A Reliable Multicast Mechanism for Sensor Network Applications *

Lewis Girod Martin Lukac Andrew Parker
Thanos Stathopoulos Jeffrey Tseng Hanbiao Wang
Deborah Estrin Richard Guy Eddie Kohler

Center for Embedded Networked Sensing
University of California, Los Angeles
Los Angeles, CA, 90095 USA
Los Angeles, CA 90095 USA

{girod,mlukac,adparker,thanos,hbwang,destrin,rguy,kohler}@cs.ucla.edu,
jeffreyt@icsl.ucla.edu

ABSTRACT

As the field of embedded networked sensing matures, useful abstractions are emerging to satisfy the needs of increasingly complex applications. This paper demonstrates *StateSync*, an abstraction for reliable dissemination of application state through a multi-hop wireless network. Using StateSync, the complexity of multihop wireless network applications and services can be reduced to processing a gradually evolving set of table entries, subject to minimal consistency checks. The StateSync abstraction defines a data model based on key-value pairs, a reliability model with a probabilistic latency bound, and an event-driven publish/subscribe API. We evaluate StateSync using three different applications. We present performance measurements using simulation and a wireless testbed.

1. THE STATESYNC ABSTRACTION

As the field of embedded networked sensing matures, useful abstractions are emerging to satisfy the needs of increasingly complex applications. This paper demonstrates *StateSync*, an abstraction for reliable dissemination of application state through a multi-hop wireless network.

The StateSync layer presents a publish/subscribe interface to a set of application-defined tables. The contents of these tables are reliably and efficiently broadcasted a specified number of hops away, using a protocol that is robust to changes to the network topology and changes in the receiver set. StateSync conforms to a minimal consistency model for received values published by a single node, but does not attempt to guarantee consistency between received values published by different nodes. Using StateSync, the complexity of the multihop wireless network is reduced to processing a gradually evolving set of table entries, subject to certain minimal consistency checks.

*This work was made possible with support from the NSF Cooperative Agreement CCR-0120778, and the UC MICRO program (grant 01-031) with matching funds from Intel.

1.1 Application Requirements

Embedded networked sensing applications inherit a long list of application requirements that are more or less unique among distributed systems. The main distinguishing characteristic is a high degree of dependence on the environment, in the face of dynamic conditions and a limited capability to discover environmental properties with certainty. Properties of the environment often affect both system performance and the application's objectives, and thus must be estimated to achieve the system's goals. These issues are at the heart of the design of successful system components for embedded networked sensing applications.

We designed StateSync to extend the ideas of previous abstractions [16] and protocols [12] to support of a specific class of applications. These applications have the following properties:

- Reliable delivery greatly simplifies the design of the application.
- A relatively large amount of data is shared, and freshness of the data is important, including assurance that the publisher of data is still active.
- The data being shared exhibits low "churn", meaning that the expected lifespan of a data element is long compared with the system latency requirements.

One example of this type of application is an acoustic localization system. Such a system needs to disseminate range estimates throughout the network in order to fuse them into a coordinate system. These range estimates vary slowly over time in response to minor changes, occasionally changing abruptly when nodes are moved. Reliability is important for this application, because inconsistent or stale data can present problems for the multilateration algorithm. The range data in this application tends to have

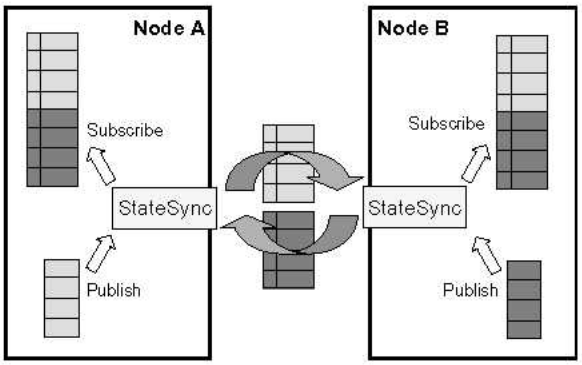

Figure 1: Publisher applications push tables of key-value pairs to StateSync, which disseminates them and delivers the complete table of all received keys to subscribers whenever a change occurs.

long lifespans, going perhaps as long as 10-30 minutes without modification. When modifications do occur, they tend to affect only a small fraction of the data being published by a given node. Despite these long lifespans, low latency is desirable, because additional latency in the propagation of updates directly affects the application-level performance of the localization system by delaying position updates.

Building applications over the StateSync abstraction not only greatly simplifies the implementation of applications, but also provides opportunities for efficiently aggregating application state changes into. Other examples of services and applications that can benefit from this type of layer are routing protocols, configuration and calibration mechanisms, and membership agreement protocols.

1.2 The StateSync Abstraction

The StateSync abstraction defines the data model, the API, and the semantics of StateSync. StateSync imposes a simple data model of typed key-value pairs. The data types are user-defined and can either specify a fixed record and key length, or use variable record and key lengths. The key-value pairs are implicitly annotated with a *flow ID* that includes a unique address for the publisher and other user-definable fields. This additional implicit key effectively assigns each publisher an independent key-space. At most one value is permitted per key: when a pair is published with the same key, type, and flow ID as an existing pair, the original pair is replaced.

The StateSync API presents a Publish/Subscribe interface. A publisher provides StateSync with a complete set of keys for a given type and flow ID to replace all existing keys for that type and flow ID. A subscriber will receive events whenever there is any update in the data matching a specified type. The complete data matching that type can be retrieved from StateSync, combined from all flows that reach the subscriber. Each key-value pair is annotated with the flow ID of the publisher of that data, as well as other metadata such as the arrival time and the distance to the publisher in hops. For fixed-length records, simple arrays of records are passed to and from the API. Figure 1 shows a block diagram of the StateSync API.

The StateSync mechanism provides semantics that are designed to be relaxed enough to be implemented efficiently in a wireless network, while still maintaining useful properties. The StateSync subscribe interface presents only the *most recent* state to a subscriber; it does not present each *intermediate* published state. This policy eliminates the need to retain a backlog or a complete history in the event of lengthy disconnection. In addition, StateSync guarantees that each state presented at a subscriber was in fact an *actual prior state* of the publisher. That is, the view at the subscriber is *never* a partial state of the publisher (such as would occur if a sequence of updates were played out of order). Third, the latency with which a state propagates from publisher to receiver conforms to a probabilistic latency bound that is a function of the number of hops, the size of the transfer, and timers in the implementation. StateSync deliberately relaxes any guarantee of consistency across disparate publishers. Consistency *is* guaranteed across the set of receivers of a given published state, after no change has occurred for the expected latency bound for the farthest node.

1.3 Related Work

The design of StateSync builds on the observations and experience of many past and present systems in sensor networks. The importance and value of a neighborhood abstraction was clearly laid out in the discussion of Hood [16]. Hood provides a way to approach several important concepts about neighborhoods, and provides a best-effort transport layer. StateSync provides a similar API to Hood, but extends its scope by defining a model that includes reliable delivery over multiple hops. The Hood and StateSync solutions in some ways address orthogonal application properties. Whereas Hood is designed to share ephemeral data in a best-effort fashion, StateSync is designed to share long-lived data with very low quiescent cost. Each of these solutions advances a significant space of applications.

Relative to much prior work that present very generalized solutions to problems in distributed systems, StateSync defines a narrower set of properties, which nonetheless represent a large application space. The StateSync API draws upon prior experience with Publish/Subscribe interfaces in the context of Directed Diffusion [11] and other early work in Sensor Networks. However, StateSync imposes more structure than a simple raw data interface, providing an interface supporting application-defined fixed-length tables. The StateSync data model of typed key-value pairs draws on experience with Tuplespace systems such as Linda [6]. However, StateSync relaxes most of the locking and group consistency semantics, because group consistency is generally too heavy-weight for the wireless networks StateSync is designed to support. The StateSync implementations build upon Diffusion Trees and upon work in reliable multicast [5], but encapsulate most of the protocol details behind an interface that is fairly implementation-independent.

StateSync's focus on maintaining a low quiescent cost of state synchronization bears much resemblance to the Trickle [12] protocol for code update on TinyOS motes. In implementing our algorithms, we focused on low latency operation, efficient support for many concurrent publishers, and prompt detection of the disappearance of a publisher. Trickle is designed for higher latency tolerance, and while Trickle can support multiple trees, the costs scale with the number of trees. The "polite gossip" mechanism of Trickle is a very effective way to reduce quiescent cost of maintaining state, but unfortunately the savings is incompatible with detecting source disappearance.

2. VARIANTS OF STATESYNC

In our exploration of the StateSync abstraction, we developed several variants of varying complexity and with different performance characteristics in terms of latency and network traffic. Since each variant conforms to a common API, we can readily compare them in the context of different applications.

In this section we present three StateSync variants, in increasing order of sophistication: *SoftState*, *LogFlood*, and *LogTree*. *SoftState* is a very simple implementation based on periodic re-flooding of the complete state with no retransmission mechanism. *LogFlood* introduces a log mechanism to enable publication of updates to existing state and implements a local retransmission protocol, while using a flooding mechanism to push data with low latency. *LogTree* introduces an overlay network consisting only of the most reliable bi-directional links, and forms distribution trees via that overlay. These variants are discussed in more detail in the following sections.

2.1 SoftState

SoftState implements a periodic refresh of the complete state published by each node. Each refresh is transmitted via a best-effort flooding service and is received by nodes a specified number of hops away. If the complete state is larger than a single MTU, the message is fragmented and reassembled across each hop. No other form of reliability is implemented, so as the state size grows the latency of *SoftState* increases rapidly. The latency of updates is a function of the refresh interval and of the probability of message loss, which is in turn a function of total state size.

SoftState is a very simple variant of StateSync with numerous drawbacks—for example, its quiescent cost is high for most applications. However, it *is* sufficient for some applications, and it can be readily implemented on low-end platforms. An application that publishes only small amounts of data and can accept the bandwidth / latency tradeoff can use this protocol. *SoftState* is also appropriate for applications with high "churn" relative to latency requirements. If the expected lifetime of the data being published is on the order of the required refresh interval, then there is little to be gained by transmitting only the portions of the state that have changed.

2.2 LogFlood

The *LogFlood* variant introduces two important mechanisms that enable higher efficiency and allow StateSync to be applied to a much larger space of applications. The first is a *log mechanism* that stores and transmits published data in the form of a log of additions and deletions of key-value pairs. This log enables the data to be broken down into small segments and transmitted and re-transmitted piecemeal. The second is a *local retransmission protocol* that can request missing segments from a neighbor based on sequence numbers. In the following sections, we will show that these two mechanisms enable much larger amounts of state to be transmitted efficiently.

2.2.1 The StateSync Log Scheme

As we have described in Section 1, StateSync is based on a key-value data model and the API is tuned to support tables of fixed length key-value pairs. These design decisions fit neatly into a log-based transport scheme, because they enable the application to define the granularity at which changes typically occur, and specify precisely which parts of the existing state need to be re-transmitted.

The StateSync log scheme is designed to provide correctness with low overhead and to support a continuous stream of log entries. The StateSync log is composed of a sequence of variable-length entries containing a 16-bit sequence number and a command field. The first entry is always an **INIT** command, and has sequence number 0. The **INIT** message contains a 64-bit log sequence number that is chosen randomly by each node on boot and is incremented whenever a new log is created. This sequence number is used to protect StateSync against inconsistency from reboots or stale data.

Following the **INIT** command, a sequence of **ADD** and **DEL** entries represent the addition and deletion of keys. An **ADD** entry adds a new key and value to the state published by a given node, replacing any previous entry with the same key. A **DEL** entry removes an existing key and value from the published state. Additional command types are used to fragment large entries that might otherwise exceed the network MTU.

Unlike protocols like TCP that use byte ranges, sequence numbers in a StateSync log are assigned at the granularity of log entries. The reason for this design choice is twofold. First, sequencing at a larger granularity reduces the required size of the sequence numbers, and thus reduces protocol overhead. Second, by always transmitting whole entries rather than byte ranges, the log entries can be processed by the application out of order, as in application layer framing [5]. The drawback of this scheme is that, unlike the case of IP fragmentation, StateSync log entries cannot be adaptively fragmented "in flight". Instead, a predefined granularity must be selected at design time, taking into account the MTU of the networks in the system and the expected size of the values published by the application. While the choice of granularity can impact the utilization of packets, in practice we have been able to use a single default value for all of our development.

The other key design problem for the StateSync log mechanism is how to address the problem of an infinitely growing log. While **ADD** and **DEL** commands often make a previous log entry redundant, those redundant log entries cannot be deleted without forfeiting the semantic requirement that StateSync subscribers always see a valid past state of the publisher. In addition, as state changes occur, an increasing fraction of the sequence space will be consumed by redundant entries. Given StateSync's relatively small 16-bit sequence numbers, this can lead to sequence number exhaustion. To address this we apply a solution similar to the "new page" abstraction implemented by the WB application[5].

Each StateSync log maintains two sub-logs: a *checkpointed* log and an *active* log, as shown in Figure 2. New additions to the log are always appended to the *active* log. When certain conditions are met—such as a maximum level of redundancy in the log—the *active* log is "checkpointed". A special **TERM** command is appended to the *active* log, and it is rotated into the *checkpointed* slot. A new *active* log is formed by incrementing the log sequence number and compressing the previous *active* log, renumbering the entries starting from sequence 0.

The checkpointing process addresses the problem of infinite logs at minimal cost. The only cost of the scheme is an additional **TERM** entry; once the terminated log is

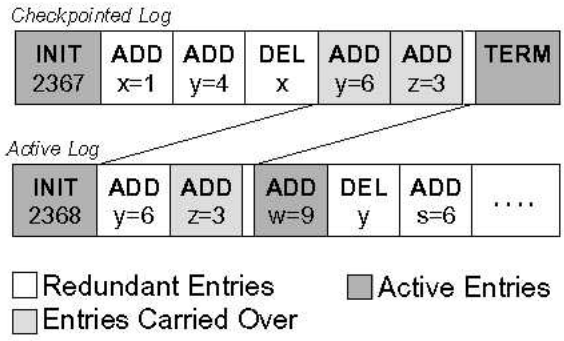

Figure 2: The StateSync Log Scheme maintains a *checkpointed* and and an *active* log. In the diagram, the first two ADD entries in the active log are carried over from the checkpointed log after the redundant entries have been compressed out.

received completely, the checkpointing process is a local operation that does not require any additional network traffic. As an optimization, StateSync will queue out-of-order entries that pertain to the new *active* log before checkpointing is complete.

2.2.2 The StateSync Retransmission Protocol

Once the state data is organized in a sequenced stream of small blocks, we can implement a local retransmission protocol. Similar to many reliable multicast protocols, StateSync's retransmission protocol is receiver-driven with proactive broadcast as an optimization [5]. Receivers add received entries to their logs and maintain state about which log entries are missing based on sequence number gaps. Receivers then schedule NACK requests for specific missing sequence ranges, with an initial delay followed by an exponential backoff. Optimizations such as NACK suppression and more sophisticated timers such as in [5] are not currently implemented.

The wire protocol used by StateSync is designed to be efficient in terms of network usage and flexible in terms of packet structure. The packet format does not have a predefined header structure, but rather is composed of a series of variable-length entries, similar to other proposed wire formats [10][1][11]. As a result, this flexible structure exhibits lower overhead and is also more amenable to piggybacking on other traffic. The wire protocol incorporates numerous optimizations, such as the ability to define a length field that applies to several subsequent log entries, or a sequence number that applies to several subsequent NACKs. For example, the overhead of sending 20 sequential range entries in our acoustic localization application is 25 bytes beyond the 400 bytes of data.

2.2.3 LogFlood Multihop Implementation

Given the log and protocol mechanisms described above, the *LogFlood* multihop implementation is straightforward. First, the retransmission protocol is extended to include the flow-id and the current hopcount of the successive data. The flow-id identifies the publisher-subscriber pair and any additional de-multiplexing bits. In this case, the publisher is identified by a network-layer address and the subscriber is always "broadcast N-hops". Because of the flexible structure of the wire protocol, entries from multiple flows can be packed into a single message.

With this minor change, a simple state machine can implement the multihop flooding protocol. Incoming messages are parsed to extract the flow they pertain to, the hopcount, and the log entries comprising the data. Any messages that are not already present in the log and that are not beyond the maximum desired hopcount are scheduled for retransmission. The hopcount of a flow is determined by recording the lowest hopcount of incoming messages on that flow, and adding 1. When the next transmission is scheduled, any outgoing entries are concatenated with their flow-id's and hopcounts into a single packet and broadcast out to neighbors.

This simple state machine, in addition to the local retransmission protocol, implements an efficient many-to-many flood that can piggyback floods from different sources onto the same packets. However, it is not guaranteed to be reliable; if the last packet is lost, the retransmission protocol cannot discover that there is a sequence number to NACK. To solve this issue, *LogFlood* also floods a periodic refresh message, beginning a fixed time after the last new log entry was flooded. These messages are small and can be piggybacked as described above, but still represent a significant quiescent transmission overhead. They also place limits on join latency. The quiescent cost scales roughly as nk where n is the total number of nodes and k is the number of nodes in the flood radius, The join latency is determined by the refresh rate.

2.3 LogTree

The *LogTree* variant builds on the log scheme and local retransmission protocol described in Section 2.2. However, where LogFlood used a flooding protocol for proactive dissemination and end-to-end reliability, *LogTree* implements a distribution tree for each publisher in order to reduce redundant transmissions without significantly impacting latency. *LogTree* also reduces the quiescent cost of the reliability mechanism to 1 message per node per refresh interval, compared with k messages per node for *LogFlood*. To accomplish this, *LogTree* introduces an underlying layer called *ClusterSync*.

2.3.1 ClusterSync

The *ClusterSync* mechanism serves two functions. First, it estimates the topology of the network and constructs an overlay network consisting only of links that meet certain criteria. Second, it provides a single-hop version of StateSync, with the same API and semantics.

To form the overlay, *ClusterSync* uses a link estimator and periodic beacons to discover the topology of the network and to continuously estimate link quality. It uses a link estimator called *RNPLite* that consumes one additional byte of overhead per packet and computes link estimates based on the principles in [2]. *ClusterSync* uses the link estimates to select links for the overlay that meet certain criteria, including bi-directionality, a minimum link quality metric, and a connectivity metric that prefers neighbors with distinct neighborsets.

The single-hop version of StateSync uses the same log scheme and retransmission protocol as other versions of StateSync. End-to-end reliability is achieved by each node periodically including its latest sequence number in the beacon message it sends for link estimation. When other *ClusterSync* traffic is present, beacon messages and sequence numbers are

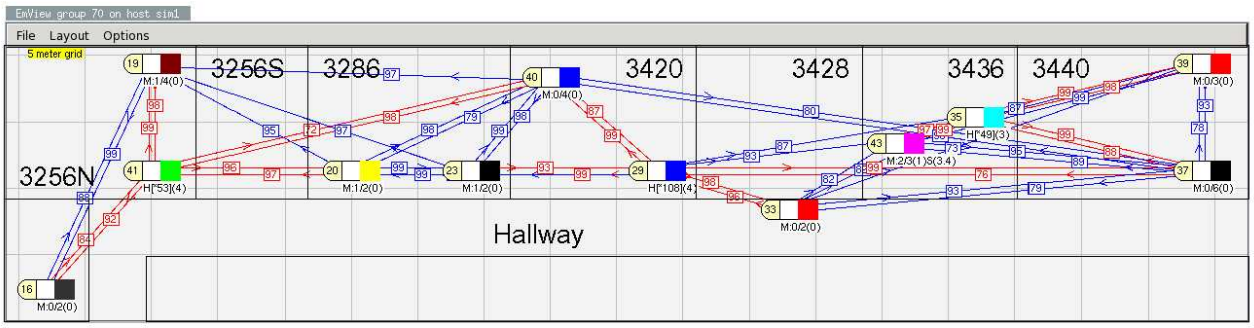

Figure 3: A screen shot from our graphical visualizer displaying the current state of the wireless testbed deployed in our building. The scale of the map is 5 meters per grid square.

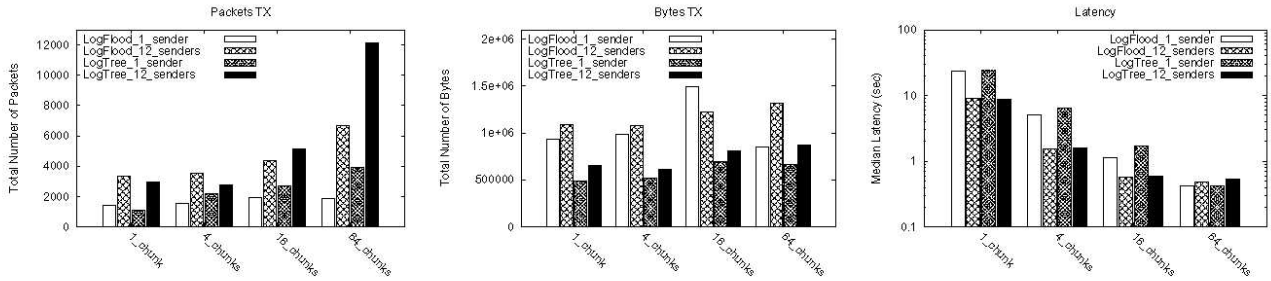

Figure 4: Results of benchmark tests on the testbed. Each grouping of bars represents four 20-minute experiments in which 64K of data is published in a fixed number of chunks, issued at regular intervals. The four experiments in each grouping consist of two runs each of *LogFlood* and *LogTree*, first with 1 publisher and then with 12 publishers.

piggybacked on existing traffic.

The *ClusterSync* mechanism has many advantages that pertain to applications. Many applications benefit from *ClusterSync*'s stable overlay network and prompt detection of topology changes. While topology information is not always necessary to the correctness of an application, it often simplifies the application and results in greater responsiveness (see Section 4.2 for such an application). The stable overlay also presents a more stable definition for the hopcount used to limit the scope of state dissemination. From an application's perspective, it is often more important that the receiver set be stable than that they be a specific "distance" away. *ClusterSync* also provides an efficient way to reliably disseminate state variables to immediate broadcast neighbors. *ClusterSync* will provide the greatest performance improvement to applications that need to publish keys with *long lifespans*, since the up-front cost of reliable transfer is amortized by higher efficiency during quiescent periods.

2.3.2 *LogTree*

LogTree is a multihop StateSync variant that builds distribution trees to yield a performance improvement over *LogFlood*. It builds its trees in the overlay topology constructed by *ClusterSync*, and uses *ClusterSync* to publish routing and flow metadata.

LogTree implements a distance vector algorithm that computes a route and a number of hops back to each publisher. The route to each publisher is used to select a peer for requesting local retransmissions and the hopcount to the publisher is used to determine whether or not to proactively forward new data. Each node also advertises its "preferred" upstream peer for transmission, which is used to prune the proactive distribution tree. All of this routing metadata (i.e. flow ID, hopcount, and preferred upstream peer) is published to adjacent nodes in the overlay network through the *ClusterSync* mechanism. Because *ClusterSync* is reliable, *LogTree* only needs to process updates from *ClusterSync* and keep pushing its most recent routing state back to *ClusterSync*. The *ClusterSync* layer handles all of the complexity of message loss and of timing out stale data and stale neighbors.

LogTree implements end-to-end reliability using a similar mechanism. In addition to the other per-flow routing metadata, a log sequence number is published via *ClusterSync*. This sequence number propagates along with the distance vector messages to inform all nodes of the most recent sequence number published by the source node. In order to limit the traffic pushed through *ClusterSync*, *LogTree* sets a 5 second holdoff timer after each new data element is pushed before pushing a new sequence number out via *ClusterSync*. This information enables nodes to request retransmissions in the event that the most recent message of the log was lost.

2.3.3 *Optimizations to LogTree*

Our experiments with *LogTree* show that it outperforms *LogFlood* and *SoftState* in terms of total volume of data transferred, and does not suffer that much in terms of latency (see Section 3). However, in order to achieve these

results we implemented two optimizations: *flooding mode* and *flow-ID compression*.

Flooding mode addresses the startup latency of *ClusterSync* and of building distribution trees. The original *LogTree* implementation suffered latency problems in the event that the overlay network had not yet formed, or when the distribution tree for a particular source had not yet been constructed. To address this, we modified *LogTree* to proactively flood messages in cases where hopcount was not yet reached and neighbors were observed that did not report an active tree. This optimization achieves similar latency to *LogFlood*, while only incurring the bandwidth penalty as the distribution tree is still being constructed.

Flow-ID compression is an optimization that allows the routing metadata to scale better as the number of distribution trees grows. Each node defines a dictionary that locally maps flow-IDs to small integers, and publishes this dictionary through *ClusterSync*. This enables a full 12-byte flow-ID to be replaced by a 1-byte nickname, reducing the size of published route metadata and reducing the size of headers on data messages that pertain to a given flow. This technique might be applied to other nicknaming problems, although it can increase join latency as the complete dictionary must be replayed to new neighbors.

3. EXPERIMENTAL DESIGN

In this section, we describe how we measure the performance of our StateSync variants, both in a set of benchmark tests, and in the context of running applications.

3.1 Metrics and Experimental Setup

Our criteria are primarily focused on two metrics: the distribution of latency in state propagation, and the network traffic incurred by our mechanisms. The latency is determined by logging the activities of the application or benchmark, matching up publish states with subscribe states, and logging the time lag. Network traffic is determined by measuring the number of bytes and packets that pass through the network interface, and in some cases by measuring statistics gathered directly from the mechanisms.

Our measurements were taken from simulations and tests on a wireless testbed. The testbed experiments were run from a centralized server with remote connections to a set of 12 802.11 radios hosted by Stargates distributed throughout our building, as shown in Figure 3. The simulations were run within the EmStar [7] environment on a typical workstation. Simulations of the Localization and Sink Tree applications were also run with a larger, 50-node topology. For validation purposes, we also ran simulations using the same topology as the testbed experiments, and found that the differences were negligible.

3.2 Benchmark Tests

In order to characterize the abstract performance of our different mechanisms, we ran a series of benchmarks. The results of those benchmarks are shown in Figure 4.[1] These benchmarks are intended to measure the performance of the *LogFlood* and *LogTree* variants when driven with simple workloads. Each experiment lasted 20 minutes, and published 64K of data via StateSync. The only difference from one experiment to the next was the distribution of the data in time (i.e. when it was published) and the number of nodes involved in publishing.

In the first set of experiments, only one node published data and we varied the number of "chunks" the data was broken into. Each chunk was published at a uniform division of the 20 minutes. From Figure 4, we can see that *LogTree* always sends fewer bytes and generally achieves comparable latency.

In the second set of experiments, all 12 nodes in the network published at each interval, dividing the same total amount of data among them. With 12 senders, both variants incur greater traffic cost, but we see *LogFlood* degrade more rapidly than *LogTree*.

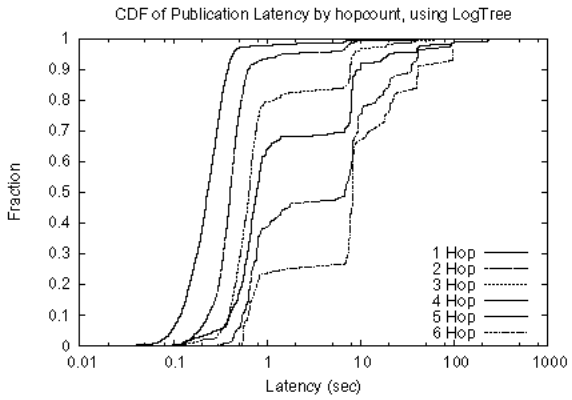

Figure 5: The latency distribution by hopcount.

This benchmark data also provides us with some idea on how to model latency as a function of the amount of data being pushed and the number of hops. From Figure 4, we see that the latency scales roughly linearly with the size of the input data. This result was expected, given the various forms of rate limiting implemented in the local retransmission protocol. In addition, Figure 5 shows that the distribution of latencies also increases as a function of the number of hops from the publisher. The bimodal distribution in latency reflects the probability of a loss that results in additional delay before the 5 second holdoff timer expires and the new sequence number is pushed.

3.3 Determining Application Suitability

Latency and traffic consumed are a good starting point for determining whether StateSync is helpful to an application. StateSync is most appropriate to applications that need notification when state is stale or when the source of some data has disappeared. In cases such as these, epidemic protocols are not appropriate because they will mask stale data; the only possible solution is some kind of refresh mechanism. The advantage of StateSync is that it defines a reliable transport that can protect a large collection of state variables with a single aggregate refresh.

To quantify an application's needs we characterize the application using two metrics: the application's specific latency requirements, and the level of "churn" in the applica-

[1]While *LogTree* sends less data than *LogFlood*, it sends a *larger* number of packets. This occurs because the current implementation *ClusterSync* sends its own independent packets rather than piggybacking them on other traffic. Although it is not yet fully implemented, *ClusterSync* was specifically designed with piggybacking in mind and support for piggybacking is currently under development.

tion's data, defined by the expected lifetime of a key-value pair. If the expected lifespan of application data is low enough compared with the required latency bound, then a simple periodic refresh may be cheaper than the expected cost of a reliable transmission protocol. However, if the lifespan of application data is likely to be much longer than the latency with which stale data is to be detected, then the additional overhead of a reliable protocol is justified. This argument holds true to an even greater extent in cases where the quantity of data being refreshed further increases the cost of refresh. Figure 6 shows the distribution of key lifetimes for the three applications we will discuss in the next section.

4. APPLICATION PERFORMANCE

We developed three embedded networked sensing applications that use StateSync so as to ascertain the merits of our abstractions and test our implementation. We then measured the performance of each application and characterized their use of the StateSync layer. While we do not claim that this exhausts the space of possible applications, it does provide some insight into application requirements that impact the design of StateSync.

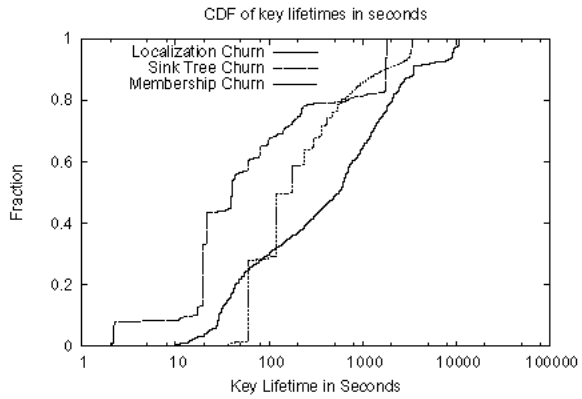

Figure 6: The distribution of key lifetimes for all three applications. The mean key lifetimes are: Localization: 1506 ± 121; **Mote Membership:** 538 ± 36 ; **Sink Tree:** 388 ± 97.

The next three subsections describe our applications in more detail. In each case, we first explain the intent of the application and how it uses StateSync, and then show the results of tests in simulation and from a wireless testbed.

4.1 Acoustic Localization

Localization is an important problem in embedded networked sensing systems, which has been discussed extensively [14][4][3][15][8]. Localization is an interesting problem from a networking perspective because it is an intrinsically collaborative application. The data required to estimate node locations are measured at many points in the network and must be fused together to estimate a consistent map of locations. In addition, inconsistencies and gaps discovered in the data fusion process can be fed back to adaptively drive the measurement process.

Because the signal processing and estimation components of the localization system are already fairly complex, the StateSync abstraction is a powerful tool. Using StateSync, the complexity of the multihop wireless network is reduced to processing a gradually evolving set of table entries, subject to certain minimal consistency checks.

4.1.1 Application Characteristics

The acoustic localization application is well suited to the properties of StateSync. Typical large deployments of this type of system yield between 10 and 20 ranging records per node [13], resulting in 400 bytes of published data per node. The curve for Localization in the "churn" graph in Figure 6 shows that these ranging records tend to have long lifespans, meaning that simple soft-state refresh approaches will be costly over time compared with mechanisms that can reliably cache the data. At the same time, low latency is desirable because the latency in state update directly affects the length of time that the system operates with incorrect position information, which in turn could lead to sensing and actuation based on inaccurate location estimates.

4.1.2 Mapping to the StateSync API

Our acoustic localization application consists of two components. The ranging component computes the physical distance and orientation of a node relative to a peer by measuring the time of flight of acoustic signals. Range and orientation estimates are published via StateSync to all nodes within a predefined hopcount. The multilateration component subscribes to all published range estimates, and uses a multilateration algorithm to fuse the range estimates into a consistent coordinate system.

4.1.3 Applying the StateSync Model

The reliability and consistency model of StateSync is used to ensure consistency in the datasets that are fed to the multilateration algorithm. In the event that a node is rotated or moved, the ranges and direction estimates relating to that node are no longer valid. This in itself is not a serious problem, as it will only result in estimating the node's location as its last location. However, if further ranging experiments lead to a mixture of old and new range and orientation estimates, these inconsistencies are likely to cause the multilateration algorithm to fail.

In our application, this problem is addressed using a per-node "orientation sequence number" that is incremented each time the node moves or otherwise invalidates its ranges. The ranging component indicates its current orientation sequence number when it requests peers to range to it. This enables nodes that receive acoustic range signals to annotate their published estimates with the sequence number that was in effect at the acoustic sender at the time that the experiment occurred. Published estimates are also annotated with the publisher's sequence number, indicating that those estimates are relative to their current position. Whenever a node increments its sequence number, it deletes all ranges it had previously published, and then publishes its new sequence number.

In spite of StateSync's relatively loose consistency semantics, this protocol enables the multilateration component to maintain a consistent dataset. To maintain consistency, the multilateration component records the current sequence number published by each node, and all published data annotated with other sequence numbers is ignored. The only exception is for range notification messages that arrive with a subsequent sequence number: as an optimization, these messages are processed and published ahead of the arrival

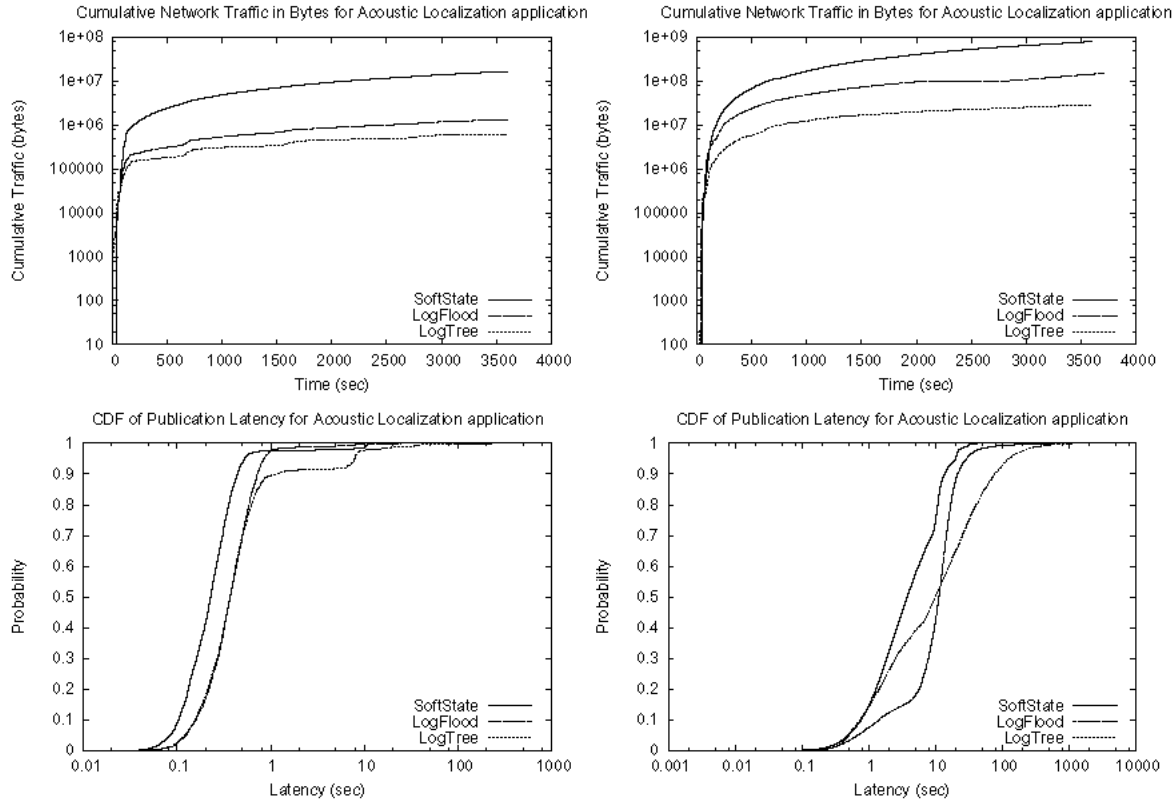

Figure 7: Results of tests of our Acoustic Localization application. The left pair of graphs are measurements from our 12 node testbed, while the right pair are from a 50 node simulation. The latency graphs show a CDF of latency in seconds, demonstrating that *LogTree* achieves an expected latency about twice that of *LogFlood* at lower scales, although with a higher fraction of long latencies at the higher scale. The mean latency for *LogTree* is 31.54 ± 0.58; for *LogFlood* is 14.33 ± 0.12. The "knee" at 5 seconds in the data from the small network is caused by the 5 second holdoff on publishing a new sequence number in *LogTree*.

of a sequence number update from the source node. Because StateSync's semantics guarantee that states from an individual publisher arrive in sequence, the table received from a node can never itself contain inconsistent sequence numbers, and StateSync reliability guarantees that the table update will occur within a probabilistic time bound.

4.1.4 *StateSync Performance*

To test the performance of StateSync when used in our localization application, we ran several tests with different variants of StateSync underneath our application. We ran each test for 1 hour on our wireless testbed, and for 1 hour in the simulator, running a 50 node network. The ranging process was primarily driven by the multilateration algorithm, which would cause 3 range requests in rapid succession, followed by an exponential backoff. In addition, we simulated three of the nodes "moving" by forcing them to invalidate their range information at three particular times. These invalidations result in a burst of ranging activity and cause the backoffs to be canceled.

Figure 7 shows the results of testing our localization application using different variants of StateSync. The graphs show two types of information: the cumulative bytes transmitted throughout the network as a function of time, and the distribution of latency in a published state arriving at a subscriber.

From the graphs we see that in terms of bytes transmitted, *LogTree* performs better than *LogFlood*, both in terms of the amount of overhead during transfers as well as in the rate of traffic during quiescent periods. During transfers, *LogTree*'s pruned distribution tree provides significant savings over flooding, especially as the size of the network grows to reach the maximum hopcount of the published flows. During quiescent periods, *LogFlood*'s periodic re-flood of the latest sequence number is considerably more costly than the *ClusterSync* beacon traffic that refreshes the *ClusterSync* sequence number (that in turn protects the *LogTree* per-flow sequence numbers).

The latency graphs show that *LogTree* has twice the expected latency of *LogFlood*. This can be explained by a number of factors, including higher hopcounts on average than in the flooding case, and lower redundancy when data is sent via the distribution tree. However, for many applications the latency cost is balanced out by the reduction in traffic.

4.2 Sink Tree

Sensor network applications commonly exhibit stylized communication patterns that can be described as many-to-one, or one-to-many. As a result, the creation of sink and source trees are common tasks in sensor network applications. Sink trees are especially efficient for building for-

warding tables in situations where there are few consumers of data relative to the number of potential producers. Similarly, source trees are ideal in the case of few producers with many consumers. We used StateSync to build a sink tree, from which, a forwarding table can be provided to support multihop communication.

4.2.1 Application Characteristics

The data required for creating and maintaining a sink tree builds naturally on the communication abstraction provided by StateSync. Many of the decisions that must be made rely on having the latest available information about nodes within broadcast range, and not on historical information. This memoryless-like property, where the protocol relies only on current state, is one important attribute of applications that are well suited to use StateSync.

Neighbor attributes usually include a combination of local information and cumulative information. Link quality, sleep schedule [18], and geographic location are all examples of local information that have been used in various tree-based routing protocols. Hop-count, expected number of transmissions, and data interest [11] are examples of cumulative information that have been similarly used in tree-based routing protocols. Usually there is some data originating from sinks that periodically propagate throughout the network to maintain the sink tree, such as a sequence or epoch number.

Timeliness is also important, as it can mean the difference between a functioning and non-functioning routing protocol. The exact latency requirements depend on the expected characteristics of the application and environment. Sufficiently late information is worse than no information, as it can lead to oscillations or routing loops.

4.2.2 Mapping to the StateSync API

Our sink tree implementation uses a distance-vector algorithm. For illustrative purposes, we chose a simple hop-count metric, though it would be easy to substitute this with any other metric. To avoid loops and the count-to-infinity problem, nodes provide the exact path used to reach the sink. By advertising the path, in addition to a hop-count, neighboring nodes can discard next-hop candidates if they see themselves appear in the path. A similar technique is used in BGP for inter-domain routing.

Using StateSync allows us to concentrate our development efforts on simple operations over a table of key-value attributes. In particular, we do not use the multi-hop features of StateSync for this application. Instead, we are only interested in the state of nodes that are within broadcast range.

Nodes publish a table that reflects the path used to reach the nearest sink. Each entry in the table corresponds to a hop along that path. The *key* for an entry is the hop number, and the *data* is the Node ID at the hop number.

Nodes periodically publish table updates based on a timer. This is done to avoid correlated bursts of control traffic that are triggered by a single event, especially if it occurs near the sink.

4.2.3 Applying the StateSync Model

The *LogTree* variant of StateSync provides a number of additional features that ease the implementation of Sink Tree. *LogTree* will only synchronize with a subset of well connected neighbors. In particular, this means that data from nodes with intermittent connectivity will never be seen by applications using *LogTree* over a single hop. Further more, *LogTree* will proactively remove table entries from nodes that disconnect from the overlay. These properties relinquish applications like Sink Tree from the burden of identifying and rejecting potentially large numbers of unsuitable candidates due sub-par connectivity.

LogTree further provides applications with the benefit of implicit liveness information. The resulting forwarding table from Sink Tree is more stable and automatically provides routes high hop-by-hop reliability. The trade-off is that the Sink Tree will be limited to a sub-graph of the overlay network formed by *LogTree*.

LogFlood and *SoftState*, in contrast to *LogTree*, opportunistically uses all data it over-hears from any node in order to gather and update state. For some applications, such as Sink Tree, this behavior may require that applications incorporate link-quality and liveness information through some other means.

4.2.4 StateSync Performance

We tested different implementations of State Sink using the Sink Tree application. Each test was run for 30 minutes on our wireless testbed, and 30 minutes on a 50 node simulated network. The 50 node network was evenly distributed within a square field at the same density as the testbed. In each of the tests, four randomly chosen nodes were configured to advertise themselves as sinks throughout the duration of the test. Dynamics in the system were thus driven by the temporal variations of link quality.

Figure 8 shows the results of testing Sink Tree with different implementations of StateSync. As in Figure 7 for Acoustic Ranging, we show the cumulative bytes transmitted over time, and the distribution of state update latency.

Consider the top two cumulative traffic graphs in Figure 8. We first focus on the curves of *LogTree* and *LogFlood*. The difference in cumulative traffic here can be attributed to a number of things. First, the basic overhead in the quiescent state for *LogFlood* is strictly greater than *LogTree*. The reason is that *LogTree* is establishes session state with its neighbors, which allows them to agree upon short nicknames that only is meaningful within the context of the shared session state. Conversely, *LogFlood* must always communicate state using longer, globally unique identifiers. *LogTree* shares state with a smaller number of neighbors, which means that the consequences of a state change at a single node is smaller in scope compared to *LogFlood*. Therefore, given the same amount of dynamics in the system, fewer state updates will be triggered in *LogTree* relative to *LogFlood*.

We next consider the difference in cumulative traffic between *LogFlood* and *SoftState* in Figure 8. A node's sphere of influence in these two schemes is identical, so the primary differentiator is protocol overhead. *LogFlood*'s sequence number incurs an overhead that is constant with respect to the table size, where as *SoftState*'s overhead grows linearly. In the case of Sink Tree, however, the entire table published by a node is typically smaller than than the globally unique sequence number used by *LogFlood*, enabling *SoftState* to perform better.

In Figure 8, the latency graphs show that *LogTree* suffers from having a higher fraction of long delays compared to *LogFlood*. This is similar to the results found in Figure 7, but the reasons are different. *LogTree* is penalized heavily when a node connects to a different neighbor due to a topology change in the overlay network. All of the existing table

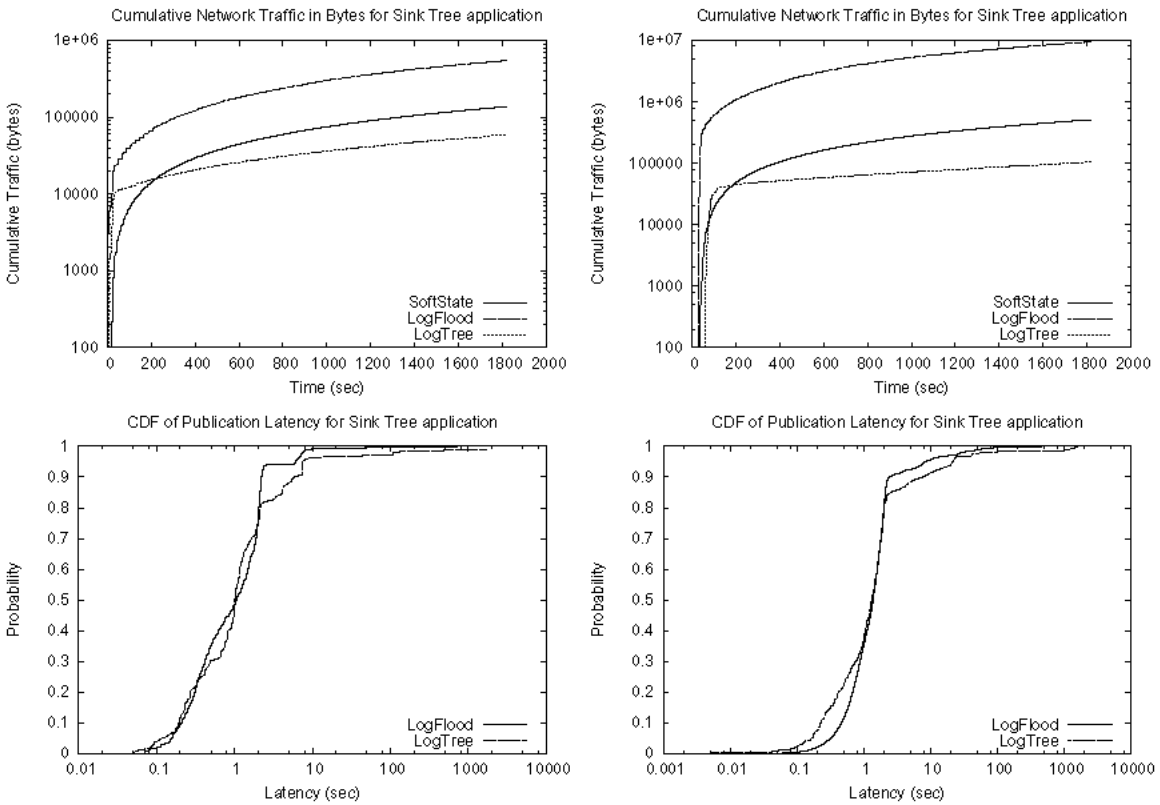

Figure 8: Test results of our Sink Tree application. As in Figure 7, the left pair of graphs are measurements from our 12 node testbed, and the right pair are from a 50 node simulation. The latency CDFs demonstrate that the latencies observed with *LogFlood* tend to be a little lower than *LogTree*, though both scale similarly with respect to the number of nodes. The upper two cumulative traffic CDFs attest to the advantage of using *LogTree* over any of the other techniques.

entries available from the new neighbor are considered to be *delayed* by the age of the table entries, rather than from the time that the two neighbors connected to one another. Since Sink Tree uses StateSync in single-hop mode, then anytime a new edge is added to the overlay network, a huge price is paid in terms of latency statistics. This is not true for Acoustic Ranging. Adding a new edge does not necessarily connect two nodes that have no state about the other, since Acoustic Ranging is using State Sync over multiple hops.

4.3 Mote Membership

The Berkeley/Crossbow mote is the most popular low-power, resource-constrained wireless sensor network platform. Since the radio range of the motes is limited, the vast majority of mote applications use a *multihop* tree-based routing protocol (e.g. Surge [17]) to send the data back to the sink. A single multihop tree (rooted to a single sink) is sufficient when the network topology is small. However, in larger deployments, or when the network traffic is significant, it is beneficial to have multiple trees, in order to reduce latency and increase reliability. When multiple sinks are present, the motes will pick the "best" one (based on a routing metric, like ETX) to send their data to.

A sink, being the root of the mote multihop tree, has information about all the motes that belong to that tree. However, it has no information about the rest of the motes in the network; those motes report only to their corresponding sinks. In order to acquire the complete membership information, one needs to query each individual sink and concatenate the results. For the purposes of system monitoring as well as sink-to-mote routing, it would be helpful if *each* sink had the *complete* mote membership information, consisting of its local motes as well as remote motes, attached to the other sinks.

The *Mote Membership* application is designed to accomplish the aforementioned goal. Each mote registers with its sink when it joins that sink's multihop tree. If the mote changes multihop trees, it registers with its new sink. The sinks keep track of their local motes through the mote multihop routing protocol and *export* that information to the rest of the sinks, using StateSync.

4.3.1 Mapping to the StateSync API

Each mote registers with its sink when it joins that sink's multihop tree. If the mote changes multihop trees, it registers with its new sink. Each sink has a *membership table* that is comprised of *local* entries as well as *synced* entries. The local entries are the motes that have joined this sink's multihop tree. Each sink periodically *publishes* the part of its table that contains the local entries, using the StateSync API. Upon receiving an exported table, the other sinks merge those membership entries into their own tables. Those entries appear as *synced* entries and also contain the ID of the publisher. This information is sufficient to create

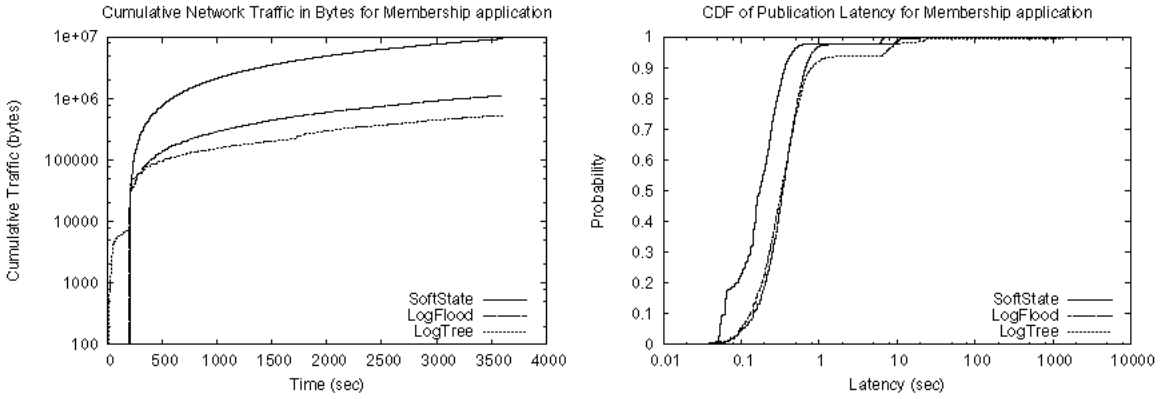

Figure 9: Results of tests of our Mote Membership application, taken from our 12-node testbed. The graph on the left depicts the cumulative network traffic as a function of time, for the three different variants of StateSync. *LogTree* incurs the lowest overhead, followed by *LogFlood* and *SoftState*. The graph on the right depicts the CDF of latency. In this case, *SoftState* is the fastest, while *LogFlood* performs marginally better than *LogTree*. Again, the "knee" at 5 seconds in *LogTree* is attributed to the 5 second holdoff on publishing a new sequence number.

a *global membership view* at *each* of the sinks.

4.3.2 Applying the StateSync Model

The StateSync model guarantees that states from an individual publisher arrive in sequence. However, since motes don't explicitly inform their previous sink when they change multihop trees, a possible inconsistency might occur in which two or more sinks will consider the same mote as local. A sequence number scheme is used to resolve this ambiguity. The sequence number is incremented whenever a mote changes sinks; therefore, larger sequence numbers indicate the most recent choice of the mote. By including the sequence number in the data that each sink publishes, StateSync guarantees that the ambiguity can be reliably resolved within a probabilistic latency bound. As an example, sinks A and B both consider mote 1 to be a local entry. The entry of sink A has a larger sequence number than the entry of sink B. When the information is published via StateSync, sink B will mark node 1 as a *synced* entry, thereby adhering to the sequence numbering scheme.

4.3.3 StateSync Performance

In order to test the performance of StateSync, when used in the Mote Membership application we conducted a set of experiments, similar to the other two applications. Each of the experimental runs was conducted on our testbed, using 12 Stargate nodes. However, unlike Acoustic Localization and Sink Tree, Mote Membership also included mote code. The mote part of the Mote Membership application was written in EmTOS [9] and simulated using the EmStar mote simulator model. Thus, the Mote Membership experiments used a *hybrid* experimental environment, with the sink code running on real hardware and the mote code running on the simulator. This feature of EmStar enabled us to run experiments with 125 simulated motes.

Figure 9 presents the results of our experiments. Again, we are interested in measuring the performance of different StateSync variants in terms of transmission overhead and latency. As in the other two applications, we notice that *LogTree* has the best performance in terms of traffic overhead, while SoftState has the highest overhead. In terms of latency, SoftState is the fastest. *LogFlood* and *LogTree* while still slower than SoftState still perform adequately—more than 90% of the data has a latency of less than 1 second, which, for the purposes of Mote Membership, is more than adequate. Therefore, based on the result, we can ascertain that using the tree-based StateSync is quite beneficial to Mote Membership; it induces the least traffic overhead while still maintaining a rather acceptable latency bound.

5. FUTURE WORK

We see many opportunities for improvements in our StateSync implementations. Our initial version of *ClusterSync* does not piggyback its messages on other traffic, resulting in low channel utilization. Our retransmission protocol may also yield some potential improvements, for example by implementing NACK overhearing and suppression. We also see room for improvement in the algorithms that *ClusterSync* uses to define its overlay (much of which is not described in this paper). In fact, further experimentation is required to get an improved understanding of the true costs and benefits of the *ClusterSync* overlay.

We also have much to learn from an applications perspective. Our existing applications are still in the prototype stage and we continue to evolve the *StateSync* API to meet their needs. There are a number of areas that we have yet to explore with applications, including transport types other than distribution trees (e.g. reliable sink trees and reliable point-to-point connections). We also have yet to fully explore consistency issues and applications traditionally related to consistency such as leader election algorithms.

6. REFERENCES

[1] R. Braden, T. Faber, and M. Handley. From protocol stack to protocol heap: role-based architecture. *SIGCOMM Comput. Commun. Rev.*, 33(1):17–22, 2003.

[2] A. Cerpa, J. L. Wong, M. Potkonjak, and D. Estrin. Statistical model of lossy links in wireless sensor

networks. In *IPSN '05: Proceedings of the Fourth ACM/IEEE International Conference on Information Processing in Sensor Networks*, 2005.

[3] J. Chen, L. Yip, J. Elson, H. Wang, D. Maniezzo, R. Hudson, K. Yao, and D. Estrin. Coherent Acoustic Array Processing and Localization on Wireless Sensor Networks. *Proceedings of the IEEE*, 91(8), August 2003.

[4] K. Chintalapudi, R. Govindan, G. Sukhatme, and A. Dhariwal. Ad-hoc localization using ranging and sectoring. In *INFOCOM '04: Proceedings of the IEEE Inforcom 2004*, 2004.

[5] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Trans. on Networking (TON)*, 5(6):784–803, 1997.

[6] D. Gelernter and A. J. Bernstein. Distributed communication via global buffer. In *PODC '82: Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 10–18, New York, NY, USA, 1982. ACM Press.

[7] 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 *Proceedings of the 2004 USENIX Technical Conference*, Boston, MA, 2004. USENIX Association.

[8] L. Girod and D. Estrin. Robust range estimation using acoustic and multimodal sensing. In *International Conference on Intelligent Robots and Systems*, Oct. 2001.

[9] 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. To appear.

[10] B. Greenstein, E. Kohler, and D. Estrin. A sensor network application construction kit (snack). In *Proceedings of the second international conference on Embedded networked sensor systems*. ACM Press, 2004.

[11] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 56–67, New York, NY, USA, 2000. ACM Press.

[12] P. Levis, N. Patel, D. E. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of 1st Symposium on Networked Systems Design and Implementation (NSDI 2004), March 29-31, 2004, San Francisco, California.*

[13] W. Merrill, L. Girod, B. Schiffer, D. McIntire, G. Rava, K. Sohrabi, F. Newberg, J. Elson, and W. Kaiser. Dynamic networking and smart sensing enable next-generation landmines. *IEEE Pervasive Computing Magazine*, Oct-Dec 2004.

[14] D. Moore, J. Leonard, D. Rus, and S. Teller. Robust distributed network localization with noisy range measurements. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 50–61, New York, NY, USA, 2004. ACM Press.

[15] A. Savvides, C.-C. Han, and M. B. Strivastava. Dynamic fine-grained localization in ad-hoc networks of sensors. In *MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 166–179, New York, NY, USA, 2001. ACM Press.

[16] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *MobiSYS '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 99–110, New York, NY, USA, 2004. ACM Press.

[17] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 14–27. ACM Press, 2003.

[18] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient MAC protocol for wireless sensor networks. In *Proceedings of IEEE INFOCOM*, 2002.