Byzantine Resilient Synchronization for Content and Presence Updates in MANETS

Bogdan Carbunar, Michael Pearce, Shivajit Mohapatra, Loren J. Rittle, Venu Vasudevan
Applied Research and Technology Center, Motorola
Schaumburg, IL, 60195
Email: {carbunar,michael.pearce,mopy,ljrittle,cvv012}@motorola.com

***Abstract*—In this paper, we present techniques for synchronizing nodes that periodically broadcast content and presence updates to co-located nodes over an ad-hoc network, where nodes may exhibit Byzantine malicious behavior. We first propose an algorithm for synchronizing the periodic transmissions of all the nodes in an attacker-free multi-hop network. This allows nodes to save battery power by switching off their network cards without missing updates from their neighbors. We then introduce a suite of spoofing attacks and show that they are able to disrupt synchronization and destabilize the network even when launched by a single attacker in large, multi-hop networks. Finally, we devise a rating based algorithm that rates neighbors based on the consistency of their behavior. By favoring well-behaved nodes in the synchronization process, we show that we can address the issue of Byzantine malicious behavior very effectively. Our evaluation shows that the algorithms are computationally efficient and, for the setup considered, extend the device lifetime by 30% over an always-on Wi-Fi scenario. Moreover, in the presence of attacks, our rating based algorithm quickly stabilizes the synchronization process and reduces the number of lost updates by 85%.**

I. Introduction

The global penetration of mobile phones that have rich media and wireless networking capabilities has ushered in a new paradigm in mobile computing with new emerging social behaviors. New enabling technologies now allow users to search, locate, download and share dynamically created content with friends and family from their mobile devices. With ad-hoc networking capabilities in mobile devices, we are beginning to see the above trend shift from wide-area communities of users to dense local-area social situations (e.g. coffee shops, train stations, football fields etc.) [1], [2], [3]. Such a shift presents opportunities to design proximity aware systems that deliver novel social experiences. For example, fans watching a football game can automatically share pictures taken on their mobile phones with each other, while commenting/rating pictures being taken around them.

Designing systems for highly dynamic ad-hoc environments presents several interesting research challenges, including the difficult problem of providing scalable, energy efficient presence and content updates. To keep information fresh in such environments, the distribution mechanisms have to focus on frequent, small meta-data updates rather than large infrequent payloads, which could also be a cause of significant battery drain from a mobile device. One approach to address this issue is to synchronize transmission times of all participating nodes in the system. Transmission synchronization presents energy saving opportunities through dynamic power management of the network interface. That is, nodes can switch off their wireless interfaces between transmissions. However, in uncontrolled ad-hoc environments a single malicious user can easily disrupt network stability and synchronization, affecting either the nodes' power savings or their ability to receive updates from their neighbors. Therefore, designing synchronization algorithms that are resilient to Byzantine behavior of nodes is of paramount importance.

The synchronization problem can be stated as follows: Given a set of nodes participating in an ad-hoc network, where each node has a duty cycle consisting of active and sleep intervals and uses a multicast presence distribution protocol for periodically exchanging updates, design an algorithm to achieve synchronized transmission times between all the nodes of the network. While this approach does not force nodes to synchronize their duty cycle schedules, it may require them to shortly and periodically wake up during their sleep intervals in order to send their updates and receive those of their neighbors. The difficulty of the problem lies in the fact that nodes have arbitrary join/leave times, flexible initial periods between transmissions and unsynchronized clocks. Since nodes may become co-located only for short intervals, we are particularly interested in algorithms that achieve synchronization in a timely manner and efficiently handle network updates.

Our Randomized Future Peak Detection algorithm (FPDR), introduced in Section IV solves the above problem. FPDR is built on a Content and Presence Multicast Protocol (CPMP) [4] (described in Section II). CPMP is a simple mechanism for nodes to send updates, including the relative time of their next transmission, to their neighbors. FPDR enables each node to decide its next transmission time based only on the CPMP packets received from its neighbors. For each packet received, a node uses only information that can be inferred directly from it (e.g., arrival time and relative time of next transmission). While its minimalistic trust in the data contained in updates shields FPDR from a large class of malicious attacks, it also makes it vulnerable to spoofing attacks. Our second contribution then consists of devising three spoofing attack classes, where the perpetrator spoofs a large number of device identifiers and sends packets on their behalf. Our simulations from Section VII show that for networks of a few tens of nodes, a single attacker implementing either of these attacks

1-4244-2507-5/08/$20.00 ©2008 IEEE

can keep the entire network not only from synchronizing, but also from reaching a stable synchronization point.

We address FPDR's vulnerabilities by designing a Rating Based Algorithm, RBA, presented in Section V. In RBA, nodes locally maintain rating values for each of their neighbors, based on the stability (and predictability) of their behavior. The use of ratings enables nodes to discard from their synchronization process, updates received from unreliable or unstable sources. We prove that this approach effectively thwarts the tower and dispersion attacks. Moreover, we present several attacks targeted specifically against rating based synchronization mechanisms and show that RBA efficiently prevents or isolates their effects.

Our implementation evaluation on a Motorola A910 phone shows that our protocols are both power and computation efficient. By using a simple duty cycle protocol for turning off the Wi-Fi card, we can achieve more than a 30% device lifetime extension. Moreover, the overhead for processing a packet (either sent or received) is around 25 μs. Our simulation results show that in the absence of attackers, FPDR can synchronize large and dense networks in reasonable time (e.g., 8 minutes for a 40 node network). Moreover, even in the presence of attacks, RBA is not only able to quickly stabilize the synchronization process (e.g., less than 9 minutes for a 100 node network), but also to reduce the update loss rate to approximately 15%.

II. System Model

A. Update Dissemination

We build our synchronization algorithms on top of a content/presence dissemination protocol called Content and Presence Multicast Protocol (CPMP). A full description of CPMP can be found in [4]. The CPMP protocol was designed specifically to support social content consumption experiences in ad-hoc wireless local area network environments. The protocol describes a messaging format that allows nodes to disseminate content and/or presence updates, such as content currently being consumed and content that is being sought for future consumption at each node. CPMP messages are transmitted periodically to inform nearby devices of updated content presence information using IP multicast. Each CPMP message also includes a field (T_X) that specifies the number of seconds in which to expect a new CPMP message from that node. Then, a CPMP header has the following format

$$\texttt{CPMP}, \texttt{device\_identifier}, \texttt{T}_\texttt{X}.$$

By transmitting CPMP messages at approximately the same time CPMP messages are expected, an implementation can avoid permanently powering on the wireless LAN radio, which can result in energy savings.

B. Assumptions

Each device X has a unique identifier, $Id(X)$, which can be either the device IP or MAC address. We assume devices can form ad hoc networks, where connectivity is dictated by the ability of devices to establish connections with other devices in their vicinity. We are interested mainly in multi-hop networks, since the single-hop scenario has substantially easier solutions.

Initially, each node keeps its network interface active for an interval of length t_a, followed by k sleep intervals, each of the same length t_a. The value of k offers an obvious tradeoff between the resulting battery savings and the time it takes to discover new neighbors. Since nodes are not synchronized, they start their active and sleep intervals independently of each other. Nodes hear any CPMP broadcast made during the interval t_a and miss them when the network interface is in sleep mode. Each node wakes up at every interval ($T_X = t_a$) and broadcasts its own CPMP update. This behavior is repeated for the entire life of the node. Note that T_X does not need to be equal to t_a. However, by imposing this constraint, we can ensure that during each active interval a node will receive all the updates sent by nodes within transmission range. The algorithms that we propose in this paper work even if $T_X > t_a$, however, nodes may take longer to synchronize.

An important assumption that we avoid to make, is that devices have public key certificates certified by a trusted authority. While such an assumption would simplify our solutions, it would also significantly impact performance. If nodes were to authenticate their CPMP updates and given that new nodes may join at any time, a node would need to include its certificate in each update. This operation would be both expensive in terms of communication and computation (signature verification) overhead.

Finally, we do not consider physical-layer attacks such as jamming, but instead confine ourselves to attacks occurring above the MAC layer. Previous work addressing jamming attacks can be found in [5], [6], [7].

C. Attacker Model

We assume nodes may be corrupted and exhibit Byzantine behavior. Such nodes may run modified code and behave in an unpredictable manner. We assume that an attacker can spoof any device identifier. This can be easily performed since knowledge of device identifiers is not a verifiable operation. As mentioned before, we chose this alternative since using cryptographic primitives for verification is an expensive alternative, in itself prone to denial-of-service attacks.

III. Our Approach

Let us first define several notions that we will use throughout the paper.

Definition 3.1: (**Stable State**) A network is said to be in a *stable state* if the T_X value of each node remains unchanged over time.

Definition 3.2: (**Synchronization**) Two nodes are said to be synchronized if (i) they are within transmission range and (ii) the times of their CPMP update transmissions coincide. A *cluster of synchronization* is a sub-set of the nodes of the network that transmit at the same time. A network is said to be *synchronized* if it has reached a stable state, with a single cluster of synchronization.

For example the black (blue) nodes in in Figure 1 are part of cluster C_1 and the grey (red) nodes are part of cluster C_2.

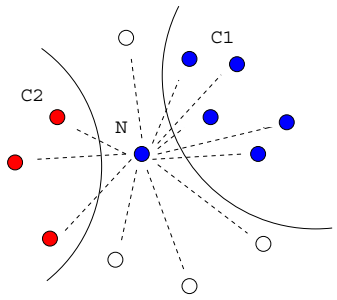


Fig. 1. Example node N (center) with 12 neighbors. Dotted lines represent bidirectional communication links. Nodes shown in black (blue in color) will transmit at the same time, as will nodes shown in grey (red). White nodes will each transmit in a different time slot.

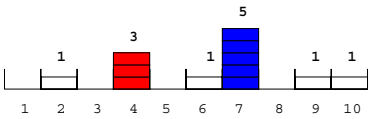


Fig. 2. Example $slotArray$ structure maintained by node N of Figure 1.

Time Discretization: We divide each interval (active or sleep) into s sub-intervals, called *slots*, of length $t_s = t_a/s$. Each node N maintains an array $slotArray$ of size s, where each entry in the array stores a list of packets. Let N's current active interval, of length t_a, be its Tth interval. The next interval, the $T+1$st, will be a sleep interval. At the beginning of the Tth interval, N resets all entries of $slotArray$. During the active interval, N listens on its network interface, collects all the packets received and places them in the $slotArray$ structure in the following manner. For each packet pkt received from a neighbor during the Tth interval, N computes the neighbor's next transmission time, as promised by the T_X field of the packet and then stores the packet in the slot of index

$$slot(pkt) = ((t_{curr} + T_X)\%t_a)/t_s$$

of $slotArray$, where t_{curr} is the time when the packet was received. Figure 2 shows the $slotArray$ structure of node N, for a possible transmission pattern of the nodes shown in Figure 1. Considering values of $t_a = 20$ and $s = 10$ and that the blue nodes send at time $t_{curr} = 38$ with a $T_X = 16$, their packets will be stored in $slotArray$ on the entry corresponding to slot $(54\%20)/2 = 7$.

At the end of the active interval, N decides the time of its next transmission based on the information carried by the packets in $slotArray$. The exact procedure for performing this operation defines the algorithm's performance. In the following sections we instantiate two different ways for nodes to choose the slot with which to synchronize. We use the expression "node N synchronizes with slot s" to denote the fact that N's transmissions will occur in the sth slot of following intervals (nodes send updates once per active/sleep interval). By synchronizing with slot s, node N implicitly synchronizes its transmission with all its neighbors that have also chosen slot s for transmission. Note that nodes only synchronize their transmission times (slots), not their active/sleep schedules.

Algorithm 1 Generic Counting Algorithm. The invoked $getStartActiveInt$ and $getStartSleepInt$ methods provide the time when the next ACTIVE or SLEEP interval begins. The methods $initState$, $setTX$ and $processPackets$ will be instantiated in the following sections.

```
1. Object implementation GENERIC;
2.   inQ : InputQueue;      #packet recv queue
3.   pktList : Pkt[];        #list of packets
4.   T_X : int;              #time to next transmission
5.   nextSendCPMP : int;     #abs next transmission time
6.   t_curr : int;           #current time
7.   s : int;                #number of slots per interval
8.   t_a : int;              #period length
9.   t_s : int;              #duration of a slot
10. slotArray : Slot[s];     #packet organizer
11. state : int;             #node state
12. Operation main()
13.     while (true) do
14.         t_curr := getCurrentTime();
15.         if(t_curr = getStartActiveInt()) then
16.             initState();
17.             state := ACTIVE;
18.         else if(t_curr = getStartSleepInt()) then
19.             setTX();
20.             state := SLEEP;
21.         else if(state = ACTIVE) then
22.             processPackets(t_curr);
23.         fi
24.     od
```

Generic Algorithm: Algorithm 1 shows the pseudo-code of the high-level behavior of a node. The $main$ method (lines 12-24) consists of an infinite loop (line 13). The behavior is dictated by the current time (t_{curr}, line 14). If the node is at the beginning of an active interval (line 15) it calls the $initState$ method to initialize the $slotArray$ structure (line 16) and switches its state to ACTIVE. If an active interval has just completed and the node enters a sleep interval (line 18), the $setTX$ method is called to process the $slotArray$ structure and decide the node's future transmission time (line 19). The node then switches to a SLEEP state (line 20). If none of these conditions is satisfied, but the node is in an ACTIVE state (line 21), the algorithm calls the method $processPackets$ in order to retrieve all the packets received at time t_{curr} and update its $slotArray$ structure (line 22). We will later instantiate the $initState$, $setTX$ and $processPackets$ methods for actual solutions.

IV. Randomized Future Peak Detection Algorithm

In this section we propose a lightweight solution for synchronizing all the nodes in a multi-hop network, in the absence of Byzantine failures. The algorithm is called *randomized future peak detection* (FPDR). As previously mentioned, at the end of each active interval, a node running FPDR has to choose one of the s slots of each interval and synchronize with it. The chosen slot is called the *winner slot*.

Our strategy is the following. Let $total = \sum_{i=0}^{s} |slotArray[i]|$ be the total number of packets

Algorithm 2 Randomized Future Peak Detection Algorithm (FPDR). The method $getAllPackets$ of the input queue inQ returns all the packets received in the past t_s seconds (current slot).

```
1. Object implementation FPDR extends GENERIC;
2.   total : int;            #number packets received
3.   rand : int;             #pseudo − random generator
4.   Operation initState()
5.     for (i := 0; i < nSlots; i++) do
6.       slotArray[i] := new pkt[]; od
7.     total := 0;
8.   end
9.   Operation setTX()
10.    if(total! = 0) then
11.      sel := rand.nextInt() % total + 1;
12.      for (i := 0; i < nSlots; i++) do
13.        if(slotArray[i].size() < sel) then
14.          sel := sel − slotArray[i].size();
15.        else winnerSlot := i;
16.        fi
17.      od
18.      if(winnerSlot! = nextSendCPMP % t_a) then
19.        T_X := winnerSlot;
20.        nextSendCPMP := t_curr + T_X;
21.      fi
22.    fi
23. end
24. Operation processPackets(t_curr : int)
25.    pktList := inQ.getAllPackets(t_s);
26.    for (i := 0; i < pktList.size(); i++) do
27.      index := ((t_curr + pktList[i].T_X) % t_a)/t_s;
28.      slotArray[index].add(pktList[i]);
29.      total := total + 1;
30.    od
31. end
```

received by the node during the active interval. Then, the node will synchronize with a slot $x \in 1..s$ with probability $p_x = |slotArray[x]|/total$. For instance, using the example shown in Figure 1, node N will choose the 7th slot (chosen by 5 of its neighbors) for its transmission with probability 5/12 and the 4th slot (chosen by 3 of its neighbors) with probability 3/12.

Algorithm 2 presents the details of the FPDR algorithm, which instantiates the generic solution shown in Algorithm 1. The $initState$ method (lines 4-8), executed at the beginning of each active interval, resets each entry of the $slotArray$ structure and also the $total$ variable, which will count the number of packets received in that interval. The $processPackets$ method (lines 24-31) is executed periodically and uses the network interface's input queue inQ to retrieve all the packets received before its call time (line 25). For each such packet, the node computes the next transmission time as promised by the T_X field of the packet. It then determines the slot corresponding to that future time (line 27) and adds it to the entry in $slotArray$ corresponding to that slot (line 28). The $total$ variable is also incremented, to account for this packet (line 29).

The $setTX$ method (lines 9-23), executed at the end of each active interval, decides the node's winner slot. This is done by randomly picking a selector (sel) value in the interval $1..total$ (line 11) and using it to choose the winner slot in a weighted

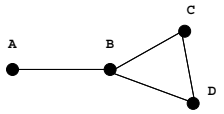


Fig. 3. Example network whose histogram is showed in Figure 4.

probabilistic fashion (lines 12-17). $setTX$ then synchronizes the node with the winner slot, by setting the node's T_X value to the $winnerSlot$ value (line 19) and correspondingly updates the time of the node's next transmission (line 20). The $nextSendCPMP$ value encodes the absolute time when the node will transmit its next CPMP update. To save space, the actual transmission is not shown in pseudo-code.

Example: Figure 4 shows a possible outcome of the FPDR algorithm for the network illustrated in Figure 3. Each node starts with an active interval. In this example, the duty cycle of each node consists of an active interval followed by a sleep interval. Initially, after each interval, a node sends a CPMP packet advertising the time of its next transmission. In this example, node B is the first to start and later to send a CPMP packet. The packet, received by both A and C (D is not yet active) makes both nodes synchronize with B. The first synchronized transmission of A, B and C takes place at time $T(A,B,C)$. Later, when node D receives two updates, from B and C, it places them in the same slot and synchronizes with them. The first synchronized transmission of the entire network takes place during the next interval (at time $T(A,B,C,D)$).

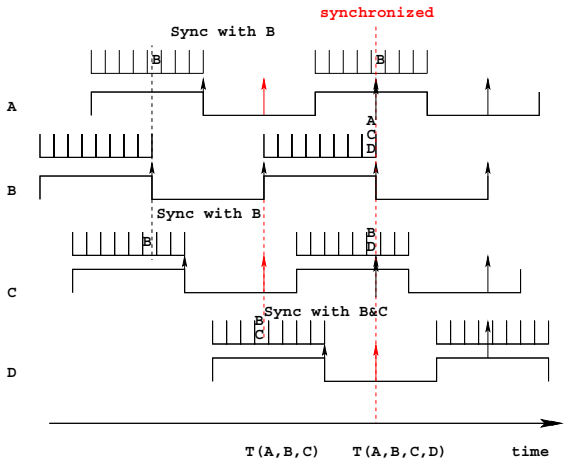


Fig. 4. Histogram of a four node network shown in Figure 3. Each node has a duty cycle consisting of one active period (up zones) and one sleep period (down zones). Each active period of a node has a list of slots (shown above up zones). Node transmissions are shown with black and grey (red) arrows.

Joining/leaving nodes: When a node joins a synchronized network, it becomes synchronized with the network after its first active interval. This is because all the packets it receives are placed in the same slot. Leaving nodes are straightforward to handle since the remaining nodes will have one less node to sync with.

A. Resilience to Attacks

In FPDR nodes do not propagate information, thus preventing malicious nodes from spreading inaccurate data. However, nodes running FPDR are unable to verify the authenticity of the received packets' senders. This weakness can be exploited by the following spoofing attacks. Let T be the current interval and let $M_1(T), .., M_m(T)$ be a set of device identifiers to be spoofed by an attacker $\mathcal{M}$ during this interval. Then, let $pkt_i(T) = \langle M_i(T), t_i, T_{X_i} \rangle$ denote the CPMP packet sent at time t_i (within interval T) by $\mathcal{M}$, as if coming from spoofed node $M_i(T)$, with a value $T_X = T_{X_i}$.

Tower Attack: $\mathcal{M}$ creates the packets $pkt_i(T)$ such that $s = t_i + T_{X_i} = t_j + T_{X_j}$, $i, j = 1..m$. Effectively, all the neighbors of $\mathcal{M}$ will believe that the next transmission of the nodes $M_1(T), .., M_m(T)$ will coincide and will likely attempt to synchronize with them. However, during the next interval, $\mathcal{M}$ can spoof another set of device identifiers and construct their CPMP updates such that they will all be placed in another slot $s' \neq s$. The number of packets from $\mathcal{M}$ that are placed in the same slot is m, called the *height* of the tower.

Dispersion Attack: Let $I(T) = \{I_1(T), .., I_\rho(T)\}$ be a set of time slots during interval T, where ρ is called the attack's *dispersion factor*. Let $|I_j(T)|$ denote the number of packets that are sent in slot $I_j(T)$. $\mathcal{M}$ generates packets $pkt_i(T)$ such that $slot(pkt_i(T)) \in I(T)$, $i = 1..n$ and $|I_j(T)| = \sigma$, for all $j = 1..\rho$. $\sigma = n/\rho$ is called the *height* of the attack.

Effects of Tower and Dispersion Attacks: The tower and dispersion attacks have the purpose of artificially increasing the number of packets received by neighbors of $\mathcal{M}$. In FPDR, a node has a higher chance of choosing as winner a slot with more packets in it. Then, by constantly changing the slots where the updates from spoofed devices will be placed, an attacker influences its neighbors to change their transmission slots. As our simulations from Section VII show, this behavior quickly propagates to the entire network, preventing not only its synchronization, but also the stabilization of the synchronization process.

V. A Rating Based Algorithm

In the following we propose a Rating Based Algorithm (RBA) that addresses the issues previously exposed. RBA requires each node to build statistics of its neighbors' transmission stability: a neighbor that consistently sends its CPMP updates in the same slot is considered more stable than a node that regularly changes slots. In effect, each node is building a *rating* value for each neighbor. Ratings are used only locally and are not propagated to neighbors. After an active interval, a node will synchronize with the slot in which the packet from the highest rated neighbor has been placed. Thus, neighbors with lower ratings (e.g., newly seen or unstable) will have little chance to influence the synchronization process.

The ratings are built as follows. When a node A receives a packet from a neighbor B, if the packet's slot coincides with the slot of B's previous transmission, B's rating is incremented. If not, B's rating is dropped to 0. Note that B's rating is set to 0 also if B misses one transmission or if B is a newly seen neighbor.

In order to prevent impersonation attacks, we also extend the CPMP protocol to include a minimal overhead authentication information. While this additional data does not prove that the node is who it claims to be, it allows its neighbors to make a reasoning of the form: "this packet was sent by the same device that has sent a similar packet one interval ago". We achieve this by using "hash chain" constructs. That is, when a node sends a first CPMP packet, it includes a $H^n(R)$ field, which denotes the unique R value hashed n times. When the node sends a second packet, it includes the value $H^{n-1}(R)$. This value allows its neighbors to verify that only the same node could have sent the second packet. This process is continued until the node reaches its $(n+1)$th transmission, when, along with the cleartext R value it also includes a new hash chain, $H^n(R')$, for a fresh R'. A CPMP packet for the kth transmission of node N has the following format

$$\texttt{CPMP}, \texttt{Id(N)}, \texttt{T}_\texttt{X}, \texttt{H}^{\texttt{n}-\texttt{k}}\texttt{(R)}.$$

Our Rating Based Algorithm, RBA, whose pseudocode is shown in Algorithm 3, implements these mechanisms in the following manner. For each neighbor B from which a packet has been received in the past active interval, a node A maintains B's last transmission slot (the HT hashtable of line 2), B's rating value (stored in the r hashtable from line 3) and B's previously sent hash value (stored in the h hashtable from line 4).

Node A processes each packet received in the $processPackets$ method of Algorithm 3 (lines 5-21) by first looking up the sender's identifier, B, in its HT structure (line 11). If the identifier has not been seen before, the rating of B, r_B, is set to 0 (line 17). Then, the packet's slot index, computed on line 9, is inserted in the HT hashtable (line 18) and the packet's hash is stored the h hashtable (line 19).

If the identifier has been seen before (line 11), A verifies that the hash contained in the packet comes indeed from the same B that has sent a packet before (line 12). If the check does not verify, A drops the packet. Otherwise, if the packet's slot index coincides with B's previous transmission slot (line 14), B's rating is incremented (line 15). If however, B has sent the packet in a different slot, its rating drops to 0 (line 16).

Similar to FPDR, B's packet is placed in the $slotArray$ entry corresponding to the packet's slot (lines 9-10). At the end of the active interval, the $setTX$ method (lines 22-36) processes the $slotArray$ structure in order to find the slot containing the packet sent by the sender with the highest rating value (lines 24-31). Node A will then synchronize with this slot (lines 32-35). We can now prove the following result.

Theorem 5.1: RBA thwarts the tower and dispersion attacks.

Proof: (Sketch) Due to space constraints, we consider only the tower attack. Since the dispersion attack can be viewed as a generalization of the tower attack, our reasoning

Algorithm 3 Rating Based Algorithm.

```
1.Object implementation RBA extends GENERIC;
2.  HT : int[];          #history table
3.  r : int[];          #rating table
4.  h : LargeInt[];          #hash list
5.  Operation processPackets(t_curr : int)
6.     pktList := inQ.getAllPackets(t_s);
7.     for (i := 0; i < pktList.size(); i++) do
8.        sdr := pktList[i].getSender();
9.        index := ((t_curr + pktList[i].T_X) % t_a)/t_s;
10.       slotArray[index].add(pktList[i]);
11.       if (HT.contains(sdr) = true) then
12.          if (hash(pktList[i].getHash()) ! = h[sdr])
13.             break; fi
14.          if (HT[sdr] = index) then
15.             r[sdr] := r[sdr] + 1;
16.          else r[sdr] := 0; fi
17.       else r[sdr] = 0; fi
18.       HT[sdr] := index;
19.       h[sdr] := pktList[i].getHash();
20.    od
21. end
22. Operation setTX(t_curr : int)
23.    maxR := 0;
24.    for (i := 0; i < nSlots; i++) do
25.       for (j := 0; j < slotArray[i].size(); j++) do
26.          sdr := slotArray[i][j].getSender();
27.          if (r[sdr] > maxR) then
28.             winnerSlot := i;
29.             maxR := r[sdr];
30.          fi
31.       od od
32.    if(winnerSlot! = nextSendCPMP % t_a) then
33.       T_X := winnerSlot;
34.       nextSendCPMP := t_curr + T_X;
35.    fi
36. end
```

applies also to the dispersion attack. Let the ids spoofed by a malicious node $\mathcal{M}$ during interval T be $M_1(T), .., M_m(T)$. All the packets generated by $\mathcal{M}$ advertise the same future transmission slot. Let $N_1, .., N_n$ be $\mathcal{M}$'s neighbors. Let us first consider the case where throughout the attack, $\mathcal{M}$ uses the same ids, that is $M_i(T) = M_i(T')$ for any two intervals $T \neq T'$. Let those ids be denoted by $M_1, ..M_m$. If the slot chosen by $\mathcal{M}$ for the transmissions of $M_1, .., M_m$ changes at most every c cycles, the ratings of $M_1, ..M_m$ will never exceed c in the view of nodes $N_1, .., N_n$. Then, any $N \in \{N_1, ..N_n\}$ that has a neighbor with a rating larger than c, will synchronize with that neighbor. If N does not have any neighbor with a rating larger than c, then following $M_1, ..M_m$ is the best strategy.

Let us now consider the case where $M_i(T) \neq M_i(T')$, that is, $\mathcal{M}$ changes the spoofed ids throughout the attack. Note that as soon as a node stops receiving packets from a neighbor, it removes its record from local storage, thus effectively dropping that neighbor's rating to 0. Thus, old spoofed ids will be immediately discarded by $\mathcal{M}$'s neighbors. Let l_i be the number of consecutive duty cycles in which an id M_i is used by the attacker. Then, the rating of M_i is $r[M_i] = min(c, l_i)$. Thus, $\mathcal{M}$ has the following tradeoff, either (i) use smaller c or l_i values and become irrelevant during the synchronization decision process of nodes $N_1, .., N_n$ or (ii) use larger c and l_i values and actually speed up the synchronization process of its neighbors. ■

We propose now several attacks targeted specifically against rating based synchronization mechanisms and show RBA's defenses against them.

Circular Cascade Attack: $\mathcal{M}$ generates a fixed number of device identifiers, $M_1, .., M_n$. That is, $M_i(T) = M_i(T')$, for $i = 1..n$ and for all $1 \leq T, T'$ time intervals. Let $r[M_i]$ denote the rating of M_i and $S[M_i]$ denote the slot where M_i's updates are placed. W.l.o.g., let $r[M_1] \geq r[M_2] \geq .. \geq r[M_n]$. Then, one active interval after detecting that a sufficient number of its neighbors have synchronized with slot $S[M_1]$ (chosen by M_1, the best rated spoofed id), $\mathcal{M}$ changes M_1's transmission slot. Since M_1's rating drops to 0, $\mathcal{M}$'s neighbors are forced to re-synchronize. Since $r[M_2]$ may be sufficiently high, some of $\mathcal{M}$'s neighbors will synchronize with M_2, by choosing slot $S[M_2]$ for their next transmission. $\mathcal{M}$ repeats this process for the entire duration of the attack, effectively cycling through ids $M_1, .., M_n$. For more details on a possible implementation of this attack see Section VII-C.

However, the following result shows that RBA isolates the effects of the cascade attack.

Theorem 5.2: RBA isolates the circular cascade attack to the neighbors of $\mathcal{M}$.

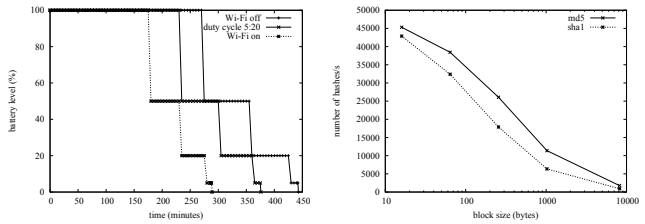
Proof: Consider a case where node A has $\mathcal{M}$ and B as neighbors, such that B is not within the communication range of $\mathcal{M}$. When, after $r[M_1]$ active intervals $\mathcal{M}$ changes M_1's transmission slot, node A is forced to re-synchronize with another of its neighbors. Then, since in B's view, A's reputation drops to 0, $\mathcal{M}$'s behavior will not impact B. ■

Framing Attack: $\mathcal{M}$ monitors the transmissions of its neighbors and detects the one with the highest rating. Let that node be N. At some point, $\mathcal{M}$ starts spoofing N. This attack can not only desynchronize the common neighbors of $\mathcal{M}$ and N, but also frame N and ruin its reputation. It is easy however to see that the hash-chains embedded in the CPMP updates prevent this attack from affecting nodes running RBA. Specifically, since for a given value v it is computationally infeasible to find a value x such that $H(x) = v$, $\mathcal{M}$ is unable to spoof the packets transmitted by other nodes.

Denial of Storage Attack: RBA stores a constant amount of data for each id recently seen. Thus, an attacker that sends many packets with different source ids may end up exhausting the storage space of its neighbors, making them drop statistics concerning valid nodes. However, this is not a concern. First, in RBA, a node stores statistics only for ids that have sent a packet in the last active interval. Second, the data stored for a node id consists of 3 values: the transmission slot, the rating and the hash value last received from that node. If the hash takes 10 bytes and a node id takes as much as 100 bytes, with a storage capacity of 10MB (SD cards of 1GB are available) a node can store statistics for at least a few tens of thousands of node ids. If an attacker sends so many packets

during each interval, it is more likely to cause a jamming attack (see [5], [6], [7] for defenses against jamming attacks).

VI. Implementation Evaluation



(a) (b)

Fig. 5. Motorola A910 phone implementation evaluation. (a)Battery lifetime with various Wi-Fi operation modes. The x label shows the time since the beginning of the experiment and the y axis shows the percentage of battery left. By using a duty cycle for the Wi-Fi card, we are able to extend the device lifetime by almost 90 minutes, over the 290 minutes lifetime when the Wi-Fi is always on. (b)MD5 and SHA1 hashing performance. For 16B blocks, both MD5 and SHA1 are able to perform 40000 hashes per second. RBA requires a single hash computation per packet sent/received.

We have evaluated the performance of our protocols on Motorola Martinique (A910) phones , which are Linux/Java phones with an ARM architecture. The A910 has an Intel XScale-PXA27x processor, 10MB RAM and is equipped with an internal Wi-Fi card.

Battery Lifetime Extension: We have investigated the battery savings enabled by the synchronization of transmission times. For this, we have explored three Wi-Fi card usage modes, namely, (a) always off, (b) always on and (c) duty cycle. The duty cycle consists of a 5s active interval (Wi-Fi on) followed by 3 sleep intervals, each 5s long. During each sleep interval, the Wi-Fi card is off for 4s and on for 1 second, to allow the node to receive transmissions from neighbors (collision resolution and accounting for unsynchronized clocks). We have used a program running in the background, using TAPI (Telephony API), to probe the remaining battery level once a minute and print it to a local file. The A910 has only 4 battery levels, of 100%, 50%, 20% and 5% of the fully charged battery (1100 mAh). Figure 5(a) shows our findings. While the maximum saving mode ("always-off") extends the battery life by around 54% (2 hours and a half), the duty cycle behavior extends the battery lifetime by 30% (one hour and a half) over the "always-on" behavior. The reason for the only 23% additional savings enabled by the always-off versus the duty cycle mode, is that other phone components (e.g., CPU, light) also consume battery power.

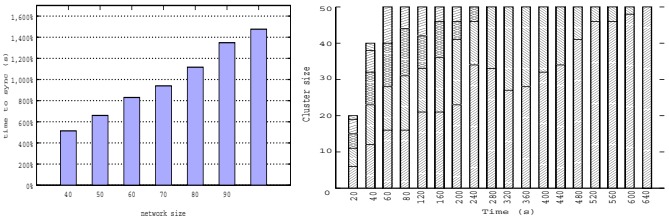
Crypto-Hash Evaluation: To understand the effects of hashing (used to prevent framing attacks) on RBA's computing performance, we have evaluated it using OpenSSL, that we ported to the arm architecture. We have tested two algorithms, MD5 and SHA1, for block sizes ranging from 16 to 8192 bytes. In RBA, a node computes a hash-chain of length n to last it for n transmission, thus, a single hash computation is required per transmission. Moreover, when a node receives a packet, it only needs to compute one hash and perform one comparison.

Figure 5(b) shows the number of hashes per second performed for a given block size. We have used the log scale for the x axis. While both MD5 and SHA1 perform more than 40000 hashes per second on 16B blocks, the number decreases exponentially with the block size, reaching around 1000 hashes per second for 8192B blocks. However, for our implementation we perform hashes on random values that are less than 16B long. The time to compute and verify a hash is then less than 25 μs, which is quite reasonable.

VII. Simulation Results

WhereFiSim: We designed and implemented a Java based simulator called "WhereFiSim" for evaluating the performance of our algorithms on larger scale networks that were hard to realize in practice. We deployed nodes uniformly at random in a 150×150 m^2 rectangular area. Each node, modeled by a A910 phone, has a transmission range of 30m. Each node has a start-up time. We simulated a worst-case scenario, where nodes join in close sequence, of one node per second. This is a worst case scenario since nodes start up unsynced and none have synchronized by the time the last one joins. Similar to the implementation evaluation, each node's duty cycle consists of one 5s active interval, followed by three sleep intervals of 5s each. Each interval is split into 5 slots, thus, the length of a slot is $t_s = 1s$. Each simulation starts at time zero.

A. FPDR Performance



(a) (b)

Fig. 6. FPDR performance. (a) Synchronization times for networks between 40 and 100 nodes. Each point is an average over 10 runs. The synchronization delay increase is linear in the size of the network. 40 nodes synchronize in roughly 8 minutes. (b) Evolution of synchronization cluster sizes on a network of 50 nodes. While the whole network synchronizes in 10 minutes, many nodes are much earlier synchronized with a large fraction of their neighbors.

In the first experiment we studied the synchronization performance of FPDR in a scenario where all the nodes are well-behaved. Figure 6(a) shows the time required by all the nodes to synchronize their transmissions times, for networks of 40 to 100 nodes. Each point shown is an average over 10 independent runs. The increase in the synchronization delay is roughly linear, with less than 500 seconds (8 minutes) for networks of 40 nodes and less than 1400 seconds (23 minutes) for networks of 100 nodes. This increase is due to larger network degrees (higher node densities), since FPDR probabilistically

chooses one of the slots chosen by its neighbors. Note that for networks of 100 nodes, all the nodes have joined after 100s.

Figure 6(b) magnifies one point of this experiment, showing the evolution in time of the sizes of the clusters of synchronization for one run of an experiment for a 50 node network. By time 50s all the nodes have joined, by time 220s two of the initial four clusters have merged and by time 280s there are only 2 more clusters left. While the sizes of the two remaining clusters are initially balanced, all the nodes synchronize by time 640s.

B. Effects of Malicious Attacks

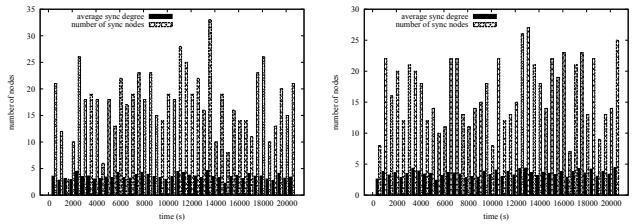


Fig. 7. FPDR: average degree of synchronization and the number of nodes synchronized with all their neighbors for the (a) tower and (b) dispersion attacks. Even though at times many nodes are synchronized, it does not last. Moreover, between successive measurements, the actual sets of synchronized nodes change significantly.

To understand the effects of tower and dispersion attacks on FPDR we have implemented an attack scenario consisting of a single malicious node, $\mathcal{M}$, deployed at the center of the $150 \times 150m^2$ area. We study a worst case scenario, where $\mathcal{M}$ is the first to join the network (time 0s). For both attacks, $\mathcal{M}$ sends 10 packets during each of its active or sleep intervals. Each packet has a different source header. During the tower attack the packets are sent such that all will be placed in the same slot by $\mathcal{M}$'s neighbors. Thus, the tower height is 10. During the dispersion attack, each packet has a random T_X value, making the average dispersion factor $\rho = 5$ and the height of the attack $\sigma = 2$ (on average 2 packets in each slot).

We study the evolution in time of two metrics. The first metric is called the *average sync degree* and represents the average number of neighbors with whom a node is synchronized. The second metric is the number of *fully synced nodes*, that is, the nodes that are synchronized with all their neighbors. Figure 7 show the effects of the tower and dispersion attacks on these metrics, on a network of 50 nodes. The experiment is run for 20000s and points reported are 2000s apart. The small bars are the average sync degree and the longer bars are the number of fully synced nodes. For both attacks, the number of fully synced nodes varied significantly during a long run of the attacks, between 3 and 33. The average sync degree varied between 2.24 and 4.48, when the network's degree (the average number of neighbors per node) is 5.76. Thus, even though at times FPDR is almost able to synchronize the network, its success is only temporary. We have performed numerous, longer than 20000s experiments and neither synchronization nor stabilization was ever achieved.

C. RBA Resilience to Attacks

When studying RBA, we are interested in its behavior not only under the above tower and dispersion attacks but also under a cascade attack instance. In the implemented cascade attack, the attacker uses a fixed list of 10 spoofed node ids and generates their different ratings by having these nodes "join" in consecutive intervals. Each spoofed node sends in the same slot for 10 consecutive cycles and then changes transmission slot. Thus, at all times, exactly one of the spoofed ids has a rating of 10, one has a rating of 9 and so on. However, during each cycle, the leader id (rating of 10) changes its slot and ends up with a rating of 0.

A node running RBA bases its synchronization decision on the history of past transmissions of its neighbors. For this reason, in our simulation, at start-up, each node runs FPDR, while simultaneously building the neighbor transmission slot statistics needed by RBA. Later, at a predefined time, the node switches to RBA and uses the existing statistics to make more reliable decisions. We tested this strategy by experimenting with networks of 40 to 100 nodes running FPDR for 1000s, then switching to RBA. A single node launches one of the tower, dispersion or cascade attacks previously mentioned. Figure 8(a) shows the average sync degree corresponding to each attack for these networks and Figure 8(b) shows the number of fully synced nodes. Each bar is an average over 10 random runs. In both figures, the first bar is for the dispersion attack, the second bar is for the tower attack and the third bar is for the cascade attack. The fourth bar in Figure 8(a) is the network degree. For all attacks, both the average sync degree and the number of completely synced nodes increase linearly with the number of nodes in the network. For the dispersion and tower attacks, the number of fully synced nodes ranges between 55 and 65% of the total number of nodes in the network and for the cascade attack this value ranges between and 44% and 55%.

The average sync degree of nodes is very close to the network's degree, showing that on average, nodes are synchronized with all but one or two of their neighbors. Specifically, for a 100 node network, for the dispersion attack, the update loss rate when the network is stable is less than 14% and for the tower attack the loss rate is less then 16%. For the same experiment, Figure 8(c) shows the time required by nodes running RBA to reach a stable state of synchronization (see Definition III). The first observation is that the increase in stabilization time is roughly linear. However, RBA stabilizes the synchronization process of 100 nodes in less than 120 seconds. The second observation is that for all attacks (except for the 40 nodes network) RBA requires a similar synchronization time.

In the second experiment we study the effects of the size of the history on RBA's effectiveness against the above attacks launched against a network of 100 nodes. For this we have increased the time for building transmission statistics (when nodes run FPDR) from 400 to 6000 seconds. Figure 9 shows the average sync degree (the small bars) and the number of fully synced nodes (the higher bars) for this experiment,

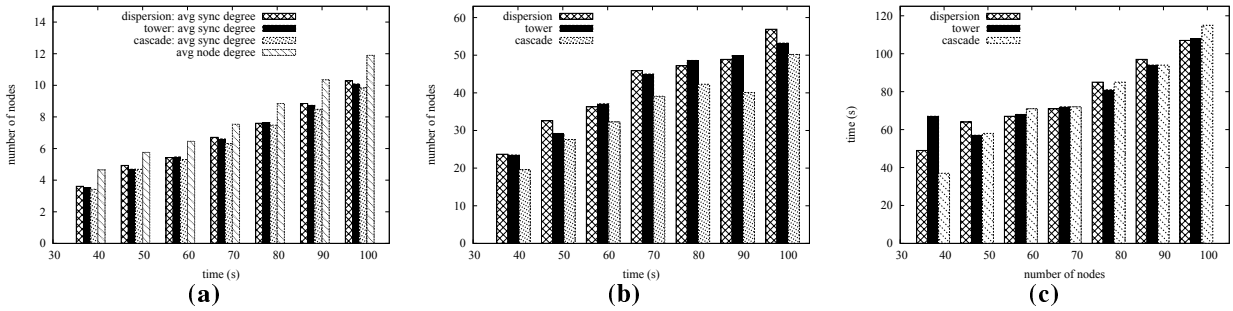


Fig. 8. Effects of dispersion, tower and cascade attacks launched by a single node against RBA in networks of 40 to 100 nodes. RBA consistently brings the synchronization to a stable state. (a) The average sync degrees achieved by RBA during a dispersion (first bar), tower (second bar) and cascade (third bar) attack. The fourth bar is the network's degree. The increase in average sync degrees is linear and very close to the network's degree. (b) The number of fully synced nodes achieved by RBA under a dispersion (first bar), tower (second bar) and cascade (third bar) attack. The increase is linear with the number of nodes. Around 50% of the nodes are fully synced. (c) Time required by RBA to stabilize synchronization. While exhibiting a slightly sub-linear increase with the network size, even for 100 nodes the stabilization time is under 2 minutes.

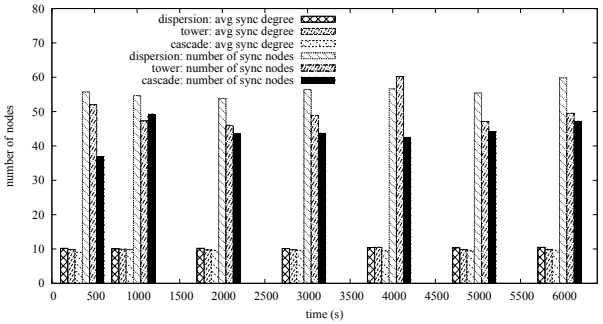


Fig. 9. RBA on a 100 nodes network, when the switch time between FPDR and RBA increases from 400 to 6000 seconds. No significant advantage for longer wait times can be observed, thus, RBA can be launched soon after node start-up.

each an average over the outcome of 10 experiment runs. It is interesting to observe that a longer time for building statistics before actually using them, offers no significant advantage. Thus, using only 400 seconds for running FPDR before switching to RBA is enough to synchronize nodes with almost all their neighbors, effectively defending against tower, dispersion and cascade attacks. Note that for a network of 100 nodes, less than 9 minutes are enough to bring the network to a stable state (400 seconds can be used to gather statistics and 120 seconds are enough for RBA to bring the network to a stable state).

VIII. Related Work

Medium Access Control in Sensor Networks: Perhaps closest to our contribution is the research on the uses of time slots for medium access control mechanisms, in reducing battery consumption in sensor networks. Notably, the first result in this direction is the slotted MAC protocol proposed by Ye et al. [8]. In S-MAC, each node has an active/sleep duty cycle. Nodes broadcast their schedules once every cycle and a node can adopt a neighbor's schedule, which is called "primary schedule". Nodes may form clusters, where all the nodes in a cluster have the same primary schedule. Nodes that border on multiple clusters are forced to monitor the schedules of all the clusters they border. S-MAC was subsequently extended by T-MAC [9], DSMAC [10] SCP-MAC [11] and MS-MAC [12]. While our approach is similar in the use of duty-cycles, there are two notable differences. First, we do not attempt to synchronize node schedules, but only the periodic, node transmission times. Second, these protocols do not consider the possibility of malicious, Byzantine faults.

In this respect, Lu et al. [13] have shown that the blind trust of nodes running S-MAC (and all subsequent protocols) in their neighbors' schedules, can lead to simple attacks of disastrous consequences. Moreover, they have proposed a simple, threshold technique to address the proposed attacks. Our work however is different also with respect to the vulnerability to attacks. This is because our protocols nodes do not trust the validity of the data contained in the packets received from neighbors. Thus, none of our protocols (FPDR and RBA) is vulnerable to the attacks proposed by Lu et al. [13].

Fault Tolerant Clock Synchronization: Lamport and Melliar-Smith [14] were the first to study the problem of achieving clock synchronization in the presence of Byzantine faults. They proved that $3m + 1$ clocks are enough to synchronize the non-faulty clocks in the presence of m faults. Solutions for fault tolerant clock synchronization in distributed systems take either a software or a hardware approach. The software approach is flexible and economical, but requires additional messages. The hardware approach uses special hardware at each node to achieve tight synchronization with minimal time overhead. For an overview of solutions please see [15]. Note however that our protocols do not rely on synchronized clocks.

MANET Power Management: State-of-the-art research on ad-hoc networks continues to search for ways to optimize energy while minimizing the penalties incurred due to latency, dropped packets and partitioned networks caused by induced low-duty cycles on the network interface. In [16], the author identifies three broad categories in which power management for ad-hoc networks can be classified: rendezvous based

wakeup [17], [18], [19], [20], [21], [22], [23], [24] where all nodes are synchronized to listen to the medium around the same time, asynchronous wake-up [25], [26] where nodes are not synchronized but the wakeup cycles are designed to overlap, and booted wakeup [27], [28], [29] wherein a low-power alternate radio (e.g. Bluetooth) is used to sense the medium and boot up the wireless interface when required. For our application requirements and specific ecosystem which needed all nodes to receive updates frequently and reliably, the scheduled rendezvous based wakeup approach was the obvious choice.

Inadvertent Synchronization: Of particular relevance to our contributions is also the work of Floyd and Jacobson [30], who studied the process of inadvertent synchronization of periodic routing messages. The authors investigated a network of 20 nodes sending periodic messages at a time offset chosen randomly between 0 and 120 seconds. They showed that the nodes achieved transmission synchronization without external interference, after almost 100000 seconds (27 hours) into the experiment (smaller clusters were observed earlier, but the largest cluster started developing at around 80000s). We note that a similar behavior might also be observed in our case. Since the time a node spends processing packets is proportional to the number of packets received, a node will tend to naturally synchronize with slots where it receives more packets. However, while the synchronization of periodic routing messages in the Internet can lead to congestion and should be avoided, in our case synchronization is not only desirable but it should occur quickly (order of minutes instead of hours).

IX. Conclusion

In this paper we study the problem of synchronizing the periodic transmissions of nodes in a multi-hop network, in order to enable battery lifetime extensions without missing neighbor's updates. We first propose a solution that is lightweight and scalable, but vulnerable to attacks. We then extend the solution to use node transmission stability as a metric for synchronization and show that this technique is efficient against a wide array of attack types. Our implementation and simulations show that our protocols are computationally inexpensive, provide significant battery savings, are scalable and efficiently defend against attacks.

X. Acknowledgments

We would like to thank Nitya Narasimhan for extensive initial discussions and the anonymous reviewers for their detailed feedback.

References

[1] P. Persson, J. Blom, and Y. Jung, ""DigiDress: A Field Trial of an Expressive Social Proximity Application","in *UbiComp*, 2005.

[2] ""The Nokia Sensor Project", http://europe.nokia.com/a4144923." [Online]. Available: http://europe.nokia.com/A4144923

[3] H. Caituiro-Monge, K. Almeroth, and M. del Mar Alvarez-Rohena, ""Friend Relay: A Resource Sharing Framework for Mobile Wireless Devices","in *ACM International Workshop on Wireless Mobile Applications and Services on WLAN Hotspots*, Sept 2006.

[4] S. Mohapatra, B. Carbunar, M. Pearce, R. Chaudhri, and V. Vasudevan, "Where-Fi: a Dynamic Energy-Efficient Multimedia Distribution Framework for MANETs," in *Proceedings of the SPIE/ACM Multimedia Computing and Networking (MMCN)*, 2008.

[5] A. D. Wooda, J. A. Stankovic, and G. Zhou, "DEEJAM: Defeating Energy-Efficient Jamming in IEEE 802.15.4-based Wireless Networks," in *Proceedings of the 4th Annual IEEE SECON*, 2007.

[6] W. Xu, W. Trappe, and Y. Zhang, "Channel surfing: defending wireless sensor networks from interference," in *Proceedings of the 6th IPSN*, 2007.

[7] M. Cagalj, S. Capkun, and J.-P. Hubaux, "Wormhole-based antijamming techniques in sensor networks," *IEEE Transactions on Mobile Computing*, vol. 6, no. 1, 2007.

[8] W. Ye, J. Heidemann, and D. Estrin, ""An energy-efficient MAC protocol for wireless sensor networks","in *IEEE INFOCOM*, 2002.

[9] T. V. Dam and K. Langendoen, ""An adaptive energy-efficient MAC protocol for wireless sensor networks","in *ACM SenSys*, Nov 2003.

[10] P. Lin, C. Qiao, and X. Wang, "Medium access control with a dynamic duty cycle for sensor networks," in *Proceedings of the IEEE WCNC*, vol. 3, 2004, pp. 1534–1539.

[11] W. Ye, F. Silva, and J. Heidemann, "Ultra-low duty cycle mac with scheduled channel polling," in *ACM SenSys*, 2006, pp. 321–334.

[12] H. Pham and S. Jha, "An adaptive mobility-aware mac protocol for sensor networks (ms-mac)," in *Proceedings of the IEEE MASS*, 2004, pp. 214–226.

[13] X. Lu, M. Spear, K. Levitt, and S. F. Wu, "The synchronization attack and defense on energy-efficient listen-sleep slotted mac protocols, Tech. Rep. CSE-2007-33, 2007.

[14] L. Lamport and P. M. Melliar-Smith, "Synchronizing clocks in the presence of faults," *J. ACM*, vol. 32, no. 1, pp. 52–78, 1985.

[15] P. Ramanathan, K. G. Shin, and R. W. Butler, "Fault-tolerant clock synchronization in distributed systems," *IEEE Computer*, vol. 23, no. 10, pp. 33–42, 1990.

[16] T. Armstrong, "Wake-up based power management in multi-hop wireless networks," in *Term Survey Paper for QoS Provisioning in Mobile Networks*, 2005.

[17] B. Awerbuch, D. Holmer, and H. Rubens, ""The Pulse Protocol: Energy efficient Infrastructure Access","in *IEEE INFOCOM*, 2004.

[18] B. Chen, K. Jamieson, H. Balakrishnan, and R. Morris, ""Span: an Energy-Efficient Coordination Algorithm for Topology Maintenance in Ad Hoc Wireless Networks","in *ACM Wireless Networks Journal, Volume 8, Number 5*, Sept 2002.

[19] A. Tyrrell, G. Auer, and C. Bettstetter, ""Fireflies as Role Models for Synchronization in Ad Hoc Networks","in *Intl. Conf. on BIONETICS*, Dec 2006.

[20] R. E. Mirollo and S. H. Strogatz, ""Synchronization of pulse-coupled biological oscillators","*SIAM Journal on Applied Mathematics, Volume 50 , Issue 6*, Dec 1990.

[21] Z. Li and B. Li, ""Probabilistic power management for wireless ad hoc networks","in *ACM/Kluwer Mobile Networks and Applications (MONET)*, Oct 2005.

[22] S. PalChaudhuri and D. B. Johnson, ""Birthday Paradox for Energy Conservation in Sensor Networks","in *USENIX Symposium of Operating Systems Design and Implementation*, 2002.

[23] R. Zheng and R. Kravets, ""On-demand Power Management for Ad Hoc Networks","in *IEEE INFOCOM*, 2003.

[24] H. Jun, W. Zhao, M. H. Ammar, E. W. Zegura, , and C. Lee, ""Trading Latency for Energy in Wireless Ad Hoc Networks using Message Ferrying","in *Workshop on Pervasive Wireless Networking*, 2005.

[25] R. Zheng, J. C. Hou, and L. Sha, ""Asynchronous Wakeup for Ad Hoc Networks","in *ACM MobiHoc*, Jun 2003.

[26] L. M. Feeney, ""A QoS Aware Power Save Protocol for Wireless Ad Hoc Networks","in *Proceedings of the First Mediterranean Workshop on Ad Hoc Networks*, 2002.

[27] T. Pering, Y. Agarwal, R. Gupta, and R. Want, ""CoolSpots: reducing the power consumption of wireless mobile devices with multiple radio interfaces","in *Proceedings of ACM/USENIX MobiSys*, 2006.

[28] E. Shih, P. Bahl, and M. J. Sinclair, ""Wake on Wireless: An Event Driven Energy Saving Strategy for Battery Operated Device","in *Proceedings of the 8th MOBICOM*, 2002.

[29] M. J. Miller and N. H. Vaidya, ""Power Save Mechanisms for Multi-Hop Wireless Networks","in *Proceedings of the 1st BROADNETS*, 2004.

[30] S. Floyd and V. Jacobson, "The synchronization of periodic routing messages," *IEEE/ACM Trans. Netw.*, vol. 2, no. 2, pp. 122–136, 1994.