Multiple Controlled Mobile Elements (Data Mules) for Data Collection in Sensor Networks

David Jea, Arun Somasundara, and Mani Srivastava

Networked and Embedded Systems Laboratory,
Department of Electrical Engineering, UCLA
{dcjea, arun, mbs}@ee.ucla.edu

Abstract. Recent research has shown that using a mobile element to collect and carry data mechanically from a sensor network has many advantages over static multihop routing. We have an implementation as well employing a single mobile element. But the network scalability and traffic may make a single mobile element insufficient. In this paper we investigate the use of multiple mobile elements. In particular, we present load balancing algorithm which tries to balance the number of sensor nodes each mobile element services. We show by simulation the benefits of load balancing.

1 Introduction

Recently there has been an increased focus on the use of sensor networks to sense and measure the environment. Some practical deployments include NIMS [1], James Reserve [2]. Both these deployments focus mainly on the problem of habitat and environment monitoring. In most cases the sensors are battery-constrained which makes the problem of energy-efficiency of paramount importance.

There are multiple ways in which the sensor readings are transferred from the sensors to a central location. Usually, the readings taken by the sensor nodes are relayed to a base station for processing using the ad-hoc multi-hop network formed by the sensor nodes. While this is surely a feasible technique for data transfer, it creates a bottleneck in the network. The nodes near the base station relay the data from nodes that are farther away. This leads to a non-uniform depletion of network resources and the nodes near the base station are the first to run out of batteries. If these nodes die, then the network is for all practical purposes disconnected. Periodically replacing the battery of the nodes for the large scale deployments is also infeasible.

A number of researchers have proposed mobility as a solution to this problem of data gathering. Mobile elements traversing the network can collect data from sensor nodes when they come near it. Existing mobility in the environment can be used [3, 4, 5, 6] or mobile elements can be added to the system [7, 8, 9], which have the luxury to be recharged. This naturally avoids multi-hop and removes the relaying overhead of nodes near the base station. In addition, the sensor nodes no longer need to form a connected network (in a wireless sense). Thus a network can be deployed keeping only the sensing aspects in mind. One need not worry about adding nodes, just to make sure that data transfer remains feasible.

V. Prasanna et al. (Eds.): DCOSS 2005, LNCS 3560, pp. 244–257, 2005.

But this technique of using mobile elements comes at a cost of increased latency for data collection. As a result, this is more suitable for delay tolerant networks [10], for instance, habitat monitoring [1, 2] mentioned earlier.

In this paper we consider using multiple mobile elements for purposes of data collection. We first briefly review our prior work on single mobile element [7] in Sect. 3. Next we describe the necessity of using multiple mobile elements for scalability reasons in Sect. 4. When multiple mobile elements are used to collect data from sensor nodes (# sensor nodes $\gg$ # mobile elements), it is better to have the mobile elements serve more or less the same number of nodes. We describe these ideas of load balancing in Sect. 5. The operation with load balancing is described precisely in Sect. 6. We present simulation methodology and results in Sect. 7, and finally end with conclusions and some directions for future work in Sect. 8. We begin with presenting the related work.

2 Related Work

Various types of mobility have been considered for the mobile element. These can be broadly classified as random, predictable or controlled. An algorithm for routing data among randomly mobile users was suggested in [11] where data is forwarded to nodes which have recently encountered the intended destination node. Random motion of mobile entities was also used for communication in [4, 5], where the mobile entities were zebras and whales. An important difference in these two from others is that the sensor nodes themselves are mounted on the mobile entities (animals), and the goal is to track their movements. In [3], randomly moving humans and animals act as "data mules" and collect data opportunistically from sensor nodes when in range. However, in all cases of random mobility, the worst case latency of data transfer cannot be bounded.

Predictable mobility was used in [6]. A network access point was mounted on a public transportation bus moving with a periodic schedule. The sensor nodes learn the times at which they have connectivity with the bus, and wake up accordingly to transfer their data.

Controlled mobility was considered in our previous work [7], where a robot acts as a mobile base station. The speed of the mobile node was controlled to help improve network performance. This is briefly summarized in the next section. Controlled mobility was also used in [8], where a mobile node is used to route messages between nodes in sparse networks. However, all nodes are assumed to have short range mobility and can modify their locations to come within direct range of the mobile node which has long range mobility and is used for transferring data.

In [12], mobile nodes in a disconnected ad hoc network modify their trajectories to come within communication range, and [13] considered moving the intermediate nodes along a route, so that the distances between nodes are minimized and lower energy is used to transmit over a shorter range. This system also assumes that all nodes are mobile, which may be expensive or infeasible in many deployments where node locations depend on sensing or application requirements. A mobile base station was also used in [9] to increase network lifetime. A scheduling problem for the mobile node with buffer constraints on static nodes and variable sampling rates at each static node is studied

in [14]. Connectivity through time protocols were proposed in [15] that exploit robot motions to buffer and carry data if there is no path to the destination.

Henceforth, we will use the term **data mule**, borrowed from [3] to denote a mobile element.

3 Single Data Mule

For sake of completeness and having continuity, we briefly describe the single controlled data mule approach in this section. Sensor nodes are deployed in an area, and are sampling the physical phenomenon. There is a data mule whose job is to collect data from these sensor nodes. The data mule moves in a straight line up and down. The operation can be divided into two parts: Network algorithms (specifies how sensor nodes interact with each other and the data mule) and Motion Control algorithms (specifies how the data mule moves)

3.1 Network Algorithms

The network may be such that some sensor nodes may never hear the data mule directly. In this situation, they transfer the data through other nodes, which can directly hear the data mule. The algorithm can be divided into three phases:

1. **Initialization:** This is used to find out the number of hops each node is from the path of the data mule (initialized to ∞). The data mule moves broadcasting the beacons (with hop count as 1). All nodes which hear it mark the hop count, and also rebroadcast it (after incrementing the hop count). A node which hears a beacon with hop count less than what has, updates itself (also noting the node from which it came). At the end of this phase, all nodes know if they are on path of data mule (at 1 hop). If they are not on path of the data mule, they know the parent through which to reach a node which is on path. Basically, this is tree building, with number of trees being equal to the number of nodes on path of the data mule. All nodes are members of exactly one tree.
2. **Local multihops:** Each of the trees formed above do a local multihop within themselves, with the root of the tree collecting data of its children nodes. Directed Diffusion [16] is suitable for this.
3. **Data Collection by Data Mule:** After one round of initialization phase, the data mule moves polling for data. The nodes which hear the data mule respond with the data (their own and that of their children). To prevent loss of data due to data mule going out of range, we can have a acknowledgement based scheme.

Once the initialization phase is over, the other two proceed in parallel.

3.2 Motion Control Algorithms

Motion can be controlled in two dimensions: space (where the data mule goes), and time (how or what speed the data mule moves). By fixing the path to be a straight line, we need to decide the speed. Two options are possible:

1. Fix the round trip time (RTT) of the data mule. With this, there are few approaches possible.
   - We can traverse the path at a fixed speed (at which we get maximum efficiency from the data mule). Suppose this takes time $T(< RTT)$. Then we have $RTT - T$ spare time. We can divide this time equally among all nodes. The data mule would stop for this time at each node. We do not assume that the data mule knows the node locations. So we stop when we first hear from a node.
   - Alternately, we can cover the trail at constant speed (Length of path / RTT), and not stop at any node.
   - Finally, we can also have an adaptive speed control algorithm, where the data mule would normally move at twice the speed above. This leaves $RTT/2$ time to service the nodes by stopping at them. This time can be divided among a subset of the nodes, from which the mule had collected less data than a threshold in the previous round. Thus unlike the first case, the sets of nodes at which the mule would stop would change with each round.

   We have not gone into details of the above algorithms. Our earlier paper [7] can be referred where we have implemented a version of these ideas. Although two nodes are one hop from the data mule, the time which the data mule stays in their contact may vary, depending on the distance of the node to the path. One way to take care of this is adaptive motion control mentioned above. The mule would have collected less data from a node which is far from the path, and in the next round would stop at it, giving it more time. Nevertheless, this approach only tries to maximize the amount of data collected. It does not guarantee that all the generated data is collected.
2. Give an equal amount of service time to each node (with the mule stopping for this amount of time at each node). The service time for a node can be set equal to Buffer size / Communication data rate. This would ensure that the data mule is able to collect all the data. The mule can collect data even when it is moving, and this can be considered as a bonus. As mentioned before, all nodes need not be one hop from the data mule. For this, the root of the trees (all roots are one hop from the data mule) can be given time equivalent to the number of nodes in its tree. The RTT would depend on number of nodes in the network.

4 Multiple Data Mules

The single data mule approach presented in the previous section does not scale well. Suppose the density of the network increases due to increasing number of nodes. Considering the approach of fixed round trip time for data mule, there are more nodes from which data has to be collected, in the same amount of time. This leads to loss of data due to buffer overflows at the nodes. If the second approach of stopping at each node is used, the data mule will take a longer time to complete a round. In this case, although at time of each service, the buffer of a node is cleared, it may not be possible for the data mule to return to this node before its buffer fills again. Again this leads to loss of data. Another issue arises if the network is deployed over a larger area. The distance

over which the data mule moves increases. The battery capacity may not be sufficient for moving this length, requiring recharge on the path.

These problems can be addressed by using multiple data mules. A trivial solution would be dividing the area into equal parts and having one data mule in each. This solves the problem if the nodes are uniformly randomly deployed, so that each mule gets approximately same number of nodes to service. Each mule covers the same area. Now each mule can independently run the same single mule algorithms presented in the previous section. To analytically calculate the required number of data mules, let us define the following:

- num_nodes nodes are deployed in an area of $l \times l$ units.
- A data mule moves in a straight line from one end to another and back at speed s.
- Time to fill a node's buffer is $buffer\_fill\_time$. This can be calculated using the sampling period and buffer size. We assume that all nodes are sampling at the same rate.
- Time for the data mule to empty a node's buffer is $service\_time$ given by $buffer\_size/communication\_data\_rate$. Let us assume the second form of motion control (Sec. 3.2), where the data mule stops at each node for this amount of time.
- Round trip time (RTT) for the data mule will be $(l/s)+(num\_nodes \times service\_time)+(l/s)$.

Now, if $RTT \leq buffer\_fill\_time$, one data mule will suffice. Otherwise, $\lceil RTT/buffer\_fill\_time \rceil$ mules would be required.

Two things are to be noted with respect to the last calculation. Firstly, in the expression for round trip time, the first two terms denote the time it takes the data mule to move from one side of the area to the other along with time for data collection. The last term denotes the time to come back to the starting point. There is no data collection in the reverse path, so as to approximate the whole motion to a closed loop. Secondly, some nodes may not be able to communicate directly to the data mule. The data of these nodes will be available at root of the tree this node belongs to using local multihopping.

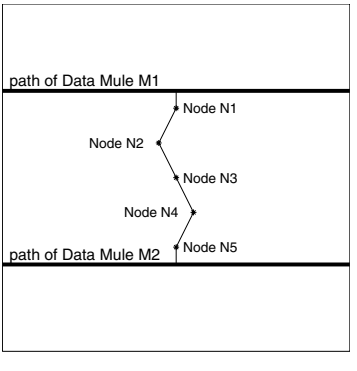

Fig. 1. Nodes between 2 mules

Table 1. Hop count at nodes in Fig. 1

Nodes	Data Mule M1	Data Mule M2
N1	1	5
N2	2	4
N3	3	3
N4	4	2
N5	5	1

(The root will be on path of data mule as mentioned in previous section). A related issue is the fact that a node will belong to more than one tree (one tree per data mule). In such cases, the node will send data towards the closer data mule. This is illustrated in Fig. 1, with the 2 data mules moving on straight line paths. Table 1 shows the hop count variable at each of the sensor nodes $N1 - N5$ due to the two data mules. As can be seen $N1, N2$ will be serviced by $M1$, and $N4, N5$ by $M2$. $N3$ is at equal hop count from both the data mules. Such ties can be broken randomly.

5 Load Balancing

The previous section made the case for using multiple data mules. If the nodes are uniformly randomly distributed, then the obvious thing to do is to divide the area into equal regions.

But in practice, at real deployments it may not be so trivial. Firstly, the nodes need not be uniformly deployed. They will be placed by the field experts such as the biologists. They would want to deploy nodes in areas where they suspect interesting activities to take place. This will naturally lead to non-uniform placement. In addition, in these environments it may not be feasible to have the data mule trails according to system designer's requirements.

5.1 Problem Description

We are given a set of nodes deployed in an area, and straight line paths for the data mules $(M1, M2, ...)$ to move (which are not necessarily equally spaced). We assume for simplicity that each node is one hop away from atleast one data mule (and atmost two data mules). Fig. 2 describes the scenario with three data mules $M1, M2, M3$. The various regions are marked $A - H$. Table 2 describes the various regions, and the data mules they can talk to. The regions which are serviced by a single data mule have no choice. The nodes in these regions will be called non_shareable nodes. But the nodes in regions which are serviced by 2 data mules, can be attached to either of them. Such nodes are called shareable nodes. The goal is to find the data mule assignment for these shareable nodes, so that each data mule services approximately same number of nodes.

5.2 Why It Is Important

Consider a simple scenario with 50 nodes and 2 data mules $M1, M2$. Suppose $M1$ has 25 non_shareable nodes, $M2$ has 5 non_shareable nodes, and 20 nodes are shared between them. If these 20 nodes are equally divided between the two data mules, $M1$ will end up servicing 35 nodes and $M2$ 15 nodes. Consider $M1$. If we use the approach of fixed round trip time for the data mule, the time given to each node will be reduced. On the other hand, if we use the approach of stopping at each node (for amount of time required to empty its buffer), the round trip time of the data mule will increase, leading to possibility of buffer overflows, when the data mule returns to service them. Instead, if both the data mules serviced 25 nodes, the above mentioned problem will be solved. For this to happen, all 20 shareable nodes are to be serviced by $M2$.

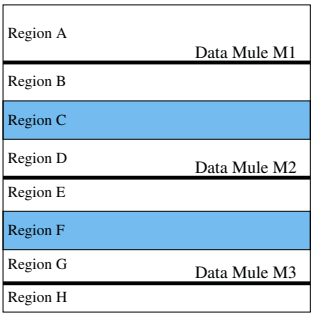

Fig. 2. Problem description

Table 2. Illustration of Fig. 2

Region	Visible Data Mule(s)	Region type
A	M1	non_shareable
B	M1	non_shareable
C	M1, M2	shareable
D	M2	non_shareable
E	M2	non_shareable
F	M2, M3	shareable
G	M3	non_shareable
H	M3	non_shareable

Load balancing is common concept in distributed systems [17]. In our case, the tasks are the servicing of sensor nodes, and the processing elements (PEs) are the data mules. In addition, there are constraints on the tasks, as to which PEs can process them. This refers to the fact that sensor nodes (if they are shareable) can only be serviced by 2 data mules.

6 Multiple Data Mules with Load Balancing

We now describe the multiple data mule approach with load balancing. This is one of the main contributions of this paper. This can be divided into five parts: initialization, leader election, load balancing, assignment, and data collection.

6.1 Initialization

The data mules make a round broadcasting the beacons. The nodes which can hear, reply back with their id's. The data mules note down the list of distinct node id's they got the response from. At the end of this round, each data mule has a list of nodes which are one hop from its path.

6.2 Leader Election

We assume that the data mules are equipped with powerful radios, and can communicate with each other. They elect a leader among themselves, and everyone sends the information gathered in the initialization round to the leader. The data mule with the smallest id becomes the leader in our case.

6.3 Load Balancing

The leader data mule has the information of all the data mules. For each data mule i, the leader can classify its nodes into 2 classes: shareable nodes and non_shareable nodes. The shareable nodes can further be classified as being shared with previous or next data mule. Let us define an array structure DM, of size equal to number of data mules (N). The structure $DM[i]$, denoting data mule i has the following members:

- $non\_shareable\_nodes$ denotes the set of nodes it is solely responsible.
- $shareable\_nodes\_neg$ denotes the set of nodes it shares with the previous data mule $i-1$. For $DM[1]$, this is a null set.
- $shareable\_nodes\_pos$ denotes the set of nodes it shares with the next data mule $i+1$. For $DM[N]$, this is a null set.
- $non\_shareable\_load$ denotes the size of the set $non\_shareable\_nodes$.
- $shareable\_load\_neg$ denotes the size of the set $shareable\_nodes\_neg$.
- $shareable\_load\_pos$ denotes the size of the set $shareable\_nodes\_pos$.

The above variables are calculated by the leader, and form the input to the load balancing algorithm. It may be noted that $DM[i].shareable\_nodes\_neg$ & $DM[i-1].shareable\_nodes\_pos$ are same. Similarly, $DM[i].shareable\_nodes\_pos$ is same as $DM[i+1].shareable\_nodes\_neg$.

- $my\_shareable\_load\_neg$ denotes the number of nodes this data mule is responsible for out of $shareable\_load\_neg$.
- $my\_shareable\_load\_pos$ denotes the number of nodes this data mule is responsible for out of $shareable\_load\_pos$.
- my_total_load is the total number of nodes this data mule will be responsible for. It is the sum of $non\_shareable\_load$, and the above two variables.
  These three variables evolve as the algorithm proceeds.

Initially all data mules are in the same single group, with first mule called the $start\_mule$, and last one called the end_mule. The idea is to make the load of (number of nodes serviced by) each data mule equal to the average load of that group. This may not always be possible. For instance, consider the simple scenario with 50 nodes and 2 data mules $M1, M2$. Suppose $M1$ has 35 non_shareable nodes, $M2$ has 5 non_shareable nodes, and 10 nodes are shared between them. The best possible result would be to assign all the 10 shareable nodes to $M2$, making it responsible for 15 nodes, and $M1$ for 35 nodes. In such a case, we divide the original group into two, and try to balance the load of each group recursively. The recursion is terminated when we reach the last mule of the group.

Table 3. Meaning of flag variables for a mule

Flag	If Flag is $TRUE$	If Flag is $FALSE$
$start\_flag$	$my\_shareable\_load\_neg = shareable\_load\_neg$	$my\_shareable\_load\_neg = 0$
end_flag	$my\_shareable\_load\_pos = shareable\_load\_pos$	$my\_shareable\_load\_pos = 0$

The group splitting happens when we reach a mule such that the *minimum* load it should take is more than the group average. We form two groups with this mule belonging to the first group. This mule becomes the end_mule of the first group. Also, the load this mule shared with the next mule is given completely to the next mule, which becomes the $start\_mule$ of the second group. A group can also split when we reach a mule such that the *maximum* load it can take is less than the group average. Here again we form two groups with this mule belonging to the first group. But now this mule takes

ALGORITHM: Load_Balance(start_mule, end_mule, start_flag, end_flag)

1. Initialize $group\_has\_split$ to $FALSE$
2. Calculate the average load of this group $group\_avg$, as shown in Fig. 4.
3. Reset the following variables:
   - $DM[start\_mule..end\_mule].my\_shareable\_load\_neg$
   - $DM[start\_mule..end\_mule].my\_shareable\_load\_pos$
   - $DM[start\_mule..end\_mule].my\_total\_load$
4. Repeat the following **for** $i = start\_mule..end\_mule$
   (a) if $group\_has\_split$ is $TRUE$, return.
      - This is to terminate recursion, when we come to next iteration of for loop during back tracking of recursion.
   (b) Calculate the minimum load that can be assigned to this mule.
      i. If $i = start\_mule$ AND $start\_flag = TRUE$
         A. $DM[i].my\_shareable\_load\_neg = DM[i].shareable\_load\_neg$
      ii. Else If $i \neq start\_mule$
         A. $DM[i].my\_shareable\_load\_neg =$
            $DM[i].shareable\_load\_neg - DM[i-1].my\_shareable\_load\_pos$
      iii. $DM[i].my\_total\_load =$
            $DM[i].non\_shareable\_load + DM[i].my\_shareable\_load\_neg$
   (c) If $i = end\_mule$
      i. If $end\_flag = TRUE$
         A. $DM[i].my\_shareable\_load\_pos = DM[i].shareable\_load\_pos$
         B. $DM[i].my\_total\_load+ = DM[i].my\_shareable\_load\_pos$
      ii. **Return**
   (d) If $DM[i].my\_total\_load > group\_avg$
      - The load on this mule is more than group average; we split into two groups.
      i. Set $group\_has\_split = TRUE$
      ii. Call **Load_Balance(start_mule, i, start_flag, FALSE)**
      iii. Call **Load_Balance(i+1, end_mule, TRUE, end_flag)**
         - All nodes shared between mules i and $i+1$ is taken by mule $i+1$.
   (e) Else (i.e. if $DM[i].my\_total\_load \leq group\_avg$)
      i. Calculate the extra load that can be given to this mule.
         - $extra\_load = group\_avg - DM[i].my\_total\_load$
         - $DM[i].my\_shareable\_load\_pos =$
            $\min(extra\_load, DM[i].shareable\_load\_pos)$
         - $DM[i].my\_total\_load+ = DM[i].my\_shareable\_load\_pos$
      ii. If $DM[i].my\_total\_load < group\_avg$
         - The maximum load this mule can have is less than the group average. Here also, we split into two groups.
         A. Set $group\_has\_split = TRUE$
         B. Call **Load_Balance(start_mule, i, start_flag, TRUE)**
         C. Call **Load_Balance(i+1, end_mule, FALSE, end_flag)**
            - All nodes shared between mules i and $i+1$ is taken by mule i.
      iii. Else (i.e. if $DM[i].my\_total\_load = group\_avg$, **continue**
         - We continue to check mule $i+1$

END

Fig. 3. Load Balancing Algorithm

ALGORITHM: Calculate group average

– Input parameters: $start\_mule$, end_mule, $start\_flag$, end_flag
– Procedure:
  1. Initialize $group\_load = \sum_{i=start\_mule}^{end\_mule} DM[i].non\_shareable\_load$
  2. Switch depending on ($start\_flag$, end_flag)
     - $(TRUE, TRUE)$
       * $group\_load+ = \sum_{i=start\_mule}^{end\_mule} DM[i].shareable\_load\_neg+$
         $DM[end\_mule].shareable\_load\_pos$
     - $(TRUE, FALSE)$
       * $group\_load+ = \sum_{i=start\_mule}^{end\_mule} DM[i].shareable\_load\_neg$
     - $(FALSE, TRUE)$
       * $group\_load+ = \sum_{i=start\_mule}^{end\_mule} DM[i].shareable\_load\_pos$
     - $(FALSE, FALSE)$
       * $group\_load+ = \sum_{i=start\_mule+1}^{end\_mule} DM[i].shareable\_load\_neg$
  3. $group\_avg = \frac{group\_load}{end\_mule - start\_mule + 1}$
– Return $group\_avg$

END

Fig. 4. Algorithm for calculating $group\_avg$ in step 2 of Fig. 3

all the load it shares with the next mule. We use two flags $start\_flag$, and end_flag to denote these states. Mules are affected by these flags only if they are $start\_mule$, or end_mule respectively. Table 3 describes these variables. Initially, when there is only one group comprising all the mules, the values of these flags do not matter, as the $start\_mule$ (mule $\#1$) does not have any predecessor, and end_mule (mule $\#N$) does not have any successor.

The precise algorithm is given in Fig. 3. Initially, it is invoked with parameters $(1, N, FALSE, FALSE)$[1]. The algorithm has comments explaining each of the steps, and is also described below. One thing to be noted is the use of local boolean variable $group\_has\_split$. There are recursive calls inside the *for* loop in step 4. If it goes inside recursion, there is no more meaning for the current group. When control comes back to this *for* loop again during backtracking of recursion, we should not process the remaining mules in the *for* loop. Step $4.a$ achieves this.

We begin by calculating the group average in step 2, making sure not to count shareable nodes twice. The average depends on the two flag values, as shown in Fig. 4. We next run a loop for all nodes in the group. First, we calculate the minimum load the mule under consideration should take (step $4.b$). In particular, if this mule is not the $start\_mule$, $4.b.ii.A$ calculates the part of of shareable_load which the previous mule did not take.

The group splitting can happen in two cases. Firstly, if the minimum load (which was calculated in step $4.b$) that has to be assigned to the mule under consideration is more than the group average. When this happens we break into two groups, and the

[1] as mentioned previously, the two flags do not matter initially.

mule under consideration becomes part of the first group. This is shown in $4.d$ in the algorithm. We recursively call the algorithm for the two groups. If the above does not happen, we try to assign some shareable_load , which this shares with the next mule, as shown in $4.e.i$. Now the other reason of splitting can arise. If the maximum load that can be assigned to this mule is less than the group average, we split into two groups, as shown in $4.e.ii$.

The current recursion ends when we reach end_mule in step $4.c$. In addition, if the end_flag is $TRUE$, we add some more load, as shown in $4.c.i.A$. This is in accordance to Table 3.

The worst case complexity of this load balancing algorithm is $O(n^2)$, where n is the number of data mules. This occurs when the group splitting always happens such that one of the resultant groups has only one mule, and the other group has all the remaining mules.

6.4 Assignment

The load balancing algorithm of the previous subsection outputs three counts for each data mule: $my\_shareable\_load\_neg$, $my\_shareable\_load\_pos$, my_total_load. Now we have to find the corresponding sets i.e.$my\_shareable\_nodes\_neg$ and $my\_shareable\_nodes\_pos$. Two data mules would be sharing some nodes. These nodes are ordered by node id. The idea is to assign the first part of this ordered set to the first mule (resulting in $DM[i].my\_shareable\_nodes\_pos$), and the second part to the second mule (resulting in $DM[i+1].my\_shareable\_nodes\_neg$). The size of the two parts depends on the counts mentioned above. Finally, each data mule is responsible for nodes in the three sets: $non\_shareable\_nodes$, $my\_shareable\_nodes\_neg$, and $my\_shareable\_nodes\_pos$. After the assignment has taken place, the leader can inform all the data mules, the set of nodes they have to service.

6.5 Data Collection

With the assignment done, the data mules traverse their paths, polling for data. The shareable sensor nodes do not know which of the two data mules they belong to. The nodes respond for data when they hear the poll packet. The data mule will send back an acknowledgement only if it is responsible for servicing that node. The sensor node marks the data mule from which it hears an acknowledgement, and does not respond to poll packets from the other data mule in future.

7 Simulation Methodology and Results

We now present our simulation methodology. We implemented our algorithms in TinyOS [18]. The simulator used is TOSSIM [19]. The advantage of this combination is that, the same TinyOS code can be put on real sensor nodes. To simulate mobility, we use tython [20]. We consider three schemes for sharing the shareable load between data mules.

Table 4. Simulation Results

	Data Mule $M1$	Data Mule $M2$	Data Mule $M3$	Data Mule $M4$
$non\_shareable\_load$	13	5	5	9
$shareable\_load\_neg$	0	3	3	2
$shareable\_load\_pos$	3	3	2	0
FCFS	16	8	5	11
Equal Sharing	15	8	7	10
Load Balancing	13	9	9	9

1. First Come First Serve (FCFS): The shareable sensor node will get attached to the data mule from which it hears the beacon packet first in the Initialization round of data mules (Sect. 6.1).
2. Equal sharing: Each adjacent data mules have a set of shareable nodes between them. Here, half the shareable nodes are assigned to one data mule, and the other half to the other mule.
3. Load balancing: Result of applying the load balancing algorithm of Sect. 6.3.

For the simulation topology, we had 40 sensor nodes, and 4 data mules, $M1 - M4$. The nodes are randomly distributed inside the 100x100 grid region from $(24, 30)$ to $(76, 70)$. $M1$ moves from $(32, 10)$ to $(32, 90)$, $M2$ moves from $(44, 90)$ to $(44, 10)$, $M3$ moves from $(56, 10)$ to $(56, 90)$ and $M4$ moves from $(68, 90)$ to $(68, 10)$. To have effect of closed-loop path, whenever a data mule reaches its end point, we immediately place it at the starting point, from where it starts moving again. The nodes are placed inside a smaller region than whole grid to avoid edge-effects.

The result after the leader mule ($M1$) gets information from other data mules is shown in upper half of Table 4. This result depends on topology. Now we execute the three schemes of load sharing presented above, resulting in my_total_load values shown in lower half of Table 4. As can be seen, the result of load balancing does not necessarily result in equal distribution of load, because non_shareable_load of $M1$ is more than the group average. This results in group splitting, and we end up balancing the load of the second group.

With these three node assignments to mules, we ran the experiment for 5 rounds, with the first strategy mentioned in Sect. 3.2 of fixed round trip time. The RTT was set to 120 time units, and it took each mule 40 time units to complete a round. This gave 80 time units for stopping at the nodes, which was divided equally among the nodes to be serviced. Fig. 5 shows the average number of packets received per node per round, at each of the data mules. Results are shown for the three assignments. It is evident that load balancing leads to more uniformity. Although data mules $2, 3, 4$ service the same number of nodes (9) in the load balanced case, we see a minor variation. This is due to fact that we are collecting data even when moving, in addition to when being stopped. So if the nodes assigned to $M4$ are closer to its path (when compared to nodes assigned to $M3$ and its path), we end up collecting more data at $M4$.

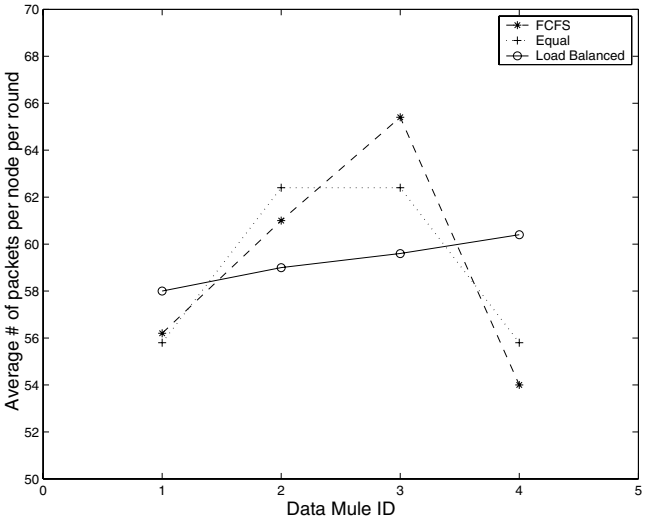

Fig. 5. Simulation Results

8 Conclusions and Future Work

Deployments of sensor networks are taking place. Using a controlled mobile element is a promising approach to collect data from these sensor nodes. We showed that as the network scales, using a single mobile element may not be sufficient, and would require multiple of them. The sensor nodes and (or) the mobile elements may not be uniformly placed in practice, necessitating the use of load balancing, so that each mobile element as far as possible, serves the same number of sensor nodes. We gave a load balancing algorithm, and described the mechanism these multiple mobile elements can be used. Finally we presented simulation results justifying our approach.

The work presented here can be extended in many directions. For load balancing, we can remove the assumption that each sensor node can talk to at least one mobile element. This will lead to case similar to Figure 1. Now, when doing load balancing, we also need to consider the cost of doing multihop, to reach either of the mobile elements. We can also extend to cases where mobile elements can be added or removed once the system is in operation. Node dynamics can be handled by running the initialization and load balancing periodically.

References

1. Kaiser, W.J., Pottie, G.J., Srivastava, M., Sukhatme, G.S., Villasenor, J., Estrin, D.: Networked Infomechanical Systems (NIMS) for Ambient Intelligence. Technical Report 31, Center for Embedded Networked Sensing, UCLA (2003)
2. Cerpa, A., Elson, J., Estrin, D., Girod, L., Hamilton, M., Zhao, J.: Habitat Monitoring: Application Driver for Wireless Communications Technology. In: SIGCOMM Workshop on Data Communications in Latin America and the Caribbean. (2001)

3. Shah, R.C., Roy, S., Jain, S., Brunette, W.: Data MULEs: Modeling a Three-tier Architecture for Sparse Sensor Networks. In: IEEE Workshop on Sensor Network Protocols and Applications (SNPA). (2003)
4. Juang, P., Oki, H., Wang, Y., Martonosi, M., Peh, L., Rubenstein, D.: Energy-efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with Zebranet. In: ACM ASPLOS. (2002)
5. Small, T., Haas, Z.: The Shared Wireless Infostation Model-A New Ad Hoc Networking Paradigm (or Where there is a Whale, there is a Way). In: ACM MobiHoc. (2003)
6. Chakrabarti, A., Sabharwal, A., Aazhang, B.: Using Predictable Observer Mobility for Power Efficient Design of Sensor Networks. In: IPSN. (2003)
7. Kansal, A., Somasundara, A., Jea, D., Srivastava, M., Estrin, D.: Intelligent Fluid Infrastructure for Embedded Networks. In: ACM MobiSys. (2004)
8. Zhao, W., Ammar, M., Zegura, E.: A Message Ferrying Approach for Data Delivery in Sparse Mobile Ad Hoc Networks. In: ACM MobiHoc. (2004)
9. Luo, J., Hubaux, J.P.: Joint Mobility and Routing for Lifetime Elongation in Wireless Sensor Networks. In: IEEE INFOCOM. (2005)
10. Fall, K.: A Delay-Tolerant Network Architecture for Challenged Internets. In: ACM SIGCOMM. (2003)
11. Dubois-Ferriere, H., Grossglauser, M., Vetterli, M.: Age Matters: Efficient Route Discovery in Mobile Ad Hoc Networks Using Encounter Ages. In: ACM MobiHoc. (2003)
12. Li, Q., Rus, D.: Sending Messages to Mobile Users in Disconnected Ad-hoc Wireless Networks. In: ACM MobiCom. (2000)
13. Goldenberg, D., Lin, J., Morse, A.S., Rosen, B., Yang, Y.R.: Towards Mobility as a Network Control Primitive. In: ACM MobiHoc. (2004)
14. Somasundara, A., Ramamoorthy, A., Srivastava, M.: Mobile Element Scheduling for Efficient Data Collection in Wireless Sensor Networks with Dynamic Deadlines. In: IEEE RTSS. (2004)
15. Rao, N., Wu, Q., Iyengar, S., Manickam, A.: Connectivity-through-time Protocols for Dynamic Wireless Networks to Support Mobile Robot Teams. In: IEEE ICRA. (2003)
16. Intanagonwiwat, C., Govindan, R., Estrin, D.: Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In: ACM MobiCom. (2000)
17. Shirazi, B.A., Hurson, A.R., Kavi, K.M.: Scheduling and Load Balancing in Parallel and Distributed Systems. IEEE Computer Society Press (1995)
18. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for network sensors. In: ACM ASPLOS. (2000)
19. Levis, P., Lee, N., Welsh, M., Culler, D.: TOSSIM: accurate and scalable simulation of entire tinyOS applications. In: ACM Sensys. (2003)
20. Demmer, M., Levis, P.: Tython scripting for TOSSIM (2004) Network Embedded Systems Technology Winter 2004 Retreat.