**KimonoNet  
Enabling Efficient and Reliable Inter-UAV   
Fluid Data Communication Link**

*Document*  
Version 1.0.06  
March 21, 2012

*Authors*  
Bollens, Eric <ebollens@oit.ucla.edu>   
Hung, James <james.h@ucla.edu>  
Khalapyan, Zorayr <zkhalapyan@oit.ucla.edu> Norris, Wade <wade.norris@ucla.edu>

*KimonoNet addresses a peer-to-peer network research topic related to routing and transport control issues in sparse networks of highly mobile ad hoc peers such as unmanned aerial vehicles. Traditional distance-vector and link-state algorithms are not suited for such a topology given the high mobility and churn of its constituents, which causes routes to shift too quickly for global route propagation. Therefore, this approach leverages a coordinate-based routing system proposed by Greedy Perimeter Stateless Routing and adds two-hop neighbor awareness to minimize control packets and improve reliability in fluid, sparse network graphs.*

# 1. Introduction

## 1.1. Background Information

In recent years, the military has increased computing power in the field significantly; however, the increase in network coverage has not followed suit. One option to increase coverage is deployment of an ad hoc network whereby network-enabled equipment may form routes through a remote operational zone back to a secure base station.

Rather than providing the route via stationary relays, which make for easy targets, unmanned aerial vehicles and other mobile equipment with basic network capabilities may provide such a network in hostile environments. However, a network may manifest as a very sparse graph with limited routes to any endpoint. Further, given the mobility of nodes and possible calamities that might befall them, the network may also suffer high churn. Consequently, an implementation must adapt quickly to topological changes and provide reasonable service even when individual routes exist only briefly.

Beyond military application, this research topic may provide insight into broader issues in ad hoc networking. To address the dynamics of ad hoc networks, a number of protocols including Ad hoc On-Demand Distance Vector Routing (AODV) and Dynamic Source Routing (DSR) have taken the approach of on-demand routing, ascertaining a path at send-time rather than predetermining routes. However, these protocols suffer from disadvantages including long setup times, inconsistency and unreliability. Other ad hoc protocols such as the Optimized Link State Routing Protocol (OSLR) employ a table-driven approach reminiscent to traditional distance-vector protocols, but they struggle to adequately handle the rapidly changing nature of such a network with inordinate control traffic.

## 1.2. Objectives

Seeking to minimize both setup time and control traffic, this project foregoes traditional distance-vector and link-state protocols, as well as on-demand approaches. Instead, it uses Greedy Perimeter Stateless Routing, a coordinate-based approach first proposed by Karp and Kung [], and then extends it by propagating two-hop neighbor knowledge through control beacons. This strategy increases the interval between beacons, thus reducing control traffic; it also provides the foundation for other improvements based on this knowledge.

To these ends, this project provides a comprehensive description of the KimonoNet protocol including algorithms, message formats, structures, states and functions. While this document summarizes its key principles, a full description of the protocol may be found in *KimonoNet: Protocol Specification Document* [].

Further, this project provides a KimonoNet client prototype that runs under a JVM and provides both a production mode for field deployment and a test mode for simulations.

Finally, this project includes results from simulation that measure the viability and effectiveness of KimonoNet through metrics including packet delivery, routing overhead, average latency and path optimality.

## 1.3. Scope

### 1.3.1. Network Layers

The KimonoNet protocol is an ad hoc mesh overlay. Optimally, this protocol should be implemented directly over Layer 2. However, its flexibility also supports implementations on Layers 3 and 4 that leverage wireless broadcast or multicast.

For convenience, the KimonoNet prototype it built on UDP multicast. To support test mode, it also includes special provisions to simulate multiple nodes in a single environment.

### 1.3.2. Destination Locations

The KimonoNet protocol as described forthwith and in associated material supports mobile source nodes but requires a destination node of known location. In the event that a destination is not static, it assumes that another mechanism will be used to locate the destination. This design decision may present challenges for some communication, but it is purposeful in that it decouples the protocol from search mechanisms, for which significant research has already been devoted.

For simplicity, the prototype client does not provide a search algorithm to locate a mobile destination but instead assumes that the destination is fixed. A flooding search algorithm could be implemented in a future iteration so that a source can locate a mobile destination.

### 1.3.3. Data Payloads

This protocol provides an overlay for routing and transport across a fluid mesh network. As such, the protocol is ambivalent to the data it carries. It relies on the source to package this data and the destination to interpret this data. Consequently, it may transport application-layer data or lower-layer payloads tunneling across KimonoNet.

However, while KimonoNet supports tunneling, the implications of such are not considered forthwith. Users seeking to leverage duplex protocols such as TCP may encounter significant challenges. This is because KimonoNet assumes destinations of known location, and thus requires either a searching algorithm or that the mobile host initiate the communication. However, even when one of these conditions is met, by the time the data propagates to the other party, it may be stale and inaccurate. Large network diameters accentuate this problem.

### 1.3.4. Simplifying Assumptions

This protocol makes two additional simplifying assumptions: (1) it does not consider altitude, and (2) it assumes symmetric communication between any two nodes in the graph whereby, if one node can communicate with another, the reverse is true as well.

## 1.4. Use Cases

Military operations with highly fluid network topologies serve as the motivation for this project. In such scenarios, UAVs and other network-enabled devices move rapidly and may even disappear from the network due to failure or destruction. Consequently, the approach proposed must not require the continuing existence of any node in the network, but it may predict topology changes when possible to help reduce control traffic.

Further, due to the sparse nature of the network graph, the protocol must operate efficiently given limited routing options. The majority of nodes in the network will likely be out of range of any individual node, and thus it must leverage local knowledge and still make near-optimal forwarding decisions.

This routing protocol may extend beyond military operations. It has applicability in any network with high churn where the location and velocity of nodes are generally known. As such, this ad hoc communication protocol could easily support orbiting satellites and maritime expeditions. However, this algorithm will not cover scenarios where location is not available or where velocity changes unexpectedly; in such situations, other approaches are better suited.

## 1.5. Use Cases (1.4 is also Use Cases?)

Two primary constituents exist in this scenario:

1. Autonomous nodes that send new packets and route received packets.
2. End points that receive data packets routed across the network.

In the motivating scenario, unmanned aerial vehicles meet the former whereas command posts and external uplinks satisfy the latter.

This protocol assumes the initial communication is sent from an autonomous node to an end point of known location. This avoids the need to introduce a search algorithm. However, a full implementation may add such a mechanism to support communication between any pair of nodes in the network and to efficiently support duplex transport.

### 1.5.1. Autonomous Nodes

The primary constituents of the network, autonomous nodes have (1) awareness of their position and velocity, (2) a NIC that supports ad hoc communication, (3) a network layer that supports broadcast, multicast or promiscuous packet delivery, and (4) a running instance of the KimonoNet client. These nodes are regarded as autonomous because they make independent decisions about position and velocity without considering its implications on routing.

Autonomous nodes collect data, accomplish objectives and then seek further instructions. In order to transmit this data or receive further instructions, these nodes introduce data packets into the ad hoc network. These packets are addressed to the known location of an endpoint, and then forwarded through the network based on the routing algorithm.

Beyond introducing packets into the network, autonomous nodes must also receive data packets passed to them by other nodes and then forward them on based on the routing algorithm.

### 1.5.2. End Points

A command post or other external uplink to the Internet serves as the destination endpoint of a packet originating within the ad hoc network. Because this protocol does not consider a mechanism to determine endpoint location, it requires the originator to know the location of the end point and that this position does not change over time.

As with autonomous nodes, end points must also have (1) awareness of their position and velocity, (2) a NIC that supports ad hoc communication, (3) a network layer that supports broadcast, multicast or promiscuous packet delivery, and (4) a running instance of the KimonoNet client.

After receiving a packet from an autonomous node, an end point may use location and velocity information contained within the packet to reply. Because an autonomous node has variable position, all responses from an end point should have QoS classification of time-sensitive and be loss tolerant.

## 1.6. Constraints

### 1.6.1. High Churn

Due to the operating environment, the routing protocol must handle high churn effectively.

Each autonomous node has a velocity that affects its position over time; consequently, the neighborhood for a given node may change swiftly as nodes enter and recede from a network horizon. The routing algorithm handles this by considering two-hop neighbors and extrapolating position over time based on velocity to determine which nodes will be leaving range and which will be entering, thus reducing route maintenance traffic.

Further, given the motivating scenario, nodes may become unavailable unexpectedly due to failure or destruction. The algorithm presented must handle this independent influence on churn adequately, compensating for the fact that a route may become unavailable quickly and without warning. However, this protocol acknowledges that the unexpected disappearance of a node may cause the loss of a packet, but it should take strides to minimize the likelihood.

### 1.6.2. Mixed Horizons

In a sparse network, only a small set of nodes is within range of any individual node; further, adjacent neighborhoods likely share only a few nodes, if any. The routing algorithm must thus take this into account, as adjacent neighborhoods may not be able to communicate directly and a route must instead be established through other nodes to traverse the void.

This constraint is the reason that distance-vector and link-state management is non-optimal: by the time route information has propagated, routes may have changed substantially. This protocol thus operates under the assumption that nodes cannot self-route data through the entire network, and instead that nodes make decisions based solely on their local neighborhood.

## 1.7. Related Work

G. G. Finn proposed greedy route selection for coordinate-based routing in 1987, using a flood-based approach to resolve situations where greedy selection failed []. B. Karp and H. T. Kung reduced routing overhead by providing a method for detouring without flooding by way of graph planarization []. Their work, namely Greedy Perimeter Stateless Routing, serves as the basis for KimonoNet, which is then extended through two-hop awareness for velocity-based prediction of neighborhood changes.

# 2. PROTOCOL

At its heart, the KimonoNet protocol has two primary objectives:

* Nodes provide information that allows other nodes to recognize them as neighbors and predict how their position will change over time.
* Nodes forward data packets through the network to the intended final destination while making near optimal routing decisions at each hop based solely on local network knowledge.

In order to accomplish these goals, KimonoNet leverages work by Karp and Kung on GPSR and extends it such that a node updates its neighborhood proactively based on position and velocity information rather than only when it receives new updates from neighbors.

## 2.1. Beaconing

### 2.1. Beacon Initialization

When a peer first enters the network, it transmits a beacon packet. This beacon packet includes a unique identifier for the node, as well as a location, velocity and timestamp. All nodes within the node’s network horizon receive this beacon and update their routing tables accordingly.

### 2.2. Beacon Updates

At regular intervals, each node transmits a beacon packet similar to the beacon transmitted during initialization. Again, this packet includes a unique identifier for the node, as well as a position, velocity and timestamp. Beyond this information, the beacon includes information about any neighbors that the sender has identified within its network horizon.

Because these beacon packets contain information about both the node and its neighbors, a recipient can build two-hop awareness of both its neighbors and the neighbors of its neighbors, which it may then use to make routing decisions proactively even before it receives updates from these nodes.

### 2.3. Beacon Acknowledgement

When a node receives a beacon that does not include information about itself, it adds the information from the beacon to its own routing table and then responds with a beacon acknowledgement. This response, like recurring beacon updates, should include both the node’s information and information about all nodes within the node’s network horizon.

Beacon acknowledgements exist to reduce the amount of time it takes for a node to learn about its local topology. They provide a form of on-demand awareness whereby, if one node learns about a new node, then this node informs the new node of its existence as well. This procedure will not loop because, once a node learns of another node, it places it in its beacon, and therefore the node will not reply back with its own location again.

## 2.2. Routing

### 2.2.1. Greedy Forwarding

First proposed by G. G. Finn [], greedy forwarding selects the node closest to its final destination. In most topologies, this results in optimal routes because it stays ahead of other possible selections. KimonoNet implements greedy forwarding as its preferred routing mechanism.

When a node running KimonoNet receives a KimonoNet packet in greedy forwarding mode, it determines the closest peer among its neighbors to the destination as follows:

The KimonoNet protocol recommends the haversine formula for the method [*sinnott*]:

When node hardware cannot perform these computations effectively or where network horizons are mathematically insignificant in relation to the Earth’s radius , distances may be computed as if on two-dimensional plane. If ranges are large and precision is not needed to the meter, then the law of cosines provides an alternative. Meanwhile, if ranges are large and the hardware allows, distance calculations might be extended to support the ellipsoid nature of the Earth as opposed to a uniform radius sphere.

The key advantage of greedy routing is that it requires only local knowledge for forwarding decisions. Each router must maintain state only for its immediate neighbors, and beacon packets travel only to immediate neighbors, thus drastically minimizing state and control traffic. This makes greedy forwarding well suited for large, fluid networks because changes do not have to propagate through the whole network, and nodes do not have to keep track of the whole graph, nor any scale-dependent subgraph.

While greedy forwarding makes near optimal path selections under most scenarios, and it requires routers to maintain minimal state, it has two problems that must be acknowledged:

(1) In specially constructed topologies, it can be demonstrated that greedy selection is sub-optimal, potentially to a significant extreme. However, most real-world networks are not constructed as such, and this paper leaves discussion of this issue to related works.

(2) The current node may be a local maximum, namely closer to the destination than any of its neighbors. In this case, the node cannot select a neighbor further away than itself or else the packet may simply loop back. Retreating from a greedy selection also defeats the stay-ahead argument. Therefore, when such a maximum is encountered, it must provide a mechanism to route around the void.

### 2.2.2. Perimeter Forwarding

When first considering greedy forwarding for coordinate-based routing systems, G. G. Finn resolved the local maxima issue via flooding []. However, flooding creates inefficiencies in terms of network traffic, and minimizing these inefficiencies while flooding requires nodes to maintain state. Consequently, B. Karp and H. T. Kung described a perimeter-based alternative that detours around a void when a node is a local maximum [].

KimonoNet implements the perimeter detour approach presented by B. Karp and H. T. Kung. Under this mechanism, when a node is nearer to the destination than any of its neighbors, it switches to perimeter forwarding mode and selects the neighbor sequentially next counter-clockwise from the bearing to the destination. This process continues until the packet reaches a position closer to the destination than when perimeter forwarding began, at which point greedy forwarding may then resume again. If the node undergoes a cycle before this time, then it has encountered an unreachable destination.

This method ensures that a packet will not travel indefinitely because it will be dropped as soon as it reaches the start of the cycle again. However, in order to ensure that all possible nodes are visited, a cycle must not be reached prematurely. To prevent this, B. Karp and H. T. Kung present two methods: a no-crossing heuristic and graph planarization. The former maps perimeters and removes edges that cross a face it has already identified. While B. Karp and H. T. Kung found this worked in 99.5% of cases [], an edge may be removed that partitions the graph. The latter approach does not have this issue because it determines a planar graph and removes edges not in this graph.

When a node running KimonoNet receives a KimonoNet packet in perimeter forwarding mode, it determines the closest peer among its neighbors to the destination as follows:

The KimonoNet protocol recommends the following for the method [*williams*]:

As with , if node hardware cannot perform these computations or network horizons are mathematically insignificant, distances may be computed as if on a two-dimensional plane.

When a packet is in perimeter mode, the state machine for KimonoNet also tries to transition back to greedy mode if possible. Comparing the distance from the destination to both the current node and the node where the packet entered perimeter mode, a transition occurs if the current node is closer. This is because greedy forwarding is near optimal whereas perimeter forwarding is simply meant to detour around a void and get a packet closer to the destination.

The implementation of perimeter forwarding mode stores all necessary state within the packet. This ensures that nodes do not have to increase retained state as the number of packets in perimeter mode increases.

### 2.2.3. Predictive Neighbor Maintenance

Because beacons contain not only location, but also velocity and time, this protocol can perform simple calculations to predict changes to its neighborhood over time.

In the case of one-hop awareness, this reduces control packets and increases reliability and efficiency. This is because each node can compute how its neighbors’ positions will change over time. A node thus knows more quickly when a neighbor is no longer within its network horizon or when a neighbor has moved to a position that makes it more or less optimal for routing, all without the neighbor having to actually send another beacon.

Further, because a beacon includes information not just about a node but also its neighbors, it is thus possible to have two-hop awareness. This both reduces control packets and increases route efficiency. With a simple computation, a node can determine when a neighbor of a neighbor will enter into its network horizon and thus become a direct neighbor itself. This allows a node to more quickly take advantage of another node in its routing decisions, even before the two have explicitly exchanged beacons.

In order to efficiently implement this scheme, KimonoNet uses two routing tables. From received beacons, it maintains a two-hop routing table of all nodes learned from the beacons. Asynchronous to other tasks, it then uses this two-hop routing table to compute a one-hop routing table of neighbors directly within its network horizon [ § 3.2]. In the beacon packets that the node sends, it includes information about all nodes in its one-hop routing table.

Given the radius of the Earth , the speed of the node , the time since the neighbor information was generated , the bearing of the node , and the location of the node as defined in ROUTING-TABLE-2, the new location may be calculated as [*williams*]:

As with and , alternative, less computationally complex methods may be employed using a two-dimensional plane where necessary and possible. It should be recognized, however, that these calculations are propagated between neighboring nodes through beacons and therefore, while uniformity is not a requisite, a lack thereof may lead to differing ways that nodes regard neighbors.

### 2.2.4. Quality-of-Service

KimonoNet provides three grades of service:

(1) Control data is critical to network integrity such as beacons. It must be transmitted without delay and interpreted immediately upon reception, but it does not require forwarding.

(2) Communication data is time-sensitive data that is forwarded across the network. It is handled in priority over standard data, but in deference to control data.

(3) Standard data is handled when no control or communication data is outstanding to be sent.

The requirements document initially stipulated a fourth QoS garde for reliable data delivery []. However, this version of the protocol does not include support for reliability, as the author quickly found it evident that a good deal more research was needed to effectively implement reliable QoS. More discussion on reliable QoS may be found in § 5.2.3.

# 3. Implementation

## 3.1. Architecture

### 3.1.1. Overview

In terms of core framework architecture, the application can be subdivided into the services layer, the packet handling layer, and the underlying peer representation layer. Within the services layer there are three key services that run within separate threads – the beacon service, the data service, and the geo-device service. Furthermore, the packet handling layer is responsible for packing, unpacking, and validating various types of packets such as common-header packets, beacon packets, and 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 the 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.

Most notably, KimonoNet’s implementation introduces several interesting concepts that harvest Java’s advantages over other low-level native programming languages such as C or C++. Although at the cost of highly-optimized code, the developers found the object-oriented features as well as Java’s advanced support for threaded applications very useful. The sections below will delve deeper into specific services, architectures, and peer-to-peer application implementation innovations.

### 3.1.2. Conventions & Notation

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, while all constants were written in upper case with each word separated by an underscore. 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.

## 3.2. Service Layer

Each service is defined by a Service interface that mandates various thread management methods such as 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, routing-table-1 and routing-table-2, which stores the one-hop and two-hop neighbor information respectively, 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 the Java native library were used over other common thread management techniques. Furthermore, for the sake of optimization, static or immutable structures were marked by the final keyword in order to facilitate caching within CPU cores.

### 3.2.1. Beacon Service

The beacon service is primarily responsible for sending beacon packets, handling received beacon packets, and updating 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. In order to account for potential contention during transmission at the MAC layer, the timeout value is offset by a random additive value that is approximately equal to 20% of the set timeout value. As demonstrated by the simulation, this has proved to substantially decrease control overhead.

### 3.2.2. Data Service

The data service allows for the sending and receiving of Data Packets at individual node. The Data Service is initialized within a Peer Agent and has two threads running concurrently. One is for sending packets by popping them out of a priority queue and another is for receiving packets and handling them appropriately. The receiving service listens for inbound Data Packets and when one is received it either throws it away because it is not meant to be handled by this node, delivers it to this node if this is the final destination peer, or adds the Data Packet to the queue to be routed and handled by the sending thread. When packets are added to the queue the sending thread is notified to wake up and handle them appropriately.

The sending thread pops packets from the queue based on their Quality of Service. CONTROL packets are handled first then COMMUNICATION then finally REGULAR packets. If the packet queue is empty the thread sleeps so as to not waste CPU time and is alerted by the receiving thread when packets are added to the queue. When a packet is popped from the queue it is sent to the routing protocol to determine the next hop in the routing of this packet from the peer tables.

When a packet is sent to the routing protocol for determination of the next hop it initially updates the peer tables based on the velocities and the amount of time that has passed since their last update. If peers from peer table 2 have come into range of the node then they are transferred to peer table 1, meanwhile if peers in peer table 1 have gone out of range they are dropped from the peer table. Once the routing tables have been updated the packet is routed according to GPSR specification as explained in the protocol documentation.

The fundamental element for communication within the Data Service is the Data Packet. The Data Packet contains a set of information to be sent from any of the sources in the network and routed through peers to a fixed sink node. The packet contains the appropriate information used in routing the packet in addition to the common header shared with all Packets. In addition to the common header the Data Packet contains the address of the destination node, the location of the destination node, the address of the next hop in the routing of this Data Packet, an enumerable byte that designates the current forwarding mode of this packet (GREEDY or PERIMETER), a short integer specifying the length of the data payload attached to this packet, an enumerable byte designating the quality of service of this communication (CONTROL, COMMUNICATION, or REGULAR), an integer checksum for the set of header information. Additionally there are four Peer Location variables in the extended header that are set to null when the packet is in GREEDY forwarding mode and are set appropriately for routing when in PERIMETER forwarding mode.

The Data Packet can be set up from a parcel, a byte array, or using Peer Agent as the source a Peer as the Destination and a byte array for the payload. Additionally getters and setters for the header information is available for adjustments in the routing algorithm. The packet also allows for parceling of the packet which also sets up the CRC field appropriately as a CRC32 checksum for the common header, data header, and extended data header.

### 3.2.3. Geo-Device Service

The geo-device service implements a polling architecture that at a certain frequency polls a GeoDevice for the current GPS location and velocity. Since there are various ways of fetching the device’s GPS location, the GeoDevice was specified as an interface with two underlying implementations: a DefaultGeoDevice that represents a stationary node and a 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.

## 3.3. 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 the 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.

## 3.4. Environment Configuration

Given that each peer may run within various types of devices and configurations, agents store a PeerEnvironment structure that serves as a flexible and extendable 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.

## 3.5. Network Structures & Utilities

The current communication channel used by the peers is defined by a Connection interface that is implemented by UDPConnection and UDPMulticastConnection. Although for production mode, the use of UDPConnection is recommended, within the simulation environment, however, UDPMulticastConnection is utilized instead to allow multiple nodes to attach to the same port number and communicate with each other. The Connection interface allows for connecting to or disconnecting from the network, as well as for reading or writing bytes. Even though UDP connection does not include an extensive connection setup, the connect() method defined by the Connection interface is initialization of resources used for the connection.

To allow for easy switching between different modes of communication and for configuring port and address information, the network architecture includes the PortConfiguration object and the PortConfigurationProvider interface that is implemented by ProductionPortConfigurationProvider and SimulationPortConfigurationProvider. Each port configuration specifies a single port number for the beacon service, two port numbers for the data sending and the data receiving components of the data service, and the 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 a CRC32 check for packet content validity. If either of these checks fails, the packet is discarded and no further processing takes place.

## 3.6. Packet Handling

Conventional implementations of packet byte stream handling usually utilize the complicated pointer architecture of the C programming language 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, presents a Parcel object that acts as a stacked byte buffer with LIFO access to all primitive data types in Java and also arbitrary access to allow for easy CRC computations. For example, to construct a peer packet, or a byte array representation of a peer, one would call: Parcel.combineParcelables(address, location, velocity). Since address, location, and velocity each implements the Parcelable interface, they know how to create a parcel representation and hence, all the 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 with various types of primitive data types supported in Java such as float, double, int, byte, and char. 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.

## 3.7. Testing & Simulation

### 3.7.1. Logging and Debugging

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

### 3.7.2. 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 were 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.

### 3.7.3. Simulation

The simulation framework is written completely from scratch and does not rely on any third-party libraries such as ns3 or JiST/SWANS. The simulator is divided into two components – a command line based simulator called KiNCoL and a graphical user interface (GUI) based simulator. The command line based simulator is meant for developers and advanced users, while the GUI based simulator aims for ease of use and understandability. More details on the usage of these two tools will be put forth in the upcoming sections.

Aside from the user interface differences, the command line and GUI utilities share a similar design. They essentially take user input and set up a simulation environment based on this input.

Peers are created with random MAC addresses, random longitudes and latitudes, and random bearings, and are attached to random waypoint GPS devices that dictate their movements. In the command line based tool, the first peer is automatically designated as the destination. As mentioned earlier, for the purposes of our current research, destination nodes are always stationary. Therefore, in the simulation, destination nodes have a default GPS device instead in order to fix its position. In the GUI based tool, the user is allowed to select any node to serve as the receiver. Each peer is also attached to a statistics monitor, which enables the simulation framework to gather important metrics during a simulation run.

Once the simulation environment has been set up, it is ready to be run. To start the simulation, the program starts up the services of each peer, and waits a grace period for all peers to be fully initialized. Then in each iteration, it selects a random (mobile) peer to send a data packet to the (stationary) destination. After each data packet is sent, there is a delay. In the command line simulator, the number of data packets to send is set by the user as a command line argument, and therefore the simulation will stop after the specified number of data packets has been sent. On the other hand, the GUI simulator will run indefinitely until the user stops the simulation. In each iteration, statistics is updated. If a hostility factor has been set, the simulator also randomly chooses to destroy a peer based on this probability factor.

The command line simulator displays the statistical results at the end of a simulation run, while the GUI simulator provides a real-time display of the statistics. Once the simulation is stopped, the user can even copy the statistics output to the clipboard at the click of a button.

The GUI is completely written in Java Swing using built-in Java libraries. The WSIWYG editor provided by the Google WindowBuilder plugin for Eclipse was used to design the GUI, and further refinements were done by hand to add functionality and to make the program more object-oriented. For instance, certain user interface elements such as JTable were extended so that we could add our own customizations and encapsulate functionality.

Since the GUI simulator allowed for the addition and deletion of peers at any time (as opposed to the command line simulator), the peers were stored in an ArrayList dynamic array rather than a traditional static array as in the command line simulator.

A polling architecture is implemented to provide real-time user interface updates during a simulation run. The actual simulation code runs in a separate thread. This thread separation is required in order to maintain the responsiveness of the user interface, which is essential for a seamless user experience. The main thread utilizes a Java Timer to gather information from the running simulation at a constant interval. Such information includes statistics and the current position of each node. These data are either simply displayed on the user interface or used internally to paint the graphical representations of the nodes on the map in the user interface.

# 4. Simulation

## 4.1. Simulation Tools

Over the course of this development cycle, the authors developed two simulation tools for KimonoNet, a command-line simulator called KiNCoL and a graphical user interface (GUI) simulator.

Both simulation tools use the random waypoint model to simulate autonomous peers in the network. This model randomly selects a starting location for each peer and a random waypoint that the node will travel to at a uniform speed. Once the node reaches a waypoint, it then selects another waypoint and proceeds to that waypoint in a similar fashion.

Because the simulators run on a single host with a single NIC, a constraint is imposed that data packets are sent every 500 milliseconds. In real-world scenarios, this constraint would not exist, but it does in simulation due to a single NIC.

The simulation environment also assumes a single static destination while sending packets from network sources. This may cause some runs to yield low deliverability if a node is partitioned from the rest of the network.

## 4.2. KiNCoL Simulator

KiNCoL is a command line simulator that provides a simple way to test the protocol under various node configurations. It is run by setting the attribute mode-cl for KimonoNet, followed by a number of other configurable attributes:

1. number-of-peers: The total number of nodes in the network.
2. map-width (meters): The width of the world that the peers move within.
3. map-height (meters): The height of the world that the peers move within.
4. hostility-factor (decimal range ): The likelihood of a node vanishing from the network.
5. peer-speed (meters/second): The speed at which peers move between random waypoints.
6. number-of-packets: The number of data packets sent in the simulation.
7. beacon-timeout (milliseconds): The interval between sending beacons.

KiNCoL makes an assumption that all nodes have the same transmission distance of 150 meters. The number-of-peers, map-height and map-width attributes should be set with this in mind. For finer-grained control of nodes, the GUI simulator should be used instead.

KiNCoL also provides an alternative command-line mode mode-cl-gpsr that uses only one-hop awareness as similar to GPSR. This can be used to study the difference between one-hop and two-hop knowledge for the same beacon interval or in conjunction with an increased beacon interval to see how much added control traffic is needed for the same level of reliability with one-hop awareness.

## 4.3. GUI Simulator

The GUI simulator provides an alternate mechanism for testing the protocol with finer-grained control over the configuration of each individual node.

For each node added, the simulator supports configuring the following settings individually:

1. longitude: Position in world.
2. latitude: Position in world.
3. address: Unique identifier for node.
4. speed: The speed at which the peer moves between random waypoints.
5. bearing: The initial angle of the peer, although this will change over time based on the random waypoint model.

Once nodes have been arranged, one node should be selected and marked as the destination. The GUI simulator, like KiNCoL, will fix the position of this node, and all data packets will be routed through the network to this node.

The GUI simulator also supports editing several configurations globally:

1. beacon-service-timeout-random-additive (seconds): Nodes randomly wait a certain amount of time within this percentage of the beacon-service-timeout interval before sending to prevent synchronization of updates.
2. max-beacon-peers: How many peers a node will include in its neighbor report.
3. max-transmission-range (meters): The radius of the network horizon for each node.
4. beacon-service-timeout (milliseconds): The interval between sending beacons.
5. packet-loss-rate (decimal range ): The likelihood of a packet being lost during transmission.
6. hostility-factor: (decimal range ): The likelihood of a node vanishing from the network. This is set from the menu.
7. map-dimensions (meters): The height and width of the map. This is set from the menu.

The GUI simulator provides the ability to create and analyze node arrangements. Nodes are graphically represented on an interactive map. The user can simply click on a node to select it, drag the node to change its longitude and latitude, and rotate the mouse wheel to change its bearing. If one sets node speeds to zero, the performance of particular topologies may also be studied.

The GUI simulator also offers other intuitive features aimed to increase the user’s convenience and productivity, such as real-time statistics and peer property displays during an active simulation run, as well as a mouse tooltip that displays the cursor’s current position on the map in longitude and latitude.

## 4.4. Two-Hop Awareness

The KiNCoL simulation tool was used to study the behavior of nodes under KimonoNet with two-hop awareness.

Consider the following KiNCoL configuration:

* 27 peers, each with a speed of 15 m/s
* 450 m x 450 m map
* No hostility factor
* 10 seconds between beacons
* 100 data packets sent at 2 packets per second

When nodes move at 15 m/s with a transmission distance of 150 meters, this configuration sends a beacon after the node has moved the distance of the transmission radius. Further, this configuration places an average of twelve nodes within each radius, thus an -sparse graph.

This study included twelve runs under this configuration with two outliers excluded due to heavily partitioned random distributions.

*Average Delivery Rate.* On average, KimonoNet data packets had a delivery rate of 90.1% in -sparse networks.

*Data Packets*. A total of 4.3K data packets were transmitted to transport data across the -sparse network.

*Control Overhead.* Out of the 5.6K packets sent between nodes in the network, 1.2K were routing packets for a control overhead of 23.2%.

*Greedy Ratio*. Of the 8.9K packets involved in data transmission, 80.4% of them were forwarded via greedy routing mechanism.

While delivery ratio was far lower than reported in Karp and Kung [], this is because their simulation environment placed an average of 20 nodes within each transmission radius, whereas this research topic studies networks with an average of 8 nodes in the network horizon.

## 4.5. One-Hop Awareness

Karp and Kung achieved a delivery ratio of 97% in networks with twenty nodes average within transmission range. However, this research topic focuses on sparser networks, namely of twelve nodes average within the horizon.

Maintaining all settings as in § 4.5 but running in mode-cl-gpsr, this simulation again involved twelve runs with two outliers excluded due to heavily partitioned random distributions.

*Average Delivery Rate*. On average, one-hop awareness yielded a delivery rate of 79.2% in -sparse networks. This was 10.9% lower than with two-hop awareness.

*Data Packets*. In order to transport data across the -sparse network with only one-hop awareness, a total of 9.1K data packets were transmitted. This was 212% of the network traffic required for delivery of the same number of data packets with two-hop awareness.

*Control Overhead.* With beacon intervals the same, total routing traffic was 1.4K, roughly the same as in the two-hop simulation. However, given that 9.1K data packets were sent, this yielded a control overhead of 13.3%. However, although this ratio is significantly smaller than in two-hop awareness, this is a result of an increase in data packets required for transmission, not a decrease in control traffic.

*Greedy Ratio*. Of the 9.1K packets involved in data transmission, 51.7% of them were forwarded via greedy routing mechanism. This is 28.7% less forwarded optimally for one-hop awareness in the -sparse network. The sub-optimality of perimeter forwarding is evident when coupled with the fact that one-hop aware routing required 212% more packets to route the same amount of data over the network.

## 4.6. Difficulties

### 4.6.1. NIC Oversaturation

The most significant difficulty encountered during simulation came from oversaturation of the test machine’s NIC. This manifested in the form of CRC errors in packet transmission. To handle this issue, data intervals were reduced to the point that such conflicts did not occur. However, this led to an inflation of the control traffic ratio metric that rendered it all but useless.

This issue would likely not manifest in real world environments because all traffic is not routed through the NIC, but instead only between nodes within the same horizon. In an 8-sparse network, this would be inconsequential.

If the test environment were built to forego the use of a NIC to test only protocol internals, this difficulty would not have been encountered. However, because of the limited time for implementation, the KimonoNet prototype includes both the simulation and the production modes within the same executable, and thus it is befallen to this implementation challenge.

### 4.6.2. Perimeter Routing Implementation

Based on simulation results and investigation into its implementation, the method for perimeter routing in the prototype is non-optimal. However, while this may have reduced the efficiency of perimeter routing, this does not invalidate the conclusion that two-hop awareness provides greater likelihood of successful greedy routing.

# 5. ConclusionS

## 5.1. Outcomes

This research topic focuses on coordinate-based routing in sparse, fluid networks, motivated by the use case of providing routing over network-enabled autonomous nodes such as unmanned aerial vehicles. To this extent, it proposes an extension to GPSR [] that increases node neighborhood knowledge beyond the network horizon to include neighbors of neighbors for two-hop awareness. It then uses this knowledge to proactively update its routing table.

While B. Karp and H. T. Kung demonstrated the optimality of GPSR in -sparse networks, simulation results in this paper show how deliverability decreases in sparser, more fluid networks, studying an -sparse network with beacon transmission rates directly equivalent to the amount of time it takes a node to travel the radius of the transmission horizon.

For such -sparse networks, where each node has less than half the connectedness studied by B. Karp and H. T. Kung, as well as greater mobility, the authors found that two-hop awareness provided approximately a 10% higher delivery ratio. Further, simulation results showed that more packets took advantage of greedy routing with two-hop awareness, although a flaw in the perimeter forwarding implementation may have inflated the value ascertained somewhat.

This outcome was roughly as expected. This is because nodes with one-hop awareness can only consider neighbors that they have already encountered, whereas nodes with two-hop awareness more quickly take advantage of nodes that enter into their network range.

For one-hop awareness to have the same delivery ratio as in two-hop awareness, more control traffic is required as the frequency of beacon packets a node must send increases to more quickly inform peers of their existence.

While the simulation encountered issues with NIC oversaturation and the implementation of perimeter routing, simulation results still affirm that two-hop awareness has a positive effect on routing in fluid, sparse networks.

## 5.2. Future Work

### 5.2.1. Modeling and Simulations

The prototype and simulation of KimonoNet demonstrated that under fluid, sparse networks, such as where each node has an average of eight neighbors within its network horizon, two-hop awareness is indeed advantageous. However, further modeling based on an implementation that accounts for the difficulties covered in § 4.6 is still necessary to comprehensively affirm the outcomes reported in this paper. Given the well-structured, object-oriented approach of the prototype, such modeling can be achieved by simply amending the existing implementation rather than reengineering the KimonoNet protocol altogether.

### 5.2.2. Mobile Destination Nodes

As specified in § 1.3.2, neither the protocol nor the implementation provides support for mobile destination nodes. In the protocol, this is a conscious design decision, as a discovery service is fundamentally different than a routing service; the protocol thus decouples the two concepts so that changes to one do not impact the other. However, any production implementation likely requires such a discovery service, whereas the prototype in this project does not implement mobile destination discovery, as its primary purpose was to test routing performance.

Support for mobile destination nodes provides two advantages: (1) routing between any two peers in the mesh network and (2) support for duplex protocols. The UAV use case does not require the former use case, but support for duplex communication is needed to implement reliable transport.

### 5.2.3. Reliable Quality-of-Service

Although considered in the project proposal and requirements, a reliable quality-of-service was not implemented as part of this initial effort. However, given an almost-certain need for reliable delivery in most military scenarios, KimonoNet should be extended to provide reliable delivery.

Tunneling a reliable transport protocol like TCP provides one way to implement reliable delivery. However, as per § 1.3.3, without support for mobile destinations, this is not possible because TCP and other reliable transport protocols require bidirectional communication to provide acknowledgements. If duplex communication is implemented within KimonoNet, this approach benefits from flexibility. However, it comes at the cost of requiring end-to-end communication to detect and resolve losses, as opposed to network-enabled reliability, which can handle losses locally within transit.

Consequently, the KimonoNet protocol itself might also be developed further to provide reliable delivery itself. As opposed to end-to-end reliability, nodes along the transit path may assist with reliability, thus reducing traffic and time for loss detection and recovery. However, this requires each node to maintain more state, one of the things that GPSR seeks to avoid. It also more tightly couples the reliability mechanism into the KimonoNet protocol.

### 5.2.4. Predictive Forwarding Delay

The KimonoNet protocol maintains two-hop awareness in order to perform predictive neighbor maintenance. However, it could also more aggressively use this knowledge to delay data forwarding if the topology is anticipated to change favorably. In a relatively fixed network, this doesn’t provide an advantage, because changes will be minimal, but in a highly fluid network, over a relatively short period of time, a neighborhood could change significantly.

As a basic example, consider the case where a node is a local maximum. Under GPSR and the KimonoNet protocol described to this point, the packet would enter perimeter forwarding. However, if a node recognizes that a neighbor in its two-hop routing table will soon become a greedy candidate, it could forestall transmission until that time, and then send the packet via greedy, the preferred routing mechanism. However, the mechanics of this must be explored more thoroughly, as this case will cause selection to fall-behind in time, even if still making a near-optimal route selection.

# 6. Acknowledgements

## 6.1. Group

Given the scope of this project and the limited time constraints, the group divided the work into distinct units of responsibility for each member. In parallel, the team developed the protocol, prototype and simulation environment, and then they integrated these three distinct pieces as the term drew to a close.

Eric Bollens created the conceptual framework for KimonoNet and was the primary author of its documentation. He guided its use of existing protocols and proposed improvements for suited to fluid, sparse networks. He also compiled the project proposal and requirements document and wrote both the protocol specification and the final report.

Zorayr Khalapyan managed architecture of the KimonoNet prototype, including its overall class hierarchy and its implementations of peer packet parceling, beaconing and communication. He also wrote the first draft of the project proposal and assisted with the testing infrastructure.

Wade Norris implemented routing and transport within the KimonoNet prototype based on the protocol specification. This included greedy and perimeter routing and predictive neighbor maintenance. He also contributed heavily to the requirements document.

James Hung developed unit tests for foundation classes in the prototype and then implemented the simulation environment for the KimonoNet prototype. This simulation environment included both the GUI-based simulator and the KiNCoL command-line utility.

## 6.2. Credits

We thank G. Pau, a mentor who advised us throughout this process, which itself was an outcome of his CS 114: Peer-to-Peer Networks class at UCLA.

Further, we thank B. Karp and H. T. Kung for their work on Greedy Perimeter Stateless Routing [], as well as G. G. Finn, who first proposed greedy routing through a coordinate-based system [].

# 7. REFERENCES

**ERIC TODO**

[] Bollens, E., Hung, J., Khalapyan, Z., Norris, W., Mar. 2012, KimonoNet: Protocol Specification Document; University of California, Los Angeles, CA.

[] Bollens, E., Hung, J., Khalapyan, Z., Norris, W., Feb 2012, KimonoNet: Software Requirements Specification; University of California, Los Angeles, CA.

[] Finn, G. G., Mar. 1987, Routing and addressing problems in large metropolitan-scale internetworks; Tech. Rep. ISI/RR-87-180, Information Sciences Institute.

[] Karp, B. Geographic Routing for Wireless Networks; Ph.D. Dissertation, Harvard University, Cambridge, MA., Oct. 2000.

[] Karp, B., Kung, H. T., 2000, GPSR: Greedy Perimeter Stateless Routing for Wireless Networks; in Proceedings of the sixth ACM/IEEE International Conference on Mobile Computing and Networking (MOBICON 2000), Boston, MA, Aug. 2000, pp. 243-254.

[] Sinnott, R. W., 1984, Virtues of Haversine; Sky and Telescope, vol. 68, no. 2, p. 159.

[] Williams, Ed, 2011, Aviation Formulary;  
 <http://williams.best.vwh.net/avform.htm>.