

Titan: A Tiny Task Network for Dynamically Reconfigurable Heterogeneous Sensor Networks

Clemens Lombriser, Daniel Roggen, Mathias Stäger, Gerhard Tröster

Wearable Computing Lab, ETH Zurich
CH-8092 Zurich, Switzerland
{lombriser,roggen,staeger,troester}@ife.ee.ethz.ch

Abstract. Context recognition, such as gesture or activity recognition, is a key mechanism that enables ubiquitous computing systems to proactively support users. It becomes challenging in unconstrained environments such as those encountered in daily living, where it has to deal with heterogeneous networks, changing sensor availability, communication capabilities, and available processing power.

This paper describes Titan, a new framework that is specifically designed to perform context recognition in such dynamic sensor networks. Context recognition algorithms are represented by interconnected data processing tasks forming a task network. Titan adapts to different context recognition algorithms by dynamically reconfiguring individual sensor nodes to update the network wide algorithm execution.

We demonstrate the applicability of Titan for activity recognition on Tmote Sky sensor nodes and show that Titan is able to perform processing of sensor data sampled at 100 Hz and can reconfigure a sensor node in less than 1ms. This results in a better tradeoff between computational speed and dynamic reconfiguration time.

1 Introduction

Ubiquitous computing systems aim to integrate seamlessly into the environment and to interact with users in unobtrusive ways. As a consequence, the human-computer interface needs to become more intelligent and intuitive [1,2]. One way to achieve this is to make the computing system aware of the context of the user. By knowing what the user is doing, the computing system can proactively support him.

Activities of a person may be recognized using small, wirelessly interconnected sensor nodes worn on the person's body, like motion sensors such as accelerometers integrated in garments or sensors placed in the environment, such as microphones. The sensor nodes must integrate seamlessly into the wearer's clothes. Therefore they need to be of small dimensions and can only offer very limited computational power and memory. Yet activity recognition on such resource-limited devices is still possible using appropriate algorithms [3], which work on an intelligent choice of sensor signal features correlated to the activities to be recognized [4,5].

Context recognition becomes challenging in real-world, unconstrained environments, such as those which are likely to be encountered in daily living situations. In such environments, assumptions on the number and type of sensors available on the body and in the environment are difficult to make. Sensors in the environment may be only available for a short time, e.g. as the user walks by, and context recognition methods must be able to quickly incorporate such temporarily available information. Context recognition gets even more complex when hardware, communication, or power failures are considered.

This paper describes for a system the first time specifically designed to support the implementation and execution of context recognition algorithms in such dynamically changing and heterogeneous environments. We refer to this system as *Titan* – a Tiny Task Network. Titan represents data processing by a data flow from sensors to recognition result. The data is processed by *tasks*, which implement elementary computations like classifiers or filters. The tasks and their data flow interconnections define a task network, which runs on the sensor network as a whole. The tasks are mapped onto the single sensor nodes according to the sensors and the processing resources they provide.

Titan dynamically reprograms the sensor network to exchange context recognition algorithms, handle defective nodes, variations in available processing power, or broken communication links. It has been designed to run on sensor nodes with limited resources such as those encountered in wireless sensor networks.

We have implemented and tested Titan on Tmote Sky [6] sensor nodes and evaluated the performance of the task network execution. The comparison to existing approaches shows that Titan offers a better tradeoff in computational speed vs. dynamic reconfiguration time.

This paper is organized as follows: Section 2 reviews existing approaches for data processing systems in wireless sensor networks. Section 3 describes Titan. It is then evaluated and discussed against a selection of other existing systems in Section 4. In Section 5 and 6 we highlight our future work and eventually conclude this paper.

2 Related Work

An analysis of context recognition methods based on body-worn and environmental sensors was carried out in [7] and has led to the development of the *Context Recognition Network* [8]. It has been designed to run on Linux and to compute context information on a central powerful computer that collects sensor data from the environment. However, this system does not tackle the issue of context recognition in heterogeneous and dynamically organized sensor networks, where the computation is distributed over the whole network.

An approach to dynamic reconfiguration of data processing networks on sensor networks is DFuse [9]. It contains a data processing layer that can fuse data while moving through the network. To optimize the efficiency, the data processing tasks can be moved from one node to another. However, DFuse is targeted at devices with higher processing capabilities than sensor nodes provide.

The *Abstract Task Graph (ATaG)* [10] with its DART runtime system [11] allows to execute task graphs in a distributed manner. The task graph is compiled during runtime and adapted to the configuration of the network. Similar to DFuse, DART imposes too high requirements on the underlying operating system, such that it cannot be run on sensor nodes we want to use.

Dynamic reconfigurability was investigated by providing dynamic loading of code updates in Deluge [12], TinyCubus [13], SOS [14], or [15]. Dynamic code updates however rely on homogeneous platforms (i.e. the same hardware and OS), which is unlikely to be the case in large scale unconstrained networks such as the ones we consider here. In addition, dynamic code loading is time consuming and requires the node to stop operating while the code is uploaded. This may lead to data loss during reconfiguration.

A platform independent approach is to use a virtual machine like Maté [16]. Applications running in Maté use instructions that are interpreted by virtual processors programmed onto network nodes. The performance penalty of the interpretation of the instructions can be alleviated by adding application-specific instructions to the virtual machine [17]. These instructions implement functionality that is often used by the application and execute more efficiently. In contrast to Maté, Titan uses processing tasks as basic building blocks, which include more functionality and thus execute more efficiently.

3 Titan

3.1 Overview

The goal of Titan is to provide a mechanism to dynamically configure a data processing task network on a wireless sensor node network. As the physical location of the sensors on the sensor nodes is important for their contribution to the context recognition application, the individual sensor nodes must be individually reprogrammable.

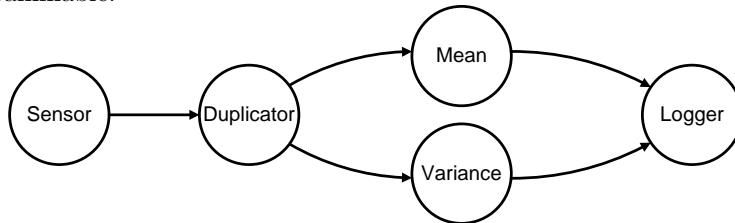


Fig. 1. This task network describes a simple application which reads sensor data, computes two features and stores the results in the Flash memory

Titan provides a set of *tasks*, of which each implements some signal processing function. *Connections* transport the data from one task to another. Together, the tasks and connections form a *task network*, which describes the application to be run on the wireless sensor network. Figure 1 shows an example of such a task network that reads data from a sensor, computes the mean and variance, and stores both values for logging purposes.

Programming data processing in this abstraction is likely to be more intuitive and less error prone than writing sequential code. The inner working of every processing task have to be thoroughly checked only once. This can be done in an isolated way and reduces the complexity of debugging.

Tasks have a set of input ports, from which they read data, process it, and deliver it to a set of output ports. Connections deliver data from a task output port to a task input port and store the data as *packets* in FIFO queues.

The application is issued by a *master node*. It analyzes the task network and splits it into *task subnetworks*, which are to be executed on the individual nodes. The connections between tasks on different nodes is maintained by sending the packets in messages via a wireless link protocol.

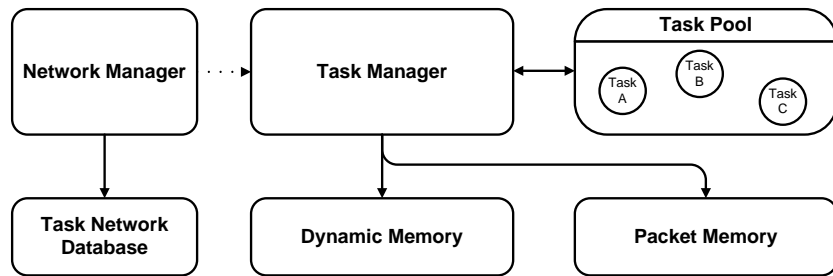


Fig. 2. Main modules of the Titan architecture. The arrows indicate in which direction functions can be called

Titan is built on top of the TinyOS operating system [18], which has been designed to be used on very resource limited sensor nodes. The Titan architecture is shown in Figure 2 and includes the following main modules.

- The **Network Manager** organizes the network to execute the task networks. It connects to the **Task Network Database** where the task network descriptions are located and decides which sensor nodes will execute which part of the task network. These modules are only available on sensor nodes with enough resources to perform the network management.
- The **Task Manager** is the system allowing to reconfigure a sensor node. It maintains a task pool with all tasks available on the sensor node and instantiates the tasks according to the Network Manager’s requests. The Task Manager is responsible for reorganizing the task subnetwork executed on the local sensor node during a reconfiguration.
- A **Dynamic Memory** module allows tasks to be instantiated multiple times, and reduces static memory requirements of the implementation. The tasks can allocate memory in this space for their individual state information. This module is needed as TinyOS does not have an own dynamic memory management.
- The **Packet Memory** module stores the packets used by the tasks to communicate with each other. The packets are organized in FIFO queues, from which tasks can allocate packets before sending them. This data space is shared among the tasks.

3.2 Tasks

Titan defines a global set of tasks. Each task implements a data processing algorithm or an interface to the hardware available on the sensor node. Every node in the sensor network implements a subset of all tasks, depending on its processing capabilities. The tasks go through the following phases when they are used:

1. **Configuration** – At this point, the Task Manager instantiates a task. To each task it passes configuration data, which adapts the task to application needs. Configuration data may include sampling frequency, the window size to use, or similar. The task can allocate dynamic memory to store state information.
2. **Runtime** – Every time a task receives a packet, it is allowed to execute to process the data. Titan provides the task with the state information it has set up during the configuration time. Tasks are executed in the sequence they receive a packet, and each task runs to completion before the next task can start.
3. **Shutdown** – This phase is executed when the task subnetwork is terminated on the node. All tasks have to free the resources they have reserved.

3.3 Connections

Packets exchanged between the tasks carry a timestamp and information of the data length and type they contain. Tasks reading the packets can decide on what to do with different data types. If unknown data types are received, they may issue an error to the Task Manager, which may forward it to the Network Manager to take appropriate actions.

To send a packet from one sensor node to another, Titan provides a *communication* task, which can be instantiated on both network nodes to transmit packets over a wireless link protocol as shown in Figure 3. During configuration time the communication task is told which one of its input ports is connected to which output port of the receiving task on the other node. The two communication tasks handle communication details, such as routing or reliable transmission of the packet data in the wireless sensor network. The communication task is automatically instantiated by the Network Manager to distribute a task network over multiple sensor nodes.

The recommended maximum size of a packet for Titan is 24 bytes, as it can easily be fitted with 5 bytes header into a TinyOS active message. The active message is used to transmit data over wireless links and offers 29 bytes of payload.

3.4 Dynamic Allocation of the Task Network

A programmer designs his application by selecting tasks from the task list Titan provides and interconnecting them to form a task network. Task parameters as

well as location constraints can also be defined. The description of the resulting task network is then loaded into the *Task Network Database* in the network.

When the execution of a specific task network is requested, the Network Manager first inspects the capabilities of the sensor nodes in the environment by broadcasting a *service discovery* message containing a list of tasks to be found. Every node within a certain hop-count responds with the matching tasks it has in its Task Pool, and adds status information about itself. Using this information, Network Manager decides whether the task network can be executed, and which node runs which tasks. This is currently done by first placing tasks with location constraints on the first node found with matching requirements. In a second step, the remaining tasks are greedily added to the nodes. If the node capacities are reached, new nodes are added to the processing network until all tasks have been placed.

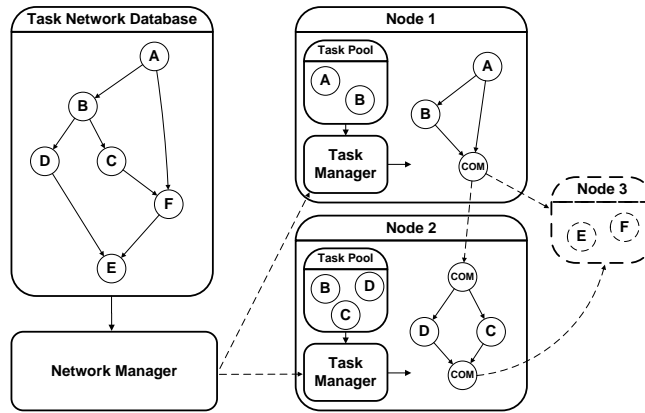


Fig. 3. Allocation of a task network: parts of the task network are configured onto each participating node, depending on their sensors or computational capabilities. Interconnections across sensor nodes are realized over special communication tasks

When data needs to be exchanged across nodes, communication tasks (see Section 3.3) are automatically inserted. The resulting task subnetworks containing the tasks to be executed on every sensor node are then sent to each participating node's Task Manager, which takes care of the local instantiation as shown in Figure 3. After the configuration has been issued, the Network Manager keeps polling the Task Managers about their state and changes the network configuration if needed. On node failures, the Network Manager recomputes a working configuration and updates the subnetworks on individual sensor nodes where changes need to be made, resulting in a dynamic reorganization of the network as a whole.

3.5 Synchronization

When sensors are sampled at two sensor nodes and their data is delivered to a third node for processing, the data streams may not be synchronized due to dif-

fering processing and communication delays in the data path. As a consequence, a single event measured at the two nodes can be mistaken for two.

If the two sensor nodes are synchronized by a timing synchronization protocol, a timestamp can be added to the data packet when it is measured. The data streams can then be synchronized by matching incoming packets with corresponding timestamps. Timing protocols have been implemented on TinyOS with an accuracy of a few 10 μs [19,20].

If the two sensor nodes are not synchronized, the sensor data can be examined as in [8]. The idea is to wait until an event occurs that all sensors can measure, e.g. a jump for accelerometers on the body. Subsequent packets reference their timestamp to the last occurrence of the event. This functionality is provided in the *Synchronizer* task.

4 Evaluation

We have implemented and tested Titan on Tmote Sky motes [6]. The memory required by the implementation is listed in Table 1, showing the memory footprints of a plain TinyOS implementation, the virtual machine Maté, the code distribution framework Deluge, and Titan. These numbers change when compiled for different platforms, but give an indication of the size of the Titan implementation.

The space reserved for dynamic and packet memory RAM can be tailored to the needs of the application and the resources on the node. The task number and type in the Task Pool on the node determines the amount of ROM memory requirement, and can be adapted to the platform as well. The memory footprint shown in Table 1 includes all Titan tasks as listed in Table 3.

Platform	ROM	RAM	Interface function	Cycles	Time (μs)
TinyOS ¹	16520	541	pasteContext	85	16
TinyOS with Deluge	26896	1089	getContext	145	28
Maté	37004	3146	allocPacket	370	70
Titan	35024	1422	sendPacket	290	55
dynamic memory		+ 4096	hasPacket	25	5
packet memory		+ 1440	getNextPacket	425	81
			Transfer 1 packet (avg)	1026	195

Table 1. Memory footprints (bytes). The Tmote Sky module provides 48k ROM and 10k RAM.

Table 2. Cycle count of the most important Titan interface functions

Table 2 lists the time needed for the most important functions Titan offers to the tasks. All times have been measured by toggling a pin of the microcontroller on the Tmote Sky. The average packet transfer is measured from the point where the sending task calls the sending function to the time where the receiving task has retrieved the packet and is ready to process its contents. This time is roughly

¹ As distributed with Tmote Sky modules, and instantiating the Main, TimerC, GenericComm, LedsC, and ADCC components

200 μ s. For the recognition of movements, acceleration data is usually sampled at less than 100 Hz [4], which leaves the tasks enough time for processing.

Table 3 shows the currently available tasks for Titan. The delays given in the table indicate how long each task needs to process a data packet of 24 bytes, which is the recommended Titan packet size as mentioned in Section 3.3.

Name	Descriptions	Delay	RAM	ROM
Duplicator	Copies a packet to multiple output ports	192 μ s	–	250
FBandEnergy	Computes the energy in a frequency band from FFT data	200 μ s	12	410
FFT	Computes a 32 bit real-valued FFT over a data window of $n = 2^k$ samples (delay for 128 16 bit samples)	186ms	$16 + 4n + n_s$	4714
Led	Displays incoming data on the mote LED array	36 μ s	–	260
Mean	Computes the mean value over a sliding window of size n	318 μ s	$12 + n_s$	494
Merge	Merges multiple packets into one	328 μ s	12	454
MinMax	Looks for the maximum and minimum in a window of size n	193 μ s	8	484
ExpAvg	Computes an exponential moving average over input data	222 μ s	8	416
Synchronizer	Synchronizes data by dropping packets until an user defined event occurs	220 μ s	10	476
Threshold	Quantizes the data by applying a user-defined number n of thresholds	95 μ s	$4 + 2n$	424
TransDetect	Detects value changes in the input signal, and issues a packet with the new value	201 μ s	2	474
Variance	Computes the variance over a sliding window of size n	1510 μ s	$16 + n_s$	720
Zero crossings	Counts the number of zero crossings in the data stream	176 μ s	8	370

Table 3. Titan task set and delay T_i on a Tmote Sky. RAM indicates the number of dynamic memory bytes allocated, ROM the bytes of code memory used. The delay has been computed for a packet of 22 bytes data. n_s gives memory bytes needed to store n in the data type used, e.g. for 16 bit values $n_s = 2n$

Most task delays are in the range of a few hundred microseconds, which shows that a task network has enough time to execute when using a sampling frequency of 100 Hz. Even an FFT can be performed over 128 samples in 180ms, leaving 86% of the sample time for processing. Whether a whole task network can process the sampled data in real-time needs further analysis. If the sensor data is sampled with a frequency of f_{ADC} and the recorded samples are issued in packets of N_{ADC} samples, the time left for processing of the local task network is:

$$T_{free}(N_{ADC}, f_{ADC}) = \frac{N_{ADC}}{f_{ADC}} - N_{ADC} \cdot t_{sample} - t_{ADCMsg} \quad (1)$$

Where t_{sample} is the time needed by the sensor to sample one sample, and t_{ADCMsg} the time needed to issue a packet.

The time needed for the processing of the data, i.e. executing the task network is determined by the delays T_i of the allocated tasks with their configuration D_i , and the number N_p of messages exchanged, which needs the time t_p .

$$T_{used}(T) = N_p \cdot t_p + \left(\sum_{\forall i \in T} T_i(D_i) \right) + O(T) \quad (2)$$

The TinyOS scheduler overhead is included by the $O(T)$ function. The execution of the task network is thus feasible if the following inequality holds true:

$$T_{used} < T_{free} \quad (3)$$

In a heterogeneous network, the times needed to execute a task differ from node to node, such that an adaption of the times are needed. This can be done by a simple factor as proposed in [7], where every node indicates a speed grade that is multiplied with the task execution time. A more exact approach would be to store a task execution time table on every node, which the Task Manager uses to determine whether the assigned task network is actually executable.

4.1 Configuration Times

To analyze the time of a reconfiguration, we have configured a node with a task subnetwork containing a counter task that increments every second and sends its data to the LED task, which shows the counter value on the Tmote’s LEDs. The task subnetwork description has a size of 19 bytes and fits in a single configuration message. Table 4 shows times needed from the reception of the configuration message to the point where the task subnetwork runs.

Task	Time [μs]
Process configuration message	260
Clearing existing task subnetwork	56
Configuration & Startup	196
Total (with OS overhead)	650

Table 4. Analysis of the reconfiguration process

If a reconfiguration needs multiple configuration messages, Titan stops the current task subnetwork on the reception of the first message. The configuration of the new task subnetwork is then continued every time new configuration packets are received. As soon as the task subnetwork information is complete, Titan starts the execution and notifies the Network Manager of that fact. The continuous processing of the incoming configuration messages reduce the delay after the reception of the last message, as it only includes the configuration and startup time.

4.2 Case Study

To compare the Titan framework to other systems, we have chosen a simple application and have implemented it in three systems: Titan, Maté [16], and Deluge [12]. Maté provides a virtual machine on network motes and thus allows to distribute code in heterogeneous networks, while Deluge allows to reprogram

notes and thus allows to upload platform optimized application-specific code. Thus the comparison involves a general and a specialized solution next to Titan.

The test application continuously samples a sensor at 10 Hz, calculates the maximum, the minimum, and the mean over 10 samples, and sends them to another node. We have measured the number and total size of the configuration messages to be sent, and evaluated how long the processing of the samples takes on the node. The results can be seen in Table 5, respectively in Table 6.

Platform	microseconds	Platform	Size [Bytes]	Packets
Titan	3.68 <i>ms</i>	Titan	71	4
Maté	24.0 <i>ms</i>	Maté	75	4
Deluge	201 μ s	Deluge	29588	1345

Table 5. Processing delay for the case study **Table 6.** Configuration data size study

The numbers show that Titan executes 6.5 times faster than the Maté virtual machine and has a similar configuration size. Deluge on the other hand has an application specific image and is about 18x faster than Titan, but, due to the large number of configuration messages, it needs several seconds for transferring a program image and reboot. This time is not acceptable in the context recognition applications that we envision, where sensors, computational power, and communication channels may change dynamically and in unpredictable ways depending on the user location, motion, social interaction, etc. Deluge does allow to store a certain number of configurations, depending on the node Flash memory, but this allows only a small number of different task sets to execute, while Titan can be reconfigured to a much broader range of applications.

Note that we have chosen a simple application capable of running on Maté. Maté is not able to support sampling rates higher than 10 Hz. It neither can compute a FFT at 10 Hz in realtime, which Titan is able to do. Being able to compute a FFT in realtime is important as many features for activity recognition are gained from the frequency space [3,5].

5 Discussion and Future Work

The work presented here shows that Titan is capable of performing simple context recognition tasks. We are working on extending the Titan task set with classifiers and context recognition algorithms that optimally exploit the sensors on the body and in the environment. These algorithms adapt to the sensors available at the moment and reorganize the computation to changing conditions. This will allow Titan to tackle more complex context recognition tasks.

There are many parameters in the algorithms for the distribution of the task network and the monitoring and failure handling that need to be investigated in more detail. A better approach would use power or transmission cost metrics to arrange tasks on the network. It would also be interesting to enable Titan to dynamically rearrange single tasks during the execution instead of reconfiguring whole nodes if changes to the task network configuration need to be made.

Titan currently only allows to run one configuration at a time. Multiple task networks could be running in parallel and be configured on the nodes separately. This would require the Task Manager to handle accesses to resources that are available only one time on a node and are used with different parameters.

6 Conclusion

We have described the Titan architecture and how it can execute context recognition algorithms in dynamically changing and heterogeneous sensor networks. Titan provides a set of tasks with functionality and interconnects them to form a task network, which describes the data processing algorithm. The task networks are then split into task subnetworks and assigned to the different nodes in a heterogeneous network.

In unconstrained, every-day environments the available sensors, computational power, and communication links may change unpredictably. Titan provides fast reconfiguration and high processing speed, which make it an ideal platform for context recognition in such environments in comparison to virtual machine or dynamic code upload approaches. Titan provides a set of data processing tasks that can be used for various applications and offers easy programming. The task networks are automatically adapted to the sensor node network and are able to compute user context in the network.

The results of the implementation show that a sensor node reconfiguration can be made in less than 1 ms, and sampling rates of 100 Hz can be supported with enough free processing time for recognition algorithms. Thus Titan offers a better tradeoff between processing time and dynamic reconfiguration delay than related approaches.

Acknowledgment: This paper describes work undertaken in the context of the e-SENSE project, 'Capturing Ambient Intelligence for Mobile Communications through Wireless Sensor Networks' (www.ist-e-SENSE.org). e-SENSE is an Integrated Project (IP) supported by the European 6th Framework Programme, contract number: 027227

References

1. Mann, S.: Wearable Computing as Means for Personal Empowerment. In: Proceedings of the 3rd International Conference on wearable Computing (ICWC). (1998) 51–59
2. Starner, T.: The Challenges of Wearable Computing: Part 1 and 2. *IEEE Micro* **21**(4) (2001) 44–67
3. Stäger, M., Lukowicz, P., Tröster, G.: Implementation and Evaluation of a Low-Power Sound-Based User Activity Recognition System. In: Proceedings of the 8th IEEE International Symposium on Wearable Computers (ISWC). (2004) 138–141
4. Huynh, T., Schiele, B.: Analyzing features for activity recognition. Proceedings of the 2005 joint conference on Smart objects and ambient intelligence: innovative context-aware services: usages and technologies (2005) 159–163

5. Bharatula, N.B., Stäger, M., Lukowicz, P., Tröster, G.: Empirical Study of Design Choices in Multi-Sensor Context Recognition Systems. In: Proceedings of the 2nd International Forum on Applied Wearable Computing (IFAWC). (2005) 79–93
6. Moteiv Corporation: Tmote Sky: <http://www.moteiv.com> (2005)
7. Anliker, U., Beutel, J., Dyer, M.,ENZler, R., Lukowicz, P., Thiele, L.: A Systematic Approach to the Design of Distributed Wearable Systems. *IEEE Transactions on Computers* **53**(8) (2004)
8. Bannach, D., Kunze, K., Lukowicz, P., Amft, O.: Distributed Modular Toolbox for Multi-modal Context Recognition. In: Proceedings of the 19th International Conference on Architecture of Computing Systems (ARCS). (2006) 99–113
9. Kumar, R., Wolenetz, M., Agarwalla, B., Shin, J., Hutto, P., Paul, A., Ramachandran, U.: DFuse: A Framework for Distributed Data Fusion. In: Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys), New York, NY, USA, ACM Press (2003) 114–125
10. Bakshi, A., Prasanna, V.K.: Programming Paradigms for Networked Sensing: A Distributed Systems’ Perspective. In: 7th International Workshop on Distributed Computing (IWDC). (2005)
11. Bakshi, A., Pathak, A., Prasanna, V.K.: System-level Support for Macroprogramming of Networked Sensing Applications. In: Proceedings of the International Conference on Pervasive Systems and Computing (PSC). (2005)
12. Hui, J.W., Culler, D.: The dynamic behavior of a data dissemination protocol for network programming at scale. In: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, ACM Press (2004) 81–94
13. Marron, P.J., Lachenmann, A., Minder, D., Hahner, J., Sauter, R., Rothermel, K.: TinyCubus: A Flexible and Adaptive Framework Sensor Networks. Proceedings of the Second European Workshop on Wireless Sensor Networks (2005) 278–289
14. Han, C., Kumar, R., Shea, R., Kohler, E., Srivastava, M.: A Dynamic Operating System for Sensor Nodes. In: 3rd International Conference on Mobile Systems, Applications, and Services. (2005) 163–176
15. Dulman, S., Havinga, P.: Architectures for Wireless Sensor Networks. In: Proceedings of the International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP). (2005) 31–38
16. Levis, P., Culler, D.: Maté: A Tiny Virtual Machine for Sensor Networks. *ACM SIGOPS Operating Systems Review* **36**(5) (2002) 85–95
17. Levis, P., Gay, D., Culler, D.: Active Sensor Networks. In: Proceedings of the 2nd USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI). (2005)
18. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for network sensors. In: Architectural Support for Programming Languages and Operating Systems. (2000)
19. Elson, J., Girod, L., Estrin, D.: Fine-grained network time synchronization using reference broadcasts. In: Proceedings of Fifth Symposium on Operating Systems Design and Implementation (OSDI). (2002) 147–163
20. Ganeriwal, S., Kumar, R., Srivastava, M.B.: Timing-sync Protocol for Sensor Networks. In: Proceedings of the 1st International Conference on Embedded Networked Sensor Systems. (2003) 138 – 149