

# Data Centric Storage with Diffuse Caching in Sensor Networks

Keng-Teck Ma, King-Yip Cheng, King-Shan Lui, Vincent Tam  
Department of Electrical and Electronic Engineering  
The University of Hong Kong  
Pokfulam Road, Hong Kong, China  
{ktma, kycheng, kslui, vtam}@eee.hku.hk

## Abstract

In a sensor network that adopts *data-centric storage (DCS)*, information of the same kind is kept in the same set of nodes. That is, when a sensor detects a certain event, no matter where it is, it sends the information to the designated location. When another node wants the information, it can send a query to that location to retrieve it. Unfortunately, existing protocols based on DCS are prone to the hot-spot problem where some nodes have to handle lots of messages. In this paper, we study how to apply *diffuse caching* on top of DCS. We diffuse popular information to other nodes so as to share the workloads. Our diffusing mechanism is adaptive that the distribution varies based on the popularity of the event type. We evaluate our protocol using simulations and the results show that our protocol successfully alleviates the hot-spot problem and reduces the message overheads.

## 1 Introduction

Research in sensor networks has received much attention in the past few years. Sensor networks differ from traditional wireless networks in many aspects and many mechanisms developed for traditional wireless networks cannot be directly applied in sensor networks. One of the major tasks of sensors is to detect events, such as a change in temperature, the appearance of an animal, etc. A particular node in the network may be interested in knowing about the event information. A node may know about the types of event it is interested in but without any knowledge about the locations where such events happen. To get an answer, a query has to be sent to a node that possesses the information. A conventional approach to facilitate query nodes to acquire what they want is flooding. In the *push-based* approach, whenever an event happens, the information is flooded to all the nodes in the network. Then, every node will have all the information and does not need to send out any query at all. In the *pull-based* approach, a node that detects an event keeps the information in its own storage without sending the information to any other node. When a node wants that information, it

floods the network with a query. The node that possesses the requested information replies the query.

Although flooding is a simple technique and is commonly used in many wireless network routing protocols, it is not very suitable for sensor networks. Sensor nodes are limited in energy and reducing the energy used in message transmission is one of the major optimization criteria of sensor network protocols. However, the number of messages required in each flooding is proportional to the number of nodes in the network. To make the problem worse, as the message will be broadcasted by nearby nodes at about the same time, collisions happen frequently and it leads to retransmissions of messages which consume extra energy. This problem is very serious in sensor networks since there are a large number of nodes and nodes are close to each other. As a result, flooding should be used only when necessary. Referring back to the *push-based* approach, every node receives the information of an event. However, it is not necessary if only a few number of nodes are interested in the information. Energy is wasted in distributing the event information to nodes that do not require the information. The *pull-based* approach has a similar drawback. The query messages that are sent to nodes other than the node that detected the event are useless. It is obvious that energy can be saved if information is kept and distributed in a more intelligent manner.

To allow efficient routing of queries, the concept of *data-centric storage (DCS)* is introduced [1]. When a sensor detects an event, it decides which location is responsible for keeping the information and the information is sent to a node in that location. The location depends on the type of information and can be computed using a globally known hash function [2]. Then, when a node requests a particular piece of information, it can find out the location by the hash function and send a query to that location. If a certain event is very popular and there are a lot of nodes requesting it, the node that keeps the information will suffer from the hot-spot problem. To solve this problem, sharing the workload is the only way. Many papers suggest that instead of having one node keeping the information, multiple nodes should possess the same piece of information to share query processing [3, 4, 5]. However, the sharing patterns of [3, 4, 5] are not adaptive to the dynamic nature of the popularity of a certain event. The patterns may be the same for all event types or remain the same for all the time for a certain event type. To solve the hot-spot problem without any unnecessary overhead in information distribution, the mechanism should be adaptive to the popularities of different event types. In this paper, we adopt the idea of diffuse caching to replicate information in multiple nodes. Our mechanism dynamically adjusts the event distribution pattern based on the popularities of the event types.

The rest of the paper is organized as follows. Related work is discussed in Section 2. Section 3 presents our network model and protocol. Simulation results are presented in Section 4. And we conclude our paper in Section 5.

## 2 Related Works

Many protocols have been developed based on the data-centric storage [2, 6, 7, 3, 4, 8, 5].

The GHT protocol [2] is based on the data-centric storage [1] idea and stores events at nodes nearest to some pre-hashed globally known geographical locations. Similar idea is also proposed in GEM [7]. However, the nodes are not location-aware and an embedded tree is implemented to provide relative positions.

Other protocols, [6] and [9], address efficient handling of queries with scalar ranges, such as “list all events whose temperatures lie between  $50^\circ$  and  $60^\circ$ , and whose light levels lie between 10 and 15.” This paper is not focused on this kind of queries but it should be noted that our caching mechanism can be integrated with these protocols. [8] develops a *multi-resolution querying* scheme. An event is detected by a set of nodes instead of one. Different subsets of these nodes provide different resolutions of the event.

The event model in this paper is similar to those used in [10, 11, 2, 7, 3, 4, 5] that an event is something of interest and can be any kind of information. Events are independent of each other and occur randomly at anywhere. Multiple events can belong to the same *event type*. For example, there may be multiple locations in the network detecting an elephant and these detection events all belong to the same event type. A query specifies the event type that it wants. Rumor Routing [10] can only answer one event of the event type while the others [11, 2, 7, 3, 4, 5] can answer the complete event set of the same type. Note that Rumor Routing and the method in [11] do not follow the DCS approach. Like the events, we assume that queries are raised randomly. This assumption is different from [12] which assumes a certain node would make multiple queries within a certain time interval so that an optimal path can be set up after the first query of that node is issued by flooding.

GHT [2] basically keeps events of the same type in a certain node. However, this leads to the *hot-spot* problem. If a particular event type is very popular, the node that keeps the information will be overloaded with query processing. To solve the hot-spot problem, keeping the same information in multiple locations is the only way. The method described in [3] suggests to divide the network into geographical regions. Information of an event is kept in all nodes in that region instead of only one node. [4] suggests that several locations can be used to keep the information based on resiliency levels. [5] proposes hashing an event type to a certain height level and events of that type are distributed across nodes around that level. A drawback of these protocols is the replication of events is the same for all event types, no matter whether it is popular or not. If an event type is not popular and there are only a few queries, the information replication is not necessary and energy is wasted in spreading the information.

Yin and Cao study the integration of caching mechanism in an ad-hoc network [13]. Neighboring nodes share the events based on some condition. However, the mechanism employs the weak cache consistency

model and is not easy to answer queries that request the whole set of events of the same type. The hot-spot problem in range query processing is first studied in [14]. Two algorithms, Zone Partitioning (ZP) and Zone Partial Replication (ZPR), are developed to work on top of the DIM scheme [6]. ZP solves the hot-spot problem by decomposing the information kept in a node while ZPR replicates information to *all* direct neighbor nodes based on some condition. ZP is not applicable to non-range queries that we study in this paper and ZPR simply replicates information in a region. To the best of our knowledge, our protocol is the first protocol to adaptively distribute information to *individual* neighbors according to query frequencies and arrival patterns.

### 3 DCS with Diffuse Caching

#### 3.1 Network Model

The network model for our protocol is identical to that of GHT [2]. The estimated boundary of the network is known and each sensor node has a good estimation of its own location. We also adopt the modified GPSR [15] as specified in [2] to route messages. Since our mechanism works on top of GHT (and can be extended easily for other similar protocols), we first briefly describe the protocol and some terminologies.

Each event occurs in the network belongs to a certain *event type*. There can be multiple events for a single event type. The idea of DCS is when a sensor detects an event, it decides which location is responsible for keeping the information and the information is sent to a node in that location. The location depends on the event type and can be computed using a globally known hash function. Since it is possible that there is no node in the hashed geographical coordinate, GHT develops a mechanism to assign a node in the neighborhood (usually the closest node) to keep the event. The node is called *home node*. Then, when a node (*query node*) requests information of a particular event type, it can find out the location by the hash function and send a query to that location. In most cases, the home node receives the query and sends a reply back. This reply can contain all the events of the requested type that happened within a certain past period, or the most recently happened event. In this paper, we focus on the first kind of reply. An answer to a query contains all the events of the requested event type. Following GHT, each event is contained in a separate message.

#### 3.2 Basic principle

The home node of a particular event type keeps all the events of that type. In GHT, all queries are directed to the home node. If a certain event type is very popular, the home node suffers from the hot spot problem. The idea of diffuse caching is to allow the home node to share its events with some neighboring nodes. We call

these neighboring nodes *caching nodes*. A caching node can reply a query right away instead of forwarding the query further to the home node if it receives one. Then, we alleviate the hot spot problem of the home node since the home node does not have to process all the queries. On the other hand, both the queries and replies are following shorter paths. The number of messages is thus reduced as well.

To maintain the same list of events in both the home node and caching nodes, whenever a new event is reported to the home node, the home node has to diffuse the event to the caching nodes. If there is no query for that event type, this additional message will be *wasted* and introduce unnecessary message overhead. Therefore, we should not blindly diffuse event information but try to outweigh the additional overhead by message saving. In our protocol, the home node diffuses information to its neighbor only when the *potential saving* is larger than the *potential waste*. When the potential saving of a neighbor is large enough, the home node *promotes* that neighbor. This process is called *promotion* and the home node is the *promoter* of its neighbor. We also define *potential cost* to be *potential waste* - *potential save*. In other words, if potential cost is a negative value, the potential saving outweighs the potential waste; otherwise, the potential waste is larger. In a similar fashion, a caching node in turn can promote its neighbors if the savings outweigh the wastes. As the ratio of queries to events increases, more nodes will be caching the queried events and we have the diffusion effects. Note that like the home node, a caching node is specific to a particular event type. A caching node of event type *A* may not necessarily be a caching node of event type *B* at the same time.

The popularity of an event type is hardly static. That is, an event type can be very popular for a while but then is queried very infrequently later. In this case, a caching node has to "step down" to become a normal node. This decision is again dependent on the potential saving and potential waste, but instead of being made by the promoter, it is made by the caching node itself. If the potential waste outweighs the potential saving, the caching node notifies its promoter to stop forwarding events to it.

### 3.3 Information Needed

For each event type, say *A*, the home node and each caching node of event type *A* keep a potential cost lookup table. The table in node *n* has one entry for each of *n*'s neighbors specifying the potential cost of each neighbor. If *n* is a caching node, it should also keep its own potential cost.

Promotion is done when the potential cost of a certain neighbor is smaller than a certain threshold. On the other hand, caching node *n* has to step down if its potential cost is too high and none of its neighbors except its promoter is a caching/home node. *n* also keeps track of the status of its neighbors. Node *n* should mark the neighbors who have been promoted to caching nodes so that it knows it has to diffuse information to these nodes. Definitely, *n* should unmark a step-down neighbor. Initially, only the home node has the potential cost

table for the event type that it is responsible for. All potential costs of its neighbors are initialized to be zero and all neighbors are unmarked since they all are not caching nodes in the beginning.

Apart from potential costs of neighbors, a caching node should also record the number of events in the event lists that have been queried by its neighbors. Events and queries are interleaving. New events can occur after the last query is issued. Therefore, it is possible that there are some events in the event list that have not been queried. In the following discussion, we use  $num\_queried(p)$  and  $num\_N\_queried(p)$  to represent the number of events that have been queried and have not been queried by neighbor  $p$ , respectively. Note that for different event types, there should be different  $num\_queried(*)$  and  $num\_N\_queried(*)$ . Since we are describing the situation for a certain event type, we drop the variable for clarity of presentation.  $num\_queried(p)$  and  $num\_N\_queried(p)$  are updated whenever a new event or a query arrives from  $p$ . These two quantities are used for updating potential costs of neighbors. Please note that  $num\_queried(p) + num\_N\_queried(p)$  is the total number of events kept in the node.

In summary, the home node and each caching node  $n$  keeps a table containing the following fields for each neighbor  $p$ . Section 3.5 describes how to initialize and update these variables.

- $cost(p)$ : potential cost of  $p$
- $num\_queried(p)$ : number of events that are kept in  $n$  that have been queried by a query forwarded to  $n$  by  $p$
- $num\_N\_queried(p)$ : number of events that are kept in  $n$  that have not been queried by any query forwarded to  $n$  by  $p$
- $isCachingNode(p)$ : indicator of whether  $p$  has been promoted to a caching node

### 3.4 Promotion and Step-Down

A home/caching node  $n$  promotes a non-caching neighbor  $p$  if  $cost(p)$  is less than a certain threshold,  $delta_1$ .  $n$  should send a  $\langle PROMOTE \rangle$  message and the list of events that it is maintaining to  $p$ .  $n$  should also mark  $p$  as a caching node by updating  $isCachingNode(p)$  in its table.

While promotion is a decision made by a neighbor, a step-down conclusion is made by the stepping down node itself. When a caching node  $n$  realizes its potential cost  $cost(n)$  is larger than  $delta_2$ , it should unmark the caching node flag of its entry in its own table. If no neighbor is a caching node except its promoter,  $n$  should step down. It informs its promoter  $x$  by sending a  $\langle STEP - DOWN \rangle$  message and can clear the potential cost table and the events.  $x$  should update the status of  $n$  upon receiving the  $\langle STEP - DOWN \rangle$  message. It should also reset  $cost(n)$ . Details are provided in the next section.

$\delta_1$  should be non-positive while  $\delta_2$  should be non-negative. The choices of  $\delta_1$  and  $\delta_2$  should be dependent on the applications. For many applications, using global small value, possibly 0, for  $\delta_1$  and  $\delta_2$  produce good results in cost savings. In complex applications, the value of  $\delta_1$  and  $\delta_2$  can be dynamic and relying on various parameters, which is beyond the scope of this paper. In our simulations, we set both  $\delta_1$  and  $\delta_2$  to be 0.

### 3.5 Potential Cost Updates

We now describe the different situations one by one and explain how to adjust the variables and when to perform promotion and step-down check. An example is presented in Section 3.6.

#### Case I: Home node receives the FIRST event of the type it is responsible for

The home node keeps the event in its memory and initializes the information for each neighbor  $p$ :

- $cost(p) = 0$
- $num\_queried(p) = 0$
- $num\_N\_queried(p) = 1$
- $isCachingNode(p) = False$

#### Case II: Home node receives an event and it is not the first one

The home node should keep the event in its memory and increment  $num\_N\_queried(p)$  for every neighbor  $p$  by 1 indicating this new event has not been queried by any neighbor. The event is then forwarded to all caching neighbors.

#### Case III: Node $q$ receives a $\langle PROMOTE \rangle$ message and a list of events from $n$

$q$  knows that it has been promoted as a caching node and it does the following:

1. mark  $n$  as the promoter and keep the list of events in memory
2. for each neighbor  $p$  where  $p \neq n$ ,
  - $cost(p) = 0$
  - $num\_queried(p) = 0$
  - $num\_N\_queried(p) = \text{number of events in the list}$
  - $isCachingNode(p) = False$
3. update its own information

- $cost(q) = -(\text{number of events in the event list})$
- $num\_queried(q) = 0$
- $num\_N\_queried(q) = \text{number of events in the list}$
- $isCachingNode(q) = True$

**Case IV: Caching node  $n$  receives a new event from its promoter**

1.  $cost(n) = cost(n) + 1$

$n$  increases its potential cost to reflect the potential waste of having made  $n$  a caching node if no more query arrives.

2. for each neighbor  $p$ ,  $num\_N\_queried(p) = num\_N\_queried(p) + 1$  and also  $num\_N\_queried(n) = num\_N\_queried(n) + 1$

$num\_N\_queried(*)$  are incremented by 1 since there is one more event that has not been queried.

3.  $n$  checks whether it has to step down

- if  $cost(n) > delta_2$ ,  $isCachingNode(n) = False$
- if  $isCachingNode(n) = False$  and  $isCachingNode(p) = False$  for all neighbor  $p$ ,  $n$  has to step down and send  $\langle STEP - DOWN \rangle$  to its promoter

4. If  $n$  does not have to step down, it forwards the event to all caching neighbors and keeps the event in its memory

**Case V: Caching/Home node  $n$  receives a query from neighbor  $p$**

In this case, neighbor  $p$  must be a normal node because if it were a caching node, it should have answered the query by itself instead of forwarding it to  $n$ . Let  $cost(p)$  be the potential cost of  $p$  in the lookup table of  $n$ .  $n$  performs the following:

1. Update  $cost(p)$

$$cost(p) = cost(p) - 1 - num\_queried(p) + num\_N\_queried(p)$$

There are three adjustment components:  $-1$ ,  $-num\_queried(p)$ , and  $+num\_N\_queried(p)$ .  $p$  sends a query to  $n$ . Consider that if the events had already been cached in  $p$ , then  $p$  would not have sent the query message to  $n$  and 1 query message could have been saved. This accounts for the  $-1$  in the first term. Furthermore, since  $p$  is nearer to the query node than  $n$ , all the reply messages could have been travelled one less hop if  $p$  had been a caching node. The saving is at least  $num\_queried(p)$  and so

we reduce  $cost(p)$  by this entity. On the other hand, if the events had been cached in  $p$  but no query had arrived, at least  $num\_N\_queried(p)$  event message would have been potentially wasted. In other words,  $num\_N\_queried(p)$  is an indication of how infrequent queries are.  $p$  should not be promoted if  $num\_N\_queried(p)$  is large.

2. Update  $num\_queried(p)$  and  $num\_N\_queried(p)$

$$num\_queried(p) = num\_queried(p) + num\_N\_queried(p)$$

$$num\_N\_queried(p) = 0$$

All the events have been queried by  $p$  and so  $num\_queried(p)$  should be the same as the number of events kept. Accordingly,  $num\_N\_queried(p)$  is set to be 0 reflecting that there is no event that has not been queried.

3. Check if  $p$  has to be promoted and reply the query

If  $cost(p) < delta_1$ , send  $\langle PROMOTE \rangle$  and the list of events to  $p$

4. If  $n$  is a caching node, update  $cost(n)$

$$cost(n) = cost(n) - 1 - (num\_queried(n) + num\_N\_queried(n))$$

We decrement  $cost(n)$  by the number of events because it is the saving we obtain by making  $n$  a cache node. The -1 is the saving of the query message.

#### **Case VI: Caching/Home node $n$ receives a $\langle STEP - DOWN \rangle$ message from neighbor $p$**

1. Update the information of  $p$ ,

- $isCachingNode(p) = False$
- $cost(p) = 0$
- $num\_N\_queried(p) = num\_N\_queried(n) + num\_queried(n)$
- $num\_queried(p) = 0$

2. if  $isCachingNode(n) = False$  and all neighbors are not caching nodes except its promoter,  $n$  has to step down and send a  $\langle STEP - DOWN \rangle$  message to its promoter

### **3.6 Example**

We now use an example to illustrate the promotion and the step-down processes. Consider the network in Figure 1 where  $h$  is the home node of a certain event type.  $p$  and  $q$  are neighbors of  $h$  while  $r$  is a neighbor of  $q$ . Figure 1(a) - (f) show a series of events that occur. Table 1(a) - (f) give the information in the tables of

$h$  and  $q$ . All the events and queries in the figures belong to the same type. We use a number to specify the order of events and queries that  $e1$  stands for the first event and  $q2$  stands for the second query. In the tables, the owner of the table is specified in the top left-hand corner. That is, Table 1(a) is the table of  $h$  while Table 1(f) is the table of  $q$ . Due to space limitation, “num\_N\_Q” stands for  $num\_N\_queried$  and “num\_Q” stands for  $num\_queried$ .

When  $h$  receives the first event from  $q$  (Figure 1(a)),  $h$  initializes its table as shown in Table 1(a).  $num\_N\_queried(*)$  are set to 1 while  $cost(*)$  and  $num\_queried(*)$  are 0. When the second event arrives from  $q$  (Figure 1(b)),  $num\_N\_queried(*)$  are incremented by 1. Suppose now  $q$  sends a query to  $h$  (Figure 1(c)).  $cost(q)$  becomes  $0 - 1 - 0 + 2 = 1$ .  $num\_N\_queried(q)$  is now 0 and  $num\_queried(q)$  is 2 as illustrated in Table 1(c). Since  $cost(q)$  is still larger than  $delta_1$ ,  $q$  is not promoted. When the second query arrives at  $h$  (Figure 1(d)),  $cost(q)$  becomes  $1 - 1 - 2 + 0 = -2 < delta_1$  (Table 1(d)) and  $h$  has to promote  $q$ .  $h$  promotes  $q$  by sending a  $< PROMOTE >$  message and the events it has received to  $q$ .  $h$  should also mark  $q$  as a caching node as shown in Figure 1(e).  $q$  then initializes its table to have an entry for itself and its neighbor  $r$  (Table 1(e)). In Figure 1(f),  $h$  receives a new event from another node. Since  $q$  is a caching node,  $h$  forwards the event to  $q$ .  $q$  should then update  $cost(q)$  and  $num\_N\_queried(*)$ . As  $cost(q)$  is still less than  $delta_1$ ,  $q$  does not have to step down. Suppose now  $h$  forwards two more events to  $q$  as in Figure 1(g).  $cost(q)$ ,  $num\_N\_queried(q)$ , and  $num\_queried(q)$  are all incremented by 2 as in Table 1(g).  $cost(q)$  is now smaller than  $delta_2$  and so  $q$  has to step down by sending a  $< STEP - DOWN >$  message to  $h$  (Figure 1(h)).  $h$  then resets the information of  $q$  in its table as illustrated in Table 1(h).

## 4 Simulation

In this section, we evaluate the performance of our protocols and compare it with GHT [2] using the J-Sim simulator [16]. The standard J-Sim wireless physical and MAC layer models are used. We have simulated two different query node patterns. The first pattern is modelled according to the situation described in [2]. There is only one query node and all events happened before the first query is issued. In the second pattern, there are multiple query nodes and queries and events are interleaved. As the focus of this paper is about the replication effectiveness based on query arrival pattern, node failure is not considered in the simulations. All nodes are assumed to have sufficient energy. If replication for node failure is required, GHT with Replicas approach can be used. Our mechanism can still be applied by assuming all replicas as home nodes. The hash function used is not well defined in the GHT paper. Therefore, the constrained hashing as described in [5] is implemented.

The evaluation metrics we use are:

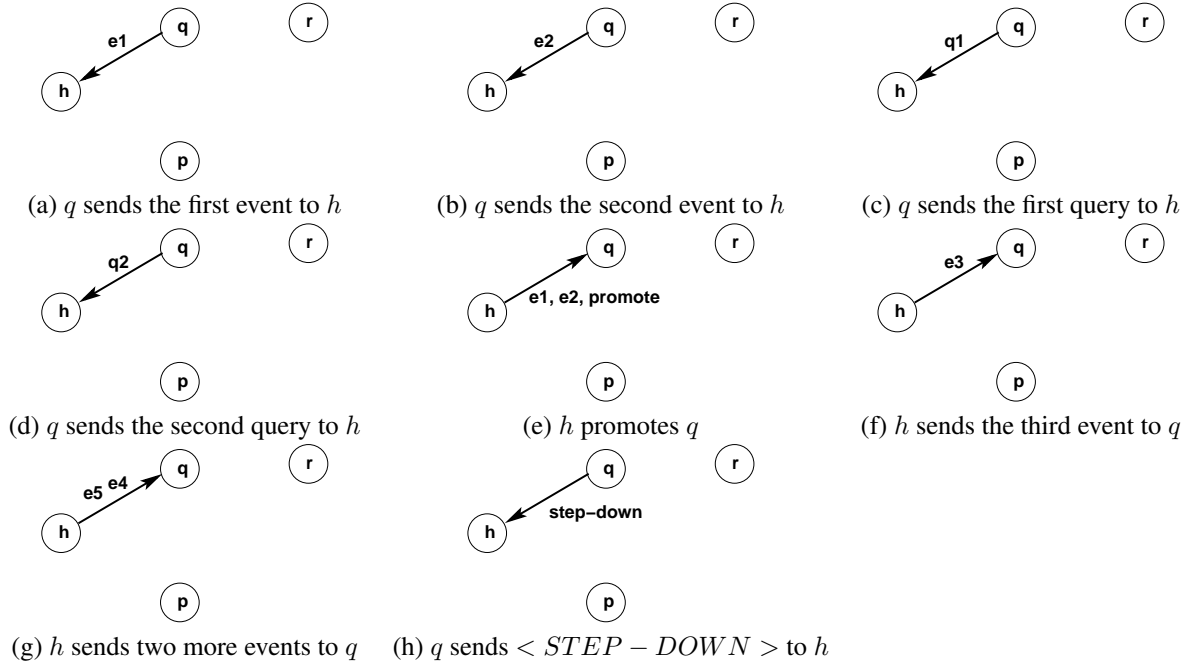


Figure 1: An Example of Promotion and Step-Down in DCS with Diffuse Caching

$h$	cost	num_N-Q	num_Q	is_Cached
$p$	0	1	0	F
$q$	0	1	0	F

(a) After first event

$h$	cost	num_N-Q	num_Q	is_Cached
$p$	0	2	0	F
$q$	0	2	0	F

(b) After second event

$h$	cost	num_N-Q	num_Q	is_Cached
$p$	0	2	0	F
$q$	1	0	2	F

(c) After first query

$h$	cost	num_N-Q	num_Q	is_Cached
$p$	0	2	0	F
$q$	-2	0	0	T

(d) After second query

$q$	cost	num_N-Q	num_Q	is_Cached
$q$	-2	0	2	T
$r$	0	2	0	F

(e) After promotion

$q$	cost	num_N-Q	num_Q	is_Cached
$q$	-1	1	2	T
$r$	0	3	0	F

(f) After third event

$q$	cost	num_N-Q	num_Q	is_Cached
$q$	1	3	2	F
$r$	0	5	0	F

(g) After fifth event

$h$	cost	num_N-Q	num_Q	is_Cached
$p$	0	5	0	F
$q$	0	5	0	F

(h) After  $q$  steps down

Table 1: Information in Tables of  $h$  and  $q$

- Success Reply Rate (RR)

This measures how many queries get their answers successfully. A good algorithm should have a very high percentage.

- Maximum number of events a node has to store/cache (MS)

This reflects the maximum storage a node needs and the smaller the better.

- Average number of events a node has to store/cache (AS)

It is also a measure of storage requirement.

- Maximum communication cost of a node (MC)

The communication cost is computed by adding the energy cost of transmitting messages and receiving messages. According to [17], the power consumption for transmitting is 14.88 mW while reception needs 12.50 mW.

- Average communication cost (AC)

#### 4.1 Single Query Node

The simulation setup is summarised in Table 2. Note that the simulation setup is similar as described in [2]. That is, all the events are detected and stored in the network before any queries are issued. For each of the network size, we generated 10 different networks, each has a different hashing function and query pattern. Each number in the simulation result tables is an average value of the 10 networks.

Node Density	$1node/256m^2$
Radio Range	$40m$
GPSR Beacon Interval	$1s$
GPSR Beacon Expiration	$4.5s$
Planarisation	$GG$
Mobility Rate	$0m/s$
Number of Nodes	50, 100, 150, 200
Simulation Time	$500s$
Query nodes	1
Query Generation Rate	$2qps$
Query Start Time	$242s$
Refresh Interval	$10s$
Event Types	20
Events Detected	10

Table 2: Simulation Setup of the Single Query Node Case

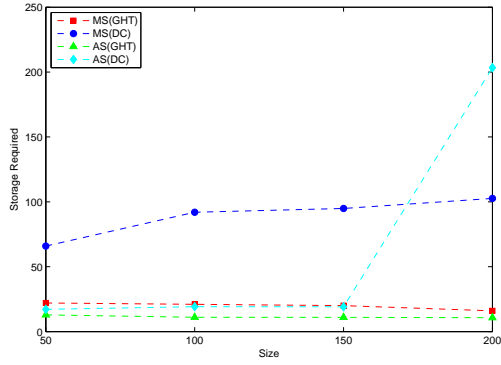


Figure 2: Storage needed in Single Query Node networks

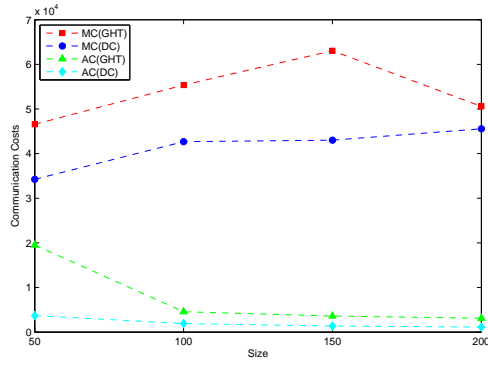


Figure 3: Communication costs in Single Query Node networks

The results are shown in the table 3 and Figure 2, 3. We label our mechanism as DC. From the results, it is clear that our protocol requires more storage but results in lower communication costs, (about 40% of the original GHT). We argue that using more storage will have little impact in the sensor network as long as each node has sufficient storage. In other words, the unused memory in a node is effectively wasted. Therefore, our protocol trades these unused storage into savings in energy by reducing the communication costs.

Size	RR(GHT)	RR(DC)
50	99.89	99.89
100	100	100
150	99.95	99.94
200	99.79	99.79

Table 3: Reply Success Rates in Single Query Node networks

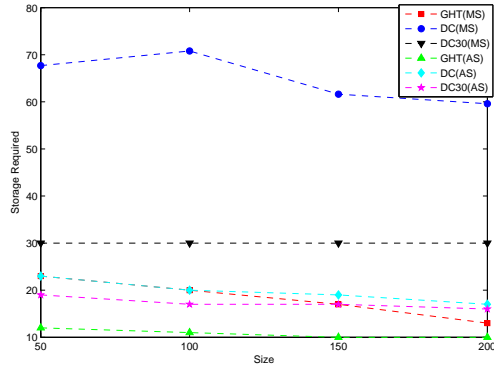


Figure 4: Storage needed in Multiple Query Node networks

## 4.2 Multiple Query Nodes

We then run the simulations in which every node in the network can be a query node, and the query and events are interleaving temporally. Furthermore, we also simulated networks with limited storage of 30 events (labeled as DC30), which is the maximum required by GHT under our experimental setups in all networks. The results are show in Tables 4 and Figure 4, 5.

The results show that even when the storage is limited, DC still outperforms the original GHT by reducing transmission costs to about half. However, the reply success rate of limited storage DC decreases slightly compared to both GHT and DC with unlimited storage. This is because some of the caching nodes do not have enough storage and cannot reply the query with the complete event set. Also, DC’s success rate also decreases compared to GHT. This is due to the scenario when a query has been received by a caching node, but the new events are still in transit from the home nodes to the caching nodes. This lag results in some queries being replied with fewer events than expected. It should be highlighted that the differences in success ratios are very small.

Size	GHT	DC	DC30
50	98.4	99.4	97.9
100	99.3	99.1	98.6
150	99.3	98.8	98.4
200	99.4	98.4	97.9

Table 4: Success Reply Rates in Multiple Query Node networks

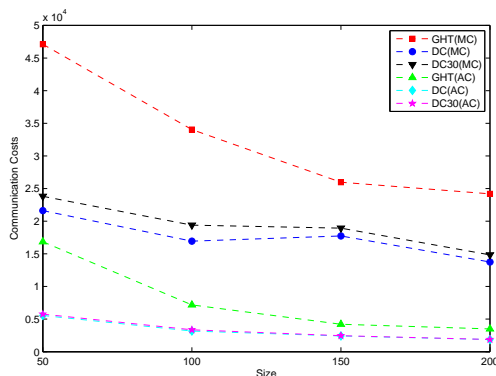


Figure 5: Communication costs in Multiple Query Node networks

## 5 Conclusion

In this paper, we study the problem of distributing events and queries in sensor networks using data-centric storage. We firstly identify the weaknesses of existing approaches and then describe our new protocol which caches events. We measure the performance of our protocol using extensive simulations and compare with GHT, a pioneer work in this area. Simulation results show that our protocol outperforms GHT in the experimental settings. Therefore, we conclude that our protocol is promising for data-centric storage in sensor networks.

## References

- [1] S. Shenker, S. Ratnasamy, B. Karp, R. Govindan, and D. Estrin, “Data-Centric Storage in Sensornets,” in *ACM SIGCOMM HotNets*, 2002.
- [2] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker, “GHT: A Geographic Hash Table for Data-Centric Storage,” in *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, 2002, pp. 78 – 87.
- [3] K. Seada and A. Helmy, “Rendezvous Regions: A Scalable Architecture for Service Location and Data-Centric Storage in Large-Scale Wireless Networks,” in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.
- [4] R. Tamishetty, L. Ngoh, and P. Keng, “An efficient resiliency scheme for data centric storage in wireless sensor networks,” in *IEEE 60th Vehicular Technology Conference*, vol. 4, Fall 2004, pp. 2936 – 2940.

- [5] M.-H. Chan, K.-S. Lui, and V. Tam, "Efficient Event and Query Distribution in Sensor Networks," in *The First International Conference on Collaborative Computing: Networking, Applications and Work-sharing*, 2005.
- [6] X. Li, Y. J. Kim, R. Govindan, and W. Hong, "Multi-dimensional Range Queries in Sensor Networks," in *ACM SenSys*, 2003, pp. 63 – 75.
- [7] J. Newsome and D. Song, "GEM: Graph EMbedding for Routing and Data-Centric Storage in Sensor Networks Without Geographic Information," in *ACM SenSys*, 2003, pp. 76 – 88.
- [8] J. Chen, Y. Guan, and U. Pooch, "An efficient data dissemination method in wireless sensor networks," in *IEEE Global Telecommunications Conference (Globecom)*, vol. 5, 2004, pp. 3200 – 3204.
- [9] B. Greenstein, D. Estrin, R. Govindan, S. Ratnasamy, and S. Shenker, "DIFS: A Distributed Index for Features in Sensor Networks," in *IEEE International Workshop on Sensor Network Protocols and Applications (SNPA)*, 2003.
- [10] D. Braginsky and D. Estrin, "Rumor Routing Algorithm For Sensor Networks," in *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, 2002, pp. 22 – 31.
- [11] X. Liu and Q. H. Y. Zhang, "Combs, Needles, Haystacks: Balancing Push and Pull for Discovery in Large-Scale Sensor Networks," in *ACM SenSys*, 2004.
- [12] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva, "Directed diffusion for wireless sensor networking," in *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, 2005.
- [13] L. Yin and G. Cao, "Supporting Cooperative Caching in Ad Hoc Network," in *IEEE Transactions on Mobile Computing*, vol. 5, no. 1, 2006.
- [14] M. Aly, P. K. Chrysanthis, and K. Pruhs, "Decomposing data-centric storage query hot-spots in sensor networks," in *ACM MOBIQUITOUS*, 2006.
- [15] B. Karp and H. T. Kung, "GPSR: Greedy Perimeter Stateless Routing for Wireless Networks," in *ACM Mobicom*, 2000, pp. 243 – 254.
- [16] H. ying Tyan and et al, "J-Sim," <http://http://www.j-sim.org>.
- [17] C. Schurgers, V. Tsiatsis, S. Ganeriwal, and M. Srivastava, "Topology Management for Sensor Networks: Exploiting Latency and Density." in *International Symposium on Mobile Ad Hoc Networking & Computing*, 2002.