Randomized Dining Philosophers to TDMA Scheduling in Wireless Sensor Networks

Injong Rhee, Ajit C. Warrier,
Dept of Computer Science
North Carolina State University
Raleigh, NC 27695
rhee,acwarrie@ncsu.edu

Lisong Xu
Dept of Comp. Sci. and Eng.
University of Nebraska
Lincoln, Nebraska 68588
xu@cse.unl.edu

***Abstract*— A randomized dining philosophers algorithm is presented for a realistic semi-synchronous model where message delays vary within an unknown bound, and clocks may run at a different speed without any synchronization. In order to predict the unknown bounds, the algorithm employs a simple network delay measurement technique. The algorithm has an expected running time and message complexity of $O(\delta)$ with high probability. δ is the maximum number of contenders for a process in the system (while $\delta \ll n$, n being the total number of processes). A version of the algorithm, called DRAND, is shown to be used for TDMA scheduling or channel assignment for wireless networks. This algorithm is the first scalable implementation of RAND, a commonly used, centralized channel assignment algorithm. The algorithm can also be used for distributed graph coloring in a semi-synchronous environment. Given any general graph, the algorithm produces a chromatic number up to $\delta + 1$ (in this case, δ is the maximum number of edges). Compared to existing algorithms on distributed graph coloring in the PRAM model, the algorithm can generate equal or less number of colors; often, far less than $\delta + 1$. DRAND is implemented in TinyOS and tested in a real wireless sensor network with Mica2 nodes. The experiment shows that DRAND is scalable and robust in a real wireless network setting.**

I. Introduction

The Dining Philosophers (DP) problem is a classical resource allocation problem that formulates a common synchronization need of multiple processes in accessing a set of exclusive resources. More precisely, the DP problem can be defined as follows [19]. There are n philosophers in the system and a fork set $\mathcal{F}$. Each philosopher rotates its state from *thinking*, *hungry*, *eating* and *releasing*. In order to eat, each philosopher needs a fixed set of forks (a subset of $\mathcal{F}$), and it needs to acquire all of them to start eating. We say that two philosophers are *contending* if their fork sets contain a common fork. When finished eating, it releases its forks for use by its contenders or by itself later when it becomes hungry again. No two contending philosophers can eat at the same time. The goal is to minimize the waiting time of hungry philosophers (or known as *response time*). The DP problem captures the type of synchronization and resource allocation requirements commonly arising in distributed systems such as database transaction systems and distributed file systems where multiple processes need to update several data items or files "consistently" at the same time. It has also been applied to finding a TDMA schedule in cellular networks [10], and can be applied to finding unique IDs among neighboring nodes in ad hoc networks. There are several elegant deterministic DP solutions (e.g., [19], [27], [3], [6], [9]).

Our work is motivated by two factors. First, recently increased interests in wireless sensor networks have heightened the need for a good MAC (media access control) protocol. If any two RF (radio frequency) nodes within a close range happen to transmit in the same frequency at the same time (which is called *collision*), it significantly degrades the signal-to-noise ratio of the received data. There are two main approaches to this problem, namely *contention-based* and *scheduling-based* ones. A contention-based approach allows nodes to transmit at any time. If collision is detected, it makes transmitting nodes retry after some delays. A scheduling-based approach is to schedule the transmission of neighboring nodes by assigning different time slots or frequencies to any two neighboring nodes whose transmission may interfere. Before having a schedule in place, nodes may communicate with each other using a contention-based MAC protocol (e.g., IEEE 802.11, B-MAC [24]). TDMA follows the latter approach. The problem of finding a schedule in TDMA (or any scheduling schemes) can be modeled by the DP problem since any two nodes in an interference range can be viewed as sharing a fork.

Second, existing DP solutions are still too complex for an implementation in wireless networks involving nodes with limited resources (such as sensors). The best known, asynchronous DP algorithm is by Choy and Singh [9] and has a response time of $O(\delta^2)$ where δ is the maximum number of contending philosophers at any time. The response time of the scheduling algorithm is important because wireless nodes can move (although not as dynamically as in mobile ad hoc networks), run out of power or become damaged and new nodes can be added to an existing deployment. Thus, the scheduling algorithm needs to rerun periodically to reflect the changed topology, and an efficient algorithm saves resources spent in periodic rescheduling. In addition, its code complexity must be minimal because of small memory footprint in some wireless nodes (often within a few kilobytes). From a practical perspective, implementing and debugging a complex distributed algorithm such as [19], [27], [3], [6], [9] in a wireless setting (where communication environments are constantly changing and unreliable) is not trivial.

We apply three approaches to improve the status quo. First,

we use randomization to simplify the implementation greatly and also to reduce the response time and message complexity. Existing solutions, to the best of our knowledge, are deterministic. Second, most of relate randomized algorithms run in the synchronous system where each process runs in a lock step with a fixed message delay (e.g., PRAM). We relax this synchrony by having process speeds and message delays varied within unknown bounds. We also allow processes to have unsynchronized clocks whose rates may drift within an unknown bound. These timing assumptions reflect realistic system environments since processors do not run infinitely fast (or slow) and most data transmissions are received within some number of retransmissions if communicating entities are "wirelessly" connected. A similar model is also defined as a *semi-synchronous* model in earlier studies (e.g., [14], [1], [11], [2]). Third, we use a measurement-based technique where each process passively measures and estimates the message delays. The estimated values are used to improve the performance of the algorithm.

The resulting DP algorithm has expected response time and message complexity of $O(\delta)$ with high probability[1]. The performance of the algorithm also depends on the accuracy with which the network delay measurement can predict the future network delays of the network. As the network timings become more predictable (or synchronous), the algorithm runs faster.

We first present a randomized one-time dining philosophers (ODP) algorithm wherein each philosopher eats exactly once, and then extend the algorithm with the doorway algorithm of Lamport [17] for the full scale DP algorithm. We show that the ODP algorithm is a scalable, distributed implementation of RAND [22], a famous centralized TDMA scheduling algorithm. This is the first of such to the best of our knowledge. RAND models TDMA scheduling as a node coloring problem in a conflict graph where the contention between any two nodes in an interference range is represented by an edge between the two nodes. RAND sorts all the nodes in the graph in a random total order and assigns to each node, in that order, the minimum color (or slot number) that has not been taken by its adjacent, but preceding (by the order) nodes. Since RAND requires the knowledge of the global network topology to get the total ordering, it is not scalable for a large scale network. Yet, RAND is the most commonly used channel assignment strategy because of its simplicity [22]. It is also frequently used as a performance benchmark for other distributed heuristic algorithms (e.g., [29], [5]) whose performance is still inferior to RAND in terms of *maximum slot numbers*. The maximum slot number of a TDMA scheduling algorithm is the maximum number of slots used by the algorithm for an input graph (which is highly related to the utilization of wireless channels because a smaller number means more concurrency among nodes in sharing the channel). The maximum slot number of RAND is $\delta + 1$. Note that obtaining the optimal slot number is NP-hard [22]. Our implementation of RAND is *exact* in the sense that for any input conflict graph, our ODP algorithm can produce any channel assignments producible by RAND but using only local information within $O(\delta)$ time and message complexity; thus we call our ODP algorithm *Distributed RAND* or *DRAND*.

DRAND can be applied to distributed graph coloring. Graph coloring has also been extensively studied in the theoretical computer science community (perhaps less known in the networking community, yet has many networking applications such as ID assignment). The best-known distributed graph coloring algorithms are randomized ones developed for PRAM, a completely synchronous environment (e.g., [18], [16], [15]). In these algorithms, a node chooses a color randomly from a palette of $\delta + 1$ colors and synchronizes with its neighbors to compare its color with its neighbors. If they are different, it gets to keep the chosen color. Otherwise, it has to start again. The best known chromatic number for any general graph achieved using this approach is by Luby [18] and is $\delta + 1$ and its running time is $O(\log n)$ with high probability. However, given the same graph, DRAND can generate in a distributed manner the same or less chromatic number than the palette-based coloring; in most cases, far less. To see this, consider a star-shaped graph with $\delta + 1$ nodes where δ nodes have only one edge to a node in the middle so that the middle nodes have δ edges while the others have one edge. In this graph, the palette-based technique requires up to $\delta + 1$ colors while DRAND (and RAND) always requires only two colors.

In this paper, we present our dining philosophers algorithm and its application to TDMA scheduling. We also implemented DRAND in TinyOS and tested the protocol in wireless sensor networks created by Mote2. We report that the performance of DRAND is suitably scalable and robust to run in a wireless network setting.

II. PROBLEM AND MODEL DEFINITIONS

A. Dining Philosophers Problem

There is a set $\mathcal{P}$ of n processes (or philosophers) in the system and a set of distinguishable forks, $\mathcal{F}$. Each process rotates through the following states in the following order: *thinking*, *trying*, *eating*, and *releasing*, and initially it is in the thinking state. A process always remains in the thinking and eating states for a finite time after which it enters the trying and releasing states respectively. Each process i specifies a fixed set of forks f_i, $f_i \subset \mathcal{F}$ and we call this set, the *fork set of process* i. We say that two processes i and j are *contending* if the intersection of their fork sets is not empty. We assume that each process has at most δ contenders in the system. A process has a priori knowledge on the IDs of its contenders (through some form of neighbor discovery protocols). It is the task of a DP algorithm (by specifying the codes for processes and forks) to ensure that a process eventually enters the eating state once it is in the trying state (the liveness condition) and no two contending processes are in the eating state at the same time (the safety condition). A fork has knowledge of the processes that have it in their fork sets. The *one-time dining philosophers* (ODP) problem is the dining philosophers problem in which each process enters the trying state only once.

Processes can communicate with their contenders and forks only by exchanging messages. A message is identified by a

[1]Asymptotically, the probability that the algorithm requires more than the expected running time or messages is less than $1/e^c$ for some constant c.

source (process or fork) and a destination(s) (process or fork) and its message content. A process can unicast a message to one contending process or a fork in its fork set or broadcast a message to the set of its contending processes or its fork set. A fork can unicast or broadcast a message only to the processes who have them in their fork sets. A process cannot send a message to non-contenders (i.e., in order to communicate with non-contenders, a message must be "routed" through a series of unicasts or broadcast involving intermediate processes or forks). Both unicast and broadcast are counted as one message and the size of each message is bounded by a fixed number of bits. This model captures the notion where contending processes are within a constant number of hops away. Note that broadcast does not incur any additional overhead over unicast, in a radio communication.

Now we define the semi-synchronous aspect of the system. We assume that processes and forks keep (possibly unsynchronized) clocks whose rates may drift at an unknown drift rate ϵ. One *step* of a process or a fork consists of one message transmission and some finite number of "local steps" to change its local states (the idea is the local steps take much less time than data transmission). The time to take a step is bounded in between two unknown real numbers c_1 and c_2 ($c_1 \leq c_2$). Basically, this model captures the notion that some unknown constant bounds the processing speed ratio of any two processes (and forks), which is a quite realistic assumption as no process is infinitely faster than other processes.

Each message is received by its destination(s) within a bounded time. All messages from one entity to another are received in the FIFO order by the destination. We assume that the message delay is always bounded in between unknown real numbers d_1, and d_2 ($d_1 \leq d_2$), the delay of a message includes the time taken for its destination to receive that message, and $d_2 \gg c_2$. The assumption about message bounds might be viewed too strong for wireless networks since radio transmission is typically unreliable due to signal fading and mobility. However, in real networks, packet losses are handled by retransmission, and it is reasonable to assume that any two connected nodes can have their messages delivered to each other within a finite number of retransmissions (if packets cannot be delivered after an infinite number of retransmissions, the two nodes are simply disconnected). **We use bounded delays only for the analysis of the algorithm because the performance of any system without real-time bounds cannot be specified in real-time. Intuitively, any algorithm that works in the unknown bound model also works correctly in an asynchronous model where bounds on the delays do not exist, but are finite. This is because the algorithm cannot make use of the specific values of the bounds as they are unknown to the algorithm, and thus, must work correctly with any finite value of the bounds.**

We, however, recognize that one may design a more optimized algorithm without use of a reliable underlying network layer where all messages are acknowledged and retransmitted. So we also describe how our semi-synchronous algorithm can be extended to a weaker model where communication can be unreliable but lost packets can be restored by a finite number of retransmissions (i.e., no bound). In this model, we can specify which messages need to be reliable or can be unreliable without affecting the correctness of the algorithm.

Our model does not capture network disconnection, node failures and asymmetric links. We use this model to make it easy to describe our algorithms. Our approaches to handling these anomalies are discussed in Section VI. More discussion on these practical issues can also be found in [23].

The performance of an algorithm for the DP problem is measured by the maximum real time taken (called *response time*) and messages sent by a process before entering the eating state from the trying state.

B. TDMA Scheduling Problem

We now define the TDMA scheduling problem. A network is represented by a graph $G = (V, E)$ where V is the set of nodes, and E is the set of edges. An edge $e = (u, v)$ exists if and only if u and v are in V and u is in a radio communication range from v and vice versa (i.e., all edges are bidirectional). In this model, a node can unicast or broadcast a message to its adjacent nodes. The real time is divided into a non-overlapping equal time period *time frame* which is divided into the *MaxSlot* number of non-overlapping equal time periods, called *time slots*. The slots are numbered from 1 to *MaxSlot*. We assume that *MaxSlot* is sufficiently large to handle all the assignment strategies for an input graph. Informally, the objective of the scheduling is that each node picks a time slot during which it can transmit without "conflict". We say that two nodes u and v are in *conflict* if and only if u and v are in one or two hops away from each other. Typically this definition of conflict is used in a broadcast mode of TDMA scheduling where any two nodes within a two hop range can have a radio interference at some node in their transmission ranges due to the *hidden* terminal problem [22] and their radio broadcast transmission causes that node to receive a degraded signal. A list of conflict relations in wireless networks is discussed in [22]. We discuss in Section IV how this definition of the scheduling problem can be mapped to the DP problem defined above.

We define the *TDMA scheduling problem* to be a problem of finding a time slot for each node, given an input graph G, such that if any two nodes are in conflict, they do not have the same time slot. This problem is often known as the *static channel assignment* problem or *reuse channel assignment* problem. After each node finds its slot, it (re)uses that slot at each time frame for collision-free data transmission. Thus, an algorithm that minimizes the number of time slots being assigned allows the system to minimize the frame size (originally set to *MaxSlot*), thus increasing channel utilization. After the channel assignment, the maximum time slot being assigned in the network must be broadcasted to the entire network. Our definition of the TDMA scheduling problem deals only with channel assignment part. We assume that synchronizing with all the nodes on the maximum slot number is not counted for the cost of the scheduling algorithm.

In practice, effective use of a TDMA schedule may require strict time synchronization. But we do not assume time synchronization for obtaining the schedule. This may seem odd because time synchronization is an implicit assumption of

most TDMA systems so it should come free for the scheduling. But wireless nodes, especially sensor nodes, often come with very limited battery and system resources. Frequent time synchronization itself can be of burden to these nodes, especially in a large-scale deployment. Thus, networking solutions that do not require time synchronization are more preferred (even if required, very loosely synchronized so that they don't have to run the synchronization algorithm very often). Also TDMA can still be useful without tight synchronization when using a large time slot (which can afford enough guard time for clock drift). These issues are more relevant in practice and we address them in details in [23].

III. ONE-TIME DINING PHILOSOPHERS

We now present a randomized ODP algorithm. Informally, the algorithm is very simple and runs as follows. When a process becomes trying, it tosses a coin whose probability of getting *head* or *tail* is 1/2. If a node gets *head*, then it runs a local lottery in which its winning chance is set to the inverse of the maximum neighborhood size of its contending processes. If it becomes a winner of the lottery, it sends a "request" to its fork set. When a fork receives a request, if it has not granted its fork to another process, then it sends a "grant" for its fork to the sender of that request; otherwise, it sends a "reject" message back to that process. If a lottery winner receives grants from all of its forks, then it starts to eat, and after finishing eating, send a "release" to all the forks. When receiving a reject, it also sends a release to any forks it has received grants from. If a trying process has not won the lottery, or won it before but lost it because of rejection, it tries the coin toss again after sometime T where T is some multiple of estimated message delays. The more predictable the system timing in an execution is, the more T gets close to a constant factor to the actual maximum message delays during the execution. T and the probability of winning the lottery determine the performance of the algorithm.

For the formal specification of the algorithm, we use Dijkstra's guarded commands ($B_1 \rightarrow A_1, B_2 \rightarrow A_2, \ldots$) where B_j is a condition and A_j is the sequence of steps which will be atomically executed when B_j becomes true. For more details on guarded commands, please refer to [12]. In the specification, all the variables are local variables of process j or fork k. $state_j$ is the state of process j. We add one more state to the possible states of the process, called *hopeful*, to distinguish a lottery winner that has an outstanding request for a fork. $C(j)$ is the set of j's contenders that have not yet eaten. The variable *granted* in a fork k keeps track of the process that k sent a grant to. Its initial value is -1 indicating that the fork is not granted to anyone yet. An additional message called *fail* is added to notify to its contenders when a hopeful process receives a reject. A message *finish*(i) is added to notify process j that process i has eaten so that it can update $C(j)$ and similarly $C(j)$ messages to update $\max|C(i)|$ for all i in $C(j)$. A variable s is the request sequence number (initially zero) and is being echoed back by a fork for all the messages. T_j is initially set to some constant. Process j runs some message exchanges among forks initially and set T_j to the two times the maximum message round trip time measured. We assume that this initialization and neighbor discovery (to know the IDs of its contenders) are performed before the execution of the following algorithm.

Process j's algorithm:

If ($state_j = trying$ and (time T_j past since its last coin toss) or (the first time to become *trying*)) $\rightarrow$
  Toss a coin. It gets *head* or *tail* with probability 1/2;
  If it gets *tail*, then go to **L**. Otherwise continue;
  $p(j) = 1/(\max|C(i)| + 1)$, $\forall i \in C(j)$;
  Run a lottery with probability $p(j)$;
  if j is a winner of the lottery, then
    t_1 = the current time stamp of j's clock;
    $s = s + 1$;
    Broadcast a *request*(s) message to its fork set;
    $state_j = hopeful$;
  else
  L: set a timer to T_j and go to sleep;
    (it wakes up only when it receives a message or the timer expires).

When (receiving a *grant*(s') and $(s = s')$) $\rightarrow$
  if (($state_j = hopeful$) and (*grant* messages received from all in its fork set from the last time it became *hopeful*))
    $state_j = eating$;
    EAT;
    $state_j = release$;

If ($state_j = release$) $\rightarrow$
  Broadcast a *release*(s) message to its fork set;
  $state_j = thinking$;

When (receiving a *reject*(s') and $(s = s')$ and ($state_j = hopeful$)) $\rightarrow$
  t_2 = the current time stamp of j's clock;
  if $(T < (t_2 - t_1))$ $T = 2(t_2 - t_1)$;
  $state_j = trying$;
  Broadcast a *fail*(s) message to its fork set;

When (receiving a *finished*(i) message) $\rightarrow$
  if i is in $C(j)$ then
    $C(j) = C(j) - \{i\}$;
    Broadcast *updateNeighbor*$(j, |C(j)|)$ to j's contenders;

When (receiving an *updateNeighbor*(i, k) from i) $\rightarrow$
  if $k < |C(i)|$, then update $|C(i)| = k$

Fork k's algorithm:

When (receiving a *request*(s) from a process j) $\rightarrow$
  if ($granted \neq -1$)
    send a *reject*(s) message to j;
  else

send a *grant*(s) message to j;
$granted = j$;

When ((receiving a *fail*(s) message from j) and
($granted = j$)) $\rightarrow$
$granted = -1$;

When (receiving a *release*(s) message from j) $\rightarrow$
$granted = -1$;
Send a *finished*(j) to all of the processes
that have fork k in their fork set;

Now we analyze the ODP algorithm.

Theorem 1: (*Safety*) No two contending processes eat at the same time.

Proof: A process needs to receive a *grant* message from all of its forks. Once a fork sends a *grant* message to another process, if it has not received a *fail* or *release* message from that process, it does not send a *grant* message again. A process sends a *fail* or *release* message only when it is not in the hopeful or eating state. Therefore, when a process receives a *grant* from a fork, no contenders of its own are in the eating state. Since the fork does not send another *grant* until the process that currently received the earlier *grant* from the fork is outside of the eating state, no two contenders are in the eating state. ■

Below we analyze the running time of the ODP algorithm. To quantify the running time, we divide an execution of the algorithm into *rounds*. We define a *round* of a process j to be the time period between two consecutive coin tosses by j (i.e., T_j) or the time period for j to perform the last coin toss, finish eating and release its forks.

As indicated in the introduction, the performance of the algorithm depends on the predictability of the network. More precisely, it depends on the accuracy of T_j in predicting the time period from the time that one of j's contenders tries its last lottery to the time that it releases its forks after eating. If a process enters the eating state, the maximum time duration from the last time when it became hopeful until the release message is sent can be bounded by a constant K where

$$K = c_2 + d_2 + c_2 + d_2 + c_2 = 2d_2 + 3c_2 \quad (1)$$

Eq. (1) accounts for the followings: (1) one step to run the lottery and send a request (first c_2), (2) one message delay for a request to arrive (d_2) and one step to send a *grant* message (c_2), (3) all grant messages from its contenders to arrive (d_2), (4) one step to eat and send the *release* message (c_2). Note that c_2 and d_2 are (unknown) constants in our model.

The maximum number of times a process can try the lottery during K can be bounded by Δ where:

$$\Delta_j = K/T_j \quad (2)$$

T_j is set to 2RTT where RTT is an estimate of the round trip message delays measured by j's clock. If we assume that $d_2 \gg c_2$, although T_j is updated during the execution of the algorithm, T_j is always less than or equal to $2(d_2 + \epsilon \cdot d_2)$, and bigger than or equal to $2(d_1 - \epsilon \cdot d_1)$ by the definition of the model. So since d_1, d_2, and ϵ are (unknown) constants, Δ_j is bounded by a constant. If the message delays are predictable, then Δ_j will be a small constant. Let Δ be the constant upper bound on Δ_i for all contenders i of j.

Theorem 2: (*Liveness*) The expected number of rounds during which a process remains in the *trying* and *hopeful* states is less than $2(\delta+1) \cdot e^{0.5\Delta}$, and the probability that it remains in those states longer than some constant factor c times the expected, is less than or equal to $1/e^c$.

Proof: Consider the first round of process j. During the round, a contender i may try the coin toss and lottery for at most Δ times. Let $coin(i,k) = head$ denote the event that contending process i gets a head, when it flips a coin for the k^{th} time in that round. Let $L(i,k)$ denote the event that process i wins the lottery for the k^{th} try. Note that if the event $L(i,k)$ is true, then $coin(i,k) = head$ is also true.

Process j eats in its first round if it wins the lottery at the first try and finishes eating while no other contenders win the lottery during that round. Since contenders i can have at most Δ lottery tries during the period that j has that round, the probability $Pr(j\ leave)$ that process j finishes eating in the current round is bounded as follow.

$$Pr(j\ leave) \geq Pr(L(j,1)) \prod_{i \in C(j)} \prod_{k=1}^{\Delta} (1 - Pr(L(i,k))) \quad (3)$$

$$\geq \quad Pr(L(j,1)) \prod_{i \in C(j)} (1 - Pr(L(i,\Delta)))^{\Delta} \quad (4)$$

$$\geq \quad \frac{1}{2(\delta+1)} \prod_{i \in C(j)} (1 - \frac{1}{2(|C(j)|+1)})^{\Delta} \quad (5)$$

$$= \quad \frac{1}{2(\delta+1)} (1 - \frac{1}{2(|C(j)|+1)})^{|C(j)|\Delta} \quad (6)$$

$$> \quad \frac{1}{2(\delta+1)} (\frac{1}{\sqrt{e}})^{\Delta} \quad (7)$$

Eq. 4 is because $Pr(L(i,\Delta)) \geq Pr(L(i,k))$ for $1 \leq k \leq \Delta$. Eq. 5 is because of the followings: (1) $Pr(L(i,\Delta)) \leq 1/(2(|C(j)|+1))$ because contender i uses, in setting p_i, the inverse of the maximum of the neighbor sizes of its contender set which includes j (so at the minimum, $|C(j)|+1$), and $C(j)$ does not change while j wins the lottery to finish eating, and (2) $Pr(L(j,1)) \geq 1/(2(\delta+1))$ because δ is the maximum contender set size of any node in the network. Eq. 7 is because $(1 - \frac{1}{2(|C(j)|+1)})^{|C(j)|} > 1/\sqrt{e}$.

The above analysis gives the lower bound of the probability that process j eats in any round because there is no special treatment because of the first round, and the bound conveniently depends only on constants δ and Δ, i.e., the lower bound is fixed for every round.

Let M be the random number representing the number of rounds before a process eats with the lower bound probability. M clearly has a geometric distribution.

$$Pr(M = k) \quad = \quad P_{low}(1 - P_{low})^{k-1} \quad (8)$$

where $P_{low} = \frac{1}{2(\delta+1) \cdot e^{0.5\Delta}}$.

From the above, we can obtain the upper bound on the expected number of rounds that a process remains in the trying and hopeful states as follows.

$$E[M] = \frac{1}{P_{low}} = 2(\delta + 1) \cdot e^{0.5\Delta} \quad (9)$$

Finally, we calculate the probability that a process in the trying and hopeful states longer than the upper bound on the expected.

$$Pr(M > c \cdot E[M]) = \sum_{k=c\cdot E[M]+1}^{\infty} P_{low}(1 - P_{low})^{k-1} \quad (10)$$

$$= (1 - P_{low})^{c\cdot E[M]} \quad (11)$$

$$= (1 - \frac{1}{E[M]})^{c\cdot E[M]} \leq \frac{1}{e^c} \quad (12)$$

■

Theorem 3: The expected message complexity of the ODP algorithm is $O(\delta)$.

Proof: In one round, a contender can try the lottery for Δ times. In each try, it can be a winner and gets rejected, thus sending $O(1)$ messages. Therefore, in one round, it can send $O(\Delta)$ messages. Since there are $O(\delta)$ rounds on average, each process can send $O(\Delta \cdot \delta)$ messages on average. ■

IV. TDMA SCHEDULING AND RAND

The TDMA scheduling problem can be mapped to the ODP problem as follows. Each node v becomes a fork and also a process (so there are $|V|$ number of forks and processes). Any two nodes that are in the one-hop distance share their forks, and any two nodes that have a common neighbor will share the fork of that neighbor. Each node needs to receive the forks of its one-hop neighbors and also its own, in order to eat. This mapping allows a node to contend with all the nodes in a two-hop neighborhood (thus matching the definition of the scheduling problem) because any nodes within two hops will be contending for at least one common fork. For a practical implementation, we can make a process and a fork run in the same (wireless) node as threads (or tasks in TinyOS) or combining the codes of the two into one task does not invalidate the correctness.

An ODP algorithm can solve the TDMA scheduling problem by making each process choose, during eating, the minimum time slot that is not assigned to its contenders and notify its contenders about its choice after eating. A process can find the minimum slot because it eats only when no contenders are eating (so mutual exclusion is given). By this mapping, when a process chooses a time slot, its contenders do not select that slot. The resulting time slot schedule is a TDMA schedule. In this algorithm, the order in which processes eat determines the schedule. If any random order is permissible by the execution of the ODP algorithm, then it essentially implements RAND. This is trivially true from the definition of RAND where each unassigned node is chosen in a random order and picks the minimum color not assigned to by its contending nodes.

Our ODP algorithm does allow any random order in eating since the entrance of a process to the eating state is randomly decided by its winning chance of the lottery and odds that all other contenders lose the lottery. We omit the formal proof to save the space. We call the resulting TDMA scheduling algorithm as *Distributed* RAND (DRAND).

V. DINING PHILOSOPHERS

To solve the DP problem using an ODP algorithm, we need to provide a mechanism to handle starvation because a process may repeatedly enter the trying state after eating, possibly preempting the forks of its contenders. We use *the doorway concept* [17], [9], [27]. The doorway concept allows contending processes to set priority based on the arrival to the "doorway". Any contenders who cross the doorway will have a higher priority (multiple processes may do so at the same time) over the processes outside the doorway, and processes outside the doorway need to wait for those contending processes inside the doorway to finish eating. The main idea behind the doorway concept is that once any two contenders find each other outside the doorway, they do not need to check with each other again; thus only when the contenders currently inside the doorway finish, they can enter the doorway to contend for forks. The processes leaving the doorway need to check with all of its contenders before trying for the forks again. This guarantees starvation freedom.

Below we present the doorway algorithm for process j. A set $I(j)$ keeps track of contenders that are inside the doorway, and a set $O(j)$ keeps track of those outside the doorway. The algorithm is merged with the ODP algorithm of process j in Section III to obtain the full scale DP algorithm. To facilitate the merge, we add an additional state, called *pending*, to the process states. After the thinking state and before moving to the trying state, a process enters the pending state in which it executes the doorway algorithm. If a process passes the pending state, it is inside the doorway and changes to the trying state. We omit the formal proof of correctness to save the space.

The doorway algorithm for process j:

When ($state_j = pending$) →
    $I(j)$ = the contender set of j;
    Broadcast a *pending* message to
    its contenders;
When (receiving *pending* from k) →
    If ($state_j = trying$)
        $O(j) = O(j) \cup k$;
    else
        Send an *outside* message to k;
When ((receiving *outside* from k) and ($state_j = pending$) →
    $I(j) = I(j) - \{k\}$;
    If ($I(j)$) is empty
        $state_j = trying$;
If (($state_j = release$) and ($O(j)$ is not empty)) →
    Broadcast an *outside* message
    to processes in $O(j)$;
    $O(j) = \emptyset$;
    Broadcast a *release*(s) message
    to its fork set;
    $state_j = thinking$;

VI. NODE AND COMMUNICATION FAILURES

The practical implementation of DRAND in a real wireless network must deal with several practical, yet important technical issues. These issues arise mainly from packet losses, communication asymmetry, and node and communication failures that our system model does not capture. These anomalies can severely hamper the progress of DRAND possibly causing deadlocks if they are not handled properly. This section discusses our approaches to these issues. In Section VII, we also demonstrate their efficacy by a real implementation and deployment of DRAND in a wireless sensor network.

Our model assumes that all messages are delivered in a bounded time. As indicated earlier, we claim that the correctness of the algorithm is still ensured even under weaker timing models where messages are delivered within a finite time (instead of a bounded time). Note that if a message cannot be delivered between two nodes even after an infinite number of retransmissions, then the two nodes are not connected (i.e., they are not in a one hop distance). (We handle later in this section the case where two previously connected nodes become disconnected during the operation of DRAND.) However, in these weaker models, the running time of the algorithm cannot be bounded in real-time.

DRAND can be further optimized for implementation in an unreliable network by selectively retransmitting only *request* and *grant* when their responses are not received. After sending a *request* message, if a node does not receive *grant* or *reject* from any of its one-hop neighbors, then it retransmits the *request*. Likewise, after senidng a *grant* message, if a node does not receive a *release* or *fail* message, then it retransmits the *grant*. All the other messages can be (re)transmitted in response to these retransmitted *request* or *grant* messages if loss occurs. For instance, if a node receives a retransmitted *request* message, then it reacts the same way as it did when receiving the original *request*. The sequence number s is used to discern whether the *request* message is a new or retransmitted one. In order to reduce duplications of *grant* messages, the retransimitted *request* message may contain the list of those nodes that have responded to its earlier requests.

As communication and node failures are common in wireless networks, it is possible that even these retransmitted *grant* and *grant* messages do not get any response even after many retransmissions. Some links are very unstable and their conditions are highly time-varying. In these cases, although the initial neighbor discovery found them to be within a one-hop distance, their communication quality can get worse during the execution of DRAND. This can cause deadlock or livelock. To handle these situations, we allow nodes to "give up" after some number of retransmissions. Since only *requests* and *grants* are messages that require any response, when a node does not receive any response from a one-hop neighbor for a fixed number of retries, then it removes the neighbor from its neighbor list. This allows the node to make progress with a response from the removed neighbor. This strategy can also be applied to asymmetric links. Consider a situation where a node A considers another node B as a one-hop neighbor, but B does not. While A keeps retransmitting *requests* to B, B cannot respond to A. In this case, A will eventually make progress by dropping B from its neighbor list.

One may argue that although two nodes may not communicate in a stable manner, they might still in an interference range so that TDMA must take care of the interference. Our definition of conflict is limited in capturing this situtation. This interference irregularity can occur even among nodes that cannot communicate with each others at all [28]. These issues reveal the fundamental limitations of TDMA (in a distribtued manner) as it is very difficult, or even impractical, to obtain TDMA scheduling in this network that can completely eliminate any network inteference. To deal with these situations, we claim that TDMA must be combined with other contention resolution schemes such as CSMA. We explore this research direction further in [23].

VII. EXPERIMENTAL RESULTS

Setup. In this section we present the performance of DRAND on simulated and real-life wireless environments. For simulated performance results, we use the Network Simulator [4] while the real-life experiments are conducted on TinyOS running on Mica2 [20] motes. In our TinyOS implementation, we use the default setting of B-MAC (CCA is on, LPL is off, and acknowledgment is disabled) and no clock synchronization. We measure the maximum running time and message counts for each run taken by all nodes in a network to decide on their slots. According to the analysis in Section III, they are linearly proportional to the number of neighbors. In this section, we verify this claim on both single-hop and multi-hop wirless topologies. The source code for NS and TinyOS can be found from http://www.csc.ncsu.edu/faculty/rhee/export/zmac.

Single-Hop Experiments. Our test environment consists of a varying number of Mica2 wireless sensors placed within a one-hop neighborhood. We vary the number of nodes in the network from one node to twenty nodes. Figure 1 shows a snapshot of one test run where 20 motes are located around one mote in the middle. Before the run, we ensure that all

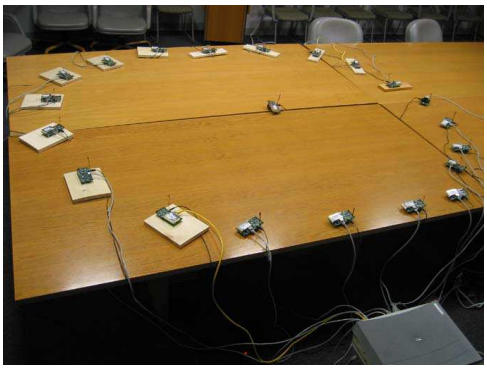

Fig. 1. The one-hop testbed in a conference room with dimension 8 meters by 10 meters. The ethenet cables are used to download programs (but not for the execution of DRAND).

nodes are within a one-hop transmission range and they are also placed at least 2 feet above the ground in our laboratory.

For each setup, we repeat the test ten times and report the average and its standard deviation errors.

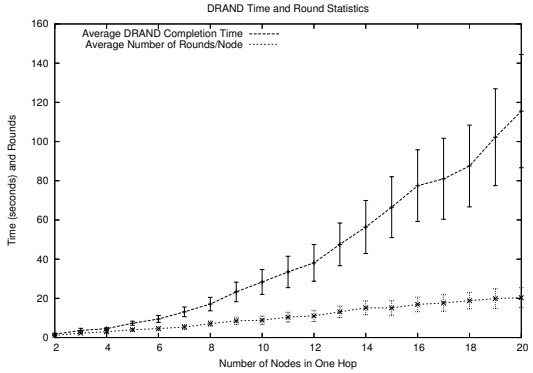

Fig. 2. The running time of DRAND and the number of rounds as the neighborhood size increases in the one-hop topology.

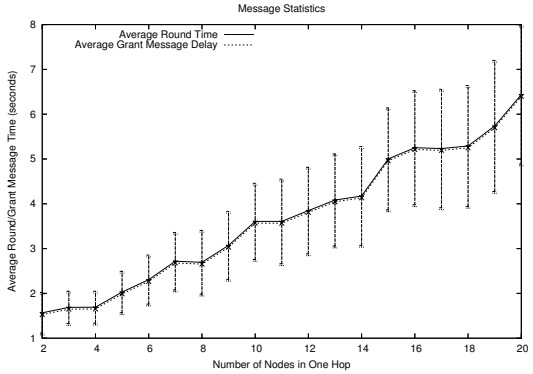

Fig. 3. The average round time duration and grant message delay in the one-hop topology.

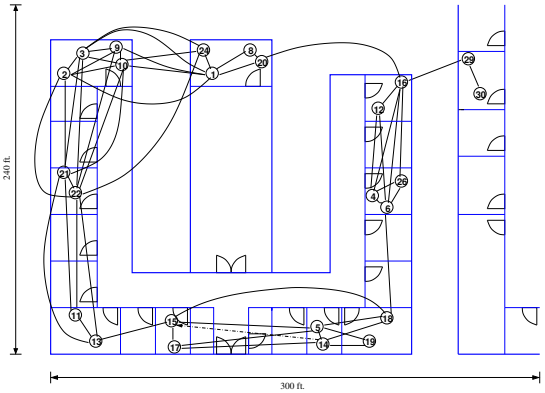

Fig. 4. Real-life wireless sensor network testbed topology. 25 Mica2 motes are placed over two buildings.

Figure 2 shows the average of the maximum running time that a node has taken to decide on its slot and the average of the maximum number of rounds taken by a node in each setup. The error bars denote 95% confidence intervals. Our measurements show that the running time is approximately quadratic with the number of nodes. However, the number of rounds required for each run grows linearly with the number of neighbors following the analysis. Figure 3 shows the average per-round time which, indeed, grows

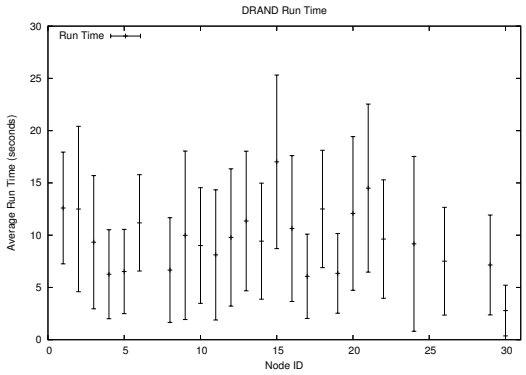

Fig. 5. The running time of DRAND for each node in the testbed

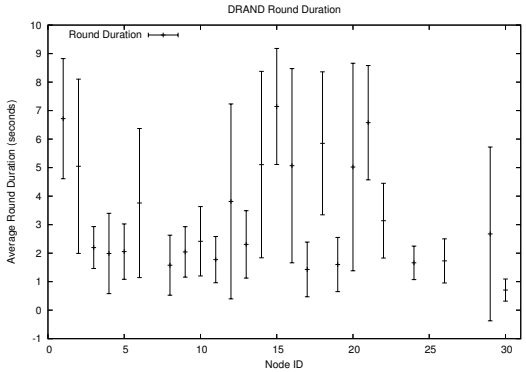

Fig. 6. The average round time duration and the grant message delay.

linearly as the number of neighbors increases. This increase is almost entirely attributable to that in *grant* message delays, shown in a dotted line. This confirms that the discrepancy between the asymptotic analysis in Section III and our measured data comes from the different ways in accounting for a message delay. The asymptotic analysis does not account for any message delay increase due to the increased number of senders. But in practice, wireless communication delays increase proportionally as the number of transmitting neighbors increase because all share the same channel. If the additional message delays incurred by contention is negligible, our result follows the analysis. We don't show any message counts, but report that they are linearly growing as the number of neighbors increaes. In this experiment, no nodes are dropped from the neighbor lists.

Multi-Hop Testbed Results. In this section we look at the performance of DRAND on a 25-node sensor network topology shown in Figure 4. The radio connectivity between two nodes (shown by lines connecting them) varies in quality, with some links having loss rates as high as 30-40%. The DRAND algorithm recovers gracefully from deadlocks caused by such lossy links, as explained earlier in Section VI. In the current implementation, a node drops a neighbor from its neighbor list if it has not responded to 10 consecutive request or grant messages. The experiment is repeated 30 times, and we report the average and standard deviation errors, with 80% confidence intervals.

On this topology, each node first runs a neighbor discovery

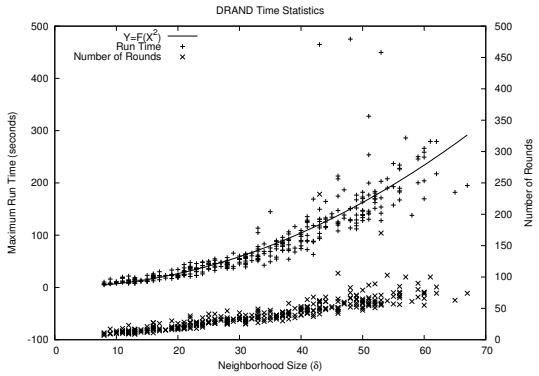

Fig. 7. The running time of DRAND as the neighborhood size increases.

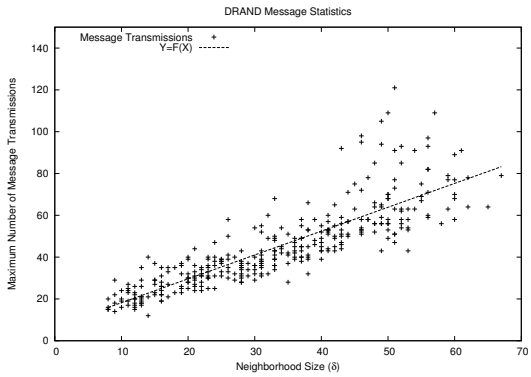

Fig. 8. The maximum number of message transmissions per node during the execution of DRAND.

protocol to get its neigborhood information. We report the DRAND algorithm completion time, and the DRAND round time for each node, excluding the neighbor discovery phase in Figure 5 and Figure 6 respectively. All nodes finish the slot assignments in 20 seconds on average. Nodes with sparse neighborhood (29, 30, 17, 19) have shorter running times in contrast to those with dense neighborhood (1, 21). Some nodes have longer running times despite having sparse neigborhoods (15) and also have larger variances in their running time and round time duration. This is because such nodes have lossy links to their neighbors, requiring retransmissions of request and grant messages. Node 15 for example, discovers node 18 during the neighbor discovery phase, but subsequently dropped it during the DRAND phase when it did not respond to repeated request messages. We found about 4 neighbor list drops in the experiments.

Multi-Hop NS Simulation Results. We now look at how DRAND scales up to large scale wireless networks using the Network Simulator (ns). The network topology consists of nodes placed randomly on a 300mx300m surface. Nodes have a radio range of 40m and a link capacity of 2Mbps. The density of the network is changed by varying the number of nodes from 50 to 250. We use IEEE 802.11 as the base MAC protocol. The experiment is repeated 15 times and the maximum of the values for all nodes are reported. As before, nodes run a neighbor discovery protocol on startup to get their neighborhood information. The maximum DRAND running time and number of rounds for a node with varying network density are shown in Figure 7. The performance closely fits a quadratic curve as in the one-hop test while the number of rounds still grows linearly. As explained earlier, this is because of the linear increase in the message delays as the neighborhood size increases as is seen in the Figure 3. Figure 8 shows the maximum messages sent by a node in each run. It also linearly grows with the number of neighbors following the analysis.

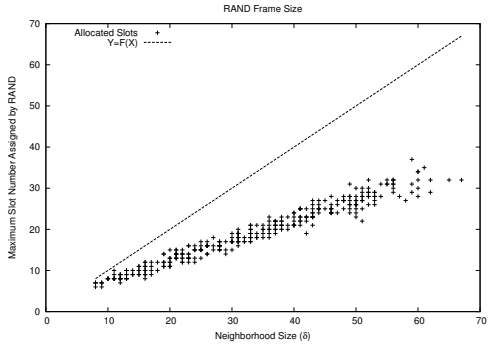

Fig. 9. The maximum number of time slots being assigned by DRAND for input graphs with various densities.

In DRAND, the number of time slots being assigned is bounded by $\delta+1$. But in practice, the number of time slots that DRAND assigns can be far less than that. Figure 9 shows the number of slots used for input graphs with various densities. The dotted line indicates δ. Each data point represents the maximum number of time slots being assigned by DRAND for a different network. In all runs, the maximum slot number used by DRAND is far less than $\delta + 1$. This is in contrast to the performance of other algorithms such as [5], [29], [25], [8] and [18] that require $\delta + 1$ or more.

VIII. RELATED WORK

Dijkstra [13] first modeled the resource allocation problem as a ring of 5 processes, called the dining philosophers, where each process shares a resource with each neighbor. Later, Lynch [19] generalized the problem to an arbitrary conflict graph where a node represents a process and an edge represents a sharing of resources between two processes. Lynch's solution uses an edge coloring algorithm to set a partial ordering on the shared resources, so that each process requests its needed resources in that order. The response time is $O(c^\delta)$. The message complexity is $O(\delta)$. Styer and Peterson [27] extended Lynch's algorithm to develop a dining philosophers algorithm with response time $O(\delta^{\log\delta+1})$ and message complexity $O(\delta^{\log\delta+1})$. Choy and Singh [9] developed a dining philosophers algorithm with response time $O(\delta^2)$ and message complexity $O(\delta)$. They also include some discussion of fault-tolerance. Awerbuch and Saks [3] provides a dining philosophers algorithm with response time is $O(\delta^2(\log^*|\mathcal{U}|))$, where $\mathcal{U}$ is the universal set from which process IDs are drawn [2], and the message complexity is $O(\delta^2\log^*|\mathcal{U}|)$. Bar-Ilan and Peleg [6] developed a *synchronous* algorithm that

[2] $\log^* n = \min\{i : \log^i n \leq 1\}$.

improves on Awerbuch and Saks' algorithm to have response time $O(\delta(\log^* |\mathcal{U}|))$ in a *synchronous* network.

Ramanathan [22] defines a unified framework, called UxDMA, where any channel assignment strategy can be applied to all the possible combinations of different conflict relations. Given a set of constraints among nodes defined by a conflict relation, obtaining the optimal channel scheduling that minimizes the number of channels is known NP-hard [22]. By representing the conflict relation among nodes into a conflict graph, UxDMA employs graph coloring, which is also NP-hard, as the core of the channel assignment strategy, and proposes several heuristic schemes based on greedy graph coloring. Among them, RAND is the most commonly used channel assignment strategy because of its simplicity [22]. However, as RAND requires the global knowledge of the input graph, it is not practical for large-scale sensor networks.

The theoretical computer science community has also independently worked on graph coloring extensively. The best distributed algorithms for graph coloring are randomized ones (e.g., [18], [15], [16]) developed for a completely synchronous environment. $\delta+1$ coloring algorithms by Luby [18] color any general graph with $\delta+1$ colors. Their running time is $O(\log n)$ with high probability. Another distributed algorithm by Grable and Panconesi [15] solves Brooks-Vizing coloring which uses fewer than $\delta+1$ colors for some special types of graphs.

TDMA scheduling is an extensively studied subject (see [26]). Most of early work is centralized and has performance dependency to $O(n)$ where n is the total size of the network. Recent distributed solutions [5], [29], [25], [8] improve the performance by removing global topology dependency. These algorithms are developed for mobile environments where nodes can frequently move and typically uses many more time slots than $\delta+1$ (for some protocols, e.g., [8], [29], these bounds are not given). NAMA [5] uses a hash function to determine priority among contending neighbors. One main drawback of this hashing based technique is priority chaining; even though a node gets a higher priority in one neighborhood, it may still have a lower priority in other neighborhood. This chaining can build up to $O(n)$, yielding a very inefficient schedule. Thus the maximum slot number of NAMA is $O(n)$. FPRP [29] allows nodes to select slots randomly using a five-phase algorithm. But it is possible that a node may not be assigned to a slot and requires many runs to increase the chance that a node gets assigned to a slot. SEEDEX[25] uses a similar hashing scheme as NAMA based on a random seed exchanged in a two-hop neighborhood. Its maximum slot number is $\delta+1$ as each node can pick randomly (instead of the minimum) a channel among those not taken by the others.

REFERENCES

[1] R. Alur, H. Attiya and G. Taubenfeld, "Time-Adaptive Algorithms for Synchronization," *Proc. 26th ACM Symposium on Theory of Computing*, Montreal, Quebec, Canada, May 1994, pp. 800–809.

[2] H. Attiya, C. Dwork, N. A. Lynch and L. J. Stockmeyer, "Bounds on the Time to Reach Agreement in the Presence of Timing Uncertainty," *Journal of the ACM*, vol. 41, no. 1, January 1994, pp. 122–152.

[3] B. Awerbuch and M. Saks, "A dining philosophers algorithm with polynomial response time," *Proc. 31st IEEE Symposium on Foundations of Computer Science,* St. Louis, MO, pp. 65–74, Oct. 1990

[4] S. Bajaj et al. "Improving simulation for network research," Technical Report 99-702b, Univ. Southern California, March 1999.

[5] L. Bao and J. J. Garcia-Luna-Aceves, "A new approach to channel access scheduling for ad hoc networks," *ACM MobiCom*, 2001, pp. 210–221.

[6] J. Bar-Ilan and D. Peleg, "Distributed resource allocation algorithms," *Proc. 6th International Workshop on Distributed Algorithms,*, pp. 277–291, Sept. 1992.

[7] K. Chandy and J. Misra, "The drinking philosophers problem," *ACM Transactions on Programming Languages and Systems,* vol. 6, pp. 632–646, 1984.

[8] I. Chlamtac and A. Farago, "Making transmission schedules immune to topology changes in multi-hop packet radio networks," *IEEE/ACM Trans. Networking*, vol. 2, no. 1, pp. 23–29, Feb. 1994.

[9] M. Choy and A. Singh, "Efficient fault-tolerant algorithms for resource allocation in distributed systems," *Proc. 24th ACM Symposium on Theory of Computing,* pp. 593–602, May 1992. Also to appear in *ACM Transactions on Programming Languages and Systems*.

[10] M. Choy and A. Singh", "Efficient Distributed Algorithms for Dynamic Channel Assignment," *Proc. 7th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, pp. 208-212, October 1996.

[11] B. A. Coan and J. L. Welch, "Transaction Commit in a Realistic Timing Model," *Distributed Computing,* vol. 4, 1990, pp. 87–103.

[12] E. Dijkstra, "A Discipline of Programming," Prentice Hall, Englewood Cliffs, 1976.

[13] E. Dijkstra, "Hierarchical ordering of sequential processes," *Acta Informatica,* vol. 1, fasc. 2, pp. 115–138, 1971.

[14] C. Dwork, N. Lynch and L. Stockmeyer, "Consensus in the Presence of Partial Synchrony," *Journal of the ACM,* vol. 35, no. 2, 1988, pp. 288-323.

[15] D. Grable and A. Panconesi, "Fast Distributed Algorithms for Brooks-Vizing Colourings," *Proc. of SODA: ACM-SIAM Symposium on Discrete Algorithms*, 1998.

[16] A. Johansson, "Asymptotic Choice Number for Triangle-Free Graphs," DIMACS, Sept., 1996.

[17] L. Lamport, "On Interprocess Communication," *Distributed Computing*, pp. 77–101, Vol. 1, No. 2, 1986.

[18] M. Luby, "Removing randomness in parallel computation without processor penalty," *Journal of Computer and System Sciences*, vol. 47, no. 2, pp. 250–286, Oct. 1993.

[19] N. Lynch, "Upper bounds for static resource allocation in a distributed system," *Journal of Computer and System Science,* vol. 23, pp. 254-278, 1981.

[20] Crossbow technology inc. mica2 wireless measurement system datasheet.

[21] S. Raghunath, K. K. Ramakrishnan, S. Kalyanaraman, and C. Chase, "Measurement Based Characterization and Provisioning of IP VPNs," *Proc. of IMC'04*, Oct., 2004.

[22] S.Ramanathan, "A unified framework and algorithms for (T/F/C)DMA channel assignment in wireless networks," *Proc. of IEEE INFOCOM*, 1997, pp. 900–907.

[23] I. Rhee, A. Warrier, M. Aia, and J. Min, "Z-MAC: a Hybrid MAC for Wireless Sensor Networks," Technical Report, Department of Computer Science, North Carolina State University, April 2005. http://www.csc.ncsu.edu/faculty/rhee/export/zmac.

[24] J Polastre, J Hill, and D Culler. "Versatile low power media access for wireless sensor networks," In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys)*, Baltimore,MD, November 2004.

[25] R. Rozovsky and P. R. Kumar, "SEEDEX: a MAC protocol for ad hoc networks," in *International Symposium on Mobile ad hoc networking & computing*, 2001, pp. 67–75.

[26] J. A. Stankovic, T. E. Abdelzaher, C. Lu, L. Sha, and J.C. Hou, "Real-time communication and coordination in embedded sensor networks," *Proceedings of IEEE*, vol. 91, no. 7, pp. 1002–1022, Jul. 2003.

[27] E. Styer and G. Peterson, "Improved algorithms for distributed resource allocation," *Proc. 7th ACM Symposium on Principles of Distributed Computing,* Toronto, Canada, pp. 105–116, August 1988.

[28] G. Zhou, T. He, S. Krishnamurthy, and J. Stankovic. "Impact of Radio Irregularity on Wireless Sensor Networks," *MobiSys'04*, Boston, MA, June 2004.

[29] C. Zhu and M. S. Corson, "A five phase reservation protocol (FPRP) for mobile ad hoc networks," *Wireless Networks*, vol. 7, no. 4, pp. 371–384, Sep. 2001.