UCLA
Technical Reports

Title
Mote Herding for Tiered Wireless Sensor Networks

Permalink
https://escholarship.org/uc/item/9nv096c4

Authors
Thanos Stathopoulos
Lewis Girod
John Heidemann
et al.

Publication Date
2005

eScholarship.org Powered by the California Digital Library
University of California

Mote Herding for Tiered Wireless Sensor Networks

Thanos Stathopoulos[†] Lewis Girod[†] John Heidemann[‡] Deborah Estrin[†]

Center for Embedded Networked Sensing
[†] UCLA, Department of Computer Science
[‡] USC, Information Sciences Institute
{*thanos@cs.ucla.edu, girod@lecs.cs.ucla.edu, johnh@isi.edu, destrin@cs.ucla.edu*}

Abstract

We propose *Mote Herding*, a new system architecture for large scale, heterogeneous sensor networks. Mote herding uses a mix of many 8-bit sensor nodes (motes) and fewer but more powerful 32-bit sensor nodes (microservers). Mote herding groups motes into *flocks* that are connected via a multihop network to a microserver acting as a *shepherd*. Shepherds exploit their greater communications and compute power to form an overlay network, with many flocks joining to form a *herd*. By keeping each flock small and utilizing several shepherds, the herd can support many nodes with better latency, reliability, and energy efficiency than homogeneous architectures. Using the Mote Herding abstractions, we have implemented a set of services that run across both platforms, namely a mote routing service, a data reliability service and a resource discovery service that is based on three subservices. We evaluate the performance of our services using simulations and emulations of both sample scenarios and a habitat monitoring application. Our results show that the mote routing service is able to maintain better than than 99% connectivity at high network densities, while incurring 60% lower transmission overhead than two other routing protocols. We also show that the data reliability service is able to deliver 100% of the data even when more than 30% of the network exhibits failures. Finally, we show that the habitat monitoring application that uses Mote Herding is able to deliver all of the data with low control overhead and latency.

1 Introduction

The past year has seen the realization of several successful sensor network deployments [26, 19, 2, 23]. This deployment experience is moving the field into a more mature state, with small and medium-sized networks running sampling applications. Pushing sensor networks beyond simple "sample and return" applications will introduce new engineering challenges, particularly regarding scaling to larger numbers of nodes, more and more distant communication, and use of more sophisticated sensor processing algorithms inside the network.

Currently, wireless sensor network devices fall into two main categories. The first category includes tiny, inexpensive, resource-constrained nodes that operate for long periods of time on battery power. The most widely used device in this class is the Crossbow Mote [14] based on the 8-bit AVR microcontroller. The second category includes small 32-bit nodes that host sophisticated peripherals and megabytes of RAM and flash and typically either run duty-cycled or connected to large energy sources. The XScale-based Intel Stargate [16] platform is a popular representative of this category. The type of device used in a sensor network is paramount in defining the capabilities and limitations of the entire system.

However, neither class is suitable for *all* applications. In general, low-cost platforms are ideal for applications that can operate with that level of functionality, and suited for highly energy-optimized, vigilant applications. But there are also applications that require more computation or storage and can justify the additional costs [4]. Moreover, many applications will require a mixture of both in order to combine densely deployed lower-end sensors with more sparsely deployed higher-end nodes [2]. It is for those reasons that the sensor network community has been increasingly exploring *tiered architectures* where the system is composed of a mixture of platforms with different costs, capabilities and energy budgets.

While adding more powerful hardware increases the capability of the network, there are currently few guidelines about how to structure software to easily take advantage of a heterogeneous sensor network.

Mote Herding is a system architecture and a set of services that targets *large scale*, *heterogeneous* wireless sensor networks. Its basic network abstraction and building block is the *flock*. The flock is a collection of motes that connect via a multi-hop routing tree to a microserver

called the *shepherd*. The microservers communicate among themselves with a separate, higher-capacity network over which they can exchange information more rapidly and more reliably than motes. The collection of flocks and corresponding shepherds forms the *herd*. The end result is a *tiered* system, that can disseminate data efficiently through an overlay network and multiple trees.

Mote Herding is based on the principle that moving complexity from the mote network to the microserver network aids development of more reliable and efficient services. Mote Herding accomplishes this by leveraging the CPU and storage resources of microservers to improve the computational capability, decision-making and long-term reliability of the deployed network. To demonstrate this we have developed several Mote Herding *services* based on this principle. We consider two classes of services: *flock* services run on a single flock and the corresponding shepherd, while *herd* services span the entire herd and use flock services to interface to the mote network.

The main contribution of this paper is to introduce Mote Herding as a new architecture for large, tiered, wireless sensor networks. We present the design principles behind this architecture (Section 2), and demonstrate the effectiveness of these principles by designing communication services tailored to this architecture (Section 3) and building higher-level services on this foundation (Section 4). We then evaluate the benefits of tiered architectures (Section 5). Our results show that our routing service is able to maintain higher than 99% connectivity in medium or high network densities while incurring a low overhead. In addition, our reliability service is able to deliver 100% of the data with low latency even in the presence of considerable link failures. Throughout the paper we present related work as it arises.

2 Mote Herding Architecture

The basic network abstraction of the Mote Herding architecture is the *flock*. The flock is a collection of motes that can communicate (with the routing protocol we describe in Section 3.1) to a microserver, called the flock's *shepherd*. Shepherds use standard IP-based ad hoc routing over an 802.11 network to run a *reliable state replication* protocol (described in Section 3.2) that allows them to easily share changing information like flock membership. The *herd* is the collection of *flocks* and *shepherds* that forms the entire mote/microserver network. Figure 1 illustrates the components of the architecture.

Each mote has a one-to-one relationship with its shepherd, and so is part of exactly one flock at a time. The shepherd is responsible for its own flock and is also *authoritative* for it: for example, queries from other mi-

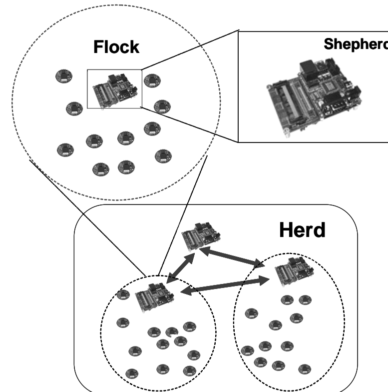

Figure 1: The Mote Herding architecture and its components, the *flock*, the *shepherd* and the *herd*

croservers about motes in a shepherd's flock are answered authoritatively by the shepherd. This however does not preclude other microservers in the network keeping cached copies of another shepherd's list. In that aspect, the design is similar to DNS.

To simplify on-mote routing, intra-flock routing is tree-based, with each mote knowing only how to reach the shepherd of its flock . In addition, inter-flock communication can only take place in the *shepherd* network. In this architecture, arbitrary mote-to-mote routing is therefore only possible through the herd-level overlay network. Compared to the intra-flock network, herd-level (inter-flock) routing is mesh-based, treating all microservers as peers.

Based on the above, Mote Herding can be thought of as a *locally centralized/globally distributed* architecture. Each flock has centralized properties, since data fusion and flock control happens on the shepherd, while the network of *shepherds* is a distributed system that shares information about each flock, delegates tasks to the appropriate shepherds and makes collaborative decisions.

While there have been several specific approaches to exploit the capabilities of tiered networks [24, 15], development thus far has been difficult because each approach has been hand-crafted. Mote Herding begins to explore guidelines to clarify how services should be developed in tiered networks, as some other concurrent projects are also doing [12]. Mote herding preserves the peer-nature of original sensor networks, but only at the microserver tier, adopting a centralized approach to simplify the mote level.

2.1 Design Principles

Mote Herding revolves around the following primary design principle:

Shift system complexity from motes to microservers for those operations that require distributed decision making on the motes.

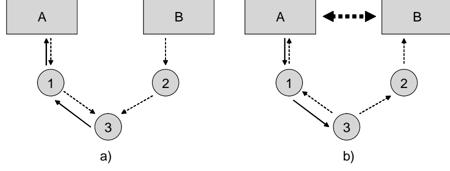

Figure 2: A sink selection scenario that illustrates the difference between mote-based and microserver-based distributed decision making.

This design principle yields several corrolaries:

1. Centralize decision making on a microserver thereby improving system performance as decisions can be made based on a more complete data set.
2. Reduce volatile state required for system functionality on motes.
3. Utilize computational and communication resources on microservers along with their peer nature to support more complex functionality.
4. Program a significant part of the system in a familiar and resource-rich environment.

In a collaborative distributed system, nodes make decisions based on their local information as well as information shared by other nodes. In the majority of cases, the *accuracy* and *correctness* of the decision is directly proportional to the amount of information available to the node, both spatially (i.e., reports from as many nodes as possible) and temporally (i.e., history of past reports). The resource constraints of the 8-bit platform however place a limit on the spatial and temporal fidelity of the data. However, the limited storage and network capacity available to an 8-bit mote platform constraints their spatial and temporal views. As a result, decisions made on a mote often must be based on limited information, even assuming complete message reception.

On the other hand, the 32-bit platform has ample storage and computational power to support collaborative decision-making algorithms and more complex algorithms than are possible on motes. They also have sufficient network bandwidth to operate in a distributed manner. This observations suggests that microservers can serve as "concentrators" or "fusion points", taking input from multiple motes in a flock and combining this information to provide better decisions than an individual mote can make alone. Such an operation requires that motes send data back to their microserver. However, this does not apply to all forms of data; only data required for decision-making needs to be sent. Sensor data is *local* to a node and does not require knowledge of or communication with other nodes. In contrast, neighbor estimation, path estimation and best sink selection are operations that need data from other nodes and as such incur a communication cost as well as a storage cost.

Figure 2 presents an example that illustrates the differences between mote-based and microserver-based decision making. Mote 3 wishes to attach to the best available sink, based on some well-defined selection metric. In (a), the microservers send beacons (dashed arrows) that get propagated to the mote network. The mote receives the beacons from both microservers and selects the best one to reply to. In (b), the mote sends a join beacon (dashed arrows) that eventually reaches the microservers. The microservers compare metrics and the "winner" replies to the mote. In (a), the memory and bandwidth requirements on mote 3 scale linearly with the number of microservers. In (b), memory and bandwidth requirements are $O(1)$. By delegating decisions to the microservers, we can therefore reduce the motes' state and bandwidth requirements and thus improve the *network density* scaling properties.

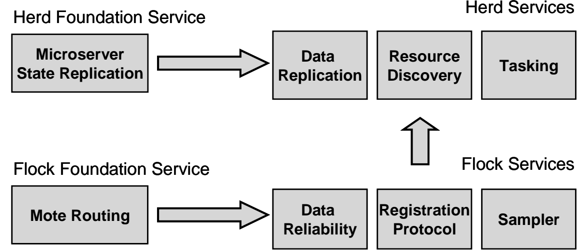

Figure 3: The Mote Herding Service Classes and their corresponding foundation services.

The mote network bandwidth still places a limit on the amount of data that can be sent to a single microserver. As networks grow in numbers of nodes, at some point sending all relevant data to a central point becomes impossible. If *multiple* microservers exist, each gathering system data for its own flock, the microservers can then exploit their high-bandwidth network to *share* that information among themselves. By replacing a flat network with two radio tiers it becomes possible for each microserver to maintain a view of the entire network while not overburdening individual motes. Adding multiple microservers to the system therefore has a dual effect: it reduces the *effective* diameter of the mote network while giving each microserver a *global view*.

2.2 Mote Herding Services

Building on the concepts of *flocks* and *herds*, Mote Herding services are split into flock- and herd-wide services.

Flock services are services that operate on a flock of motes under supervision of a single shepherd. Those services are "hybrid" in the sense that they reside on both motes and microservers. Flock services adhere to the design principles outlined above: motes deliver data to the shepherd; the shepherd makes decisions based on that data and then forwards those results back to each mote. These services emphasize the *master-slave* relationship

that exists between flock motes and their shepherd. Flock services can use other services in the same flock (we show example interactions in Sections 4.1 and 4.2.1), but they do not interact with motes in other flocks.

Herd services are services that operate on the entire network, building on flock services to provide more complete applications. Unlike flock services, herd services run only on the higher-performance microserver network. If they need to interact with motes, they do so with an appropriate flock service. Herd services treat all microservers as peers, allowing distributed and collaborative decision making based on both flock services and other herd services.

The commonality across all services is distributed decision making. Multi-hop communication and information sharing is therefore the foundation of other services. These *foundation* services for communication are simple multi-hop routing inside flocks, and a state replication service called *StateSync* at the herd level. We explore these foundation services in the next section then present several higher-level services built on top of this foundation in Section 4. Figure 3 shows the relationship between foundation and composite services at the flock and herd levels.

3 Foundation Services

Mote Herding foundation services provide the essential functionality on which higher-layer services build upon. These services are bidirectional mote-to-microserver routing at the flock level, and reliable state replication at the herd level.

3.1 Flock Foundation Service: Mote Routing

Flock services depend on communication between motes and their shepherd. Motes pass data, possibly over multiple hops, to their shepherd, while the shepherd dispatches control messages to motes. This section explores our design of centralized, bidirectional routing within a flock.

Tree-based routing is widely used in sensor networks, where all nodes construct a multi-hop routing tree to a centralized root (or gateway or sink). In Mote Herding we construct many such trees, one per flock, between the motes of the flock and their shepherd.

MintRoute [22] and Directed Diffusion [13] are two routing protocols that are often used in sensor networks. The *Multihop* routing protocol which is part of the ESS deployment at James Reserve [26] combines a simplified Diffusion and MintRoute. Each of these protocols uses a *distance-vector*-like routing algorithm to determine routes back to the sink, and employs *link estimation* and *neighbor tables* to dynamically discover the topology of the wireless network. With distance vector routing algorithms, each node independently selects a next hop from its neighbors that reduces its distance towards the sink. Some protocols use different different metrics (such as ETX [6]) to define the "best" next hop, possibly based on link quality, available energy, or other characteristics.

Recent experiences with the ESS deployment have uncovered several underlying issues that stem from the aforementioned properties of those routing algorithms combined with the resource constraints of the mote platform. Those issues are: (a) neighbor-table overflow, (b) count-to-infinity and (c) routing loops. We address each of these issues in more detail.

To populate its neighbor table, the mote needs to learn about its neighbors. The state requirements for the neighbor table are therefore $O(n)$, where n is the number of neighbors that a mote can have. Because of resource constraints, motes are typically configured with relatively small, statically sized neighbor tables. In dense wireless sensor networks, the number of potential neighbors can exceed the available storage capacity allocated for the neighbor table [22]. In that case, the designer of the routing protocol must implement a *neighbor eviction policy* [22]. In doing so however, the selection of a parent is now based on a *subset* of the actual neighbor list, thus artificially limiting the choices (and paths) a mote can have. Moreover, eviction/insertion policies suffer from *thrashing*, where a mote is inserted in the table only to be removed in the next iteration. This can lead to routing instabilities if this mote was indeed the next hop towards the sink.

The distance-vector protocol is known to suffer from the "count-to-infinity" problem. This can adversely affect the performance of routing by creating *loops*. As a result, special care (like poison-reverse or split-horizon) must be taken to avoid loops or repair them if they form. In MintRoute, motes monitor forwarding traffic and snoop on the parent address in each neighbor's messages in order to identify neighboring children and thus disqualify them from the parent selection process. Nevertheless, those mechanisms are not infallible and loops can still form (especially n-hop loops). In particular, in one of our MintRoute simulations, we experienced a system-wide count-to-infinity problem that resulted in all the motes in the network having routing loops.

The aforementioned issues are examples of distributed decision-making on the motes that is based on *incomplete* and potentially, *stale*, data.

In mote herding we use CentRoute to communicate between motes and their shepherd. Following the principles layed out in Sections 2.1 and 2.2, CentRoute shifts all decision making to the shepherd. CentRoute ad-

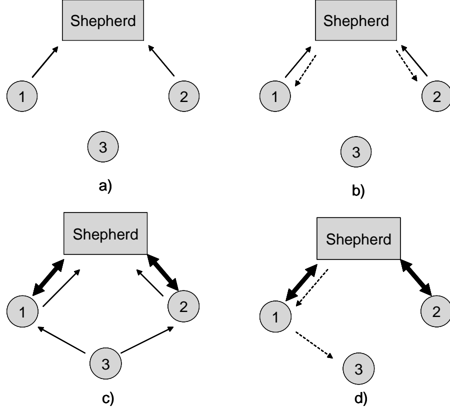

Figure 4: A CentRoute tree forming (join) operation.

dresses the main problems described above: since it runs on a microserver with megabytes of memory, neighbor table size is not a limitation. Since all path selection and routing decisions are made centrally, different motes will not come to different decisions that can lead to inconsistencies. In Section 5.1.1 we conduct a number of simulations to show that these approaches in CentRoute are successful at providing high network connectivity and reducing the number of loops.

3.1.1 Routing tree formulation

Following the principles outlined in section 2.1 we designed CentRoute to be a dynamic single-shepherd routing protocol based on source routing, with all routing decisions made on a centralized point, the shepherd.

In order to join a tree and thus be part of a flock, each mote periodically broadcasts join request messages. If a mote that is already part of a tree receives this message, it *forwards* it towards its shepherd, via its parent, by generating a unicast packet. In addition, the forwarding mote adds its own address to the packet, thus recording the path the packet is taking towards the sink. This is essentially a similar approach to Source Routing [17]. Upon reception of a request message, either by direct broadcast or via a forwarded packet, the microserver will generate a join reply packet and unicast it towards the requester, by reversing the packet's received path and using source routing. In addition, the microserver stores this path information to use whenever it needs to send a unicast packet to that mote. When the mote receives the join reply message, it becomes attached to the replying microserver—its shepherd (we discuss multi-shepherd resolution in Section 3.1.4). It also sets the last mote that forwarded the join reply as its *parent*. Finally, since the mote is now part of a flock, it stops generating further join request messages.

The join operation in CentRoute happens in *phases*, since, in order to forward a join request packet, a mote needs to be attached to a tree. When the network starts up, no motes (besides those directly attached to microservers) are parts of a tree. Initially, only the motes that are one hop away from a microserver will be within range of an attached node (in this case, the microserver). In the next round, the motes that are two hops away will be able to attach, and so on, until the entire network becomes connected. Figure 4 shows an example tree forming operation. In (a), all motes send a join request, but only motes 1 and 2 are within range of an existing tree (the shepherd itself). In (b), the shepherd sends join reply messages (dashed arrows) to motes 1 and 2, thereby grafting them to the tree. In (c), mote 3 sends another join request. This time, motes 1 and 2 which are now part of the tree forward the join request towards the shepherd. In (d), the shepherd grafts node 3, via the path that goes through mote 1. Mote 3 also sets its parent to be node 1, so any subsequent upstream traffic will be sent to it.

The centralized nature of the tree formulation in conjunction with source routing and the absence of *any* mote decisions concerning path selection results in a considerable reduction in the number of transient loops (Section 5.1.1). In particular, routing state is always set up by constructing a path from the shepherd to the mote via a source routed packet. This packet trails behind it a loop free path to the shepherd so any old state in the network that somehow happened to meet this path would therefore have a loop free path to the shepherd as well. As a result, no route is ever added that can have a loop, and no "old" route that is disrupted can result in a loop because all it can do is join a loop free route, or not work at all. A formal analysis of the loop performance of CentRoute is part of future work.

3.1.2 Routing Metric and Link Quality Estimation

Since path selection is done on a centralized point in CentRoute, the motes need to provide the shepherd with sufficient information for that decision. When a mote receives a request, it will include a *link quality estimate* when forwarding the request towards the shepherd.

In the example of Figure 4, motes 1 and 2 will write their estimate of link quality to mote 3 into the request before passing the join request towards the shepherd. These estimates computed at join time enable the shepherd to annotate a tree with link quality estimates along each edge, and thus to select paths using ETX [6] as a routing metric.

Motes in CentRoute use the join request beacons to get link quality estimates, by having each mote send several beacons in quick succession. In order to reduce control overhead, receiving motes that are part of the tree only forward an *aggregate* join request that contains the the total number of broadcast requests received by that mote

in that period. Using this information, the microserver has sufficient data to form an *instantaneous* link quality estimate and subsequently, a path estimate.

An important property of the estimation algorithm is that link quality estimates need only be kept for as long as the mote is trying to join. This is in contrast with traditional link estimation schemes where estimates need to be kept at all times and cannot be discarded unless a mote is evicted from the neighbor table.

3.1.3 Stability and Maintenance

In order to increase the stability of the network, we designed CentRoute so that it doesn't try to find improved paths (in terms of its routing metric) once the original paths have been established. As long as the individual link qualities don't change radically, the initially selected path will not be too far from the *optimal* path, at any point in time. As a result, the protocol can form more stable topologies, while sacrificing optimality at some future point in time. This process also has an "annealing" effect towards stability, because temporally stable links, once discovered, are likely to remain selected.

Even though CentRoute does not attempt to find a better path when one has been established, it still needs to ensure that the existing path is *valid*, i.e., all links from the mote to its shepherd are still viable. CentRoute uses link-layer ACKs and packet retransmissions to deduce link failure. Once the link fails, the mote at the point of failure considers itself detached from the tree. It then invokes the *join* mechanism, described in 3.1.1, by starting to transmit join request messages. If a mote receives a join request from its parent, it immediately detaches itself from the tree and also invokes the *join* mechanism. Since the repair mechanism is in effect the same as the join mechanism, it inherits its loop prevention properties.

In order to reduce the control overhead when the network is quiescent CentRoute is an *on-demand* protocol. As a result, CentRoute does not require periodic control messages to maintain its paths; instead it discovers and repairs broken paths on demand, when triggered by data packets. This approach trades off reduced control overhead for potentially higher path repair time.

3.1.4 Dynamic shepherd selection

To increase the robustness of the network, CentRoute was designed to support *dynamic shepherd selection*. Since the mote assignment to flocks and shepherds is not pre-configured, a mote can associate with a different shepherd and flock if its current shepherd becomes unavailable, such as due to microserver failure. We choose not to allow motes to associate with multiple shepherds in keeping with our goal of minimizing mote-side complexity (Section 2.1).

When multiple microservers exist in the network, a mote join request can be received by several microservers, as shown in Figure 2. Since mote-to-shepherd assignment is one-to-one, the "competing" microservers need to select which flock the mote should join. As each microserver already knows the best path to the mote from its own flock, the microservers need only compare the values of their local path metrics for the mote in question. Essentially, this is the same operation as path selection on a single microserver, with information about alternative paths through other flocks provided by other microservers.

3.2 Herd Foundation Service: Microserver State Replication

Herd services run on the microserver network, joining all microservers into a common distributed system. To support distributed algorithms that depend on information from multiple flocks, *microserver state replication* is provided as the foundation of herd services.

Mote Herding uses a protocol called StateSync [10] as its foundation service for reliable state replication across the herd. StateSync is a proactive publish-subscribe system that strives to provide low-latency updates to published data over multiple hops with low overhead. StateSync is a reliable protocol with a probabilistic latency bound that publishes a node's state over multiple hops by proactively sending differences. Through the StateSync API, a node publishes a table of data, and the StateSync system automatically computes the diff required to update the subscribers. StateSync is designed to support applications that need reliable publication of their current state and timely propagation of updates, for cases where the subscriber needs access to the data at a higher rate than the rate of change of the data values. ISIS also provided reliable shared data, but targeted primarily at LANs [3].

StateSync's proactive publishing model does not fit all applications. Those that have a high ratio of data elements and updates to data access require a cached request-reply protocol such as Hood [21]. This type of service is important future work for Mote Herding.

4 Flock and Herd Services

Using the foundation services we can now build composite flock and herd services. In addition to the foundation services for flock and herd communication, Mote Herding currently provides two flock services, mote data reliability and a mote registration protocol and three herd services, resource discovery, resource caching and query dissemination. We have considered a number of possible additional services, including a global tasking service

that distributes tasks to each mote in the network, a data replication service that replicates flock data at each microserver to improve robustness and a system monitoring service that continually monitors the entire network. Implementation of these additional services is future work.

4.1 Mote Data Reliability

In mote-based Wireless Sensor Networks, the wireless communication is known to show variable and often large loss rates [5, 25, 22]. As such, application designers need to take packet loss as a given and design applications to be loss-tolerant. An important class of applications require high reliability and would benefit from a *data reliability* service.

DataRel is the Mote Herding data reliability protocol, a flock service that strives to provide guaranteed data delivery in a lightweight manner. As a result, we designed DataRel to be a window protocol that uses end-to-end ACKs and source-based packet buffering, in a manner similar to TCP. DataRel can use end-to-end ACKs efficiently, since the CentRoute service allows for unicast packet transmissions from the shepherd to any mote in the flock. In addition to end-to-end ACKs, DataRel utilizes link-layer ACKs and hop-by-hop retransmissions to improve the probability of successful delivery, contrary to PSFQ [20] and similar to RMST [18]. RMST operates solely on 32-bit nodes over Directed Diffusion with all nodes being peers while DataRel is a mote-to-shepherd reliability protocol. PSFQ is a mote-based hop-by-hop protocol that uses NACKs for error recovery and a controlled flooding algorithm for data distribution.

The mote's shepherd is not necessarily the *final* destination for a data packet. Once the packet reliably reaches the shepherd, it can be forwarded to its final destination using a conventional IP reliability service, like TCP. However, for the purposes of the DataRel protocol—and since DataRel is a flock service—we consider the endpoints to be the source mote and its shepherd. This approach, which is similar to Split-TCP [1], is based on the realization that, like satellite and terrestrial networks, the mote network and the microserver network have very different communication properties, especially in terms of bandwidth, latency and bit error rate. Alternatively, one could view DataRel as an example of delay-tolerant networking [8], where the flock and the herd are separate DTN regions.

4.2 Resource Discovery

Our first composite service at the herd level is *resource discovery*. Resource discovery is essential in sensor networks where nodes have different capabilities, such as different sensors or actuators, or where applications wish to select subsets of the network based on other factors, such as location or coverage area.

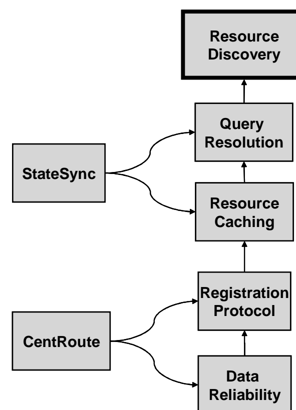

Figure 5: The Resource Discovery service connection diagram.

Common approaches to resource discovery include: direct queries on demand, in-network caching or caching at a central site. The one-phase pull variant of Directed Diffusion [13] is an example of a direct query approach, while the Ninja Service Discovery Service [7] is an example of an in-network caching approach. The caching approaches in general provide faster query resolution speed, at the expense of additional complexity and overhead for handling cache misses or "stale" data.

Much like routing, the main factor that influences the choice between on-demand and a priori approaches is the amount of temporal variation in the data. Sensor data is tied to physical phenomena and as such can change rapidly or slowly, depending on the physical process being measured. For example, an acoustic signal changes much more rapidly than a temperature value. In addition to sensor data, link quality, instantaneous queue size and received signal strength indicator are other examples of frequently varying data. On the other hand, hardware and software capabilities of a platform constitute information that changes slowly over time. For example, the actual sensors installed on a mote (as opposed to their values) or the mote's radio constitute information that characterizes the mote and does not change often. In terms of overhead, an on-demand polling protocol is more appropriate for high temporally varying data, while an a priori caching approach is better suited for low temporally varying data as information can remain "fresh" for larger periods of time.

We have implemented ResDisc using the mote herding principles. Since mote resources for the most part exhibit low temporal variation, we adopted the in-network caching approach. The design of ResDisc is based on three sub-services. The *registration protocol* flock service provides resource data from the motes in the flock. The *resource caching* herd service is responsible for ex-

porting local resource data provided by the registration protocol as well as maintaining its cache of resource data from other shepherds. Finally, the *query resolution* herd service is responsible for replying to resource discovery queries. Figure 5 presents the Resource Discovery service diagram.

In the following sections, we describe the registration protocol and resource caching service in more detail. The query resolution service has not been completely implemented yet and as such we defer its discussion to future work.

4.2.1 Registration Protocol

The goal of the registration protocol is to "register" resource information of a mote with its shepherd, as well as keep the shepherd updated if this information changes. Consequently, the registration protocol is a *flock service*.

To reduce communication overhead as well as improve the update time, we designed the registration protocol to be event-based as opposed to using a periodic refresh mechanism. As a result, the protocol is invoked automatically when one of the following events occurs: the mote joins a shepherd's tree, a registered resource changes value or when instructed by an application. A further optimization that has not been implemented yet includes responding to direct queries from the shepherd. To ensure correct ordering of events as well as resolve potential ambiguities, we used a simple sequence number scheme in which a new sequence number is generated when a new event occurs.

The absence of a periodic refresh mechanism introduces a reliability issue, as registration messages, if lost, will not be repeated. As a result, the registration protocol uses the DataRel mote data reliability service introduced in section 4.1 to guarantee reception of its messages by the shepherd. Finally, to allow for flexibility in the registration data, we chose to use a variable-size type-length-value (TLV) format for the actual data transfer.

4.2.2 Resource Caching

One of the goals of the resource discovery service is the ability to reply to any query as quickly as possible, by having all available information on each microserver. To accomplish this, we designed the resource caching service, a herd service that supplements the registration protocol flock service.

Resource caching is a simple service that is responsible for exporting resource data obtained through the registration protocol to other shepherds. At the same time, it receives similar data from the remote shepherds and caches it. To accomplish this functionality as well as ensure consistency among caches throughout the microserver network, resource caching uses the state replication herd foundation service. In each shepherd, resource caching maintains a table containing all motes it has data for (potentially all motes in the network). This table can then be used as an input to the query resolution service.

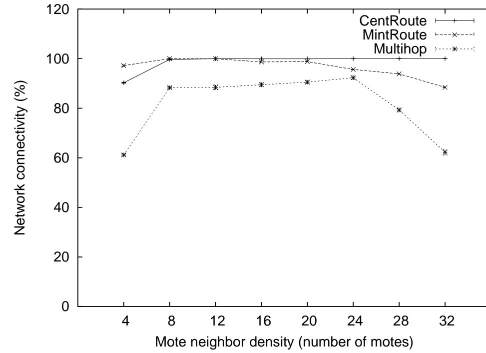

Figure 6: Network connectivity percentage as a function of mote neighbor density for CentRoute, MintRoute and Multihop.

5 Evaluation

In this section we evaluate the three Mote Herding services that have been fully implemented, namely routing and data reliability for flocks, and resource discovery for herds. We also use a habitat monitoring application to evaluate a set of Mote Herding services working in collaboration. We run our experiments using the EmStar framework [9]. The flock services are evaluated using the EmTOS [11] emulation module of EmStar, which allows development of fully functional NesC applications in the resource-rich environment of a 32-bit platform.

5.1 Mote Routing

Our routing experiments focus on three important aspects of any routing algorithm, specifically network connectivity, loop probability and control overhead. In all our routing experiments, the experimental topology consisted of a 100-mote simulated network, arranged in a 10-by-10 grid with uniform 10m spacing between motes. We then place a single sink, in the middle of the bottom row of the grid at coordinates (50m,0m). To study variable network densities we keep this topology fixed and vary the simulated radio range. In all our routing experiments, we compare the performance of CentRoute to MintRoute and Multihop. The neighbor table size on both MintRoute and Multihop was set at their default value of 16. We vary network density, so at high densities we expect the number of neighbors to exceed this fixed routing table size. The periodic beacon rate for MintRoute and Multihop was set to 30 seconds. Since CentRoute is an on-demand protocol, it requires data trans-

missions to perform route maintenance, as seen in Section 3.1.3. Therefore, the CentRoute experiments contained data transmissions, at a rate of one packet every 30 seconds per mote. The maximum number of transmission retries per packet, used in the CentRoute repair mechanism was set to 5.

5.1.1 Network Connectivity

First we look at how well each routing protocol maintains connectivity to the sink. We study how connectivity varies as we change the network density by increasing the simulated radio range.

For the purpose of our experiments, we define *mote neighbor density* to be the number of one-hop neighbors of a given mote that have a link quality of *70% or higher*. Since we use a grid topology with even spacing, the neighbor density increases in steps of 4. We note that the *actual* number of motes that a single mote can hear can be considerably higher than its neighbor density [22] and that property is reflected by our simulated channel model.

In order to measure network connectivity, we periodically trace a path from each mote in the network towards the sink. If a path is found, we consider the mote to be connected for that particular time sample. At each time sample, network connectivity is the sum of all the motes that have a path to the sink. The duration of all our routing experiments was 3 hours and our sampling period was 10 seconds. The first 5 minutes of each experiment were discarded, so as to allow each routing algorithm to reach stable state. In this and all following experiments, error bars in experimental results are 95% confidence intervals.

Since CentRoute does not contain any neighbor tables or any other per-neighbor state that is limited to a maximum number of motes, we expect it to perform better as mote neighbor density increases. By contrast, we expect MintRoute and Multihop to exhibit loops and blackholes as density increases, for the reasons given in Section 3.1.

From Figure 6, we note that with CentRoute the network is almost always connected, regardless of mote neighbor density. In fact, the performance of CentRoute actually *improves* as density increases, rising from 90% at mean density of 4 neighbors to 99.9% when nodes have 12 or more neighbors. In contrast, MintRoute starts at 97% connectivity then reaches 99.9% in medium densities, but connectivity falls off as density exceeds the static routing table size (above 16 neighbors). This degradation is because, when neighborhood size exceeds the routing table size, there are several cases in both MintRoute and Multihop when mote A considers mote B a neighbor but not vice versa. This disparity is due to table thrashing—mote A has mote B in its table, but mote B has evicted mote A. As a result, a mote is unable to find a parent and stays disconnected for large periods of time. This problem could possibly be solved by exchanging neighbor evictions at the cost of even greater complexity in the mote network.

Neighbor density (motes)	**CentRoute loop prob. (%)**	**MintRoute loop prob. (%)**	**Multihop loop prob. (%)**
4	0	0.47	2.11
8	0	0.01	1.92
12	0	0.02	1.88
16	0	0.03	2.48
20	0	0.03	3.65
24	0	0.04	4.8
28	0	0.04	3.12
32	0	0.02	7.27

Table 1: Average loop occurrence probability as a function of density for CentRoute, MintRoute and Multihop.

Multihop exhibited its best performance in medium densities (8–24 neighbors/mote). At high densities (24–32 neighbors/mote), Multihop performance is hampered by a high probability of a *loop* occurrence in addition to the table thrashing problem. At 4 neighbors/mote, Multihop could not connect the network because of its routing metric, an 8-bit value that decreases exponentially (based on link quality) as the path length increases. In low densities, when path lengths are considerable, the routing metric reaches its minimum value of 0 after a few hops and as such no more motes can be added to the path. In effect the current Multihop implementation has an arbitrary maximum path length cutoff point.

The initial decreased performance of CentRoute is due to its centralized nature in conjunction with the tree formulation and repair mechanism. A path in CentRoute will be invalidated if *any* node in the path fails to transmit a packet to its parent, after a certain number of retries (currently set at 5 retries per attempt). Therefore the probability of a path invalidation is directly proportional to the path length, which in turn is inversely proportional to neighbor density.We could reduce this problem by increasing the number of retries at low densities, or after repeated failures.

To understand the relative effect of loops on network connectivity, we explicitly measured the number of loops that occurred for each protocol. The CentRoute protocol, due to its centralized shepherd-based path construction and source routing is expected to have very few, if any, loops (Section 3.1.1). The distance-vector algorithms are expected to have more loops than CentRoute. Multihop, in particular, that has no loop-prevention mechanism is expected to have the worst performance.

Table 1 shows the percentage of loops encountered as a function of density for the three routing protocols. CentRoute did not exhibit any loops that were detectable

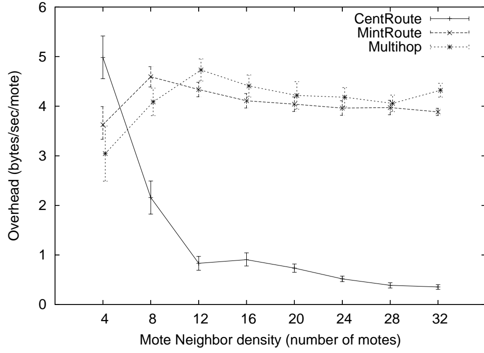

Figure 7: Per-mote byte overhead as a function of mote neighbor density for CentRoute, MintRoute and Multihop.

Usage (bytes)	RAM	ROM	RAM cost per neighbor
CentRoute	1274	17184	0
MintRoute	1689	12588	18
Multihop	1560	17292	19

Table 2: Mote RAM and ROM usage in bytes for CentRoute, MintRoute and Multihop.

with our 5-second sampling period. A proof that CentRoute is loop free is an area of active work. MintRoute loop probability was highest in the low density case but was an order of magnitude less in all other case. We believe that this disparity is due to *n-hop loops* in conjunction with the particular geometric properties of the grid topology. MintRoute does not allow a mote to select any of its neighboring child motes as a parent, thus preventing 1-hop loops. N-hop loops cannot be detected however. At very low densities, the limited number of alternative paths in the network combined with the larger path lengths and the geometric properties of a grid topology result in n-hop loops occurring more often. Finally, Multihop which lacks a loop-prevention mechanism and also uses a non-monotonic routing metric has the highest probability of a loop occurrence.

5.1.2 Control Overhead

The control overhead of the routing algorithms is the transmission overhead incurred by their operation as well as their memory usage on the mote. The transmission overhead of MintRoute and Multihop is due to their periodic neighbor beacons and route advertisement messages. The transmission overhead of CentRoute contains join requests as well as join forward and join reply messages. In this section we conduct experiments to verify that CentRoute control overhead is similar to overhead of alternatives.

CentRoute repairs can be quite expensive especially if the paths are long and the failure happens close to the sink, as a considerable number of motes will be affected. In contrast, the distance-vector protocols do *local* repairs which are not as expensive. As a result, we expect the CentRoute overhead to be high when the network is sparse and get progressively lower as the density increases.

Figure 7 depicts the average per-mote transmission overhead in bytes per second for the three routing protocols as a function of mote neighbor density. In the lowest density, CentRoute has the highest cost of all the three protocols. This is due mainly to the repair mechanism's inability to find good paths, as the available selections are limited. As the network progressively becomes more dense, CentRoute can find alternate paths to the sink that are more stable than the initial selection. The repair mechanism does not need to be invoked as often and transmission overhead is reduced considerably.

MintRoute and Multihop, on the other hand, initially start with a low transmission overhead but grow greatly as the neighbor density increases. Since neighbor tables are transmitted by each mote as part of the link estimator, higher densities result in a higher transmission overhead, up to a saturation point that is based on the maximum size of the table. The overhead reduction for MintRoute and Multihop in very high and very low densities is due to their route advertisements, which are only transmitted when a mote is connected. As shown in Figure 6, network connectivity for MintRoute and Multihop is reduced at very low and very high densities and as such less route advertisements are transmitted.

A second kind of overhead is memory usage of each algorithm. Table 2 shows the RAM and ROM footprint of each of the three routing algorithms, using the TinyOS default message length (29 bytes payload). We note that CentRoute has the lowest RAM usage which doesn't increase with the neighborhood size. In contrast, MintRoute and Multihop have comparable RAM usage and per-neighbor costs. The per-neighbor cost places a considerable limitation in the operation of the distance-vector alrogithms. Increasing the neighbor table from its default size of 16 to 50 so as to support larger numbers of neighbors incurs an additional RAM cost of about 680 bytes, while the size of the table itself is almost 1000 bytes, i.e., 25% of the total RAM of the Mica2 mote. The real advantage of CentRoute here is that it shifts the burden of variable size data structures like the routing table to resource-rich shepherds, while the other protocols must allocate routing tables statically sized to accommodate worst-case neighborhood sizes.

Based on all our routing experiments we feel that the CentRoute foundation service has achieved its design goals. By centralizing the routing decisions on the microserver and removing per-neighbor state on motes CentRoute manages to scale well with increasing neighbor density as well as avoid loops while at the same time

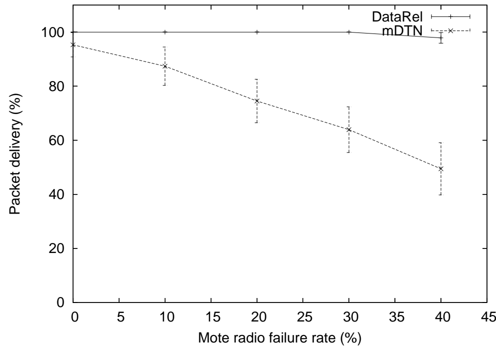

Figure 8: Average packet delivery ratio as a function of mote radio failure rate for DataRel and mDTN.

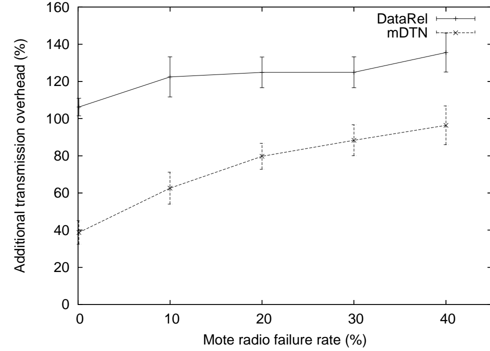

Figure 9: Per-mote additional transmission overhead as a function of mote radio failure rate for DataRel and mDTN.

incurring a low overhead. However, the centralized property of CentRoute is not well suited for sparse networks with long path lengths. The reason is due to the overhead associated with transmitting all the control data back to a single point in addition to the expensive repair mechanism is invoked more frequently.

5.2 Data Reliability

Our reliability experiments focus on the performance of two mote data reliability protocols: the DataRel flock service introduced in section 4.1 and the mDTN reliability protocol used in the ESS deployment [26]. a hop-by-hop protocol called *mDTN* that is based on the principles of *Delay Tolerant Networking* [8]. MDTN uses link-layer ACKs and hop-by-hop retransmissions to reliably deliver the data. If next-hop delivery fails, mDTN stores the packet in non-volatile storage (the mote's flash memory) for later retransmission. DataRel ran over the CentRoute service while mDTN ran over the Multihop routing protocol. For our experiments, we again used the 100-mote grid topology plus a single sink in the middle of the bottom row. However, we now fixed the mote neighbor density to 12 and instead changed the *mote radio failure probability*. We used a simple failure model, in which every mote has a given probability k for its radio link to fail. The failure generator runs periodically and applies the failure probability to every mote in the network. In the next iteration, the failure generator "revives" failed motes before applying the failure probability again. The result is that at any point in time $k\%$ of all the motes in the network will be unable to receive or transmit any packets.

In all our data reliability experiments we used a simple periodic data generating application. The data transmission rate was set to one packet every 30 seconds, while the fault generator ran every 5 minutes. Our experiments lasted until all motes had transmitted 200 packets each, or until a 6-hour limit was reached. The DataRel window size was set to 1, so an ACK was sent by the sink across the flock for each packet received. Maximum hop-by-hop retransmissions for each protocol was set to 5. During the course of our experiments we found that the default value of the mDTN retransmission timer was set too high, resulting in very high latencies and higher loss rates. As a result, we changed the retransmission timer period of mDTN from 10 minutes to 10 seconds, the same as DataRel's retransmission timer.

5.2.1 Packet Delivery Ratio

In order to measure the delivery ratio, we annotate each data packet with a sequential sequence number We then inspect the sequence number space for each mote at the receiver and discard duplicates. The remaining packets, divided by the total number of original packets generated by each mote is the per-mote packet delivery ratio. Finally, we compute the mean and 95% confidence intervals by averaging all per-mote results.

Figure 8 shows the average packet delivery ratio as a function of mote radio failure rate for the two protocols. When no failures are present mDTN delivers on average 96% of the data while DataRel delivers 100% The situation changes however as the mote radio failure rate increases. DataRel is able to keep 100% delivery rate for a failure rate of up to 30%, dropping to an average of 97.2% for a failure rate of 40%. At the same time, the performance of mDTN degrades considerably, especially at high failure rates. The performance degradation is due to the nature and implementation of mDTN. Specifically, mDTN uses link-layer ACKs to decide whether a packet has been successfully delivered to the next hop; if not, it stores it. Link-layer ACKs however carry no information about the packet's fate in *higher layers of the stack*. In mDTN, a packet can be successfully delivered to a mote but subsequently *dropped*. This is mostly due to a receive buffer overflow as a result of network congestion but can also be caused by more complex failure modes. A higher-layer ACKing scheme would solve this issue, at the expense of more transmission overhead. DataRel,

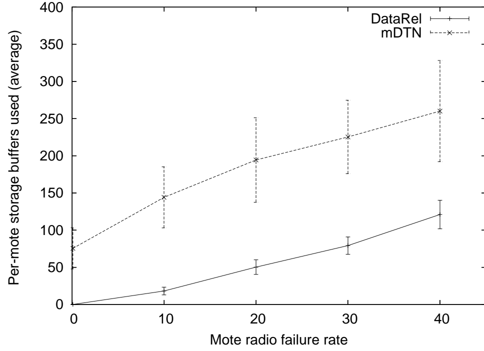

Figure 10: Per-mote average storage buffer usage as a function of mote radio failure rate for DataRel and mDTN.

being an end-to-end protocol, does not suffer from such issues.

5.2.2 Control Overhead

The control overhead of the reliability protocols has two aspects: transmission overhead and buffer overhead. The transmission overhead is the number of *extra* transmissions incurred by the protocol. If a packet needs to travel N hops to reach its destination then any additional transmissions over N constitute the transmission overhead. For mDTN the transmission overhead is related to hop-by-hop retransmissions at each mote. The DataRel transmission overhead includes hop-by-hop retransmissions (via CentRoute), as well as packet retransmission at the source and ACK transmissions and retransmissions. The buffer overhead is the amount of buffer space required at each mote to store unacknowledged packets. Since DataRel is an end-to-end protocol, we expect its transmission overhead to be considerably higher than mDTN. At the same time, since mDTN can potentially store packets on every mote along a path, we expect it to have a higher buffer usage.

Figure 9 shows the per-mote additional transmission overhead as a function of mote radio failure rate for the two reliability protocols. DataRel overhead is always above 100%. This is due to the end-to-end ACKs which even in the absence of retransmissions incur a 100% overhead for an ACK window of 1, as each packet needs to be ACKed. A larger ACK window on the sender would reduce the transmission overhead of DataRel at the expense of more buffer usage. As expected, mDTN is consistently less expensive than DataRel although the difference is less pronounced at higher failure rates. This is mostly due to the nature of the underlying Multihop routing protocol. When a mote radio fails, it takes a certain amount of time for Multihop to decide that this mote is no longer a valid parent. During this time, mDTN will continue to attempt to transmit data over that mote. When the failure rate is high this can lead to a large number of unnecessary retransmissions. In contrast, CentRoute path failure notification is very quick and as such DataRel does not suffer from this problem.

Usage (bytes)	RAM	ROM
DataRel	100	6952
mDTN	338	13278

Table 3: Mote RAM and ROM usage in bytes for DataRel and mDTN.

Latency (sec)	Mean	95% Quart.	Max
Registration Protocol	0.61	5.18	6.97
Resource Caching	0.12	0.32	0.41

Table 4: Resource update latency for the registration protocol and resource caching parts of the Resource Discovery service.

Figure 10 shows the average buffer usage required by each protocol as a function of mote radio failure rate. As expected, DataRel uses less buffers than mDTN since it only buffers a packet at the *original sender*. MDTN on the other hand can potentially buffer a packet at *every* intermediate hop as the packet makes its way towards the final destination. This case is especially probable at high failure rates.

Buffering at intermediate nodes has another effect on overhead: it requires additional logic on the reception data path as a mote needs to determine how to treat each individual packet. In addition, mDTN includes support for multiple sinks which further complicate its implementation.

Table 3 presents the RAM and ROM footprint of each protocol. Due to the aforementioned reasons, mDTN is more expensive than DataRel in terms of both RAM and ROM. The ROM cost in particular underlines the relative simplicity of DataRel compared to mDTN.

Our reliability experiments reinforce our design decision on shifting complexity from the motes to the microservers. DataRel, a simple end-to-end protocol can deliver 100% of the data even at high failure rates as opposed to the more complex mDTN. The difference is primarily due to the end-to-end ACKs made possible by using the CentRoute flock foundation service. However, as expected for an end-to-end protocol, DataRel incurs a substantial transmission overhead. Nevertheless, the overhead can be reduced by using larger ACK windows, at the expense of more storage buffer usage.

5.3 Resource Discovery

Our resource discovery experiments focus on the time required for updates to propagate to the entire network. This knowledge can be then used to provide an upper bound on the rate that a client can query the service. For this experiment, we used our experimental mote testbed consisting of 40 Mica2 motes placed at the ceiling of our building's 3rd floor, as shown in Figure 11. We used

a simulated channel for our microserver network consisting of 3 shepherds and assumed that all shepherds are within range. To simulate a resource update, we changed the "sensor type" value on each mote thus causing the registration protocol to propagate the update, as discussed in Section 4.2.1.

Our latency measurements consisted of two parts: the first part measured the time required for the registration protocol to update the resource value on the local shepherd. The second part measured the *additional time* required for the resource caching service to update all the shepherds in the network with the new data obtained from the registration protocol. The second measurement was complete when all three shepherds had converged on the same value.

Table 4 shows the mean, 95% quartile and maximum latency of an update, in seconds, for both the registration protocol and the resource caching service. Since the registration protocol operates on the mote network, it incurs a higher latency than the resource caching service which utilizes the 802.11 network. In particular, registration protocol latency is dependent on the mote's path length to the sink—with each hop adding approximately 100 milliseconds of propagation delay—as well as the quality of the path. A low-quality or broken path can result in a DataRel retransmission or even a CentRoute path repair thus having an negative effect on latency. The resource caching service does not suffer from those issues although we expect its latency performance to degrade as we increase the hop diameter of the microserver network, based on experiments presented in [10].

Based on our results, the mean time required for an update originating from a mote to reach all the shepherds is 0.73 seconds, while the maximum is 7.31 seconds. In the future, we plan to do more extensive latency measurements on our Resource Discovery service, in order to establish a more accurate upper bound as well as compare our design choices with the alternatives presented in Section 4.2.

5.4 A complete application: Habitat Monitoring

Our final set of experiments involved implementing a simple multi-sink *habitat monitoring* application using Mote Herding and comparing its performance with a similar application. In our application, motes periodically send data back to a single or multiple microservers using DataRel on top of CentRoute. We measured the application's performance with data delivery ratio, total number of transmissions and packet latency as a function of the number of *sinks* in the network and compared it with the ESS application, which uses mDTN on top of Multihop. Both applications were tested using our experimental testbed.

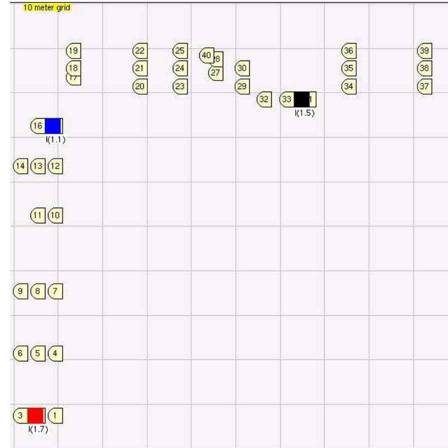

Figure 11: The experimental testbed used in our experiments. Nodes with darkened boxes were the shepherds/sinks.

Num. of sinks	Delivery Ratio (%) MH	ESS	Total transm. (packets) MH	ESS	Latency (sec) MH	ESS
1 (16)	100	84.24	1501.5	3089.4	2.75	12.27
2 (3,33)	100	90.30	710.28	2493.7	0.98	1.72
3 (all)	100	94.06	588.77	2423.9	0.24	0.63

Table 5: Delivery ratio, total number of transmissions and latency results for the Mote Herding (MH) and ESS versions of a habitat monitoring application as a function of the total number of sinks in the system.

Table 5 depicts the results of our experiments. The Mote Herding application outperformed the ESS application in terms of delivery ratio, total number of transmissions and latency. We note that increasing the number of sinks has a positive effect on the performance of both applications, especially in terms of reducing the number of transmissions and packet latency. A larger number of sinks reduces the effective mote network diameter, thereby leading to shorter paths.

Based on our encouraging initial testbed results, in the future we plan to deploy our Mote Herding-enabled habitat monitoring application in the field and compare its performance to that of real-world applications.

6 Conclusions

In this paper we presented Mote Herding, a new system architecture that targets large scale heterogeneous sensor networks, comprised of 8-bit motes as well as 32-bit microserver nodes. Mote Herding is based on two key abstractions: the *flock* which is a collection of motes connected to a single microserver called the shepherd and the *herd* which is comprised of the collection of flocks and their shepherds. Based on those abstractions, Mote Herding introduces two classes of services. Flock services are

services that run on a single flock and its shepherd. Herd services operate on the entire network but are located solely on the microserver network and interface to the mote network via corresponding flock services. A foundation service is used as a building block for all other services in its class.

Using the Mote Herding abstractions we implemented several services, including centralized intra-flock routing, a mote data reliability flock service and a resource discovery herd service based on three subservices. We evaluated the performance of our services using simulations and emulations of both sample scenarios and a habitat monitoring application and compared them with other protocols and services. Our results show that our centralized mote routing service, CentRoute, is able to maintain better than 99% connectivity at high network densities with 60% less overhead than two other routing protocols. Our data reliability service was able to maintain 100% reliability in up to 30% link failure rates. Finally, our emulated habitat monitoring application based on Mote Herding was able to consistently outperform a similar habitat monitoring application in terms of data delivery, transmission overhead and latency.

References

[1] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz. A comparison of mechanisms for improving TCP performance over wireless links. volume 5, pages 756–769, 1997.

[2] M. Batalin, G. S. Sukhatme, Y. Yu, M. H. Rahimi, G. Pottie, W. Kaiser, and D. Estrin. Call and response: Experiments in sampling the environment. In *ACM SenSys 2004*.

[3] K. Birman and R. Cooper. The isis project: Real experience with a fault tolerant programming system. *Operating Systems Review*, pages 103–107, Apr. 1991.

[4] A. Cerpa, J. Elson, D. Estrin, L. Girod, M. Hamilton, and J. Zhao. Habitat monitoring: Application driver for wireless communications technology. In *SIGCOMM Wksp. on Comm. in Latin America and the Carribean*, Costa Rica, Apr. 2001.

[5] A. Cerpa, J. L. Wong, M. Potkonjak, and D. Estrin. Statistical model of lossy links in wireless sensor networks. In *IPSN 2005*.

[6] D. S. J. D. Couto, D. Aguayo, J. Bicket, and R. Morris. A high-throughput path metric for multi-hop wireless routing. In *Mobicom 2003*. ACM, 2003.

[7] S. E. Czerwinski, B. Y. Zhao, T. D. Hodes, A. D. Joseph, and R. H. Katz. An architecture for a secure service discovery service. In *Mobile Computing and Networking*, pages 24–35, 1999.

[8] K. Fall. A delay tolerant network architecture for challenged internets. In *Proceedings of SIGCOMM 2003*.

[9] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin. Emstar: a software environment for developing and deploying wireless sensor networks. In *USENIX Tech. Conf. 2004*.

[10] L. Girod, M. Lukac, A. Parker, T. Stathopoulos, J. Tseng, H. Wang, D. Estrin, R. Guy, and E. Kohler. A reliable multicast mechanism for sensor network applications. in CENS Technical Report 48.

[11] L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson, D. Estrin, E. Osterweil, and T. Schoellhammer. Tools for deployment and simulation of heterogeneous sensor networks. In *Proceedings of SenSys 2004*, November 2004.

[12] R. Govindan, E. Kohler, D. Estrin, F. Bian, K. Chintalapudi, O. Gnawali, R. Gummadi, S. Rangwala, and T. Stathopoulos. Tenet: an architecture for tiered embedded networks. In *CENS Technical Report No. 53*.

[13] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building efficient wireless sensor networks with low-level naming. In *SOSP*. ACM, 2001.

[14] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *ASPLOS-IX*, 2000.

[15] W. Hu, V. N. Tran, N. Bulusu, C. tung Chou, S. Jha, and A. Taylor. The design and evaluation of a hybrid sensor network for cane-toad monitoring. In *IPSN 2005*.

[16] Intel. Intel stargate, http://platformx.sourceforge.net.

[17] D. B. Johnson and D. A. Maltz. Dynamic source routing in ad hoc wireless networks. In Imielinski and Korth, editors, *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.

[18] F. Stann and J. Heidemann. RMST: Reliable Data Transport in Sensor Networks. In *Proc. of the First Intl. Wkshp on Sensor Net Protocols and Appl.* IEEE, 2003.

[19] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler. An analysis of a large scale habitat monitoring application. In *SenSys 2004*.

[20] C. Wan and A. Campbell. PSFQ: A Reliable Transport Protocol For Wireless Sensor Networks. In *WSNA 2002*.

[21] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *MobiSYS '04*, pages 99–110. ACM Press, 2004.

[22] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Sensys 2003*.

[23] N. Xu, S. Rangwala, K. K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin. A wireless sensor network for structural monitoring. In *ACM SenSys 2004*.

[24] M. Yarvis, N. Kushalnagar, H. Singh, A. Rangarajan, Y. Liu, and S. Singh. Exploiting heterogeneity in sensor networks. In *IEEE Infocom 2005*.

[25] J. Zhao and R. Govindan. Understanding packet delivery performance in dense wireless sensor networks. In *Sensys 2003*.

[26] Extensible sensing system: An advanced network design for microclimate sensing. `http://www.cens.ucla.edu`.