# A Transactional Framework for Programming Wireless Sensor/Actor Networks

Murat Demirbas
Department of Computer Science & Engineering
University at Buffalo, SUNY
demirbas@cse.buffalo.edu

***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 properties of wireless broadcast communication, we outline a lightweight and fault-tolerant implementation of our transactional framework.**

## I. Introduction

Current wireless sensor networks (WSNs) do not have any significant actuation capability, however, we envision that WSNs will become increasingly more integrated with actuation capabilities. Future wireless sensor/actor networks (WSANs) will play a major role in our lives as they fulfill the proactive computing vision [34]. WSANs will be instrumental in factory automation and process control systems, such as vibration control of the assembly line platforms or coordination of regulatory valves. Another example of WSANs could be robotic highway safety/construction markers [10], where robot cones move in unison to mark the highway for the safety of workers.

With great power comes great responsibility. In contrast to WSNs, where a best-effort (eventual consistency, loose synchrony) approach has been dominant for most applications and services, consistency and coordination will be essential requirements for WSANs, since 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, and in the robotic highway markers example a robot with an inconsistent view of the system may enter in to traffic and cause an accident.

Due to the heavy emphasis WSANs lay on consistency and coordination, we anticipate 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 as it is the case in any distributed system. 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], [15], [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, and yet allow concurrent execution, we propose *TRANSACT: TRANsactional framework for Sensor/ACTor networks*. TRANSACT enables reasoning about the properties of a distributed system 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. (We call this the “conflict serializability” theorem.) Hence, TRANSACT eliminates unintentional nondeterministic executions and achieves simplicity in reasoning while retaining the concurrency of executions.

**Overview of** TRANSACT**.** The key idea of TRANSACT can be traced to the optimistic concurrency control (OCC) in database systems [21]. There are three phases in an OCC transaction: 1. *Read:* Transaction begins by reading values and writing to a private sandbox. 2. *Validation:* The database checks if the transaction could have conflicted with any other concurrent transaction. If so, the transaction is aborted and restarted. 3. *Write:* Otherwise, the transactions commits. Thus, transactions in OCC satisfy the ACID (atomicity, consistency, isolation, durability) properties.

In TRANSACT, a thread, an execution of a nonlocal method, is analogous to a transaction in OCC. A nonlocal method (which requires inter-process communication) is structured as $read*[write\text{-}all]$, i.e., a sequence of read operations followed by a write-all operation. Each read operation reads variables from some nodes in single-hop, and write-all operation writes to variables of a set of nodes in single-hop. Read operations are compatible with each other: since reads do not change the state, it is allowable to swap the order of reads across different threads (and even within the same thread as we discuss later). Similar to a write operation in OCC, a write-all operation may fail to complete when a conflict with another thread is reported. A conflict is possible only if two overlapping threads have both read-write and write-write incompatibilities (as defined in Section II-B) with respect to some variables. When a write-all operation fails, the thread aborts without any side-effects. Since the write-all operation—the only operation that changes the state—is placed at the end of the thread, if it fails no state is changed and hence there is no need for rollback recovery at any node. An aborted thread can be retried later.

**Novel contributions.** Concurrency control in TRANSACT diverges from that in the database context significantly, and introduces new challenges to address as well as new opportunities to exploit for efficient implementations. In contrast to database systems, in distributed WSANs there is no central database repository or an arbiter; the control and sensor variables are maintained distributedly over several nodes. As such, it is infeasible to impose control over scheduling of threads at different nodes, and also challenging to evaluate whether distributed threads are conflicting. By exploiting the properties of broadcast communication inherent in WSANs, TRANSACT overcomes this challenge and provides a light-weight implementation of optimistic concurrency control as we discuss in Section II-C. Moreover, TRANSACT is free of deadlocks (as none of the operations is blocking) and livelocks (as at least one of the threads needs to succeed in order to cancel other incompatible thread executions).

Process control and coordination programs are easy to write in TRANSACT. For example a leader election method is written in two lines in Figure 1. Here, the first statement reads the “leader” variable from all neighbors, and if none of the neighbors has a valid leader, the second statement assigns the node as the leader to all neighbors. Similarly, mutual exclusion, cluster construction, neighborhood discovery, recovery actions, and consensus are easy to denote using TRANSACT. Since TRANSACT guarantees that any concurrent execution is equivalent to a single-threaded execution of the program, it is easy to see why the concurrent executions of methods satisfy the denoted goals. Moreover, since a write-all operation –upon completion– updates the states of many nodes simultaneously, achieving consistency and coordination is facilitated.

Unreliable wireless communication due to collisions is a big challenge in the implementation of TRANSACT. We discuss how we cope with collisions using our receiver-side collision detection mechanism [5] in Section II-C.

## II. TRANSACT FRAMEWORK

### A. Language

A TRANSACT method consists of read and write-all operations and is of the form $read^*[write\text{-}all]$. Each read operation reads variables from a set of nodes in single-hop, and write-all operation writes to variables of a set of nodes in single-hop. A *thread* is an execution of a method, and can span across many nodes.

In Figure 1 we give some examples of TRANSACT methods for different tasks to illustrate the ease of programming in this model. TRANSACT methods return a boolean value denoting the successful completion of the method. If the method execution is aborted (e.g., due to conflicts with other threads or a lack of response to a read), it is the responsibility of the caller (application) to retry.

### B. Semantics

To keep the exposition simple we assume for the rest of the text that nodes have single thread of control, and focus on concurrent execution of threads only across nodes.

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 threads to the same variable are incompatible, so it is disallowed to swap the order of two such operations. In such a case, a causality is introduced from the first to the second thread. Similarly, two write operations to the same variable are incompatible with each other, and introduce a causality from the first thread to perform the write to the latter. If a read-write incompatibility introduces a causality from $t1$ to $t2$, and a write-write incompatibility introduces a causality from $t2$ to $t1$, then we say that $t1$ and $t2$ are conflicting. This is because, due to the causalities 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 thread to complete, when $t1$ tries to write-all, $t1$ is aborted due to the conflict.

TRANSACT provides guarantees on consistency and safety, but cannot provide very tight timeliness guarantees due to the contending nature of channel access. For example, when the bandwidth limits of the network is stretched due to a large number of communicating nodes in a region, it is not possible to provide tight real-time guarantees. Precaution should be taken to ensure the bandwidth limits are respected. Moreover, contention management schemes [11], [36] can be used to improve the real-time performance.

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

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

bool recovery_action() {
  StateColl=read(“*.state”); //read state from nbrs
  if($\neg$ legal(StateColl)) then //state is corrupted
    return write-all(correct(StateColl));
  return SUCCESS;}
```

Fig. 1. Sample methods in TRANSACT

### C. Read and Write-all operations

Broadcast communication opens novel ways for optimizing the implementation of read and write operations in OCC transactions. We identify these as follows:

1) Broadcast is atomic, that is, a broadcast is received by all recipients simultaneously
2) Broadcast allows snooping

Property 1 gives us a powerful low-level atomic primitive upon which we build the threads. Using Property 1, it is possible to order one transaction ahead of another, so that the latter is aborted in case of a conflict. (Property 1 does not rule away collisions nor asserts that a broadcast message should be reliably received by all intended nodes. We relegate the discussion of how we cope with collisions to the end of this section.) We use Property 2, i.e., snooping, for detecting conflicts between transactions without the help of an arbiter.

**Implementation of Read operation** : Since read operations are compatible with other read operations, it is possible to execute read operations—even those from the same thread—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. To avoid collisions of the reply, it is possible to exploit the order the variables are listed in the read-request message. For example, if $j.x$ occurred at the first place and $k.y$ occurred at the second in read-request, $j$ knows it should reply some time between 0-40ms of the read-request, and $k$ knows it should reply some time between 40-80ms of the read-request. This scheduling scheme is possible since the broadcasted read-request message is received by all recipients simultaneously.

**Implementation of Write-all operation** : The write-all broadcast performs a tentative write (a write to a sandbox) at each receiver. 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 small *cancel-ack* from each node to avoid an inconsistency due to loss of a cancellation message. Such small control messages are easily implementable under some WSN MACs [30], [37]. 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 nodes –if it completes successfully.

**Snooping for detecting conflicts** : As mentioned in Section II-B, any two threads $t1$ and $t2$ are conflicting if and only if a read-write incompatibility introduces a causality from $t1$ to $t2$, and a write-write incompatibility introduces a causality from $t2$ to $t1$. Detection of a conflict over distributed variables is a hard problem, further complicated by the case where the read-write and write-write incompatibilities are for different variables at separate nodes.

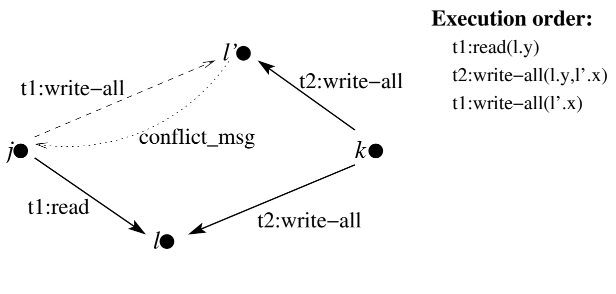

Fig. 2. Snooping for detecting conflicts

To enable low-cost detection of conflicts, we use nodes to act as proxies for detecting incompatibilities between transactions by snooping over broadcast messages. Figure 2 demonstrates this technique. Here $j$ is executing thread $t1$ 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 thread $t2$ which $write-all(l.y, l'.x)$. In this scenario $l'$ is the key. When $t1$ reads $l$, $l'$ learns about the pending $t1$ thread 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 incompatibility between $t1$ and $t2$. Later, when $t1$ writes tentatively to $l'.x$, $l'$ notices the write-write incompatibility between $t2$ and $t1$. Thus, $l'$ complains and aborts $t1$. Had there been 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.

**Fault-tolerance** : Even when single-hop neighbors are chosen conservatively to ensure reliable communication (we assume 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.

Occasional loss of a read-request message or a reply to a read-request message is detected by the node initiating the transaction when it times-out waiting for a reply from one of the nodes. After a second try of the read-request, the initiator node 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].

The collision of a write-all message is harder to detect. To this end, we employ our work on receiver-side collision detec- tion (RCD) [4], [5]. When a node receives a collided message (a carrier-sense mechanism at the MAC layer can successfully distinguish between an intense activity due to collision and an idle status of a channel [4]), it broadcasts a *collision-detected* message immediately. Note that it is not possible to identify the sender information, content, or type of the message collided, so this message is just a collision notification. Upon receiving a collision-detected message from one of the nodes involved in the transaction or a collision message (which may be due to collision of multiple collision-detected messages) within a small time frame of its write-all message, the initiator treats this as a conflict message. In this case, to avoid some intricate consistency issues that may be raised due to a re-broadcast of a write-all, a second try is not attempted and the initiator aborts its transaction by broadcasting a cancellation message as discussed above in the context of conflict-resolution.

Failure of an initiator node after it broadcasts a write-all may lead to inconsistent decisions among the nodes involved in the transaction. Even though this is a very rare fault compared to collisions and may not incite a solution, it may be possible to handle this case by devising a decentralized abort mechanism using snooping. Note that failures of other nodes are readily tolerated and do not lead to inconsistencies.

## III. 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], [22], [24], [25], [29], [31], 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.

Distributed systems community has invested significant effort on coping with concurrency issues, and proposed several formal verification frameworks, such as temporal logics of actions [23], Unity framework [3], etc. Researchers in distributed systems mostly considered wired, point-to-point network topologies, and preferred to use high-level models to think about atomicity at a coarser granularity than the underlying message-passing communication. For example, the shared memory model [9] uses a read and a write primitive: The read primitive reads atomically from all the neighboring nodes, and the write primitive writes only to the local state of the node. In the guarded-command model [3], [7], each action (a combination of read from neighbors and write to local state) is deemed atomic. These models do not provide any built-in support for deadlock/livelock prevention or serializability of the method executions —in contrast Transact provides all these.

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.

Similar to the conflict-serializability theorem in Transact, Seuss [28] provides a reduction theorem to the same effect. In contrast to the Transact model where the only allowed methods are "read" and "write-all" primitives in the $read^*[write\text{–}all]$ format and the only allowed call depth is one node, Seuss's remote procedure call based programming model is more general: call-depth is not-restricted, and the method structure is less constrained. On the other hand, the Seuss discipline requires a compile-time semantic compatibility check to be performed across nodes and allow only semantically compatible methods across nodes to run concurrently by asserting pre-synchronization inserted between incompatible methods. Note that in Transact we take an optimistic approach to concurrency control, and do not assert such restrictions. Also Seuss requires a proof of partial orders on methods at the compile-time in order to prevent the case where a method can be called malformedly as part of its execution.

Linda [2] introduced a tuple-space based programming model with two communication primitive: "in" (blocking) and "out" operation. Linda is prone to deadlocks and also does not provide support for a conflict serializability theorem.

Several programming abstractions have been proposed for sensor networks, including Kairos [14] and Hood [35]. Kairos 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 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.

## IV. Future Directions

**Implementing** Transact **under WSANs.** We will implement Transact under TinyOS [17] for the mote platforms [27]. TinyOS currently does not provide any mechanism for handling inadvertent nondeterministic executions across the nodes. To achieve conflict-serializability for TinyOS, we will implement the read and write-all operations of Transact as a TinyOS library component. Our component will *provide* read and write-all commands in its interface and implement these operations as we described in Section II-C. By asserting that the programmer use only Transact-style methods (of the form $read^*[write\text{–}all]$) for inter-process communication, we can provide the benefits of Transact framework for a TinyOS application.

TinyOS has a single-thread of control managed by the scheduler, however, due to interrupt-driven execution it is possible to have race-conditions among updates to intra-node level variables by tasks and events [13]. We will not implement transactions for preventing intra-node race conditions. To handle those, we revert to TinyOS, which provides "atomic" keyword and compile-time race condition detection for intra-node code [13]. We will initially limit our TinyOS implementation to initiate at most one Transact method at a time at a node. (Note that this does not restrict a node to concurrently participate at several transactions initiated by other nodes.) Our techniques can be extended to allow multiple threads executing in parallel at a node, and are applicable for more powerful WSAN architectures [32] that support multiple threads at a node.

Experimentation with implementations of TRANSACT will help in answering questions about the performance and overhead of the framework. For example, there are some latencies built into transactions, such as waiting for replies to a read operation or waiting for the count-down timer to expire before finalizing a write-all operation. Using table-top and later using large-scale testbeds [33], we will fine-tune these latencies. As a demonstration of TRANSACT framework, we will implement a decentralized traffic-light control application [1]. In this application, a number of remote-controlled toy cars (each carrying a Mica2 mote) will be arriving at an intersection from different directions. By running a leader-election method using TRANSACT, only one of the cars will get to proceed at a time while the others are stopped safely.

**Investigating patterns for building more efficient control programs.** The *consistent write-all* paradigm provided by TRANSACT enables a node to read from its neighbors and also to update state of its neighbors in a *consistent* and *simultaneous* manner. This new model differs from the traditional read-write models [3], [7], [9]. We believe it is possible to build more efficient control programs by exploiting the TRANSACT model than using traditional models. We will provide a library of examples/applications (patterns [12]) that achieve efficient control and coordination among nodes. One effective pattern could be to assign the controller duty of each service to a few nodes and get other nodes to serve as slaves in those respects to the nodes responsible for a service (for instance, some nodes may take over the responsibility for leader-election and consensus duties while others control resource discovery, recovery, and topology control of the network). Programmers can then adopt these patterns to achieve the same improvements in their code quickly.

**Verification support.** To enable the application developer to check safety and progress properties about her program, verification support may be included for TRANSACT. 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 due to the simplistic format of the methods. Since TRANSACT methods consist of read operations followed by one write operation at the end, it is easy to reason about the effects of a method as a guarded-command [7], or more generally as a transition [26]. This facilitates translation between TRANSACT methods and other existing verification toolkits, such as model checkers [19], [20].

## REFERENCES

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

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

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

[4] G. Chockler, M. Demirbas, S. Gilbert, and C. Newport. A middleware framework for robust applications in wireless ad hoc networks. In *43rd Allerton Conf. on Communication, Control, and Computing*, 2005.

[5] G. Chockler, M. Demirbas, S. Gilbert, C. Newport, and T. Nolte. Consensus and collision detectors in wireless ad hoc networks. In *PODC*, pages 197–206, 2005.

[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] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

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

[9] M. R. Eskicioglu. A comprehensive bibliography of distributed shared memory. *SIGOPS Oper. Syst. Rev.*, 30(1):71–96, 1996.

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

[11] R. G. Gallager. A perspective on multiaccess channels. *IEEE Transactions on Information Theory*, 31(2):124–142, 1985.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995. GAM e 95:1 1.Ex.

[13] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, 2003.

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

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

[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, Reading, Massachusetts, 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] 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.

[23] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[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] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[27] Crossbow technology, Mica2 platform. `www.xbow.com/Products/Wireless_Sensor_Networks.htm`.

[28] J. Misra. A discipline of multiprogramming. *ACM Computing Surveys*, 28(4):49–49, 1996.

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

[30] 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.

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

[32] Crossbow technology, Stargate platform. `http://www.xbow.com/Products/XScale.htm`.

[1]This demo idea is due to Nancy Lynch.

[33] OSU NEST ExScal Team. Kansei: Sensor testbed for at-scale experiments. http://www.cse.ohio-state.edu/kansei.

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

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

[36] D. E. Willard. Log-logarithmic selection resolution protocols in a multiple access channel. *SIAM J. Comput.*, 15(2):468–477, 1986.

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