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 propose a lightweight and fault-tolerant implementation of our transactional framework.

1 Introduction

Traditionally wireless sensor networks (WSNs) act mostly as data collection and aggregation networks and do not possess a significant actuation capability [2, 34]. 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 [33]. Future wireless sensor/actor networks (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.

The state of the programming technology for WSANs is currently far behind that of the available physical technology for WSANs [33]. The research on WSN programming methologies are not readily applicable in the context of WSANs, as 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, 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 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, 15, 19], 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.

Contributions of the paper. To enable ease of programming and reasoning in WSANs, and yet allow concurrent execution, we propose a programming abstraction and an associated framework, namely *TRANSACT: TRANsactional framework for Sensor/ACTor networks*. TRANSACT provides 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.

A major contribution of TRANSACT is that it simplifies the reasoning and verification of a distributed WSANs program. 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.

Secondly, TRANSACT enables ease of programming for WSANs applications. Building blocks for process control and coordination programs (such as, leader election, mutual exclusion, cluster construction, neighborhood discovery, recovery actions, and consensus) are easy to denote using TRANSACT (see Figure 1). Also, TRANSACT introduces a novel *consistent write-all* paradigm that enables a node to update the state of its neighbors in a *consistent* and *simultaneous* manner. 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.

Thirdly, TRANSACT is novel in that it proposes an efficient and lightweight implementation of a transactional framework in a distributed manner. Implementing transactions in distributed WSANs domain diverges from that in the database context significantly, and introduces new challenges to address. 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. However, by exploiting the properties of broadcast communication inherent 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. 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 incompatible transactions).

Finally, TRANSACT is robust to the face of node failures and message losses. Unreliable wireless communication (e.g., message losses and collisions) in WSANs is a big challenge in the implementation of TRANSACT. By utilizing explicit acknowledgements and eventually-reliable unicasts efficiently, TRANSACT manages to provide a consistent and atomic broadcast abstraction.

2 TRANSACT Framework

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, optionally, by a write-all operation. Each read operation reads variables from some nodes in singlehop, and write-all operation writes 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 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 $t1$ and $t2$ have a read-write incompatibility from $t1$ to $t2$ and also a write-write or a read-write incompatibility from $t2$ to $t1$ with respect to some variables (defined in Section 2.2). Conflicts are detected in an efficient manner as nodes can snoop on broadcasted messages in singlehop (see Section 2.3). 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 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.

2.1 Language

A TRANSACT method consists of read and write-all operations and is of the form $read^*[write{-}all]$. Each read operation reads variables from a set of nodes in singlehop, and write-all operation writes to variables of a set of nodes in singlehop. 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. As an example consider the become_leader method. Here the initiator node reads all neighbors and declares itself as the leader if none of its neighbors has a leader. It is easy to see that in a single-threaded execution of the system, this method ensures that there can be at most one leader within a singlehop neighborhood. TRANSACT automatically satisfies the same property for a concurrent execution of the system. In a scenario where two nodes execute become_leader concurrently, they may both decide to

declare themselves as the leader. However, since one of the write-all broadcasts will precede that of the other (we discuss the case of collisions of broadcasts and the associated message loss in Section 2.4), due to a reporting of a conflict (which follows from read-write and write-write incompatibilities between the two transactions) the other transaction is aborted.

```
bool become_leader(){
  LeaderSet=read("*.leader"); //read all nbrs
  if (LeaderSet = ∅) //declare self as leader
    then return write-all("*.leader="+self.ID);
  return FAILURE; }

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

bool recovery_action() {
  StateColl=read("*.state"); //read state of nbrs
  if(¬ legal(StateColl)) //state is corrupted
    then return write-all(correct(StateColl));
  return SUCCESS;}
```

Figure 1. Sample methods in TRANSACT

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. For example in the Consensus method, upon failure to agree on the same value, the initiator node may retry until consensus is achieved. When consensus is achieved the value is decided consistently and simultaneously by all the participating nodes.

2.2 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. Two write operations to the same variable are also incompatible with each other, and introduce a causality from the first thread to perform the write to the latter. As in Figure 2 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. Similarly, a read-write incompatibility from $t1$ to $t2$, and another read-write incompatibility from $t2$ to $t1$ (e.g., $t1$ =read($l.x$);write-all($l'.y$) and $t2$ =read($l'.y$);write-all($l.x$)) results in a conflict and abortion of one of the transactions (the one with the later write-all operation).

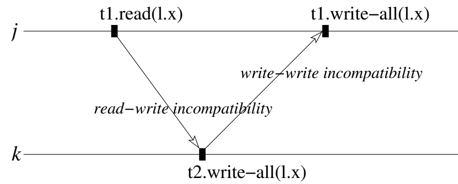

Figure 2. Conflicting transactions

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.

2.3 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. 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 propogation speed of light). As such 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 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 2.4.) 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. Each receiver replies back with a small *acknowledgment* message. Such control messages are easily implementable under some WSN MACs [30, 37]. Again, to avoid collision of acknowledgments, the order the variables are listed in the write-all message can be used. 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 nodes –if it completes successfully.

Snooping for detecting conflicts : As mentioned in Section 2.2, 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 or a read-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.

To enable low-cost detection of conflicts, we use nodes to act as proxies for detecting incompatibilities between

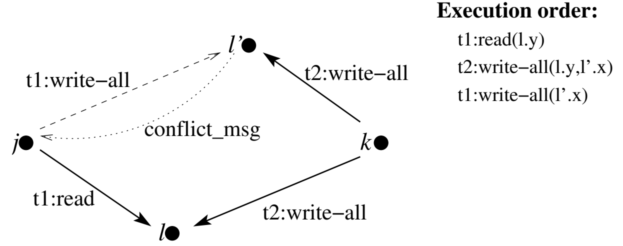

Figure 3. Snooping for detecting conflicts

transactions by snooping over broadcast messages. Figure 3 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.

2.4 Fault-tolerance

Even when singlehop 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 via utilizing explicit acknowledgements and eventually-reliable unicast.

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].

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. 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.

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" [13] in the presence of arbitrarily unreliable communication, the above assumption is necessary even for solving a most basic consensus problem in a distributed system [5]. Such an eventually-reliable unicast assumption is realistic under reasonable network loads as the MAC protocols [30,37] can resolve collisions via carrier-sense and back-offs.

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 message losses and may not incite a solution, it is possible to handle this case by devising a decentralized abort mechanism using snooping. The node that reported the conflict may act as a shepherd and cancels the transaction in case the initiator is down and does not take any action. Note that failures of other nodes are readily tolerated and do not lead to inconsistencies.

3 Related Work

Distributed systems community has invested significant effort on coping with concurrency issues. The 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 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 [4, 7], each action (a combination of read from neighbors and write to local state) is deemed atomic. Finally, Linda [3] introduced a tuple-space based programming model with two communication primitive: "in" (blocking) and "out" operation. Unfortunately, none of these models provide built-in support for the serializability of the method executions —in contrast TRANSACT provides conflict-serializabilty via the transaction abstraction. Especially for adaptations of Linda to ad hoc networks domain [28], our TRANSACT framework can be instrumental for implementing and maintaining the consistency of the underlying distributed tuple-space.

Similar to the conflict-serializability theorem in TRANSACT, the Seuss programming discipline [26] 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 $read^*[write{-}all]$ 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. This hinders compositionality and ad hoc interoperability gravely. 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.

A cached sensor transform (CST) that allows simulation of a program written for interleaving semantics in WSNs under concurrent execution is introduced in [17]. 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.

Several programming abstractions have been proposed for sensor networks, including Kairos [14] and Hood [35]. Kairos allows a programmer to express global behavior of a WSN in a centralized sequential program and provides compile-time and runtime systems for 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.

Virtual node infrastructure [9] provides an overlay net-

work of fixed virtual nodes (VNs) on top of a mobile ad hoc network to abstract away the challenges of unpredictable mobility and unpredictable reliability of the mobile nodes. Each VN is simulated by the real mobile nodes in the VN's region in the network. The implementation assumes reliable communication channels and uses a round-robin approach to achieve robust replication of the state of the VN over the real nodes. 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 environment. TRANSACT framework is orthogonal to the VN idea and provides a lightweight abstraction for implementing VNs over unrealistic communication channels.

Software-based transactional memory (STM) approach has been proposed in earlier work [16, 32], however, the scope of those work is limited to threads interacting through memory in a single process. In that domain, STM functions as an alternative to lock-based synchronization and offers optimistic synchronization, achieving increased concurrency. In contrast, TRANSACT focuses on transactions among distributed nodes and inter-node concurrency issues, and exploits singlehop wireless broadcast primitive to efficiently implement distributed transaction processing.

Finally, 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 transactions for mobile ad hoc networks [6, 22–24, 29, 31], however, these work all assume a centralized database and arbiter at the server, and try to address the consistency of hidden read-only transactions initiated by mobile clients. Work on distributed databases use two-phase locking for concurrency control and employ two-phase commit for ensuring correct completion of distributed transactions [13, 27]. In contrast to OCC, which performs a lazy evaluation to resolve conflicts (if any), two-phase locking takes a speculative approach and prevents any possibility of conflicts *by forbidding any read-write or write-write incompatibilities in the first place*. However, this aggressive strategy takes its toll on the concurrency of the system and limits number of simultaneous transactions the system can support. Two-phase locking is also prone to deadlocks, and is not composable.

4 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. By providing a library of patterns [12] for efficient control and coordination among nodes, we can help the programmers to reuse these patterns and achieve the same improvements in their code quickly.

In this paper we also outlined an efficient and lightweight implementation of TRANSACT in WSANs in a distributed manner. The major challenge for this implementation has been that, 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. We overcome this challenge by exploiting the properties of broadcast communication: a broadcast is received by the recipients simultaneously, and broadcast allows snooping. The first property gives a low-level atomic primitive that we use to order a transaction ahead of another, and the second property allows snooping for detecting conflicts between transactions in a distributed manner, without the help of an arbiter. Another challenge has been the unreliable nature of wireless communication. We overcome this challenge by utilizing explicit acknowledgements and eventually-reliable unicasts efficiently.

We are currently working on implementing TRANSACT in TinyOS [18]. TinyOS currently does not provide any mechanism for handling inadvertent nondeterministic executions across the nodes. (The "atomic" keyword and compile-time race condition detection in TinyOS helps only for preventing intra-node race conditions.) To achieve conflict-serializability for distributed TinyOS applications, we will implement the *read* and *write-all* operations of TRANSACT as a TinyOS library component. By asserting that the programmer use only TRANSACT-style methods (of the form $read^*[write{-}all]$) for inter-process communication, we will provide the benefits of TRANSACT framework for a TinyOS application. As a demonstration, we plan to implement a decentralized traffic-light control application [1]. In this application, a number of remote-controlled toy cars (each carrying a Mica2 mote [25]) 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.

In future work, we 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,

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

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 [20].

References

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

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

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

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

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

[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. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, 2005.

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

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

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

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

[21] H. T. Kung and J. 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] 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.

[24] 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.

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

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

[27] M. T. Ozsu and P. Valduriez. *Principles of distributed database systems*. Prentice-Hall, Inc., 1991.

[28] 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.

[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] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing (PODC)*, pages 204–213, 1995.

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

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

[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.