tinyDSM: A Highly Reliable Cooperative Data Storage For Wireless Sensor Networks

Krzysztof Piotrowski, Peter Langendoerfer and Steffen Peter
IHP, Im Technologiepark 25, 15236 Frankfurt (Oder), Germany,
{piotrowski|langendoerfer|peter}@ihp-microelectronics.com

ABSTRACT

The advantage of a Wireless Sensor Network (WSN) compared to a centric approach is the distribution of sensing suites. However, in order for such a system of distributed resources to work in a reliable and effective way a smart cooperation between nodes is needed. In this paper we propose a middleware approach for a highly reliable data storage that helps to assure data availability despite the well known WSN resource problems and disappearing or inactive nodes by providing a reasonable data redundancy in the system. Such a solution helps to ease the design and optimization of the data exchange between nodes as well. Our solution is configurable in order to satisfy the needs of the application on top regarding performance/requirements trade-off. The options specify the quantity and quality of the data replication. Additional features like event mechanism that monitors the data and the possibility to issue database like queries increase the applicability of our middleware. In this paper we focus on the evaluation of its capabilities regarding reliability, the consistency of replicates and the costs of the data management. The simulation results for a reasonable set-up show that the CPU load caused by the data replication is low (below 3 percent) and the average inconsistency time is as small as about 0,06 seconds for a single hop and about 0,15 seconds for a two hops replication area. There is still room for improvements, but a clear definition of problems helps to find ways to cope with them in order to achieve the chosen goals.

KEYWORDS: Cooperative Computing, Wireless Sensor Networks, Software Development, Distributed Shared Memory, Distributed Data Storage.

1. INTRODUCTION

Wireless Sensor Networks (WSNs) are considered to be the key enabling technology for a large variety of innovative applications. The majority of the applications is using the WSNs in a passive way, i.e., they are used to measure and store data that is then requested whenever the application needs it by sending queries into the WSN. An even simplier scenario is where the measured data is directly forwarded towards the sink. The driving force of our approach is to put more intelligence or at least some decision taking means into the WSN in a distributed manner in order to make the network reactive and thus, more autonomous. Since the autonomous decisions are based on the data, a reliable and robust distributed data storage is necessary to enable this functionality.

On the other hand, WSNs are exposed to harsh conditions, suffer from scarce resource and are required to have long lifetimes without requiring any maintenance. Ensuring fast and reliable access to the data stored in a WSN requires a redundant data storage. In order to assure data availability in case of disappearing or sleeping nodes the data redundancy by means of data replication is needed. The questions about the required number of replicas, the overhead caused by their distribution and the problem of the replica consistency need to be faced.

Our solution, the middleware we called tinyDSM, provides a reliable distributed data storage. It defines the shared data pieces and allows defining the replication area in terms of number of hops the data item is going to be replicated within. The level of reliability is configurable as well, providing means to determine the consistency level, e.g., by specifying the percentage or number of positive acknowledgements in order to provide a specified number of fresh replicas after each or every n-th update operation. Thus, our middleware provides configuration means for quantitative and qualitative definition of the replication.

We simulated our approach for a network of 400 nodes. One of the performance figures is the delay between the change of the value on the source node and the last update

of a copy. We assume here that all copies where updated. In order to avoid consistency problems caused by partial update due to sleeping nodes or message collisions a lifetime period for a copy may be specified to invalidate it if it is not updated on time. The simulation results show that the average inconsistency time is about 0,06 seconds for one hop and about 0,15 seconds for two hops replication area. However, this values are mostly influenced by the medium access delay added to reduce the collisions of acknowledgements at the source node, thus, there is still room for improvements.

Another feature of the middleware we want to verify in this paper is its adaptability to the desired number of copies. The simulations show as well that the adaptation algorithms behave well, that the CPU load is low and the communication overhead is acceptable.

The rest of this paper is structured as follows. Section 2 introduces shortly the goals we want to achieve with our middleware. The following section presents the detailed description of the proposed solution. Section 4 provides the evaluation of our prototype implementation. The paper concludes with a short outline on future research.

2. MOTIVATION

Before we specify our goals we need to think about the application environment, the Wireless Sensor Network. This environment can be classified as a loosely coupled multiprocessor system, with a potentially great number of nodes, each equipped with own memory, networking means and a processing unit. However, in this system there are several limitations that need to be kept in mind. The nodes have constrained resources, i.e., their computation power and the available amount of memory and energy are quite limited. Due to code memory limitations it is hardly possible to have all possible algorithms implemented and available to freely change the behaviour according to the situation. The network as a collection of nodes is constraint as well, the communication between nodes happens at low transmission speed and the link reliability is quite poor causing transmission losses and delays in case of retransmission. But then, the retransmissions cause extreme costs regarding the energy consumption. The limited available energy causes a trade-off between the lifetime of the desired system and the node activity ratio dependent on the allowed amount of power to be consumed by the node. All those constraints combined with the fact that the network is not always static, since nodes may disappear or new nodes may appear, causes the programming of WNS applications to be a very challenging task.

In order to specify the application field, we made several assumptions regarding the requirements and/or limitations a system based on our middleware has to cope with. So, first of all, we do not limit the size of the network. In such a large network the access pattern shows that a data item is usually locality bound, i.e., an unprocessed and original information is usually relevant for nodes in a certain locality only. We assume the network to be highly dynamic, meaning that, on one hand, the nodes can be added at will and, on the other hand, they can disappear permanently because of lack of energy or temporarily because of communication issues. Thus, the nodes can be rather considered as temporary medium for the data than a kind of solid infrastructure.

Our goal is to provide a reliable data storage middleware for WSN that distributes the data items among the nodes in the network assuring the availability and cooperative access to the data in a given source node bound locality. Additionally, the replicas of the original data needs to be managed in order to provide a consistent global view on each data item. This means, we need to provide mechanisms that cope with the above mentioned problems while providing data management functionality that is reliable and consistent to the chosen extent. Since there is no solution that fits all needs regarding the replication range and density, we want to have the replication configurable in terms of quantity and quality. Having that, we want to investigate the relationship between those two factors and their costs.

3. OUR SOLUTION

The basic idea of tinyDSM is to provide means that allow sensor nodes to share their data in an application dependent way. To ease the access to own data a sensor node—the owner—broadcasts the data and some nodes in its locality create local replicas of the data. By that any of these sensor nodes is able to perform directly a read operation on the local copy and is thus able, for example, to answer queries about the stored data without forwarding the query to the owner node. The choice if a node becomes a replicator is generally random but is influenced by the initial configuration regarding the replica number and/or density. The changing of a value belonging to other node, i.e., a write operation, is possible as well, however, it requires the acceptance of the owner node and the actual write is de facto done by the owner.

As already mentioned, an essential feature of data replication is that it assures the information to be available even if some nodes are exhausted or in sleep mode. Figure 1 shows the idea of locality bound replication in a WSN (a) and its advantage in case of an external query (b). The replica holders are represented by light grey dots and the owner is the black dot.

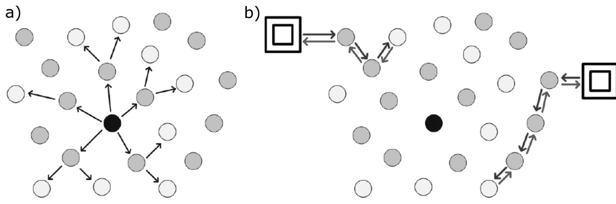

Figure 1. Data Replication (a) and an External Request (b)

The notion of locality is here specified as a kind of n-to-n relationship that associates nodes to groups. In our solution we focus on the definition of locality in terms of communication range with a specified number of hops. This coarse specification is controlled by the replication range—the first of the quantity parameters. And since the number of nodes in the replication range depends on the density of the network, we define a sub-group in that locality—the replicators. This is the target group for storing the replicas of the data item in question. The membership in this sub-group is based on a random decision influenced by another quantity parameter—either represented by the number of replicas or their density. This parameter specifies the number of requested replicas directly or by the percentage of nodes in the locality that are requested to hold the copy. The random character of the initial distribution decision provides an equal distribution of the replicators in the locality. Additionally, the increase of number of replicators is only possible in the acquisition phase of the update, in which the receivers are provided with the requested replication probability.

Current implementation assumes the replication decision to be static, i.e., once a node decides to become a replicator it is assumed to be one until it dies or is not reachable any more from any reason. This is caused by the fact that a replicator node holds the current value of the data as well as its historical values and too frequent changes in the list of replicated items would cause this list to be long causing intensive RAM usage and the history data would be fragmented. However, in an application that is not interested in historical data dynamic replicator changes could be an advantage, causing the distribution to be even more random.

On the other hand, to define and control the quality of the replication, our middleware allows several options to verify the replication achievements against the requirements and to adapt the behaviour according to the results. These options include several strategies for counting the replicas—even after every update in very reliability demanding applications. If the number of copies is too small then the replication probability sent in the update message is increased resulting in additional replicator acquisition. Of course the verification increases the communication overhead caused by the replication, introducing the trade-off between the quality of replication and its costs. And since the quality of replication indicates the level of reliability the data storage provides, the quality parameters allow to shape the middleware according to the needs and allowed expenses.

The definition of data items is currently static and global, i.e., each data item is predefined at compile time as a variable of a specified type and the set of them is global for all nodes that build up the application/network. This helps to avoid data misinterpretation in the application, e.g., if the nodes need to exchange data and use them, for example, to change own state or to perform calculations. So, imagine an example application, where the designer defines the following set of variables: *TEMPERATURE*, *HUMIDITY* and *LIGHT*, all of byte type. In that case it is clear for all nodes what is stored in each variable. Of course a definition of a variable that represents an array or structure is possible. In case of a byte array *DATA[5]*, five byte variables will be defined and each of them may be replicated separately or as a bundle. Same applies for structures. Of course, multi-purpose variables are possible as well, e.g., an integer variable *INFO* may contain temperature or humidity or light measurements depending on the current state of the global application. The defined variables may be visualized on each node as a continuous memory area divided according to the types of the variables. And each node has its own instance of each variable. The definition of the shared variable is as follows.

DISTRIBUTED type *vName* [policy parameters]

As already mentioned an instance of a variable is bound to the owner, and thus, to the locality. In order to distinguish between the values from different owners an instance of a variable is labelled with the identity of its owner or locality, e.g., the address or coordinates of the owner node. Thus, the term *TEMPERATURE@node0* defines the source of the variable, i.e., its spatial address dimension. Additionally, to distinguish the variables in the temporal dimension, each occurrence of an instance (or simply a value) is marked by a time stamp or version number. The options for spatial and temporal addressing reflect the capabilities and requirements of the application that uses our middleware.

In order to configure the specific parameters, and thus, the behaviour of the system the application developer specifies the values for the parameters for each variable. The complete set of parameters specifies the policy for that variable. The policy can be also chosen from a predefined policy file, making the development easier. Additionally, exchanging the policy file allows creating several versions

of an applications with different reliability requirements from the same application code.

Our middleware provides additional features that increase its applicability. Next paragraphs briefly describe them, but these features are not evaluated in this paper.

Every node is able to write any shared data item and in order to do it, the node sends the write request to the owner. The owner performs a local write operation and sends the update to all replicators again. This solution has been chosen because of the spatial focus of the replication, i.e., the write request may come from any node in the network, even a very distant one, but since the replication area is the vicinity of the owner, this is the best node to initiate the update of the replicates. But maybe even more important reason is to keep the right order of the write operations. This choice may cause the owner to become a bottleneck, but we claim that the complexity of the system would suffer from extending the writing operation to allow all copies to be writeable.

The event mechanism in tinyDSM monitors the chosen variables. Basically, each event a shared data item as well and its value is defined by a logic equation with terms based on other shared data items. In case the logic equation results in logic true, the middleware notifies the application about that—it fires an event. Depending on the functionality provided by the handler function it may be treated as local or global notification. The local notification may adjust some local parameter of the on-node part of the application, e.g., the sampling rate of an environment phenomenon. The second type of events handling leads to sending a predefined message, e.g., to a sink. The definition of an event is as follows.

EVENT eName *IF* condition *TRIGGER* handler() [policy params]

If necessary for the application the middleware may provide support for database-like queries. The query management allows processing complex database-like queries about current and historical data issued either by a node within the network or from outside—by the user. This feature is, however, bound to high processing costs.

The architecture of the tinyDSM middleware consists of the following modules (see Figure 2):

- *Application Logic* controls the behaviour of the nodes that build up the global application; it defines the sources of data and behaviour in case of events.
- *Event & Replication Logic* is responsible for detecting the events for the incoming data. It also takes the decisions on the replication and storage of new data and controls data locating on reading.
- *Query Logic* is responsible for interpreting incoming messages (queries or requests) and building results into answer messages. It allows the use of complex database-like queries issued by the user and read or write requests from other nodes.
- *Policies* are a virtual module that controls the behaviour of the system. Exchanging the policy file at compile time allows to create several versions of the application that fulfil different requirements using the same source code.
- *Memory Manager* controls the physical data storage on the node. Provides the logical data system in the physical data storage and operations to on it.
- *Communication Interface* controls the communication with other nodes. It hides the mapping between different kinds of messages and protocols.
- *OS Adaptation Layer* is a layer that allows running the same base skeleton implementation on different operating systems. It provides the drivers for the services provided by the OS, as well as, means to translate the application requests into tinyDSM native ones.

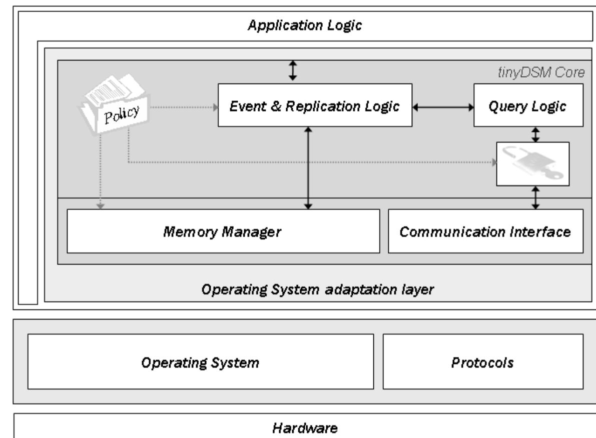

Figure 2. The Architecture of the tinyDSM Middleware

The tinyDSM skeleton is implemented in pure C programming language and the OS adaptation layer currently supports the tinyOS operating system. In this implementation we wanted to map the concept architecture as presented in Figure 2 as close as possible, but for optimization reason some modules were merged. Table 1 presents the memory footprint of the selected tinyDSM modules.

Figure 3 shows the interfaces our middleware provides to the application running on top of it, distinguishing between the part of application that resides on each node and the application as a whole, represented by the network of nodes. The part of the application that resides on the node can use the *DATA* and the *EVENT* interfaces.

Table 1. Memory Footprints for Selected tinyDSM Modules

	TmoteSky	
Software item	RAM [kB]	ROM [kB]
TinyDSMCore	0.3	5.4
MemoryManager	0.4	7.1
CommunicationInterface	0.3	0.2
	MicaZ	
Software item	RAM [kB]	ROM [kB]
TinyDSMCore	0.3	5.4
MemoryManager	0.25	9.7
CommunicationInterface	0.25	0.3

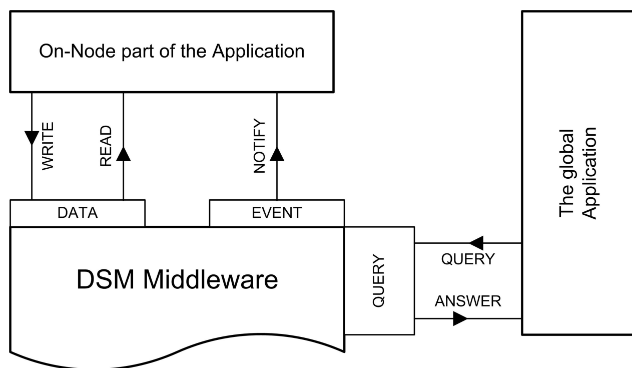

Figure 3. The Interfaces Provided by the Middleware

The *DATA* interface provides access to the shared data, allowing reading and writing. Using the *EVENT* interface the application is notified about occurrence of defined situations.

For the global application the *QUERY* interface is provided. Using this interface one can use the distributed shared memory to answer more complicated queries based on its content.

Thus, from the application perspective the middleware provides the following services.

- WRITING a shared data,
- READING a shared data,
- NOTIFICATION in case a defined state of the data is reached,
- answering complex QUERIES based on the memory content.

4. THE EVALUATION

We simulated our approach using both, the Avrora [9] sensor network simulator/emulator and the tinyOS simulator TOSSIM [12]. The simulation results achieved with the Avrora simulator were in some cases erroneous, because the radio modules of the nodes tend to block for very large or dense networks. Thus, here we present the results for TOSSIM simulation for three scenarios with the following parameters. In any case the network was a 20 x 20 nodes mesh, with constant distance between each column and row. To diversify the density of the network we used either the single or double radio range, meaning that the nodes were able to reach the nodes one or two rows/columns away, respectively. These two parameters describe the network in its physical sense and they were combined with the policy parameters as presented in Table 2. In all scenarios we defined one variable of the type byte and each node sets its instance every 5 seconds causing an update. The number of replicas to reach was set depending on the amount of nodes in the locality defined by the replication range. The initial probability of replication was constant for all scenarios and was equal to 25%. The quality parameters for the data storage was defined such that a sequence of ten update operations without acknowledgements is followed by a verification/acquisition phase that can be retried up to five times in case of failure. As already mentioned, the acquisition phase uses an adaptation mechanism that causes new nodes to join the group of replicators. Thus, in this case if the number of acknowledgements/replicas in the last update operation was too low. The policy parameters mentioned here are only a small share of the complete set.

Table 2. Network Parameters Used in the Simulations

Network name	Replication range [hops]	Radio range	Requested replicas
Network 1	1	1	2
Network 2	1	2	6
Network 3	2	1	6

We use the default Medium Access Control (MAC) mechanism used by the ActiveMessage in TinyOS. This causes that in case of simultaneous acknowledgement transmissions to one node they are not protected by any means against the Hidden Terminal effect. Thus, to increase the success rate we added a random delay before every transmission. The presented results are for simulation runs that were limited to 1800 seconds PC running time.

Figures 4, 5 and 6 present the observations of the replication process for our simulation scenarios. On each chart the x-axis presents the sequence numbers of the update operations and the y-axis the number of replicas or the number of received acknowledgements. The number of acknowledgements is here only counted until it reaches the requested number of fresh replicas. The charts show the process of distribution/replication of one randomly chosen instance of the defined variable.

In the Network 1 scenario the requested number of replicas is easily reached, and actually the number of replicas in the network is two times bigger than requested and every verification phase finishes after getting just the two required acknowledgements. Except the initial period, there are no acquisition phases. However, here we can see that

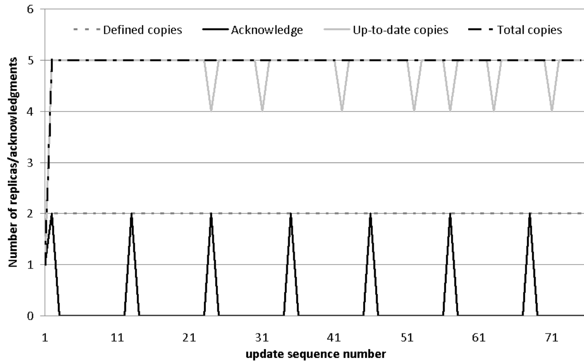

Figure 4. The Replication Process for a Chosen Node in the Network 1 Simulation Scenario

the relatively high replication probability of 25% caused the number of replicas to grow already at the beginning of the simulation exceeding the requested number more than double, probably wasting storage space.

In the Network 2 the density of nodes is increased compared to Network 1, but still, the single hop replication range causes this scenario to be straight forward as well. The number of replicas in the system is exact at the requested level for several update cycles. Then, just before the first verification phase, probably because of an update message collision, the number of fresh replicas drops. This situation was detected in the verification phase and the acquisition phase was triggered causing the increase of replicas. After that adaptation this simulation does not show any misbehaviour. The number of replicas in the system is 50% larger than the requested one causing only little overhead.

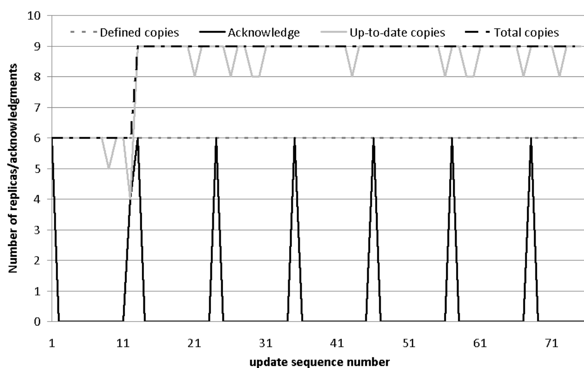

Figure 5. The Replication Process for a Chosen Node in the Network 2 Simulation Scenario

The Network 3 is a scenario with multi-hop replication range. Nevertheless, the single radio range reduces the traffic problems and the simulation does not show any misbehaviour and the requirements are satisfied. But again, there is a need to be careful with the replication probability. If its value is to big, the number of replicas may grow rapidly in case of a single verification failure. Here, the number of requested replicas was again exceeded more than twice because in the third verification phase the success condition was not satisfied. In the multi-hop version of the verification phase, the acknowledgements forwarded by the nodes closer to the owner are aggregated, so their number may exceed the requested number of replicas.

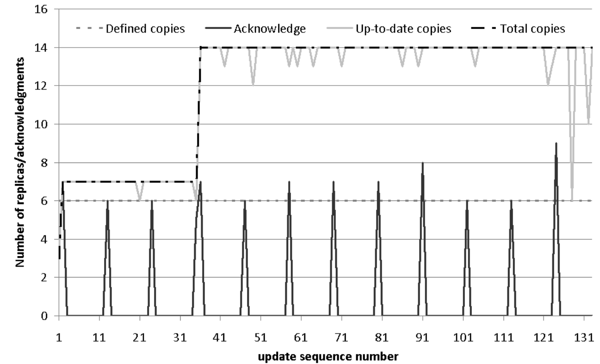

Figure 6. The Replication Process for a Chosen Node in the Network 3 Simulation Scenario

Figures 4, 5 and 6 show that the concept is feasible, but there is still room for improvement. On one hand, starting with higher value of the replication probability reduces the number of verification/advertisement phases but may cause the network to be filled with too many replicas very quick. Anyway, smart changes of the replication probability, especially if additionally depending on the current distance to the owner node may help to make the distribution of the replicas more equal. If the total number of replicas is higher than the number of fresh ones there may be a need for replica freshness mark that automatically invalidates old replicas, either based on time or on the knowledge that a new value of the variable was already replicated. Additionally, reducing the network traffic by specifying conditions that trigger an update either on time or value change basis, helps avoiding collisions and increases the chance that the updates reach the replicators. An RTS-CTS like MAC protocol would be helpful here as well.

Figures 4, 5 and 6 show the quantitative and qualitative characteristics of the replication, but with a little bit more the focus on the first factor. Figure 7, in contrast, focuses on the quality. It shows the average and maximum delays between the setting of the variable by the owner and the last update of its replica. These values can be used to represent the average and the worst case time where the view of

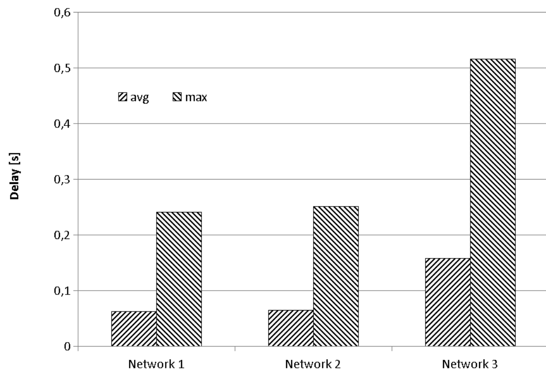

Figure 7. The Average and Maximum Delay of a Replica Update

the replicas is inconsistent, meaning, there may be still an update to come. The numbers here are increased by the random delay added to diversify the sending of the messages to avoid collisions.

Figure 8 shows the network traffic overhead—the average and maximum number of sent messages per update. We simulated the scenarios with the same settings regarding the update and verification/advertisement ratio and it is clear to see that the traffic grows exponentially with the complexity of the network. Here the way to cope with that is again to reduce the update rate by defining the trigger conditions.

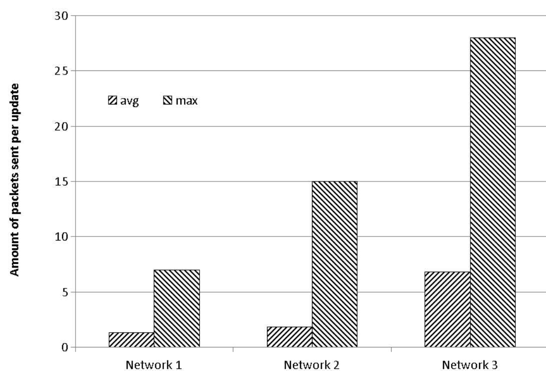

Figure 8. The Average and Maximum Number of Packets Sent per Update

The most important information achieved from the simulation done with the Avrora emulator was the CPU load. We managed to get stable results for a network of 64 nodes (8 x 8 mesh) with single hop replication range and both, single and double radio range, i.e., density parameters similar to the Network 1 and Network 2. The difference, compared to these setups, is that every update is replied with an acknowledgement message. For the double radio range the average CPU load is about 3 percent, but for the radio range of one the CPU load goes down to 1 percent. Here, the most of the load is caused by the processing of incoming packets, and grows with the traffic and the relative density of the network.

These results show that the underlying protocol layer and a proper configuration of replication strategies via policies are very important and influence the performance of the system especially with respect to the generated network overhead. Reducing the update rate reduces the network overhead, but, on the other hand, reduces the coherency level that can be achieved. The use of the acknowledgements to indicate that the replication took place, causes the most part of the network overhead, but, on the other hand, it informs the data owner about the distribution of the replicas, allows strategy adaptation and increases the overall reliability of the proposed solution.

Our results show that our approach is feasible and can be instantiated as a very efficient implementation.

5. THE CONCLUSIONS AND OUTLOOK

In this paper we evaluated the feasibility of our proposal of a reliable shared data storage middleware for WSNs. The solution we proposed combines the functionality of passive data storage, known from approaches like tinyDB [8], cougar [11] or tinyPEDS [6], enhanced by reliability means with an active data monitoring using its event detection mechanism. It defines groups of nodes similar to the concept presented by Hood [10] or Abstract Regions [5]. There are several other approaches that go in the direction of the distributed computation and data storage domain [2], [3], [1] and [4]. A reliable shared data storage with active data monitoring opens the possibility for nodes to cooperate more actively or even autonomic and independent from the central station injecting more intelligence into the network. Additionally, the replication of data with configuration capabilities helps to assure the data availability in case of node failure while providing a consistent view of the replicas. The configuration specifies the quantity and quality of the replication in terms of the area and consistency parameters. These parameters are part of a policy file, that controls the behaviour of the middleware. It provides development flexibility as well, since the policy file can be exchanged, resulting in another instance of the same application, but with different requirements and parameters.

The simulation results we presented indicate that our approach can be used in WSNs without significant negative impact on the node and network lifetime.

In the future we will research the effects of additional adaptation possibilities and improve the replication strate-

gies with respect to processing and networking effort. In order to increase the efficiency of the middleware we want to investigate the protocol layer related optimization and parametrization possibilities. Additionally, we will investigate the impact of the transmission costs to the overall cost of the external read and write requests together with a multi-hop protocol. In addition we are planning to extend the tinyDSM architecture with suitable security features and use the *UPDATE* messages for time synchronisation and location estimation.

REFERENCES

[1] L. Luo, T. F. Abdelzaher, T. He and J. A. Stankovic, "Envirosuite: An environmentally immersive programming framework for sensor networks," *Trans. on Embedded Computing Sys.*, Volume 5(3), 2006, pp. 543–576.

[2] P. Costa, L. Mottola, A. L. Murphy and G. P. Picco, "Programming Wireless Sensor Networks with the TeenyLIME Middleware," Proc. of the 8th ACM/IFIP/USENIX International Middleware Conference (Middleware 2007), Newport Beach (CA, USA), November 26–30, 2007.

[3] T. F. Abdelzaher, B. M. Blum, Q. Cao, D. Evans, J. George, S. George, T. He, L. Luo, S. H. Son, R. Stoleru, J. A. Stankovic and A. Wood, "EnviroTrack: Towards an Environmental Computing Paradigm for Distributed Sensor Networks," Proc. of the 24th International Conference on Distributed Computing Systems (ICDCS 04), IEEE CS Press, 2004, pp. 582–589.

[4] R. Gummadi, O. Gnawali, and R. Govindan, "Macro-programming Wireless Sensor Networks Using Kairos," Proc. of the International Conference on Distributed Computing in Sensor Systems (DCOSS 05), LNCS 3560, Springer, 2005, pp. 126–140.

[5] M. Welsh and G. Mainland, "Programming Sensor Networks Using Abstract Regions," Proc. of the 1st Usenix/ACM Symposium on Networked Systems Design and Implementation (NSDI 04), 2004, pp. 29–42.

[6] J. Girao, D. Westhoff, E. Mykletun, and T. Araki, "Tinypeds: Tiny persistent encrypted data storage in asynchronous wireless sensor networks," *Ad Hoc Networks Journal*, Volume 5(7), Sept. 2007, pp. 1073–1089.

[7] J. Hill, P. Levis, S. Madden, A. Woo, J. Polastre, C. Whitehouse, R. Szewczyk, C. Sharp, D. Gay, M. Welsh, D. Culler, and E. Brewer, TinyOS: http://www.tinyos.net, March 2009.

[8] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: an acquisitional query processing system for sensor networks," *ACM Trans. Database Syst.*, Volume 30(1), 2005, pp. 122–173.

[9] B. L. Titzer, D. K. Lee, and J. Palsberg, "Avrora: scalable sensor network simulation with precise timing," Proc. of the 4th international symposium on Information processing in sensor networks (IPSN 05), Piscataway, NJ, USA, 2005, page 67.

[10] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler, "Hood: a neighborhood abstraction for sensor networks," Proc. of the 2nd international conference on Mobile systems, applications, and services (MobiSys 04), New York, NY, USA, 2004, pp. 99–110.

[11] Y. Yao and J. E. Gehrke, "The cougar approach to in-network query processing in sensor networks," *ACM SIGMOD Record*, Volume 31(2), Sept. 2002, pp. 9–18.

[12] P. Lewis and N. Lee and M. Welsh, and D. Culler, "TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications," Proc. of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 03), 2003, pp. 126–137.