Delay tolerant dynamic data collection over a sensor network

Lionel BARRÈRE and Serge CHAUMETTE and Cyril de PERETTI
lionel.barrere@labri.fr serge.chaumette@labri.fr cyril.de-pereti@etu.u-bordeaux1.fr

LaBRI, Université Bordeaux 1
351 Cours de la Libération
33405 Talence Cedex, France

ABSTRACT

The work presented in this paper consists in experimenting on the collection of dynamic data distributed over a sensor network, where only neighborhood communication is supported (no routing). This can be used to track physical phenomena in the context of battlefields, crisis management or disaster response. The goal is to observe the evolution of the considered distributed variables that make the phenomenon so as to work out emerging trends and potential threats, and to undertake the required actions, i.e. to support situation management.

In this paper we describe the framework that we have designed and implemented and that makes it possible to cope with missing measurements. When some data items are missing, either because of interferences or faulty sensors, it still enables to provide a continuous view of the distributed phenomenon by using interpolation or extrapolation techniques. Our focus is on the framework to support the implementation of such features, not on the final applications. Nevertheless we illustrate its use it by means of a temperature collection application.

The work presented in this paper is carried out at LaBRI (Laboratoire Bordelais de Recherche en Informatique) and more precisely in the SOD (Systèmes et Objets Distribués) research team. It is partly achieved within the framework of the Sarah (Asynchronous services for ad hoc networks) project supported by the ANR[1]. It is also partly supported by the DGA[2] by means of a PhD grant.

1 INTRODUCTION

The goal of this paper is to describe and illustrate the software framework that we have implemented to support distributed data collection and management. Its main originality is that it makes it possible to support interpolation/extrapolation of missing values to provide the user with a continuous view of the information.

The overall setting is as follows. We have a number of sensors that sample the temperature at their location and broadcast these measurements to their neighborhood. Some identified nodes, called clients, collect this information to build a global view showing the temperature at each sensor location. A client may have incomplete information either because some messages got lost or arrive too late due to some external unpredictable event. Therefore a software framework is needed to properly collect these partial and possibly out of order messages and to support the prediction of a replacement for the missing values so as to provide a continuous perception of the collected data. We proceed in two phases.

In the first phase, we assume that each sensor node knows its position and that all the nodes share a global clock. Each sensor takes measurements at some regular interval and broadcasts these measurements. When another node receives such a message, it broadcasts it only once to its own neighborhood. The client nodes collect the received measurements[3]. They do not take any special action about missing or out of date values.

In the second phase, we more realistically consider that only some nodes know their position and share a clock (GPS). The other nodes must estimate their own location according to those of their neighbors. Furthermore, the client nodes can use our software framework to provide replacement for the values that are missing when they are needed, based on the history and the neighborhood of the faulty sensor.

At each client node the collected values are shown as a curve for each sensor and/or as a surface representing the experimentation field.

2 RELATED WORK

Wireless Sensor Networks (WSNs) make it possible to observe natural phenomena such as the evolution of temperature, humidity or to foresee earthquakes or a volcanic eruption [1]. WSNs can avoid human presence in the considered area and thus avoid human impact on the phenomenon to observe (for example animals habitat as shown in [2]).

[1] Agence Nationale de la Recherche

[2] Délégation Générale pour l'Armement

[3] The embedded module that forwards the received messages to the node by the USB interface of the PC furthermore broadcasts them (radio layer).

Collecting data by means of a WSN also decreases the risk for scientists to suffer from the consequences of a disaster (earthquake, nuclear radiation, etc.). It is also cheaper than human based data collection, therefore it makes it possible to continuously observe a phenomenon over a long period of time.

WSNs can also be useful to help for instance to rescue mountaineers or to localize a hidden sniper in a warfare context. In [3], Huang *et al.* propose a system to localize lost mountaineers, each person being equipped with a sensor that regularly records its location. When the person meets a base station, i.e a terminal connected to a static network such as Internet, the sensor transmits the collected data to this base station that in turn transmits them to the rescue teams. In [4], Simon *et al.* propose a system to localize a hidden sniper by sensing what can be assimilated to a bullet noise in a urban area.

3 TECHNICAL FEATURES

Before running the main data collection algorithm we need to synchronize and localize the sensors. Synchronization is required so that the collected measurements can be merged in a manner that makes sense relative to when they have been taken. Space localization is also mandatory because we want to relate our measurements to the exact location where they have been taken. Furthermore, synchronization and localization will also be mandatory when we will be in the process of extrapolating or interpolating a continuous view in spite of possibly missing information (see section 6).

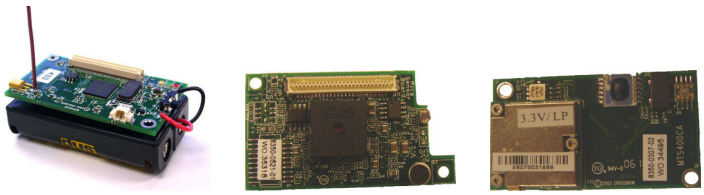
3.1 TARGET PLATFORM

Our WSN is composed of Crossbow Mica2 nodes (fig 1(a)), originally developed at the Berkeley University [5]. They are one of the most widely used platforms. The Mica2 mote has a 7.37 MHz processor, 4kB of RAM, 128 kB of flash memory and a 433 MHz radio transceiver. Sensor boards can be plugged on top of them. Our platform contains two kinds of sensor boards:

1. **Crossbow MTS300** equipped with a temperature and light sensor, a microphone and a 4kHz buzzer (fig 1(b)).

2. **Crossbow MTS420** equipped with a humidity and temperature sensor, an ambient light sensor, a barometric pressure sensor, a two-axes accelerometer and and a Sirf Star II GPS (fig 1(c)).

Mica2 nodes run the **TinyOS** [6] operating system, an open source, event driven OS dedicated to WSNs. TinyOS features include clock and timer management, radio communication and task scheduling.



(a) Mica2 (b) MTS300 (c) MTS420

Figure 1: Mica2 node and sensor boards

3.2 NETWORK ORGANIZATION

We consider two categories of nodes in our network depending on the role that they play:

1. Basic sensor nodes
   The role of basic sensors is to take a measurement, to broadcast it and to serve as forwarders for values that they receive from the other sensors of the network.

2. Client nodes
   Clients are more complex nodes. They are composed of a laptop or a PDA supplied with a Mica2 mote to communicate with basic sensor nodes. The role of clients is almost the same as that of basic sensors but they furthermore collect the measurements that they receive so as to build a global human readable view of the overall phenomenon.

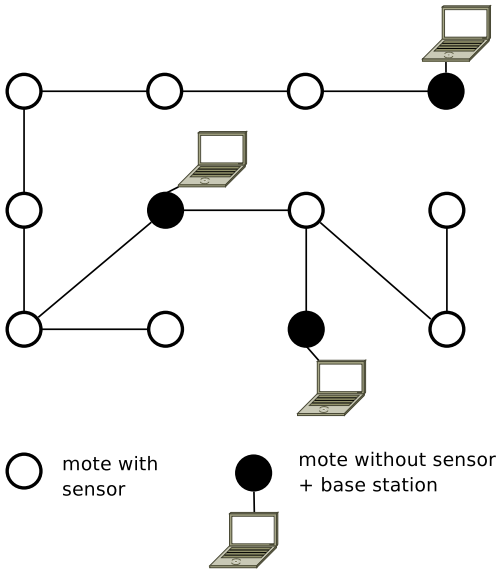


Figure 2: Sensor network

4 SYNCHRONIZATION

Synchronizing the motes is required to be able to achieve a coherent temporal fusion of the measurements taken by different sensors. The synchronization algorithm cannot only

rely on the broadcast and propagation of the time from a single node: several delays that appear during the transmission of a message have to be taken into account and must be compensated. These delays are due to the time needed for the sending mote to access the transmission channel, pack and effectively send the message. For the receiver, these delays are due to the time needed to receive and process the message.

The Flooding Time Synchronization Protocol [7] uses the Time Stamping library of tinyOS in order to get an accurate estimation of the (global) reception time of the message, taking into account the transmission related delays and the local clocks drifts.

In order to have a time synchronized network the authors in [7] rely on one mote to provide a time reference (in this case the time from its own local clock) to the other motes. Since the synchronization algorithm must be tolerant to the failure or the disconnection of any mote, it is thus impossible to assume that the same mote will remain available and can thus definitively act as the synchronization root. Therefore a robust and reliable election algorithm is implemented in the FTSP and ensures the election and if required the dynamic re-election of a root. The elected root is the mote with the smallest ID. The algorithm also ensures that if a mote with a smaller ID is placed in an already synchronized network, it will not try to foster its own local clock as the new global time, but instead it will first synchronize itself on the global time before becoming the new root.

Each synchronization message contains a time stamp, the ID of the synchronization root and a sequence number. When integrating this module within our application we have added a flag so that the synchronization and the measurement application messages can be properly differentiated. When a mote receives a new synchronization message it checks if the root ID of the message is the same or smaller than the root ID it is currently aware of and if the sequence number is greater than its highest sequence number. If this is so the received time stamp is stored and will be used as a reference point. Even though a rough estimation of the global time could be computed with only one synchronization message, it requires more than one reference point to get an accurate estimation. This is why a mote will wait till it has a significant number of reference points (set by default to three) before considering it is synchronized. It can then itself begin emitting synchronization messages.

In the future, we intend to synchronize the motes not on any local clock but on the time acquired by means of GPS sensors that would be available on several nodes of the network. Therefore we are studying how to adapt the FTSP so that it works with multiple synchronization roots.

5 TEMPERATURE COLLECTION ALGORITHM

The goal of our algorithm is to collect temperature knowing where and when each measurement was taken. The measurements must then be delivered through the network to a collecting station. As we want our network to be tolerant to the failure or disconnection of motes, we decided not to implement any routing protocol, thus the measurements are broadcasted (algo. 1) and propagated (algo. 2) by the motes.

Once a mote is synchronized, each time its timer is fired it takes the temperature and broadcasts it. To prevent useless messages (already treated or too old) from overloading the network, a time to live field (TTL) is added to the messages. This TTL, which is decremented at each hop, can be adapted depending on the network radius. In the case where many motes are in the broadcast range of each other, the time to live mechanism will not be enough to properly regulate the number of messages in the network. Therefore we added a table in which we store the IDs of the most recently received messages (each message is identified by the ID of its sender and the time when the measurement was taken). When receiving a message, each mote can thus check if it has already forwarded it and then decide not to broadcast it again.

A mote will not send its own measurements before being synchronized, even though it will forward the messages that it receives. This is typically what happens when we introduce a mote in an already synchronized network.

At the moment we consider that each mote knows its own location but in the future each mote will compute its location by using a triangulation algorithm based on the coordinates of the motes equipped with a GPS sensor.

Algorithm 1 Sensing

```
loop
  if is_synchronized() then
    t ← take_temperature()
    send_temperature(t)
  end if
  wait(intervall)
end loop
```

6 THE PROBLEM OF MISSING INFORMATION

When taking measurements on a battle field it is most likely that some (if not many) samples will be lost. There are many reasons why this may happen:

- communication can be jammed because of interferences coming from other radio equipments of generated by the enemy.
- a sensor can run out of battery
- a sensor can be damaged or even destroyed.

Algorithm 2 Forwarding

```
loop
  msg ← next(receive_queue)
  TTL ← ttl(msg);
  if never_sent(msg) then
    if TTL > 0 or TTL = −1 then
      //TTL==-1 means TTL is not used
      if TTL ≠ −1 then
        set_ttl(msg, TTL − −)
      end if
      send(msg)
    end if
  end if
end loop
```

The question is then to provide the user with a continuous view of the discontinuous collected information.

7 SOFTWARE ARCHITECTURE OF THE FRAMEWORK

The software framework that we have designed and (partly) implemented is composed of a number of modules that support the management of collected measurements and that also provide the user level applications with a number of features that make it possible to take into account missing measurements. For these missing values, the system can provide replacements that can either be interpolated or extrapolated. The overall architecture is shown figure 3

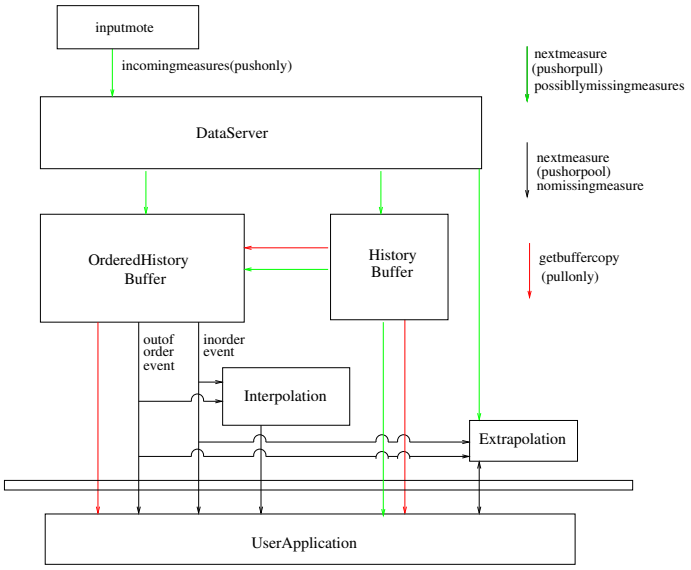


Figure 3: Our component based software architecture

The Data Server. The Data Server is in charge of collecting data from the mote that connects the station to the MANet. It is a basic interface that is just in charge or reading events from the connecting mote and invoking the event listeners that have registered (for instance the History, the Ordered History, or the Extrapolation modules).

The History Buffer. This is basically a buffer that collects and stores the measurements coming from the Data Server. It registers by the server and it is invoked as a listener each time a new measurement is collected. It then stores this data as is, in a FIFO. It can produce this FIFO to the outside either measurement by measurement (green arrow) or as a whole buffer (red arrow).

Any building block can register to this service. In figure 3 the final User Application, and the Ordered History Buffer have declared their interest.

The Ordered History Buffer. The Ordered History Buffer component collects the measurements and orders them. It produces both the data events itself (forwarding) and information about the evolution of the history (black arrows): an event can be received in order and thus be added at the end of the Ordered History Buffer. It can also arrive late, i.e. out of order, and thus be added in its sorted position in the buffer. Once again, it may be the case that an external module needs the whole ordered history as a whole buffer (red arrow).

The goal of the **Interpolation and Extrapolation Modules** is to provide a continuous view of the information even though this is not really the case.

The Interpolation Module. Based on a part of the history of measurements it can work out a reasonable estimation for the next measurement in case it happens to be missing or it does not arrive on time.

The extrapolation module. In the push mode of operation, this module produces a new event (measurement) at a regular interval. In the pull mode, the user application (or any connected module) requests a number of new values. In both modes, when it has not received the appropriate measurements that it could forward, it computes and produces them based on the history.

7.1 CURRENT IMPLEMENTATION STATUS

As of writing we have not implemented the whole framework. The part that we have implemented consists of the data server, the history buffer and the interpolation component. The inter-component communication is currently limited to what was required to produce the results shown in the rest of this paper, i.e. input mote to data server, data server to ordered history buffer, ordered history buffer to interpolation

(both push and pull modes) and to user application, and interpolation module to user application.

8 RESULTS OBTAINED BY USING OUR FRAMEWORK

The goal of this section is to show the ease of use of our framework and what it makes possible to do at the user level.

8.1 EXPERIMENTAL CONDITIONS

To perform our experimentation we setup 20 motes in a 12m x 6m room that has two windows *W1* and *W2* as shown figure 4. After starting the experimentation we opened the two windows and took the temperature every 3 seconds at each mote during 700 seconds. The nodes are separated from each other by approximately 2.5 meters.

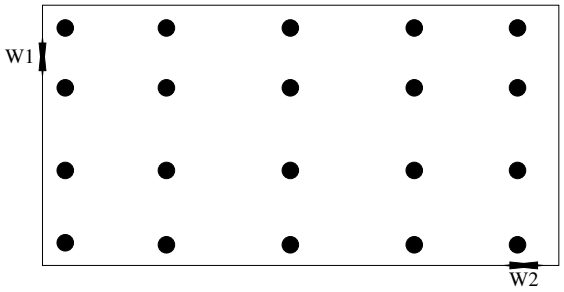


Figure 4: Map of the experimentation room

Figure 5 presents the temperature of the room in Celsius degrees at the beginning of the experiment. It shows that the average temperature is about 23°C and we can see a warm area at the bottom right corner near the window.

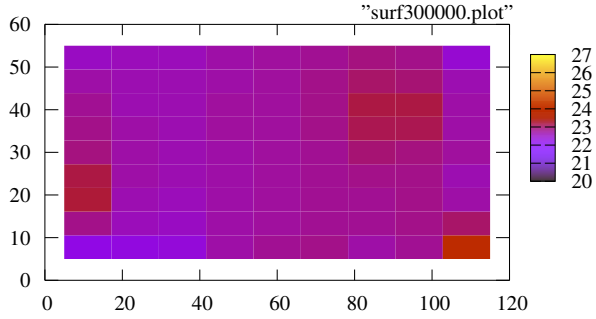


Figure 5: Experimentation room with closed windows

Then, we opened the two windows. Figure 6 shows that the average temperature decreases and that the warm part, near the window, has disappeared.

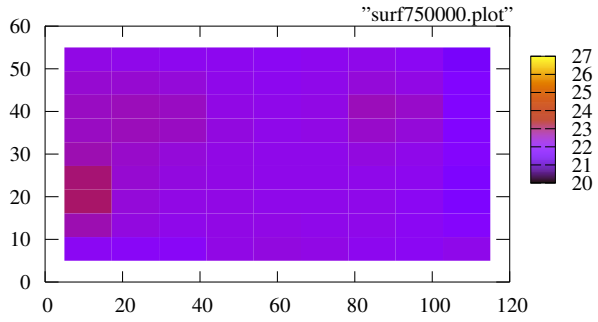


Figure 6: Experimentation room with opened windows

8.2 INTERPOLATING VALUES

Each mote periodically takes and sends temperature. Some messages can be lost because of the unreliable communication medium or because a mote is damaged or off. Figure 7 shows the measurements taken by a node of our WSN. In this case, we have proceeded so that no measurement is missing.

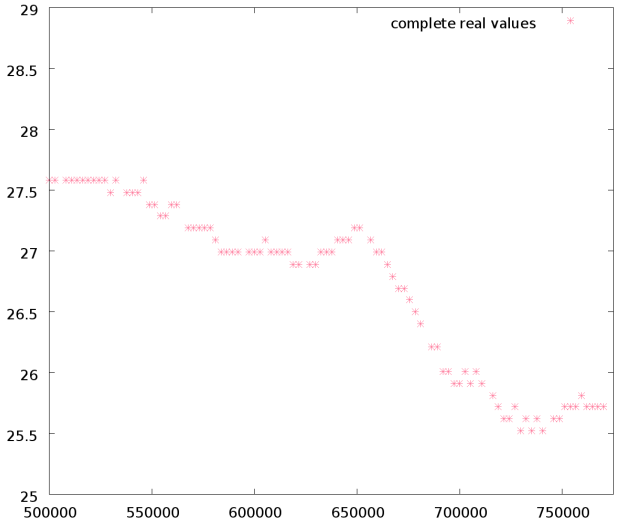


Figure 7: Complete information without interpolation

The curve shown figure 8 is composed of uncomplete real information (blue points) extracted from the real measurements shown figure 7 completed with a cubic spline interpolation (pink points). For the spline interpolation itself, we used a Java implementation available at [8]. Our framework makes it possible to easily change this method of interpolation to use one that would have some knowledge about the studied phenomenon. For example the evolution of a nuclear radiation cloud would not be as linear as the evolution of temperature and would require some knowledge about the phenomenon to reach good interpolation results.

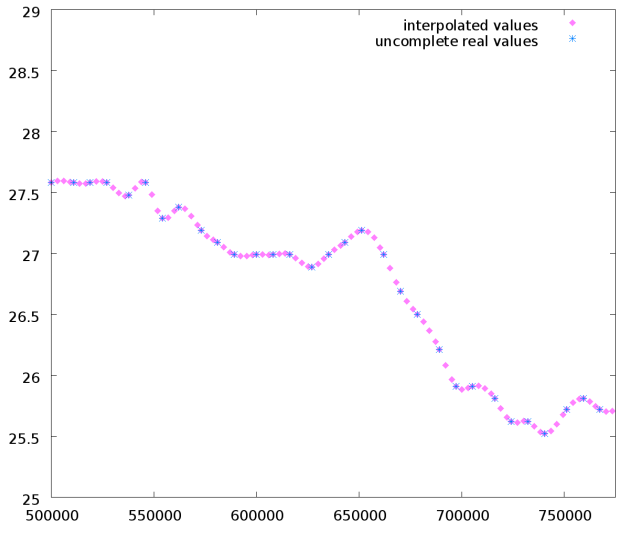


Figure 8: Uncomplete information completed with cubic spline interpolation

The source code shown figure 1 shows how simple it is to implement the application to collect an interpolate data from a WSN with our framework. This is the application that has produced the results shown figure 8. Its kernel is composed of lines 22-32. The application gets successive samples and when too many values are missing (line 27) it asks the interpolation module for a number of replacement values (line 29).

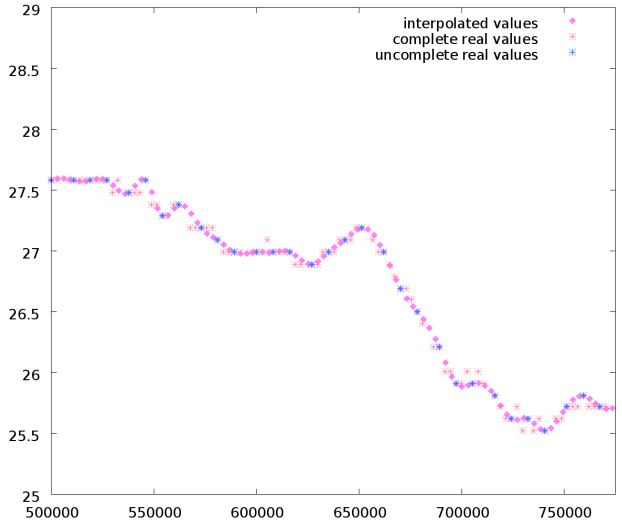


Figure 9: Comparison of real and interpolated values

Figure 9 makes it possible to evaluate the quality of our interpolation module (cubic spline) for temperature measurements. It shows that the real and interpolated curves are very similar.

Listing 1: An example of use of our framework

```
1  public class Extrapolate{
2    //Set the maximum time between 2 measures
3    public static double STEP = 1000;
4
5    public static void main(String[] args){
6      DataServerFile dataServerFile = new DataServerFile(args[1]);
7      OrderedBuffer orderedBuffer = new OrderedBuffer(dataServerFile);
8      Interpolation interpolation = new Interpolation(args[0],
9      Interpolation.CUBIC_SPLINE, orderedBuffer);
10
11     orderedBuffer.addValueListener(interpolation);
12
13     Thread thread1 = new Thread(dataServerFile);
14     thread1.start(); //Starting Data Collection
15     thread1.join(); // Waiting for the end of the measurement
16
17     ArrayList vect = interpolation.getValues();
18
19     PrintWriter pw = new PrintWriter
20     (new BufferedWriter(new FileWriter(args[2])));
21
22     for(int i = 0; i < vect.size()-1 ; i++){
23       Value v1 = (Value) vect.get(i);
24       pw.println(v1.getX()+"\t"+v1.getY()+"\t"+v1.isReal());
25       Value v2 = (Value) v.get(i+1);
26
27       while(v2.getX()-v1.getX() > STEP){
28         v1.set(v1.getID(),v1.getLat(),v1.getLon(),v1.getX()+STEP,
29         interpolation.interpolate(v1.getX()+STEP),false);
30         pw.println(v1);
31       }
32     }
33
34
35     Value v1= (Value) vect.get(vect.size()-1);
36     pw.println(v1.getX()+"\t"+v1.getY()+"\t"+v1.isReal());
37
38     pw.close();
39   }
40 }
```

9 CONCLUSION

Wireless Sensor Networks are most likely to gain more and more interest in the near future. This is especially true in the military domain where they can be used to support situation management on the battlefield.

In this paper, we have considered the class of applications where a number of sensors are used to measure some physical phenomenon at a number of locations on the field. By collecting the synchronized and localized information, the goal is to provide the final user with a global view of the phenomenon.

One of the major problems is that it is clear that in limit conditions such as what happens on a battlefield, a number of measurements can be lost (interferences, damaged sensors, etc). Therefore we have designed and developed a framework that makes it possible to provide user applications and thus the final user with a continuous view of the information even though, because of the losses described above, part of the information is missing. This framework relies on the software infrastructure that we have presented in this paper. We have illustrated its use by showing how an application in charge of monitoring the temperature can take advantage of it.

Form a practical point of view, our framework and illustrative application must be improved. As of writing the localization module is not implemented and we thus have to precisely measure where the sensors have been installed. A number of modules that were not useful to illustrate our purpose have not yet been implemented.

Nevertheless, we believe that the work that we achieved will be useful for those willing to experiment distributed data collection in an environment that neither guarantees data integrity nor network reliability, such as delay tolerant networks.

REFERENCES

[1] G. Werner-Allen, J. Johnson, M. Ruiz, J. Lees, , and M. Welsh. Monitoring volcanic eruptions with a wireless sensor network. In *European Workshop on Sensor Networks*, 2005.

[2] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, New York, NY, USA, 2002. ACM Press.

[3] Jyh-How Huang, Saqib Amjad, and Shivakant Mishra. Cenwits: a sensor-based loosely coupled search and rescue system using witnesses. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 180–191, New York, NY, USA, 2005. ACM Press.

[4] Gyula Simon, Miklós Maróti, Ákos Lédeczi, György Balogh, Branislav Kusy, András Nádas, Gábor Pap, János Sallai, and Ken Frampton. Sensor network-based countersniper system. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 1–12, New York, NY, USA, 2004. ACM Press.

[5] Jason L. Hill and David E. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, 2002.

[6] S. Madden P. Levis, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. *Ambient Intelligence*, chapter TinyOS: An Operating System for Sensor Networks, pages 115–148. Springer, 2005.

[7] Miklós Maróti, Branislav Kusy, Gyula Simon, and Ákos Lédeczi. The flooding time synchronization protocol. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 39–49, New York, NY, USA, 2004. ACM Press.

[8] Michael Thomas Flanagan. Michael Thomas Flanagan's Java Scientific Library. http://www.ee.ucl.ac.uk/ mflanaga/java/. last seen 07/2007.