Scalable Topology Control for Deployment-Support Networks

Jan Beutel, Matthias Dyer, Lennart Meier, and Lothar Thiele
Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology (ETH) Zurich
8092 Zurich, Switzerland
Email: {beutel,dyer,meier,thiele}@tik.ee.ethz.ch

Abstract— **Deployment-support networks (DSNs) have been proposed as a novel tool for the development, test, deployment, and validation of wireless sensor networks. They are expected to enhance scalability and flexibility in deployment by eliminating cable connections. In this paper, we describe our implementation of a DSN, giving details on the algorithms for topology control and maintenance. We provide various measurements of DSNs spanning up to 71 BTnode rev3 devices, featuring the largest Bluetooth scatternet reported to date. We also discuss our experience gained in the development and experimentation. Our results strongly suggest that our implementation scales well to a large number of nodes.**

I. Introduction

The recent focus on wireless sensor networks (WSNs) has brought about many different platforms. Researchers are successfully using these platforms in the development of a multitude of sensor-network applications and in the deployment of demonstrators. Setups with more than 10–20 nodes have shown to be hard to manage, especially when situated in a realistic physical environment.

Two key challenges in the deployment of realistic WSNs are the large number of devices and the necessity to embed them in a physical environment. Today, development and debugging is typically done with serial cables connecting individual sensor nodes to a control terminal. These connections are used for stepwise testing, control, monitoring, and (re-)programming of the nodes. This method has been widely adopted for the construction of testbeds.

Existing testbeds either provide only temporary connections to selected nodes, or achieve permanent connections to fixed arrays of nodes only with the use of infrastructure such as Ethernet or serial multiplexers. If larger numbers of nodes are deployed in the field, it is infeasible to connect a separate cable to each node. Reprogramming, which is frequently necessary in the development phase, then requires all nodes to be collected and redistributed. Moreover, the process of calibration, exact monitoring, and validation of sensor networks requires extra functionality that is typically provided through an independent system [1]. Embedding network programming and distributed debugging in the nodes leads to a considerable increase in device-functionality requirements and application complexity. Under stringent resource constraints or in a post-development phase, this is not an option.

Deployment-support networks (DSNs) have been proposed [2] as a non-permanent, wireless cable replacement. This approach allows to deploy and test large numbers of devices in a realistic physical scenario. The DSN is transparent, highly scalable, and can be quickly deployed. It does not disturb the target WSN any more than the traditional, cable-based approach. For the engineer, everything actually looks as if the usual cables were in place; he can thus use the same tools. The DSN nodes are attached to WSN target devices via a programming and debugging cable and form an autonomous network (see Fig. 1). The WSN nodes can then be accessed through serial-port tunnels operated over the DSN. With this tool, the limit for large-scale prototyping is pushed from simulation [3] and virtualization [4] to coordinated real-world deployment.

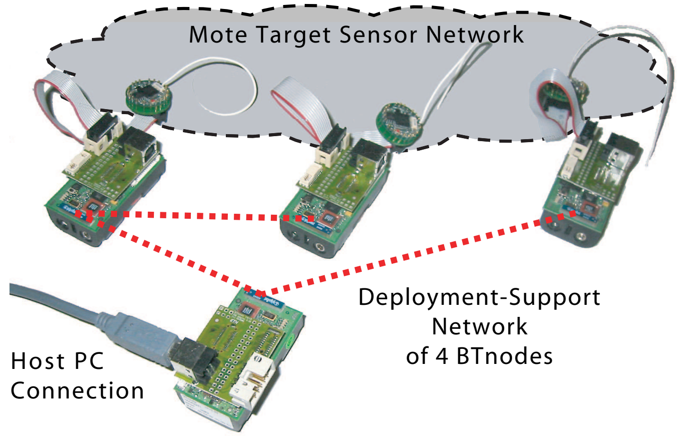

Fig. 1. Each WSN target device (Mica2Dot Mote) is attached to a DSN node (BTnode rev3) using an adapter cable. In this example, 3 Motes can be debugged via a 4-node DSN spanning 3 hops and a host PC attached to the DSN.

In this paper, we describe our implementation of and experiences with a DSN. After first experimental results on the feasibility [5], the DSN application has been ported on the improved BTnode rev3 platform running NutOS, a multi-threaded operating system for embedded devices. In Sect. II, we describe the enhanced connectivity of the BTnode rev3 and explain our topology-control and data-transport algorithms. In Sect. III, we present our experiments and performance measurements with DSNs of up to 71 nodes, featuring the largest Bluetooth scatternet reported to date. We discuss the results in Sect. IV.

II. Implementing a Robust DSN

A. The Enhanced BTnode rev3

The BTnode is a versatile, lightweight, autonomous platform based on a Bluetooth radio and a microcontroller. It has been recently revised, mainly (i) to make use of a new Bluetooth subsystem and (ii) to incorporate a second, low-power radio which is identical to the one used on the Berkeley Motes. This makes the BTnode rev3 a twin of both the Mica2 Mote and the old BTnode. Both of its radios can be operated simultaneously or be independently powered off when not in use, considerably reducing the power consumption of the BTnode.

This new device provides opportunities to create tiered architectures with high-bandwidth nodes bridging ultra-low-power devices like the Berkeley Motes to Bluetooth-enabled gateway appliances [6], or to investigate duty-cycled multi-front-end devices with wake-up radios [7] or bandwidth–power–latency trade-offs.

The BTnode is a most suitable platform for the implementation of a DSN prototype. Its Bluetooth radio perfectly meets the high bandwidth requirements for a cable replacement. Clearly, the energy consumption of a Bluetooth-based DSN node might be higher than that of the target sensor node, especially when the traffic on the Bluetooth connections of the DSN is high. However, this is not critical since the required lifetime of a DSN node is comparatively small. Several techniques can be applied to improve the energy efficiency of the DSN, for instance duty-cycling or using the various power modes of the microcontroller and the Bluetooth connections.

B. BTnode rev3 Multihop Networking

Networking in Bluetooth is organized in master–slave configurations called piconets. Prior to opening a connection to a specific node, the initiating node has to perform an inquiry to detect other nodes in the vicinity. Every piconet has one master to which up to seven active slaves can be connected. Two piconets can be connected by a node taking on dual roles (slave–slave or master–slave); a network of connected piconets is called a scatternet. The Zeevo ZV4002 Bluetooth device on the BTnode rev3 is capable of interconnecting scatternets of up to four piconets simultaneously per node, with active connections to a maximum of seven slave and three master devices. Additionally, connection requests, data transfers, and inquiries can be performed simultaneously. The previous generation of BTnodes imposed several constraints, most notably permitting only one slave role per node and making nodes with a slave role unable to perform or respond to inquiries or to open new connections [5]. With the BTnode rev3, it is not anymore necessary for the operation of our DSN that all nodes be within each other's transmission range.

The Bluetooth standard contains no specifications for the formation and control of multihop topologies or for the data transport across multiple hops. An additional functional layer must therefore provide these services. We chose a modular structure for this layer, i.e. topology control and data transport are independent of each other. In the following two sections, we will describe the two modules in detail.

C. Topology Control and Maintenance

The *connection manager* constructs and maintains a multihop network of DSN nodes. It is a robust and completely local algorithm that automatically takes care of nodes that join or leave the network. The basic principle is simple: Every DSN node periodically searches for other nodes, and subsequently randomly connects to one of the discovered nodes.

In its current form, the connection manager forms tree topologies. Since in a tree there is only one path between any two nodes, we are relieved from explicit route calculation; this reduces the system complexity considerably, easing first experiments and evaluation.

The tree structure is constructed and maintained by two parallel threads. The inquiry thread (see Alg. 1) periodically performs an inquiry and randomly connects to one of the discovered DSN devices.

The packet-handler thread (see Alg. 2) processes negotiation or tree-ID packets arriving from the lower layers. These packets are used to maintain the tree structure by preventing and detecting cycles in the network topology. All nodes connected in a tree share the same tree ID. When two nodes connect, they exchange negotiation packets and compare their tree IDs. If the two nodes were not in the same tree before the connection, their IDs differ and they have to establish a unique ID for the newly formed tree. This is done by agreeing on the larger of the two IDs and broadcasting it in a tree-ID packet to all nodes in the subtree with the smaller ID. If two nodes that are already in the same tree connect, they will notice that they share the same ID and therefore drop the connection. If a node receives a tree-ID broadcast with an ID different from its own, it adopts this new ID. If the received ID is its own ID, there is a cycle in the network and the link over which the broadcast arrived is dropped. This mechanism eliminates cycles that can arise when two subtrees are connected almost simultaneously via two different links (see Fig. 2); in this case, the cycle prevention by negotiation does not work.

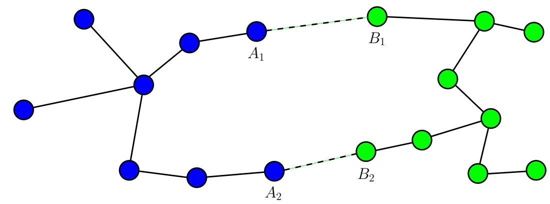

Fig. 2. Cycles can form when disconnected trees are connected almost simultaneously at two different points. The tree-ID broadcast eliminates the cycle shortly afterwards.

If a link is lost, the tree is partitioned, and the two subtrees must not share the same tree ID anymore. Therefore, if a node loses the link over which the current tree ID was received, the node broadcasts its unique device ID as its subtree's new tree ID.

Algorithm 1 Connection manager: inquiry thread

```
loop
  found_nodes := inquiry();
  node := randomly_select(found_nodes);
  connect(node);
end loop
```

Algorithm 2 Connection manager: packet-handler thread

```
loop
  packet := wait_for_packet();
  if packet.type = tree_ID_packet then
    if local_tree_ID = remote_tree_ID then
      drop connection
    else
      local_tree_ID := remote_tree_ID;
      broadcast local_tree_ID to my subtree
    end if
  end if
  if packet.type = negotiation_packet then
    if local_tree_ID = remote_tree_ID then
      drop connection
    else
      if local_tree_ID < remote_tree_ID then
        local_tree_ID := remote_tree_ID;
        broadcast local_tree_ID to my subtree
      end if
    end if
  end if
end loop
```

This local algorithm provides self-healing topologies in a robust and completely distributed fashion. It does not need exhaustive computation or communication[1]. Therefore, it is expected to scale well to a large number of nodes.

[1]The only transmitted data that grows linearly with the number of nodes is the tree ID; since it does not change very often, it is negligible in the overall communication during the operation of the DSN.

D. Multihop Packet Forwarding

The *transport manager* takes care of multihop packet forwarding. It receives information about the available connections from the connection manager and uses these connections to route packets. Note that the transport manager makes no assumptions about the underlying topology. The packet switching at every BTnode is based on virtual-circuit switching and automatically forwards traffic to the appropriate connection based on a virtual-circuit identifier.

For the transport manager, the concept of a *host* is important. A host is a DSN node to which a user device, e.g. a PC, is connected. Hence, a host is a source of commands to the DSN and a sink for data from the DSN. There can be multiple hosts in a DSN.

Communication is always initiated by a host, typically by opening a virtual connection to a specific DSN node. This is done by simply flooding the network with a route-request message. Each DSN node stores the ID of the connection such a message arrived on; this is the route to the host to be used on the return path. When the destination node sends its reply along the return path, the intermediate nodes assign a local virtual-circuit identifier to the connection the reply arrived on; this is the route to the destination node. After the setup is completed, packets can be transported over this virtual connection with only minimal header processing at the intermediate nodes.

If a link fails, the host and all endpoints of broken virtual connections are notified and remove the corresponding virtual-circuit identifiers from their local tables. A host application handles the retransmission of lost packets.

Multiple virtual connections are supported. If the connections are simultaneously active and if their paths overlap partially, the throughput of an individual virtual connection is obviously reduced.

III. Experimental Results

We have tested and measured the properties of our implementation in two different setups. The first one was a lab setup involving 2–40 nodes. We measured the per-hop transmission delay and observed the network-topology construction. In the second setup, we distributed 71 BTnodes on a large office floor, thus obtaining a larger, realistic deployment scenario.

In both setups, all nodes are running the same software. A host PC is connected over a 115 kbps serial link to one of the BTnodes. This node is a host in the DSN and receives topology information from the other BTnodes: Each node sends information about events such as new connections and link losses to the host, and additionally stores them in a local log. The logs are remotely collected by a monitoring and control application running on the host PC (see Fig. 3).

A. Per-Hop Transmission Delay

On a virtual connection spanning multiple hops, the data has to be forwarded hop by hop. The transmission and processing delays add up along the path. We measured the delay by sending time-stamped packets to an endpoint. The packets are returned to the sender, which then measures the round-trip delay. Figure 4 shows the round-trip delay divided by 2. For each hop count (up to 16 hops), we measured the delay of 200 packets of two different sizes.

The figure shows the expected linear behavior. The per-hop delay for the first hop is on average 17 ms for small packets and 45 ms for large packets. For subsequent hops, the average delays are 42 ms and 60 ms, respectively (see Fig. 4). The difference between the small- and large-packet delay is mainly the time needed for transferring more data between the Bluetooth device and the microcontroller.

For serial-port tunneling, the end-to-end delay is of importance. The maximum tolerable delay for a serial connection depends on

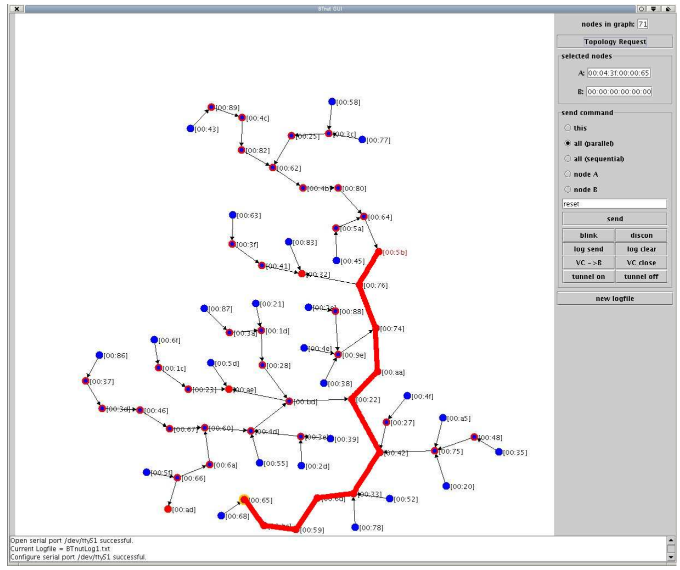

Fig. 3. The DSN monitoring tool shows a scatternet tree topology with 71 nodes. There is a virtual connection from the host node `[00:5b]` over ten hops to node `[00:65]`.

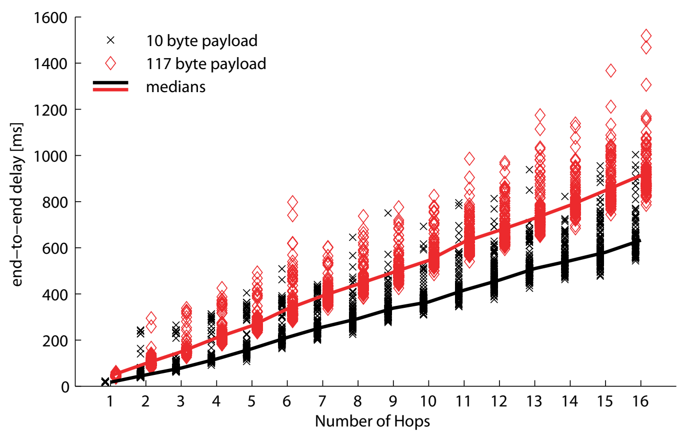

Fig. 4. Per-hop transmission delay: Average transmission delay based on 100 packets of each size.

the application. Typically, a terminal session tolerates minutes before disconnecting, in-system programming can cope with a few seconds, and an interactive user interface requires a maximal delay of 100–500 ms. Our measurements show that for the time being, we have to limit ourselves to tasks that do not require low-delay transmissions from the host to a DSN node that is many hops away.

B. Network-Topology Construction

The topology construction depends on the ability to discover other nodes and to successfully connect to them. Since a-priori assumptions about the state of remote nodes cannot be made before an actual connection, these are highly non-deterministic operations. While a node is inquiring or connecting, it might not be discovered by others. Previous measurements have shown that inquiring is a time-consuming process requiring in the order of tens of seconds [8], [9] for a reliable discovery of all nodes. Experiments have shown that for our connection manager, short but frequent inquiries accelerate the formation of large network clusters. Our experiments were conducted with the following values: inquiries last 3.8 s and pauses between inquiries are chosen randomly (to avoid that all nodes inquire simultaneously) between 3 and 20 s.

The scatternet-construction algorithm introduced in Sect. II-C is truly distributed. Since connections are established in parallel, the algorithm can be expected to scale well with an increasing number of nodes. We have verified this assumption with the following experiment.

Initially, n nodes are switched on one after the other. After all nodes are connected in a single tree, we simultaneously reset them with a broadcast command from the monitoring tool. This then provides a common time base for all nodes. All nodes log their connection and disconnection events, annotated with the time since the last reset. After all nodes are again connected, the monitoring tool retrieves these logs from all the nodes.

Figure 5 illustrates the evaluation of the network-topology construction. Each plot represents the average of ten different experiments with the same amount of nodes. After a boot-up phase of approximately 13 s, the first connections are established. At 20 s, close to 50 percent of all the connections are established, and at 70 s the construction is finished. These values are independent of the number of nodes involved.

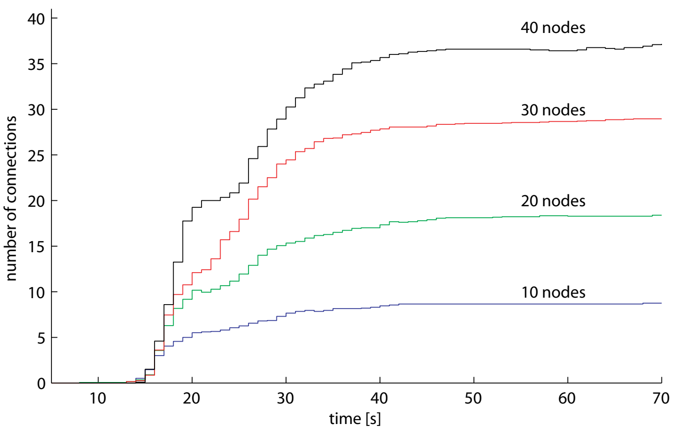

Fig. 5. Initial network-topology construction: Each curve represents the average of ten different experiments.

All n nodes are connected in one tree if and only if there are $n-1$ connections. In some of the 40 experiments, not all n nodes were connected in the end. A yet unresolved software error in the low-level event-handling routines occasionally caused a deadlock in the inquiry thread (Alg. 1). As a consequence, some nodes were not able to discover and connect to other nodes anymore and remained isolated.

This was the reason why the above experiments were not conducted with more than 40 nodes. If many nodes are reset simultaneously, not all of them can connect in the first iterations of the inquiry thread, probably due to radio interference. Thus, the probability that a node's inquiry thread enters the deadlock before the node is connected increases with increasing node density. This problem is not inherent to our algorithm and should disappear with the low-level software errors dissolved.

C. A Realistic Deployment Scenario

To test our DSN in a realistic scenario, we deployed 71 BTnodes on a large office floor. We also wanted to test our hypothesis that the problem described in the previous section should disappear if we reduce the node density.

We therefore distributed the nodes as depicted in Fig. 6, switching them on as we went along. Being switched on one after the other, all 71 nodes joined a single tree scatternet (see Fig. 3) without any problem. We then issued the reset command to all nodes. Within 70 s, 46 nodes had again connected to a tree. As more time passed, the tree did however not grow beyond this size. Essentially, the problem remained as severe as in the lab setup.

The explanation for this is that there is no sufficient difference in connectivity between the lab setup and the floor deployment. This can be seen in Fig. 6: the various connections over relatively long distances show that the average number of neighbors is still very high.

The quality of the long-distance links in Fig. 6 is probably smaller than that of the short-distance links. Furthermore, it may be desirable that the DSN nodes connected to colocated target nodes also be colocated in the DSN. An improved version of our connection manager would hence prefer high-quality links.

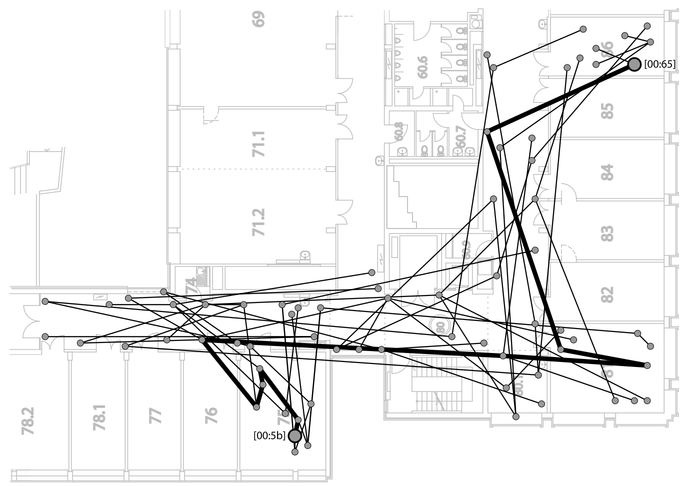

Fig. 6. The example shown in Fig. 3 was set up on a large office floor. The actual connections from Fig. 3 are shown; the virtual connection is highlighted.

IV. Lessons Learned

Our algorithm works and performs well in principle, as the experiments have shown. The remaining problems can by large be attributed to system-software issues that can stall certain nodes without a possibility for recovery.

Using simple local algorithms has been a very encouraging experience due to their traceable and comprehensible nature. The increase in complexity when actually implementing an algorithm is not to be underestimated. In conjunction with the complex behavior of the devices, the exact interpretation of an effect can be very hard.

In current research, there is an apparent gap between the results of theoretical and practical work [10]. Seemingly simple algorithms often rely on the availability of complex functions which are not readily supported by the actual hardware. For instance, a frequently used function such as *send to all neighbors* typically has to be divided into smaller parts such as *search for all neighbors*, *open a connection*, *send the data*, and *close the connection*. If the function additionally has to be reliable, error-handling for all the subfunctions has to be provided to account for imperfections and failures. As described in Sect. II, peculiarities of the hardware may even make some functions impossible to implement.

A setup of many nodes in a small lab facilitates frequent and repeated experiments, but is not very realistic. In experiments involving radios, the high node density of a lab setup can actually lead to interference problems that would not occur in a sparse field setup. As a countermeasure, we reduced the inquiries of nodes connected to the host. This resulted in less interference and therefore faster connection buildup. Since we had observed that inquiring nodes occasionally lose links, the reduction of inquiries also increased the stability of the DSN.

For the thorough analysis of a running system, the reliable extraction of data from the system is vital. This requires storing the data locally and synchronizing it upon extraction. When aggregating data from many nodes, coordination and throughput issues have to be managed. The data of unreachable (e.g. dead) nodes typically is of primary interest. For its extraction, it has to be stored in non-volatile memory and collected after a restart of the node.

V. Conclusions and Outlook

We have presented our implementation of a DSN on the BTnode rev3 platform. Our measurements indicate that our implementation scales well to a large number of nodes. With a DSN spanning 71 BTnodes, we have presented the largest Bluetooth scatternet reported to date.

After the first promising assessment of the DSN idea on the BTnode rev2, the transition to the new hardware was accomplished smoothly. The increased performance and reliability of the new devices has eased the implementation work considerably and has even surpassed our expectations.

The experiments reported here concentrate on the startup phase of the network; the maintenance phase necessitates further investigation. Thorough testing and measurements are needed to find optimal values for parameters such as the inquiry period and duration or the number of connection retries. These values can then be compared to those obtained by simulation [11]. The final result would be a network that is highly agile in the startup phase and that reduces the frequency of operations in the operating phase for maximum stability and minimum overhead.

Our first experiments were conducted with a simple tree topology. For increased resilience against link or node failures, redundant links are very desirable. The modular structure of our software allows us to easily replace the current connection manager. A comparison of various topologies with the reference tree topology can then be performed to trade off resilience against complexity.

The experiments have shown that the impact of the range of the devices was underestimated. In order to improve the topology control we plan to investigate and incorporate link metrics such as RSSI values.

The next step, after our implementation has been further refined, is to test the DSN with a real sensor-network application. This will allow us to measure further parameters using real data. Future activities will include the measurements of throughput, power consumption, and the interference with other radios.

In this paper, we show that scalable DSNs can be implemented on currently available hardware. Our experiences have spawned a number of interesting questions to be pursued. The biggest problem encountered here was actually developing the DSN without having a DSN already in place for the development work.

Acknowledgment

The work presented in this paper was supported by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

References

[1] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler, "An analysis of a large scale habitat monitoring application," in *Proc. 2nd ACM Conf. Embedded Networked Sensor Systems (SenSys 2004)*. Nov. 2004, pp. 214–226, ACM Press, New York.

[2] J. Beutel, M. Dyer, M. Hinz, L. Meier, and M. Ringwald, "Next-generation prototyping of sensor networks," in *Proc. 2nd ACM Conf. Embedded Networked Sensor Systems (SenSys 2004)*. Nov. 2004, pp. 291–292, ACM Press, New York.

[3] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: Accurate and scalable simulation of entire TinyOS applications," in *Proc. 1st ACM Conf. Embedded Networked Sensor Systems (SenSys 2003)*. Nov. 2003, pp. 126–137, ACM Press, New York.

[4] L. Girod, J. Elson, A. Cerpa, T. Stathapopoulos, N. Ramananthan, and D. Estrin, "EmStar: A software environment for developing and deploying wireless sensor networks," in *Proc. USENIX 2004 Annual Tech. Conf.*, June 2004, pp. 283–296.

[5] J. Beutel, M. Dyer, L. Meier, M. Ringwald, and L. Thiele, "Next-generation deployment support for sensor networks," Tech. Rep. 207, Computer Engineering and Networks Lab, ETH Zürich, Switzerland, Nov. 2004.

[6] J.L. Hill, M. Horton, R. Kling, and L. Krishnamurthy, "Wireless sensor networks: The platforms enabling wireless sensor networks," *Communications of the ACM*, vol. 47, no. 6, pp. 41–46, June 2004.

[7] E. Shih, P. Bahl, and M. Sinclair, "Wake on Wireless: An Event Driven Energy Saving Strategy for Battery Operated Devices," in *Proc. 6th ACM/IEEE Ann. Int'l Conf. Mobile Computing and Networking (MobiCom 2001)*. Sept. 2002, pp. 160–171, ACM Press, New York.

[8] O. Kasten and M. Langheinrich, "First experiences with Bluetooth in the Smart-It's distributed sensor network," in *Workshop on Ubiquitous Computing and Communication, Int'l Conf. Parallel Architectures and Compilation Techniques (PACT 2001)*, Sept. 2001.

[9] E. Welsh, P. Murphy, and J.P. Frantz, "Improving connection times for Bluetooth devices in mobile environments," in *Proc. Int'l Conf. Fundamentals of Electronics Communications and Computer Sciences (ICFS 2002)*, March 2002.

[10] F. Kuhn, T. Moscibroda, and R. Wattenhofer, "Initializing newly deployed ad hoc and sensor networks," in *Proc. 10th ACM/IEEE Ann. Int'l Conf. Mobile Computing and Networking (MobiCom 2004)*. 2004, pp. 260–274, ACM Press, New York.

[11] S. Basagni, R. Bruno, G. Mambrini, and C. Petrioli, "Comparative performance evaluation of scatternet formation protocols for networks of Bluetooth devices," *Wireless Networks*, vol. 10, no. 2, pp. 197–213, March 2004.