the routing-table-1 and routing-table-2, which respectively store the 1-hop and 2-hop neighbor information, are stored within blocking hash maps that allow only a single thread to access the object at a time. When possible, the use of intrinsic locks and synchronization utilities provided by Java native library were used over other common thread management techniques. Furthermore, in sake of optimization, static or immutable structure were marked by the final keyword in order to facilitate caching within CPU-cores.

# Beacon Service

Beacon service is primarily responsible for sending beacon packets, handling received beacon packets, and also updating the routing-table-1 and routing-table-2. Although initially implemented with two threads handling outgoing and incoming packets, the final implementation uses a blocking packet read method with a specific timeout after which a beacon packet is sent out. The value of the timeout, by default set to 5000 milliseconds, is a configuration variable within the peer’s environment. Understanding potential contention at the MAC layer transmission, the timeout value is offset by a random additive value approximately equal to the 20% of the set timeout value. This as demonstrated by the simulation has proved to substantially decrease control overhead.

# Data Service

Data services utilize routing-table-1 and routing-table-2 to route received packets to their appropriate destinations. Unlike the beacon service, receiving and sending streams were subdivided into threads as

# Geo Device Service

Geo device service implements a polling architecture that with a certain frequency polls a GeoDevice for the current GPS location and velocity. Since there are various ways of fetching the devices GPS location, GeoDevice was specified as an interface with two underlying implementation: DefaultGeoDevice that represents a stationary node and RandomWaypointGeoDevice that generates GPS locations based on the random waypoint model. Future production level implementations will actually fetch the GPS location from the native libraries that support GPS device drivers and update the peer’s location accordingly.

# Peer Representation

Individual peers, distinguished by a unique MAC-48 address, are associated with a GPS location, velocity, and an optional human-friendly name. Furthermore, each peer is represented by a peer agent that is responsible for managing shared-access memory, environment and configuration variables, and also for associating services layer with a specific peer. Peer agents are also attached to a StatMonitor that allows various services to report sent, received, and dropped packets for further protocol analysis.

# Environment Configuration

Given that each peer may run within various types of devices and configuration, agents store an PeerEnvironment structure that is a flexible and extendible vault for specifying peer related parameters. Current implementation includes parameters for maximum transmission range, beacon service timeout, and beacon service timeout additive ratio. The purpose of adding an environment configuration vault was for future extensibility and also for easing simulation.

# Network Structures and Utilities

The current communication channel used by the peers is defined by a Connection interface that is implemented by UDPConnection and UDPMulticastConnection. Although within production mode UDPConnection is mandated, for allowing multiple nodes to attach to the same port number and communicate to each other, within simulation environment the UDPMulticastConnection is utilized. The Connection interface allows connecting or disconnected from the network, and reading or writing bytes. Even though UDP connection does include an extensive connection setup, with the connection method initialization of resources used for the connection are processed.

To allow an easy switch between different modes of communication and also for configuring port and address information, the network architecture includes an object PortConfiguration and also PortConfigurationProvider interface that is implemented by ProductionPortConfigurationProvider and also SimulationPortConfigurationProvider. Each port configuration specifies a single port number for beacon service, two port numbers for data sending and data receiving components of the data service, and also network interface IP address used for initializing communication channels.

Currently, the communication channel simulation uses IP layer broadcast packets to send out information. The receiving side, after accepting the packet attempts a magic byte flag check and also CRC32 check for packet content validity. If either of these checks fails, the packet is discarded and no further processing takes place.

# Packet Handling and Parcel Architecture

The conventional implementation of packet byte stream handling is usually done with complicated pointer architecture within the C programming languages with various types of memory management techniques. The result is usually highly coupled or complicated code structure that is almost impossible to manage or extend. To address these issues, KimonoNet implementers sought an object oriented approach that would decrease redundant code, enhance exception handling, and also allow much more room for flexibility. The final design, largely inspired by the packet handling architecture within the Android operating system, presented a Parcel object that acted as a stacked byte buffer with LIFO access to all primitive data types in Java and also arbitrary access for allowing easy CRC commutations. For example, to construct a peer packet, or a byte array representation of a peer the following code would be utilized: Parcel.combineParcelables(address, location, velocity); Since address, location, and velocity all implemented the Parcelable interface, they know how to create a parcel representation and hence, all that peer has to do is to combine these parcels. Once the parcel is created, which is represented as a byte buffer natively defined by Java, it is easy to output a byte array to be sent over the network. This workflow is also applicable for parsing byte arrays received from the network socket back into actual objects. Adding primitive data types to parcels is also very simple: the add(…) method has be overloaded wither various types of primitive data types supported in Java such as floats, doubles, int, bytes, chars. To add a field to the parcel, the developer needs only to call the add method on the parcel and the stack structured parcel will position the bytes accordingly. Comparing this type of architecture with the C-style byte buffer processing, the developer can concentrate on the business logic of the application as opposed to coding redundant and complicated packet packing and unpacking utilities.

# Conventions, Notations, and Documentation

The application development followed all conventions common to the Java programming language. As such, all local and instance variables were written as lower camel case as well as all constants were written in upper case with word separated by underscores. Modularized within 18 packages, all classes were accordingly positioned for ease of management and version control. In terms of nomenclature of class names and variable names, moderately long and descriptive names were preferred over abridged and cryptic names. Although this produced rather long statements, it allowed to the code to be much easier to read and understand.

# Logging and Debugging

Current implementation supports three types of logging: INFO, DEBUG, and ERROR. At production level, the only output enabled is the information output while during debugging both debug and error streams may be enabled.

# Testing

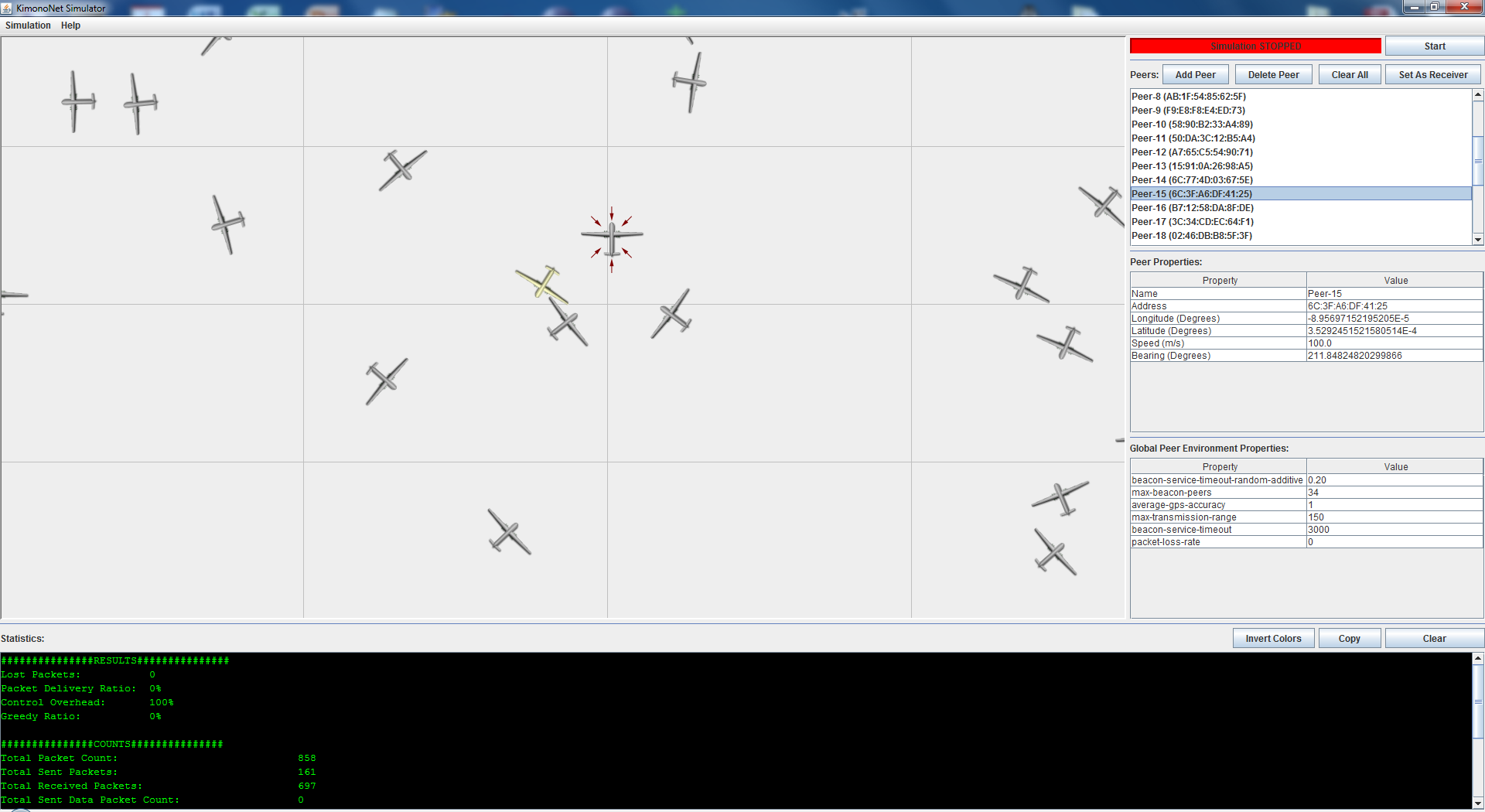
For unit testing, we decided to utilize the JUnit 4 testing framework. We believed that JUnit was the most natural choice because it was already built into Eclipse, the development environment we are using. JUnit enabled us to individually test each method of our project, and each of these tests may be run either separately or together as a test suite. The advantage of JUnit is that it does not require any modification of existing code, as all tests reside in their own Java package kimomonet.test. Furthermore, JUnit only adds one external dependency, which is required only when you want to run the tests. In other words, the JUnit library does not need to be included under normal operation. The JUnit tests are invoked by simply running our project as a JUnit test instead of the usual Java application. Test results are intuitively displayed in a panel inside Eclipse, and one could easily double-click on any error to directly jump to the problematic code for debugging.

Methods were rigorously tested to ensure that they do not throw any unexpected exceptions. Furthermore, they were double-checked against the specifications for consistency. For instance, we ensured that the packet structures follow our documentation such that all the fields are at the correct offsets and have the correct lengths. We even went as far as manually crafting packets at the byte level to use as a reference for comparison.

# Simulation Implementation

The simulation can be run in two modes: command line and graphical user interface (GUI). The command line mode is intended for developers and administrators.

In the GUI mode, the user can interactively experiment with the simulation. The GUI is very intuitive and easy to use and provides a visual representation of the nodes and their interactions for a better understanding of our protocol.



The user would normally start by specifying the map dimensions using the **Simulation 🡪 Edit Map Dimensions…** menu option. This step is optional. After that has been done, the user would click the **Add Peer** button to add nodes to the simulation. The nodes appear both in a list and graphically on a map. Selecting a node either from the list or by clicking on its graphical representation would display its properties in the table at the right. The user could easily edit the properties of the selected node, such as the node’s name, address, position, and speed. The user could even use the mouse to drag a node across the map. For convenience, the GUI automatically displays a tooltip below the mouse cursor showing the cursor’s position on the map in longitude and latitude. The user could also rotate the node (change its bearing) by spinning the mouse wheel.

Besides viewing and changing the properties of each individual node, the user could similarly see and modify global peer environment variables such as beacon service timeout.

Finally, before starting the simulation, the user would need to designate one of the nodes as a receiver using the **Set As Receiver** button, which would set the currently selected node as the destination sink. Also, if so desired, the user could set the *hostility factor*, which is the probability that node would explode (as in a hostile combat environment) using the **Simulation 🡪 Change Hostility Factor...** menu option.

To start the simulation, the user would click the **Start** button at the upper right corner. The nodes would begin to animate. Live statistics would be displayed at the bottom of the screen. During the simulation, the user could still select different nodes to view their properties, which would also be updated in real time. Nodes sending data packets and nodes that are exploding would be clearly indicated as well in their graphical representations.

On the technical side, the GUI was completely written in Java Swing. The WSIWYG editor provided by the Google WindowBuilder plugin for Eclipse was used to design the GUI.