Optimistic Concurrency Control for Multihop Sensor Networks

Onur Soysal
Google
Mountain View, CA 94043
Email: onursoysal@google.com

Bahadir Ismail Aydin
Computer Science & Engineering Dept.,
University at Buffalo, SUNY
Email: bahadiri@buffalo.edu

Murat Demirbas
Computer Science & Engineering Dept.,
University at Buffalo, SUNY
Email: demirbas@buffalo.edu

***Abstract*—In this study, we provide a lightweight singlehop primitive, Read-All-Write-Self (RAWS), that achieves optimistic concurrency control. RAWS guarantees serializability, which simplifies implementation and verification of distributed algorithms, compared to the low level message passing model. We also present a self-stabilizing multihop extension of RAWS, called Multihop Optimistic Concurrency Control Algorithm (MOCCA), to address the challenges of multihop networks. We implement RAWS on motes and investigate the effects of message loss over this novel primitive.**

I. Introduction

Wireless sensor networks (WSNs) and the emerging wireless sensor/actuator networks (WSANs) employ in-network/decentralized computation in order to reduce communication costs. Message passing is usually the only paradigm used for implementing these in-network/decentralized algorithms. Although message passing is expressive enough, it entails substantial complexity in analysis and implementation due to the concurrent execution problems. As a result, unintentional and unwanted nondeterministic executions can haunt the correctness of the decentralized algorithms. The application programmer should not be unduly burdened to detect, debug, and prevent such race conditions. Higher order abstractions should be adopted to simplify the design and analysis of decentralized algorithms by transparently solving the concurrency control problem. However, high order abstractions should themselves be implemented in an energy-efficient/low-cost manner, in order not to defeat the purpose of in-network computation. For example, even though it is possible to prevent race conditions by tightly controlling interactions between nodes and locking access to the shared resources, this would destroy concurrency, which is necessary for providing real-time guarantees for the system.

To address this problem, we provide a lightweight transaction primitive with optimistic concurrency control: Read-All-Write-Self (RAWS). Each transaction allows reading variables from singlehop neighbors and modification of local variables. In RAWS we utilize atomic broadcast nature of radios in WSNs/WSANs to ensure serializability of transactions. Serializability enables reasoning about the properties of a distributed WSAN execution as interleaving of single transactions from its constituent nodes, when in reality, the transactions at each of the nodes are running concurrently. In other words, RAWS eliminates unintentional nondeterministic executions while retaining concurrency of executions. As such, our RAWS abstraction simplifies development through simpler validation of system properties. We present RAWS in Section III, and provide analysis and implementation results as well.

With the recent emphasis on cyber-physical systems, multihop control applications are becoming more important and mission-critical. For example, in an automaticially controlled brake system scenario, result of inconsistent brake actions in different cars may be an accident. In this kind of applications, enabling concurrency in a multihop network of sensors and actors while yet keeping the consistency is a crucial task. In order to achieve that, the system should response any change very rapidly. Moreover, multihop introduces new concurrency control challenges such as long dependency chains which are hard to detect. Section IV describes these differences and their reasons in detail. While Transact is very effective for singlehop networks, it cannot satisfy these inherent requirements of multihop case. However, optimistic concurrency control is an effiecient tool for enabling single thread of execution view, still in multihop domain.

Optimistic concurrency control primitives need special care when applied to multihop domain. The atomic synchronous broadcasts available in singlehop are not directly supported in multihop networks. Moreover, in multihop networks, dependencies among transactions get more complicated as we analyze in Section IV. Our solution to address these challenges is an incremental, self-stabilizing algorithm: Multihop Optimistic Concurrency Control Algorithm (MOCCA). MOCCA provides a solution to the multihop concurrency problem without sacrificing the benefits of optimistic concurrency control. To the best of our knowledge, this is the first study on optimistic concurrency control in multihop WSNs/WSANs. We present MOCCA in Section V.

Due to the lack of transformation tools, classic coordination and collaboration algorithms (such as consensus, leader election, group membership, virtual synchrony, mutual-

This material is based upon work supported by the National Science Foundation under Grant No. 0916504.

exclusion, resource allocation) that are extensively studied in the distributed systems literature do not translate directly to the WSANs domain for achieving high-assurance programs. Since RAWS/MOCCA provides a shared memory abstraction, it acts as a middleware and enables the reuse of existing distributed algorithms designed with the shared memory model, in the WSNs and WSANs environments.

Apart from the programming convenience, concurrent execution can be beneficial from an energy efficiency perspective. Energy efficiency can be improved by reducing the communication and reducing the time required for task completion. Through increased concurrency, more data can be transferred to more recipients with less transmissions in smaller time. Concurrency and more specifically optimistic concurrency control is capable of providing significant benefits in both execution time and energy use due to the possibility of exploiting broadcast nature of radio communication. Concurrency also reduces the impact of processing delays to performance since processing delays can also be made concurrent and overlapping.

II. Related Work

Programming abstractions for WSNs and ad hoc networks. Several useful programming abstractions have been proposed for WSNs, including Kairos [10], Hood [23], abstract regions [22], and TeenyLime [4]. 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 by caching the values of the neighbors' attributes periodically, while simultaneously sharing the values of the node's own attributes. Similar to Hood, abstract regions and TeenyLime propose mechanisms for discovery and sharing of data (structured in terms of tuples) among sensor nodes. In contrast to these abstractions that target WSNs and provide best-effort semantics (loosely-synchronized, eventually consistent view of system states), RAWS and MOCCA focus on providing a dependable framework with well-defined consistency and conflict-serializability guarantees.

Linda [2], [17] and virtual node infrastructures (VN) [6] propose high-level programming abstractions for coping with the challenges of building scalable applications on top of distributed, and potentially mobile, ad hoc networks. These abstractions can be converted to shared memory programs which, in turn, can be realized through RAWS transactions.

Recently in [1], we considered the problem of transformations from shared memory to WSNs model. The performance of transformations suffer in the presence of prevalent message losses in WSNs, and we proposed a variation of the shared memory model, where the actions of each node are partitioned into slow actions and fast actions. The traditional shared memory model consists only of fast actions and a lost message can disable the nodes from execution. Slow actions, on the other hand, enable the nodes to use slightly stale state from other nodes, so a message loss does not prevent the node from execution. Since RAWS/MOCCA provides a shared memory to WSNs model transformation, the slow-fast partitioning techniques proposed in [1] also applies for RAWS/MOCCA to to enable the transformed program to be more loosely-coupled, more conflict-serializable, and tolerate communication problems (latency, loss) better.

Programming abstractions for concurrency control. Recently, there has been a lot of work on transactions for mobile ad hoc networks [19], [14], [13], [3]. Concurrency control in RAWS and MOCCA diverges from these work and the transactions in the database context significantly. These work all assume a centralized database at the server, and try to address the consistency of transactions by mobile clients. In contrast, in RAWS there is no central database repository or arbiter; the control and sensor variables are maintained distributedly over several nodes.

Distributed databases use two-phase locking for concurrency control and two-phase commit for ensuring correct completion of distributed transactions [8], [15], [16]. 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.

Software transactional memory (STM) [20], [11], [12], [18] is a concurrent programming scheme with multiple threads. In STM conventional critical sections for controlling access to shared memory are replaced by transactions. In RAWS, there is no actual shared memory as the variables are distributed among nodes.

A closely related work to this work is our work on Transact [5] which presented a transactional programming primitive for WSNs. In contrast to Transact, which uses a Read-All-Write-All model, in our work we use a Read-All-Write-Self model with much less communication cost and much smaller transaction duration. Transact model depends on explicit conflict detection and cancel operations for serializability. Although Read-All-Write-All model is quite expressive, conflict detection and cancel operations effectively triple transaction duration and introduces serious complications. Transcast focuses on singlehop neighborhood transactions and does not address the problem of detecting inconsistencies that may arise in multihop environments due to transaction chains. This work on the other hand, provides a single phase primitive with minimal communication and it is the first optimistic concurrency control algorithm for WSNs that can function in multihop networks with consistency guarantees.

III. RAWS: Read All Write Self

The RAWS primitive provides a means for each sensor node to perform non-local computations in a serializable

manner. The RAWS primitive consists of a transaction initiation message that requests to read a subset of local neighborhood followed by read responses from queried nodes. RAWS writes only to the initiator node and the set of variables to be modified is included in the initiation message. RAWS transactions utilize time-based commits where the transaction is committed (or canceled) after a fixed duration following the read query. Since only the initiator state can be modified using RAWS, only the initiator needs to keep track of the success of the transaction. Two conditions must be satisfied at the initiator for a RAWS transaction to be successful: no conflicts should be detected and all read responses must be received. The nodes in the read set, called contributor nodes, engage in this process by withholding the transmission of read responses when they detect conflicts.

Any application using RAWS can enqueue a transaction at any time, but the actual start time depends on other running transactions. Additionally, the started transaction might fail due to conflicts. When this happens the application is notified so the transaction might be repeated or other recovery action might be taken. The application starting the transaction runs Algorithm 1, and all nodes run Algorithm 2 to handle requests from initiators. Both algorithms employ Algorithm 3 for performing conflict detection.

Algorithm 1 Initiator algorithm for RAWS

```
1: add new transaction to list of transactions
2: if conflict detected then
3:    remove transaction from list of transactions
4:    return FAIL
5: else
6:    send initiation message
7:    wait until commit time
8:    if all read responses received then
9:       update variable
10:      return SUCCESS
11:   else
12:      remove transaction from list of transactions
13:      return FAIL
14:   end if
15: end if
```

Conflicts in optimistic concurrency control correspond to a set of non serializable transactions. Conflicting transactions, when run in parallel, produce a state not achievable with any serial order of executions. We give more details on our approach for detecting and preventing conflicts next.

A. Conflict Detection

Optimistic concurrency control assumes that transactions will be compatible with each other most of the time. Instead of preemptively preventing concurrency, conflicting transactions are aborted at the time of detection. As long as all conflicts are detected this scheme will be equivalent, in terms of correctness, to the more restrictive locking-based protocols.

Algorithm 2 Contributor algorithm for RAWS

```
1: wait for an initiation message
2: clear completed transactions from list
3: add new transaction to list of transactions
4: if conflict detected then
5:    remove transaction from list of transactions
6: else
7:    if involved in transaction then
8:       send read-response
9:       return
10:   end if
11: end if
```

A set of transactions is serializable if and only if their dependency graph is acyclic [9]. Conflict detection is employed to maintain this property for all concurrent transactions by labeling any cyclic dependencies as conflicts. Whenever a new transaction is started, all nodes run the conflict detection algorithm shown in Algorithm 3. Note that a directed graph will have a valid topological order if and only if it is acyclic. This fact is utilized in this algorithm for detecting conflicts.

Algorithm 3 Conflict Detection

```
1: E ← {} // Set of dependencies is initially empty
2: for all Transactions t in transaction list T do
3:    for all Transactions u in transaction list T do
4:       if t reads initiator of u then
5:          E ← E ∪ (t, u) // t depends on u
6:       end if
7:    end for
8: end for
9: topologically sort transactions T using E as order
10: if ordering possible then
11:    return FALSE // transactions are serializable so no
       conflicts
12: else
13:    return TRUE // transactions are not serializable so
       report conflict
14: end if
```

B. Theoretical Analysis of Transactional Serializibility

It is not good programming practice to initiate several transactions burstily at or around the same time, and we do not consider this as the normal use case of the RAWS primitive. Still, this case is worth considering to analyze the success rate of transactions that are highly dependent to each other. We like to mention that this is a worst case analysis with very bursty transaction initiation pattern. As we mention in Section II, the techniques in [1] can be used to alleviate those effects, but we do not consider them for our worst case analysis below.

For our analysis, we define the dependency density ratio as number of actual dependencies (i.e., the number of nodes read by the running transactions) over all possible dependencies (i.e., the number we get assuming as if each initiator reads from all of its singlehop neighbors). We find that 50% dependency rate is the theoretical upper limit to serialize several transactions. Even then there is only one topological dependency graph which is serializable at 50% dependency rate: a node depends on all other $n-1$ nodes, then second node depends same $n-2$ nodes excluding itself, and similarly latter nodes go on depending same list of nodes exluding themselves one by one until the last transaction cannot be dependent to any other nodes. So we get $(n-1)+(n-2)+(n-3)+\ldots+0 = (n^2-n)/2$. Any new dependency would create a cycle in the described dependency graph. Note that all possible number of dependencies are $n*(n-1) = n^2-n$, and this observation gives us the dependency possible maximum density ratio = 50% for a serializable transaction list.

In general, the dependency density ratio for x dependencies in an n node topology is given by the following equation.

$$\sum_{i=2}^{x} \frac{\binom{n}{i}*\binom{n^2-n-i}{x-i}-Z}{2*\binom{n^2-n}{x}}$$

Armed with this formula, we generated a simulation that tries all distributions for a given number of transactions whether it is serializable or conflicting. As for Z, the number of recounted unsafe links, we use it in our calculations but encapsulate it here due to clarity as its expansion is much larger than the equation while impact is insignificant. The number of possible distributions grow quadratic-exponentially with $O(2^{n^2-n-1})$, which we calculated as $\binom{n^2-n}{1}+\ldots+\binom{n^2-n}{(n^2-n)/2}$.

Figure 1 shows that as number of dependencies increases number of total possible distributions increases, while number of serializable distributions decreases roughly after 30% dependency density. So, even in the worst case of a crowded topology where nodes start transactions simultaneously, 30% dependency density provides a good chance of serializability.

As number of dependencies increases number of total possible distributions increases, while number of serializable distributions decreases roughly after 30% dependency density. So, even in the worst case of a crowded topology where nodes start transactions simultaneously, 30% dependency density provides a good chance of serializability.

IV. Concurrency Control in Multihop Networks

Multihop networks pose an additional problem for optimistic concurrency control as the cycles in dependencies may not be limited to a single hop neighborhood. Centralized solutions do not suffer from this problem as all transactions will be known by a central server, however this requires all transactions to be aggregated at a central location and central server needs to send coordination

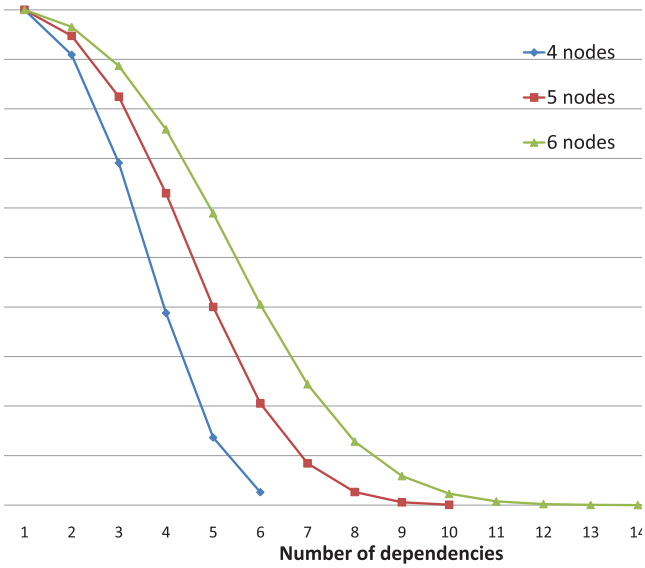

Fig. 1. Success rate with respect to number of dependencies

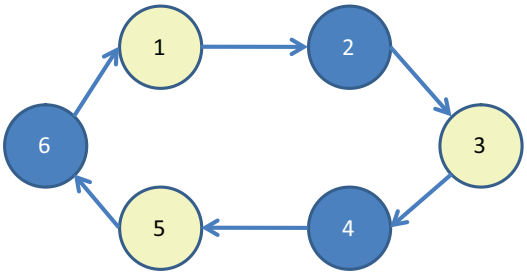

Fig. 2. A pathological multihop dependency graph. Circles correspond to nodes and arrows show dependencies between RAWS transactions running on nodes. Transactions are all concurrent and started with the numerical order.

messages back to nodes to limit concurrency. Flooding all transactions to whole network is another alternative but it has even higher communication costs as each transaction with such flooding would cost $O(n)$ communication instead of $O(1)$.

A tempting solution to this problem is piggy-backing additional dependency information to all transactions. In this approach all initiation messages would also include the set of known running transactions with all required dependency information. Even if we ignore the limitations of message sizes in radios, this method still can not capture many dependency problems. Figure 2 demonstrates a pathological scenario in which it would not be possible to solve using such an approach. In this scenario, each node starts its transaction at time unit corresponding to its node id and transaction duration is 10 time units. We denote transaction on node j depends on transaction at node i with $i \rightarrow j$. Even after all nodes initiate their transactions, no node is capable of detecting the cycle in the network. An

important observation for this sequence is that the all light nodes (0,1 and 2) start before the dark nodes (3,4 and 5). With this order, information from at most 2hop neighbors can arrive to any node. Node 0 does not have information of dependency between nodes 1 and 4, Node 2 does not have information about dependency between nodes 3 and 1, etc.

Utilizing read responses in this process as well is equivalent to repeating this process twice while keeping the known dependencies. This extension solves the case in Figure 2, but fails in a similarly constructed scenario with 10 nodes. More generally for n round of messaging, there exists a $4n+2$ node scenario that can not be solved with this method. Each round of messaging after first round increases the length of detected chains by 2 from each side, hence the 4 factor in the formula. In conclusion, this method requires $O(n)$ rounds of extra messaging and increases the size of each message by $O(n)$ thus quite infeasible for real life deployments.

Our approach in this study is to prevent these pathological cases, rather than trying to detect them. Although we sacrifice some concurrency and pay additional cost for the algorithm, we prevent inconsistencies and still achieve substantial concurrency.

A very simple way to avoid multihop dependency loops is to forbid any dependencies between concurrent transactions. This is similar to having read-only locks on the read set of RAWS transactions. Although this approach is safe, it reduces the concurrency of the system. We call this method "locking" and use it as a baseline for our experiments.

V. MOCCA: Multihop Optimistic Concurrency Control Algorithm

In order to conceptualize our method of multihop concurrency control, we introduce the concept of a *color* for each node. The *color* of a node is used in each of its RAWS transactions to limit concurrency. In addition to satisfying dependency requirements as explained in Section III-A, we require all RAWS transactions with dependencies running at a node to have same color. Now the question becomes how to assign the nodes colors so that we both prevent multihop dependency loops and provide high concurrency. More formally we can define two properties **safety** and **concurrency** as follows. **Safety**: All dependency loops occurring through execution of RAWS transactions must be serializable. **Concurrency**: The concurrency limitations on the RAWS transactions should be minimal.

For the safety property, we depend on RAWS to detect conflicts. The concurrency property on the other hand is related to the number of distinct neighboring colors for each node. The chance of cancellations caused by color constraints increase with the number of colors, which in turn decreases concurrency.

Satisfying both of these properties is closely related to a graph theory problem, subcoloring. Subcoloring corresponds to an assignment of colors to a graph's vertices where each color class induces a vertex disjoint union of cliques. Unfortunately, minimal subcoloring problem is NP-complete even for triangle-free planar graphs[7]. Instead of searching for an optimal solution which would require exhaustive and possibly centralized computations, we opt for an incremental heuristic approach called MOCCA, Multihop Optimistic Concurrency Control Algorithm.

MOCCA is an *incremental* and *self-stabilizing* algorithm for distributed subcoloring problem using RAWS transactions. By term *incremental* we refer to the fact that MOCCA operations are a set of RAWS transactions which can be interleaved to regular operations of RAWS. Moreover, any intermediate state of MOCCA still satisfies the safety property, allowing the application developer fine tune cost and benefit of optimization. This property also permits MOCCA execution without a setup phase. Self-stabilization on the other hand provides robustness for MOCCA, where local errors can be fixed after finite number of RAWS transactions.

MOCCA uses RAWS transactions to read color of each of its neighbors. Two kinds of transactions are utilized for this purpose:*update* and *modification*. *Update* transactions are read only transactions to discover whether there exists a better color for the node. *Modifications* are initiated after updates to actually modify the color. The update transactions are introduced to address our observation that color of node actually needs to change relatively few times yet there are many occasions that might lead to a change in color.

Nodes save the color of their neighbors after each update to be utilized when answering to other nodes requests. In addition, whenever a neighbor starts a MOCCA modification, the commit time of this transaction is noted as last modification time of this neighbor. Moreover the color of this node is marked unknown as modification might be changing the color of node. The read response for update and modify transactions contain color of the node and the status of the color. The status of a color can take three values: *forbidden*, *suspicious* and *safe*. A color c is labeled forbidden when there is a neighbor of the contributing node with color c which is not a neighbor of the initiator. A color c is labeled suspicious when there is a neighbor of the contributing node which is not a neighbor of the initiator whose color is unknown. Color is deemed safe in all other cases.

Initiator node of MOCCA transaction combines all read responses from its neighbors to construct a list of safe colors. A color is considered as forbidden if any of the neighbors declares that color forbidden. If a color is not forbidden but some neighbors declare that color is suspicious then the color is considered as suspicious. Otherwise the color is considered as safe. MOCCA initiator counts the number of nodes in each safe color. Among the safe colors with highest cardinalities a random one is chosen as next color. To improve stabilization of the algorithm the current color is chosen if it has the highest cardinality. This

Algorithm 4 MOCCA

1: **if** neighbor modified **then**
2: $needsUpdate \leftarrow$ **true**
3: **end if**
4: **if** $needsModification$ **then**
5: run modification RAWS
6: $color \leftarrow chooseColor()$// *update color*
7: $needsModification \leftarrow$ **false**// *no more modification is necessary*
8: **else if** $needsUpdate$ **then**
9: run update RAWS
10: $newColor \leftarrow chooseColor()$
11: **if** $newColor \neq color$ **then**
12: $needsModification \leftarrow$ **true**// *a better color is present, a modification RAWS is required*
13: **else**
14: $needsModification \leftarrow$ **false**// *no better alternative exists, so no modification is necessary*
15: **end if**
16: **if** $\exists c | suspicious(c)$ **then**
17: $needsUpdate \leftarrow$ **true**// *since there are undecided 2-hop neighbors another update is needed*
18: **else**
19: $needsUpdate \leftarrow$ **false**
20: **end if**
21: **else**
22: **return** // *stabilized; no more color operations necessary*
23: **end if**

functionality is implemented in *chooseColor*() command. We summarize MOCCA in Algorithm 4. We also prove this algorithm is self-stabilizing but due to reasons of space, we relegate the safety and stabilization proofs of MOCCA to the technical report [21].

VI. Concluding Remarks

In this work we proposed a single hop primitive Read-All-Write-Self to simplify programming of WSNs and WSANs. Our RAWS framework utilizes an optimistic concurrency control scheme and guarantees serializability for singlehop networks. We also identified challenges in implementing our RAWS primitive in a multihop environment, and showed that a set of transactions spanning multihop neighborhoods may violate serializability. To address this problem, we proposed a constraint based solution, which prevents such multihop inconsistency chains. In order to improve the multihop performance of RAWS, we reduced the concurrency constraint problem to a graph subcoloring problem. We provided an incremental, self-stabilizing algorithm for graph subcoloring named Multihop Optimistic Concurrency Control Algorithm (MOCCA). Finally, we implemented RAWS on TinyOS-based motes and our results showed that our algorithm doens't suffer from real world conditions more than mundanely.

References

[1] M. Arumugam, M. Demirbas, and S. Kulkarni. Slow is fast for wireless sensor networks in the presence of message losses. In *12th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2010.

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

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

[4] P. Costa, L. Mottola, A. Murphy, and G. Picco. Teenylime: transiently shared tuple space middleware for wireless sensor networks. In *MidSens*, pages 43–48, 2006.

[5] M. Demirbas, O. Soysal, and M. Hussain. Transact: A transactional framework for programming wireless sensor/actor networks. *Information Processing in Sensor Networks, 2008. IPSN '08. International Conference on*, pages 295–306, April 2008.

[6] 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.

[7] J. Gimbel and C. Hartman. Subcolorings and the subchromatic number of a graph. *Discrete Mathematics*, 272(2-3):139–154, 2003.

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

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

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

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

[12] M. Herlihy and E. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300. May 1993.

[13] 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.

[14] 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.

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

[16] M. T. Ozsu and P. Valduriez. Distributed and parallel database systems. *ACM Comput. Surv.*, 28(1):125–128, 1996.

[17] G. P. Picco, A. L. Murphy, and G.-C. Roman. Lime: Linda meets mobility. In *ICSE*, pages 368–377, 1999.

[18] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505. IEEE Computer Society, Jun 2005.

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

[20] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213. Aug 1995.

[21] O. Soysal and M. Demirbas. Optimistic concurrency control for multi-hop wireless sensor networks. *Technical Report, Deparment of Computer Science and Engineering, University at Buffalo, available at http://www.cse.buffalo.edu/tech-reports/2009-05.pdf*, 2009.

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

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