A Modular Network Layer for Sensornets

Cheng Tien Ee*, Rodrigo Fonseca*, Sukun Kim*, Daekyeong Moon*, Arsalan Tavakoli*,
David Culler*†, Scott Shenker*‡, Ion Stoica*

Abstract

An overall sensornet architecture would help tame the increasingly complex structure of wireless sensornet software and help foster greater interoperability between different codebases. A previous step in this direction is the Sensornet Protocol (SP), a unifying link-abstraction layer. This paper takes the natural next step by proposing a modular network-layer for sensornets that sits atop SP. This modularity eases implementation of new protocols by increasing code reuse, and enables co-existing protocols to share and reduce code and resources consumed at run-time. We demonstrate how current protocols can be decomposed into this modular structure and show that the costs, in performance and code footprint, are minimal relative to their monolithic counterparts.

1 Introduction

The field of wireless sensornets (hereafter, sensornets) has made great strides over the past few years, producing better devices, larger deployments, and more functional and stable systems. The different and varied nature of sensornet applications, coupled with heavy need for optimization, called for an exploratory phase in which boundaries between hardware/software, application/OS, and networking components were flexible and in flux [7]. As a result, there are several vertically integrated designs, created by separate research groups, which employ quite different modularities.

Across these designs, there is a general lack of consistency in terms of the functionalities implemented in modules as well as their interfaces, resulting in unnecessary coupling between modules. The creation of new protocols thus requires more effort to reorganize functionalities or even reimplement them from scratch. [1] Inconsistencies in service interfaces also cause the porting of applications onto different protocols to become non-trivial. Additionally, co-existing protocols with modules implementing overlapping functionalities unnecessarily consume more resources in terms of memory and energy, and can therefore reduce the lifetime of a sensornet. Thus, the lack of an overall sensornet architecture minimizes code reuse, complicates porting, and leads to increased memory consumption for already resource-constrained systems.

The first steps towards such an architecture were taken in [4, 21], which identified the narrow waist of the sensornet architecture as lying between the link and network layers. In this paper we take the next step in this endeavor by defining a modular network layer.

Our goals are simple: to increase code reuse and run-time sharing. Code reuse will foster more rapid protocol and application development, as well as greater synergy between various research groups. Run-time sharing refers to the sharing of code and resources such as memory and radio, and will allow several protocols to co-exist without burdensome memory requirements or contention problems. Even though most current applications are simple and require just a single network protocol, future developments may result in usage of multiple ones in the same network.

To accomplish these goals, we start by outlining the services provided by and functionalities implemented in the network layer. These requirements lead to a componentized network layer consisting of reusable modules from which we can easily construct network-layer protocols.

It is challenging to find the right granularity at which to break up functionality at the network layer; a very fine-grained decomposition will incur unnecessary run-time overhead, while too coarse a decomposition will not leverage all of the possible sharing and may result in significant reimplementation. In Section 3, we present why and how we modularize routing protocols to architect network layer, and we describe the basic modules in

*Computer Science Division, University of California, Berkeley
†Arch Rock Corporation
‡International Computer Science Institute (ICSI)

Section 4.

As we show in Section 5, we have successfully implemented multiple published sensornet routing protocols in our architecture. The ability to accommodate a wider variety of existing protocols hopefully minimizes changes with the advent of new ones. We also implemented a number of previously inexistent variations of these protocols by replacing specific modules[2]. Furthermore, the performance cost of this modularity must be low, otherwise designers will circumvent the proposed interfaces, undermining any benefits of the architecture. Thus, in Section 7 we quantify both the performance costs for our modular implementations (which are minimal) and the reductions in protocol-specific code (which are significant). We conclude in Section 8 with a discussion of future challenges for a sensornet architecture.

2 Related Work

This paper builds on two previous pieces of work. The creation of an overall sensornet architecture was proposed in [4]. Following the example of the Internet architecture, where the "narrow waist" allowed rapid innovation both above and below IP, the paper starts by trying to locate the narrow waist of a sensornet architecture. The Internet's narrow waist, IP, provides the abstraction of point-to-point (or point-to-multipoint) best effort packet delivery. This is not a suitable unifying abstraction for sensornets because they have a far wider variety of packet delivery models — such as convergecast (many-to-one), dissemination, data-centric routing, data-centric storage, and others — some of which employ application-specific processing at each hop. Furthermore, IP also provides a standard addressing scheme which is insufficient for sensornets. Given this network-layer diversity, the natural location for the sensornet narrow waist lies lower, between the link and network layers. This idea was substantiated in the SP proposal [21] as a unifying link layer abstraction. In this paper we build upon SP and address the next natural step by proposing an architecture for the network layer in sensornets.

Our work is also inspired by a number of prior efforts in creating modular systems in different contexts. The x-Kernel [10] is an operating system and framework for protocol implementation that combined composability with performance. It focuses on high-performance communication among complete, stand-alone protocols, whereas we attempt to distill many protocols to their common elements in order to maximize reuse.

The Click modular router [15], on the other hand, proposes a flexible composition model for packet-processing modules that enables fine-grained extensions to the forwarding path of an IP router. In contrast, we attempt to modularize entire protocols at a level coarse enough to reduce the effort required to piece separate components together, yet fine enough to ensure flexibility of combinations. We believe that the components proposed in this paper can be constructed from Click-like elements and are complementary.

Maté [17] is a virtual machine enabling efficient dissemination of code in sensornets. Much like Click, it is concerned with low-level modularity, and does not focus on the definition and construction of multiple network protocols. It is a tool for code dissemination, and as such can be used to distribute network protocols in sensornets. We believe that Maté is complementary to our work.

The work by Condie et.al. in [3] deals with composable transport layer protocols for DHTs, and employs a dataflow model akin to Click. The authors observe that highly-distributed Internet systems exhibit more diverse traffic patterns, and are more suitable for modular protocol designs. We can draw a parallel with the network layer in sensornets, where one of the motivations for modularity is diversity. However, they only deal with the data path, treating routing as a black box. Furthermore, their primary focus is on ease of experimenting with protocol variations, and not run-time code reuse and sharing.

MACEDON [22] provides a framework to describe overlay protocols in the form of finite state machines, enabling the concise expression of any overlay protocol resulting in ease of implementation. P2 [19] allows for declarative specification of new overlay protocols. Both MACEDON and P2 focus on reducing the effort required to generate a single overlay network layer, whereas we also consider co-existing ones.

Finally, Aspect-Oriented Programming (AOP) [13] is a programming technique that allows for isolation, composition and reuse of code that cross-cut different objects or modules. In this paper we focus on network layer properties that can be cleanly separated and encapsulated into distinct components, leaving those that affect multiple components in systemic ways to future work.

3 Modular Network Layer

In this section, we present our modular network layer that aims to achieve the two goals set forth in Section 1: (1) code reuse, and (2) run-time sharing. Achieving these goals fundamentally requires one to decompose the network layer into smaller components that can be re-used by various protocols.

To motivate and provide intuition behind our decomposition, consider the five network protocols shown in Figure 1(a). While these network protocols expose different service interfaces and come with their own implementation, a more careful inspection reveals multiple commonalities among them (Figure 1b). Identifying and encapsulating common functionalities would make

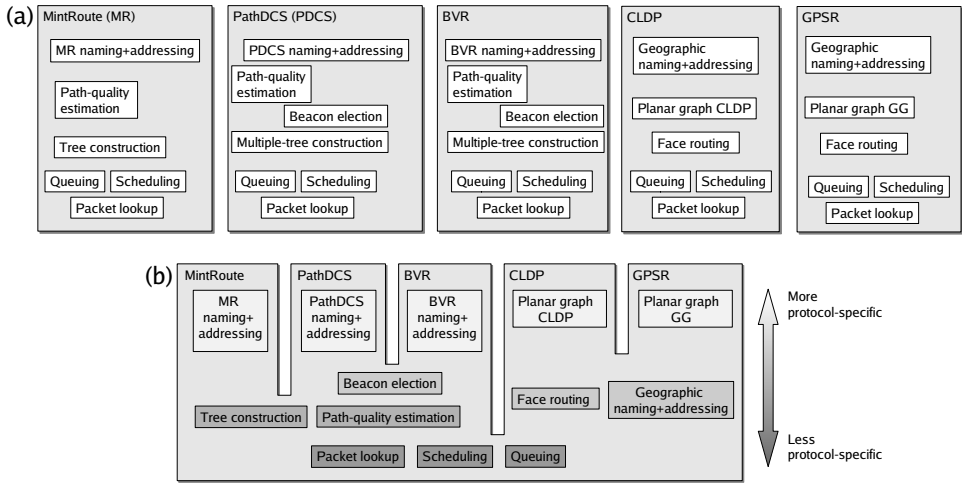

Figure 1: *Basic decomposition of five protocols. (a) Current situation: all functions are protocol-specific. (b) Common functions can be shared, reducing the total number of components in the system.*

it both easier to build new protocols and enable multiple protocols to share components during run-time.

In the following subsections, we address three questions that are central to defining a network layer:

- What are the services provided by the network layer? (§3.1)
- What are the components of the layer, and what functionality does each component implement? (§3.2)
- How do the components interact with each other, and what is the packet format? (§3.3 and §3.4)

3.1 The Network Layer Service

Similar to IP, the network layer in sensornets provides a best-effort, connectionless multihop communication abstraction to higher layers. However, unlike IP, the network layer in sensornets exposes different addressing and naming schemes, which are required to implement various communication abstractions. Figure 2 summarizes the services provided and functionalities implemented by the network layer.

To provide these services, the network layer needs to implement a variety of functions. These functions can be classified into two categories: control plane and data plane. Control plane functions include identifying and addressing nodes, as well as route discovery and maintenance. Data plane functions include packet forwarding, queue management, and packet scheduling.

In our design we assume that the network layer sits above the sensornet protocol (SP), the narrow waist of the sensornet protocol stack as proposed by Polastre et al. [21]. The lower part of Figure 2 summarizes the services SP provides to the network layer.

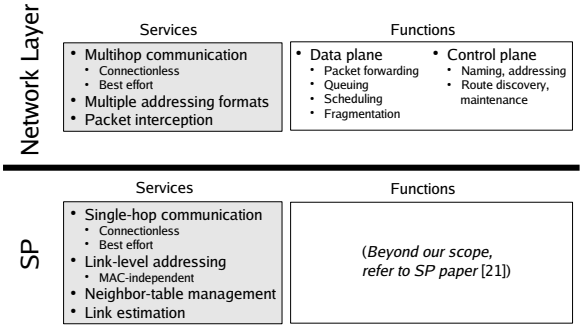

Figure 2: *Services provided by, and functionalities implemented in, the network layer, as well as services provided by SP.*

3.2 Network Layer Components

A key question in defining the network layer decomposition is the granularity we should achieve. We strove for one coarse enough so that closely related functions were grouped together (such as topology creation and maintenance), while still providing the flexibility to maximize code-reuse when implementing protocols. We began with a coarse-grained decomposition, and progressively split these components up in order to reach this desired granularity. Further, we attempted to create narrow and well-defined interfaces in order to avoid dependencies, allow for interchangeability of components and minimize composability constraints for new protocols.

At the first level, we follow the natural decomposition of the network layer into separate control and data plane components. Not surprisingly, this is similar to the way

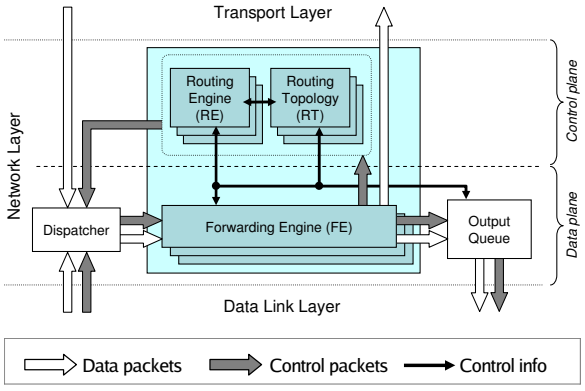

Figure 3: *The network layer decomposition, with the flow of packets and control information among the components.*

the software is structured in today's IP routers. However, this is too coarse-grained, as it enables little code reuse.

Of the two components, the control plane is typically far more complex than the data plane, as it needs to implement non-trivial functionalities such as topology discovery and routing. To facilitate further reuse, we split the control plane into two components: *routing engine* (RE), and *routing toplogy* (RT). RT is responsible for discovering and maintaining the network topologies, examples of which include trees, multi-trees, and meshes. Once the topology is created, RE computes and maintains routes over the topology. This decomposition allows one to reuse various routing protocols with different topologies. For instance, the data-centric routing protocol PathDCS [6] and the point-to-point protocol Beacon Vector Routing (BVR) [8] both use a multi-tree topology, which can therefore be reused.

While the data plane is in general simple (compared to the control plane), our second goal of achieving run-time sharing induces a natural decomposition of the data plane as well. Upon the arrival of a packet, the data plane needs to obtain the next hop(s) to forward the packet from the control plane (i.e., RE in our case). If multiple packets need to be forwarded at the same time, the packets have to be enqueued and scheduled appropriately. This suggests a decomposition of the data plane into two components: *forwarding engine* (FE) that obtains the next hop(s), and *output queue* (OQ), which implements buffer management and packet scheduling across different protocols.

Figure 3 shows our decomposition and the the interaction between components. We discuss the services and functionalities implemented by each of these components below, and provide examples in the next section[3].

Output Queue (OQ) The OQ module performs buffer management and packet scheduling across all packets forwarded by the node. Different queuing disciplines, as well as network-level transmission scheduling, can be implemented in the OQ. For co-existing protocols to use the communication resource fairly (as defined by the implemented policy), only one OQ module can be in use at any one time. This is in contrast to the FE, RE, and RT modules, multiple instances of which can operate simultaneously on the same node.

Forwarding Engine (FE) The main function of the FE is to obtain the next hop to which the packet is to be forwarded. In the case of multicast communication, multiple next hops will need to be obtained. This is achieved by having the FE query the corresponding RE based on the protocol used. Subsequently, the packet is sent to the OQ to be forwarded. The FE is agnostic to naming and addressing, to maximize module replaceability.

Other functions of an FE include local delivery when the RE determines that the local node is the destination, hooks for interception of packets for purposes of in-network aggregation, network level retransmission, and multicast. Finally, the FE may opt to perform buffer management and packet scheduling across packets belonging to the same network protocol. In contrast, the OQ operates on all packets traversing that node.

Routing Engine (RE) The RE provides naming and addressing services to the higher layers, and is the only component in the system that understands the protocol's address format. Functions implemented in the RE include (1) determining whether a packet should be forwarded, has reached its destination and thus be accepted, or dropped, (2) if the packet is to be forwarded, the next hop(s) given its destination. The RE implements the logic for determining routes given a destination, using information about an abstract representation of the network topology given by an RT module.

Routing Topology (RT) RT modules are responsible for creating and maintaining basic communication abstractions, with related routing information used by REs either to determine next hops or to construct more complex protocols. The RT is the module that will exchange control traffic with RTs in other nodes, for determining and maintaining the network topology. Examples of communication topologies are trees, geographic coordinates, or any node labeling allowing routes to be found.

We make the distinction between the RE and RT clearer with an example. One common communication topology is a tree, used for sending data to a basestation. There are several ways to build a tree: they can vary in characteristics with respect to stability, convergence, balance, or whether the tree is periodically maintained or is a one time construction. These would correspond to different RTs providing the same abstraction. An RE, on the other hand, performs the actual lookup process to de-

Figure 4: *Network packet header format.*

termine the next hop(s) to a destination, and is coupled to a topology class and not a particular RT.

3.3 Interfaces

We now provide high level descriptions of the network layer service interface as well as interfaces between the four major components.

Modules in the data path, namely FE and OQ, pass complete packets around and thus their corresponding interfaces are narrow. On the other hand, the FE/RE interface consists of two basic calls: one to obtain the next hop(s) for the packet given its destination, and the other to obtain the cost-to-destination from the current node[4].

The RE interacts with the RT to obtain the necessary information for determining routes. In defining the division between the RE and RT, the diversity in routing algorithms and communication abstractions in sensornets becomes apparent. Creating a unified interface between the two components would have increased code size, added complexity, and enabled untested and potentially unstable combinations of RTs and REs. Consequently, this interface will be somewhat more protocol specific, in the sense that each class of communication topologies will export a standard interface. On the other hand, a programmer can independently decide to split the RT component by adding a shim layer that would export a standard interface to the RE if he wants.

Finally, the components layered above interact with the network layer by specifying a protocol, an address, and providing a data packet to be sent. This service interface exposes the protocol-specific network address, which is subsequently interpreted by the RE.

3.4 Packet Header Format

Network protocols rely on node coordination to implement their services. This coordination requires interaction between similar components at different nodes, and is typically enabled using information carried in the packet headers. Thus, the packet header format ultimately dictates how components at different nodes interact with each other.

The main issue we are faced with when designing the format of the packet header is the portion of the header each component can access. Our decision is to associate a sub-header with each component involved in forwarding the packet, and allow a component to access only its own header with the rest being opaque. This way, we avoid unnecessary bindings, and make it easier to interchange different components. For example, the destination address is only understood by RE. This allows the same routing engine (for example, one that routes based on geographic coordinates), to be combined with different forwarding engines, such as opportunistic [1]. Except for the network protocol identifier, sub-headers may be absent depending on the corresponding components.

Figure 4 shows our packet header format. Since multiple network protocols can exist in a sensornet, we use a *protocol identifier* to select the appropriate components. This identifier is the only required field in the packet header. The rest of the header consists of three sub-headers, one for each component (OQ, FE and RE) involved in packet forwarding. The size and the format of each of these headers is component-specific, and we give a brief description of each below.

1. The OQ header contains scheduling and buffer management information (*e.g.,* packet priority) and is interpreted by the OQ module.

2. The FE header contains information required to forward the packet (*e.g.,* hopcount or a unique message identifier for suppressing packet duplicates).

3. The RE header holds information required to determine the next hop.

4 Module Examples

In order to demonstrate the feasibility of the modular network layer, we describe examples of individual components in this section, and show in the next section how they can be composed to implement several of currently available network protocols.

4.1 Output Queue Modules

An Output Queue (OQ) module is the one place in the system which all outgoing packets must traverse, a necessary condition to implement packet scheduling for all network protocols. Thus, while multiple types of OQs exist, only one can be in use in a node at any time. A basic module may implement simple priority scheduling, whereby control packets are given higher priority when queue drops occur and when selecting the next packet to send. More complex scheduling can be implemented to improve end-to-end fairness, as well as to reduce physical-layer contention[5] amongst neighboring nodes. Finally, scheduling of packet transmission can be influenced by the routing topology, useful in the case of in-network aggregation. We begin with the description of a basic, simple priority scheduler.

The *Basic* OQ provides simple priority scheduling and queue management functionality. Given a packet of high priority, the basic module transmits it before one of

lower priority. The determination of a packet's priority is dependent on the network protocol. For instance, some protocols may require their control packets to be sent as soon as possible and preferably not dropped when the queue is full, and may set these as high priority.

Flexible Power Scheduling (FPS) [9] is a network-level, time-division multiple access (TDMA) algorithm that aims to provide high utilization and fairness on a per-destination basis. FPS divides time into cycles, with a fixed number of slots in each cycle. In each cycle, FPS allocates slots on a per-destination basis. At the next-level, slots allocated for a particular destination D at node N are divided among the neighbors that forward packets to D through N. This allocation is based on the neighbors' queue occupancies: FPS allocates more slots to the neighbor with more packets destined to it. This policy aims to balance supply and demand at each neighbor, and at the same time achieve high utilization. Note that since FPS requires knowledge of the flow to which a packet belongs, interaction with the RE is necessary: classification of the packet is based on the destination address, the format of which is known only to the RE.

Epoch-based Proportional Selection (EPS) [5] is another example of an OQ. Unlike FPS which uses a static total number of slots per cycle, EPS dynamically adjusts this based on the current demand. EPS enforces fairness using non-work-conserving, weighted round-robin servicing of children's queues, using the number of upstream nodes of each child as the weights. This number is carried within each data packet transmitted in the OQ header. Similar to FPS, EPS requires classification of packets based on their network destination, thus interaction with the RE is necessary as well.

Finally, unlike the FPS and EPS components, the *Epoch* module's transmission schedule is determined by external components such as the routing topology. Using this knowledge, the Epoch module allows nodes further away from the destination to transmit before the rest, enabling aggregation of data at each intermediate hop towards the collection node. This reduces the total number of packets transmitted and thus energy consumed.

4.2 Forwarding Engines

Forwarding Engines (FEs) are components that are more protocol-specific in the data-plane, determining how and when packets are to be forwarded. Opportunistic forwarding and multicast are examples of FEs. We begin by describing a basic forwarding engine.

The *Basic* FE obtains per-packet next-hop information from the corresponding RE and checks for packet interception requests from higher layers. Additional functions include detection of routing loops and suppression of duplicate packets.

The *Opportunistic Forwarding* engine implements per-packet suppression functionality similar to ExOR's [1]. Packets eligible for forwarding include those received from neighbors further away from the destination. These packets are held onto for a pre-determined period of time; if no similar packets from nodes equally far or closer to the destination are received within this time, the packet is sent, otherwise its transmission is suppressed.

Note that the precise next-hop neighbor need not be identified. Instead, we require knowledge of the cost to destination, which can be provided by certain REs.

Multicast FEs forward multiple copies of the same packet to different next-hop nodes. These FEs only *provide* the functionality, they do not *decide* whether a packet should be multicast. This decision is made by the RE during packet lookups, when the RE implicitly indicates that multicast should take place by returning the list of all next-hops.

4.3 Routing Engines

A Routing Engine (RE) can build upon basic communication abstractions provided by Routing Topologies (RTs) to construct more complex ones. MintRoute, PathDCS and Beacon Vector Routing (BVR) are examples of such REs. On the other hand, simpler ones such as Broadcast can operate independent of RTs.

The *Broadcast* RE handles all packets that are logically broadcast to all one-hop neighbors. Thus, this simple RE does not provide any specific next-hop to which a packet should be forwarded, neither does it provide a cost-to-destination metric. It is a basic RE that can be used by all protocols, either at the transport or network layer, that require logical one-hop broadcasts.

The rest of the protocols, PathDCS, MintRoute and BVR, will be discussed in greater detail in Section 5. Briefly, PathDCS [6] provides data-centric routing capabilities using routing trees rooted at random nodes (beacons). Each piece of data is mapped onto a network path using the beacons as guides, and the destination node for that data is the terminating node of the path. BVR [8] on the other hand uses the same many-to-one routing abstraction to construct a logical coordinate system based on hop distances from the roots of these beacons. For simpler protocols such as MintRoute [24], the RE can be very light-weight since there is little additional function-

ality to be implemented. We note that in general REs are more specific to the network protocol than FEs or RTs.

4.4 Routing Topology

Communication abstractions can be composed from a few basic Routing Topologies (RTs), which REs can use to construct more complex ones. Interfaces provided by RTs can vary significantly with the abstractions provided. Examples of RTs include MTree, Gradient and Geographic, which we describe below.

The *MTree* RT provides many-to-one routing abstraction using trees. The primary metric used in tree construction is the minimization of expected packet transmissions to the root. Periodic control information exchanged between neighbors determine the bidirectional, one-hop as well as end-to-end path quality. MTree constructs M routing trees rooted at random nodes in the network, except for the first which is at the base, or collection, station. Thus, this module can be also used for basic route-to-base applications.

Since MTree maintains routing information in the form of routing tables, it can provide hop-distances to each root. This information can be used by BVR to construct a logical coordinate system on which point-to-point routing can be implemented, by a basic FE to detect loops, or by PathDCS to determine the maximum number of hops to take towards a certain beacon (root). Next-hop information can be used by FPS to determine the parent from which supply slots can be requested.

The *Gradient* topology is similar to MTree in that each node maintains its cost-to-destination. However, this module does not specifically determine the next-hop to which a packet should be forwarded. Such a topology is simple to construct and maintain, and is useful for purposes of scheduling and opportunistic forwarding. The Epoch module, focusing on scheduling, is advantageous for in-network aggregation of data as mentioned earlier. Opportunistic Forwarding, which requires just the cost-to-destination and not the specific next-hop, can use the function provided by Gradient as well.

Finally, the *Geographic* RT provides geographic coordinates via, for example, the Global Positioning System. This module can be used by multiple other components: by a GPSR-like RE to provide point-to-point routing, or by the application to determine, say, the location of a fire. The Geographic RT can be augmented with more functions. For instance, it can periodically probe neighboring nodes to obtain their coordinates, enabling it to provide information such as the closest next-hop node towards a given destination.

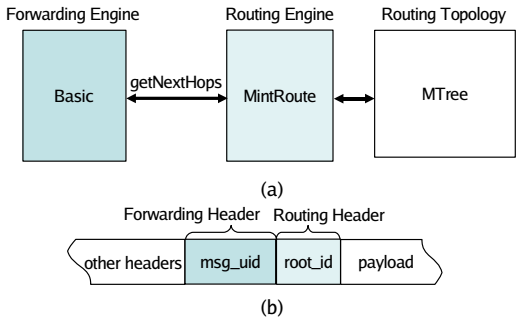

Figure 5: *(a) Main modules in the implementation of MintRoute. (b) Packet header contents.*

Also, Euclidean distance to destination can be provided as the cost metric.

5 Composition of Protocols

We now describe how various existing protocols can be composed from the modules providing services and implementing functions required of the network layer. Table 1 provides a summary of several protocols and their corresponding components. Since there are modules that, by their very nature, cannot interoperate efficiently or at all due to their inability to provide required functions, there exist constraints that restrict possible module combinations. We elaborate on this at the end of this section.

5.1 Collection

MintRoute [24] is a collection protocol that routes packets towards the root of a tree, and is the basis for many data gathering applications. A message is recursively forwarded to the current node's parent until the root is reached. Figure 5 shows the main components that make up MintRoute. We use Basic FE that provides routing loop detection ability, MTree as the RT, and a MintRoute-specific RE.

Although multiple routing trees are provided by MTree, MintRoute uses that rooted at the collection point. Similar to the monolithic version of MintRoute, the destination address is therefore implicit and need not be included in the RE header. To remove duplicate packets due to retransmissions, the FE uses unique identifiers placed in the FE header. Finally, the OQ provides simple priority scheduling, and can give higher priority for control traffic from MTree than for data traffic.

A variant of collection protocols involves in-network aggregation of data. Synopsis Diffusion (SD) [20] is one such protocol, providing duplicate-insensitive aggregation in sensor networks. A gradient is set up originating from the destination node, with nodes further away sending packets earlier so that those closer can aggregate before they forward. This reduces the total number

Table 1: Decomposition of current and composition of new network protocols. Implemented modules in *italics*.

Network Protocol	Output Queue	Forwarding	Routing	Topology
		Existing Protocols		
MintRoute	*FPS/Epoch/Basic*	*Basic*	-	*MTree*
GPSR	*Basic*	*Basic*	GPSR	Geographic Coords + GG/RNG Planarization
CLDP	*Basic*	*Basic*	GPSR	Geographic Coords + CLDP Planarization
PathDCS	*Basic*	*Basic*	*PathDCS*	*MTree*
BVR	*Basic*	*Basic/Re-xmit*[6]	*BVR RE*	*MTree*
Synopsis Diffusion[7]	*Epoch*	*Basic*	-	*Gradient*
Directed Diffusion	*Basic*	Multicast	Directed Diffusion	-
AODV	*Basic*	*Basic*	On-demand point-to-point	-
GRAd	*Basic*	*Opportunistic*	On-demand point-to-point	-
ExOR	*Basic*	Opportunistic[8]	OSPF	-
Trickle[9]	*Basic*	*Basic*	*Broadcast*	-
		New, Hybrid Protocols		
Opp. MintRoute	*FPS/Epoch/Basic*	*Opportunistic*	-	*MTree/Gradient*
Alternate Paths	*Epoch/Basic*	*Basic/Re-xmit*	-	*MTree*
Scoped Trickle	*Basic*	*Basic*	*Scoped Broadcast*	-

of transmissions by each node. The natural RT to use is thus Gradient, and the scheduling mechanism can be provided by the Epoch OQ with information from Gradient. At intermediate hops, data packets have to be intercepted by the FE and sent up the stack for aggregation. Thereafter, the packet is scheduled for transmission in the OQ module.

5.2 Point-to-Point

We next describe two classes of point-to-point network protocols in sensornets, based on either logical or actual geographic coordinates. We begin with the latter. There is a large number of variations on Geographic Routing [2, 12, 16, 14], and we present the basic idea here. Each node maintains knowledge of its coordinates as well as those of its neighbors'. Next-hop(s) to which a packet is forwarded is(are) determined using the destination's coordinates carried within the RE header.

Two routing phases exist, *greedy* and *face routing*. In greedy routing, nodes forward the packet to the neighbor closest to the destination. If the current node is closest compared to all its neighbors, the forwarding node switches to the next phase: face routing. Packets are then forwarded along the face edges of an underlying planar graph, changing faces when appropriate and applying rules that guarantee progress towards the destination. In this phase, additional state has to be carried in the packet[10]: the current phase, the node's coordinates when face routing begun (L_p), as well as the coordinates (L_f) and the edge (e_0) where the packet entered the current face. At each step the node checks if greedy can resume and does so if possible.

Figure 6 shows the modules implementing geographic routing and the contents of their corresponding headers. The routing topology components provide two abstractions: (1) Geographic Coordinates, which maintains and provides the coordinates of the current node and its neighbors, and (2) Planar Graph, which provides a planarized version of the underlying connectivity graph. The Planar Graph functionality can, for instance, be provided by CLDP, Gabriel Graph, or Relative Neighborhood Graph. The RE is responsible for determining the next-hop to the destination, and maintains state in the RE header. By replacing the face routing rule, the RE can implement GPSR or GOAFR routing, reusing the RT.

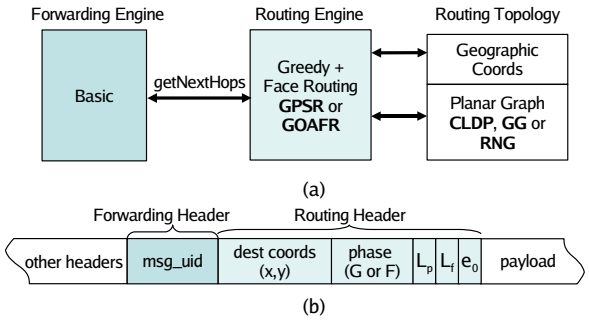

Figure 6: *(a) Main modules in the implementation of Geographic Forwarding. Notice that there can be many variations (GPSR, CLDP, GOAFR, GOAFR+CLDP), with reuse of many modules. (b) Packet header contents. The routing header includes state for face routing.*

In contrast with Geographic Routing, Beacon Vector Routing [8] (BVR) uses a topology-based logical coordinate system to find routes. A subset of the nodes is chosen as *beacons*, and all nodes in the network learn their distances in hops to each of these beacons. Routing is performed in a greedy fashion by minimizing a

distance function on the coordinates. Next-hop information can be determined using the destination's, current node's and neighbors' coordinates. BVR's RE consists of a simple module that derives coordinates from MTree and computes the distance function between two coordinates. Routing is performed in three phases. First, greedy routing is attempted. If that fails, packets are routed towards the root closest to the destination. Upon reaching that root, scoping flooding is performed. The routing header consists of the destination's coordinates and identifier, the minimum cost the packet has seen so far, the current routing phase, and the time-to-live for scoped flooding. The FE used by the original BVR tries to send messages to alternate next-hops upon transmission failure. The BVR RE can also be paired with other forwarding disciplines, such as opportunistic, since the cost-to-destination can be obtained from the virtual coordinates created.

5.3 Data-Centric

PathDCS [6] is an example of a data-centric routing protocol. Rather than map data onto specific locations in a geographic or logical coordinate system, PathDCS maps it onto a path through the network. This path is divided into parts, or segments, which is in turn defined by a certain number of hops to take towards a particular network beacon. The storage location for the data is then the terminating point of the concatenated segments. Mapping data storage location onto terminating nodes of existing paths ensures that a node always exists at that location. This eliminates the need to know the network boundaries, a requirement necessary for coordinate-based systems.

The PathDCS network protocol can be decomposed into a protocol-specific RE that runs atop the MTree topology. Its RE header contains the data key, the current segment being traversed, and the current number of hops to traverse towards the next beacon. PathDCS can use other kinds of FEs, such as Opportunistic, and is compatible with OQ modules providing scheduling functionalities.

Directed Diffusion (DD) [11] is another example of a data-centric routing protocol. DD names data using attribute-value pairs, and interests for data are disseminated through the network. The dissemination process sets up gradients allowing nodes with the relevant data to forward them to the querying nodes. In Directed Diffusion, multiple overlapping interests can be aggregated, enabling data to be sent once from the source to the aggregation point before being duplicated for forwarding towards each interest source. Thus, DD is primarily composed of an RE that sets up gradients from each interest source, as well as a multicast FE. In this case, no routing topology module is required.

5.4 Dissemination

Trickle [18] is a code dissemination and maintenance protocol. Since the objective of Trickle is to have all nodes in the network run the same code, it can be considered a transport layer protocol that implements one-to-all reliable transfer of data. To support Trickle, the network layer provides a simple one-hop broadcast RE. A received, logically broadcast packet is passed to the Trickle transport layer. Trickle subsequently uses delayed transmission and suppression to control the rate at which messages are broadcast in the network. Although functionally similar to opportunistic forwarding, Trickle's suppression mechanism requires transport layer knowledge, and therefore is not placed in the FE.

5.5 New, Hybrid Protocols

The network layer modularity simplifies the creation of new protocols by swapping one component for another, or making slight modifications (Table 1). For instance, in the case of the collection protocol MintRoute, we can route to any of the roots provided by MTree by including the destination root address in RE's header (Figure 5b).

Network-level retransmissions can also be added by having the RE return all next-hops closer to the destination. An FE providing the retransmission function can attempt to resend packets to alternate next-hops. Such an RE can also be used by an FE that probabilistically selects a next hop to balance forwarding load. Lastly, replacement of the basic FE with the Opportunistic FE yields opportunistic collection routing.

A different kind of collection protocol can be implemented using the Gradient RT and the Opportunistic FE. The former readily provides knowledge of the current node's distance to destination, required by the latter. One can imagine various performance benefits to be gained from such a pairing, but the evaluation is beyond the scope of this paper.

5.6 Composability Constraints

We end this section by looking at the constraints encountered when attempting to combine different routing, forwarding, and output queue modules. In the following discussion we use the term routing to mean the combination of RE and RT. Table 2 lists the combinations that are feasible and those that aren't. We see that the basic OQ works with all protocols, as does the basic FE. Network-level retransmission is a functionality embedded in the FE, but is effective only if routing provides multiple distinct next-hops. From column *Re-Xmit*, this is not the case for Table Driven and Broadcast routing. Opportunistic forwarding, in turn, requires a globally meaningful cost-to-destination metric. PathDCS, MintRoute, Gradient, Geographic, and BVR routing pro-

Table 2: Constraints on the composition of routing engine/topologies with forwarding engines and output queuing. The composition may be good($\surd$), not optimal (*), or not possible (-).

	Forwarding Engine				Output Queuing		
Routing	Basic[11]	Re-xmit[12]	Opport.[13]	Multicast	Basic[14]	Epoch	FPS
MintRoute	$\surd$	$\surd$	$\surd$	-	$\surd$	$\surd$	$\surd$
Gradient	$\surd$	$\surd$	$\surd$	-	$\surd$	$\surd$	$\surd$
Table Driven	$\surd$	-	-	-	$\surd$	*	*
PathDCS	$\surd$	$\surd$	*	-	$\surd$	$\surd$	$\surd$
BVR	$\surd$	$\surd$	$\surd$[15]	-	$\surd$	*	*
Geographic	$\surd$	$\surd$	$\surd$	-	$\surd$	*	*
Broadcast	$\surd$	-	-	-	$\surd$	*	*
Directed Diffusion	-	-	-	$\surd$	$\surd$	*	*

vide such a cost field for each destination, although for the last two there are local minima, and routing solely based on these may not lead to the destination. Finally, while both Epoch and FPS are currently designed for collection routing, and thus will work optimally only with MintRoute and Gradient routing, they can be extended for the case of multiple destinations.

6 Completing The Picture

The previous sections discussed details of the major parts of the network layer, but at the high level. In this section we discuss the rest of the components and bring everything together by providing a short description of the actual packet forwarding process.

6.1 Miscellaneous Components

Four additional components, the Protocol Service, Network Service Manager, Buffer Manager, and the Dispatcher, are minor components but are still essential to the network layer. We describe them below.

Protocol Service As described in Section 3, we decomposed each network protocol into a Forwarding Engine, Routing Engine, and possibly a Routing Topology. To simplify the usage of these components, we wrap them in a Protocol Service module, an instance of which exists for each different network protocol. This module provides a unified service interface to higher-layer programmers and specifies the necessary connections between the wrapped components.

Network Service Manager The Network Service Manager aggregates and maintains information related to service requests originating from higher-layer components. For instance, applications can register with the service manager hooks to intercept certain protocols' packets. The service manager is then queried by the forwarding engines to determine if interception is necessary. Since there may be multiple applications requiring such services, and since these functionalities are required of all protocols, we gather them into a single module.

Buffer Manager The traditional way of managing buffers, that is, RAM space required to hold packets received or due for transmission, is to have each component statically allocate space at compile time. Sharing of buffers between different components is not possible, thus in general the system is less able to accommodate bursts of buffer requests and usage of available buffers are less efficient. The Buffer Manager tackles this issue by aggregating available buffers from all components at node initialization time, and provides them on request based on some user-defined policies. Since this module works across layers, it is not part of the network layer, but instead simply provides buffer aggregation services. Thus, in-depth design and evaluation of this module is beyond the scope of this paper.

Dispatcher The dispatching module maintains a protocol table with an entry for each network protocol running on the node. Upon receiving a packet, the packet's protocol identifier is used to determine the corresponding forwarding engine which the packet should be sent. This protocol table will generally be configured at compile time and be uniform across a sensor network.

6.2 Packet Forwarding Procedure

To show how various parts of the system fit, we next describe the forwarding process undertaken upon reception of a packet using Figure 7.

1. Messages first arrive at the dispatcher either locally or from the network. The dispatcher determines the protocol identifier, either from the higher-layer component if local, or otherwise from the message itself.

2. The message is subsequently sent to the corresponding FE based on the identifier.

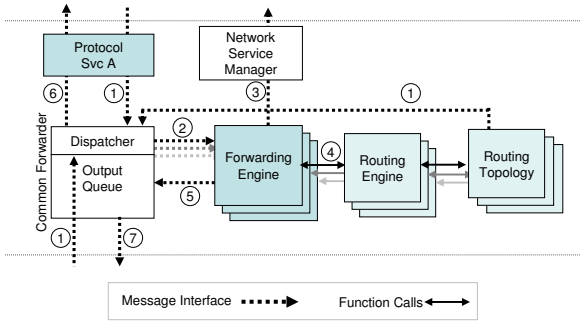

Figure 7: *Data path in the network layer.*

3. The FE checks whether an application has registered to intercept messages of this protocol, if so the message is handed to the application, otherwise

4. the RE is queried to determine the specific next hop(s) for the message, or provide some cost value to determine if the current node is closer to the message destination. Since the RE is the only component that understands the address format, it can attach the identifier of the flow to which the message belongs for purposes of scheduling later in the OQ.

5. FE then sends the message to the OQ specifying whether it should be forwarded or sent to a higher-layer component.

6. At the OQ, if the message is determined by the RE to be destined for this node, or by the Network Service Manager to be intercepted, it is sent up the stack, otherwise

7. the message is to be forwarded, and is scheduled for transmission based on the implemented policy.

7 Evaluation

In this section we evaluate a subset of the protocols implemented with the proposed network layer. We revisit our goals of code reuse and run-time sharing of components by multiple protocols, with a minimal penality at storage and performance, and show that they are largely achieved.

7.1 Methodology

Our two basic metrics in the evaluation are code size and memory footprint on the one hand, and forwarding delay on the other. We require only the addition of one byte for the protocol identifier and not the transmission of additional packets. This, coupled with the fact that energy consumption is dominated by communication rather than computation in today's sensornets, lead us to believe that this consumption should not increase significantly and thus we do not include it as a metric.

To quantify real benefits and costs, we compare the implementations of just three protocols (due to space constraints), PathDCS, BVR, MintRoute. These implementations include the original, monolithic version (Original), implementation over SP without the network layer modularization (SP),[20] and our modularized version running on the network layer, on top of SP (NLA & SP). All evaluations are performed on TelosB motes [25].

To measure the forwarding cost, defined to be the delay encountered by a packet when it first enters the network layer until it is sent to the link layer for transmission, we instrumented the code to read a hardware clock with the granularity of microseconds, properly accounting for the (small) measurement overhead.

7.2 Code size and memory footprint

One of the main objectives of creating a modular network layer for sensor networks is to increase code reuse, thereby allowing for multiple protocols to coexist cleanly and efficiently on a single node and simplifying future network protocol design. There is some overhead, however, to allow demultiplexing and configuration among different modules, when compared to the monolithic implementations. Table 3 compares code and memory sizes between the different implementations. Code size refers to the amount of program memory occupied by the networking code, while memory footprint refers to the amount of RAM allocated. The former is important since it is desirable that non-application code not consume significant amounts of limited memory. Volatile memory, however, is more crucial, since there is a direct relationship between RAM and energy consumption.

Three observations can be made from Table 3:

1. When we combine different protocols we observe clear gains: the two combinations use up 40% and 58% less memory, and occupy 18% and 37% less program memory. Of course, the gains are limited by the extent to which protocols share underlying primitives, but we note that our network layer realizes these gains, and that we have seen considerable potential sharing among existing and new protocols (c.f. Table 1).

2. For the individual protocols (upper section), code size and memory size are comparable. As we will show next, much of the difference in code size resulted from additional functionalities. The memory consumption numbers are similar, which is good from an energy-conservation perspective.

3. The last observation relates to code reuse at development time. We have decomposed network protocols such that protocol-specific code is limited to a

Table 3: Code and Memory Size Comparison of Architectures

Network Protocol[16]	Code Size[17]			Memory Footprint[18]		
	Mono	SP	NLA & SP	Mono	SP	NLA & SP
MintRoute	2562	2356	6140(92)	1400	1273	1862
PathDCS	6786	4968	6450(988)	1764	1981	1766
BVR	6422[19]	-	9060(1512)	1411	-	1889
Synopsis Diffusion	-	-	3686	-	-	872
Protocol Coexistence						
MintRoute + PathDCS	9348	-	7684	3164	-	1894
MintRoute + PathDCS + BVR	16430	-	10354	4575	-	1917

single component in most cases, typically the routing engine. The numbers in parenthesis in Table 3 show the code sizes that are protocol-specific and not likely to be reusable. Protocol-specific code is substantially less in our implementation[21]. For the Original and SP versions, these numbers are essentially the same: most of the code is not reusable as a result of tight integration, or unusability of modules by other designers due to the lack of common interfaces or unclear division of functionalities implemented in modules. In the case of the SP implementations, neighbor-table management, and link estimation are moved into SP itself, but the remaining network layer code is again protocol-specific, hindering substantial code reuse.

We now return to the increase in code size observed for BVR and MintRoute, by examining, with reference to Table 4, how the Original and NLA implementations of BVR are decomposed. The decomposition of MintRoute presents similar trends and is omitted for brevity. The modules are grouped by approximate functionality. It is comforting that the code implementing primary BVR functions (Core Protocol Code) are similar in size. The communication stack on which each implementation sits has a queue and a link estimator in the original version. In addition, SP provides neighbor-table management, simple one-hop scheduling, and duty cycling[22] of the radio. These additional features account for SP's larger code size. The last group, Additional Features, are unique to our implementation, and account for most of the difference in total code size. These include dynamic demultiplexing, tools for maintaining header independence, multiple queues for buffers[23], and dynamic memory management. We note that not all of these extra features are a requirement of the network layer, and that in most cases one can implement simpler and smaller modules.

Finally, we look at each individual component, drawn from a general library which developers can use to assist in creating new protocols. Table 5 provides the code size and memory footprint for these components. In general, we consider routing engines to be the heart of network protocols, and are thus less likely to be reused. On the other hand, components such as the MTree routing topology are much more general and can consequently be utilized by a wider variety of protocols. These generic components are the key factor in enabling clean co-existence of multiple protocols with a substantially smaller overall code size and memory footprint relative to their monolithic counterparts.

Table 4: Detailed Comparison of BVR Implementations

Monolithic		NLA & SP	
Component	Size(B)	Component	Size(B)
Core Protocol Code			
Router	1194	Routing Engine	1512
Topology State	2638	Routing Topology[24]	2136
Coordinate Table	1422	Coordinate Table	1422
Coordinate Functions	754	Coordinate Functions	754
		Forwarding Engine	504
	6008		**6328**
Underlying Communication Stack			
Queuing Buffer	394	SP	4244
Link Estimator	2856		
	3250		**4244**
Additional Functionalities			
		Service Manager	490
		Output Queue	1822
		NetService	324
		BufferManagerM	252
			2888

7.3 Performance

Next we evaluate the modular network layer from a performance point of view. Table 6 provides comparisons of forwarding delay, per module, for the original and NLA & SP implementations of MintRoute, PathDCS and BVR.

As one would expect, packets traversing our network layer experience more delay than they do in the monolithic architecture. The additional delays are incurred due to component and layering abstractions: in the monolithic architecture, it is assumed that only one link layer exists at each node. It is thus possible to have the packet header format known to all components in the system. On the other hand, to improve portability and reuse, the network layer uses the *payload* and *length* meta-data fields to indicate the location of the next payload for the next component receiving the packet. This reduces the

Table 5: Code Size and Memory Footprint of Individual Components

Component	Code Size (B)	Memory (B)
Output Queues		
Basic	1822	396
Epoch	1892	396
Flexible Power Sched.	2696	564
Forwarding Engines		
Basic	384	64
Opportunistic	1830	169
Retransmit	504	64
Routing Engines		
Broadcast	42	0
MintRoute	92	0
PathDCS	988	2
BVR	1512	0
Routing Topologies		
MTree	2766	88
Gradient	372	82
BufferManager	252	84
Network Protocol Service	324	24
Service Manager	490	8

Table 6: Forwarding Cost Comparison

	Mono	NLA & SP	
Protocol	Time(μs)	Component	Time(μs)
MintRoute	**65**	Routing Engine:	19
		Routing Topology:	24
		NetService:	12
		Output Queue:	573
		Forwarding Engine:	178
			806
PathDCS	**181**	Routing Engine:	165
		Routing Topology:	24
		NetService:	12
		Output Queue:	573
		Forwarding Engine:	178
			952
BVR	**3752**	Routing Engine:	366
		Routing Topology:	2795
		NetService:	12
		Output Queue:	573
		Forwarding Engine:	178
			3924

need to know, say, every possible MAC header in existence. The tradeoff is the necessity of additional operations required to access these fields. Furthermore, additional overhead due to function calls is incurred since the network layer is composed of multiple small modules instead of just one that is tightly integrated.

We observe a fixed overhead cost induced by NetService, Output Queue, and Forwarding Engine components that is significant compared to the cost of the simpler protocols. For BVR, which involves more complex lookup processing, the relative overhead is considerably smaller, representing a 4.3% increase in forwarding time. These numbers must be placed in perspective: although performance is important, unlike the Internet, it is not the primary goal. Most applications we see are very *low data rate*, *low duty-cycle*, and the cost in terms of energy of processing a byte is low compared to the cost of sending a byte over the radio. On the same platform, it takes at least $6.25ms$ to forward a common packet of about 40 bytes[25], and thus even with BVR we are still operating under "line speed". In none of these examples will packet processing be a bottleneck. Lastly, the components can be further optimized to reduce processing time.

8 Conclusion

In this paper we proposed, implemented, and evaluated a modular network layer for sensornets that aims at maximizing composability and reusability of protocol modules. We verified that many existing protocols fit naturally in the architecture, and that less effort is required to create new ones. Through evaluation of modules and protocols implemented in the new network layer, we are able to obtain up to 58% memory reduction and 37% less code when running protocols concurrently. Furthermore, we believe that the additional processing latency incurred is acceptable in the context of sensornets.

Looking forward, the work towards a sensornet architecture is still incomplete. One consequence of establishing more strict layering is that certain functionalities, like power management, security, reliability, and time synchronization, need to be accessible to multiple layers. Our hope is to address these cross-layer issues in the future.

Notes

[1]As evidence of that, the original implementations of four routing algorithms — MintRoute, PathDCS, BVR, and CLDP — have four different and incompatible implementations of common modules such as link estimation, neighborhood and queue management.

[2] Note that our goal is to facilitate the implementation of network-layer solutions, it does not actually, say, make end-to-end transfer of data more reliable. As such, we do not focus on implementing new functionalities.

[3]Minor components, like the dispatcher, are discussed later in § 6.

[4]This can be used by protocol implementing some form of opportunistic forwarding.

[5]in the case of shared mediums

[6]Network retransmission to alternate next-hops.

[7]Higher layer computes synopsis.

[8]The complete ExOR algorithm is not implemented.

[9]This is just the network layer support for Trickle. The main algorithm runs in the transport layer.

[10]See [12] for details.

[11]Includes variations, like duplicate detection and max TTL.

[12]Requires alternate next-hops.

[13]Requires global cost-to-destination function.

[14]Includes variations like priority, round-robin, fair queuing

[15]Does not guarantee delivery due to local minima.

[16]A dash '-' indicates that no implementation existed in that particular format.

[17]In bytes. This code size includes only network layer code, thus the code size of SP, approximately 4200 bytes, is not included in the figures for the SP and NLA & SP implementations.

[18]In bytes. In order to provide a fair comparison, the memory footprint figures for *SP* and *NLA & SP* implementations include the neighbor table overhead portion of the SP memory footprint, as monolithic implementations maintain their own neighbor table.

[19]This figure does not include the code for link estimation, as that feature is not currently provided in our implementation. With that capability included, the code size is 9278 bytes

[20]The 'MintRoute on SP' combination is the only one in which the original SP implementation [21] is used. All other experiments were run using an enhanced version [23], which has several additional features, and thus, larger code size.

[21]We do not have a figure for Synopsis Diffusion because all of the protocol-specific functionality for duplicate-insensitive aggregation is at the application level. The network layer components are all reusable.

[22]Duty-cycling refers to the turning off of the radio from time to time to conserve energy, which can be consumed even when listening to the channel.

[23]Including queues for different message priorities.

[24]The RT in this comparison is not MTree, but an equivalent RT ported directly from the original BVR code, to make the comparison closer.

[25]The usual size in a sensornet.

References

[1] BISWAS, S., AND MORRIS, R. Exor: opportunistic multi-hop routing for wireless networks. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications* (New York, NY, USA, 2005), ACM Press, pp. 133–144.

[2] BOSE, P., MORING, P., STOJMENOVIC, I., AND URRUTIA, J. Routing with guaranteed delivery in ad-hoc wireless networks. In *ACM Wireless Networks* (November 2001).

[3] CONDIE, T., HELLERSTEIN, J. M., MANIATIS, P., RHEA, S., AND ROSCOE, T. Finally, a use for componentized transport protocols. In *Proceedings of the Fourth Workshop on Hot Topics in Networks* (November 2005).

[4] CULLER, D., DUTTA, P., EE, C. T., FONSECA, R., HUI, J., LEVIS, P., POLASTRE, J., SHENKER, S., STOICA, I., TOLLE, G., AND ZHAO, J. Towards a sensor network architecture: Lowering the waistline.

[5] EE, C. T., AND BAJCSY, R. Congestion control and fairness for many-to-one routing in sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems* (New York, NY, USA, 2004), ACM Press, pp. 148–161.

[6] EE, C. T., RATNASAMY, S., AND SHENKER, S. Practical data-centric storage. In *Proceedings of the Third USENIX/ACM NSDI* (San Jose, MA, May 2006).

[7] ESTRIN, D., GOVINDAN, R., HEIDEMANN, J. S., AND KUMAR, S. Next century challenges: Scalable coordination in sensor networks. In *Mobile Computing and Networking* (1999), pp. 263–270.

[8] FONSECA, R., RATNASAMY, S., ZHAO, J., EE, C.-T., CULLER, D., SHENKER, S., AND STOICA, I. Beacon-Vector Routing: Scalable Point-to-point Routing in Wireless Sensor Networks. In *Proceedings of the Second USENIX/ACM NSDI* (Boston, MA, May 2005).

[9] HOHLT, B., DOHERTY, L., AND BREWER, E. Flexible power scheduling for sensor networks. In *IPSN'04: Proceedings of the third international symposium on Information processing in sensor networks* (New York, NY, USA, 2004), ACM Press, pp. 205–214.

[10] HUTCHINSON, N. C., AND PETERSON, L. L. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering 17*, 1 (1991), 64–76.

[11] INTANAGONWIWAT, C., GOVINDAN, R., AND ESTRIN, D. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proceedings of the International Conference on Mobile Computing and Networking* (Aug. 2000).

[12] KARP, B., AND KUNG, H. T. GPSR: greedy perimeter stateless routing for wireless networks. In *International Conference on Mobile Computing and Networking (MobiCom 2000)* (Boston, MA, USA, 2000), pp. 243–254.

[13] KICZALES, G., LAMPING, J., MENHDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds., vol. 1241. Springer-Verlag, Berlin, Heidelberg, and New York, 1997, pp. 220–242.

[14] KIM, Y., GOVINDAN, R., KARP, B., AND SHENKER, S. Geographic routing made practical. In *Proceedings of the Second USENIX/ACM Symposium on Networked System Design and Implementation (NSDI 2005)* (Boston, MA, May 2005).

[15] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click modular router. *ACM Transactions on Computer Systems 18*, 3 (August 2000), 263–297.

[16] KUHN, F., WATTENHOFER, R., ZHANG, Y., AND ZOLLINGER, A. Geometric ad-hoc routing: Of theory and practice. In *Principles of Distributed Computing* (2003).

[17] LEVIS, P., AND CULLER, D. Maté: a tiny virtual machine for sensor networks. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)* (Oct. 2002).

[18] LEVIS, P., PATEL, N., CULLER, D., AND SHENKER, S. Trickle: A self-regulating algorithm for code maintenance and propagation in wireless sensor networks. In *First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)* (2004).

[19] LOO, B. T., CONDIE, T., HELLERSTEIN, J. M., MANIATIS, P., ROSCOE, T., AND STOICA, I. Implementing declarative overlays. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles* (New York, NY, USA, 2005), ACM Press, pp. 75–90.

[20] NATH, S., GIBBONS, P. B., SESHAN, S., AND ANDERSON, Z. R. Synopsis diffusion for robust aggregation in sensor networks. In *Proceedings of the 2nd ACM Conference on Embedded Networked Sensor Systems (Sensys)* (New York, NY, USA, 2004), ACM Press, pp. 250–262.

[21] POLASTRE, J., HUI, J., LEVIS, P., ZHAO, J., CULLER, D., SHENKER, S., AND STOICA, I. A unifying link abstraction for wireless sensor networks. In *Proceedings of the Third ACM Conference on Embedded Networked Sensory Systems (SenSys)* (Nov. 2005).

[22] RODRIGUEZ, A., KILLIAN, C., BHAT, S., KOSTIC, D., AND VAHDAT, A. Macedon: Methodology for automatically creating, evaluating, and designing overlay networks. In *Proceedings of the first USENIX/ACM Symposium on Networked System Design and Implementation (NSDI 2004)* (San Francisco, CA, March 2004), pp. 267–280.

[23] TAVAKOLI, A., TANEJA, J., DUTTA, P., CULLER, D., SHENKER, S., AND STOICA, I. Evaluation and Enhancement of a Unifying Link Abstraction for Sensornets. Under submission.

[24] WOO, A., TONG, T., AND CULLER, D. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of the first international conference on Embedded networked sensor systems* (2003), ACM Press, pp. 14–27.

[25] Crossbow. `http://www.crossbow.com`.