Secure Synchronization of Periodic Updates in Ad Hoc Networks

Bogdan Carbunar, *Member, IEEE,* Michael Pearce, Shivajit Mohapatra, Loren J. Rittle, Venu Vasudevan, Octavian Carbunar

Abstract—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. Instead of aligning duty cycles, our algorithms synchronize the periodic transmissions of nodes. This allows nodes to save battery power by switching off their network cards without missing updates from their neighbors. We propose several novel attack classes and show that they are able to disrupt synchronization even when launched by a single attacker. Finally, we devise a rating based algorithm (RBA) that rates neighbors based on the consistency of their behavior. By favoring well-behaved nodes in the synchronization process, we show that RBA quickly stabilizes the synchronization process and reduces the number of lost updates by 85%. Our evaluation also shows that all our algorithms are computationally efficient and, for the setup considered, extend the device lifetime by 30% over an always-on Wi-Fi scenario.

Index Terms—C.2.2.a Network Protocols - Applications, C.2.4.b Distributed Systems - Distributed Applications, C.2.8.a-d Mobile Computing, E.3 Data Encryption

1 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 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. In Section 3.1 we propose a Content and Presence Multicast Protocol (CPMP) which nodes use to send updates to their neighbors. The updates contain the relative time of their sender's next transmission.

We propose to address the energy efficiency problem by synchronizing the transmission times of all the 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. Designing synchronization algorithms that are resilient to Byzantine behavior of nodes is instrumental to the correct functionality of a network.

Our main contribution consists then of a suite of synchronization protocols, built on top of CPMP. Specifically, in Section 5 we present WBS, a Weight Based Synchronization protocol that uses the size of synchronized node clusters as a catalyst for synchronization. While efficient, we show that WBS's reliance on information contained in CPMP updates makes it vulnerable to simple attacks. To address this problem, in Section 6 we propose the Future Peak Detection (FPD) algorithm. Nodes running FPD use CPMP updates to sync with their largest set of already synced neighbors, counting the number of packets received within a given interval and setting the node's next transmission to be in sync with the slot where most packets have been received. While lightweight and efficient, FPD's greedy strategy clusters the network: nodes reach a stable state without being synchronized with all their neighbors. We address this issue using randomization. Our Randomized Future Peak Detection algorithm (FPDR), presented in Section 7 is similar to FPD but uses a weighted probabilistic

- *Bogdan Carbunar, Michael Pearce, Shivajit Mohapatra, Loren J. Rittle and Venu Vasudevan are with the Applied Research Center in Motorola, Schaumburg, IL 60195. E-mail: {carbunar,michael.pearce,mopy,ljrittle,venu.vasudevan}@motorola.com*
- *Octavian Carbunar is with the National Institute of Physics and Nuclear Engineering, Magurele (Bucharest), Romania. Email: carbunar@nipne.ro*

strategy to decide a node's next transmission time based on the packets received.

The minimalist trust in the data contained in updates shields FPDR from a large class of malicious attacks. However, FPDR's inability to authenticate packet senders leaves it vulnerable to other attacks, such as spoofing. To prove this we devise novel classes of attacks against transmission synchronization protocols and show their effectiveness even when deployed by a single attacker on large networks. In Section 8 we propose a Rating Based Algorithm, RBA, to address these attacks. In RBA, nodes locally rate their neighbors, based on the stability (and predictability) of their behavior. The use of ratings enables nodes to discard updates received from unreliable or unstable sources. We prove that this approach effectively thwarts or severely limits the effects of all our attacks.

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%.

2 Related Work

This paper extends our previous work [4] in several respects. First, it provides a detailed description of the system architecture, including two application built on top of CPMP, MeCast and Zeitgeist. Second, it includes two additional synchronization algorithms, WBS and FPD, along with their analysis.

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. [5]. 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 [6], DS-MAC [7] SCP-MAC [8] and MS-MAC [9]. 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. [10] 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. [10].

Fault Tolerant Clock Synchronization: Lamport and Melliar-Smith [11] 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 [12]. 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 [13], the author identifies three broad categories in which power management for ad-hoc networks can be classified: rendezvous based wakeup [14], [15], [16], [17], [18], [19] where all nodes are synchronized to listen to the medium around the same time, asynchronous wake-up [20], [21] where nodes are not synchronized but the wakeup cycles are designed to overlap, and booted wakeup [22], [23], [24] 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 [25], 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).

3 System Model

3.1 CPMP Overview

We designed the Content Presence Multicast Protocol (CPMP) to support social content consumption experiences. CPMP provides a framework for periodically communicating information about what content is currently being consumed and what content is being sought for future consumption at each participating node. Our goal was to make the protocol efficient and scalable while including features intended to support synchronization of presence message transmissions. CPMP messages are transmitted periodically to inform nearby devices of updated content presence information using IP multicast. CPMP headers have the following format

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

containing a T_X field specifying the number of seconds in which to expect a new CPMP message from the node specified by $device\_identifier$. Note that this does not include any redundant transmissions of the current message - if retransmission is used to improve reliability, each retransmission must update the T_X field so that it accurately reflects the delay until the new message will be transmitted. By transmitting CPMP messages at approximately the same time CPMP messages are expected, an implementation can avoid powering on the wireless LAN radio for transmissions - essentially piggybacking CPMP transmission with reception. Ideally, all participating nodes will use this technique simultaneously, resulting in synchronization of CPMP activity.

CPMP packets may also include an HTTP port number on which a host may be listening for HTTP requests to support related services (such as content transfer). CPMP messages are transmitted via multicast UDP/IP frames. Each message is encapsulated in a single UDP/IP message and transmitted to the CPMP multicast address. In order to avoid IP fragmentation, the protocol requires that implementations limit message sizes to the link layer maximum transmission unit.

3.2 System Architecture

The system architecture is shown in Figure 1 with the contributions of this work highlighted in gray. The power aware CPMP (PAC) module is implemented in middleware and is the primary component of the system. Both the CPMP protocol and our algorithms are implemented within this module. The middleware exports APIs that are used by applications using the CPMP protocol. Note that "CPMP" applications could refer to a class of applications that require dissemination of periodic updates to co-located devices (e.g. presence/context information, content consumption information, heartbeats etc.). Figure 1 shows two such applications that are presented in section 3.3. It is desirable that such applications are able to disseminate updates reliably to all neighboring devices with minimum energy overhead. The CPMP protocol makes provision for such optimization by requiring applications to include their relative next transmission time in each packet. Our algorithms utilize these "transmission times" to achieve scheduled rendezvous type synchronization for all one-hop neighbors in the system.

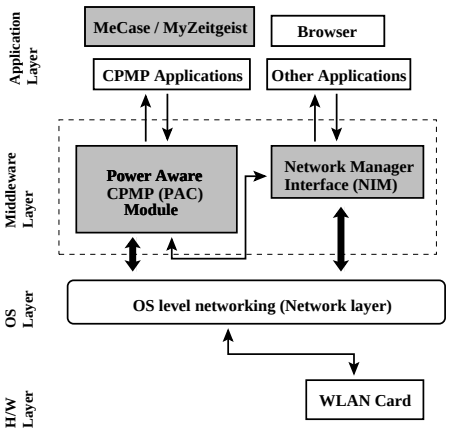


Fig. 1. System Architecture.

The PAC module talks to the network interface manager (NIM) which is a middleware abstraction used to control access to the wireless network interface hardware using OS layer API interfaces. The PAC module calculates and communicates the next expected transmission time for the "CPMP" based applications. Other applications (e.g web browser, real player etc.) could directly input their network usage requirements to the NIM module (if possible). The NIM module then uses these inputs to determine a sleep/wakeup schedule for the wireless interface card. Note that in our current approach a simplistic algorithm is employed by the NIM in case applications are unable to determine their usage requirements (e.g. browser). While it is relatively easy to optimize energy consumption (by switching network interface to a low duty cycle) on individual devices for CPMP applications (periodic), it is much harder in a highly dynamic and distributed setting with nodes constantly entering and leaving the system. The difficulty here is to achieve synchronized wakeup schedules for all devices (without synchronizing clocks or performing neighbor discovery) while still ensuring reliable delivery of updates.

3.3 Applications

Our framework monitors the media consumption activities on devices on which it runs and employs the CPMP module to update neighboring devices with metadata of the item currently consumed/produced. The use of CPMP enables devices to (i) display a list of currently consumed/produced items on neighboring devices (MeCast) and (ii) build popularity statistics for recommendation purposes (Zeitgeist). We implemented both applications on Motorola A910 and E680 phones (see [26] for more implementation details). We considered two types of content, music, which is consumed on the phones and photos, which are produced using the phones' in-built cameras. For music, CPMP updates include metadata of the currently playing item. For photos, CPMP packets include a 64 × 48 thumbnail of the last picture taken with the device. [1].

The MeCast application allows a user to tune in into the experience of any of the neighboring devices and follow its content consumption/production activities. That is, a "caster" device can transfer content over Wi-Fi from neighboring "castees" and play the content locally. This is done automatically when the user selects one of the displayed items currently consumed or produced by neighbors. However, instead of having the caster manually select each item she wants to transfer, the MeCast experience enables her to transfer the selected castee's items when and in the order in which they are consumed or produced by the castee. This is done until the caster explicitly selects a different neighbor to follow or decides to consume or produce her own items. Thus, each device effectively behaves like a radio station. Note that each device can be both a caster and a castee, but not simultaneously.

In Zeitgeist, a device captures CPMP updates sent by other devices (potentially MeCasters) and builds statistics of the most popular items consumed in its vicinity. It then presents to the user a sorted list of the most popular items, allowing the user to obtain more information on items of interest, as well as to buy the rights to use them, using the currently available internet connectivity (Wi-Fi or GPRS). In terms of access rights, we specify that for MeCast, a caster can access items transferred from a castee only as long as a direct Wi-Fi connectivity exists between them. When the caster and castee become disconnected, a DRM module in the caster erases all items transferred from the castee.

3.4 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. Our interest is in multi-hop networks, since the single-hop scenario has substantially easier solutions. We focus on quasi-static scenarios, where only a small fraction of the nodes in the network move.

1. Thumbnails are around 800B, well below the MTU of 1500B, thus requiring a single datagram.

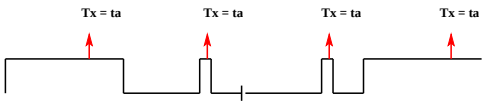


Fig. 2. Example Wi-Fi duty cycle. k, the ratio of sleep to active intervals is 2. For CPMP updates occurring during sleep intervals the Wi-Fi card is briefly turned on.

The Wi-Fi cards of devices follow a duty cycle consisting of one active interval followed by k sleep intervals (see Figure 2 for an example). Active and sleep intervals have the same length, t_a. The value of k offers an obvious trade-off between the resulting battery savings and the time it takes to discover new neighbors. On start-up, a node enters its first active interval. Since nodes are not synchronized, they start their duty cycles independently. Nodes hear CPMP broadcast packets sent during the interval t_a and miss them when the network interface is in sleep mode. Each node wakes up every interval ($T_X = t_a$) to broadcast a CPMP update. If this event occurs during a sleep interval, the node keeps its Wi-Fi card on for a brief interval of pre-defines length (1-2s). This interval allows the node to send its periodic update and also receive updates from neighbors with whom it has already synchronized.

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 $T_X < t_a$.

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 [27], [28], [29].

3.5 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.

4 Our Approach

We start by introducing several definitions that we use throughout the paper.

Definition 4.1: (**Stable State**) A network is said to be in a *stable state* if the T_X value (included in all CPMP updates) of each node does not change over time.

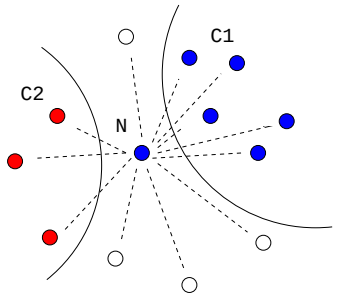


Fig. 3. Example node N (center) with 12 neighbors. Dotted lines represent bidirectional communication links. Colored nodes are synced and transparent nodes are not synced.

Definition 4.2: (**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 blue nodes in in Figure 3 are part of cluster C_1 and the red nodes are part of cluster C_2.

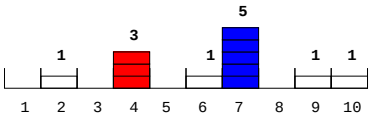


Fig. 4. Example $slotArray$ structure of node N of Figure 3.

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 4 shows the $slotArray$ structure of node N, for a possible transmission pattern of the nodes shown in Figure 3. 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 several different solutions 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 Algorithm. $getStartActiveInt$ and $getStartSleepInt$ 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 pseudocode 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.

5 Weight Based Synchronization

We first describe an algorithm that uses the size of synchronization clusters as a catalyst for synchronization. We call the algorithm WBS – *weight based synchronization*. As mentioned previously, at the end of each active interval, a node uses the *slotArray* structure to decide its next transmission time. The *slotArray* structure has s entries, one for each slot of the next (sleep) interval. The node has to choose one of these slots, called *winner slot*, and synchronize with it. That is, the node has to advertise the time of its next transmission (its T_X value in the CPMP update packet) such that the update packet will be placed into that winner slot by its neighbors.

WBS requires each node to locally maintain a variable monitoring the size of the cluster of synchronization which contains the node. The variable is called the *weight* of the node/cluster. Initially, the weight of each node is 1. Each node includes its weight in all its CPMP updates. Certainly, nodes cannot maintain globally accurate weights. Instead, each node needs to use only local knowledge – extracted from packets received from neighbors – to update the value of this variable.

During each active interval, a node keeps track of the largest cluster weight seen among all the received packets, including its own. At the end of the active interval, the node chooses the winner slot to be the one storing the packet advertising the largest weight. That is, each node chooses to synchronize with the largest synchronization cluster in its vicinity. When a node joins a cluster, the weight of the cluster increases, thus increasing the cluster's potential to attract even more members.

Algorithm 2 describes the details of WBS, instantiating the generic solution shown in Algorithm 1. The *initState* method (lines 4-7), executed at the beginning of each active interval, resets each entry of the *slotArray* structure. The *processPackets* method (lines 24-30) 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. 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).

At the end of the current active interval, the *setTX* method (lines 8-23) selects as winner the slot storing the packet containing the largest weight field. For this, it first determines the largest weight seen in any packet in the active interval and records the slot – winnerSlot (lines 9-15). If the winner slot does not coincide with its current transmission slot (line 16), the node synchronizes with the winner slot and correspondingly updates the time of the node's next transmission (lines 17-18). 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 pseudocode.

In addition, the node sets its weight to one over the largest weight seen (line 19), to reflect the operation of joining this cluster. However, if the node is already synchronized with the largest neighboring cluster (line 20), the node's weight is set to be the current weight of the cluster. This operation needs to be explicitly performed, since from the last packet received from that cluster, the size of the cluster may have increased - the cluster may have incorporated other nodes. Let n be the number of nodes in a connected network. We prove now the following result.

Theorem 5.1: In the absence of Byzantine failures, WBS synchronizes a connected network in $O(n(k+1)t_a)$ time.

Algorithm 2 WBS: Weight Based Synchronization. *initState* resets the local structures. *processPackets* updates the local structures for each received packet. *setTX* determines the winner slot to be the one containing the packet from the largest neighboring cluster of synchronization.

```
1. Object implementation WBS extends GENERIC;
2.   maxW : int; #max weight over active interval
3.   weight : int; #weight advertised in CPMP packets
4.   Operation initState()
5.     for (i := 0; i < s; i++) do
6.       slotArray[i] := new pkt[]; od
7.   end
8.   Operation setTX()
       #compute the maxW value
9.     maxW := 0;
10.    for (i := 0; i < s; i++) do
11.      for (j := 0; j < slotArray[i].size(); j++) do
12.        if (slotArray[i][j].weight > maxW) then
13.          winnerSlot := i;
14.          maxW := slotArray[i][j].weight; fi
15.    od od
       #determine new T_X and weight values
16.    if(winnerSlot! = nextSendCPMP % t_a) then
17.      T_X := winnerSlot;
18.      nextSendCPMP := t_curr + T_X;
19.      weight := maxW + 1;
20.    else
21.      weight := maxW;
22.    fi
23. end
24. Operation processPackets(t_curr : int)
25.    pktList := inq.getAllPackets(slotLen);
26.    for (i := 0; i < pktList.size(); i++) do
27.      index := ((t_curr + pktList[i].T_X) mod t_a)/t_s);
28.      slotArray[index].add(pktList[i]);
29.    od
30. end
```

Proof: The synchronization process proceeds in steps. Step σ is defined as the time when the network has at least one cluster of synchronization containing σ nodes. Each duty cycle has length $(k+1)t_a$. The process ends when one cluster has n members. If we prove that the number of duty cycles between steps is 1, the result in the theorem follows immediately.

The proof is by induction. Let w_X be the weight advertised by a node X. For the base step, all clusters of synchronization have size 1. Let us consider the case of two neighboring nodes, A and B. A and B are not synchronized otherwise the synchronization process would be in the 2nd step. If $w_A \neq w_B$ one of A or B will synchronize with the other in a single step. Otherwise, in case of tie break, WBS chooses the winner slot to be the first slot containing the max weight (Algorithm 2 lines 10-15). Since A and B are not synchronized, one packet will be in an earlier slot. Thus both A and B will choose the same slot, in a single duty cycle.

For the induction step, we assume that the network has at least one cluster with σ members. Let A be a node in such a cluster that has a neighbor B that is not synchronized with it. There has to exist such a A,B pair, otherwise the network is either entirely synchronized or disconnected. If B has a single neighbor (A) that is in a cluster of σ members it will synchronize with A's cluster in a single duty cycle – after receiving A's update. Thus, A's cluster will have $\sigma+1$ members. If B has two or more neighbors in (different) clusters of σ members each, it will synchronize with the earliest slot containing a packet from one of those neighbors (again Algorithm 2 lines 10-15). Thus, one of B's neighboring clusters of σ members will gain an additional member. The case where two or more of B's neighbors are in the same cluster is trivially similar.

If B itself is part of a cluster of σ members, the analysis is similar to the base step. That is, either A joins B's cluster or vice-versa, based on whose packet lands in an earlier slot. Either way, one cluster will end up with $\sigma+1$ members, after a single duty cycle. □

5.1 Resilience to Attacks

The ability of WBS to synchronize a network is based on the fact that nodes propagate information – the size of the largest cluster of synchronization in their vicinity. This is also the weakness of WBS, in environments where nodes may exhibit Byzantine behavior. Such nodes may include arbitrary weights in their CPMP updates, with the potential of drawing to their transmission slot first their neighbors, then nodes in their 2-hop vicinity and so on. If such nodes couple this behavior with a frequent changing of transmission slots, they may prevent the network from synchronizing. Let T be the current interval and let $\mathcal{M}$ denote an attacker. Let $w_{\mathcal{M}}(T)$ be the weight chosen by $\mathcal{M}$ during T. The following attack formalizes this behavior.

Inflation Attack: During each interval T, $\mathcal{M}$ creates and sends the CPMP packet $pkt_{\mathcal{M}}(T) = \langle \mathcal{M}, w_{\mathcal{M}}(T), t_i, T_X(T) \rangle$, where t_i is the sending time (within interval T) advertising a next transmission in $T_X(T)$. The weight $w_{\mathcal{M}}(T)$ is chosen to be a multiple of the largest weight seen by $\mathcal{M}$ during its previous (active) interval. The value $T_X(T)$ is chosen randomly every few intervals, where the changing frequency is chosen randomly from the domain $[1..f]$, where f is an integer parameter.

6 Future Peak Detection

We now propose the *future peak detection* (FPD) algorithm to address the inflation attack. Instead of relying on subjective information (the weight value contained in CPMP updates) FPD allows nodes to build a local approximation of this metric, using only objective information derived from observation – the time of update receptions.

FPD works by counting the number of packets that are stored in each slot of the current active interval. Note that each packet received during the current active interval is stored in the slot corresponding to the packet sender's next transmission time (see Section 4). FPD then makes a greedy choice for the winner slot, by choosing the slot x whose $|slotArray[x]| = max_{i=1}^{s}|slotArray[i]|$. $|slotArray[x]|$ denotes the number of packets stored in the xth entry of $slotArray$. This choice ensures that the node's next transmission is in sync with most of its neighbors. In case of ties, N chooses the earliest slot to sync.

Algorithm 3 Future Peak Detection Algorithm. $setTX$ finds the slot storing the maximum number of packets and synchronizes with it.

```
1.Object implementation FPD extends WBS;
2. maxC : int;      #max nr. of packets per slot
3. Operation setTX()
      #compute the maxC value
4.    maxC := 0;
5.    for (i := 0; i < s; i++) do
6.       if(slotArray[i].size() > maxC) then
7.          maxC := slotArray[i].size();
8.          winnerSlot := i; fi
9.    od
      #update the T_X value
10.   if(winnerSlot! = nextSendCPMP % t_a) then
11.      T_X := winnerSlot;
12.      nextSendCPMP := t_curr + T_X;
13.   fi
14.end
```

Algorithm 3 extends WBS (see Algorithm 2). The $initState$ and $processPackets$ are inherited from WBS and are excluded for space considerations.

Executed at the end of each active interval, the $setTX$ method (lines 3-14) first determines the maximum number of packets stored in any slot of the interval and marks that slot as the winner slot (lines 5-9). If the winner slot is different from the node's current transmission slot

(line 10), the node synchronizes with the winner slot, by setting the node's T_X value to the $winnerSlot$ value (line 11) and correspondingly updating the time of the node's next transmission (line 12).

FPD makes a greedy choice for the winner slot, ensuring that a node's next transmission is in sync with most of its neighbors. However, this choice fails to synchronize all the nodes. To see why this is the case, consider the network shown in Figure 3 and the corresponding $slotArray$ structure for node N shown in Figure 4. N chooses slot 7, corresponding to nodes in cluster C_1, to synchronize with. Since nodes in cluster C_2 do not receive packets from nodes in cluster C_1, they will not synchronize with the same slot.

Section 10.2 confirms this problem by showing that in most of our extensive simulations with networks of up to 100 nodes and diameters of up to 8 hops, FPD breaks the network into multiple stable clusters of synchronization.

7 Randomized Future Peak Detection

In Section 10.2 we show that for the same networks FPD is unable to completely synchronize, the situation changes when imperfect channel conditions are considered. Specifically, for a network of 100 nodes with 15% packet loss rates, FPD synchronizes the entire network in 21000 seconds. While in a network with perfect channel conditions clusters created by FPD are stable, packet loss can make nodes move from one cluster of synchronization to another, thus breaking the stability. If enough nodes switch, clusters may engulf other clusters in their vicinity, eventually creating a single cluster of synchronization.

However, relying only on packet loss is insufficient. One of our requirements is that a network synchronizes in a timely manner. To achieve this, we extend FPD with randomization: nodes choose to synchronize with their neighbors in a weighted probabilistic fashion. The algorithm is called *randomized future peak detection* (FPDR) and it works as follows.

Let $total = \sum_{i=0}^{s} |slotArray[i]|$ be the total number of packets received by a node during an active interval. Then, the node synchronizes with any slot $x \in 1..s$ with probability $p_x = |slotArray[x]|/total$. For instance, using the example shown in Figure 3, 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.

To see why this approach solves FPD's problem consider the previous example. Let us assume that first, node N synchronizes with slot 7 of cluster C_1. However, during subsequent active intervals, some nodes in cluster C_2 may choose to synchronize with node N, now of cluster C_1. Then, the other nodes in C_2 will have a higher probability of also synchronizing with N and merging with cluster C_1 (since more of their neighbors are now synchronized with cluster C_1). In time, this behavior will drag the entire network toward a single cluster of synchronization. Our simulations, described in Section 10.3 confirm that FPDR synchronized *all* the network configurations on which we have tested it.

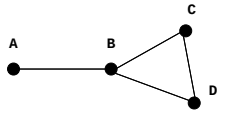


Fig. 5. Simple network - histogram shown in Figure 6.

Algorithm 4 Randomized Future Peak Detection (FPDR). $setTX$ determines the winner slot probabilistically.

```
1.Object implementation FPDR extends WBS;
2. total : int;          #total packets received
3. rand : int;           #pseudo − random generator
4. Operation setTX()
     #compute total nr. packets
5.   total := 0;
6.   for (i := 0; i < s; i++) do
7.     total := total + slotArray[i].size();
8.   od
     #choose winnerSlot probabilistically
9.   sel := rand.nextInt() % total + 1;
10.  for (i := 0; i < s; i++) do
11.    if(slotArray[i].size() < sel) then
12.      sel := sel − slotArray[i].size();
13.    else winnerSlot := i; fi
14.  od
     #set T_X value
15.  if(winnerSlot! = nextSendCPMP % t_a) then
16.    T_X := winnerSlot;
17.    nextSendCPMP := t_curr + T_X;
18.  fi
19.end
```

Algorithm 4 presents the details of the FPDR algorithm, which extends WBS (see Algorithm 2). The $initState$ and $processPackets$ methods are also inherited from WBS.

The $setTX$ method (lines 4-19), executed at the end of each active interval, decides the node's winner slot. First, the method determines the total number of packets received during the active interval (lines 5-8). Then, it determines the winner slot probabilistically. This is done by randomly picking a selector (sel) value in the interval $1..total$ (line 9) and using it to choose the winner slot in a weighted probabilistic fashion. The weight of each slot i is proportional to the number of packets stored in $slotArray[i]$ and inversely proportional to the total number of packets (lines 10-14). $setTX$ then synchronizes the node with the winner slot (lines 15-18).

Example: Figure 6 shows a possible outcome of the FPDR algorithm for the network illustrated in Figure 5. Each node starts with an active interval. In this example, the duty cycle of each node consists of an active interval followed by one sleep interval ($k = 1$). 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 then 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 $T(A,B,C,D)$).

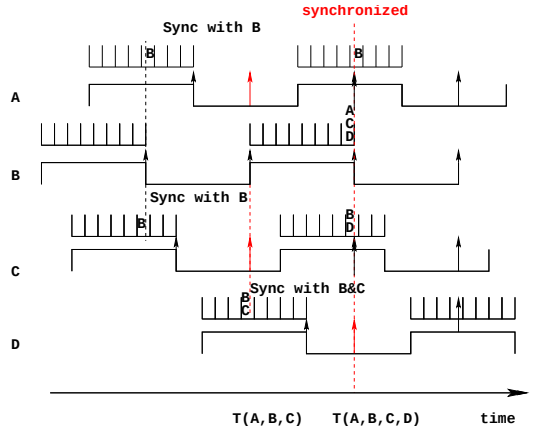


Fig. 6. Histogram of a four node network shown in Figure 5. Each node has a duty cycle consisting of one active period (up zones) and one sleep period (down zones). he slot lists for active intervals are shown above "up" zones. Node transmissions are shown with black and gray (red) arrows.

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.

7.1 Resilience to Attacks

Unlike WBS, in FPDR nodes do not propagate information (e.g., cluster sizes) thus preventing nodes from spreading inaccurate data. Thus, the inflation attack of Section 5.1 does not influence the behavior of honest nodes running FPDR. However, FPDR has another weakness: nodes are unable to verify the authenticity of the received packets' senders. This weakness can be exploited by spoofing attacks, described next. In the following, 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. In the dispersion attack the neighbors of $\mathcal{M}$ receive packets distributed over a large number of time slots, making it more difficult for all of them to choose the same slot for their next transmission. As our simulations from Section 10.4 show, this behavior quickly propagates to the entire network, preventing not only its synchronization, but also the stabilization of the synchronization process.

8 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 B's previous transmission slot, 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 hash chain root value $H^n(R)$. This root consists of the unique value R hashed n times. The format of such packets is $CPMP, Id(N), T_X, H^n(R)$.

When the node sends a second update, it includes both the previous hash, $H^n(R)$ and the previous value in the chain, $H^{n-1}(R)$. Generalized, the kth CPMP update of N has the format $CPMP, Id(N), T_X, H^{n-k}(R), H^{n-k+1}(R)$. $H^{n-k+1}(R)$ is called the "previous hash" and $H^{n-k}(R)$ is the "current hash". When N reaches its $(n+1)$th transmission, along with the cleartext R value and $H(R)$, N includes in its update a new hash chain root, $H^n(R')$, for a fresh R'. Each node needs to store n intermediate hash chain values. Used values can be discarded.

This solution allows nodes to replace advertised node ids with the hash values contained in CPMP updates. A node builds and uses a local id ($neighbor\_id$) for each of its neighbors – neighbor ratings and past behaviors are associated with these ids. The node then stores one mapping $[hash, neighbor\_id]$, between each neighbor id and the current hash value from the last update received from that neighbor.

When a node receives an update, $CPMP, Id(N), T_X, currHash, prevHash$, it first verifies the validity of the update – $H(currHash) = prevHash$. Then, the node looks to see if it stores any mapping containing the $prevHash$ value. If no such mapping exists, the node has detected a new neighbor. It then generates (randomly) a new $neighbor\_id$ and stores a $[currHash, neighbor\_id]$ mapping.

If such a mapping does exist, the node retrieves the neighbor id stored in that mapping. It then uses the id to retrieve the neighbor's rating and past history. Later, the node prepares for the next update from that neighbor, by replacing the existing $[prevHash, neighbor\_id]$ mapping with a new mapping $[currHash, neighbor\_id]$.

RBA: Our Rating Based Algorithm, whose pseudocode is shown in Algorithm 5, implements these mechanisms. Each node A stores $[hash, neighbor\_id]$ mappings in a hashtable, denoted map (line 2). Furthermore, for each neighbor from which a packet has been received in the past active interval, A maintains the neighbor's last transmission slot (the HT hashtable of line 3) and rating value (stored in the r hashtable from line 4).

Node A processes each packet received in the $processPackets$ method of Algorithm 5 (lines 5-29) by first retrieving the current and previous hash values contained in the packet (lines 8-9). A then retrieves the local id for the packet's sender (line 10), sets the packet's sender to that id (line 11) and places the packet in the $slotArray$ entry corresponding to the packet's slot (lines 12-13).

If there exists a previous hash value and the value is contained in the map structure, (line 14) then the sender of the update has also sent an update during the previous interval. If the packet's hash condition (line 15) does not satisfy, A drops the packet. Otherwise, if the packet's slot index coincides with that neighbor's previous transmission slot (line 19), the neighbor's rating is incremented (line 20). Otherwise (line 21), the neighbor's rating is dropped to 0. If however this is the first update

Algorithm 5 Rating Based Algorithm (RBA).

```
1.Object implementation RBA extends GENERIC;
2.  map : hashtable;      #map to unique ids
3.  HT : int[];           #history table
4.  r : int[];            #rating table
5.  Operation processPackets(t_curr : int)
6.     pktList := inQ.getAllPackets(t_s);
7.     for (i := 0; i < pktList.size(); i++) do
8.        currHash := pktList[i].getCurrHash();
9.        prevHash := pktList[i].getPrevHash();
10.       sdr := map.get(prevHash);
11.       pktList[i].setSender(sdr);
12.       index := ((t_curr + pktList[i].T_X) % t_a)/t_s;
13.       slotArray[index].add(pktList[i]);
14.       if (prevHash! = null
              &map.contains(prevHash)) then
15.          if (hash(currHash) ! = prevHash)
16.             break; fi
17.          map.remove(prevHash);
18.          map.put(currHash, sdr);
19.          if (HT[sdr] = index) then
20.             r[sdr] := r[sdr] + 1;
21.          else r[sdr] := 0; fi
22.       else
23.          sdr := computeNewId();
24.          map.put(currHash, sdr);
25.          r[sdr] = 0;
26.       fi
27.       HT[sdr] := index;
28.    od
29. end
30. Operation setTX(t_curr : int)
31.    maxR := 0;
32.    for (i := 0; i < nSlots; i++) do
33.       for (j := 0; j < slotArray[i].size(); j++) do
34.          sdr := slotArray[i][j].getSender();
35.          if (r[sdr] > maxR) then
36.             winnerSlot := i;
37.             maxR := r[sdr];
38.          fi
39.       od od
40.    if(winnerSlot! = nextSendCPMP % t_a) then
41.       T_X := winnerSlot;
42.       nextSendCPMP := t_curr + T_X;
43.    fi
44. end
```

received from that neighbor (line 22) a new neighbor id is generated by A (line 23) which is then mapped to the current hash value contained in the update (line 24). The rating of that neighbor is then set to 0 (line 25).

The final action performed for an update consists of recording the update's slot index, computed on line 10, in the HT hashtable (line 27).

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 8.1: 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 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 transmission slot advertised by $M_1, .., M_m$ throughout the attack is the same, this attack has a good chance of synchronizing $\mathcal{M}$'s neighborhood. 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 trade-off, 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 10.5.

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

Theorem 8.2: 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 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.

Bogus Packets: $\mathcal{M}$ does not send updates at the promised times. However $\mathcal{M}$'s reputation drops to 0 for each such event, making it irrelevant in the synchronization process.

Denial of Storage Attack: RBA stores a constant amount of data for each neighbor. Thus, an attacker that sends many packets with different hash chain values may end up exhausting the storage space of its neighbors, making them drop statistics concerning valid nodes. However, in RBA, a node stores statistics only for neighbors that have sent a packet in the last active interval. Also, the data stored for a neighbor consists only of 3 values, requiring only a few tens of bytes of storage. This enables even modest nodes to store statistics for at least a few tens of thousands of neighbors.

9 Implementation Evaluation

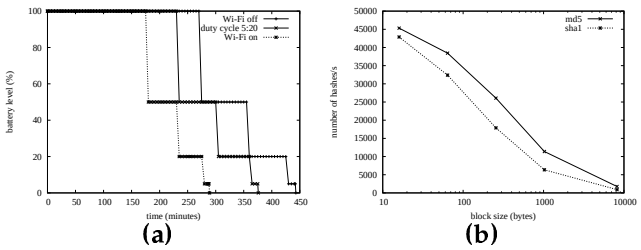


Fig. 7. Motorola A910 phone implementation evaluation. (a)Battery lifetime with various Wi-Fi operation modes shown as percentage of battery left (y axis) in time (x axis). (b)MD5 and SHA1 hashing performance.

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 7(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. Figure 7(b) shows the number of hashes per second performed for a given block size. We have used the log scale for the x axis. Both MD5 and SHA1 perform more than 40000 hashes per second on 16B blocks - used in RBA. The time to compute and verify a hash is then less than 25 μs, which is quite reasonable.

10 Simulation Evaluation

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 un-synced and none have synced 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 $t_s = 1s$. All simulations start at time zero.

10.1 Evaluation of WBS

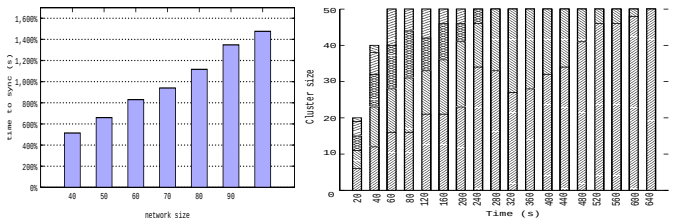
We have evaluated the performance of the weight based synchronization algorithm (WBS) for a network of 100 nodes, in conditions where the packet loss rate is 15%. Figure 8(a) shows our results. The granularity of the time scale (shown on the x axis) is 100 seconds. Even with a 15% packet loss rate, almost all nodes sync after 400 seconds and the entire network syncs after 1400 seconds.

10.2 FPD Evaluation

We have investigated FPD's clustering problem mentioned in Section 6 on a network of 100 nodes with perfect channel conditions. Figure 8(b) shows the evolution in time of the sizes of clusters of synchronization. The x axis is the time with a 10s granularity and the y axis shows cluster sizes. Initially, all nodes are in clusters of size 1. By time 150s the network reaches a stable state, with a total of 9 clusters of synchronization. We have performed this experiment for several network sizes and in the course of tens of experiments only a few have reached a stable state of complete network synchronization.

In the following experiment we have monitored the same network when the packet loss rate was 15%. Figure 8(c) shows our findings, with the x axis being again the time, but with a granularity of 1000 seconds. We notice that in this case the number of clusters reduces and after roughly 21000 seconds the network syncs completely.

10.3 FPDR Performance



(a) (b)

Fig. 9. FPDR performance. (a) Sync times for networks of 40 to 100 nodes. Linear increase: 40 nodes sync in 8 minutes. (b) Evolution of synchronization cluster sizes on a network of 50 nodes. The whole network syncs in 10 minutes.

We now study the synchronization performance of FPDR in a scenario where all the nodes are well-behaved. Figure 9(a) shows the time required by all the nodes to sync 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 9(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 sync by time 640s.

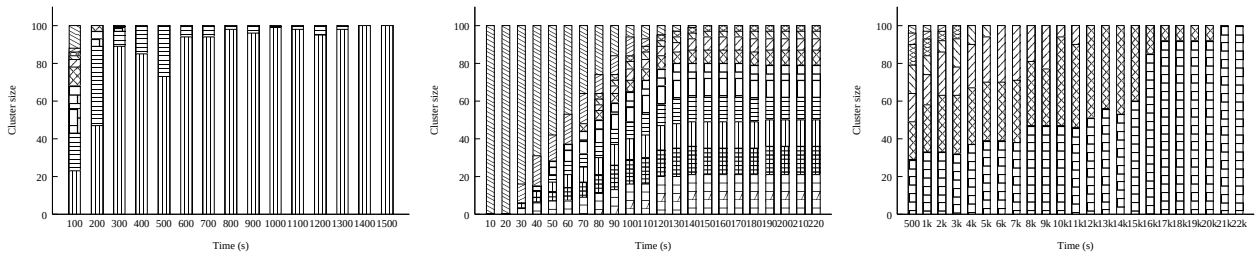


Fig. 8. Cluster sizes on a network of 100 nodes. (a) WBS: cluster size evolution for 15% packet loss rate. All the nodes sync after 1400s. (b) FPD: Cluster size evolution with perfect channel conditions. When the network stabilizes (time 220s), there are 9 clusters of synchronization. (c) FPD: Evolution of cluster sizes with a 15% packet loss rate. The entire network syncs in about 21000s.

10.4 Effects of Malicious Attacks

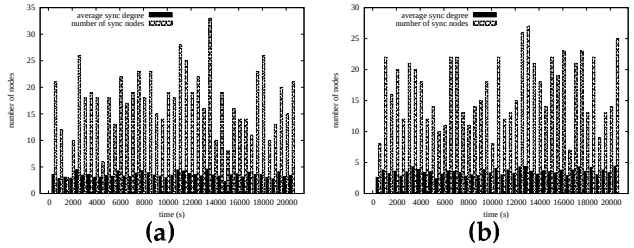


Fig. 10. FPDR: average degree of synchronization and the number of nodes synced with all their neighbors for the (a) tower and (b) dispersion attacks. Even though at times many nodes are synced, it does not last. Even between successive measurements, the membership of sets of synced nodes changes significantly.

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 device identifier. 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 synced. The second metric is the number of *fully synced nodes*, that is, the nodes that are synced with all their neighbors. Figure 10 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 sync the network, its success is only temporary. We have performed numerous, longer than 20000s experiments and neither synchronization nor stabilization was ever achieved.

10.5 RBA Resilience to Attacks

We study RBA under tower, dispersion and cascade attacks. We implement a cascade attack where the attacker uses a fixed list of 10 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 (10 rating) changes its slot and ends up with a 0 rating.

In order to allow RBA to build transmission statistics, each node first runs FPDR, then later, at a predefined time, it 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 11(a) shows the average sync degree corresponding to each attack for these networks and Figure 11(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 11(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%.

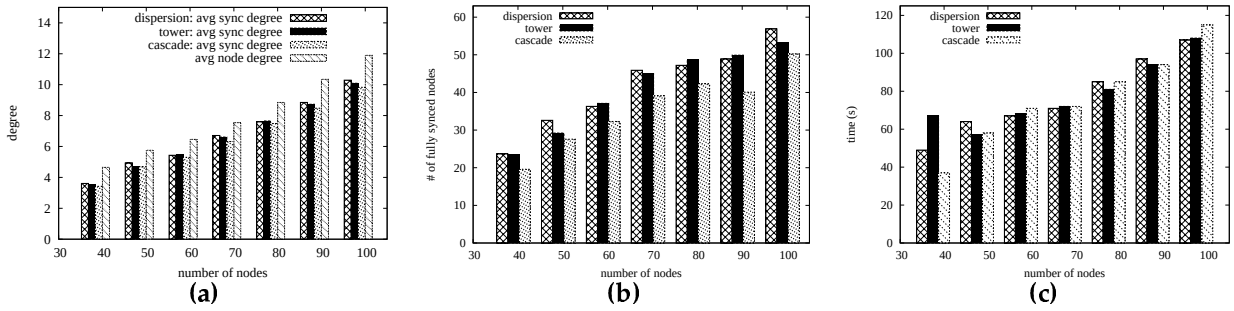


Fig. 11. Effects of attacks launched by a single node against RBA in networks of 40 to 100 nodes. (a) The average sync degrees achieved by RBA during a dispersion (first bar), tower (second bar) and cascade (third bar) attack. The fourth bar is the network's degree. The average sync degrees are very close to the network's degree. (b) The number of fully synced nodes achieved by RBA under dispersion, tower and cascade attacks. Around 50% of the nodes are fully synced. (c) Time required by RBA to stabilize synchronization. Sub-linear increase with network size: 100 nodes stabilize in under 2 minutes.

The average sync degree of nodes is very close to the network's degree, showing that on average, nodes are synced 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 11(c) shows the time required by nodes running RBA to reach a stable state of synchronization (see Definition 4.2). For all three attack types, RBA stabilizes the synchronization process of 100 nodes in less than 120 seconds.

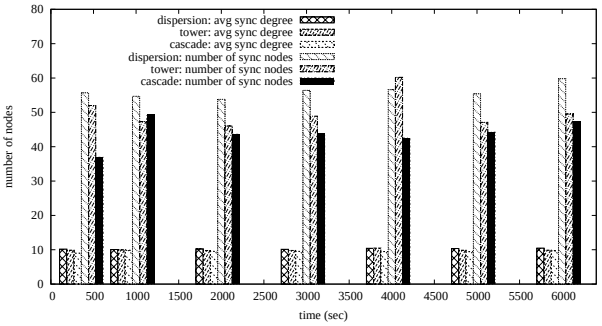


Fig. 12. RBA on a 100 nodes network, when the switch time between FPDR and RBA increases from 400 to 6000 seconds. No significant advantage for longer wait times.

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 12 shows the average sync degree (the small bars) and the number of fully synced nodes (the higher bars) for this experiment, 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 sync 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).

11 Conclusion

We study the problem of synchronizing the periodic transmissions of nodes in ad hoc networks, in order to enable battery lifetime extensions without missing neighbor's updates. We first propose several solutions, both lightweight and scalable but vulnerable to attacks. We then extend them to use transmission stability as a metric for synchronization. Our implementation and simulations show that our protocols are computationally inexpensive, provide significant battery savings, are scalable and efficiently defend against attacks.

12 Acknowledgments

We would like to thank the anonymous reviewers for their excellent 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." [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] B. Carbunar, M. Pearce, S. Mohapatra, L. Rittle, and V. Vasudevan, "Byzantine resilient synchronization for content and presence updates in manets," in *Proceedings of IEEE International Conference on Network Protocols (ICNP)*, 2008.

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

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

[7] 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.

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

[9] 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.

[10] 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.

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

[12] 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.

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

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

[15] 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.

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

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

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

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

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

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

[26] 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.

[27] 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.

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

[29] 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.

Bogdan Carbunar is a principal staff researcher in the Applied Research Center of Motorola, where he is working on video on demand technologies. He received his BS in Computer Science from Politehnica University of Bucharest, Romania in 1999, and a PhD Degree in Computer Science from Purdue University in 2005. His research interests include various aspects of security, such as secure data outsourcing and electronic payments. He is a member of IEEE.

Michael Pearce attended Iowa State University and the University of Illinois at Chicago. He is a Distinguished Member of Technical Staff at Motorola's Applied Research Center and a Motorola Science Advisory Board Associate. He is currently leading a team investigating content caching, analytics, and context aware distributed systems.

Shivajit Mohapatra received his Ph.D. in Computer Science from the University of California at Irvine in 2005. Since then, he has been working at the Applications Research Center at Motorola. His research interests include distributed middleware systems, mobile computing and game theory.

Loren J. Rittle received his B.S. degree in computer engineering from Iowa State University in 1990. Concurrent with a premature departure from graduate school (UIUC) in 1992, he joined Motorola's Communication Sector Research to work on the infamous iDEN digital radio system; eventually, as the chief software architect of the research editions of the packet data subsystem and contributor to its final over-the-air protocol definition. He is currently a principal staff member of the Motorola Applied Research Center investigating the use of Android for stationary deployments. He is named inventor on five issued US patents, many additional and non-US filings. He is a named maintainer for the GNU GCC project (2001-) with primary responsibility for the FreeBSD port and rationalization of thread support in libstdc++-v3. He served on IEEE 1451.[05] and the ZigBee WSA application profile working groups.

Venu Vasudevan is Senior Director for software Platforms Research at the Motorola Applied Research Center and Adjunct Assistant Professor at Rice University's Department of Electrical and Computer Engineering. He leads research efforts on media delivery architectures for advancing television and mobile platforms with specific emphasis on social media platforms, novel advertising experiences and mobile interactive entertainment. His specific research areas include platforms and applications that leverage distributed computing to create new media and entertainment technologies. Before joining Motorola in 1999 he has received a Ph.D. in Computer Science from Ohio State University and a B.S. from the Indian Institute of Technology in Electrical and Computer Engineering. Dr. Vasudevan has served as a speaker and panelist at industry events such as Yankee Group, Digital Hollywood and the Pelorus Group summits. He has published more than 30 conference and journal publications and has three issued patents. He won the best paper award at IEEE Percom 2009 and received recognition from the Hawaii International Conference and the ACM Convergence on Small and Personal Computers.

Octavian Carbunar is Senior Researcher at NIPNE-HH Bucharest, Romania. He has received his B.S. from the EE Department of the Politehnica University Bucharest in 1960, when he joined the Computing Center, Institute for Atomic Physics (NIPNE-HH). He coordinated the team that implemented CIFA-4, from the first generation of computers, with tubes and drums. Between 1963-69, he participated in the design and implementation of IFAC-1, a second generation multiprogrammed machine, with a rate of 1Mips. In the '70s he worked extensively on IBM 370 mainframes, developing firmware and improving algorithms. He received his Ph.D. in 1983 on improving performance using microprogramming. In the 90s he has collaborated with CERN to implement CERN graphical libraries. His current research interests lie at the intersection of earthquake simulation and parallel and grid computing.