Optimistic Concurrency Control for Multi-hop Wireless Sensor Networks

Onur Soysal
Dept. of Computer Science & Engineering
University at Buffalo, SUNY
osoysal@buffalo.edu

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

Abstract

With the inclusion of actuation capabilities, emerging wireless sensor applications are much less tolerant to inconsistencies in decisions compared to passive sensing applications. Multi-hop networks suffer from these problems more profoundly as they cannot directly utilize atomic broadcast operations for coordination.

In this study, we provide a lightweight single hop 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 multi-hop extension of RAWS, called Multi-hop Optimistic Concurrency Control Algorithm (MOCCA), to address the challenges of multi-hop networks. MOCCA improves the performance of RAWS transactions in multi-hop networks while maintaining serializability.

We implement MOCCA in JProwler simulator using TDMA- and CSMA-based MAC layers and compare it against a lightweight locking scheme and serial execution of transactions. Our results indicate that concurrent execution in MOCCA can outperform serial execution in task completion time via better utilization of broadcasts and concurrent execution. Finally, we also show that under heavy loads optimistic concurrency control provides much better performance compared to locking schemes.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed Programming*

General Terms

Algorithms, Reliability, Verification

Keywords

Transactions, optimistic concurrency control, serializability, programming abstractions, wireless sensor networks

1 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. Unintentional and unwanted nondeterministic executions can haunt the correctness of the decentralized algorithms, and 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.

In this study we utilize atomicity of broadcast messages to provide a low-cost single hop primitive with optimistic concurrency control: Read-All-Write-Self (RAWS). RAWS transactions can be used to model a limited form of shared memory operations without sacrificing performance. We limit the extent of transactions to single hop for tighter concurrency guarantees and high performance. Even with this limitation, complicated and highly concurrent algorithms can be implemented using RAWS transactions.

Multi-hop networks introduce additional challenges for concurrency control. The atomic synchronous broadcasts available in single hop are not directly supported in multi-hop networks. Moreover, in multi-hop networks, dependencies among transactions get more complicated. We demonstrate the concurrency challenges of multi-hop networks in Section 3 and present quantitative analysis of these problems in Section 6. Our solution to address these challenges is an incremental, self-stabilizing algorithm: Multi-hop Optimistic Concurrency Control Algorithm (MOCCA).

1.1 Related Work

Programming abstractions for WSNs and ad hoc networks. Several useful programming abstractions have been

proposed for WSNs, including Kairos [10], Hood [28], abstract regions [27], 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, 21] 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.

Programming abstractions for concurrency control. Recently, there has been a lot of work on transactions for mobile ad hoc networks [24, 17, 15, 16, 22, 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, 19, 20]. 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) [25, 11, 12, 23] 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 ours is the Transact work [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. The Transact model suffers from increased traffic, multiple points of failure and complex semantics. The authors also do not address inconsistencies in multi-hop environment. Our 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 multi-hop networks with consistency guarantees.

1.2 Contributions

In this study we provide a light-weight transaction primitive with optimistic concurrency control: Read-All-Write-Self (RAWS). RAWS transactions utilize atomic broadcast nature of radios in WSNs to ensure serializability. Each transaction allows reading variables from single-hop neighbors and modification of local variables. Our RAWS abstraction simplifies development through simpler validation of system properties.

Optimistic concurrency control primitives need special care when applied to multi-hop domain. We identify possible scenarios in which single hop primitives fail in a multi-hop network. We provide a solution to this problem without sacrificing the benefits of optimistic concurrency control. To the best of our knowledge, this is the first study in WSNs for optimistic concurrency control in multi-hop wireless sensor networks.

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. Our results indicate that 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. We provide detailed simulations to support these claims in Section 6.

1.3 Applications

A major application of our transactional primitive is in data aggregation and integration. Due to environmental factors and sensor characteristics, single node measurements are prone to errors. Although data integration techniques at the basestation can be used to filter out noise, this approach wastes energy as data are relayed to the basestation regardless of its quality. Distributed false positive elimination algorithms can address this issue by discarding noisy data. Detecting false positives can be considered as a special case of consensus, where nodes in a locality need to agree whether to report a detection or not. Although specifics of algorithms can be different, all require a non-local operation to include information from other nodes. A consistent, reliable and serializable primitive, such as ours, greatly simplifies implementation of such algorithms.

A more general form of the data integration/aggregation problem is the implementation of a distributed decision tree. Distributed decision tree algorithms are applicable when data needed for computation is substantially larger than the de-

sired output. High throughput sensors including cameras and microphones are unsuitable for raw data streaming to the base station. With these sensors, tasks such as anomaly and intrusion detection, target tracking and classification require local computation. Distributed execution of these algorithms would also benefit from our primitive.

Highly dynamic systems which contain mobile agents and actuators, introduce additional challenges as the environment changes quickly and unpredictably. Allocation of robots to different tasks and tracking multiple mobile targets are examples of such scenarios. Our primitives can simplify implementation of algorithms under such scenarios by reducing the complexity of common intra-node synchronization.

2 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 reads 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.

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
```

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
```

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.

2.1 Conflict Detection

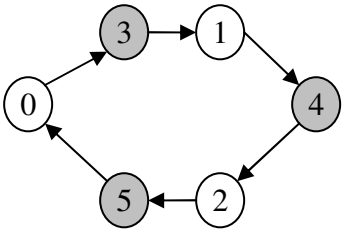
Optimistic concurrency control assumes that transactions will be compatible with each other most of the time. Instead of preemptively preventing concurrency, conflicting transactions are aborted at the time of detection. As long as all conflicts are detected this scheme will be equivalent, in terms of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

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
```

When there is no message loss and all nodes are in single hop, an initiator can run Algorithm 3 to prevent any conflicting RAWS transaction from starting. So in this ideal scenario, all transactions would be serializable. Regrettably, in real life, neither of these assumptions can be taken for



T_0	$S_5 = \{5 \rightarrow 0\}$ $S_3 = \{5 \rightarrow 0\}$
T_1	$S_3 = \{5 \rightarrow 0, 3 \rightarrow 1\}$ $S_4 = \{3 \rightarrow 1\}$
T_2	$S_4 = \{3 \rightarrow 1, 4 \rightarrow 2\}$ $S_5 = \{5 \rightarrow 0, 4 \rightarrow 2\}$
T_3	$S_0 = \{5 \rightarrow 0, 0 \rightarrow 3, 3 \rightarrow 1\}$ $S_1 = \{5 \rightarrow 0, 0 \rightarrow 3, 3 \rightarrow 1\}$
T_4	$S_1 = \{5 \rightarrow 0, 0 \rightarrow 3, 3 \rightarrow 1, 1 \rightarrow 4, 4 \rightarrow 2\}$ $S_2 = \{3 \rightarrow 1, 1 \rightarrow 4, 4 \rightarrow 2\}$
T_5	$S_2 = \{5 \rightarrow 0, 3 \rightarrow 1, 1 \rightarrow 4, 4 \rightarrow 2, 2 \rightarrow 5\}$ $S_0 = \{5 \rightarrow 0, 0 \rightarrow 3, 3 \rightarrow 1, 4 \rightarrow 2, 2 \rightarrow 5\}$

Figure 1. A pathological multi-hop 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. T_i represents the result of transaction initiation by node i and S_i is the set of transaction dependencies known at node i after this transaction.

granted. Message losses are common and networks are usually multi-hop. For mitigating first problem, we employ a topology discovery phase where we identify reliable communication links. By using only reliable links we achieve lower message loss. Having a collision free MAC layer (as in SS-TDMA[1]) also greatly reduces the impact of this problem. The second problem on the other hand requires more deliberation as we discuss in Section 3. We then describe our approach to deal with the multi-hop networks in Section 4.

3 Concurrency Control in Multi-Hop Networks

Multi-hop 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 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 1 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. Table in Figure 1 describes a trace of execution for this scenario where T_i is the transaction of node i and S_i is the set of transaction dependencies known by node i after transmission of corresponding initiation message. 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 2-hop 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 1, 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, as we show in Section 6 we prevent inconsistencies and still achieve substantial concurrency.

A very simple way to avoid multi-hop 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.

4 MOCCA: Multi-hop Optimistic Concurrency Control Algorithm

In order to conceptualize our method of multi-hop 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 2.1, 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 multi-hop 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 detectable.

concurrency The concurrency limitations on the RAWS transactions should be minimal.

For **safety** property, we depend on RAWS to detect conflicts. If all the initiator nodes in a set of concurrent trans-

actions are in single-hop with reliable communication links, all dependency loops will be detected. This corresponds to a clique topology in the graph of reliable links. In such subgraphs assigning different colors to reduce concurrency is not required.

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, Multi-hop 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 **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 not 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 functionality is implemented in *chooseColor*() command. We summarize MOCCA in Algorithm 4.

Algorithm 4 MOCCA

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

4.1 Safety

In this section we show MOCCA provides safety which corresponds to consistent execution of transactions in multihop domain. An inconsistency is a result of cyclic dependency among concurrent transactions. We identify two different kind of cyclic dependencies: clique cyclic dependencies and non-clique cyclic dependencies. Clique cyclic dependencies are a set of transactions with cyclic dependency in which all nodes form a clique in reliable communication graph. When RAWS operates without message losses, clique cyclic dependencies can be prevented by RAWS only. The last node to complete the cycle would detect a conflict with its transaction since it would know all the other transactions in the potential cycle.

Non-clique cyclic dependencies on the other hand contain nodes that are not neighbors. In this case, RAWS might not be able to detect these cycles. MOCCA's aim in safety is to prevent these non-clique cyclic dependencies. Assigning different colors to a pair of nodes is used to forbid concur-

rent execution of dependent transactions on these nodes. The set of transactions with cyclic dependencies would be a directed subgraph of the reliable communication graph. So as long as reliable communication graph does not contain any non-clique cycles composed of nodes from a single color (monochromatic), no non-clique dependency cycles can be generated through execution of RAWS transactions.

Enforcing non-clique monochromatic cycle requirement is difficult using single hop primitives so we use a slightly stronger property:

Property 1: If there exists a monochromatic path between two nodes i and j, i and j must be neighbors.

Following lemma shows the relation of **Property 1** with our goal:

Lemma 1. If **Property 1** is satisfied on graph G, there are no monochromatic, non-clique cycles in G.

PROOF. Assume there is a non-clique monochromatic cycle in the graph $G = (V,E)$ that satisfies **Property 1**. Then exists a pair nodes $i, j \in V$ incident to this cycle where $(i, j) \notin E$. Since i and j are on a monochromatic cycle there is a monochromatic path between i and j. However **Property 1** dictates there can be no such pair of nodes. We arrived to a conclusion, hence the proof is complete. □

Note that the inverse of Lemma 1 is not correct. Monochromatic trees can be formed to span the network which would violate **Property 1** but satisfy the requirement of preventing monochromatic, non-clique cycles.

Lemma 2. Given a graph $G = (V,E)$ satisfying **Property 1**, any serializable execution of MOCCA never violates **Property 1**.

PROOF. Since we only consider serializable executions of MOCCA, we can order execution of MOCCA transactions at each node. Let $C(G,t)$ be the coloring of graph G after t MOCCA executions, and $c_i(t)$ be the color of node $i \in V$ at that instant.

Assume MOCCA violates **Property 1** after execution T. MOCCA only modifies color through RAWS transactions and since RAWS transactions can only modify variables of the initiator node, MOCCA can change only color of the running node $i \in V$ so $C(G,T-1)$ and $C(G,T)$ can only differ at i. Color c of node i can not be same since **Property 1** is satisfied after execution $T-1$ and violated after execution T. This implies i with color c must be part of the newly formed cycle. Moreover i must have a one of the nodes in the cycle not adjacent to all other nodes, as otherwise **Property 1** would not be satisfied after execution $T-1$. This means, there is a path over nodes with color c from node i to a node $j \notin N_c(i)$ through a node $k \in N_c(i)$. However color of node i can only be updated if all neighbor nodes report color c to be *safe*. So node k needs to report color c as safe. A color c is reported safe if color of all nodes $l \in N_c(k)$ is modified before last update of node k and $N_c(k)$
$N_c(i) =$. So we arrive to a conclusion, hence the proof is complete. □

Lemma 2 establishes **Property 1** is not violated by MOCCA. This means as long as MOCCA is started from a configuration of colors that satisfy **Property 1**, through execution of MOCCA safety would be maintained. In the simulations we start with each node with a distinct color based on node ids. This process can be generalized to handle reboots with addition of a small bootstrap process which would let each node to discover colors of its neighbors before deciding on its initial color. The node would in this case start with a color different from all its neighbors.

4.2 Stabilization

Although safety of MOCCA ensures correct execution of RAWS transactions in multi-hop environments, unless MOCCA algorithm terminates the cost of MOCCA algorithm can be prohibitive. In this section we show MOCCA indeed self-stabilizes. We first define a progress indicator $F(t)$ for MOCCA:

$$F(t) = \sum_{v \in V} c(v,t)$$

where $c(v,t)$ is the number of nodes sharing color with node v at time t. Using this indicator we can define an invariant for MOCCA:

Invariant: $F(t) \geq F(t-1)$ for any serializable execution of MOCCA.

PROOF. Each execution of MOCCA changes color of at most a single node. Let v denote the node which changes its color at time t. We can rewrite $F(t)$ as follows:

$$F(t) = F(t-1) - 2c(v,t-1) + 2c(v,t)$$

The factor of 2 comes from the contribution of v to the $c(v,t)$ values of other nodes in his cluster. MOCCA only changes current color if it has higher cardinality then the current color then $c(v,t) \geq c(v,t-1)$. We can rewrite this as $c(v,t) - c(v,t-1) \geq 0$, which implies:

$$\begin{aligned} F(t) &= F(t-1) + 2\left(c(v,t) - c(v,t-1)\right) \\ F(t) &\geq F(t-1) \end{aligned}$$

□

At this point we note that $F(t)$ is bounded from above since the number of nodes is finite. This result does not address update transactions in MOCCA which do not modify colors. Modifications have higher priority over updates so modifications execute regardless of number of updates. When there are no valid modifications left, there are no more color changes possible. Update transactions are initiated only when there is a possibility of color change. So when no more color changes are possible eventually no more update transactions will be initiated. As shown in Algorithm 4, updates are initiated at a node under two conditions: following a neighbor's modification or detection of a suspicious color. As the modifications are completed, the first condition no longer applies and eventually all nodes stop updating. Similarly suspicious colors are result of incomplete neighbor color information in nodes. Following the end of modifications, updates will eventually ensure all the nodes will have up-to-date neighbor color information and all responses from neighbors will either be forbidden or safe.

Although these results show MOCCA stabilizes to a solution we do not claim the global optimality of our solution. A global solution of this problem even using centralized algorithms is NP-complete. However in next Section 6 we show that even this suboptimal solution provide significant performance improvements.

5 Implementation Details

In order to investigate the feasibility of MOCCA we use the Prowler [26] simulator. For performance reasons we use the Java implementation JProwler which offers same radio models with improved scalability.

High concurrency in transactions is one of the main challenges for implementation. Increased concurrency leads to competition for the medium among nodes and transactions. Multi-hop networks further complicate this issue with hidden terminal and synchronization problems. We address this problem through a cross-layer design where we enable our protocol to control the MAC layer behavior for improved performance. RAWS queries the MAC layer for message schedule information which is then used for determining transaction durations. Determining an optimal transaction duration ensures that all read responses will be received before the commit time, while keeping the length of transaction minimum.

Another improvement we make over classic MAC design is the introduction of message queue reordering. In our implementation, read response messages are given priority over new transaction initiations. This process improves throughput substantially as more transactions are able to complete. RAWS protocol can further modify message queue to combine read response messages. This process allows exploiting the broadcast nature of messages to send read reply messages to multiple initiators with a single broadcast. Our simulation results in Section 6 show that even this simple optimization can have significant performance advantages.

Reducing the amount of messages sent is crucial in both throughput and energy efficiency objectives. To reduce canceled transactions, which waste communication as the results are discarded, RAWS intervenes with MAC operation via deferred conflict detections in transaction initiations. Transaction initiators execute conflict detection just before MAC layer sends the packet to radio. This deferred check allows a large portion of the conflicting transactions to be canceled even before their initiation message is transmitted. With this optimization, traffic is reduced and concurrency is improved as conflicting transactions do not interfere with compatible transactions.

Throughput is usually inversely correlated with fairness, where optimizing throughput alone produces unfair utilization of medium. In RAWS protocol however we need to have a baseline fairness for optimal MOCCA performance. In addition, most of the performance benefits of MOCCA/RAWS come from increased concurrency. Approximating fairness leads to more concurrent transactions and better throughput. Our approach for granting fairness is simple and best-effort. We introduce a random back off between transactions similar to CSMA back offs approach. Random back off prevents undesired situations where a single node runs successive transactions starving the rest of nodes. We trade off some of the throughput performance as there are potential gaps between transactions if a single node but we obtain better throughput via increased concurrency.

Another optimization we implemented for MOCCA is the interleaving scheme. Since MOCCA algorithm is incremental, we can control the aggressiveness of optimization for better initial response times. We also noted that first few iterations of MOCCA algorithm correspond to the bulk of performance improvement and the gains diminish with more and more iterations. We use a probabilistic approach to limit the number of MOCCA iterations. MOCCA algorithm decides to iterate or process next data transaction probabilistically. This probability is reduced by 20% after each MOCCA iteration. With this interleaving scheme, MOCCA can still stabilize coloring with enough number of data transactions but when the amount of data transactions is small, the cost of MOCCA transactions do not dominate the total cost of operation.

5.1 TDMA issues

Our TDMA implementation is similar to SS-TDMA[14], where no pair of nodes with two hop distance share the same TDMA slot. Moreover, we minimize the number of slots to improve medium utilization. TDMA is quite suitable for RAWS implementation as it has very low message loss rates. TDMA also corresponds to a natural order for read responses of a transaction. Since the TDMA slots of each node can be known by their neighbors, the optimal transaction duration is easy to compute. The only drawback of TDMA model is the requirement of time synchronization. For this purpose we rely on existence of reliable multi-hop time synchronization algorithms such as FTSP [18].

5.2 CSMA issues

CSMA on the other hand does not require time synchronization. This advantage is offset by the heavy traffic caused by RAWS. CSMA suffers from lack of coordination when medium is flooded with packets. CSMA implementation also needs to deal with ordering read response messages for a transaction. We evenly distribute read responses to transaction duration in CSMA to prevent collisions between read responses to same transaction. However as show in Section 6, CSMA still has large amount of message losses and canceled transactions.

5.3 Optimal Serial Execution

When clock synchronization is available, computing transactions in a serial token passing scheme is a viable alternative. In this study we opted to compute the lower bound for this approach instead of actual implementation. Token passing with a single token can be quite inefficient in multi-hop as it fails to utilize potential for concurrent transmissions. Thus, we opt for multiple tokens passing through network for optimal serial execution. We devise minimum duration rounds where nodes with tokens are allowed to run one transaction and then they pass their tokens to next set of nodes. For assignment of rounds and nodes in each round we utilize TDMA slot assignment. TDMA slot assignment makes sure there are no nodes in two-hop neighborhood sharing the same slot. In optimal serial execution, we replace TDMA slots with optimal serial execution rounds. Thus given a topology and TDMA slot assignment, the behavior of serial execution can be predicted. Transaction completion rates and simulation durations for serial execution is computed in this manner taking read response delay in to account where applicable.

5.4 Transaction Generation Models

The set of transactions required for different applications can be substantially different. Since addressing all possibilities is infeasible we employ certain methods for generating transactions. Our initial approach is generating transactions with uniform random read set size and members of the read set again chosen randomly among neighbors. Maximum read set size in this method is determined to be the number of neighbors of the node. This method generates both very small and very large transactions corresponding to a wide range of different tasks.

A similar approach was taken in [5], where the transaction read sets were generated by uniform probabilities. Their method adds each variable to read set with 0.5 probability, which leads to more average sized transactions and less very small or very large transactions.

Keeping the transaction size constant is another option. This method also leads to a further optimization in optimal serial execution since the size of transactions are known beforehand.

Finally we use a data aggregation task to model transaction generation. In this model transaction read sets are only allowed to contain nodes with lower ids then the initiator. This process leads to an acyclic dependency graph for transactions. This transaction model is quite similar to the transaction sets required for false positive elimination and distributed decision tree implementations.

5.5 Simulation Setup

Our simulation uses a square shaped region with varying size and number of nodes. Figure 2 show one sample topology with 100 nodes in a $100m \times 100m$ region. We used a uniform random distribution to generate topologies in this study. Large circle in Figure 2, corresponds to approximate reliable communication radius which varies due to simulation.

We used the Gaussian interference model in JProwler. The interference in this simulation has a static and a dynamic component. Static component reflects multi-path effects and reflections in a link and does not change after the topology is constructed. Dynamic component on the other hand models the transient effects in channel quality common in low-power radios. While deciding on success of reception, interference of transmissions are accumulated to a total noise strength and compared against received signal strength.

A separate neighborhood detection run is made before the experiments to identify reliable communication links. The neighbor detection phase is composed of two sub-phases. First each node makes 10 transmissions and each node keeps track of the number of receptions from each transmitter. In the second sub-phase, nodes declare their list of reception counts for transmitters. A link between a pair of nodes is identified as reliable if all transmissions in both directions are received successfully. This phase is run in a very high granularity TDMA manner where only a single node is allowed to transmit in whole network. During RAWS execution multiple concurrent transactions are present, which leads to increased interference and message losses even in these reliable links.

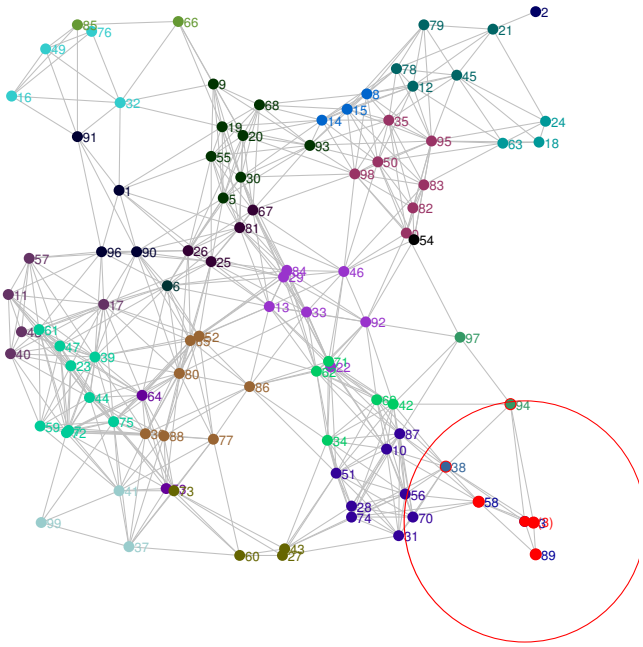


Figure 2. A sample topology used in simulations. Nodes are colored according to MOCCA algorithm. Circles show location of nodes and edges show the reliable communication links.

Our implementation keeps a detailed log of transactions executed for detecting inconsistencies. We also visualize these data for a better perspective on execution of algorithm. Figure 3 shows one snapshot from execution of MOCCA with CSMA model.

Unless otherwise noted we repeated experiments for each configuration 20 times with different random number generator seeds. For clarity we report the median value for metrics. We also include costs associated with MOCCA in our experiments. MOCCA transactions are executed concurrently with the data tasks and the total duration and message communication figures include extra messaging caused my MOCCA algorithm as well. We do not consider RAWS transactions used for MOCCA algorithm as data transactions since they are intended to improve performance rather than actual work. So the transactions used for tasks are same for all methods but MOCCA experiments include additional work for improving coloring.

We believe visualization is an important tool for discovery, implementation and analysis of algorithms, especially in complex highly concurrent environments such as WSANs. In our experience, the effort spent on meaningful visualization of execution is well deserved. For instance, we managed to fix a rare non-determinism in JProwler implementation, which stemmed from events at exactly same simulation time, using the visual representations similar to the one shown in Figure 3.

6 Simulation Results

This section presents our results on safety, scalability, and performance of RAWS/MOCCA system in multi-hop networks. We start with safety as it is the core competency for

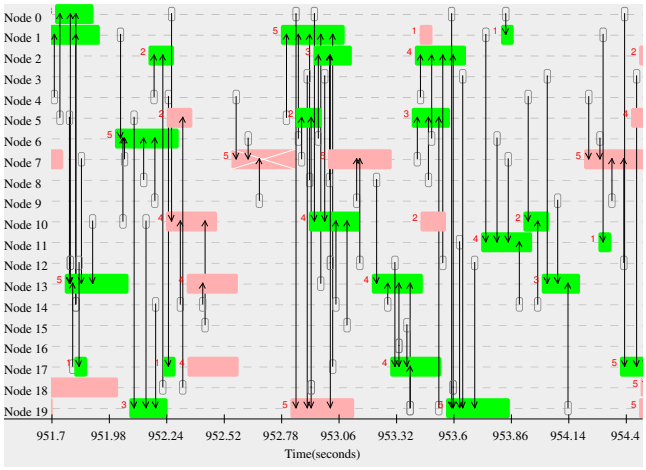


Figure 3. A sample execution from simulation using CSMA. *x* axis is time and *y* axis contains a row for each node. For clarity only 20 nodes are displayed. Filled boxes are used for RAWS transactions initiated by corresponding node with size proportional to the transaction duration. Empty boxes correspond to nodes sending a read response with arrows pointing to the target transaction. Dark (green) boxes are successful transactions whereas light (pink) boxes are failed transactions. A cross on the transaction distinguishes failure of a transaction due to a conflict.

our system. We then investigate the throughput and scalability of our system. Finally, we consider the impact of transaction set generation approaches and the processing delay in the transaction on the performance.

6.1 Safety

We first investigate the safety properties. In this part we compare same set of tasks executed concurrently, under three different configurations:

1. No concurrency control
2. RAWS without MOCCA
3. RAWS and MOCCA

Our first metric for safety is the number of inconsistent transactions. A set of transactions are considered inconsistent when they are executed concurrently but their result does not correspond to any serial execution, hence breaking serializability. Figure 4 summarizes the results we obtained from our simulations. RAWS and MOCCA working together makes the system totally consistent, whereas RAWS without MOCCA is still reasonably consistent, the percentage of inconsistencies is less than 1% compared to no concurrency control which leads to almost 60% inconsistency.

At this point we note that even a single inconsistent transaction is sufficient to cause remainder to be eventually inconsistent as well. Inconsistent transactions would corrupt state of a node, which in turn can corrupt states of neighboring nodes through other transactions. We thus investigate the ratio of runs that contain no inconsistencies to all runs. Figure 5 demonstrates this perspective with increasing number of transactions. Even with 60 transactions per node

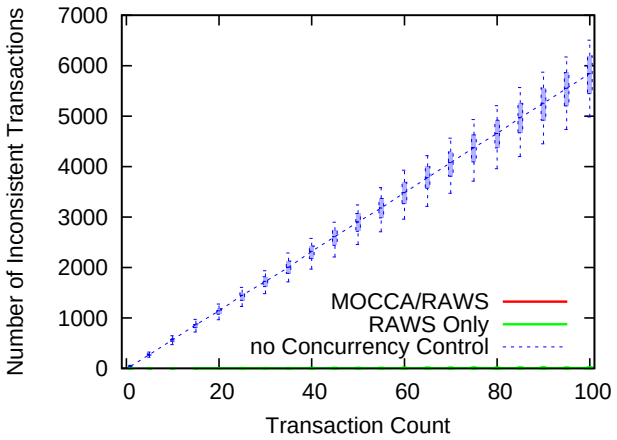


Figure 4. Number of inconsistent transactions versus the number of transactions per node.

RAWS alone causes inconsistencies in all runs. Introduction of MOCCA on the other hand prevents this from happening as we do not observe any inconsistencies with MOCCA.

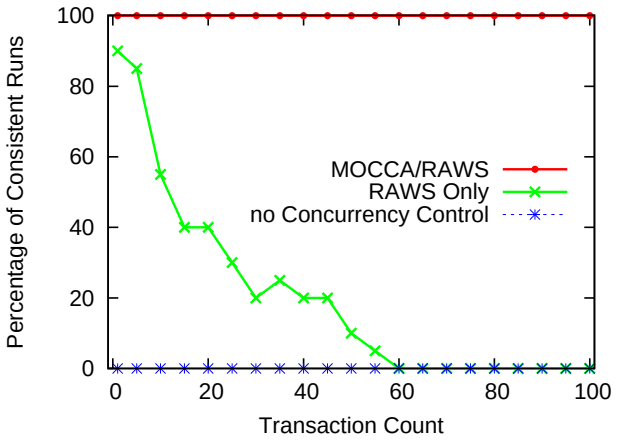


Figure 5. Percentage of consistent runs versus number of transactions per node.

Through the rest of experiments with a total of 2400 runs, we only had a single inconsistency while using MOCCA with CSMA. Total consistency in presence of byzantine message losses is impossible but we argue that our scheme reduces the probability to much more acceptable ranges especially with TDMA.

6.2 Throughput

In this subsection we investigate throughput performance of RAWS/MOCCA. High throughput improves effectiveness of not only heavy traffic but bursty traffic in the network. With high throughput more traffic can be handled by the primitive reducing the need for using more simple primitives. Multi-media networks with cameras and microphones generate large amounts of data to be transferred resulting in heavy traffic. Bursty traffic patterns are more common in WSAN tasks since sustained high throughput is difficult due to battery limitations. Applications such as intruder de-

tection and false positive elimination, cause in such traffic patterns. These tasks have high spatio-temporal correlation among sensor detections which translates to high correlation in communication. By increasing throughput we also improve the system's responsiveness to bursty traffic.

Figure 6 summarizes a single run of all protocols in a reference configuration with 100 nodes in a $100m \times 100m$ region and 400 transactions per node. This figure depicts change of completed transactions per minute through the run. With TDMA, MOCCA initially has a lower rate than Locking but this difference is quickly compensated. CSMA on the other hand shows much less difference between Locking and MOCCA.

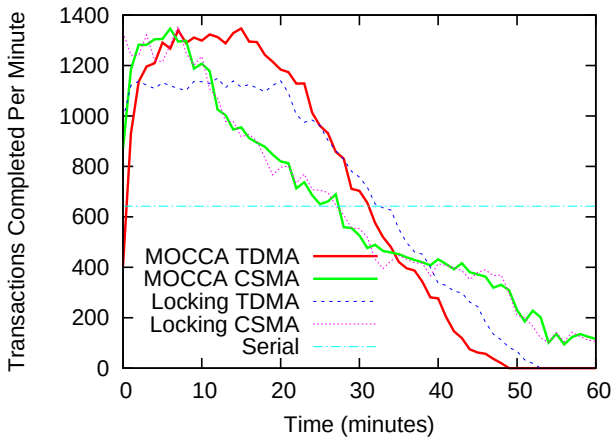


Figure 6. Transactions completed per minute versus time from single representative runs using same transaction set.

The main reason behind the drastic difference between CSMA and TDMA is the ratio of canceled transactions to the total number of started transactions. Figure 7 shows the extent of this problem where CSMA has significantly higher cancel rates especially for transactions with large read sets.

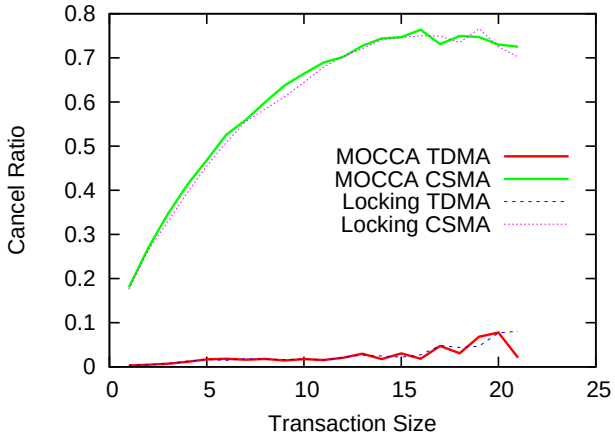


Figure 7. Ratio of canceled transactions to started transactions, grouped by number of nodes in read set. For this figure a topology with 100 nodes is used with 1000 transactions for each node.

The increased cancel rate also inversely affects the total duration of a single transaction. Since failed transactions are repeated until success, successive failures reduce performance. We observe the total duration distribution of RAWS transactions with TDMA (Figure 8) has much lower variance than RAWS transactions with CSMA (Figure 9).

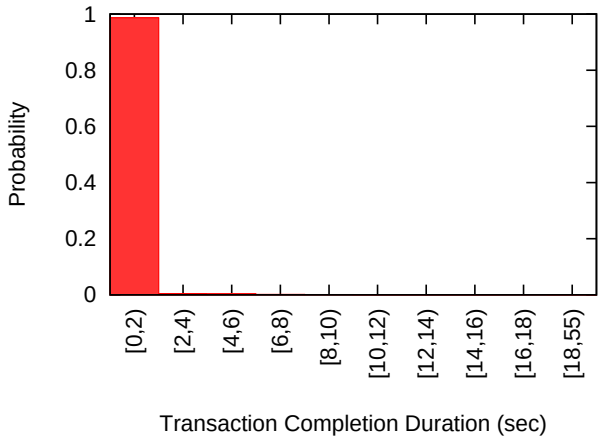


Figure 8. Distribution of completion times for individual RAWS transactions in MOCCA with TDMA.

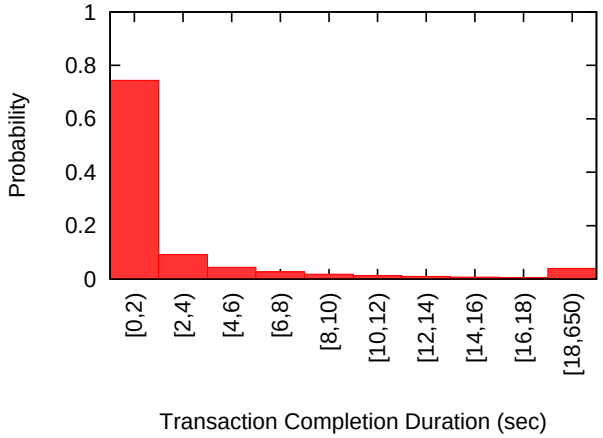


Figure 9. Distribution of completion times for individual RAWS transactions in MOCCA with CSMA.

These results point out to the limitation of CSMA for collision free communication in multi-hop environment. Heavy traffic from RAWS transactions also amplify this problem.

6.3 Scalability

An important question about the performance of MOCCA is the scalability of the method for larger networks. Number of nodes in the network is a natural parameter to consider for scalability. Figure 10 shows our results for this parameter. This result is expected since increased number of nodes also increase number of single hop neighborhoods, which in turn increases the possible concurrency in the network.

Density of the nodes on the other hand has more profound consequences. Figure 11 summarizes the effect of density

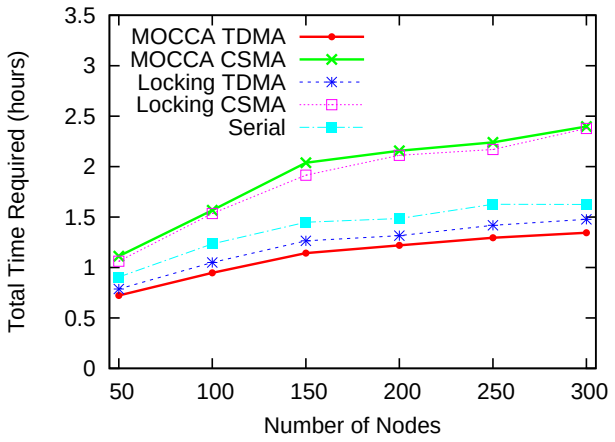


Figure 10. Total Simulation time versus the number of nodes under constant node density of 0.01 nodes$/m^2$.

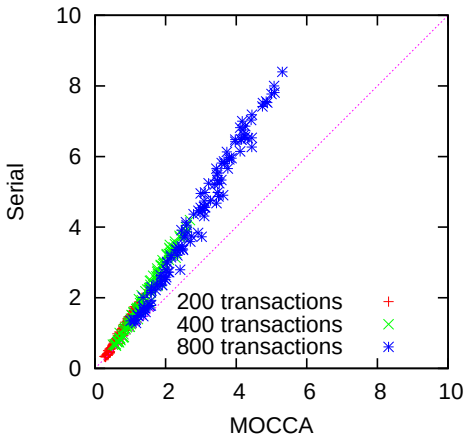


Figure 13. Comparison of corresponding MOCCA and Serial executions with respect to simulation time.

with varying number of transactions. Increased density corresponds to increased transaction sizes and increased chance of conflicts and cancels. This effects MOCCA and Locking similarly increasing the required time and messages required for completing all tasks. We observe the problems in CSMA more clearly in this perspective as serial execution is much better especially when the density of the nodes are large.

Even with the additional cost of MOCCA transactions, MOCCA achieves performance of Locking even with 200 transactions per node. With increased number of nodes the difference becomes more significant. Figure 12 shows this result comparing total simulation durations of individual runs corresponding to same transaction set in a scatter plot.

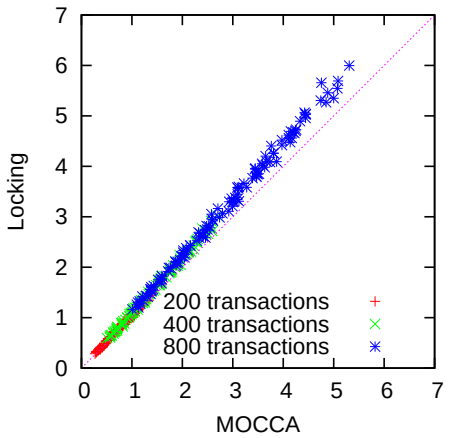


Figure 12. Comparison of corresponding MOCCA and Locking with respect to simulation time.

A similar comparison in Figure 13 between MOCCA and optimal serial execution shows even larger gap between two methods. Variance in these figures stem from both the randomness of the topology and the different densities employed.

For a better understanding of throughput we ran methods with unlimited number of transactions for a fixed duration. Figure 14 shows results of this experiment. This figure exposes a rather curious phenomenon where the average transactions completed with CSMA is much higher than TDMA when the density is low. CSMA allows transactions to be started at a faster rate as nodes can start transactions any time instead of waiting for their TDMA slot. However as we have shown in Figure 9 there is a large variance in transaction durations in CSMA. Even when CSMA can execute more transactions per minute, for the total completion time metric, the last transaction to be completed is the determining factor.

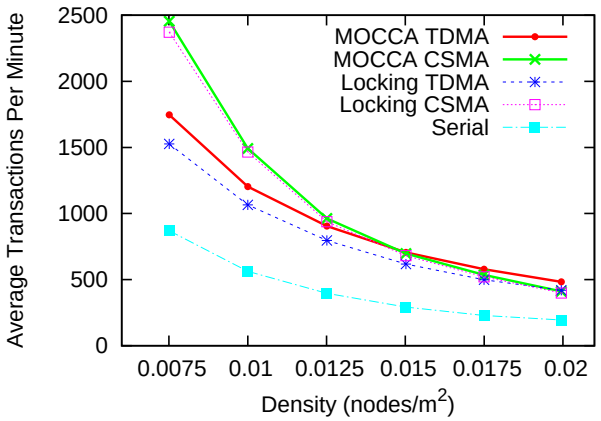


Figure 14. Average number of transactions completed per minute for different methods. Each node is allowed to run as many transactions as possible and after 2 hours of operation the average is calculated.

6.4 Transaction Set Generation

Different computation types can lead to different set of transactions. In this section we investigate such scenarios. Uniform Random transaction sets in Figure 15 is our reference model. Coin flipping model used by authors in [5] provides very similar results to our model as shown in Figure 16.

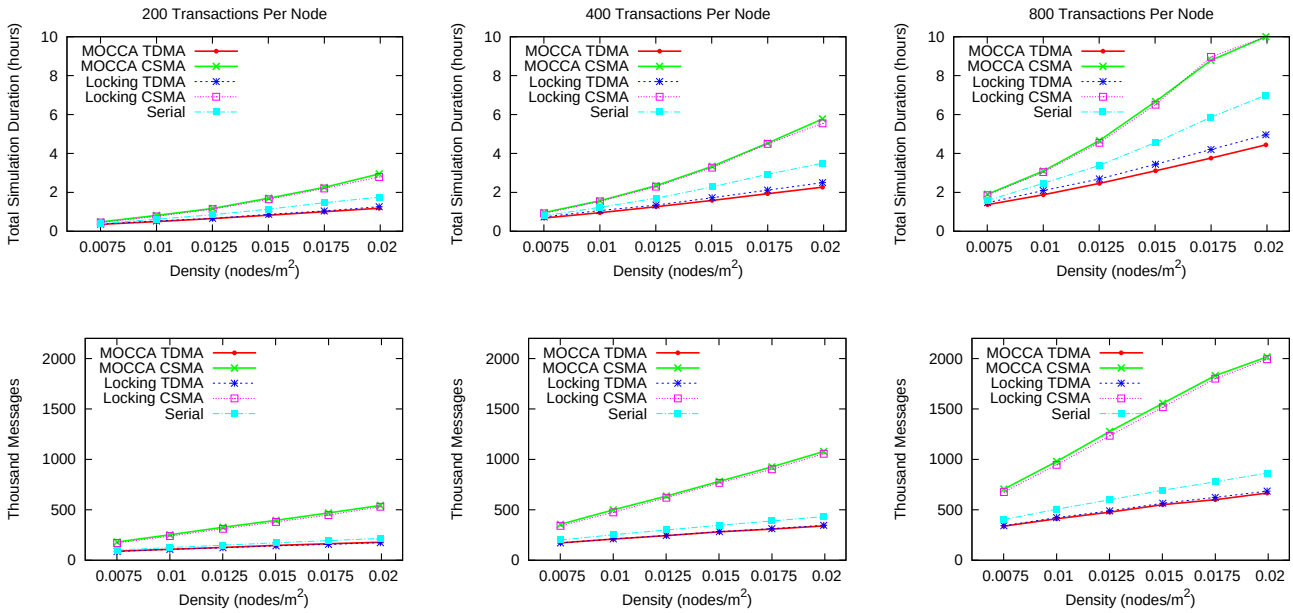


Figure 11. Effect of number of transactions and density of nodes on task completion time and number of messages sent.

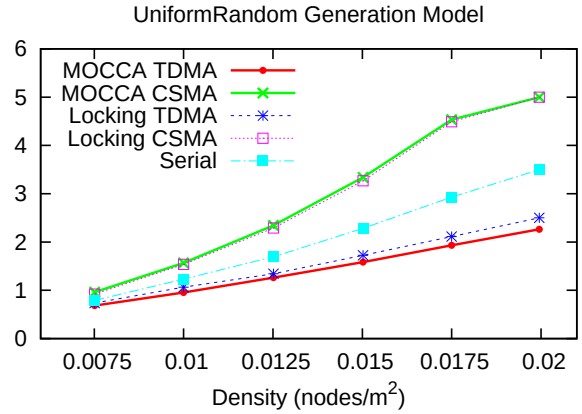


Figure 15. Performance of Uniform Random tasks with respect to simulation time with 400 transactions per node.

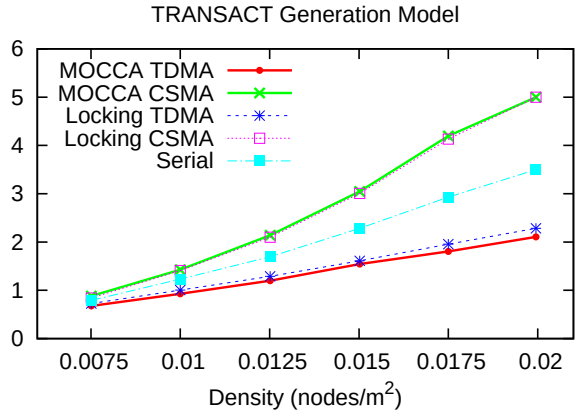


Figure 16. Performance of TRANSACT tasks with respect to simulation time with 400 transactions per node.

Data aggregation problem has a slanted distribution of transaction set sizes with many transactions with small read sets. This benefits MOCCA and Locking similarly as both methods can utilize variable sized transactions. This model also has many compatible transactions because aggregation creates a directed acyclic graph structure for transaction dependencies. Hence Figure 17 shows significant difference between concurrent methods and serial execution even with the disadvantages of CSMA.

A constant size for transactions reduces the benefits of concurrency as more and more transactions become conflicting. In addition, the serial protocol can be further optimized by reducing superframe size. Figure 18 uses such optimized serial algorithm and shows a scenario where serial execution might be faster.

6.5 Impact of Processing Delay

Up to this point we assumed that the read operations on the contributing nodes can be performed instantaneously. However, when the data is stored in external devices or when it needs to be obtained on demand (such as in a sensing scenario), a delay is induced. This delay can be well tolerated by concurrent paradigms as the delay in multiple transactions can be overlapped. Additionally, while a transaction is waiting for a read response another transaction might utilize the medium. Our experiments, shown in Figure 19, support this argument, indicating up to 10 times performance difference between the serial execution and concurrent methods.

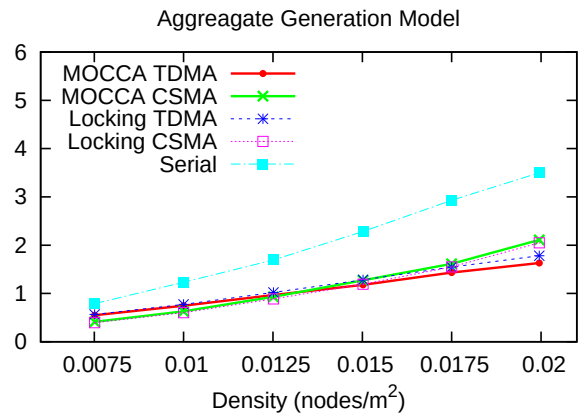


Figure 17. Performance of aggregation tasks with respect to simulation time with 400 transactions per node.

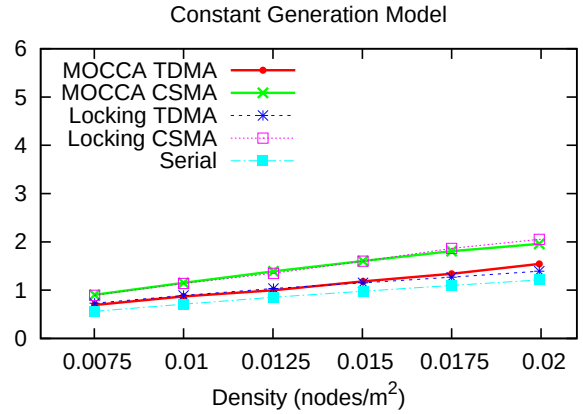


Figure 18. Performance of constant sized random tasks with respect to simulation time with 400 transactions per node.

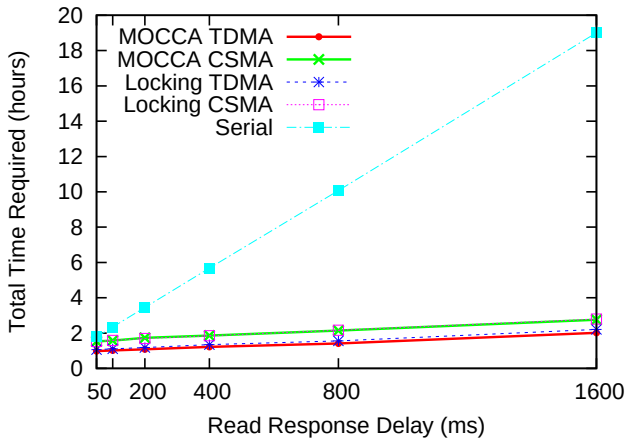


Figure 19. The effect of read response delay on total simulation duration under constant node density of 0.01 **nodes**$/m^2$ **with 400 transactions per node.**

7 Concluding Remarks

In this study 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 single-hop networks. We also identified challenges in implementing our RAWS primitive in a multi-hop environment, and showed that a set of transactions spanning multi-hop neighborhoods may violate serializability. To address this problem, we proposed a constraint based solution, which prevents such multi-hop inconsistency chains. In order to improve the multi-hop 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 Multi-hop Optimistic Concurrency Control Algorithm (MOCCA).

We implemented MOCCA in JProwler simulator with TDMA and CSMA. We compared the performance of MOCCA using these two MAC layers with an optimal serial execution and a locking based protocol. Our results indicate that, when time synchronization is available, MOCCA with TDMA can outperform optimal serial execution in both execution time and number of message transmissions. Even though we do not provide energy use comparisons, this result points out to potential energy efficiency benefits of concurrency, especially with time synchronization.

In absence of time synchronization, MOCCA can still function with CSMA albeit with performance penalties. It should be noted that, without global time synchronization, serial execution of transactions in multi-hop networks would be extremely challenging, if at all possible. We would also like to note that our simulation results considered a setup with only single variable per node. Increased number of variables per node improves concurrency further and boosts the performance benefits of MOCCA.

The implementation of RAWS/MOCCA framework on the mote platforms is our next step. There are other implementations of optimistic concurrency control in wireless sensor networks[5], which further supports the feasibility of this approach. In addition to a TDMA-based implementation, we also plan to investigate more elaborate CSMA implementation that avoids message losses through smarter scheduling of messages. Knowledge about running transactions and potential read response messages can be leveraged to improve the transaction success rates when using a CSMA MAC layer.

Finally transactional abstraction can be extended to sensing and actuation mechanisms in the node operations as well. This extension would provide a uniform interface for programming WSAN applications where many race conditions can be eliminated. A uniform interface would also simplify validation as all computing would be reduced to set of transactions.

8 References

[1] M. Arumugam and S. S. Kulkarni. Self-stabilizing deterministic TDMA for sensor networks. In G. Chakraborty, editor, *ICDCIT*, volume 3816 of

Lecture Notes in Computer Science, pages 69–81. Springer, 2005.

[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. Macroprogramming 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] A. Kozlova, D. Kochnev, and B. Novikov. The middleware support for consistency in distributed mobile applications. *Proc. of the Baltic DB&IS*, pages 145–160, 2004.

[14] S. S. Kulkarni and M. Arumugam. SS-TDMA: A self-stabilizing mac for sensor networks. In *IEEE Press. To appear*, 2005.

[15] 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.

[16] 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.

[17] 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.

[18] M. Maroti, B. Kusy, G. Simon, and A. Ledeczi. The flooding time synchronization protocol. *SenSys*, 2004.

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

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

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

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

[23] 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.

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

[25] 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.

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

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

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