TRANSACT: A Transactional Framework for Programming Wireless Sensor/Actor Networks

Murat Demirbas, Onur Soysal, Muzammil Hussain
{demirbas — osoysal — mh69}@cse.buffalo.edu
Department of Computer Science & Engineering
University at Buffalo, SUNY

Abstract

Effectively managing concurrent execution is one of the biggest challenges for future wireless sensor/actor networks (WSANs): For safety reasons concurrency needs to be tamed to prevent unintentional nondeterministic executions, on the other hand, for real-time guarantees concurrency needs to be boosted to achieve timeliness. We propose a transactional, optimistic concurrency control framework for WSANs that enables understanding of a system execution as a single thread of control, while permitting the deployment of actual execution over multiple threads distributed on several nodes. By exploiting the atomicity and broadcast properties of singlehop wireless communication, we provide a lightweight implementation of our transactional framework on the motes platform.

1 Introduction

Traditionally wireless sensor networks (WSNs) act mostly as data collection and aggregation networks and do not possess a significant actuation capability [3, 35]. However, as WSNs become increasingly more integrated with actuation capabilities, they have the potential to play a major role in our lives fulfilling the proactive computing vision [34]. Future wireless sensor/actor networks (WSANs) will be instrumental in process control systems (such as vibration control of the assembly line platforms or coordination of regulatory valves), multi-robot coordination applications (such as robotic highway construction markers [10], where robot-cones move in unison to mark the highway for the safety of workers), and in resource/task allocation in multimedia WSNs (such as video-based coordinated surveillance/tracking of suspected individuals in an urban setting).

WSANs need a radically different software than WSNs do. In contrast to WSNs, where a best-effort (eventual consistency, loose synchrony) approach is sufficient for most applications and services, consistency and coordination are essential requirements for WSANs because in many WSAN applications the nodes need to consistently take a coordinated course of action to prevent a malfunction. For example, in the factory automation scenario inconsistent operation of regulator valves may lead to chemical hazards, in the robotic highway markers example a robot with an inconsistent view of the system may enter in to traffic and cause an accident, and in the video tracking scenario failure to coordinate the handoff consistently may lead to losing track of the target.

Due to the heavy emphasis WSANs lay on consistency and coordination, we believe that concurrent execution, or more accurately, nondeterministic execution due to concurrency will be a major hurdle in programming of distributed WSANs. Since each node can concurrently change its state in distributed WSANs, unpredictable and hard-to-reproduce bugs may occur frequently. Even though it is possible to prevent these unintentional and unwanted nondeterministic executions by tightly controlling interactions between nodes and access to the shared resources [8, 14, 18], if done inappropriately, this may deteriorate a distributed system into a centralized one and destroy concurrency, which is necessary for providing real-time guarantees for the system.

To enable ease of programming and reasoning in WSANs and yet allow concurrent execution, we propose a transactional programming abstraction and framework, namely TRANSACT*: TRANsactional framework for Sensor/ACTor networks.* TRANSACT enables reasoning about the properties of a distributed WSAN execution as interleaving of single transactions from its constituent nodes, whereas, in reality, the transactions at each of the nodes are running concurrently. Consequently, under the TRANSACT framework, any property proven for the single threaded coarse-grain executions of the system is a property of the concurrent fine-grain executions of the system. (This concept is known as "conflict serializability" [12] in databases and as "atomicity refinement" [5, 27] in distributed systems.) Hence, TRANSACT eliminates unintentional nondeterministic executions and achieves simplicity in reasoning while retaining the concurrency of executions.

TRANSACT is novel in that it provides an efficient and lightweight implementation of a transactional framework in WSANs. Implementing transactions in distributed WSANs domain diverges from that in the database context significantly, and introduces new challenges. In contrast to database systems, in distributed WSANs there is no central database repository or an arbiter; the control and sensor variables, on which the transactions operate, are maintained distributedly over several nodes. As such, it is infeasible to impose control over scheduling of transactions at different nodes, and also challenging to evaluate whether distributed transactions are conflicting. On the other hand, we observe that singlehop wireless broadcast has many useful features for facilitating distributed transaction processing. Firstly, broadcasting is atomic (i.e., for all the recipients of a broadcast, the reception occurs simultaneously), which is useful for synchronizing the nodes in singlehop for building a structured transaction operation. Secondly, broadcasting allows snooping of messages without extra overhead, which is useful for conflict detection in a decentralized manner. By exploiting the atomicity and broadcast properties of singlehop wireless communication in WSANs, TRANSACT overcomes this challenge and provides a lightweight implementation of transaction processing. Since imposing locks on variables and nodes may impede the performance of the distributed WSAN critically, TRANSACT implements an optimistic concurrency control solution [21]. Thus, the transactions in the TRANSACT framework is free of deadlocks (as none of the operations is blocking) and livelocks (as at least one of the transactions needs to succeed in order to cancel other conflicting transactions).

TRANSACT enables ease of programming for WSANs by introducing a novel *consistent write-all* paradigm that enables a node to update the state of its neighbors in a *consistent* and *simultaneous* manner. Building blocks for process control and coordination applications (such as, leader election, mutual exclusion, cluster construction, recovery actions, resource/task allocation, and consensus) are easy to denote using TRANSACT (see Figure 3). In this paper we use the resource/task allocation problem as a running example in our analysis, implementation, and simulation sections. This problem is inherent in most WSANs applications, including the process control, multi-robot coordination, and distributed video-based tracking applications we discussed above. We primarily focus on singlehop coordination applications in this paper—albeit, in a multihop network setting. We discuss how to leverage on the singlehop transactions in TRANSACT to provide support for constructing multihop coordination applications in Section 2.4.

Outline of the paper. We present the TRANSACT framework in Section 2. In Section 3 we investigate conflict-serializability of TRANSACT and also analyze the frequency of having conflicting transactions among a set of concurrent transactions. In Section 4, using Tmotes and TinyOS, we give an implementation of the TRANSACT framework for solving the resource/task allocation problem. In Section 5 we present simulation results, using Prowler [33], over a multihop network for the resource/task allocation problem. Finally, we discuss related work in Section 6, and conclude in Section 7.

2 TRANSACT Framework

Overview. In TRANSACT an execution of a nonlocal method is in the form of a transaction. A nonlocal method (which requires inter-process communication) is structured as $read[write\text{–}all]$, i.e., read operation followed, optionally, by a write-all operation. Read operation corresponds to reading variables from some nodes in singlehop, and write-all operation corresponds to writing to variables of a set of nodes in singlehop. Read operations are always compatible with each other: since reads do not change the state, it is allowable to swap the order of reads across different transactions. A write-all operation may fail to complete when a conflict with another transaction is reported. A conflict is possible if two overlapping transactions have pairwise dependencies. We achieve a distributed and local conflict detection and serializability by exploiting the atomicity and snooping properties of wireless broadcast communication. If there are no conflicts, write-all succeeds by updating the state of the nodes involved in a consistent and simultaneous manner. When a write-all operation fails, the transaction aborts without any side-effects: Since the write-all operation—the only operation that changes the state—is placed at the end of the transaction, if it fails no state is changed and hence there is no need for rollback recovery at any node. An aborted transaction can be retried later by the caller application.

As outlined above, the key idea of concurrency control in TRANSACT can be traced to the optimistic concurrency control (OCC) in database systems [21]. TRANSACT exploits the atomicity and broadcast properties of singlehop wireless communication to give an efficient decentralized implementation of OCC. Conflict detection and reporting mechanism is decentralized in TRANSACT. Moreover, in TRANSACT the commit for the write-all operation is time-triggered to ensure that the write-all operation (if successful) is committed simultaneously at all the nodes involved in the transaction. The time-triggered commit mechanism leverages on the atomicity of the write-all broadcast and achieves the commit to occur simultaneously at all the nodes despite the lossy nature of the communication channel. Finally, while OCC insists on transactions to be order-preserving, TRANS-

ACT requires only conflict-serializability and hence allows more concurrency. We discuss this in more detail in Section 6.

2.1 Read and Write-all operations

Singlehop wireless broadcast communication provides novel properties for optimizing the implementation of distributed transactions :
1. *A broadcast is received by the recipients simultaneously*
2. *Broadcast allows snooping*.

Property 1 follows from the characteristics of wireless communication: the receivers synchronize with the transmission of the transmitter radio and the latency in reception is negligible (limited only by the propagation speed of light). As such Property 1 gives us a powerful low-level atomic primitive upon which we build the transactions. Using Property 1, it is possible to order one transaction ahead of another [1], so that the latter is aborted in case of a conflict. Using Property 1, we can define a transaction as a composition of an atomic read operation followed by an atomic write operation, as $T_j = (R_j, W_j)$. Atomicity of read operation is satisfied by the atomicity of broadcast. Each node involved in a read operation prepares its reply at the reception of the read broadcast. Atomicity of the write operation is satisfied by a time-triggered commit taking the write-all broadcast as a reference point.

We use Property 2, i.e., snooping, for detecting conflicts between transactions without the help of an arbiter as we discuss below.

Implementation of Read operation : Since read operations are compatible with other read operations, it is possible to execute read operations concurrently. Moreover, exploiting the broadcast nature of communication the node initiating the transaction can broadcast a read-request where all variables to be read are listed.

Implementation of Write-all operation : The write-all broadcast performs a tentative write (a write to a sandbox) at each receiver. Each receiver replies back with a small *acknowledgment* message. If after the broadcast, the writer receives a *conflict-detected* message (we discuss how below), the write-all operation fails, and the writer notifies all the nodes involved in the write-all to cancel committing. This is done by a broadcasting of a *cancellation* message, and the writer expects a *cancel-ack* from each node to avoid an inconsistency due to loss of a cancellation message. The cancellation process may be repeated a few times until the writer gets a cancel-ack from each node involved in the write-all (the above scheme can be used for avoiding collision of cancel-acks). The commit is time-triggered: If after the write-all, the writer node does not cancel the commit, the write-all is finalized when the countdown timer expires at the nodes. Since write-all is received simultaneously by all nodes, it is finalized at the same time at all the nodes –if it completes successfully.

[1]Property 1 does not rule away collisions nor asserts that a broadcast message should be reliably received by all the intended nodes; it just asserts that for all the nodes that receive the message, the reception occurs simultaneously. We relegate the discussion of how we cope with message losses and collisions to Section 2.3.

Detecting conflicts : The read operations are compatible with respect to each other, so swapping the order of any two concurrent read operations results into an equivalent computation. A read operation and a write operation at different and overlapping transactions to the same variable are incompatible, so it is disallowed to swap the order of two such operations. In such a case, a dependency is introduced from the first to the second transaction. Similarly, two write operations to the same variable are incompatible with each other. For example in Figure 1 if a read-write incompatibility introduces a dependency from $t1$ to $t2$, and a write-write incompatibility introduces a dependency from $t2$ to $t1$, then we say that $t1$ and $t2$ are conflicting. This is because, due to the dependencies the concurrent execution of $t1$ and $t2$ do not return the same result as neither a $t1$ followed by $t2$ nor a $t2$ followed by $t1$ execution. In this case, since $t2$ is the first transaction to complete, when $t1$ tries to write, $t1$ is aborted due to the conflict.

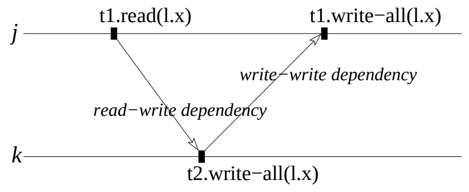

Figure 1. Conflicting transactions

Formally, we denote a transaction T_j by a tuple (R_j, W_j) where R_j is the read-set of T_j and W_j is the write-set for T_j. For any two transactions T_j and T_k, we define the following dependencies:

- $D_{rw}(T_j, T_k) \equiv R_j \cap W_k \neq \emptyset$ and executions of T_j and T_k overlap,
- $D_{ww}(T_j, T_k) \equiv W_j \cap W_k \neq \emptyset$ and write-all broadcast of T_j precedes that of T_k.

We say that there is a conflict between T_j and T_k iff :

$$\begin{array}{rl} & D_{rw}(T_j, T_k) \wedge \quad D_{rw}(T_k, T_j) \\ \vee & D_{rw}(T_j, T_k) \wedge \quad D_{ww}(T_k, T_j) \end{array}$$

That is, T_j and T_k conflict with each other if there is a pairwise read-write dependency between T_j and T_k, or there is a read-write dependency from T_j to T_k and a write-write dependency from T_k to T_j. When a conflict is detected between T_j and T_k, the transaction whose write-all post-dates the other is informed about this conflict via a *conflict-detected* message, and is aborted.

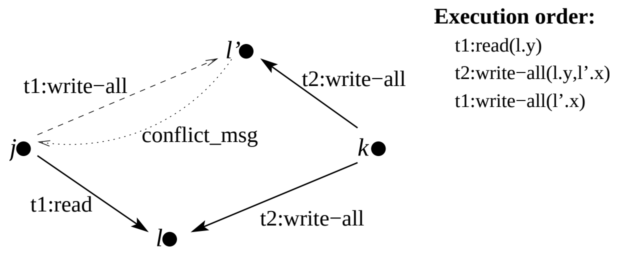

Figure 2. Snooping for detecting conflicts

To enable low-cost detection of conflicts, we exploit snooping over broadcast messages. Figure 2 demonstrates this technique. Here j is executing transaction $t1$ which consists of $read(l.y)$;$write{-}all(l'.x)$ operations that operate on its 1-hop neighbors, l and l'. Simultaneously, another node k within 2-hops of j is executing transaction $t2$ which $write{-}all(l.y, l'.x)$. In this scenario l' is the key. When $t1$ reads l, l' learns about the pending $t1$ transaction via snooping. When $t2$ writes to l', l' takes note of the simultaneous write to $l.y$ (since that information appears at the write-all message) and notices the read-write dependency between $t1$ and $t2$. Later, when $t1$ writes tentatively to $l'.x$, l' notices the write-write dependency between $t2$ and $t1$. Thus, l' complains and aborts $t1$. If there are multiple nodes written by $t1$, the affected nodes may schedule transmission of the conflict-messages in a collision-free manner by taking the write-all broadcast as a reference point.

For some scenarios, dependency chains of length greater than two are also possible. Thus, we also enforce acyclicity for such dependency chains via aborting a transaction if necessary. An example of a dependency chain of length three with a cycle is: $t1$:(read y, write-all x), $t1$:(read z, write-all y), and $t3$:(read x, write-all z). Catching such cycles among transactions in singlehop is achieved by a straightforward modification to the conflict detection rule we described above. With the modification, the snoopers search for any potentially long dependencies in their snooping table in order to detect conflicts as we discuss in Section 3.1.

2.2 TRANSACT examples

In Figure 3, we give some examples of TRANSACT methods for different tasks to illustrate the ease of programming in this model. Each method is written as if it will be executed in isolation as the only thread in the system, so it is straightforward to describe the behavior intended. For example, in the leader election method, an initiator j reads the leader variables of all its singlehop neighbors, and on

```
bool leader_election(){
  $X$= read("*.leader"); //read from all nbrs
  if ($X = \{\perp\}$) then {
    return write-all("*.leader="+ID); }
  return FAILURE; }

bool consensus(){
  VoteSet= read("*.vote");
  if($|VoteSet| = 1$) then //act consistently
    return write-all("*.decided=TRUE");
  return FAILURE;}

bool resource_allocation(candidateSet) {
  $X$= read("$\forall x : x \in candidateSet : x$.allocated");
  $X'$= select a subset of $\{x | x.allocated = \perp \wedge x \in X\}$
  if($X' \neq \emptyset$) then
    return write-all("$\forall x : x \in X' : x$.allocated="+ID);
  return FAILURE;}
```

Figure 3. Sample methods in TRANSACT

finding that none of them has set a leader for themselves, announces its leadership and sets their leader variables to point to j. During concurrent execution another initiator k may be executing in parallel to j, and isolation assumption fails. However, since either j or k performs the write-all before the other, TRANSACT aborts the other transaction re-satisfying isolation assumption for these conflicting transactions through conflict-serializability. E.g., if j performed write-all earlier than k, k's write-all will trigger conflict detections (read-write dependency from k to j, followed by a write-write dependency from j to k) and cancellation of k's transaction.

Similarly for the consensus method, the initiator– assuming isolation– reads vote variables of the neighbors, and on finding an agreement on the same vote, sets the decided variable of all neighbors so that the vote is finalized. If due to concurrent execution a node k changes its vote during a consensus method execution of an initiator j, then j's write-all will lead to a conflict-report from k and abortion of j's transaction.

Finally, the resource allocation method is similar to the leader election. The initiator reads availability of nodes in the candidateSet, and selects a subset of the available nodes, and recruits them for its task. Again TRANSACT ensures that concurrent execution of this method at several initiators do not lead to any data race conditions and inconsistencies.

TRANSACT methods return a boolean value denoting the successful completion of the method. If the method execution is aborted due to conflicts with other transactions or message losses, it is the responsibility of the caller application to retry.

2.3 Fault-tolerance

Even when singlehop neighbors are chosen conservatively to ensure reliable communication (we may consider an underlying neighbor-discovery service to this end—one that may potentially be implemented as a TRANSACT method), unreliability in broadcast communication is still possible due to message collisions and interference. Here, we describe how TRANSACT tolerates unreliability in wireless communication via utilizing explicit acknowledgments and eventually-reliable unicast.

Occasional loss of a read-request message or a reply to a read-request message is detected by the initiator when it times-out waiting for a reply from one of the nodes. Then, the initiator aborts the transaction before a write-all is attempted. In this case, since the initiator never attempted the write-all, no cancellation messages are needed upon aborting. Retrying the method later, after a random backoff, is less likely to be susceptible to message collisions due to similar reasons as in CSMA with collision avoidance approaches [1].

Similarly, loss of a write-all message is detected by the initiator node when it times-out on an acknowledgment from one of the nodes included in the write-all. The initiator then aborts its transaction by broadcasting a cancellation message as discussed above in the context of conflict-resolution.

For the loss of a conflict-detected or cancellation message we depend on the eventual reliability of unicast messages. Upon detection of a loss via timeout on an acknowledgement, if a conflict-detected or cancellation message is repeated a number of times, it should be delivered successfully to the intended recipient. It follows from the impossibility of solving the "coordinated attack problem" [2, 11] in the presence of arbitrarily unreliable communication that the above assumption is necessary even for solving a most basic consensus problem in a distributed system. We argue that such an eventually-reliable unicast assumption is realistic under reasonable network loads as the MAC protocols [31, 38] can resolve collisions via carrier-sense and back-offs. Our implementation and simulation results also validate this assumption.

2.4 Discussion

Limitations. Due to the unreliable nature of wireless communication, a streak of message losses may lead to an inconsistency in TRANSACT. TRANSACT relies on acknowledgments to ensure delivery of write and cancel messages. For conflict detection messages multiple snooper nodes are expected to help. Nevertheless, even with multiple repetitions, delivery of these messages to some involved nodes can fail, leading to inconsistencies. Similarly, a node failure that occurs during an active transaction can cause an inconsistency through inducing persistent message loss. For instance, failure of the initiator node after it broadcasts a write-all may lead to an inconsistent commit.

When transactions involved in a dependency chain are dispersed through a multihop region, it becomes difficult to detect potential cycles. We note that the likelihood of cycles over long dependency chains encompassing multiple hop neighborhoods are quite low due to the short execution duration of our transactions. An effective detection algorithm for multihop dependency chains requires network-wide queries which would be extremely costly. In our current work we are investigating the frequency of such chains and possible remedies.

Multihop extensions to TRANSACT. It is easy to leverage on TRANSACT's singlehop transactions to provide support for constructing multihop programs. To this end, our strategy is to use TRANSACT to implement higher-level coordination abstractions, such as Linda [4] and virtual node architecture [9].

In Linda, coordination among nodes is achieved through invocation of in/out operations using which tuples can be added to or retrieved from a tuplespace shared among nodes [4, 7, 29], however, maintaining the reliability and consistency of this shared tuplespace to the face of concurrent execution of in and out operations at different nodes is a very challenging task. Through its serializable singlehop transaction abstraction, TRANSACT can achieve consistent implementation and maintenance of the tuplespace.

Virtual node architecture [9] is another high-level programming abstraction for distributed nodes. It provides an overlay network of fixed virtual nodes (VNs) on top of a mobile ad hoc network to abstract away the challenges of unpredictable reliability and mobility of the nodes. Each VN is emulated by the physical nodes residing within a singlehop of the VN's region at a given time. The network of VNs serve as a fixed backbone infrastructure for the mobile ad hoc network and allows existing routing and tracking algorithms for static networks to be adopted for these highly dynamic environments. Existing VN layer proposals assume reliable communication channels and use a round-robin approach to achieve consistent replication of the state of the VN over the physical nodes [9]. Our TRANSACT framework provides a lightweight singlehop transaction abstraction for implementing VNs consistently over realistic communication channels.

3 Analytical Results

3.1 Transaction Serialization

A set of transactions are serializable if and only if their dependency graph is acyclic [12]. In the TRANSACT framework, depending on the arrival order of read and write operations, incompatibilities create dependencies. In this section we outline our approach for identifying these dependencies in order to maintain serializability.

Consider two transactions $T_i = (R_i, W_i)$ and $T_j = (R_j, W_j)$. Note that without any incompatibilities, these transaction are always serializable. For investigating incompatibilities, without loss of generality we assume R_i comes before R_j. Then, we have the following execution orders for the atomic read and write operations:

- R_i, W_i, R_j, W_j: In this case the dependencies between transactions are irrelevant since T_i completes before T_j and they are not actually concurrent.
- R_i, R_j, W_i, W_j: In this case if there is read-write incompatibility between T_i and T_j, we introduce a dependency from T_i to T_j. If there is read-write incompatibility between T_j and T_i, we insert a depency from T_j to T_i. Finally if there is write-write incompatibility between T_i and T_j, we introduce a dependency from T_i to T_j.
- R_i, R_j, W_j, W_i: This is only slightly different from previous scenario. Read-Write incompatibilities correspond to same dependencies. Write-Write incompatibility, on the other hand, causes a dependency to be inserted from T_j to T_i.

The dependencies between all concurrent transaction pairs are tracked through TRANSACT execution. We construct the dependency graph with nodes as transactions and directed edges to represent dependence relations. No transaction that would cause a cycle in this dependency graph is allowed to commit.

3.2 Concurrency in Transactions

Since it is impossible to model all applications for TRANSACT, we use a simple transaction model to analyze the effect of concurrency in creating data race conditions and conflicts. In our model a transaction reads from a random subset of the TRANSACT variables and writes to a random subset of its read-set. This model could be appropriate for modeling some resource/task allocation problems.

Given n variables involved in two concurrent transactions, we define three cases:

- Independent: The write-sets of these transactions are distinct from the read-sets of the other. Essentially these transactions are totally independent. We denote probability of such transactions with $P_i(n)$.
- Dependent: These transactions have some kind of dependency with each other due to incompatibilities. We denote the probability of these transactions with $P_d(n)$.
- Conflicting: These transactions can not be run in parallel because they have dependency to each other. No serial ordering is possible for these kind of transactions. The probability of these transactions is $P_c(n)$.

In order to calculate these probabilities we first calculate the probability of incompatibilities. The probability of a Read-Write incompatibility ($P_{RW}(n)$) depending on the number of shared variables can be calculated as follows:

$$P_{RW}(n) = \frac{\sum_{j=1}^{n}\sum_{i=j}^{n}\binom{n}{i}\binom{i}{j}\frac{2^n-2^{n-j}}{2^n-1}}{\sum_{j=1}^{n}\sum_{i=j}^{n}\binom{n}{i}\binom{i}{j}}$$

Here we choose a non empty subset (corresponding to first read set $\binom{n}{i}$) and then chose a non empty subset of this read-set (corresponding to the first write-set $\binom{i}{j}$). For this subset of j elements we calculate the intersection probability with another random subset corresponding to the second read-set.

Similarly Write-Write incompatibility can be derived. This time we also need to choose the second write-set so the expression is a bit longer:

$$P_{WW}(n) = \frac{\sum_{j=1}^{n}\sum_{i=j}^{n}\sum_{k=1}^{n}\sum_{l=k}^{n}\binom{n}{i}\binom{i}{j}\binom{n}{l}\binom{l}{k}\frac{\binom{n}{k}-\binom{n-j}{k}}{\binom{n}{k}}}{\sum_{j=1}^{n}\sum_{i=j}^{n}\sum_{k=1}^{n}\sum_{l=k}^{n}\binom{n}{i}\binom{i}{j}\binom{n}{l}\binom{l}{k}}$$

Using these probabilities, $P_i(n)$ can be calculated as follows:

$$P_i = (1 - P_{WW})(1 - P_{RW})^2$$

While calculating P_d we need to consider arrival time of write messages. We assume this is also random with uniform distribution. Thus only 50% of Write-Write incompatibilities cause a conflict with a Read-Write incompatibility:

$$P_d = P_{WW}(1 - P_{RW})^2 + 2(1 - P_{WW})(1 - P_{RW})P_{RW} + P_{WW}(1 - P_{RW})P_{RW}$$

The conflict probability is given by:

$$P_c = P_{WW}P_{RW}^2 + P_{WW}(1 - P_{RW})P_{RW} + (1 - P_{WW})P_{RW}^2$$

Figure 4 summarizes the predictions of this model. With a single resource there will definitely be conflicts and with

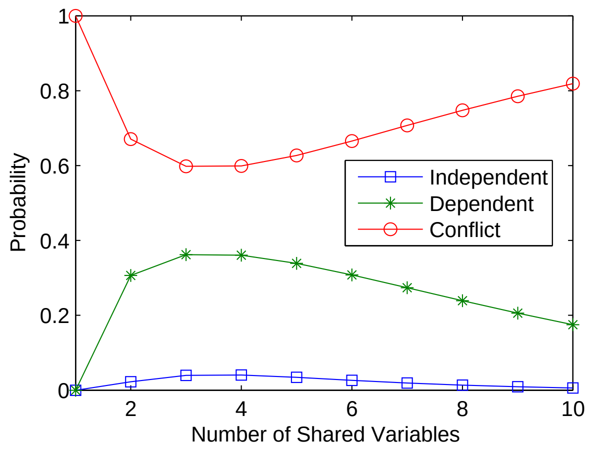

Figure 4. Probabilities of being independent/dependent/conflicting given the number of shared variables between two transactions

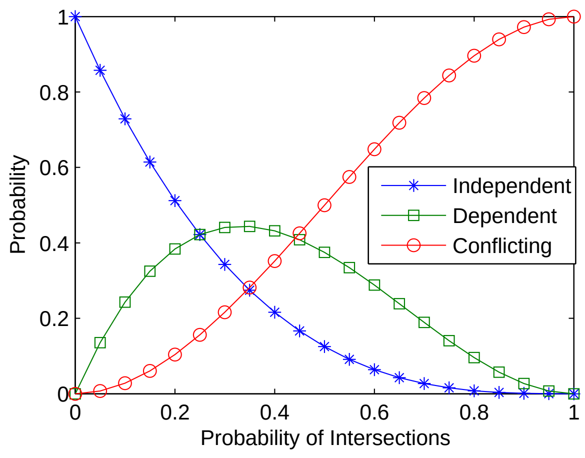

Figure 5. Probabilities of being independent/dependent/conflicting given the probability of sharing an element between read or write sets of two transactions

increasing number of variables we first observe less conflicts (around 3 and 4 variables) and with more variables conflict probability approaches to 1. With this model independent transactions have very low probability and conflicts are common.

Figure 5 on the other hand shows the predicted probability of conflicts and incompatibilities given the probability of intersection. Even with relatively low probabilities having independent transactions has low probability and conflicts are highly probable.

4 Implementation Results

We developed an implementation of TRANSACT over T-mote Invent and T-mote Sky motes [26] in the form of a TinyOS component, called TRANSACT. The TRANSACT component keeps the state of the ongoing transactions and abstracts communication and state maintenance details from the application developer by exporting an interface with split phase semantics [17]. Our TRANSACT implementation is close to 1500 lines of NesC code, and is available at `http://ubicomp.cse.buffalo.edu/transact`.

Test application. In order to test the reliability and feasibility of transactions in our TRANSACT implementation, we also implemented a resource/task allocation application similar to the one we presented in Section 2.2.

In this application, nodes try to obtain control of shared resources for their individual tasks. We call the nodes that try to initiate transactions *initiators* and the nodes that maintain the variables as *resources*. Initially, each initiator is assigned a random subset of the resource nodes to read, and a random subset of their read-sets to write to—in order to allocate those resources. Initiators cannot complete their tasks with partial resources. The application code is aware of the transaction status and the failed transactions are repeated until success is reported by TRANSACT. That is, an initiator keeps retrying until it can allocate the resources it requested.

Experiments. We use a total of 12 motes. One of these motes is reserved for synchronization and book keeping and referred to as the *basestation*. We developed a custom Java application to automate the starting of the experiments and collecting of the results through serial communication with the basestation. Each data point in our graphs is calculated over 50 runs of the corresponding configuration. At the end of each run we check the variables in the resource nodes for correctness. We call a run successful if the resultant values in the resource motes are the correct and consistent values.

We synchronize the initiators through a synchronization message broadcasted from the basestation, and this way start all the transactions at the same instant. Since all nodes are within singlehop, the MAC layer is able to prevent message collisions via carrier-sensing and backoff in relatively low contention configurations, however message losses become common as we increase the number of initiators and resources to stress-test our TRANSACT implementation. We experiment with fairly large number of initially synchronized concurrent transactions to provide a worst-case scenario performance analysis for TRANSACT.

Figure 6 shows the settling times (the time between the first and last message transmitted in a run) using various

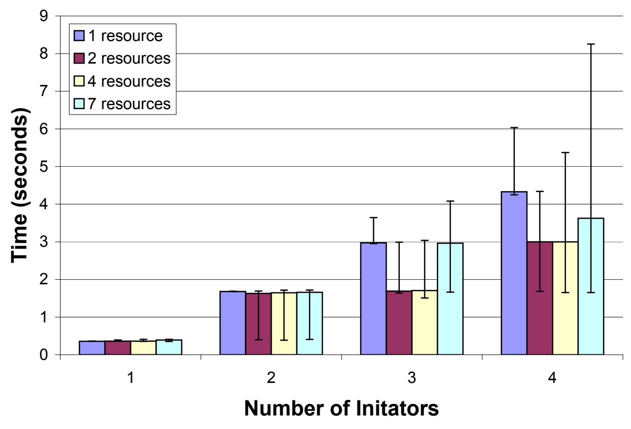

Figure 6. Settling time

configurations of the resource allocation application. In this figure, the bars represent median duration of 50 runs and error bars correspond to 80% confidence interval. An important observation from the figure is the general increase in the settling time with the increasing number of initiators. As the number of concurrent transactions are increased, more conflicts and collisions are reported, leading to aborted and retried transactions, and hence, an increased settling time.

From Figure 6 we observe that increasing the number of resources—while keeping everything else constant— affects the settling time in a manner predicted by our analysis in Section 3.2. We find that having a single resource leads to the worst completion time for the 3 and 4 initiator cases. This result is due to the following. Since no transaction is allowed to have empty read or write sets, when there is a single resource, this causes all the transactions to read from and write to the same resource. As there is no concurrency possible among these conflicting transactions, we observe a performance loss. When using 2 or 4 shared resources, conflicts among initiators are less likely, so the settling time for 2 and 4 resources are less than that with a single resource even though more nodes are involved in a transaction in the 2 and 4 resources case. This result is very consistent with the probability of conflicting transactions presented in Figure 4.

In order to provide more context for the settling time durations of our transactions we like to mention that a message transmission takes around 3msecs on CC2420 radios without any CSMA backoffs. Thus a fast *unreliable* read of a single resource followed by a write operation takes at least 10msecs to complete. Note that this bare-bones best-case time do not allow any parallelism among multiple initiators. In our experiments we fix the transaction durations to a very conservative length throughout all the configurations in order to accommodate concurrent transactions. Also as we have mentioned above, our performance results are meant to be worst-case completion times with fairly large number of initially synchronized concurrent transactions, which leads to several conflicts and collisions.

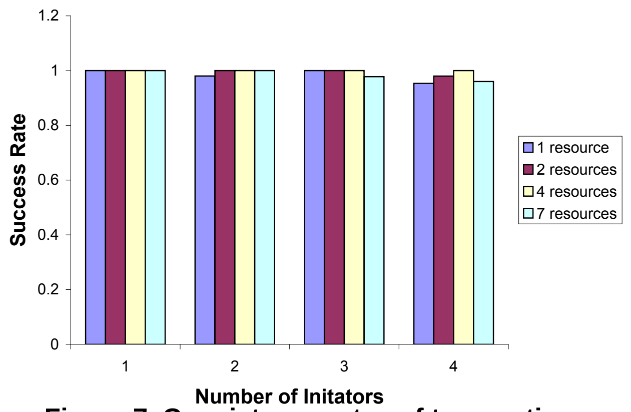

Figure 7. Consistency rates of transactions

Another important parameter we investigate in our experiments is the consistency of the transactions in TRANSACT. We verify the consistency of transactions by querying the resulting resource states after each run via the basestation. Figure 7 shows these results. With a few exceptional cases where a sequence of critical conflict or cancel messages are lost, TRANSACT provides consistency and reliability across different number of initiators and resources.

5 Simulation Results

In order to perform larger-scale experiments, we implemented TRANSACT over the WSN simulator Prowler [33], which simulates the radio transmission/propagation/reception delays of Mica2 motes, including collisions in ad-hoc radio networks, and the operation of the MAC layer. We have modified Prowler to account for the transmission rates of the faster Tmote CC2420 radios (instead of the default Mica2 CC1000 radios), so that our simulation results are closely aligned with our Tmotes implementation results. Our simulation code for TRANSACT is about 1500 lines of distributed/per-node code, and is available at `http://ubicomp.cse.buffalo.edu/transact`.

Our experiments are performed on a 10x10 grid of 100 nodes, where each node has 8 neighbors. Each data point in our graphs is calculated over 50 runs of the corresponding configuration. At the beginning of each run, the initiator nodes are randomly selected to perform a *resource allocation* task, by reading from a random set of their neighbors and then writing to some random subset of their read-sets. Our simulations stress-test TRANSACT by iterating through an increasing number of initiators in the network (from 5 initiators upto 20 initiators denoted along the X-axis). All the initiators start their transactions in the beginning of the run, with only the CSMA mechanism to arbitrate between their messages. An aborted transaction is retried by the initiator.

Protocol	Writes Acks	Consistent Writes	Conflict-Serializabilty
unreliable	×	×	×
ev-reliable	√	×	×
reliable	√	√	×
locking	√	√	√
TRANSACT	√	√	√

Table 1. Transactional protocols we consider

In our simulations, we compare TRANSACT with 4 other transactional protocols: *Reliable, eventually reliable, unreliable, and locking*. The first 3 protocols gradually leave out more mechanisms of TRANSACT and provide lesser guarantees for transaction executions. *Reliable* protocol waives the conflict-detection mechanism in TRANSACT, but may still cancel a transaction if write-acks are not received from all participants. *Ev-reliable* forgoes the transaction cancellation from the *reliable*, and replaces this with re-transmission of the write-all in case of missing write-acks. *Unreliable* waives even the write-ack mechanism of ev-reliable type, and performs a bare-bones write operation. Finally, for the *locking* protocol, we implemented a version of *strict two-phase locking* [12]. In addition to the release of the locks by the initiator upon a commit, we also implemented leases on the locks to prevent any indefinite locking of a resource in case the *release-lock* messages get lost. These five protocols are summarized in Table 1.

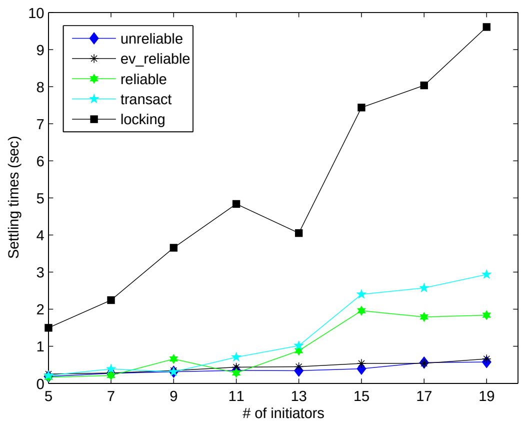

Figure 8. Settling time of various transactional protocols

Figure 8 shows the settling times (the time between the first and last message transmitted in a run) for each protocol. *Unreliable* is naturally the fastest. As the reliability requirements of the transaction protocols increase, we observe a corresponding increase in the settling times. The conflict serializability mechanism of TRANSACT imposes only a little overhead over *reliable*, whereas the overhead associated with the *locking* protocol is huge. This is because TRANSACT allows more concurrency than *locking* as we discuss in Section 3.2. While TRANSACT can execute dependent transactions in parallel (provided that they are not conflicting), locking can execute only independent transactions in parallel. Since Figure 4 shows that the probability of independent transactions are very low for the resource/task allocation application, locaking protocol ends up executing transactions one after the other rather than in parallel. Thus, as the number of initiators increase settling time for locking increases quickly.

In Figure 8, as the contention due to the number of initiators increase, the settling times for all of the protocols are affected. With 20 initiators almost all nodes in the network are involved in transactions, either as participants or as snoopers. Since only the CSMA mechanism arbitrates among these concurrent initiators, message losses due to hidden terminal problems become common occurrences in this multihop setting. We observe that hidden terminal problems start to degrade the performance seriously for the *reliable*, TRANSACT, and *locking* protocols, as we increase the number of initiators in the network.

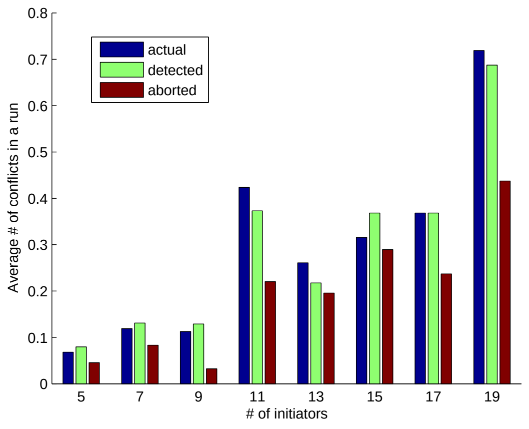

Figure 9. Conflict detection in TRANSACT

Figure 9 demonstrates the effectiveness of TRANSACT in detecting conflicting transactions. In order to construct this graph, we have generated extensive logs for read, write, cancel, snooping operations at the nodes, and later ran a script on the simulation logs from each node to determine the actual number of conflicts, and use this as a reference to compare with the number of conflict detections reported by the snoopers. As seen in the bar graphs, the conflicts detected and aborted by TRANSACT are close to the actual number of conflicts. The difference between the actual and detected number of conflicts is due to loss of messages which drops the effectiveness of snoopers' conflict detection abilities. The difference between the number of conflicts detected and aborted is due to the loss of the conflict-notification and writeall-cancel messages.

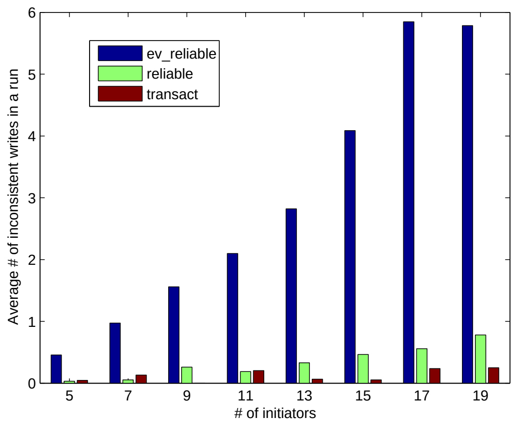

Figure 10. Inconsistent writes

Figure 10 shows the average number of inconsistent writes. Thanks to the cancel mechanism, *reliable* and TRANSACT protocols achieve very few write-inconsistencies compared to *ev-reliable*. Write inconsistency in *ev-reliable* protocol arises due to the loss of write message at some participants. In *reliable* and TRANSACT, a write inconsistency may be only due to the failure to abort a write operation before its commit timer expires.

6 Related Work

Concurrency control in TRANSACT diverges from that in the database context significantly as we discuss in the Introduction. Recently, there has been a lot of work on transaction models for mobile ad hoc networks [6, 23–25, 30, 32], however, these work all assume a centralized database and an arbiter at the server, and try to address the consistency of hidden read-only transactions initiated by mobile clients.

Software transactional memory (STM) [15] is a concurrent programming scheme with multiple threads. In STM conventional critical sections for controlling access to shared memory are replaced by transactions. In TRANSACT, there is no shared memory as the variables are distributed among nodes.

Although TRANSACT is closer to an optimistic concurrency control (OCC) approach than a locking approach, there are significant differences between the semantics of transactions in TRANSACT and that in OCC protocols of database systems. TRANSACT relaxes the order preserving properties of OCC and provides more concurrency. For example, in Figure 5, TRANSACT allows transactions labeled as dependent to be executed concurrently as they still can be ordered in a conflict-free serialization schedule. OCC protocols on the other hand introduce some order among transactions through explicit transaction numbers [21], which prevents approximately half of the dependent transactions in Figure 5 from executing concurrently.

Several programming abstractions have been proposed for sensor networks [13, 28, 36, 37]. Kairos [13] allows a programmer to express global behavior expected of a WSN in a centralized sequential program and provides compile-time and runtime systems for deploying and executing the program on the network. Hood [37] provides an API that facilitates exchanging information among a node and its neighbors. In contrast to these abstractions that provide best-effort semantics (loosely-synchronized, eventually consistent view of system states), TRANSACT focuses on providing a dependable framework for WSANs with well-defined consistency and conflict-serializability guarantees.

A cached sensor transform (CST) that allows simulation of a program written for interleaving semantics in WSNs under concurrent execution is introduced in [16]. CST advocates a push-based communication model: Nodes write to their own local states and broadcast so that neighbors' caches are updated with these values. This is not directly equivalent to writing neighbor's state, due to complications arising from concurrency and not being able to directly hear writes from 2-hop neighbors to a 1-hop neighbor. CST imposes a lot of overhead for updating of a continuous environmental value (e.g., a sensor reading changing with time) due to the cost of broadcasting the value every time it changes. In contrast to the CST model, TRANSACT uses pull-based communication, and hence it is more efficient and suitable for WSANs. CST targets WSN platforms and supports only a loosely-synchronized, eventually-consistent view of system states. TRANSACT is more amenable for control applications in distributed WSANs as it guarantees consistency even in the face of message losses and provides a primitive to write directly and simultaneously to the states of neighboring nodes.

7 Concluding Remarks

We presented TRANSACT, a transactional, optimistic concurrency control framework for WSANs. TRANSACT provides ease of programming and reasoning in WSANs without curbing the concurrency of execution, as it enables reasoning about system execution as a single thread of control while permitting the deployment of actual execution over multiple threads distributed on several nodes. TRANSACT offers a simple and clean abstraction for writing robust singlehop coordination and control programs for WSANs, which can be used as building blocks for constructing multihop coordination and control protocols. We believe that this paradigm facilitates achieving consistency and coordination and may enable development of more efficient control and coordination programs than possible using traditional models.

In future work, we plan to employ TRANSACT for implementing a multiple-pursuer/multiple-evader tracking application over a 200 node WSN, using several iRobot Roomba-Create robots interfaced with the motes [22] as pursuers and evaders. Using TRANSACT, we will implement the consistency critical components of the in-network tracking service, such as evader association and handoff, updating of the distributed tracking directory/structure, and maintenance and recovery of the tracking structure in the face of node failures and displacements. In addition, the pursuer robots will utilize TRANSACT to implement collaborative stalking and cornering of an evader, as well as group membership and intruder assignment among the pursuers.

We also plan to integrate verification support to TRANSACT in order to enable the application developer to check safety and progress properties about her program. Since TRANSACT already provides conflict serializability, the burden on the verifier is significantly reduced. Hence, for verification purposes it is enough to consider a *single-threaded coarse-grain execution* of a system rather than investigating all possible fine-grain executions due to concurrent threads. Another advantage TRANSACT provides is the simplistic format of the methods, which facilitates translation between TRANSACT methods and existing verification toolkits, such as model checkers [19, 20].

8 Acknowledgements

We thank Gustavo Alonso for shepharding our paper.

This work was partially supported by NSF Career award #0747209.

References

[1] Wireless lan medium access control(mac) and physical layer (phy) specification. IEEE Std 802.11, 1999.

[2] E. A. Akkoyunlu, K. Ekanadham, and R. V. Huber. Some constraints and tradeoffs in the design of network communications. *SIGOPS Oper. Syst. Rev.*, 9(5):67–74, 1975.

[3] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, 2002.

[4] N. Carriero and D. Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, 1989.

[5] K. M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley Publishing Company, 1988.

[6] I. Chung, B. K. Bhargava, M. Mahoui, and L. Lilien. Autonomous transaction processing using data dependency in mobile environments. *FTDCS*, pages 138–144, 2003.

[7] P. Costa, L. Mottola, A. Murphy, and G. Picco. Teenylime: transiently shared tuple space middleware for wireless sensor networks. In *MidSens '06: Proceedings of the international workshop on Middleware for sensor networks*, pages 43–48, 2006.

[8] E. W. Dijkstra. Cooperating sequential processes. pages 65–138, 2002.

[9] S. Dolev, S. Gilbert, L. Lahiani, N. Lynch, and T. Nolte. Timed virtual stationary automata for mobile networks. *9th International Conference on Principles of Distributed Systems (OPODIS)*, 2005.

[10] S. Farritor and S. Goddard. Intelligent highway safety markers. *IEEE Intelligent Systems*, 19(6):8–11, 2004.

[11] J. Gray. Notes on data base operating systems. Technical report, IBM, 1978.

[12] J. Gray and A. Reuter. *Transaction Processing : Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[13] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using *kairos*. In *DCOSS*, pages 126–140, 2005.

[14] P. B. Hansen, editor. *The origin of concurrent programming: from semaphores to remote procedure calls*. Springer-Verlag, 2002.

[15] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software transactional memory for dynamic-sized data structures. pages 92–101, Jul 2003.

[16] T. Herman. Models of self-stabilization and sensor networks. *IWDC*, 2003.

[17] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. *ASPLOS*, pages 93–104, 2000.

[18] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[19] G.J. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, 2003.

[20] G.J. Holzmann. Spin and promela online references. `http://spinroot.com/spin/Man/index.html`, November 2004.

[21] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[22] T. E. Kurt. *Hacking Roomba*. John Wiley, 2006.

[23] K.-Y. Lam, M.-W. Au, and E. Chan. Broadcast of consistent data to read-only transactions from mobile clients. In *2nd IEEE Workshop on Mobile Computer Systems and Applications*, 1999.

[24] V. C. S. Lee and K.-W. Lam. Optimistic concurrency control in broadcast environments: Looking forward at the server and backward at the clients. *MDA*, pages 97–106, 1999.

[25] V. C. S. Lee, K.-W. Lam, S. H. Son, and E. Y. M. Chan. On transaction processing with partial validation and timestamp ordering in mobile broadcast environments. *IEEE Trans. Computers*, 51(10):1196–1211, 2002.

[26] Moteiv. `http://www.moteiv.com/`.

[27] M. Nesterenko and A. Arora. Stabilization-preserving atomicity refinement. *13th International Symposium on Distributed Computing (DISC)*, 1999.

[28] R. Newton and M. Welsh. Region streams: functional macroprogramming for sensor networks. In *DMSN '04: Proceeedings of the 1st international workshop on Data management for sensor networks*, pages 78–87, 2004.

[29] G. P. Picco, A. L. Murphy, and G.-C. Roman. Lime: Linda meets mobility. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 368–377, 1999.

[30] E. Pitoura. Supporting read-only transactions in wireless broadcasting. In *9th Int. Workshop on Database and Expert Systems Applications*, page 428, 1998.

[31] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 95–107, 2004.

[32] J. Shanmugasundaram, A. Nithrakashyap, R. Sivasankaran, and K. Ramamritham. Efficient concurrency control for broadcast environments. In *SIGMOD '99*, pages 85–96, 1999.

[33] G. Simon, P. Volgyesi, M. Maroti, and A. Ledeczi. Simulation-based optimization of communication protocols for large-scale wireless sensor networks. *IEEE Aerospace Conference*, pages 255–267, March 2003.

[34] D. Tennenhouse. Proactive computing. *Commun. ACM*, 43(5):43–50, 2000.

[35] M. Tubaishat and S. Madria. Sensor networks : An overview. *IEEE Potentials*, 2003.

[36] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *NSDI*, pages 29–42, 2004.

[37] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *MobiSys*, pages 99–110, 2004.

[38] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient mac protocol for wireless sensor networks. In *INFOCOMM*, pages 1567–1576, 2002.