Bluetooth and Sensor Networks: A Reality Check

Martin Leopold
Department of Computer Science, University of Copenhagen
leopold@diku.dk

Mads Bondo Dydensborg
Department of Computer Science, University of Copenhagen
madsdyd@diku.dk

Philippe Bonnet
Department of Computer Science, University of Copenhagen
bonnet@diku.dk

ABSTRACT

The current generation of sensor nodes rely on commodity components. The choice of the radio is particularly important as it impacts not only energy consumption but also software design (e.g., network self-assembly, multihop routing and in-network processing). Bluetooth is one of the most popular commodity radios for wireless devices. As a representative of the frequency hopping spread spectrum radios, it is a natural alternative to broadcast radios in the context of sensor networks. The question is whether Bluetooth can be a viable alternative in practice. In this paper, we report our experience using Bluetooth for the sensor network regime. We describe our tiny Bluetooth stack that allows TinyOS applications to run on Bluetooth-based sensor nodes, we present a multihop network assembly procedure that leverages Bluetooth's device discovery protocol, and we discuss how Bluetooth favorably impacts in-network query processing. Our results show that despite obvious limitations the Bluetooth sensor nodes we studied exhibit interesting properties, such as a good energy per bit sent ratio. This reality check underlies the limitations and some promises of Bluetooth for the sensor network regime.

Categories and Subject Descriptors

C.2.1 [**Computer Systems Organization**]: Computer-Communication Networks—*Wireless communication*; C.2.2 [**Computer Systems Organization**]: Network Protocols; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems

General Terms

Design, Performance, Experimentation

Keywords

Bluetooth, sensor nodes, network self-assembly, mac layer

SenSys'03, November 5–7, 2003, Los Angeles, California, USA.

1. INTRODUCTION

It is now possible to develop software for sensor networks and to conduct experiments using sensor nodes readily available through research groups or commercial companies. These sensor nodes, based on commercial off-the-shelf components, guarantee a good trade-off between cost (of development and production), reliability and performance. One of the key differences between sensor nodes is their radio component: it impacts not only energy consumption but also software design (network self-assembly, multihop routing and in-network processing).

We can distinguish two types of radio components for sensor nodes: those based on *fixed frequency carriers*, i.e., all sensor nodes within communication range compete for a shared channel in order to transmit data, and those based on *spread-spectrum* transmissions such as Bluetooth, i.e., sensor nodes within communication range use separate channels to transmit data. Roughly, the former type of radio favors connectionless data broadcast while the latter favors connection oriented communications. In this paper, we focus on the use of Bluetooth modules as radio components for sensor nodes.

Bluetooth was initially designed as a cable replacement technology. Does it make sense to consider it in the context of sensor networks? Spread spectrum radios are serious candidates for sensor network usage because of their resilience to interferences (notably in the free 2.4 GHz band). The WINS prototypes from UCLA, for instance, relied on this type of radio. The mass production of Bluetooth radios ensures robustness and decreasing costs. Bluetooth modules are thus valid candidates, but how suited are they to the sensor network regime?

- A Bluetooth module embeds both the physical layer and the MAC layer through the three bottom layers of the Bluetooth stack (baseband, link manager and host controller interface). As a consequence there is no need to implement a MAC layer as part of the sensor node software. Is the Bluetooth MAC layer, based on channel reservation through frequency hopping, adapted to the sensor network regime? How much of an overhead is a Bluetooth module embedded on a sensor node?
- The Bluetooth protocol is complex with its six layers and its drastic compliance requirements. Is it possible to define a stripped down version of the Bluetooth software stack adapted to the footprint requirement of a sensor node?
- Bluetooth's multihop capabilities (scatternets) have been announced for years. However these announce-

ments have not been backed up by product releases. How can we establish a multihop network with Bluetooth based sensor nodes?

- A typical assumption in sensor networks is that each sensor node can communicate with its neighbors to collect information used for collaborative signal processing, routing or in-network processing. Because Bluetooth-based sensor nodes have to establish connections before they send or receive data, a Bluetooth-based sensor network can only be operational after a self-assembly phase during which connections are established. What is an appropriate network-assembly algorithm relying on Bluetooth's device discovery mechanism?
- When two devices are connected, one of them is a master and the other a slave. Nodes are arranged in clusters composed of one master and up to seven slaves. Slaves are following the hopping sequence dictated by the master and they are only allowed to transmit data once the master has contacted them. During network assembly, the choice of masters and slaves is not neutral. If node A is sending data to node B, what is the impact of the choice of master and slave? What is the impact of the number of slaves connected to the master?
- Proposals for in-network query processing [11] assume that the underlying radio supports connectionless data broadcast. What is the impact of Bluetooth on these proposals? In particular, they rely on the introduction of time division multiplexing (TDM) at the application level to synchronize the transmission and processing of data across nodes. Does Bluetooth alleviate the need for application based TDM?

Because we favour a pragmatic approach, we have decided to experiment with actual Bluetooth-based devices in order to study these questions. We chose the BTnodes developed at ETH Zurich [10]. The BTnodes rely on an Atmel microcontroller similar to the one used in the Berkeley motes [4]. Because no Bluetooth module currently supports scatternets, we equipped the BTnodes with two radios in order to enable multihop networking. Using two radios, it is possible to combine clusters of Bluetooth nodes into a multihop topology. This decision was inspired by the dual-radio node design from Sensoria [16]. The BTnodes are detailled in Section 2.

In this paper, we report our experience with Bluetooth-based sensor nodes. Specifically, we make the following contributions:

1. We designed and implemented a tiny Bluetooth stack for TinyOS. We decided to use TinyOS [9] and port it to the BTnodes in order (a) to benefit from its programming model for the design and implementation of our stripped down Bluetooth stack and (b) to benefit from the library of existing components. We measured the code footprint as well as the throughput and the energy consumption on the BTnodes running our Tiny Bluetooth stack.
2. We developed a network assembly procedure that leverage Bluetooth's device discovery protocol. Our procedure is inspired by BlueTree [20] and is adapted to the configuration of the BTnodes with two radios. We measured the latency and energy consumption of our procedure.
3. We adapted the in-network query processing approach of TinyDB for a Bluetooth-based sensor network. We focused on the TDM scheme managed by TinyDB to drive query processing on individual nodes.

Our results suggest that Bluetooth based sensor networks could be appropriate for a niche of applications, such as mounted operations in urban terrain, that necessitate heavy data exchanges during a few criticial periods within a timeframe of up to a week.

2. BTNODES

The BTnodes were developed by ETH Zurich in the context of the Smart-Its project [15]. They are based on the Atmel ATmega128L microcontroller - an 8 bit microcontroller (MCU) clocked at 7.4 MHz, with 4 KiB[1] on chip memory and an external memory chip of up to 64 KiB. The MCU has digital and analog I/O ports that can be used to connect external sensor devices through Molex plugs on the edge of the board. The nodes are equipped with a Bluetooth module (Ericsson ROK 101 007) together with an onboard antenna. Two UARTs connect the MCU with the embedded Bluetooth chip and one of the Molex plugs. Four leds can be used for debugging purposes. The board also contains a voltage regulator: the BTnode can be plugged to power supplies ranging from 3.3 V to 12 V.

MCU	Atmel ATmega128L at 7.372 MHz
Memory	Built in: 128 KiB Flash, 4 KiB SRAM 4 KiB EEPROM External: 64 KiB RAM
I/O	8 Channel 10-bit A/D-converter 2 programmable serial UARTS
Embedded Radio	Ericsson ROK 101 007
External Radio	BTTester (Ericsson ROK 101 007)

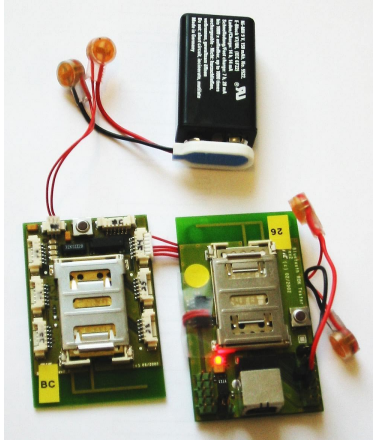
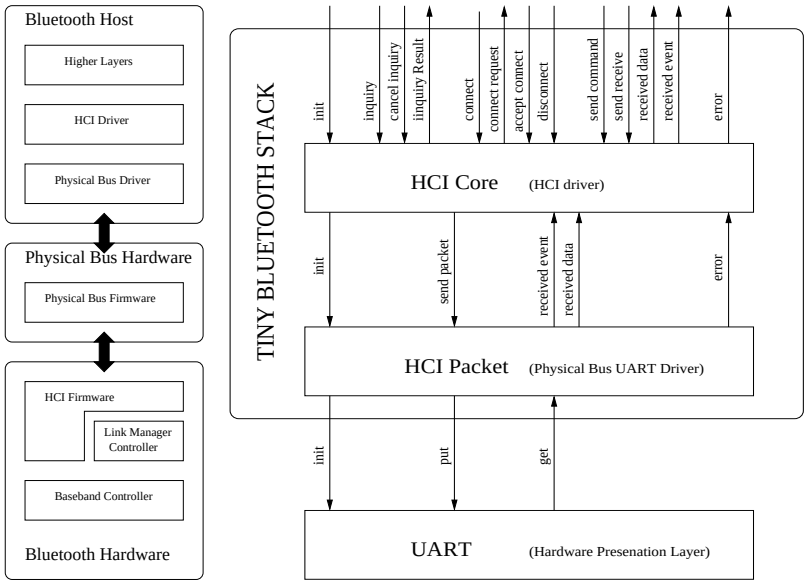


Figure 1: BTnodes characteristics

As explained in the Introduction, we used dual-radio nodes for our experiments in order to assemble a multihop network.

[1]EIC standard 60027-2 defines KiB as 1024 bytes [6]



(a) Bluetooth Stack

(b) Tiny Bluetooth Stack

Figure 2: Design and Implementation of the Tiny Bluetooth Stack

We thus connected an extra Bluetooth module to the BTnode (via the Molex plug connected to an UART). For this purpose, we used the BTtester (also from ETH Zurich), a serial dongle based on the ROK 101 007 Ericsson chip (including an onboard antenna and a monitoring led).

Figure 1 summarizes the characteristics of the BTnodes. Further documentation can be obtained on the BTnode project page [5].

The Bluetooth specification defines two software layers abstracting the higher layers from the hardware characteristics:

1. **Physical Bus Driver**: This layer abstracts the characteristics of the physical bus; on the BTnodes the UART connecting the MCU and the Bluetooth radio.

2. **HCI Driver**: This layer maps the interface of the underlying HCI layer (the upper layer embedded on the Bluetooth radio) into the programming model used for implementing the higher layers.

3. TINY BLUETOOTH STACK

In this section, we report on the design and implementation of our Tiny Bluetooth stack. We compare it to the ETH Bluetooth stack for the BTnodes and to existing communication stacks developed for TinyOS.

3.1 Design

Our goal with the Tiny Bluetooth stack was to make it possible for TinyOS programs[2] to access a Bluetooth radio. We are not interested in implementing a full-fledged Bluetooth stack: while the three lower timing sensitive layers are embedded within the radio modules and thus a given, the higher computation intensive layers (l2cap, sdp, profiles) are essentially dealing with the connection of heterogeneous devices (e.g., a mobile phone and a headset) without providing solutions to problems such as cross layer optimizations or multihop routing, which characterize the sensor network regime. The ability to connect to heterogenous devices is not of paramount interest in sensor networks, which are composed of sensor nodes tailored to operate together as one unit. We thus decided not to implement those three higher layers. Rather, we focus the Tiny Bluetooth stack on the interface to the Bluetooth hardware and its mapping to TinyOS. Network assembly and multihop routing are implemented on top of the Tiny Bluetooth stack.

The Bluetooth specification distinguishes the *Bluetooth host*, on which the three upper layers of the protocol are implemented from the *Bluetooth hardware* on which the three lower layers are implemented. The host and the hardware are connected via a physical bus (e.g., USB, UART, RS232). Figure 2(a) illustrates the connection of the Bluetooth host and the Bluetooth hardware.

Our design is based on these hardware abstraction layers. The Tiny Bluetooth stack is composed of two components: *HCI Packet* that corresponds to the physical bus driver and *HCI Core* that corresponds to the HCI driver. The interfaces of both components are described in Figure 2(b) (we omit the events associated to each TinyOS command according to the asynchronous programming model).

TinyOS applications access the HCI Core component to communicate via Bluetooth. The HCI Core component supports the main HCI commands related to (i) the initialization of the Bluetooth radio, (ii) the inquiries used for device

[2]We could reuse most existing components of TinyOS except of course those abstracting hardware elements differentiating the BTnodes from the Mica and Rene motes, e.g., the uarts, the leds, and the clock.

discovery, (iii) the connection establishment, (iv) the transmission of data and (v) the reporting of errors. It relies on the services of the HCI Packet layer that maps send commands and receive events onto the UART component (a native TinyOS component slightly modified for the BTnodes). We detail below the implementation of these components.

3.2 Implementation

The HCI Core component is responsible for making the HCI interface available in the TinyOS programming model. This task is facilitated by the fact that both HCI and TinyOS are based on an asynchronous programming model. TinyOS commands are naturally mapped onto HCI commands. HCI events need to be dispatched into TinyOS events.

Figure 3 illustrates how events are handled: The UART component generates an event for each byte it receives. This event is handled by HCI Packet in the UART interrupt context. Once a packet is constructed with bytes from the UART, the HCI Packet event handler posts a task that in turn generates an event (this way this event will not be processed in the UART interrupt context). Once the HCI Core component receives this event it posts a task whose goal is to generate the appropriate TinyOS event given the nature of the received packet.

The key aspect for the implementation of the Tiny Bluetooth stack, as for any TinyOS program, is memory management. It must be simple in order to minimize code footprint (i.e., we should avoid a complex allocation/deallocation scheme) and it must optimize data footprint. As a consequence we rely on statically allocated buffers reused across components. We rely on the following two principles for the management of these buffers [8]:

- *transfer of ownership*: one component hands ownership of a buffer to another component.
- *buffer trading*: components switch buffers – a component passes the ownership of a buffer to another component and obtain the ownership of another in return.

In order to be able to take advantage of the *buffer trading* style of memory management, we design a buffer structure that is used throughout the Bluetooth subsystem. This buffer structure, called **gen_packet**, is inspired by Linux socket buffers [17]. It defines a chunk of data as well as a start and an end pointer that allow for easy header removal/insertion. A sending application moves the start pointer to use the trailing part of a buffer when filling in the data chunk. A receiving application moves both the start and the end pointers when a header is added or removed. In this way no layer needs to copy any part of the package before handing it off to an other module. Of course this has the disadvantage that the buffer is required to be *big enough* to store HCI Packets[3]

When exchanged across components a **gen_packet** is typically typecast into a specific buffer type (for a given HCI command or event). This typecasting allows passing buffers between modules without copying from general to specific structures. Figure 4 show the generic packet as well as one specific packet type.

[3]The maximum size for the data payload of an HCI packets is 668 bytes. Depending on the nature of the data sent by the application it might be possible to choose buffers of smaller size in order to minimize memory footprint. The chosen buffer size is communicated to the Bluetooth module that segments all packets that would not fit in the buffer.

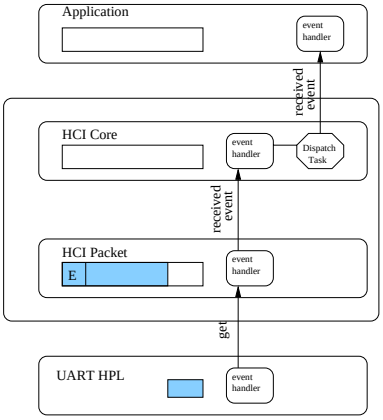


Figure 3: Event Processing in Tiny Bluetooth

In our example, on Figure 3, the UART component receives bytes that are simply passed by value to the HCI Packet component. Bytes are accumulated into a **gen_packet** buffer. Those buffers are traded between the application, the HCI Core and HCI Packet components. Once an HCI Packet is done filling up a packet, it makes it accessible to HCI Core and obtains an empty buffer in exchange. It can thus start filling it with the bytes it obtains from the UART as the previous buffer is being processed by HCI Core. Similarly, once HCI Core is done processing an event it makes it accessible to the application and obtains an empty buffer in exchange so that the application does not have to copy a buffer to process its contents. Similar mechanisms are used when processing commands.

3.3 Related Work

Oliver Kasten [10] implemented a Bluetooth stack for the BTnodes as part of the Smart-its software package. One of their priorities was to make the BTnodes accessible from any Bluetooth device. As a result, they implemented part of the higher Bluetooth layers (link control, connection management and profile). Our emphasis was different: we focused on an efficient access to the Bluetooth radio for TinyOS applications deployed on BTnodes. Minimal support for heterogeneity could be added by implementing support for connectionless l2cap packets in a component on top of the Tiny Bluetooth stack.

The native TinyOS stack from UC Berkeley [18] as well as S-MAC from ISI [19] both rely on a broadcast radio component that support bit-level communication. As a result, their stack (i) implement a CSMA mechanism for media access control and (ii) implement byte as well as packet abstractions on top of the radio itself. In comparison, our Tiny Bluetooth stack is thin: it basically encapsulates the three lower layers of the Bluetooth stack embedded in the radio module. We compare the footprint of these stacks as well as their performance in Section 6.

4. MULTIHOP NETWORK ASSEMBLY

Because Bluetooth is connection oriented, the BTnodes need to be assembled into a multihop network before any

```
typedef struct {
    uint8_t *end;
    uint8_t *start;
    uint8_t data[HCIPACKET_BUF_SIZE];
} __attribute__ ((packed)) gen_packet;
```

```
typedef struct {
    uint8_t *end;
    sniff_mode_cp *start;
              uint8_t data[HCIPACKET_BUFSIZE-sizeof(sniff_mode_cp)];
    sniff_mode_cp cp;
} __attribute__ ((packed)) sniff_mode_pkt;
```

Figure 4: Generic Packet Type (gen_packet) and Example of a Specific Packet Type. The generic packet type defines a buffer together with start and end pointers. The specific packet type, sniff_mode_packet, is used for sending the HCI command that turns a connection into the power saving sniff mode. This specific packet type defines a buffer of the same size as the generic packet but more structured (the buffer is composed of an array of integers – meant to store a packet header – to which is concatenated a variable of type sniff_mode_cp – that contains the HCI command definition and its arguments). The start pointer is of type sniff_mode_cp so that a header can easily be added.

form of multihop routing can be established (e.g., a routing tree for TinyDB as discussed in the next section).

4.1 Design

We must take into account the following characteristics of our platform when designing a multihop network assembly procedure:

1. *Bluetooth Connection Establishment.* Bluetooth connections are established between a master and a slave. The assembly procedure must establish the role of each node with respect to a connection. Note that nodes cannot exchange information before they have established a connection. In addition, slaves cannot communicate with other slaves or overhear the communication taking place on other connections. As a result, we cannot use protocols involving spontaneous communication among neighbor nodes.

2. *Dual Radio Approach.* There are three possible configurations for each dual-radio node: (i) a node can be connected as slave on its two radios, (ii) a node can be connected as slave on one radio and as master with up to seven connections on the other radio, or (iii) a node can be connected as master with up to seven connections on both its radios.

3. *Device Discovery Protocol.* In order for two devices to discover each other, they must be in two complementary states at the same time: *Inquiry* and *inquiry scan.* The inquiring device continuously sends out *is anybody out there* messages hoping that these messages (known as ID packets) will collide with a device performing an inquiry scan. To conserve power a device wanting to be discovered usually enters inquiry scan periodically and only for a short time known as the inquiry window. During this period, the device listens for inquiry messages.

The main challenges for the assembly procedure are thus (a) to pick up pairs of nodes that should be connected and (b) to decide the attribution of slave and master for each connection.

A first approach would be for the nodes to discover their physical location (e.g., each node discovering its neighbors), and to exchange this information with each other in order to reach a decision concerning their configuration. Such an approach would allow to construct robust networks with multiple paths between nodes. Bluetooth however offers limited support for such solutions. The device discovery protocol can be used to discover neighbors, however connections need to be established between pairs of nodes to distribute the discovered information.

A second approach consists in configuring each node *a priori*. Each node is configured with a radio operating as a master and the other operating as a slave. This obliterates the need for the discovery and information exchange phases from the first approach. The second approach constitutes a baseline. We chose to implement it on top of our Tiny Bluetooth stack.

Our baseline solution, inspired by BlueTree [20] (discussed below), is the following. When a node boots up, it enables one of its radios (the *slave* radio) and starts looking for another node to connect to. In this stage, the node will not be discoverable/visible for other nodes; it considers itself an orphan looking for a network. If it discovers other nodes, it tries to connect to one of them as a slave[4]. If the connection succeeds, it will consider itself member of the network, and turn on its other radio (the *master* radio), making it discoverable and ready to accept connections from nodes that are not currently members of the network.

If the connection as a slave fails, it is because the master has reached its limit on the number of connection it can accept (recall that a master can connect to seven slaves). The node then tries to connect to one of the other nodes it has found in its vicinity. If there is no such other nodes ready to accept a connection then the node tries to connect again to the first node it contacted. If a master connected to seven slaves receives three repeated connection requests from the same node N, then it disconnects one of its slaves and accepts the connection from the node N. It has been shown that when a master is connected to more than five slaves, additional slaves are in connection range with at least one of the connected slaves [20]. As a consequence, it is probable that the disconnected node will find a node that it can connect to in its vicinity.

When a node is disconnected from its parent (on its slave radio), it does not try to find a new parent node to which it could connect (because it is probable that it will try connecting to its own children). Instead, it disconnects in turn all the connections on its master radio. As a consequence, when a node is disconnected (due to a failure or because a master has more than seven slaves), all nodes directly or indirectly connected to this node will end up being disconnected. They start again as orphan nodes; the assembly procedure is restarted, and a connected multihop network is reconstructed (if at all possible). This is again part of our

[4]Note that when a node discovers another node it is by default a master; connecting as a slave requires a role switch.

baseline approach, smarter solutions for the reconnection of an orphan node would imply exchanging information across nodes in order to reach an appropriate decision.

In order to bootstrap the assemply procedure, one of the nodes, say a gateway, starts-up with its *master* radio turned on, thus being discoverable for other nodes. Its *slave* remains disabled, or is used to connect to the *external world*. The network topology obtained with this assembly procedure is a tree, that we call a **connection tree**, rooted at the gateway node that starts-up the assembly procedure. Note that if we assume that all nodes start-up with their slave radio enabled, then the connection tree will be bushy: once a node activates its master radio, all nodes in its vicinity will have a chance to connect to it before they get a chance to connect to its children.

It is desirable that a sensor network be accessible via several gateways. One has a privileged role as the root of the connection tree. Other gateways are connected as leaves (they do not activate their master radio once they are connected to the network via their slave radio). If the root of the connection tree fails, then all the nodes disconnect and the assembly procedure can restart with another gateway taking the responsability of being root of the connection tree.

4.2 Related Work

Sohrabi et al. [16] describe two assembly procedures for the Sensoria dual-radio nodes (equipped with proprietary frequency hopping radios). The first procedure proceeds as follows: the two radios of a node are tuned to two fixed channels. During network assembly, messages are exchanged on those channels to form clusters (in each cluster a node is elected as a master while the other nodes are slaves). The constraint that a node cannot be master on its two radios ensures that the clusters constructed for the two radios are different. Such overlapping clusters cover the entire network with a high probability; but there is no control over the topology of the connection tree. The second procedure relies on a discovery phase during which one of the radios is tuned to a control channel in order to broadcast information. Once a node has received information about its neighbors it takes a decision on its own role: a master remains tuned to the control channel and exchange data to its slaves on its second radio while a slave exchanges data on both its radios. Both procedures rely on tuning one or both radios onto a fixed channel so that nodes can broadcast information to their neighbors. This is not possible with Bluetooth.

Basagni et al. [20, 2, 13] proposed a set of scatternet formation protocols for Bluetooth devices: BlueTree, BlueStar, BlueMesh. The basic protocol is BlueTree that constructs a multihop network with a tree topology. BlueTree proceeds in two phases. First, every node obtains information from its neighbors (nodes spend *enough time* inquiring and responding to discover their neighbors). Second, a designated node, the *blueroot*, initiates the construction of the BlueTree. This node is assigned the role of a master and its neighbors becomes its slaves. Recursively, each node that has become a slave is assigned the role of master with respect to its unconnected neighbors. The blueroot is thus the root of the connection tree, all the intermediate nodes in the connection tree are both slaves and masters, while the leaves are just slaves. The constraint that one master cannot be connected to more than seven slaves is handled via tree reorganization. This work rely on the assumption that Bluetooth supports scatternets (intermediate nodes are master and slaves on a single radio). Our assembly procedure is an adaption of the BlueTree protocol for the dual-radio configuration. The main difference is that our approach does not include a discovery phase, rather nodes can join the network at any time (an unconnected node has its slave radio turned on and is looking for a master).

5. IN-NETWORK QUERY PROCESSING

TinyDB developed at UC Berkeley and Intel Research [11, 12], processes declarative queries over sensor data within the sensor network. The proposed query language allows user to collect, filter and aggregate data produced in the sensor network. In-network query processing proceeds in two phases. First, the query is disseminated from a gateway node to all appropriate sensor nodes inside the network. This first phase results in the establishment of a routing tree rooted at the gateway and spanning all the sensor nodes producing data for the given query. Second, data is processed when transmitted up the routing tree.

Each sensor node runs an instance of TinyDB, which is responsible for (i) establishing and maintainting the routing tree, (ii) processing query fragments over data received via the network or read on a local sensor and (iii) transmitting processed data up the routing tree. Aggregates are processed in a distributed manner: each node computes a partial state record based on the values it reads and obtains from its children. The partial state records are transmitted and the actual aggregate value is obtained at the root of the routing tree.

TinyDB is a TinyOS library. It relies on basic TinyOS building blocks essentially for sending and receiving data over the network, for setting the power saving mode, for setting timer events and for reading sensor data.

TinyDB has been designed for the Mica motes [4], assuming communication based on connection less broadcast on a shared channel radio. The choice of a Bluetooth radio challenges some of the assumptions that underly its design and implementation. In the rest of this section, we discuss the key issues when designing a version of TinyDB based on our Tiny Bluetooth stack that we call BlueTinyDB for convenience.

5.1 Topology Management

For each query, TinyDB constructs a routing tree rooted at the gateway on which the query is submitted. The routing tree is constructed by having nodes pick one of their neighbours as their parent.

This procedure for routing tree construction is very much similar to the one we described in the previous section for network self-assembly. The construction of the BlueTinyDB routing tree consists in constructing a directed version of the connection tree, with the edges oriented towards the gateway on which the query is submitted. The oriented edges are maintained as a parent-child relationship (independent from the master-slave relationship maintained by the self-assembly component).

BlueTinyDB constructs the parent-child relationship while disseminating a query. Each node retrieves the list of neighbors from the self-assembly component. The node from which the query is received becomes the parent, the other neighbors become children, regardless of whether they are master or slaves. There are two issues to be considered:

- There might be several gateways connected to a cluster of nodes and there might be several queries submitted to different gateways at the same time. Those queries will share the same connections. It is thus desirable that the connection tree is as bushy as possible, in order to minimize the height of the routing trees, regardless of the gateway to which a query is submitted.

- Madden et al. [12] have proposed Semantic Routing Trees to restrict the distribution of a query to those nodes who actually participate in a given query. The idea is to maintain meta-data on each node to decide whether the nodes accessed through a given child will participate in the answer to the query (in which case the query is distributed further to this child) or not (in which case the query is not distributed to this child). The fact that all routing trees share the same connection tree facilitates the collection and the maintenance of this meta-data.

5.2 TDM at the Radio Level

TinyDB organizes a time division multiplexing (TDM) scheme at the application level in order to organize the processing and the transfer of data along the routing tree. The design of the TDM scheme described by Madden et al. [11] is driven by the following observations:

1. The output of a TinyDB query is a stream of values. To each value is associated a time-stamp. Aggregates are performed on values whose time-stamp belong to a same time interval, or *epoch*. Because the processing of an aggregate is distributed across a set of nodes, it should be synchronized across nodes in order to ensure that data is indeed processed within an epoch.

2. It is desirable to power down a node while it is not processing or transmitting data.

TinyDB's TDM scheme is implementated at the application level as follows. First, the query is disseminated together with timing information that allow nodes (i) to synchronize clocks and (ii) to set the timer that will drive the TDM scheme. Then each node is put to sleep (i.e., the microcontroller is forced into power saving mode using the TinyOS Snooze command) until it is woken up by a timer event. When a node is asleep, it cannot process sensor or network data. Once a node is awake it receives data from its children, reads sensor data, processes the query locally and transmit the partial state record to its parent. The timers are set on each node so that there is an overlap between the intervals children are sending and parents are listening.

The main problem with this approach is the timer setup: How to assign time slots in the TDM scheme at the application level? The duration of each slot depends on the duration of an epoch (defined in the query), the height of the tree (data produced during an epoch should be processed together), the breadth of the tree (each node must receive data from all its children before it transmits processed data up the routing tree) and the density of nodes (there can be collisions between nodes within transmission range whether they are siblings in the routing tree or not). The TDM scheme also relies on clock synchronization across nodes, which is only approximate at the application level[5]. In addition, modifications of the routing tree should lead to a modification of the slot duration on each node. In practice, the slot duration will be a rough estimate (e.g., the duration of an epoch divided by the size of the tree) and the control of the TDM scheme will be at best an approximate leading to the loss of values that cannot be transmitted and processed during a given epoch.

Using Bluetooth, it is possible to let the radio drive the TDM scheme. Bluetooth relies on a TDM scheme at the physical and MAC layers to implement the frequency hopping transmission. It also provides a power saving mode that allows reducing the duty cycles between connected devices: the *sniff* mode. In sniff mode, two devices negociate specific slots where communication can begin. If no communication takes place at these sniff slots, the devices may spend the time until the next sniff slot in low power mode. Otherwise, communication takes place until one of the devices decides to end the communication. A message sent from the host to the Bluetooth module while the radio is in low power mode will first be transmitted when a sniff slot is reached. A master can enter the sniff mode for each individual connection it manages. The period of the sniff slots is a parameter given by an application when a connection enters the sniff mode. The application has however no control over the timing of the sniff slots.

BlueTinyDB relies on the sniff mode as follows. On each node, all connections (to all children and to the parent) enter sniff mode with a period equal to the epoch specified in the query and the microcontroller is put in sleep mode. Whenever the node receives data from a child, it updates the partial state record for the given query. Once all data with a same time-stamp have been received[6], the rest of the query is evaluated (the partial state record is consolidated, filter predicates are applied) and the processed data is sent up the routing tree. The processed data is actually sent only when the sniff slot for the parent connection is encountered. Note that BlueTinyDB does not explicitely set a timer to control the TDM scheme; it is the Bluetooth radio that manages timing through the sniff mode.

The use of the sniff mode to drive the TDM scheme in BlueTinyDB raises the following issues:

- Because BlueTinyDB has no control over the timing of the sniff slot, it cannot guarantee that data is transmitted and processed from the leaves of the routing tree up to the root within a single epoch. Processing is thus pipelined across epochs. We thus relaxes the initial constraint to process aggregates within an epoch. This approach was suggested by Madden et al. as an alternative to their TDM scheme [14].

- The microcontroller sleeps until it is woken up by incoming data received over the network (or possibly by data produced by a local sensor). Depending on the number of children, the microcontroller switches from operational to power saving mode several times per epoch. The duration of the intervals during which the microcontroller is in power saving mode depend on the timing of the sniff slots.

[5]A child node uses the timing information contained in the message from its parent (how long until the end of the epoch) to synchronize its clock (+/− 1 ms).

[6]If a child fails to send data, it is simply ignored.

5.3 Separated Channels

TinyDB relies on the TinyOS MAC layer to avoid collisions between nodes transmitting during the same interval. Those collisions occur not only between parents and children but between any two nodes who are in transmitting range of each other. If the density of nodes is high, then we can expect a high rate of collisions. As a consequence, TinyDB monitors channel contention and adaptively reduces the number of packets sent as contention rises.

The problem of channel contention is not acute in the context of a Bluetooth radio. Each connection constitutes a separated channel. There is only interference between pairs of nodes hopping on the same frequency at a given point in time. This is a marginal problem.

Note that separated channels are expected to boost the performance of flooding as collisions are avoided between parents and children as well as between branches of the routing tree.

Note however that separated channels do not permit the snooping used in TinyDB to optimize the performance of certain aggregate operators (such as MAX) or to compensate for the loss of some messages.

6. EXPERIMENTS

6.1 Calibration of the BTnodes

The Tiny Bluetooth stack allows us to run experiments with the BTnodes in order to calibrate them with respect to throughput as well as energy consumption.

6.1.1 Code Footprint

Our first challenge was to squeeze a tiny version of the Bluetooth stack within the BTnode. The code footprint for our Tiny Bluetooth stack is less than 3 KiB. It is thus comparable to the native TinyOS stack (approximately 2 KiB [9]) and one order of magnitude less than the Smart-its Bluetooth stack (approximately 30 KiB [15]).

In the dual radio configuration, we have duplicated the Tiny Bluetooth stack for the sake of simplicity. The code footprint of these stacks together with network assembly and multihop routing is approximately 8 KiB. We estimate that we can reduce the code footprint by 30 to 40% if we avoid duplicating the stack. The data footprint (taking into account the non initialized variables—traditionnally denoted as bss) is 1 KiB due to the packets statically allocated in the Tiny Bluetooth stack.

Table 1 breaks down the code, bss (non initialized variables) and data (initialized variables)[7] footprint for the dual-radio configuration.

6.1.2 Throughput

Bluetooth specification promise high throughput. Can our Tiny Bluetooth stack take advantage of this potential on the BTnodes?

We measured throughput on point-to-point connection between master and slave. Figure 5 shows the results we obtained for all possible Bluetooth encodings and two payload sizes.

[7]The distinction between initialized and non initialized variables is interesting as non initialized variables are not part of the code uploaded to the nodes.

Description	code	bss	data
Support & TinyOS core	1180	0	
UART0 & interrupts	346	4	
UART1 & interrupts	292	5	
hciPacket0	604	155	
hciPacket1	588	155	
hciCore0	1624	159	
hciCore1	1590	159	
Assembly component	4796	1021	16
Total	11020	1658	16

Table 1: Code size breakdown (bytes)

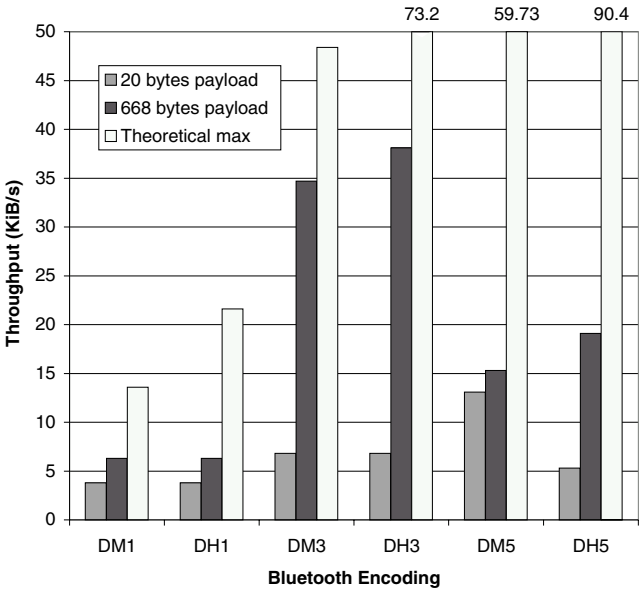


Figure 5: Throughput Master to Slave

Bluetooth defines 6 encoding that correspond to the combination of two levels of resistance to noise (due to different levels of redundancy in the encoding – DM: high resistance and DH: low resistance) and three configurations of the TDM scheme (the node transmits for 1, 3 or 5 time slots). The two payload sizes we consider are 20 bytes, i.e., enough to transport a few integers which is the case for many sensor network applications, and 668 bytes which is the maximum HCI packet size, i.e., enough to transport high bandwidth traffic such as images or sounds.

For payloads of 20 bytes, the encoding used does not make a significant difference. We achieve a throughput of about 5 KiB/sec. For the maximum payload of 668 bytes, throughput is significantly higher: it varies from 6 to 35 KiB/sec. Here, the number of slots used for transmission and the resistance to noise have a significant impact. Our result shows that the best throughput is achieved with 3 slots transmissions (more data is transmitted than with 1 slot transmission and there is less retransmissions than with 5 slots).

Note that the throughput we achieve is far from the theoretical max. First, our Tiny Bluetooth stack accesses the Bluetooth module via the UART and the maximum through-

put supported by the UART is approximately 45 KiB/sec[8]. Second, the Bluetooth module generates superfluous messages that take up bandwidth on the UART interface (hard-coded Ericsson string events).

	1	2	3
Aggregate	38.1 KiB/s	25.4 KiB/s	19.3 KiB/s
Pr. slave	38.1 KiB/s	12.7 KiB/s	6.4 KiB/s

Table 2: Throughput for an multipoint asymmetric DM3 connection, aggregate and individual bandwidth

Additional experiments have shown that slave to master communication achieves a throughput which is very similar to the master to slave communication shown in Figure 5. This means that data can flow up the routing tree regardless of whether individual connections are master-slave or slave-master.

The master is responsible for allocating the channel bandwidth for each slave in a multipoint connection. We setup the master to connect to a number of slaves and send packets in round-robin order on each connection. We measured the bandwidth on each connection. It would be expected that the total bandwidth stays the same and that each slave receives a fair share.

Table 2 shows that the aggregate bandwidth drops considerably but that each slave receives a fair share of that bandwidth. It is peculiar that the aggregate bandwidth drops as much as 50 %—the lower layers of Bluetooth allow the master to switch between slaves without any additional overhead. Hence the ROK modules wastes in-air slots by not sending meaningful data.

Madden et al. [12] write that it is reasonable to expect that the Mica motes transmit approximately 500 bytes per second. We thus achieve with the BTnodes a throughput, which is between one and two orders of magnitude higher for point-to-point connections and at the very least a factor of two for multipoint connections.

6.1.3 Energy Consumption

The throughput results are encouraging; however energy consumption is the key metric for the calibration of the BTnodes. We thus measured both current draw and voltage[9] for different regimes of our experimentation platform. Figure 6 summarizes our results.

Our first goal was to measure energy consumption for an idle BTnode (in black on the figure). According to the manufacturer [1], the 8 MHz MCU can use up to 12 mA at 5 V (60 mW) in idle sleep mode. With a slightly lower clock frequency, we observe a lower energy consumption (about

[8]Note that we had to modify TinyOS to operate the UART at full speed. The UART relies on two registers for sending bytes: one, called the shift register, contains the byte currently being sent, the other contains the byte to send next. We fixed the UART module so that it generates an event each time the UART moves a byte to the shift register instead of generating an event each time the shift register is empty and the UART has no more data to send.

[9]We used an input voltage of approximately 5V. We observed that voltage varied slightly during the experiments. We thus decided not to assume constant voltage to compute the energy consumption.

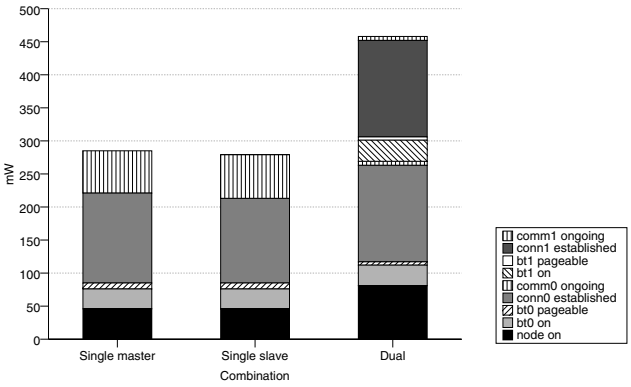


Figure 6: Energy usage breakdown. The graph presents a master and a slave equipped with one radio (bt0) as well as a dual-radio node (with bt0 and bt1). In the legend, comm0 and comm1 denote connections established on bt0 and bt1 respectively.

46 mW). In the dual case, the base energy consumption is 80 mW. This is mostly due to the (unused but turned on) led on the BTtester, which consumes about 22 mW, as well as the voltage regulator and the serial voltage level converter.

Turning a Bluetooth radio on consumes about 30 mW extra (in idle mode). Making the radio pageable, or inquiriable requires an additional 9 mW. A BTnode with a single radio waiting for a connection thus consumes 89 mW. In the dual radio case, turning on both radios and making them pageable and inquirable consumes about 155 mW.

Nodes use about 136 mW to maintain connections. Additional experiments showed that putting a connection in sniff mode saves a marginal 5 mW. This suggests that some optimizations might have been missed in the Bluetooth module design.

Once a connection is established sending or receiving data consumes an additional 65 mW when transferring at 6 KiB/-sec when using a single radio and 5 mW for each radio when transferring 10 packets per second.

6.1.4 Discussion

All in all, a single-radio BTnode uses approximately 50 mW when idle and 285 mW when communicating, while these numbers are up to 80 and 450 mW for a dual-radio BTnode. If we compare with the Mica motes from UC Berkeley, these numbers are high. Indeed, Madden et al. [12] report 10 mW for for an idle node and 60 mW when communicating.

The BTnodes consume five times more energy than the Mica motes doing nothing! This is due to the fact that the MCUs are placed in different sleep modes. As we have seen with TinyDB in Section 5.1, the Mica motes favour applications that manage themselves the time they spend in sleep mode. This way, the MCU can be put to sleep in *power save* mode, where only the external clock can send wake-up signals, i.e., the motes do not get data from sensors or from other nodes while the MCU is in sleep mode. The *power save* sleep mode is the most energy efficient.

The BTnodes favour applications that are in sleep mode until events are received from the Bluetooth radio or the sensors. This precludes the use of the *power save* mode because

events generated by the Bluetooth radio would be ignored. The MCU is best put in *idle* sleep mode until an event is received from the UART or from the clock. The *idle* sleep mode is however less energy efficient than the *power save* mode. We can estimate to 8 days the life expectancy of a BTnode in *idle* sleep mode with two standard AA batteries.

We noted that energy consumption was high when transmitting data. However, if we consider the ratio energy per bit transmitted, then the BTnode exhibits rather good performances. Running at 6 KiB/s, we obtain approximately 5.5 μJ/bit for a single radio and 10 μJ/bit for two radios. These numbers are close to the ones reported in the original TinyOS paper [9] (approximately 4 μJ/bit) for an ideal transmission without interference or loss. According to Madden et al. 15 μJ/bit is a more realistic number for the Mica motes.

Another characteristics of these results is that connection maintenance consumes a lot of energy, both on master and slave. Using the sniff mode does not make a significant difference. These results suggest that connections should only be established for short periods. There is thus a trade-off between the cost of connection maintenance and the cost of network assembly. We explore this trade-off in the next Section.

In summary, all these remarks suggest that the BTnodes are well suited for applications that are active over a limited time period, with few unpredictable bursts of very heavy network traffic (taking advantage of the high throughput). An example of such application could be a sensor network deployed to secure a building in a mounted operation in urban terrain. Such a network could have a life expectancy of up to a week, operating in sleep mode until individuals are detected in which case as much situational information as possible could be obtained (including possibly images or sound on a suite of point-to-point connections).

6.2 Network Self-Assembly

Because it is expensive to maintain a connection it is likely that the network will be assembled repeatedly when data needs be transmitted. Network assembly should thus be rapid and energy efficient.

Figure 7 shows energy consumption as a function of time on a BTnode during network assembly. The figure is annotated with letters corresponding to the different phases of the experiment. As the node is turned on (a), it initializes its *slave* radio. After a while, it discovers its parent-to-be, connects to it (b) and enables its *master* radio that becomes discoverable. The connection is established within 20 seconds.

For the sake of clarity, we turn on the children nodes one after the other. After 30 seconds, the first child is turned on, it is detected and a connection is established on the node's *master* radio (c). After 40 second, the second child is turned on and a second connection is established (d). Energy consumption corresponds to the calibration presented in the previous Section. Additional connections result in a very limited increase in energy consumption (about 3-4 mW).

Once those connections are established, the node routes data from both children to its parent: both children send packets with payload of 5 and 50 bytes every 10 seconds for a 100 seconds. Note that the children do not send in synch. The peaks of energy consumption we observe from 40 seconds until 160 seconds correspond primarily to the master

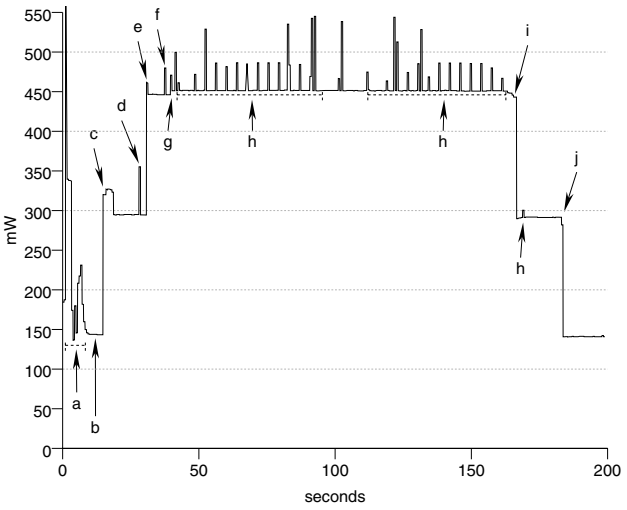


Figure 7: Energy Usage during Network Assembly and Transmissions

radio being discoverable (e) and also to the transmission of data packets (f)[10].

After 160 seconds, both children are disconnected (g) and after 180 seconds the parent is disconnected (h). The node remains idle with both Bluetooth radios disabled but powered on.

Additional experiments show that our assembly procedure requires in most cases between 5 and 10 seconds per level in the connection tree assuming all nodes have already initialized their Bluetooth radios[11]. Indeed, children can only be discovered once the connection to the parent is established.

Energy consumption corresponds to what could be predicted after calibrating the nodes. There is no extra overhead introduced by the assembly procedure. In order to save energy, it thus seems a good idea to assemble the network repeatedly, for relatively short periods during which high volumes of data can be transmitted.

7. CONCLUSION

Our experiments with the BTnodes suggest that Bluetooth based sensor nodes could be appropriate for a niche of applications exchanging unpredictable bursts of data during a limited time period.

It is difficult to predict that Bluetooth will emerge as an alternative of choice for a larger class of sensor network applications. First, as long as scatternets are not supported we will have to rely on a dual-radio approach which is expensive and energy inefficient. Second, using Bluetooth for commercial purpose requires a certification (to guarantee that heterogeneous devices can communicate) which is irrelevant and unrealistic in the context of sensor networks. Third, the encapsulation of the three lower layers on hard-

[10]A similar experiment without data transmission also exhibits energy peaks but with a more regular intensity.

[11]We observe that connections take most of the time 5 to 10 seconds with some outliers requiring 20 to 30 seconds. We have studied the characteristics of device discovery and connection establishment in previous work [3].

ware does not allow applications to access relevant information (concerning time synchronization or device discovery in particular). Fourth, our experiments show that connection maintenance is expensive even with a low duty cycle in power saving (sniff) mode. This renders unpractical the possibility for the radio clock to drive TDM on behalf of the application.

Instead of using a Bluetooth module separate from the MCU, both could be integrated on one die. This system-on-a-chip approach has already been adopted by CSR for their Bluetooth module [7]. The MCU runs both the lower layers of the communication stack and the applications. The arguments mentioned above disqualify Bluetooth as a first choice for sensor nodes based on a system-on-a chip design. In addition, the dual-radio approach is not reasonable on such integrated systems because of interferences. We believe however that a form of spread-spectrum radio is appropriate for system-on-a-chip sensor nodes because the timing sensitive hardware necessary to implement such radios can be leveraged by the application (as we have shown in this paper). It is however necessary to devise an effective solution for the scatternet problem. It is necessary to devise aggressive power saving strategies based on the characteristics of the radio as well as the application. These are topics for future work.

8. ACKNOWLEDGEMENTS

We thank Oliver Kasten and Jan Beutel from ETH Zurich for interesting discussions and for their help with the BTnodes. We also wish to thank Janus B. Lundager for his time and his multi-meter. We appreciate the feedback from the anonymous reviewers and from our shepherd Mark Smith.

9. REFERENCES

[1] Atmel home page. `http://www.atmel.com/`.

[2] S. Basagni and C. Petrioli. A scatternet formation protocol for ad hoc networks of Bluetooth devices. In *Proceedings of the IEEE Semiannual Vehicular Technology Conference, VTC Spring 2002*, Birmingham, AL, May 6–9 2002.

[3] A. Beaufour, M. Leopold, and P. Bonnet. Smart-tag based data dissemination. In *Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications (WSNA-02)*, pages 68–77, New York, Sept. 28 2002. ACM Press.

[4] Hardware design for the berkely motes. `http://webs.cs.berkeley.edu/tos/hardware/hardware.html`.

[5] Btnode project page. `http://www.tik.ethz.ch/~beutel/btnode.html`.

[6] I. E. Commision. Letter symbols to be used in electrical technology - part 2: Telecommunications and electronics. IEC standard 60027-2, 2.nd edition, 2000.

[7] Cambridge silicon radios. `http://www.csr.com/`.

[8] J. Hill. A software architecture supporting networked sensors. Master's thesis, UC Berkeley, 2000.

[9] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-00)*, pages 93–104, 2000.

[10] O. Kasten and M. Langheinrich. First experiences with bluetooth in the smart-its distributed sensor network. In *Workshop on Ubiquitous Computing and Communications, PACT*, 2001.

[11] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A tiny Aggregation service for ad-hoc sensor networks. In *Proceedings of the 5th ACM Symposium on Operating System Design and Implementation (OSDI-02)*, Operating Systems Review, pages 131–146, New York, Dec. 9–11 2002. ACM Press.

[12] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *SIGMOD*, 2003.

[13] C. Petrioli and S. Basagni. Degree-constrained multihop scatternet formation for Bluetooth networks. In *Proceedings of the IEEE Globecom 2002*, Taipei, Taiwan, R.O.C., November 17–21 2002.

[14] M. J. F. Samuel R. Madden, Robert Szewczyk and D. Culler. Supporting aggregate queries over ad-hoc wireless sensor networks. In *Workshop on Mobile Computing and Systems Applications*, 2002.

[15] Smart-its home page. `http://www.smart-its.org`.

[16] K. Sohrabi, W. Merrill, J. Elson, L. Girod, F. Newberg, and W. Kaiser. Scalable self-assembly for ad hoc wireless sensor networks. In *IEEE CAS workshop*, 2002.

[17] The linux kernel documentation `http://tldp.org/lpd/tlk/tlk.html`.

[18] A. Woo and D. Culler. A transmission control scheme for media access in sensor networks. In *Proceedings of the Seventh Annual International Conference on Mobile Computing and Networking (MOBICOM-01)*, pages 221–235, New York, July 16–21 2001. ACM Press.

[19] W. Ye, J. S. Heidemann, and D. Estrin. An energy-efficient MAC protocol for wireless sensor networks. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Society (INFOCOM-02)*, volume 3 of *Proceedings IEEE INFOCOM 2002*, pages 1567–1576, Piscataway, NJ, USA, June 23–27 2002. IEEE Computer Society.

[20] G. Záruba, S. Basagni, and I. Chlamtac. BlueTrees—Scatternet formation to enable Bluetooth-based personal area networks. In *Proceedings of the IEEE International Conference on Communications, ICC 2001*, Helsinki, Finland, June 11–14 2001.