A Probabilistic Replication and Storage Scheme for Large Wireless Networks of Small Devices

Daniela Gavidia
Vrije Universiteit Amsterdam
De Boelelaan 1081a, 1081HV
Amsterdam, The Netherlands
daniela@cs.vu.nl

Maarten van Steen
Vrije Universiteit Amsterdam
De Boelelaan 1081a, 1081HV
Amsterdam, The Netherlands
steen@cs.vu.nl

Abstract

Nodes in wireless ad hoc networks are often limited in terms of resources, such as storage, power, and bandwidth. A downside of this is the fact that local storage at one node cannot accommodate the vast amount of data contained in the network. In this paper, we present SharedState, a scheme for storage, replication, and distribution of common-interest data in wireless networks of resource-constrained devices (e.g. sensor nodes or embedded devices). SharedState works under the assumption that individual nodes would greatly benefit from having access to the wealth of information in the network, but are unable to store it locally at once. SharedState strives to make data available to every node by providing local access to a subset of the whole collection of data items in the network at any moment in time and ensuring that this subset is updated periodically. This is accomplished by probabilistic propagation and replication of data items, ensuring the availability and persistence of information in the face of changing network conditions. We evaluate the performance of SharedState by studying the effectiveness with which nodes can gather information from the network. In addition, we optimize the bandwidth usage of our proposed solution by minimizing unnecessary communication based on feedback from the local neighborhood.

1. Introduction

The usual paradigm for wireless networks consists of wireless nodes - often laptops or smartphones - connecting to a base station in order to access a resource (for example, a data repository or a local printer). Wireless ad hoc networks break away from this model by focusing on the interaction between nodes to create a network on-the-fly, without relying on a preexisting infrastructure. The ad hoc model shuns the centralized approach in favor of operating in a distributed fashion. The resources and services are therefore provided by the nodes themselves, making cooperation between nodes absolutely necessary.

Nodes in a wireless ad hoc network are often mobile, portable, resource-constrained devices. In this paper, we focus on resource-constrained embedded devices designed with specific applications in mind (for example, a wireless sensor network that monitors the presence of people in a building). Unlike phones or PDAs with wireless capabilities, these devices typically use low power RF radios that provide limited bandwidth and communicate through broadcast (Section 3 describes the target platform for SharedState in more detail). The networks that these devices create do not rely on fixed infrastructure for services, but they self-organize to provide certain functionality.

One of the main challenges with wireless devices is that they are inherently unreliable, as they might fall out of reach due to mobility or leave the network unexpectedly. As a result, nodes - and the data they carry - are constantly joining and leaving the network. While most of the data that a node stores locally may only be relevant to the node itself, some nodes might have information that could be of interest to the community in general. Such information may include, for example, configuration information, advertisements or general announcements. Viewing these pieces of information as community knowledge and making them available to the general population would enhance coordination efforts in the network and create a cohesive environment.

The goal of this project is to provide a middleware layer capable of storing data items published by any node in the network and make them available for all nodes. In essence, SharedState acts as a distributed repository of shared data. Unlike a publish/subscribe system, nodes do not subscribe to receive certain information. In our system, all nodes are considered to be possible subscribers. The purpose is then not just to deliver a data item to the interested parties, but to store the data item in the network so that any interested

node could retrieve the item presently or in the future. In other words, the ultimate goal of SharedState is to ensure the availability and persistence of data items of interest to all (or most) nodes.

SharedState acts as a loosely coupled communication platform, such that producers and consumers of data are decoupled in time and space. Consumers can recover a data item from the network when they deem it necessary: producers and consumers do not need to be present at the same time in order to share data. Likewise, they can do so without being within communication range of each other. SharedState takes care of delivering the data items to consumers, who might be located anywhere in the network. Producers are oblivious to any consumer's location.

Contribution Working under the assumption that the information contained in the network greatly surpasses the storage capacity at each node, this paper presents SharedState, a scheme designed to move data items through the network and create replicas of the items at various locations. Specifically, the contributions of this paper are:

- We introduce a scheme for disseminating and replicating data items through the network. This new protocol is characterized by its low complexity and minimal state needed at each node, making it suitable for a wide range of wireless devices.
- We evaluate the efficiency of collecting data items from the network by testing the worst case scenario: each node discovers new items solely by querying its local store. Additionally, we use static topologies in our experiments. The lack of mobility means that data items can propagate only through multiple hops, instead of being carried by mobile nodes to different locations. We show that acceptable discovery rates can be achieved even under these conditions, suggesting that mobility and queries involving surrounding nodes would only improve performance.
- We evaluate our protocol through simulations using TOSSIM. We test the effect of node density on the performance of the system, first using uniform topologies of various densities and then with a non-uniform topology where nodes concentrate at a central point. We show that by allowing individual nodes to decide when to communicate based on neighborhood information, we can take advantage of node density to decrease the number of transmissions by individual nodes. Global knowledge of the network topology and its properties is not required. By relying solely on local information, our algorithm remains effective in larger networks.

2. Related Work

The problem of improving data access and availability in wireless environments has been approached in various ways. One approach is to encode [2, 1] the data into a number of pieces in such a way that the original item can be reconstructed by collecting a subset of the pieces. By distributing the pieces through the network, ubiquitous access is provided. In our case, we assume that the data items we propagate are small read-only data files and we achieve availability by replicating the items. The number of replicas and their location is determined probabilistically.

Cooperative caching for ad hoc networks is another related area. Its aim is to share cached data among multiple nodes by having some nodes host the data and handle requests from other interested nodes [9, 12]. Popular data items are cached at various locations leading to resources being saved by requesting the item from a nearby node. Additionally, access can be obtained even when the original source is unavailable. While cooperative caching also makes use of data replication, it differs from our work in that its aim is to improve the experience of being connected to the infrastructure (Internet). Conversely, SharedState focuses on sharing lightweight data items *created by nodes in the network* that can be disseminated in the background using a limited amount of resources. Instead of a request/forward model, nodes using SharedState discover data items by periodically exchanging them.

Projects focused on data dissemination for ad hoc networks are also relevant to our work. 7DS[8] focuses on allowing access to data available on the Internet, so that when a node's access fails it can get the data from its peers. The types of networks 7DS addresses are different from ours. The authors consider that the network is rarely connected (sparse) and that nodes do not necessarily cooperate. The nodes themselves are more powerful than the ones we consider. While 7DS does implement policies for power management, storage space is not a major concern, as it is for us. In PeopleNet[7], users forward data to pre-defined geographic regions (according to topic) and within each region the system tries to match queries and responses. The nodes in each region become a database for a particular topic, with items being replicated in many nodes. Within each region, data dissemination/replication occurs in a p2p fashion, whenever two devices encounter each other. In RANDI[11], nodes also communicate when they encounter each other, but also proactively if a certain amount of time has passed since the last broadcast. These two systems rely on p2p communication and neighborhood discovery/awareness. SharedState intends to be as lightweight as possible, relying solely on broadcast. There is no need to keep track of the identities of neighboring nodes.

In the realm of wireless sensor networks, data-centric storage (DCS [10]) addresses the storage problem by storing data by type at designated nodes, making data retrieval more efficient. Replication of data at strategic locations has been proposed to improve scalability and robustness [4]. Unlike our work, these approaches require a routing layer and replication is done in a deterministic fashion.

Closer to our work are probabilistic protocols for data dissemination, where the decision to broadcast a piece of data is made locally based on a probabilistic algorithm. Due to their simplicity, these types of protocols are appealing for small devices lacking in computing power. They are also resilient to failures and mobility, which makes them attractive for wireless environments. Probabilistic protocols have been used as an alternative to flooding [5, 3] and for concrete applications like code dissemination [6].

3. System Model

The system we envision consists of a collection of nodes with wireless communication capabilities. Participating nodes are required to contribute resources to the system in the form of storage space. Every node contributes a limited amount of storage space to maintain a local data store of shared information. Nodes access their local data stores to discover previously unseen or interesting items. We will refer to the local data store as the node's *cache* in the remainder of the paper.

The caches are updated periodically with data items broadcast by nodes in the local neighborhood. Items have unique ids and are time stamped when created, allowing the system to keep the latest version of an item by overwriting older versions. Since all data exchanges occur within one hop, routing is not necessary. By relying purely on broadcast, we intend to make the system suitable for a wide range of wireless platforms, including simple and inexpensive devices that use broadcast at the physical layer. Moreover, nodes do not need to keep track of their neighbors.

Communication is limited to periodic updates that are broadcast by each node. Each update message contains a set of data items selected by each node. The frequency with which nodes can broadcast updates is a network-wide parameter, which should be set considering the workload and bandwidth that we desire to allocate for the service.

In addition to the cache, nodes allocate space for an *input buffer* to receive update messages from neighboring nodes and an *output buffer* for the entries to be broadcast. Each node uses the input buffer (which should be, at most, as big as the cache) to accumulate data items received during a fixed period of time, which we call a *round*. At the end of a round, the node updates its cache with the items from the input buffer and broadcasts the set of data items in its output buffer to update its neighbors. We say that a node alternates between two modes: active or passive. Each node takes an active role once per round, when it updates its local cache and decides which items to broadcast. After taking care of these tasks, it falls into a passive mode, where it silently awaits for updates from its neighbors.

We abstract a framework to describe the core structure of a replication and storage protocol like SharedState. There are three main operations (see Figure 1) that a node needs

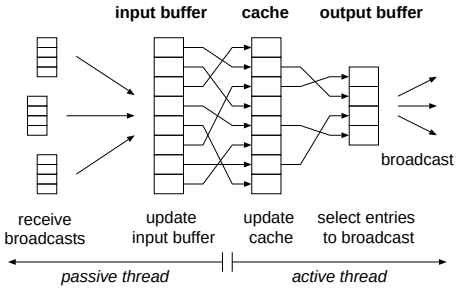


Figure 1. Visual representation.

to execute: a) handle incoming items (passive mode), b) update its cache (active mode) and c) select which items to broadcast (active mode). The specific way in which these three events are implemented has a direct impact on the characteristics of the propagation and replication of items. In Section 4 we describe the implementation details of SharedState which is one particular instance of this framework. Although several alternatives were explored, due to lack of space we will focus on the one that gave the best results during our evaluation.

Target Platform The experimental platform for which SharedState was specifically developed consists of very simple, inexpensive units that integrate a radio, an antenna and an embedded processor in one module. These nodes are meant to be expendable and, as such, they have modest features. Nodes operate on a fixed duty cycle and communication is based purely on the broadcast of small data packets of a fixed size (in the order of tens of bytes). In other words, nodes wake up periodically to communicate and process information and then go to sleep for the rest of the cycle.

The reason for choosing to operate by broadcasting/processing periodically is to have a predictable use of resources, enabling us to taylor the duty cycle and packet size according to the requirements of our applications and the desired lifetime of the network. We imagine that a sensor application (sending an alarm whenever high temperatures are measured, for example) needs to be long-lived and has very small data packets. For this application, the broadcast interval can be set to a value that allows the batteries to last for the desired period (two years, for example). By operating periodically, we eliminate the risk of having nodes run out of energy prematurely due to being located at *busy* spots, as can occur in event-triggered systems. Of course, this comes at a price. The tradeoff is that nodes are required to communicate even when there are no new events to report. This is an acceptable compromise, considering that we aim to deploy very large networks where having clear expectations of the lifetime of nodes is important.

Applications Application areas for SharedState include dissemination of topological information, membership management and service discovery. An example applica-

tion could be asset management, where active tags attached to objects keep track of each other so that logical groups stay together (e.g. a set of boxes in a warehouse or a collection of documents).

4. SharedState

The key to ensuring the availability and persistence of items in the network lies in a strategy of massive replication and relocation of replicas. The replication of items is a natural consequence of the probabilistic methods used for the selection of items to be stored and propagated. As items are propagated, they become available to the nodes who stored them locally. The periodic update of caches ensures that nodes can discover items as they flow through. Discovery is gradual, however, as nodes can only store a limited amount of items in their caches.

The issue of data persistence is critical in a wireless ad hoc network, since nodes may come and go on a regular basis. Whenever a node leaves, the data items it carries disappear with it. For this reason, maintaining a set of replicas per item is necessary. We refer to each replica of an item as an *entry*. While a data item is a piece of information, an entry is the representation of the data item in the network and for each data item several entries may exist. Instead of explicitly trying to maintain a particular number of entries per item, we allow competition between entries to determine the number of entries per item in the network.

Whenever a node broadcasts an entry, there is a chance that it might be replicated if more than one of the node's neighbors decides to keep it. Likewise, whenever a node updates its cache, some entries are discarded due to lack of space. Because of this, the number of entries per item is constantly experiencing variations and, since there is no preference for any particular data item, competition for space in a node's cache is fair. This ensures that each data item has on average the same number of replicas in the network and that the number of replicas adjusts dynamically according to the number of different items published. That is, when there is a large number of different items in the network, each item has few entries. Conversely, when there are few different items present, each item has several entries.

4.1. The Protocol

The way nodes manage their entries depends on the actions they take in their active and passive modes. Figure 2 gives a detailed account of the steps involved in the execution of a node's active and passive thread. Before giving a more thorough explanation of the events that take place in each thread, one distinction between nodes should be noted. Of all the nodes that participate in the system, only a subset acts as a source of data items. That is, at any point in time, only some nodes take the role of producers of information. The only difference between a producer and a consumer is the fact that the producer makes an effort to insert its own

```
/*** Active thread ***/
// Runs periodically every T time units

 1: PHASE I : Update cache
 2: for all entry in inputBuffer do
 3:    if cache.contains(entry) then
 4:       cache.remove(entry)
 5:       inputBuffer.remove(entry)
 6: while cache.slotsAvailable() < inputBuffer.size() do
 7:    randomEntry = cache.removeRandomEntry()
 8:    if outputBuffer.slotsAvailable() then
 9:       outputBuffer.add(randomEntry)
10: cache.addAll(inputBuffer)
11: inputBuffer.clear()
12:
13: PHASE II : Select entries to broadcast
14: if outputBuffer.slotsAvailable() then
15:    if !outputBuffer.contains(localEntry) then
16:       outputBuffer.add(localEntry)
17:    while outputBuffer.slotsAvailable() do
18:       randomEntry = cache.copyOfRandomEntry()
19:       if !outputBuffer.contains(randomEntry) then
20:          outputBuffer.add(randomEntry)
21:
22: broadcast(outputBuffer)
23: outputBuffer.clear()

/*** Passive thread ***/
// Runs when receiving a transmission

 1: for all received entries do
 2:    if inputBuffer.slotsAvailable() then
 3:       if inputBuffer.contains(entry) then
 4:          inputBuffer.keepMostRecent(entry)
 5:       else
 6:          inputBuffer.add(entry)
```

Figure 2. SharedState pseudocode.

item (represented in Figure 2 as *localEntry*) in the network whenever possible. Other than that, they execute the same algorithm. Moreover, at any point a consumer may take the role of producer if it has some information to add to the collective knowledge base.

4.2. Active Thread

Each node in the network executes the active thread once per round. The algorithm executed by the active thread is divided in two phases: a) updating the cache and b) selecting which entries to broadcast.

Phase I - Update Cache: In this phase, the node has to decide what to do with the entries accumulated in the input buffer. The result should be an updated cache with as little correlation as possible to the previous one. The reason for this is that applications that access the cache have already seen the entries in the previous version. Showing the same entries again does not provide any value for the application layer. In the majority of cases, the cache will already be full forcing the node to decide which entries from the input buffer should be placed in the cache and which entries from

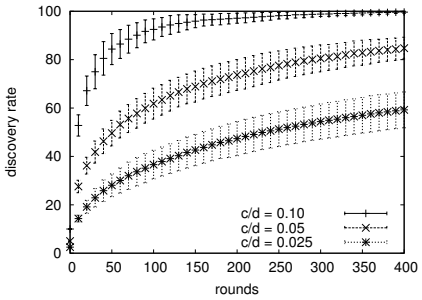


Figure 3. Discovery rate for different rates of cache size c versus number of items d.

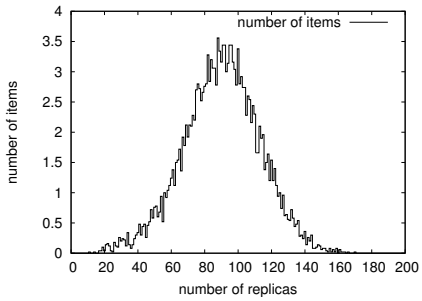


Figure 4. Histogram of the number of replicas per item in a network of 900 nodes with 180 items (cache size = 18).

the cache should be removed.

The strategy for updating a node's cache is the following. First, if an entry has already been seen (it is in the cache) and also appears in the input buffer, it is discarded from both. This helps make space for new entries in the cache. Second, all remaining entries in the input buffer should be placed in the cache. If there are not enough empty slots available in the cache, random entries from the cache are removed to make space for all the entries from the input buffer. The entries removed from the cache are placed in the output buffer until it reaches its maximum capacity. The ones that do not fit into the output buffer are discarded.

Phase II - Select Entries to Broadcast: In Phase I, some entries were already placed in the output buffer. These entries, having been removed from the cache in Phase I, have preference to be broadcast. The motivation for this is to lower the risk of items disappearing entirely from the network. If there are not enough entries to fill the output buffer, the local entry (which is available if the node is a producer) is added. If this is not enough, random entries are selected from the cache and a copy of each is placed in the output buffer. Once the selection of entries has finalized, the node broadcasts the chosen entries and clears the output buffer for the next round.

4.3. Passive Thread

Each node executes the passive thread whenever a broadcast is received. Therefore, the passive thread may execute several times in one round (depending on the number of neighbors a node has). Whenever a node receives a broadcast, the entries received are put into the input buffer. No duplicates (entries with the same id) are allowed. If a duplicate is received, the version with the freshest timestamp is kept. The input buffer has a limited capacity, therefore, entries have to be discarded once the buffer is full.

5. Basic Properties

Discovery Rate The ultimate goal of SharedState is to make data available to all nodes by storing it in the network. Since nodes themselves do not have enough storage space to store all of the available data items, they can "discover" data items from the network when required by the application layer. Discovering items can be done by inspecting the local cache (0-hop query), consulting immediate neighbors (1-hop query) or recruiting the help of neighbors to inspect caches n-hops away (n-hop query). In this paper, we consider only 0-hop queries.

We evaluate the performance of the protocol by observing the *discovery rate* of items. The discovery rate is defined as the number of items that a node discovers by examining its cache over a period of time versus the total number of items in the network. The discovery of items is gradual, as nodes update their caches once per round. The discovery rate, therefore is measured over a number of rounds and with every passing round it increases or stays the same.

The speed at which nodes discover items is directly related to the fraction of all items that they can store locally. In other words, for a collection of nodes with a cache size of c and d different items in the network, the fraction $\frac{c}{d}$ determines the discovery rate. Figure 3 presents the discovery rate over time for three different experiments. 900 nodes, each with a cache size of $c = 18$ were arranged in a 30×30 grid. The nodes can reach only their neighbors to the North, South, East and West. After the network has been running for 300 rounds, a number of test nodes start measuring their discovery rates. The graphs show the average discovery rate and standard deviation. For each experiment, a different number of items in the network d was used ($d = 180, 360, 720$). A higher value of d means that a node can store a smaller percentage of the d items locally. As Figure 3 illustrates, the discovery rate slows down when the fraction of items that a node can store in its cache decreases.

Fairness in Replication Given that our network has a fixed storage capacity (the sum of the space available at each node), the number of replicas that an item can have is limited. Items have to compete for the limited space in a node's cache. When there is a large number of items in the network, this competition is intense. When there are few

items, the competition is less fierce. Nevertheless, the protocol does not favor any particular item, resulting in every item having the same chance of creating or losing a replica.

Figure 4 presents a histogram of the number of replicas per item over a period of 50 rounds. The results correspond to the experiment in Figure 3 where $d = 180$. Our 900-node network, with $c = 18$, has 900×18 available slots. With 180 different items in the network, each item should have 90 replicas. Figure 4 shows that this is the mean value of the distribution. Due to the constant competition for space, the number of replicas for a particular item is always fluctuating.

6. Density Awareness

An important characteristic of wireless networks is that neighborhoods are defined by the proximity between nodes. If a given area is densely populated, one node's broadcast will be overheard by a large number of nodes. On the flipside, if the area is sparsely populated, the broadcast will be received just by a few nodes in the sender's range.

The protocol introduced in Section 4 does not take density information into account. Nodes blindly broadcast update messages every round, regardless of whether their neighbors would be able to handle the traffic or not. While this may not be a problem in sparse networks, in densely populated areas excessive communication could be detrimental due to collisions. It should also be noted that the size of the input buffer limits the amount of updates that a node can effectively make use of in one round. Once a node's input buffer is full, the subsequent updates have no effect on the outcome. With this in mind, we propose a slight modification of the original SharedState protocol to optimize the use of bandwidth by reducing the amount of ineffective communication.

The key to reducing the number of ineffective broadcasts is identifying when a node becomes "overloaded" by transmissions from its neighbors. If that is the case, the node cannot derive any benefit from receiving more broadcasts. It would be desirable, then, to decrease the chances that its neighbors send more broadcasts in the remainder of the round. This can be accomplished if nodes inform their neighbors of their "overload level," defined as the fraction of ineffective broadcasts received in one round. Nodes can use the overload levels of their neighbors to decide if they should broadcast in the next round or not.

We say that a node is overloaded if its input buffer is already full when it receives a broadcast. Since several broadcasts are often received in one round, each node can calculate its own overload level by keeping track of the number of broadcasts received in a round and of how many of those were received when the input buffer was already full. Let $R_i(x)$ represent the sources of broadcasts received by node x during round i and let $U_i(x)$ represent the number of ineffective broadcasts. The calculation of the overload level is based on the observations made during the previous round and takes place in the active thread, once per round, in the following way:

$$O_i(x) = \frac{U_{i-1}(x)}{|R_{i-1}(x)|}$$

Ideally, a node's overload level should be close to 0, indicating that the node rarely receives ineffective broadcasts. However, the node itself does not have direct control over its own overload level. The local overload level is determined by the behavior of the node's neighbors. For this reason, we propose an improved version of the protocol where nodes are required to append their own overload level when broadcasting a message. Each node can then accumulate these reported overload levels to get a sense of the overall overload level in its neighborhood. With this information, each node can determine if it should skip a broadcast based on whether the broadcast would benefit its neighbors or not. Let $ProbSkip_i(x)$ be the probability used by node x to decide whether to skip a broadcast or not at round i. $ProbSkip$ is calculated in the active thread as follows:

$$ProbSkip_i(x) = \frac{\left[\sum_{y \in R_{i-1}(x)} O_{i-1}(y)\right] + O_i(x)}{|R_{i-1}(x)| + 1}$$

The probability of skipping a broadcast, $ProbSkip$, is an estimate of the overload level in node x's neighborhood calculated based on the overload levels reported by x's neighbors (y) that communicated in the previous round and x's own overload level. It is important to note that this estimate may not always be very precise. The reason for this is that if a node decides not to broadcast, its neighbors will not be updated on the node's overload level. Therefore, the calculation of the overload level in the neighborhood is done with only partial information. The inclusion of the node's own overload level helps make up for the missing reports of some neighbors. In any case, as will be shown later on, an estimate - even if it is not very precise - is good enough to result in considerable resource savings.

Before moving on, it should be noted that since each node can calculate its overload level locally, an alternative optimization could be proposed where $ProbSkip$ is simply calculated based on the local overload level (under the assumption that it accurately reflects the overload levels in the neighborhood). While this strategy would work in homogeneous topologies where nodes have roughly the same number of neighbors, it does not perform as well in more complex scenarios. For example, take a situation where a node is surrounded by obstacles and as a result only has one neighbor. The neighbor, however, is surrounded by many other nodes and is often overloaded. In this case,

the first node will never be overloaded and will always broadcast, contributing to the overload of its only neighbor. The second node, meanwhile, experiences higher overload levels, meaning that it will skip some rounds. The results are detrimental to both nodes: the first receives broadcasts only sporadically, as its only neighbor tends to skip broadcasts, while the second node becomes even more overloaded. Under the scheme that we proposed earlier, the first node would be aware that its neighbor is often overloaded and would skip some rounds to relieve its neighbor's load. At the same time, the second node measures less overload in its neighborhood and would tend to broadcast more often, benefitting the first node. Through experimentation, we observed that the method proposed earlier performs better overall. We compare its performance against the original SharedState in the remainder of the paper.

7. Performance Evaluation

We evaluate the effectiveness of the SharedState system by observing i) the *discovery rate* of items (as defined in Section 5) and ii) the number of broadcasts generated. Section 6 introduced a modified version of the original algorithm aimed at reducing resource consumption by letting nodes skip broadcasts according to the overload levels in their neighborhoods. In order to quantify the improvements introduced by the modified algorithm, we gathered statistics (over a test interval of 50 rounds) on: a) *Broadcasts sent* in the whole network per round and b) *Broadcasts received* by a node per round. These statistics give us some insight into the usage of the communication medium and the workload of the nodes, both of which we aim to reduce.

We tested our storage system by implementing it as a TinyOS application and using TOSSIM to run experiments under different scenarios. In order to observe the effect of varying node densities in the performance of the protocol, various topologies with different node densities were used. The topologies were generated with the *LinkLayerModel* tool that comes bundled with TinyOS. This tool generates network topologies using a theoretical propagation model. Within a 100×100 meter terrain, we explored the following simulation scenarios:

Uniform Node Distribution For the "uniform" setting, the physical terrain is divided into a number of cells (based on the number of nodes) and a node is randomly placed within each cell. Since the tool requires that the number of nodes be a square, we generated topologies with the following numbers of nodes: $100, 144, 196, ...900$, or $(10 + 2n)^2$ for $n = 0...10$.

"Center of attraction" Distribution This scenario emulates a more realistic situation where nodes gather around a point of interest. We crafted a topology with 576 nodes where the highest concentration of nodes occurs at the center of the terrain $(50, 50)$. The position of the nodes was determined by selecting a random angle between 0 and 360 degrees and a distance from the center according to a normal distribution. The minimum distance between nodes was set to 1 meter.

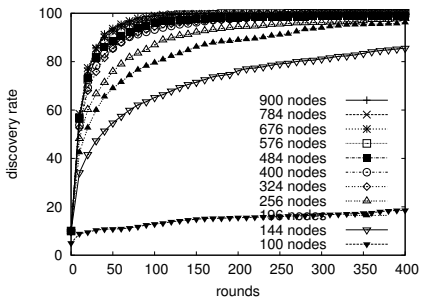


Figure 5. Discovery rate over time for networks of different sizes.

Besides the distribution of nodes, system-wide parameters and the role of nodes had to be defined. For all experiments, the following settings were used:

- Of all nodes in the network, 80 nodes were chosen at random to be publishers. These 80 nodes produce 80 items that are disseminated through the network.
- 20 nodes selected at random are chosen as "test nodes." These nodes measure their discovery rates and numbers of broadcasts sent and received.
- For all nodes, the cache and the input buffer can hold only 8 entries. The output buffer can hold 4 entries.
- All nodes execute the active thread once per second.

8. Uniform Node Distribution

In this section, we will focus on the behavior of the system when the collection of participating nodes is spread uniformly over an area of 100×100 meters.

8.1. Discovery Rate and Node Density

Figure 5 shows the discovery rate measured over time for the different topologies, starting from a sparse 100-node network and increasing in density up to 900 nodes in the same 100×100 meter terrain. Notice how the increase in density leads to higher discovery rates. It can be clearly observed that for a sparse network of 100 nodes the discovery rate after 50 rounds is considerably low (about 11%) and does not improve substantially over time. This is due to the low connectivity between nodes. With only 100 nodes in the 100×100 meter terrain, the network is too sparse. However, even a slight increase in density (144 nodes) already yields much better results.

To better understand the discovery rate results, it is necessary to have more insight into the connectivity of the different topologies. Figure 6 shows the average number of transmissions received by a node in one round for the different topologies used in Figure 5. A linear relationship be-

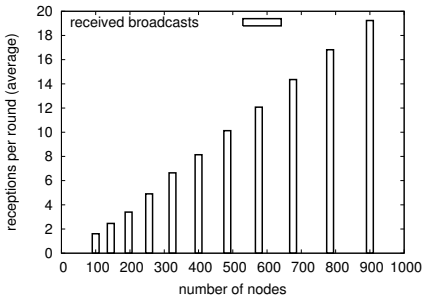


Figure 6. Av. number of broadcasts received per node for networks of different sizes.

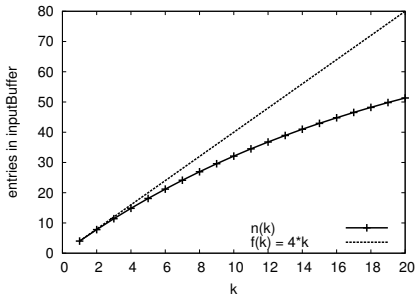


Figure 7. Number of entries in the input buffer after receiving k transmissions ($s = 4, d = 80$).

tween the number of nodes in the network and the number of received transmissions can be distinctly observed. Taking a closer look at Figure 6, we can see why the discovery rate for the 100-node network was so low. With an average number of receptions per node of 1.6, it is not uncommon for nodes to be unreachable or for the network to become partitioned at times. At the other end of the spectrum, we have the 900-node network with an average of 19.23 transmissions received per node in one round.

The linear increase in receptions with the number of nodes does not translate to a linear increase in performance (measured by the discovery rate), as shown in Figure 5. After a considerable improvement in discovery rate when going from the sparse 100-node network to the more populated 144, 196 and 256-node networks, further increases in density fail to have a substantial effect in the discovery rate. The reason for this can be traced back to the limited size of the input buffer. With the input buffer being twice the size of the output buffer for our experiments, the first transmission received fills up half of the input buffer. The subsequent transmissions gradually fill up the rest. Since the input buffer becomes full after receiving a few transmissions, the higher number of transmissions received in the denser topologies do not provide much additional benefit.

We can model the way the input buffer fills up under the assumptions that: a) each entry received is selected randomly from the collection of d different items in the network, b) each transmission from a neighbor consists of s randomly selected entries and c) the input buffer is infinite. Let n_k represent the number of entries in the input buffer after k transmissions.

For $k = 1$, all entries are kept, resulting in $n_1 = s$. For the second transmission, some of the s received entries might already exist in the input buffer. Therefore, n_2 only increases by $s \cdot \left(1 - \frac{n_1}{d}\right)$ entries, where $\frac{n_1}{d}$ is the probability that an entry is already in the input buffer. Likewise, n_3 increases by $s \cdot \left(1 - \frac{n_2}{d}\right)$ entries over n_2. A general expression for n_k can be derived in terms of s, d and n_{k-1}:

$$n_k = n_{k-1} + s \cdot \left(1 - \frac{n_{k-1}}{d}\right), n_0 = 0$$

The second term of the equation represents the increase in entries for each additional transmission. Since the probability of receiving an entry that is already in the input buffer, $\frac{n_{k-1}}{d}$, increases as the input buffer accumulates more entries, the second term becomes smaller with every new transmission.

Using the same parameters as our TOSSIM simulations (number of items $d = 80$ and output buffer size $s = 4$), n_k is plotted in Figure 7. The graph illustrates how later transmissions have less impact on filling up the input buffer by comparing n_k with a curve depicting a linear increase in number of entries with each transmission received. We can also observe from this graph that, under this model, our input buffer (which can hold only 8 entries) would be full by the third transmission. This helps explain why the 100-node network underperforms: the input buffers are rarely used to their full capacity.

On the other hand, the 256-node network, where the average number of broadcasts received is 4.8 with a standard deviation of 1.5, makes full use of the input buffers and performs considerably better. Notice, however, how the experiments with more than 256 nodes show slightly better discovery rates. We speculate that this is due to the fact that when nodes have more neighbors, the s entries broadcast by each node are more likely to be truly random selections from the d possible items in the network. For more sparse networks where nodes have only a few neighbors, the entries at neighboring nodes are more likely to be correlated, slowing down the discovery of new items.

8.2. Taking Advantage of Node Density

The SharedState protocol works as expected, but suffers from a common problem in wireless networks: unnecessary transmissions. Section 6 introduced a modified version of the protocol aimed at optimizing the use of bandwidth by taking node density into account. Knowing that the input buffers have a limited capacity, it is evident that at some point additional transmissions are not effective anymore and resources are wasted on them. In this section, we show that our modified algorithm can reduce the waste of resources

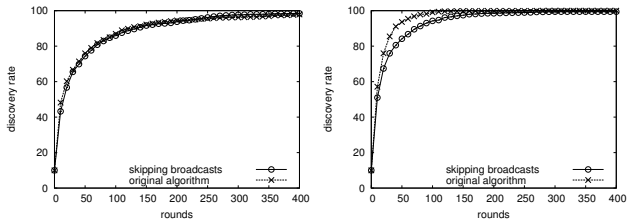


Figure 8. Comparing the original algorithm and the improved version in terms of discovery rate for: a) 256 nodes and b) 784 nodes.

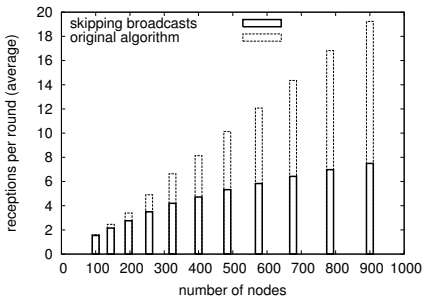


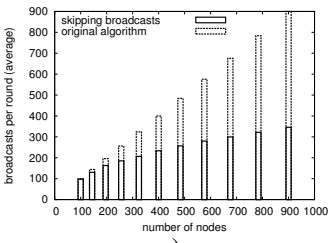
Figure 9. Av. number of broadcasts received per node for networks of different sizes.

while still delivering good performance.

We start by comparing the discovery rate over time for the original algorithm and the density-aware version. Figure 8 shows the results for two selected topologies: a sparse (256 nodes) and a dense (784 nodes) one. The performance is virtually the same in the sparse topology. For the dense topology, the discoveryrate is slower during the initial phase of the experiment. Nevertheless, it recovers and matches the original algorithm in the later stage of the experiment.

Given that performance has not been compromised by the changes introduced to the original algorithm, we proceed to study the effect the changes have on the workload of the network. First, we measure the average number of broadcasts received per node during one round and compare the results to the original ones. Figure 9 presents the new measurements alongside the results shown in Figure 6, clearly showing the reduction in the number of broadcasts received per node. As a result, the nodes have less transmissions to handle in each round. This is not surprising, given that in the density-aware version of the algorithm nodes refrain from broadcasting based on the overload levels measured in their neighborhoods. It can be expected, then, that nodes in denser topologies would experience higher overload levels and, therefore be more likely to skip broadcasts.

Figure 10 showcases the reduction in the number of broadcasts sent per round in the whole network. Figure



Number of Nodes	Broadcast Ratio
100	0.97
144	0.91
196	0.83
256	0.72
324	0.64
400	0.58
484	0.53
576	0.49
676	0.44
784	0.41
900	0.38

b)

Figure 10. Number of broadcasts sent per round for networks of different sizes.

10.a) highlights the impact of density in the decrease of broadcasts being sent. As the topologies become more dense, and the number of neighbors per node increases, nodes are more likely to be overloaded increasing the probability of broadcasts being skipped. The companion table to the right gives a more detailed account of the reduction in broadcasts, with the column titled "Broadcast Ratio" referring to the ratio of the number of broadcasts sent in one round using the density-aware protocol versus the number of broadcasts using the original version. While the difference is minimal in the sparse 100-node network, as the networks become more dense, it is clear that the new algorithm allows the nodes to save transmissions by reducing the number of ineffective broadcasts.

9. Center of Attraction

In this section, we observe the behavior of our system using a more realistic node distribution where nodes (576 in total) are arranged around a central point. While the uniform node distributions used previously may approximate the layout of a sensor network (in a field, for example), we think that this scenario resembles more closely a social event, such as an outdoor barbeque, where people gather around a central location (a bonfire, for example).

We start by comparing the original algorithm and the modified version using the new centralized topology. In terms of discovery rate, the results vary slightly (see Figure 11), with the original version outperforming the modified version in the initial rounds of the experiment. After that initial period, both versions perform similarly, with the modified version gaining an edge over the original protocol towards the end of the experiment.

Given that the density-aware version of the protocol behaves as expected, we proceed to analyze the resource usage in the network. For this experiment, values of ProbSkip for every node run were collected over a 400-round. The histogram in Figure 12 shows the percentage of nodes using a value of $ProbSkip$ that falls within a certain range. It is clear that the graph is skewed towards high values of ProbSkip, indicating that most nodes skip some rounds. In fact, a large majority skips more that 50% of broadcasts. It

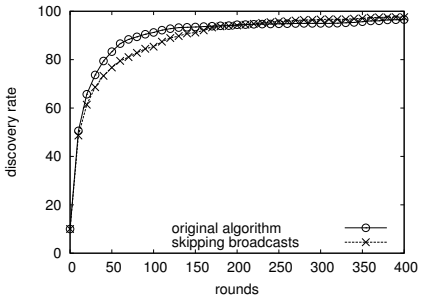


Figure 11. Comparing the original algorithm and the improved version in terms of discovery rate.

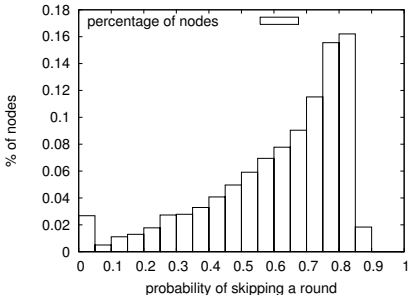


Figure 12. Distribution of nodes according to their probability of skipping a round.

should be noted, though, that a few nodes (roughly 3%) do not skip any broadcasts. These are the nodes in the outer regions of the terrain, which have only a few neighbors and need to take advantage of every broadcast.

The relationship between the distance from the center and the probability of skipping a round becomes evident in Figure 13. During a period of 50 rounds, every node reported its ProbSkip and distance from the center. This graph plots each pair $(distance, ProbSkip)$ as one point and clearly shows a trend where nodes closer to the center report lower values for ProbSkip.

10. Conclusions

In this paper, we have demonstrated that it is possible to build an effective shared-storage solution based purely on probabilistic methods. Moreover, we have shown that by taking into account only local neighborhood information the use of resources, namely bandwidth and energy, can be drastically reduced without having a mayor impact on performance. We achieve this by allowing nodes to regulate their output to prevent their neighbors from being overloaded, which in turn benefits them by saving transmission costs. This results in resource savings according to the density of the area, without the need to explicitly disseminate topology information. We conclude that the combination of

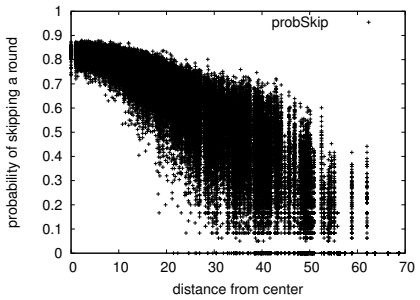


Figure 13. Probability of skipping a round in relation to the distance from the center.

a probabilistic approach and local-only decision making is key to the scalability of systems such SharedState.

References

[1] S. Chessa and P. Maestrini. Dependable and secure data storage and retrieval in mobile, wireless networks. *Int. Conf. Dependable Systems and Networks (DSN'03)*, page 207, 2003.

[2] A. Dimakis, V. Prabhakaran, and K. Ramchandran. Ubiquitous access to distributed data in large-scale sensor networks through decentralized erasure codes. In *Proc. of Information Processing in Sensor Networks (IPSN '05)*, April 2005.

[3] V. Drabkin, R. Friedman, G. Kliot, and M. Segal. RAPID: Reliable probabilistic dissemination in wireless ad-hoc networks. In *SRDS '07*, Beijing, China, October 2007.

[4] A. Ghose, J. Grossklags, and J. Chuang. Resilient data-centric storage in wireless ad-hoc sensor networks. In *Proc. of MDM '03*, Melbourne, Australia, January 2003.

[5] Z. J. Haas, J. Y. Halpern, and L. Li. Gossip-based ad hoc routing. In *Proceedings of IEEE INFOCOM 2002*, 2002.

[6] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *NSDI '04*, 2004.

[7] M. Motani, V. Srinivasan, and P. S. Nuggehalli. Peoplenet: engineering a wireless virtual social network. In *MobiCom '05*, pages 243–257, New York, NY, USA, 2005. ACM.

[8] M. Papadopouli and H. Schulzrinne. Effects of power conservation, wireless coverage and cooperation on data dissemination among mobile devices. In *MobiHoc '01*, pages 117–127, New York, NY, USA, 2001. ACM.

[9] F. Sailhan and V. Issarny. Cooperative caching in ad hoc networks. In *MDM '03*, Melbourne, Australia, January 2003.

[10] S. Shenker, S. Ratnasamy, B. Karp, R. Govindan, and D. Estrin. Data-centric storage in sensornets. *SIGCOMM Comput. Commun. Rev.*, 33(1):137–142, 2003.

[11] O. Wolfson, B. Xu, and R. M. Tanner. Mobile peer-to-peer data dissemination with resource constraints. In *Proc.of Mobile Data Management (MDM '07)*, 2007.

[12] L. Yin and G. Cao. Supporting cooperative caching in ad hoc networks. *IEEE Transactions on Mobile Computing*, 5(1):77–89, January 2006.