KimonoNet

Enabling Efficient and Reliable Inter-UAV Fluid Data Communication Link

Document
Version 1.0.09
March 23, 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 the deployment of ad hoc wireless networks 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, such 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 might 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 of conventional wired 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 B. Karp and H. T. Kung [6], and then extends it by propagating two-hop neighbor knowledge through control beacons. This strategy allows longer effective intervals 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 is available in *KimonoNet: Protocol Specification Document* [1].

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 simulation results that measure the viability and effectiveness of KimonoNet through metrics including packet delivery, routing overhead, and path optimality.

1.3. 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.

1.4. User Characteristics

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.

1.4.1. Autonomous Nodes

Autonomous nodes, the primary constituents of the network, have (1) awareness of their position and velocity, (2) a NIC that supports ad hoc communication, (3) support for 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 the 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 packets passed to them by other nodes and forward them based on the routing algorithm.

1.4.2. End Points

A command post or other external uplink to the Internet serves as the destination endpoint of a packet originating from within the ad hoc network. Because this protocol does not consider a mechanism to determine endpoint location, it requires that 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 have (1) awareness of position and velocity, (2) a NIC that supports ad hoc communication, (3) support for 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 within the packet to reply. Because an autonomous node has variable position, all responses from an end point should have quality-of-service classification of time-sensitive and be loss tolerant.

1.5. Scope

1.5.1. 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 over Layer 3 or 4 while leveraging wireless broadcast or multicast.

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

1.5.2. Destination Locations

As described here and in associated materials, the KimonoNet protocol 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 provide the destination location. This design decision may present challenges for some communication, but it is purposeful in that it decouples the routing protocol from search mechanisms, for which significant research has already been devoted and significant advances are still likely expected.

For simplicity, the prototype client does not provide a search algorithm either to locate a mobile destination but instead fixes the destination in simulations. A flooding search algorithm could be implemented in a future iteration so that nodes may communicate with mobile destinations.

1.5.3. 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 tunnel duplex transport 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, the location and velocity of the initial sender may be stale and inaccurate. Large network diameters accentuate this problem.

1.5.4. 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.

1.6. 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 changes swiftly as nodes enter and recede from the network horizon. The routing algorithm handles this by considering neighbors and the neighbors of 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 impact of such a disappearance.

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 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 this void.

This constraint is the reason that distance-vector and link-state protocols are very non-optimal: by the time that 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 nodes must 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 [4]. B. Karp and H. T. Kung reduced routing overhead by providing a method for detouring without flooding through graph planarization [5] [6]. Their work, namely Greedy Perimeter Stateless Routing, serves as the basis for KimonoNet, which then extended to leverage two-hop awareness for velocity-based prediction of neighborhood changes.

2. PROTOCOL

At its heart, the KimonoNet protocol must define two basic functions and the interplay between them:

1. Nodes provide information that allows nearby nodes to recognize them as neighbors and predict how their position will change over time in order to effectively route data in (2).
2. Nodes forward data packets through the network to an intended final destination, making near optimal routing decisions at each hop based solely on local network knowledge ascertained by (1).

In order to accomplish these goals, KimonoNet leverages GPSR [6] 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 node identifier, 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 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 a two-hop graph of both its neighbors and the neighbors of its neighbors. It may then use this information 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 routing table and then responds with a beacon. 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 about another node, the other node appears in the node's neighborhood report, and therefore the other node will not reply back with its own location.

2.2. Routing

2.2.1. Greedy Forwarding

First proposed by G. G. Finn [4], 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 thus uses greedy forwarding as its preferred routing mechanism.

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

$$p = \min(\{x | \forall r \in R, x = DIST(r, d)\})$$

The KimonoNet protocol recommends the haversine formula for the $DIST$ method [8]:

$$DIST\big((\psi_1, \phi_1), (\psi_1, \phi_1)\big) = 2r * \arcsin\left(\sqrt{\begin{matrix} \sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos(\phi_2 - \phi_1) \\ + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\psi_2 - \psi_1}{2}\right) \end{matrix}}\right)$$

In some implementations, if IEEE 754 64-bit floating points numbers are available or precision to the meter is not needed, the law of cosines may prove more efficient for computing $DIST$ [9]. Meanwhile, if node hardware cannot perform these computations effectively or if network ranges are mathematically insignificant in relation to the Earth's radius r, distances may be computed as if on two-dimensional plane. In some scenarios, it also might be fitting to extend the calculations 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 generally makes near optimal path selections and requires minimal router state, it has two problems that must be acknowledged:

(1) In specially constructed topologies, it can be demonstrated that greedy selection is sub-optimal by a substantial amount. 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, a different mechanism must be used 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 [4]. However, flooding creates inefficiencies in network traffic, and minimizing traffic while flooding requires nodes to maintain state. As such, B. Karp and H. T. Kung described a perimeter-based alternative that detours around a void when a node is a local maximum [5] [6].

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 resume. If the node undergoes a cycle before this time, then the destination is unreachable.

This method ensures that a packet will not travel indefinitely because it will be dropped as soon as it traverses all reachable nodes and arrives back at the start of the tour. 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 heuristic worked in 99.5% of cases [6], an edge may be removed that partitions the graph. The latter approach does not have this issue because it computes a planar graph and removes edges not in this graph.

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

$$N = \{x | \forall r \in R, x = BEARING(r, s)\}$$

$$\text{if } B = \{x | n \in N, x = n_b - s_b > 0\}: \min(B)$$

$$else\ B = \{x | n \in N, x = n_b - s_b \leq 0\}: \min(B)$$

The KimonoNet protocol recommends the following for the $BEARING$ method [9]:

$$BEARING\big((\psi_1, \phi_1), (\psi_1, \phi_1)\big) = \operatorname{arctan2}(\cos(\phi_1)\sin(\phi_2) - \sin(\phi_1)\cos(\phi_2)\cos(\psi_2 - \psi_1), \sin(\psi_2 - \psi_1)\cos(\phi_2))$$

As with $DIST$, 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 transitions back to greedy mode as soon as 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 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 calculations to predict neighborhood changes over time.

In the case of one-hop awareness, this increases reliability and efficiency because each node can compute how neighbor 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 under the KimonoNet protocol includes information not just about a node but also its neighbors, it is possible to have two-hop awareness. This increases route efficiency reduces and reliability and also decreases required control traffic. With a simple computation, nodes can predict when a neighbor of a neighbor will enter into the network horizon and thus become a direct neighbor. 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. 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 r, the speed of the node s, the time since the neighbor information was generated Δt, the bearing of the node θ, and the location $(longitude, latitude) = (\psi_0, \phi_0)$ of the node as defined in ROUTING-TABLE-2, the new location $(longitude, latitude) = (\psi_1, \phi_1)$ may be calculated as [9]:

$$\phi_1 = \operatorname{asin}\left(\sin(\phi_0)\cos\left(\frac{s\Delta t}{r}\right) + \cos(\phi_0)\sin\left(\frac{s\Delta t}{r}\right)\cos(\theta)\right)$$

$$\psi_1 = \psi_0 + \operatorname{arctan2}\left(\cos\left(\frac{s\Delta t}{r}\right) - \sin(\phi_0)\sin(\phi_1), \sin(\theta)\sin\left(\frac{s\Delta t}{r}\right)\cos(\phi_0)\right)$$

As with $DIST$ and $BEARING$, alternative, less computationally intensive methods may use a two-dimensional plane where necessary and viable based on node and network conditions. It should be recognized, however, that these calculations are propagated between neighboring nodes through beacons and therefore, while calculation uniformity is not a requisite, a lack thereof may lead to differing ways that nodes regard their neighbors.

2.2.4. Quality-of-Service

KimonoNet provides three qualities of service:

(1) Control data is data 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 forwarded across the network. It is handled with 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 grade for reliable data delivery [2]. However, this version of the protocol does not include support for reliability, as the authors quickly found that a good deal more research was needed to effectively implement reliable delivery. More discussion on this topic may be found in § 5.2.3.

3. IMPLEMENTATION

3.1. Architecture

3.1.1. Overview

The core framework architecture of the application may be subdivided into three overarching components:

1. The service layer includes threads for beacon dispatch, data reception and dispatch, and geo-awareness.
2. The packet handler manages packing, unpacking, and validation of packets in the network.
3. The peer representation structure encapsulates all information pertaining to a single peer operating in a specific configuration.

Below the core framework, the implementation also provides utilities for low-level network management and byte manipulation.

For gathering and analyzing information pertaining to performance and efficiency of the KimonoNet protocol, the framework attaches to a stats monitor under the master-slave paradigm that allows for analysis of both simulation and production behavior.

The use of the Java language system also offers several critical advantages over low-level, native languages such as C or C++. The application relies heavily on the object-oriented features of Java, as well as Java's robust threading support.

3.1.2. Conventions & Notation

The implementation attempts to follow as many conventions of the Java language as possible [3]. As such, all local and instance variables use lower camel case notation, while constants are written in upper case with underscore word separation. Modularized within 18 packages, the classes are accordingly positioned for ease of management and version control. In terms of nomenclature, this implementation prefers moderately long and descriptive names to abridged, cryptic names. Although this produces rather long statements, it improves code readability significantly.

3.2. Service Layer

Each service implements an interface that mandates various thread management methods such as a graceful start and stop. Further, given that each service is likely run in a separate thread, the application uses blocking or synchronized structures for shared memory access. For example, `ROUTING-TABLE-1` and `ROUTING-TABLE-2`, two critical structures in the KimonoNet protocol, are stored within blocking hash maps that allow only a single thread access at a time. When possible, the implementation preferred such intrinsic locks and synchronization as provided by the Java native library. Furthermore, caching was improved by marking static and immutable structures with the `final` keyword.

3.2.1. Beacon Service

The beacon service has three responsibilities: transmission of beacon packets, interpretation of received beacon packets, and computing updates to the routing tables. The implementation uses a blocking method to read packets, and, after a timeout is reached, a new beacon is dispatched. This timeout is a configuration variable within the peer's environment.

Recognizing synchronization and contention issues for beacons at the link layer, the timeout value is offset by a random additive value one-fifth of the beacon timeout interval. In tests, this design decision substantially improved beacon transmissions and cut control overhead.

3.2.2. Data Service

The data service provides transmission and reception of data packets. The data service runs within a peer agent and includes two concurrent threads. One data service thread sends packets by popping them from a priority queue, and the other receives packets and handles them appropriately.

The receiving service listens for inbound data packets; when one is received, it may (1) throw it away because it is not intended for this node, (2) deliver it to the upper layer protocol if this is the final destination for the packet, or (3) add the data packet to the packet queue to be 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, then `COMMUNICATION` then finally `REGULAR` packets. If the packet queue is empty, the thread sleeps so as not to waste CPU; it is again awoken by the receiving service when a packet is added to the queue. After a packet is popped form the queue, the routing protocol is used to determine the next hop for packet based on data in the routing table.

When a packet needs routing, the service first updates `ROUTING-TABLE-1` based on the position, velocity and time of all nodes contained in `ROUTING-TABLE-2`. If peers from `ROUTING-TABLE-2` have come into range of the node, they are transferred to `ROUTING-TABLE-1`. Meanwhile, if nodes have left transmission range, they are removed from `ROUTING-TABLE-1`. Once the routing tables have been updated, the packet is then routed based on GPSR and the data contained within `ROUTING-TABLE-1`.

The data packet is the fundamental element of communication within the data service. This packet contains information travelling from some source node to some destination that must thus be routed over the network. As such, it contains the address of the destination node, the location of the destination node, the address of the next hop ascertained by the routing protocol, 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`), and an integer checksum for the set of header information. Additionally, it includes an extended header section used for `PERIMETER` forwarding which is set `NULL` when in `GREEDY` mode. Alternatively, the protocol itself allows for these bytes to be stripped when in `GREEDY` mode to reduce payload size.

The data packet is constructed from a parcel, byte away or in an object-oriented fashion using the 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 are available for adjustments during routing. The packet also allows for parceling of the packet [7], which computes and sets the CRC32 checksum field for header contents.

3.2.3. Geo-Device Service

The geo-device service polls for the current GPS location and velocity at a certain frequency. Since various ways exist for fetching the GPS location, the `GeoDevice` is an interface with two implementations: `DefaultGeoDevice`, representing a stationary node such as a sink, and `RandomWaypointGeoDevice`, which defines GPS locations based on the random waypoint model. Future production level implementations would 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-readable name.

Furthermore, each peer has an agent responsible for managing shared memory, environment and configuration variables, and for associating the services layer with the peer. Agents are also attached to a `StatMonitor` to report sent, received, and dropped packets for analysis.

3.4. Environment Configuration

Agents also store a `PeerEnvironment` structure, which is a flexible and extendible vault for specifying peer-related parameters. The current implementation includes settings for maximum transmission range, beacon service timeout, and beacon service timeout additive ratio. This structure exists to enable future extensibility and also simplifies implementation of configuration during simulation.

3.5. Network Structures & Utilities

The communication channel used by peers is defined by a `Connection` interface. 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.

The prototype includes `UDPConnection` and `UDPMulticastConnection`. Although the former is mandated for production mode, allowing multiple nodes to attach to the same port number and communicate with each other, `UDPMulticastConnection` was used in simulation mode for simplicity.

To allow an easy switch between different modes of communication and also for configuring port and address information, the network architecture includes both a `PortConfiguration` object and an interface `PortConfigurationProvider` interface that the production and simulation port configuration providers both implement. Each configuration specifies a port number for beacon service, port numbers for data transmission and retrieval respectively, and a network interface IP address used for initializing communication channels.

Currently, the simulation uses IP broadcast 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 check fails, the packet is discarded and no further action occurs.

3.6. Packet Handling

Conventionally, packet handling is often implemented using pointer arithmetic and complex memory management techniques in C. This leads to highly coupled and complicated code structures that are almost impossible to manage or extend. To address these issues, the KimonoNet implementers sought an object-oriented alternative that would decrease redundancy, enhance exception management, and support greater flexibility and extensibility. The final design, inspired by packet handling in the Android operating system [7], presents a Parcel object that acted as a stacked byte buffer with LIFO access to all primitive data types as well as arbitrary access simplifying CRC computations.

To construct a peer packet under this paradigm, the following code is sufficient:

```
Parcel.combineParcelables(add
ress, location, velocity);
```

Since address, location, and velocity all implement the `Parcelable` interface, the peer must simply combine the parcels. 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. This method also supports all primitive types.

Once combined, the data is represented as a Java-native byte buffer, output as a byte array and sent over the network. This workflow also applies to parsing byte arrays received from the network.

Comparing this architecture with the C-style byte buffer processing, developers here can concentrate on business logic of the application as opposed to coding redundant and complicated packet packing and unpacking functions.

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

The implementation heavily utilizes the JUnit 4 testing framework to ensure the accuracy of each method in the implementation. This framework does not require modifications to existing code, and instead reside in their own package. Furthermore, JUnit only adds one external dependency, which is required only to run the tests. In other words, the JUnit library does not need to be included under normal operation.

Unit tests are invoked by running the project as a JUnit test instead of as a standard Java application. Methods were rigorously tested to ensure that they produce expected output and do not throw unexpected exceptions. Furthermore, they enforced consistency with the protocol specification, especially in the case of packet formats with the correct lengths and offsets. The test suite is rigorous all the way down to manually crafting packets at the byte level to use for comparison.

3.7.3. Simulation

The simulation environment includes two interfaces, a command-line simulator (KiNCoL) and a graphical user interface (GUI). The former is intended primarily for use by developers and in statistical analysis, while the later provides an environment where a user can interact with the simulation. The simulation framework does not rely on any third-party libraries such as ns3 or JiST/SWANS. While usage details of the simulation tools are covered in § 4.1, this section establishes related implementation details.

At their core, the KiNCoL and GUI utilities share a similar design. They accept user input and construct 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 and its position is fixed. In the GUI based tool, the user may select any node as the receiver. In both cases, the simulation environment then attaches all peers to the 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. For each iteration, it then selects a random (mobile) peer to send a data packet to the (stationary) destination. After each data packet is sent, it introduces a short delay and handles statistics. If a hostility factor has been set, the simulator also randomly chooses to destroy a peer based on this probability factor. In the command line simulator, the number of data packets sent is user-defined, and therefore the simulation will stop after the specified number of data packets has been sent. The GUI simulator, meanwhile, will run indefinitely until manually stopped. The command line simulator displays the statistical results at the end of a simulation run, while the GUI simulator provides a real-time statistical output.

The GUI was designed to be intuitive and easy-to-use, providing a visual representation of nodes and their interactions, as well as the ability to change various settings, some globally and some individually for each node, whereas KiNCoL is a statistical tool. The GUI was written

The GUI was written in Java Swing using built-in Java libraries. Google's WindowBuilder plugin for Eclipse was used to design the interface, and further refinements were made by hand. The GUI simulator also had to make some extra allowances for real-time user-introduced changes, such as the ability to add or delete a peer at any time. The real-time user interface updates are made through polling whereby the simulation actually runs in a separate thread. This architectural decision ensures responsiveness of the user interface, which is essential for a good 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. This data is used both for statistical output in the user interface and to paint a graphical depiction of the map and its nodes.

4. SIMULATION

4.1. Simulation Tools

Over the course of this development cycle, the authors developed two simulation tools, a command-line simulator (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 [0,1]): 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 alternative 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 several global configurations settings:

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 [0,1]): The likelihood of a packet being lost during transmission.
8. `hostility-factor`: (decimal range [0,1]): The likelihood of a node vanishing from the network. This is set from the menu.
6. `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 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. While real-world scenarios might have very different transmission ranges than studied here, the principle is the same. Further, this configuration places an average of eight nodes within each radius, thus termed an 8μ-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 8μ-sparse networks.

Data Packets. A total of 4.3K data packets were transmitted to transport data across the 8μ-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. Out of the 8.9K packets involved in data transmission, 80.4% of them were forwarded via greedy routing mechanism.

While the delivery ratio is lower than reported in B. Karp and H. T. Kung even with only one-hop awareness [6], this is because their simulation environment placed an average of twenty nodes within each transmission radius, whereas this research topic studies networks with an average of eight nodes in the network horizon. The sparseness of this network graph greatly changes deliverability.

4.5. One-Hop Awareness

B. Karp and H. T. Kung achieved a delivery ratio of 97% in networks with twenty nodes average within transmission range [6]. However, this research topic focuses on sparser networks, namely with an average of only eight nodes in each node's network horizon.

Maintaining all settings the same as in § 4.4 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 8μ-sparse networks. This was 10.9% lower than with two-hop awareness.

Data Packets. In order to transport data across the 8μ -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 number as in the two-hop simulation. However, given that 9.1K data packets were sent, this yielded a control overhead of 13.3%. Although this ratio is significantly lower than in two-hop awareness, this is a result of increased 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 8μ-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 than when two-hop awareness was provided.

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 [5] [6] that increases node neighborhood knowledge beyond the node's 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 strength of GPSR in 20μ-sparse networks, simulation results in this paper show how deliverability decreases in sparser, more fluid networks, studying an 8μ-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 8μ-sparse networks, where each node has less than half the connectedness studied by B. Karp and H. T. Kung, in addition to greater mobility per beacon interval, 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 anticipated. 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 § 1.5.2, 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 § 1.5.3, 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. In parallel, the team developed the protocol, prototype and simulation environment, and then they integrated these distinct pieces as the term drew to a close.

Eric Bollens created the conceptual framework for KimonoNet and was the primary author of the protocol documentation and the final paper. Throughout the term, he guided KimonoNet's development and proposed improvements suited to fluid, sparse networks. He also compiled the project proposal and requirements document, aas well as collecting, interpreting and compiling the simulation results presented in this paper.

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, assisted with the testing infrastructure, and put in time making simulation results more accurate, packaging the KimonoNet implementation, and writing up details about the implementation for inclusion within this paper.

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 and made numerous bug fixes for the packaged version of the KimonoNet implementation.

James Hung developed unit tests for foundation classes in the prototype and implemented the simulation environment for KimonoNet. This included work on the GUI-based simulator and the KiNCoL command-line utility. He also edited and assisted on both the final paper and packaging of the KimonoNet implementation.

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 [5] [6], as well as G. G. Finn, who first proposed greedy routing through a coordinate-based system [4].

7. REFERENCES

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

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

[3] Code Conventions for the Java Programming Language; Sun Microsystems, 1999.

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

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

[6] 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.

[7] Parcelable; Android Developers: Reference, Google, Inc., 2012.

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

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