Soft-TDMAC: A Software-based 802.11 Overlay TDMA MAC with Microsecond Synchronization

Petar Djukic, *Member, IEEE,* Prasant Mohapatra, *Fellow, IEEE,*

Abstract—We implement a new software-based multi-hop TDMA MAC protocol (Soft-TDMAC) with microsecond synchronization using a novel system interface for development of 802.11 overlay TDMA MAC protocols (SySI-MAC). SySI-MAC provides a kernel independent message based interface for scheduling transmissions and sending and receiving 802.11 packets. The key feature of SySI-MAC is that it provides near deterministic timers and transmission times, which allows for implementation of highly synchronized TDMA MAC protocols. Building on SySI-MAC's predictable transmission times we implement Soft-TDMAC, a software based 802.11 overlay multi-hop TDMA MAC protocol. Soft-TDMAC has a synchronization mechanism, which synchronizes all pairs of network clocks to within microseconds of each other. Building on pairwise synchronization, Soft-TDMAC achieves tight network-wide synchronization. With network-wide synchronization independent of data transmissions, Soft-TDMAC can schedule arbitrary TDMA transmission patterns. For example, Soft-TDMAC allows schedules that decrease end-to-end delay and take end-to-end rate demands into account. We summarize hundreds of hours of testing Soft-TDMAC on a multi-hop test-bed, showing the synchronization capabilities of the protocol and the benefits of flexible scheduling.

Index Terms—Multi-hop TDMA MAC, 802.11 Overlay MAC, Network Synchronization

1 Introduction

New applications of wireless multi-hop networks such as Voice-over-IP and other audio/video streaming services require medium access control (MAC) with guaranteed Quality-of-Service (QoS). While the IEEE 802.11 protocol is a de facto standard for multi-hop wireless networks, its Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) MAC performs poorly in multi-hop wireless networks and cannot provide guaranteed QoS [2]. To achieve guaranteed QoS, one needs to resolve packet collisions, which are the main cause of 802.11's CSMA/CA problems. An effective way to resolve packet collisions is to use synchronized multi-hop MAC protocols, which remove collisions by scheduling interfering links in non-overlapping time or frequency intervals.

Current standardization efforts favour synchronized multi-hop MAC protocols. IEEE is currently working on the 802.11s[3] multi-hop MAC protocols, which has a synchronized mode. IEEE and the 3rd Generation Partnership Project (3GPP) are independently developing synchronized multi-hop protocols for cellular networks.

These industry efforts mean that issues of protocol development, resource management, and dynamic scheduling for synchronized multi-hop protocols will come to the forefront of research. While theories in these research areas can be developed mathematically and tested with simulations, to influence industry, researchers need more convincing arguments, showing that theories can be implemented in practice. It is unlikely that the equipment implementing new standards can be modified to develop synchronized protocols for research purposes. Customizable hardware [4] is prohibitively expensive. The alternative is to use commodity 802.11 hardware to develop synchronized TDMA overlay MAC protocols.

So far, 802.11 synchronized overlay TDMA MACs have proven difficult to implement due to the lack of tight synchronization. Without tight synchronization, TDMA protocols resort to using large gaps between transmissions to prevent collisions, decreasing the efficiency of the protocol. We solve the synchronization problem and provide a new software platform for implementing synchronized MAC protocols with commodity 802.11 hardware.

We design and implement a system interface for 802.11 overlay TDMA MAC protocols under the Linux operating system (SySI-MAC) and show its effectiveness with a new software-based TDMA MAC protocol (Soft-TDMAC). SySI-MAC provides a simple message-based interface for overlay MAC protocol implementations to schedule transmissions, send packets, and receive packets. SySI-MAC intercepts network packets from the Linux kernel and passes them to the overlay MAC protocol, which enqueues them for a period of time, until it requests SySI-MAC to transmit them. To transmit the overlay MAC packets, SySI-MAC disables the 802.11 CSMA/CA functionality and embeds the packets into 802.11 broadcast frames. Disabling CSMA/CA ensures that wireless transmissions have predictable transmission times, necessary for highly synchronized TDMA MAC protocols.

Building on SySI-MAC's predictable transmission times, we implement Soft-TDMAC, a software-based multi-hop TDMA MAC protocol. The main feature of Soft-TDMAC

- *P. Djukic is with MeshIntelligence Inc., Ottawa, Canada e-mail: petar.djukic@meshintelligence.ca*
- *P. Mohapatra is with Department of Computer Science, University of California – Davis, One Shields Ave., Davis, CA, 95616, U.S.A., e-mail: prasant@cs.ucdavis.edu*
- *This research was supported in part by the National Science Foundation through the grants CNS-0831914 and CNS-0709264, and by the US Army Research Office through the MURI grant W911NF-07-1-0318.*
- *Work was performed while the first author was a postdoctoral researcher at the University of California, Davis.*

A preliminary version of this work [1] was presented at the IEEE INFOCOM 2009, Rio de Janeiro, Brazil.

is its tight synchronization. Tight synchronization is necessary to minimize guard times between transmissions and to increase protocol efficiency. Soft-TDMAC synchronizes pairs of nodes to within one, 16 μs, TDMA slot and builds a synchronization tree for network-wide synchronization. We show that by building the synchronization tree, Soft-TDMAC simplifies synchronization in the network and minimizes the worst case synchronization error.

To ensure that nodes can join the network without causing packet collisions, Soft-TDMAC has a carefully designed network entry procedure, which roughly synchronizes the entrant nodes before they transmit any packets. Soft-TDMAC also provides a layer-2 routing mechanism and forwards packets over multiple-hops. For further efficiency, Soft-TDMAC packs smaller network packets into larger wireless transmissions.

Soft-TDMAC is especially appropriate for medium size mesh networks, where the issues of TDMA scheduling does not come into play. In mesh networks, access points aggregate traffic from clients before it enters the mesh; one client leaving or joining the network does not cause a very large variation in inner mesh traffic. So, from the point of view of the mesh the traffic is close to constant, making TDMA scheduling easy. More importantly, Soft-TDMAC can be used to devise and validate more elaborate scheduling techniques appropriate for networks with highly variable traffic.

We summarize hundreds of hours of testing Soft-TDMAC on a multi-hop test-bed. Our experimental results show that Soft-TDMAC synchronizes multi-hop networks to within a few microsecond sized TDMA slots. With no collisions and in good channel conditions, TCP achieves almost the full channel bandwidth. To show how the lack of tight synchronization affects TCP, we experiment with different frame sizes. We emulate protocols without tight synchronization by increasing transmission and frame durations. Our experiments show that the TCP rate decreases with larger frame sizes. We also show that with flexible TDMA scheduling Soft-TDMAC can take advantage of TDMA schedules, which reduce end-to-end delay and increase end-to-end rates.

1.1 Related Work

Software based TDMA MAC protocols have been previously proposed for commodity 802.11 networks [5], [6], [7], [8], [9], [10], [11], [12]. With the exception of the protocol in [12], these protocols are not tightly synchronized [5], [6], [7], sometimes use external synchronization [8], [9], or do not support TDMA schedules independent of the synchronization mechanism [10], [11].

Without tight synchronization, TDMA protocols need to use large slots to introduce gaps between transmissions and prevent collisions. This approach was used to implement TDMA-like 802.11 overlay MAC protocols, avoiding the need for good synchronization [5]. A TDMA MAC with tight pairwise synchronization, provided through wired connections, is proposed in [8], [9]. However, without bounds on transmission times this protocol still uses large TDMA guard times.

With restrictions on types of schedules allowed by the MAC, it is possible to implement collision-free TDMA MAC protocols over commodity 802.11 hardware without perfect synchronization [10], [11]. In 2P [10] and its derivatives [11], collisions are prevented with a token passing mechanism. After a node transmits, it passes the token to one of its peers, which then transmits while the other node receives. This approach limits the types of schedules allowed by the protocol and complicates routing in the network [13].

Achieving microsecond precision network-wide synchronization, which is essential for development of efficient TDMA MAC protocols, has proven to be a hard problem. While protocols such as the Network Time Protocol (NTP) [14] may be able achieve synchronization to within about 100 μs in wired networks, and may even synchronize the network to about 1 μs using the so-called post-facto synchronization [15], they do not work in 802.11 wireless networks. The main problem in achieving precise clock synchronization in 802.11 networks is in estimating one-way propagation delay between pairs of wireless nodes, which is variable due to the 802.11 CSMA/CA collision-avoidance mechanism. Specialized network synchronization algorithms running on 802.11 based wireless networks are able to guarantee synchronization to within milliseconds [16] or hundreds of microseconds [17]. With customizable Mica Berkeley node wireless hardware, the precision of single-hop synchronization can be brought to about 20 μs [18].

One way to remove the variability of 802.11 delays is to to time-stamp packets just before they are transmitted [11], [12]. Our approach is to disable the 802.11 CSMA/CA by changing the 802.11 QoS parameters, ensuring that wireless transmissions have predictable transmission times and one-way propagation delays are almost constant.

Recently, it was shown that a beacon flood can synchronize 802.11 networks in the microsecond range [12]. This synchronization mechanism requires prior knowledge of the network topology and transmission delays between all pairs of nodes. The authors provide a heuristic for the NP-complete problem of finding a good sequence of beacon re-transmissions. By comparison, Soft-TDMAC's synchronization protocol requires no prior knowledge of the network topology or pairwise delays, and self-adapts its synchronization for the conditions in the network.

With the Atheros "MadWiFi" driver [19] it is also possible to build software MAC research platforms [6], [7], [8], [9]. We use the driver in this paper, however our dependence on the MadWiFi driver is weaker than [6], [7], [8], [9]. Our software does not use of any special hardware features, e.g. hardware timers [9], and it does not directly bind to the driver. It only relies on the 802.11 QoS features provided by the MadWiFi driver, which are also available in other wireless drivers.

1.2 Contributions

A main contribution of this work is the design and implementation of a new multi-hop TDMA protocol with very tight synchronization – Soft-TDMAC. Due to the lack of tight synchronization, many TDMA MAC protocols [5], [6], [7], [8], [9], [10], [11] built on top of 802.11 hardware use millisecond long TDMA slots, which decreases protocol efficiency and results in long frame sizes affecting upper layers. Soft-TDMAC uses phase-locked loops (PLLs) for pairwise synchronization and builds a minimum-hop network-wide synchronization tree to achieve tight network-wide synchronization. We show that building the minimum-hop synchronization tree minimizes the worst case synchronization error in the network. With tight synchronization, Soft-TDMAC uses microsecond sized TDMA slots, which make it very efficient.

Since Soft-TDMAC nodes are tightly synchronized, Soft-TDMAC can schedule transmissions with arbitrary transmission patterns, allowing for testing of TDMA scheduling algorithms. This allows us to compare two very different scheduling strategies for multi-hop TDMA networks on a test-bed. In the first scheduling strategy, which we call odd-even scheduling, pairs of nodes alternate their transmissions with equal transmission times, while in the second strategy, which we call minimum-delay scheduling, transmissions times and schedules are chosen to minimize the end-to-end scheduling delay and maximize end-to-end throughput. Scheduling delay occurs when packets are forwarded from an inbound link to an outbound link, but the outbound link was scheduled to transmit first in the frame [20].

We also develop SySI-MAC, a general system interface for development of other TDMA MAC protocols over commodity 802.11 hardware. SySI-MAC allows for implementation of new MAC protocols in the Linux userspace, making it easier to use than previous approaches, which required development of protocols in the Linux kernel [6], [7], [8], [9], or even wireless card firmware [12]. Since most of the SySI-MAC interface does not depend on specialized kernel calls, it is also available for the ns-2 simulator [21], making it easy to prototype and debug new TDMA MAC protocols, before the integration testing with hardware.

The rest of the paper is organized as follows: we describe the Soft-TDMAC protocol in Sec. 2 and its synchronization mechanism in Sec. 3; we describe SySI-MAC in Sec. 4; we describe our test-bed and show the performance results of Soft-TDMAC in Sec. 5; finally we conclude in Sec. 6.

2 Soft-TDMAC Multi-hop TDMA MAC

We now describe the Soft-TDMAC multi-hop MAC protocol, showing its frame structure, control information, and neighbour and route discovery. We explain Soft-TDMAC's synchronization mechanism in the next section.

Soft-TDMAC network boots up when the first node comes online. We use the 802.16 mesh protocol jargon and call this node "base-station". The base-station provides a timing reference for the rest of the network and presents a natural gateway to the wired network. Soft-TDMAC does not prevent any network traffic flow patterns, however in all of our experiments we use the base-station as the gateway.

We use the convention that the base-station is node 0, while the rest of the nodes are numbered from 1 to `MAXNODEID`. In the sequel we assume that $\texttt{MAXNODEID} = 31$ corresponding to the parameters used in our tests. This parameter can be changed at compile time. Assignment of node numbers is independent of node locations and does not affect the running of the routing and synchronization algorithms.

Soft-TDMAC divides the time into T_s second long TDMA slots and groups the slots into fixed size frames. Each frame consists of N_f slots, for a frame duration of $T_f = N_f T_s$ seconds. The first N_c slots in the frame are reserved for network beacons (control sub-frame); the last $N_d = N_f - N_c$ slots in the frame are used for data traffic (data sub-frame). The TDMA parameters T_s, N_f, and N_c are configurable during the network boot-up, but stay fixed while the network operates. In the sequel, we assume that $T_s = 16$ μs, corresponding to the value used in our tests.

Each transmission contains the Soft-TDMAC header containing the length of the packet, sender's node number, the link number, and for data packets, the data sub-header. The link number and the node number uniquely identify a directional link, originating at the sender. The link number of all binary 1's indicates broadcast transmissions and is reserved for network beacons. The data sub-header contains the node number of the final destination of the packet and optionally the information about the subsequent data packets, which are packed in the same transmission. So, one Soft-TDMAC data transmission may carry a number of smaller IP packets to increase the protocol efficiency. The maximum Soft-TDMAC payload is 2012 bytes, corresponding to the limitations of using 802.11 as the physical layer technology.

Soft-TDMAC nodes transmit network beacons in the control sub-frame. Network beacons are 48 bytes long and are transmitted at the lowest modulation rate. Soft-TDMAC allocates 20 TDMA slots for each beacon transmission, where 16 of those slots are used for slack. While this amount of slack is not strictly required due to Soft-TDMAC's tight synchronization, we still use it to make network beacon transmissions robust during network entry and resilient to synchronization errors.

Network beacons contain information advertising the sender's neighbours. A "neighbour" refers to all of the nodes that the sender is aware of, including the nodes that the sender cannot hear from directly. For each neighbour, the beacon contains the sender's hop count to neighbour, used for building the routing tables, and synchronization hop count to the base-station, used for building the network synchronization tree. Since a beacon can only carry information about a limited number of neighbours, each node keeps a circular list of its neighbours and cycles through it to ensure that all neighbours are eventually advertised.

The hop count to each advertised neighbour is sufficient information for a distance vector routing algorithm similar to the Routing Information Protocol (RIP) [22]. Soft-

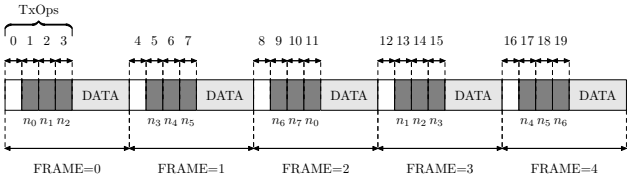


Fig. 1. Network Beacon Schedule.

TDMAC maintains a next-hop minimum-distance routing table for the nodes it is aware of and forwards packets based on that routing table.

Soft-TDMAC sends network beacons in the control sub-frame with a fixed schedule. There are no restrictions on the type of schedules in the data sub-frame. However, in our experiments we group all transmissions of the same link as continuous sets of TDMA slots to increase protocol efficiency. We discuss scheduling in the data sub-frame in the context of our results.

The schedule in the control sub-frame is fixed. Each frame has $\texttt{CTRL\_LEN} = \lfloor N_c/20 \rfloor$ control sub-frame transmission opportunities (TxOps), where $\lfloor \cdot \rfloor$ is the floor function. The first TxOp in the control sub-frame is silent for all nodes and is used to run the clock synchronization algorithm (Fig. 1). Using the same TxOp to run the synchronization algorithm ensures that all nodes arrive at their synchronization decision at about the same time. This approach also puts a hard cap on how long the synchronization algorithm is allowed to run at 320 μs. We tested the algorithm to ensure it never goes over the cap, so we are sure that its running never interfered with transmissions.

After the first TxOp, the next $\texttt{CTRL\_LEN} - 1$ control TxOps are active and are used to send network beacons. Node n_i transmits its network beacon every $\texttt{CTRL\_REUSE}^{\text{th}}$ TxOp, where CTRL_REUSE is a configurable parameter. If there are M nodes in the network, then $\texttt{CTRL\_REUSE} \geq M$. Node n_i transmits in each TxOp j for which

$$i = (j - \texttt{FRAME} - 1) \pmod{\texttt{CTRL\_REUSE}}, \qquad (1)$$

where

$$\texttt{FRAME} = \left\lfloor \frac{j}{\texttt{CTRL\_LEN}} \right\rfloor,$$

counts the number of frames since network boot-up, and $j \geq 0$ is the number of TxOps since network boot-up such that $j \pmod{\texttt{CTRL\_LEN}} \neq 0$.

For example, if $\texttt{CTRL\_LEN} = 4$ and $\texttt{CTRL\_REUSE} = 8$, node n_4 transmits in control TxOp $j = 6$, corresponding to the 3rd TxOp in frame 1, then again in control TxOp $j = 17$, corresponding to the 2nd TxOp in frame 4, and so on.

Each network beacon is time-stamped with the current TDMA frame and control sub-frame TxOp number, which is sufficient information to find time-offset between nodes required by the synchronization algorithm. The number of bits used to transmit the frame and slot information is limited to 28 and 4 bits, respectively. The synchronization algorithm takes the number of available bits into account while findings the difference in time-stamps to ensure that there are no overflow errors.

3 Soft-TDMAC Synchronization

One way to synchronize the network is for all nodes to find clock offsets to the base-station and synchronize to it. However, we show that without estimation errors finding the clock offset to the base-station is equivalent to finding pairwise clock offsets between nodes on a path to the base-station. We use this observation and build a synchronization tree where all pairs of nodes are synchronized to each other. We also show that synchronizing nodes along the minimum hop path to the base-station minimizes the worst-case synchronization to the base-station for each node along the path, which prompts us to build a minimum-hop synchronization tree to the base-station.

Our approach is different from the approach in the Network time Protocol (NTP) [14], which estimates the clock offset to the clock source over multiple hops. NTP's approach is not possible in Soft-TDMAC for two reasons. First, the clock information is in the control sub-frame, which has limited bandwidth to ensure the protocol is efficient. Second, transmitting the clock offsets over multiple hops introduces additional random delays, making it harder to obtain the synchronization level required for efficient TDMA MAC protocols.

First, we explain our network time model. Then we show that the minimum synchronization tree minimizes the worst-case synchronization error and explain how the tree is built. We explain other components of Soft-TDMAC synchronization next: pairwise time-stamp exchange that finds pairwise clock offsets and phase-lock loop (PLL) used for pairwise synchronization. Finally, we explain the network procedure, which is required to prevent packet collisions before the pairwise exchange of packets synchronizes pairs of nodes.

3.1 Network Clock Model

We use the oscillator clock model [23], where each node's system clock is derived from an oscillator with the output

$$u(t) = \cos\left[2\pi \int_0^t f(\tau) d\tau\right], \quad t > 0,$$

where $f(t)$ is the oscillator's instantaneous frequency and t is the time. The oscillator's instantaneous frequency consists of a fixed nominal frequency $F_{\texttt{nom.}}$ and a time varying phase $\vartheta(t)$,

$$f(t) = F_{\texttt{nom.}} + \vartheta(t).$$

The instantaneous phase $\vartheta(t)$ represents the time variability of the oscillator, so the oscillator time model comprises all possible instantaneous frequencies that change randomly over time (for example due to changes in environmental temperature). The exact distribution of the instantaneous frequency is not necessary for any derivations in this work. In the sequel, we use the nominal frequency $F_{\texttt{nom.}} = 10^9$ Hz,

corresponding to the nanosecond precision of the Linux system clock.

System time is obtained by counting the number of oscillator cycles. Node j's system time is

$$C_j(t) = F_{\texttt{nom.}}\, t + \theta_j(t), \tag{2}$$

where $\theta_j(t) = \int_0^t \vartheta_j(t)d\tau$ is the residual phase of the node's oscillator at time t, and $\vartheta_j(t)$ is the time varying phase of node j's oscillator. Without any loss of generality, in the sequel we assume that $\theta_j(0) = 0$ for all nodes j.

The network time is generated from the clock at the base-station with

$$T_0(t) = C_0(t) = F_{\texttt{nom.}}\, t + \theta_0(t), \tag{3}$$

where $C_0(t)$ is the system time of the base-station (node 0) at time t, and $\theta_0(t)$ the residual phase of the base-station's oscillator.

All other nodes j generate local network time from their system clock with

$$T_j(t) = C_j(t) - \Delta_{0,j}(t-\varepsilon), \tag{4}$$

where $\Delta_{0,j}(t-\varepsilon)$ is the clock offset between node j's local network time and the network time at the base-station, found ε seconds before the current time,

$$\Delta_{0,j}(t-\varepsilon) = T_j(t-\varepsilon) - T_0(t-\varepsilon) = \theta_j(t-\varepsilon) - \theta_0(t-\varepsilon).$$

For small ε, $\lim_{\varepsilon\to 0} \left|\theta_j(t-\varepsilon) - \theta_j(t)\right| = 0$ and

$$C_j(t) - \Delta_{0,j}(t-\varepsilon) \underset{\varepsilon\to 0}{=} F_{\texttt{nom.}}\, t + \theta_0(t),$$

meaning the a node can synchronize itself to the base-station if it can estimate its clock offset to the base-station accurately and timely.

3.2 Synchronization Tree

We show that estimating clock offset to the base-station is equivalent to estimating pairwise clock offsets between pairs of nodes on a path to the base-station. With estimation errors, synchronizing pairs of nodes along the minimum hop path to the base-station minimizes the worst-case synchronization of each node to the base-station.

Soft-TDMAC estimates pairwise clock offsets with a pairwise time-stamp exchange in the network beacons, which is explained in detail next. Here we assume that after the k^{th} beacon period, pairs of nodes j and i have available a clock offset estimate

$$\Delta_{ij}(kT_p) = \theta_j(kT_p) - \theta_i(kT_p),$$

which is valid during $(k+1)^{\text{th}}$ beacon period, time $kT_p \le t < (k+1)T_p$, where the network bacon period T_p is approximately

$$T_p \approx \frac{\texttt{CTRL\_REUSE}}{(\texttt{CTRL\_LEN}-1)} T_f, \tag{5}$$

$\texttt{CTRL\_LEN} - 1$ is the number of beacon transmissions in a frame, and CTRL_REUSE controls how often nodes transmit in the control sub-frame (1).

Suppose that nodes are labeled $n_0, n_1, \ldots, n_j$, along a path. Node n_1, on the first hop, has the clock offset to the base-station (node n_0) in after just one beacon period (k^{th} beacon period). Its network time is

$$\begin{aligned} T_1(t) &= F_{\texttt{nom.}}\, t + \theta_1(kT_p) - \Delta_{01}(kT_p) + \varepsilon_1 \\ &= F_{\texttt{nom.}}\, t + \theta_0(kT_p) + \varepsilon_1 \end{aligned}$$

for $kT_p \le t \le (k+1)T_p$, where $\theta_j(kT_p)$ is the residual phase of node j at time kT_p, and ε_1 represents the error in the clock offset estimate and clock drift. The residual phase of node n_1 in the next beacon period, $(k+1)T_p \le t \le (k+2)T_p$, is

$$\theta_1\Big((k+1)T_p\Big) = \theta_0(kT_p) + \varepsilon_1.$$

After the beacon period k node n_2, at the second hop, knows the clock offset node to node n_1, but this clock offset does not include the adjustment we just showed. However, after the $(k+1)^{\text{th}}$ beacon period that adjustment is propagated and the time two hops away from the base-station at node n_2 is

$$\begin{aligned} T_2(t) &= F_{\texttt{nom.}}\, t + \theta_2\Big((k+1)T_p\Big) - \Delta_{12}\Big((k+1)T_p\Big) + \varepsilon_2 \\ &= F_{\texttt{nom.}}\, t + \theta_0\Big((k+1)T_p\Big) + \varepsilon_1 + \varepsilon_2 \end{aligned}$$

for $t \ge (k+2)T_p$, where ε_2 represents estimation and clock errors in the second hop.

Generalizing, after j beacon periods, the network time at node n_j, j hops away from the base-station, is

$$T_j(t) = F_{\texttt{nom.}}\, t + \theta_0(kT_p) + \sum_{i=1}^{j} \varepsilon_i$$

for $t \ge (k+j+1)T_p$, where ε_i estimation and clock errors for hop i. So, j hops away from the base-station the synchronization error is

$$\|T_j(t) - T_0(t)\| = \left\| \sum_{i=1}^{j} \varepsilon_i \right\| \le j\varepsilon_{\text{clock}},$$

where $\varepsilon_{\text{clock}}$ is the maximum single-hop synchronization error. We show the origins of synchronization errors next.

Given the above development, we conclude that using the pairwise clock offsets is equivalent to finding the clock offset to the base-station and that the minimum-hop synchronization tree minimizes the worst case synchronization error. The above result is our motivation to synchronize pairs of nodes in the network according to a hierarchy based on a shortest path synchronization tree. We note that stronger results than this can be obtained with first-order clock drift models [24]. However, clock modeling and analysis is beyond the scope of this work.

Soft-TDMAC builds a minimum hop synchronization tree from the synchronization hop count broadcast in network beacons. Synchronization hop count is used to run distance vector algorithm on each node and find the shortest synchronization path to the base-station, similar to the Soft-TDMAC routing algorithm. The collection of these paths is the synchronization tree in the network.

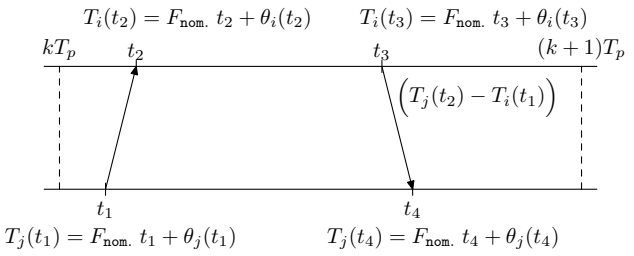


Fig. 2. Pairwise Time-stamp Exchange.

3.3 Pairwise Clock Offsets

Soft-TDMAC estimates pairwise clock offsets with a time-stamp exchange in the network beacons. To simplify notation, in the sequel we use $[\cdot]$ to emphasize the discrete nature of some quantities, $\Delta_{ij}[k] \triangleq \Delta_{ij}(kT_p)$, where T_p is the network beacon transmission period (5).

In beacon period k, the slave sends a network beacon at time $t_1 \geq kT_p$, with its local network time $T_j(t_1) = F_{\text{nom.}}\ t_1 + \theta_j(t_1)$ (Fig. 2). The master slave relationship is determined from the synchronization tree – the node closest to the base-station is the master. After receiving the network beacon at time t_2, the master uses its network time $T_i(t_2) = F_{\text{nom.}}\ t_2 + \theta_i(t_2)$ to find the network time difference to node j, $T_i(t_2) - T_j(t_1)$. Some time later, the master sends a network beacon at time t_3, with its network time $T_i(t_3) = F_{\text{nom.}}\ t_3 + \theta_i(t_3)$ and the previously found network time difference. After receiving the beacon, at time $t_4 \leq (k+1)T_p$, the slave finds its network time difference to the master, $T_j(t_4) - T_i(t_3)$.

After the exchange, the slave node estimates its clock offset to the master with

$$\begin{aligned}\Delta_{ij}[k] =& \frac{1}{2}\left[\left(T_j(t_4) - T_i(t_3)\right) - \left(T_i(t_2) - T_j(t_1)\right)\right] \\ =& \theta_j[k] - \theta_i[k] + \frac{1}{2}\left[\varepsilon_{\text{drift}}^{(j,i)} - \varepsilon_{\text{drift}}^{(i,j)}\right] + \frac{1}{2}\varepsilon_{\text{prop}},\end{aligned} \tag{6}$$

where $\theta_i[k] \triangleq \theta_i(kT_p)$ and $\theta_j[k] \triangleq \theta_j(kT_p)$ are the residual phases at the slave and the master at time $t = kT_p$, respectively,

$$\varepsilon_{\text{drift}}^{(j,i)} = \theta_j(t_4) - \theta_j(kT_p) - \left[\theta_i(t_3) - \theta_i(kT_p)\right]$$

is the difference between node j's clock drift in the time period kT_p to t_4 and node i's clock drift in the time period kT_p to t_3, and

$$\varepsilon_{\text{drift}}^{(i,j)} = \theta_i(t_2) - \theta_i(kT_p) - \left[\theta_j(t_1) - \theta_j(kT_p)\right]$$

is the difference between node i's clock drift in the time period kT_p to t_2 and node j's clock drift in the time period kT_p and t_1, and

$$\varepsilon_{\text{prop}} = F_{\text{nom.}}\ [(t_4 - t_3) - (t_2 - t_1)]$$

is the difference in one-way propagation delays.

In the unlikely scenario that the one-way propagation delays are the same and that there is no clock drift, (6) calculates the exact clock offset between the master and the slave at the beginning of the exchange. However, this is not the case in general.

The errors due to clock drift are negligible since high manufacturing standards are used even for stock hardware platforms used in our test-bed. For example, a relatively large 500 μs/s clock drift [14] translates to a clock drift of a few microseconds in each millisecond long T_p interval.

We minimize the errors due to the differences in propagation delay with careful engineering, which ensures low propagation error for most transmissions. Nevertheless, as we observe later, the Atheros 802.11 hardware used on our test-bed sometimes inexplicably delays data packets, which causes asymmetrical propagation delays with relatively large propagation errors. The software detects clock offsets with asymmetrical propagation delays from the round-trip propagation time and eliminates them from consideration for synchronization.

The slave finds the round-trip delay with

$$\begin{aligned}\delta_{ij}[k] &= [T_j(t_4) - T_i(t_3)] + [T_i(t_2) - T_j(t_1)] \\ &= F_{\text{nom.}}\ \left[(t_4 - t_1) - (t_3 - t_2)\right] + \left[\varepsilon_{\text{drift}}^{(j,i)} + \varepsilon_{\text{drift}}^{(i,j)}\right],\end{aligned}$$

where the first term in the last summation is the round-trip delay and $\varepsilon_{\text{drift}}^{(i,j)}$ and $\varepsilon_{\text{drift}}^{(j,i)}$ are defined as before. Since the slave expects the round-trip delay to be

$$\delta_{ij}[k] \approx 2 * T_{\text{TxOp}} \gg 2 * \|\varepsilon_{\text{drift}}^{\max}\| \geq \varepsilon_{\text{drift}}^{(j,i)} + \varepsilon_{\text{drift}}^{(i,j)},$$

where T_{TxOp} is the duration of network beacon transmissions, $\varepsilon_{\text{drift}}^{\max}$ is the maximum clock drift in period T_p, and $\delta_{ij}[k] \geq 0$, it can easily detect clock offset estimates with asymmetrical delays. For $T_{\text{TxOp}} = 320$ μs used in our tests, valid return trip times are $\delta_{ij}[k] \approx 640$ μs.

3.4 Software Phase-Lock Loop

The raw pairwise clock offsets are input into a software PLL to smoothly synchronize clocks. The PLL drives the slave's clock offset to 0 as the number of pairwise exchanges increases. The PLL averages clock offsets over the last w_u time-exchanges and uses the smoothed clock offset to update times (SYNCHRONIZE-TO-MASTER).

The algorithm has two major parts. The first part of the algorithm (Steps $1-4$) calculates the average clock offset $\overline{\Delta}_{ij}[k_u]$ for the current re-synchronization period from clock offset estimates $\Delta_{ij}[k]$. The re-synchronization period is different from the network beacon period and is demarcated by the updates of the local clock offset (Step 7). Before adding a clock offset to the average, the algorithm checks the validity of it's return-trip time (Step 1). Clock offset estimates with return-trip times less then 0 or greater than $\delta_{\text{prop}}^{\max} = 800$ μs are ignored. The algorithm adds valid clock offset estimates to the running average of the clock offsets in Step 3 to obtain the average clock offset from w_u valid offsets

$$\overline{\Delta}_{ij}[k_u] = \frac{1}{w_u}\sum_{k=1}^{w_u}\Delta_{ij}[w_u(k_u - 1) + k], \tag{7}$$

for the re-synchronization period k_u.

The second part of the algorithm (Steps $5-15$) uses the averaged clock offset to update the node's clock off-

Algorithm 1 SYNCHRONIZE-TO-MASTER($\Delta_{ij}[k]$, $\delta_{ij}[k]$)

Initialize globals: $l = 0$, $k_u = 1$, $\overline{\Delta}_{ij}[0] = 0$, $w_u = W_{\min}$

1: **if** $0 < \delta_{ij}[k] < \delta^{\max}_{\text{prop}}$ **then**
2: $l \leftarrow l + 1$
3: $\overline{\Delta}_{ij}[k_u] \leftarrow \overline{\Delta}_{ij}[k_u] + (\Delta_{ij}[k] - \overline{\Delta}_{ij}[k_u])/l$
4: **end if**
5: **if** $l = w_u$ **or** $\overline{\Delta}_{ij}[k_u] > \Delta_{\max}$ **then**
6: $k_u \leftarrow k_u + 1$ /* $k_u = k/w_u$, if $\overline{\Delta}_{ij} \leq \Delta_{\max}$*/
7: $\hat{\Delta}_{ij}[k_u] \leftarrow \hat{\Delta}_{ij}[k_u - 1] - \overline{\Delta}_{ij}[k_u - 1]$
8: $l \leftarrow 0$
9: **if** $\overline{\Delta}_{ij}[k_u] > \Delta_{\max}$ **then**
10: $w_u \leftarrow \max\{w_u/2, W_{\min}\}$
11: **else**
12: $w_u \leftarrow \min\{w_u + 1, W_{\max}\}$
13: **end if**
14: $\overline{\Delta}_{ij}[k_u] \leftarrow 0$
15: **end if**

set for the current re-synchronization period, $\hat{\Delta}_{ij}[k_u]$. The algorithm only updates local clock offsets every $w_u{}^{\text{th}}$ time it is run (Step 5), or if the average clock offset is greater than the target clock offset $\Delta_{\max} = T_s/2 = 8$ μs. We note that using the target of $\Delta_{\max} = 8$ μs is sufficient to schedule any transmission pattern with the granularity of one 16 μs TDMA slot.

The update window, w_u, is determined based on how far the slave's clock is from the master's clock (Steps $9-13$). The update window is an integer in the range $w_u \in [W_{\min}, W_{\max}]$. In our tests we use $W_{\max} = (5\text{ s})/T_p$, to ensure the maximum time between clock updates of 5 s, and $W_{\min} = 1$, to ensure that at least clock offset estimate is available before the update. If the algorithm finds that the clock offset to the master is larger than $\Delta_{\max}$, it halves the update window (Step 10). On the other hand, if the algorithm finds that the clock offset is smaller than $\Delta_{\max}$, the update window is increased by one frame (Step 12).

The effect of the shorter, or longer, update window is seen from the software-based PLL model of the algorithm (Fig. 2). The low-pass filter $F(z, w_u)$ corresponds to z-transform of the averaging of clock offsets on Step 3 of the algorithm (7). Updating the local clock offset in Step 7 of the algorithm corresponds to the integration element $D(z)$. We note that the PLL is a multi-rate system as indicated by the two samplers operating at different frequencies. Since the update window changes the behaviour of the low-pass filter, it becomes a parameter governing the performance of the PLL.

We simulate the performance of the software PLL in Matlab. The performance results can also be derived analytically, however that type of analysis is beyond the scope of this work. The simulation starts with the slave and master perfectly synchronized and then the master's clock is pushed forward by some amount. Fig. 4 shows the behaviour of the slave's clock offset, in terms of the percentage of the original clock offset, as the number of clock exchanges increases. Clock offset eventually goes to zero, meaning that the algorithm synchronizes the clocks over time.

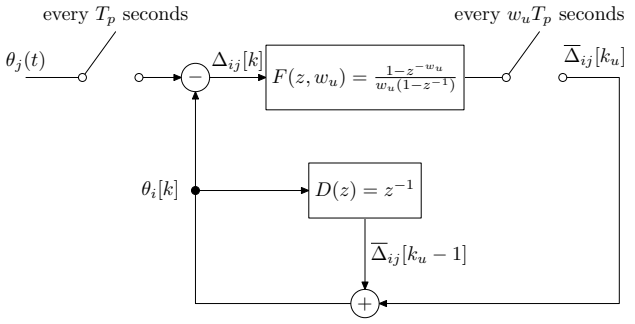


Fig. 3. Software PLL Control System Model.

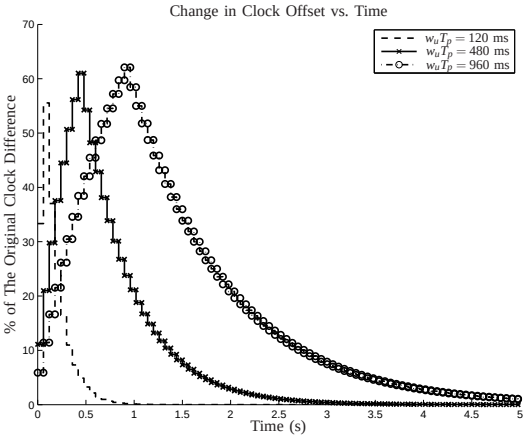


Fig. 4. Performance of the Software PLL.

Considering Fig. 4, Steps 9-13 in the synchronization algorithm can be interpreted as follows: if the clocks are far from each other, the algorithm aggressively decreases the settling time of the PLL to speed up the synchronization; on the other hand, if the clocks are close to each other, the algorithm cautiously increases the settling time of the PLL to filter out the noise in the clock offset estimates.

3.5 Network Entry

Initially, a node is *unsynchronized*. An unsynchronized node collects network beacons for the maximum allowed re-synchronization time. While unsynchronized, a node estimates the clock offset to the synchronized nodes by assuming a one-way propagation delay of 15 slots (240 μs). After the re-synchronization time expires, the unsynchronized node uses the estimated clock offset to synchronize its clock and declares itself *roughly synchronized*. Even though at this point the node is not fully synchronized to the network, network beacon transmissions have enough slack (120 μs), to allow it to transmit network beacons without collisions.

After the node is roughly synchronized, is starts to transmit network beacons and uses synchronization algorithm for precise synchronization. After 20 runs of the synchronization algorithm, the node declares itself *synchronized*

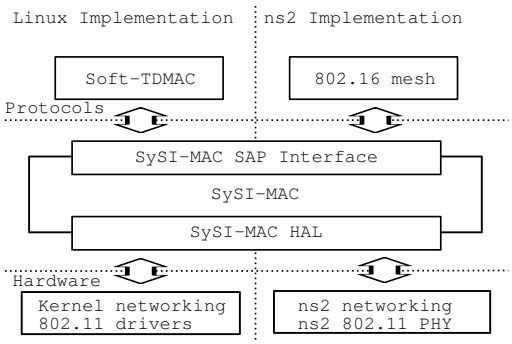


Fig. 5. Soft-TDMAC Architecture.

and starts to send data packets. Waiting for 20 runs of the synchronization algorithm limits the total duration of network entry to at most 105 seconds (5 seconds to achieve rough synchronization and at most $20 * 5$ seconds to achieve full synchronization). However, since in practice the re-synchronization period is always smaller than the maximum 5 seconds, network entry typically takes 20 to 30 seconds.

4 SySI-MAC Linux Architecture

The SySI-MAC system interface for 802.11 overlay TDMA MAC protocols, is positioned between an overlay protocol implementation and the 802.11 MAC implemented either in real hardware or in a simulator environment (Fig. 5). SySI-MAC's message-based Service Access Point (SAP) interface provides a simple way to add and implement new overlay protocols, while the Hardware Abstraction Layer (HAL) provides a uniform access to the 802.11 MAC implementations across different technologies for easy porting of SySI-MAC.

In the SySI-MAC's Linux implementation parts of the code reside in the Linux userspace, while the rest reside in the Linux kernel. The SAP interface and the implementation of hardware independent functions are implemented as a userspace library. A part of the HAL resides in the userspace, while the other part resides in the kernel. The userspace part of the HAL provides timers and inter-process communications with the part of the HAL residing in the kernel. The kernel part of HAL is implemented as a kernel module and provides a direct link to the kernel's network service, which manages the 802.11 device drivers.

The relationship between the components is most easily explained with a traversal of a network packet. When the kernel module receives an IP packet from the Linux kernel IP networking stack, it forwards it to the userspace part of the HAL. The SAP then hands the packet to the overlay MAC, which process it, enqueues it and requests a timer interrupt from the SAP for the next packet transmission. After some time, the HAL creates a timer interrupt and delivers it to the overlay MAC, which then hands over the packet to the SAP as a transmission request. The userspace part of the HAL hands-off the packet to the kernel module, which requests the kernel's networking service to send the packet over the 802.11 hardware.

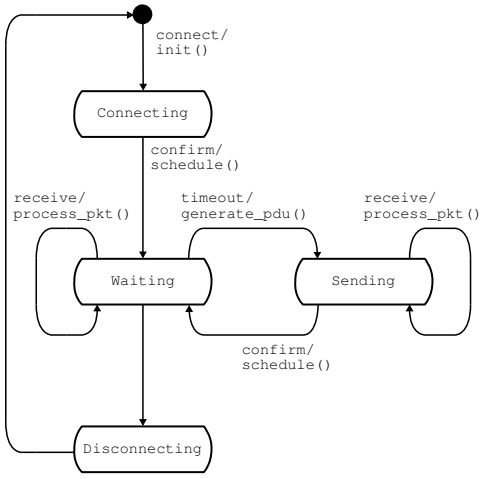


Fig. 6. Connection State Diagram.

We have previously implemented SySI-MAC's SAP and HAL in ns-2 [26]. In this work, the kernel module and a POSIX based userspace library implement a Linux version of the HAL. We rely on the real-time extensions to the Linux Kernel [27] for microsecond precision timers.

There are three reasons for dividing SySI-MAC architecture into userspace and kernel components. First, since the majority of the library is in userspace, with few dependencies on its environment, it is possible to plug the library into a network simulator [26], or to port it to environments other than Linux, which support POSIX threads [28]. Second, putting the overlay MAC code into the userspace removes the difficulties of programming in the kernel due to the lack of full debugging facilities, standard system routines and the kernel's limited, undocumented, and constantly changing threading and timer interfaces. Three, by putting the most important components of our software into the userspace, we avoid tying future overlay MAC protocol implementations to the restrictive GNU Public License [29].

4.1 SySI-MAC Service Access Point Interface

Due to space constraints, we cannot provide the full details of the SySI-MAC's SAP interface and SySI-MAC's HAL implementation. These can be automatically generated from the SySI-MAC's source code [30]. We only describe the interface briefly to indicate how it formalizes the development of new TDMA MAC protocols.

The key idea of the SAP interface is to use link layer "connections" as an abstraction for a TDMA MAC protocol. We use the term connection to associate transmissions with schedules and to signify pairwise wireless links.

A new TDMA MAC protocol is implemented by creating new connection types, which implement functionality specific to the protocol. Each connection implements a connection state machine, which allows it to use SySI-MAC SAP interface. Fig. 6 shows connection state machine (the message received by the connection and the procedure that SySI-MAC calls on the connection after the message is processed are denoted with `message/procedure()`).

Connections start in the `Connecting` state. For a connection that implements network entry, the `Connecting` state is the state during which it synchronizes to the network. On the other hand, data connections may use the `Connecting` state to send control messages to establish pairwise links between wireless nodes.

After a connection finishes with the `Connecting` state it enters the `Waiting` state where it receives packets or timer events. If the connection receives a timer event in the `Waiting` state, it enters the `Sending` state, where it sends its packets. After the hardware confirms the sending of the packet, the connection returns to the `Waiting` state. We note that the state machine allows connections to receive packets in the `Sending` state, even though that may be an obvious hardware, software or synchronization error. With this approach, the SAP interface lets the overlay MAC handle serious errors.

The final state of each connection is the `Disconnecting` state. The intention of this state is to allow connections to release memory and other resources when they are no longer needed.

New protocols are implemented by implementing the `process_pkt()`, `generate_pkt()` and `schedule()` procedures for new connection types. The `process_pkt()` procedure receives the packets from the wireless hardware or the kernel's network stack and depending on the type of connection it may send these packets further up the stack, enqueue them, generate new packets, or perform management procedures. The `generate_pkt()` procedure is called in response to a timer event, which was previously requested by the `schedule()` procedure. This procedure requests an immediate packet transmission, with the option of specifying physical layer parameters. The `generate_pkt()` function have a short execution time to ensure that the delay from the timer to when the packet is transmitted is small and non-variable. In our protocol implementation, we achieve this goal by pre-generating all overlay MAC outgoing packets in the `process_pkt()` function, which has almost no time constraints.

The `schedule()` function allows connections to implement their schedules. It is called after a connection exits the `Connecting` state and every time the connection exits the `Sending` state to allow a connection to request the timer message for a future transmission.

4.2 TDMA Hardware Abstraction

SySI-MAC provides a "virtual" TDMA hardware abstraction to its overlay MAC protocol clients. The hardware abstraction maps overlay MAC transmissions into slotted time transmissions (Fig. 7), taking into account the fact that the overlay MAC's packets are transmitted as the payload of actual 802.11 packets. To simplify scheduling in the overlay MAC protocol, SySI-MAC virtual TDMA slots are grouped into fixed length frames. Each TDMA slot carries a fixed number of bytes, which depends on the 802.11 physical modulation rate used during packet the transmission.

Overlay MAC transmissions have two parts in virtual time. The virtual *guard time* covers hardware and software

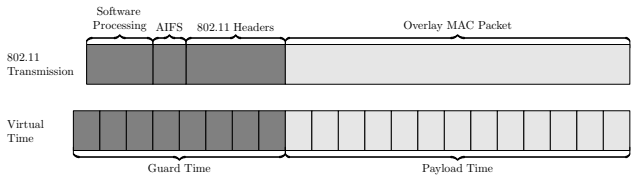


Fig. 7. Virtual TDMA Time.

overheads, while the virtual *payload time* covers the time required to transmit an overlay MAC packet without 802.11 overhead (Fig. 7). The overheads include the software processing time, which corresponds to how long it takes from the timer event requesting the transmission to the actual request for a transmission, the 802.11 arbitration inter-frame space (AIFS) pause, which occurs after 802.11 hardware determines that the channel is free, and the time required to transmit 802.11 protocol headers. Abstracting all overhead times into a single guard time, allows for TDMA protocols agnostic of the underlying transmission technology.

While the 802.11 transmission time is fixed by hardware on all systems, the guard time is not. In the simulator environment, the guard time can be found theoretically by closely considering the 802.11 standard [26]. However, on the Linux system, the guard time depends on the software processing time – how fast the processor is and how quickly the context switch between timer interrupts and SiSY-MAC occurs. We determine guard times experimentally, as shown in the next section.

5 Test-bed Results

We summarize hundreds of hours of evaluating Soft-TDMAC on a 4 node test-bed (Fig. 8). All nodes in the test-bed where situated on a typical office desk, with the distance between nodes of less than 1 m. Due to the physical size of the test-bed, all nodes can hear transmission from all other nodes.

Test-bed nodes are Hewlett-Packard nc6000 laptops, which use the Pentium M processors running at around 1.5 GHz. The laptops come pre-installed with the wireless cards using the Atheros Communications AR5212 chipset. We use the MadWiFi driver [19]. We add profiling software to the driver, which allows us to measure 802.11 transmission times. We also add code to change per-packet 802.11 modulation rates. Since we access the MadWiFi driver through the Linux networking sub-system, we effectively change only about 10 lines of code in the driver.

The laptops run Linux kernel 2.6.23 [25] with the real-time extensions [27]. The Linux real-time extensions streamline the kernel to remove unnecessary software locks and provide preemptive priority-based thread scheduling, which is necessary for precise software timers. The system software is installed with the Gentoo Linux software distribution. The userspace uses the POSIX real-time thread implementation provided by glibc-2.6.1. All software is compiled with gcc-4.1.2.

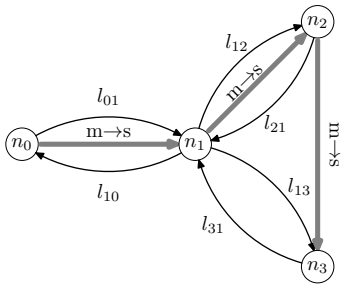


Fig. 8. Logical Topology of the Test-bed.

We use the 802.11a wireless channel 100, operating at 5.5 GHz, which we found to be free of other wireless nodes. We set the 802.11 AIFS time and the 802.11 contention window to the lowest values allowed by the underlying hardware. These parameters can be modified with the newer Linux wireless drivers such as the MadWiFi driver [19] for the Atheros wireless chipsets. Since Soft-TDMAC requires fixed transmission times for synchronization, we disable the 802.11 retransmission mechanism.

We use TDMA slots with the duration $T_s = 16\mu$s, corresponding to 4 802.11a Orthogonal Frequency Division Multiplexing (OFDM) symbols [31]. With these settings, a TDMA slot can carry 12, 24, 36, 48, 64, 96, or 108 bytes, corresponding to the 802.11a's 6 Mbps, 12 Mbps, 18 Mbps, 24 Mbps, 36 Mbps, 48 Mbps and 54 Mbps modulation rates, respectively.

5.1 Empirical Tuning of TDMA Guard Times

To obtain virtual guard times, we perform series of experiments where a node transmits 20,000 packets at 5 ms intervals. We repeat experiments for different packet lengths and modulation rates. We use our modifications of the MadWiFi driver to record the time between the userspace timer and the hardware interrupt indicating that the card finished sending the packet. At the lowest 802.11a modulation rate of 6 Mbps, the longest packet transmission of 2012 bytes takes around 3.0 ms, so each transmission ends before the next transmission begins.

Fig. 9 shows the duration of packet processing time (t_{proc}), 802.11 hardware transmission time (t_{tran}) and the total time to send the packet ($t_{\text{send}} = t_{\text{proc}} + t_{\text{tran}}$) for a 1 second portion of one of the runs for 48 byte packets transmitted at 6 Mbps. This experiment corresponds to network beacon transmissions. We note that t_{proc} is almost constant with a negligible amount of variability (less than a few microseconds). Enabling the QoS features of the MadWiFi driver also ensures that t_{tran} is almost constant.

We note that sometime around 67 s into this experiment the card experienced what we call a "hiccup" – it inexplicably delayed the packet for a long time (> 1 ms). From our observations, we know that hiccups happen after the packet is already on the Atheros card and that they do not happen due to 802.11 back-off. We believe that the Atheros card periodically performs a maintenance function that delays

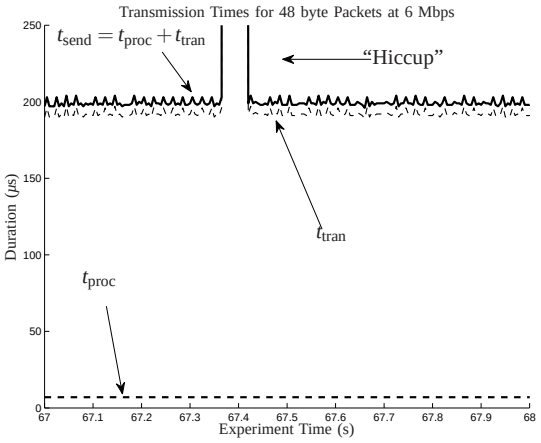


Fig. 9. Transmission Delays.

the packets. We have confirmed this phenomenon on three types of Atheros cards.

We have found that the hiccups are a very rare occurrence that did not significantly affect the experiments we show later in the paper. The filtering mechanism in the synchronization algorithm is sufficient to remove the hiccups as a factor in the synchronization.

Using our measurements over various packet sizes (Fig. 10), we determine an envelope, which always covers the duration of packet transmissions

$$\hat{\varepsilon}_m(l) = 11 * T_s + \left\lceil \frac{l}{b_m} \right\rceil * T_s, \tag{8}$$

where $\hat{\varepsilon}_m(l)$ is the envelope for an l byte overlay packet transmitted at modulation m, $b_m \in \{12, 24, \ldots, 108\}$ is the number of bytes carried in a TDMA slot for modulation m, $T_s = 16$ μs is the slot duration, and $\lceil \cdot \rceil$ is the ceiling function, which rounds a number to the nearest integer higher than the number. This envelope is shown as "max envelope" in Fig. 10.

Observing our data, we found the maximum envelope is actually very pessimistic – most of the data transmissions are shorter than the maximum envelope. So, we find a tighter envelope based on statistical properties of data.

Using the measured times for the 48 byte network beacons transmitted at the lowest rate and our knowledge of 802.11 operation, we find an envelope $\varepsilon_m(l)$, which covers most of the transmission times. Consider the transmission of a 48 byte packet. The measured 802.11 transmission time for the 48 byte packets is $t_{\text{trans}} = 192$ μs on average, with the standard deviation of 2 μs. There are $h_{802.11} = 36$ bytes of 802.11 overheads in each transmission. Since the wireless card broadcasts 84 bytes of data, at 6 Mbps the transmission should only take 112 μs, which corresponds to 7, 16 μs TDMA slots. So, the Atheros hardware adds around 80 μs of delay to each transmission. We believe 40 μs of that delay corresponds to the 10 802.11a OFDM symbols of preamble specified by the standard [31] and the other 40 μs correspond to the Atheros "post-transmission back-off" [9]. On average, the processing time is $t_{\text{proc}} = 7$ μs

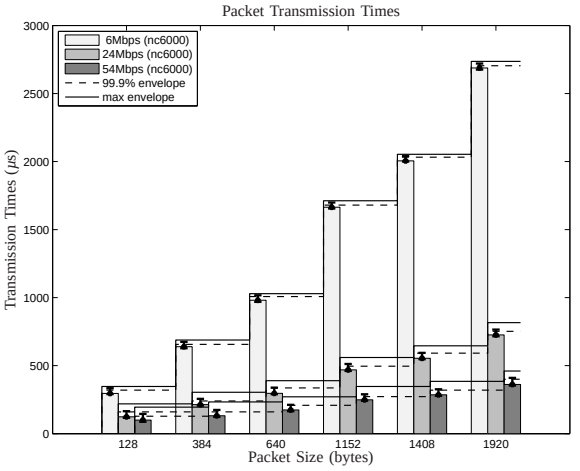


Fig. 10. Envelope on Transmission Times.

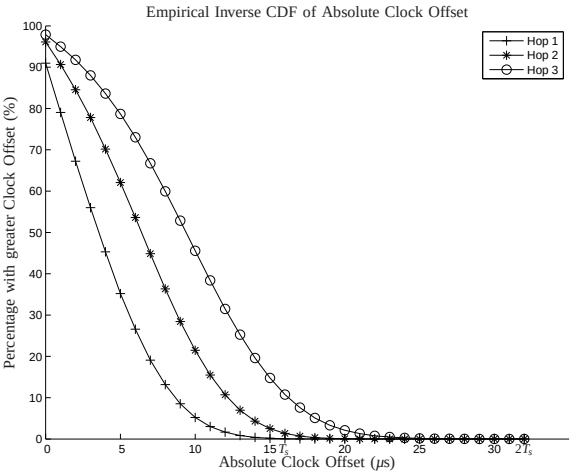


Fig. 11. Multi-hop Synchronization.

with the standard deviation of 0.2 μs.

Using the measured 802.11 transmission times and the processing times, we calculate the average sending time for a 48 byte packet as

$$t_{\text{send}} = t_{\text{proc}} + t_{\text{trans}} = 7\ \mu\text{s} + 80\ \mu\text{s} + 7 * 16\ \mu\text{s} < 13 * 16\ \mu\text{s}.$$

So there are 7 TDMA slots for the payload time and 6 slots for the guard time to cover 87 μs for hardware and software processing.

Generalizing for all packet sizes, the new envelope is

$$\varepsilon_m(l) = 6 * T_s + \left\lceil \frac{l+36}{b_m} \right\rceil * T_s, \qquad (9)$$

where $\varepsilon_m(l)$ is the envelope for an l byte overlay packet transmitted at modulation m, b_m is the number of bytes carried in a TDMA slot for modulation m, T_s is the slot duration.

We test the envelope against a subset of measured transmission times (Fig. 10). The envelope, (9), is shown as the envelope "99.9% envelope" and indeed covers 99.9% of the packets.

Since the 99.9% envelope, (9), is more efficient than the maximum envelope, we use it for data transmissions. For example, for transmissions at 54 Mbps, the 99.9% envelope transmissions are 62.5% shorter than the maximum envelope transmissions for packet size of 128 bytes and 20% shorter for the maximum packet size of 2012.

In order to make the Soft-TDMAC protocol robust, we make the network beacon transmissions very robust. We use the maximum envelope, (8), to find that each network beacon transmission takes 15 TDMA slots. In addition, we also add 5 extra guard symbols for the total network beacon transmission time of 320 μs. The average transmission time for a 48 byte packet is about 200 μs, so network beacons can tolerate synchronization errors of up to 120 μs.

5.2 Evaluation of Soft-TDMAC Synchronization

We setup an experiment on the test-bed to measure the clock offset between the nodes in the network. In this experiment, the frame size is 20 ms, the control sub-frame is $N_c = 50$ slots long, for `CTRL_LEN` $= 2$ TxOps in each control sub-frame, and we set the control sub-frame re-use factor to `CTRL_REUSE` $= 8$. With these settings the time-stamp exchange period $T_p = 160$ ms. We have also run Soft-TDMAC with other values of `CTRL_LEN` and `CTRL_REUSE`, with similar results.

TABLE 1
Absolute Clock Offset Statistics

Statistic	Hop 1	Hop 2	Hop 3
Mean	3.70 ± 0.03 μs	6.83 ± 0.04 μs	9.71 ± 0.04 μs
Std. Dev.	4.21 ± 0.02 μs	4.57 ± 0.02 μs	5.59 ± 0.03 μs
99 prct.	13 μs	17 μs	22 μs
99.9 prct.	16 μs	20 μs	26 μs
Maximum	58 μs	84 μs	56 μs

Since all nodes are in the range of each other, using the minimum synchronization tree, each node would normally select the base-station (node n_0) as its clock master. However, we manually set the synchronization tree to $n_0 \to n_1 \to n_2 \to n_3$ (thick arrows in Fig. 8), to test clock synchronization over multiple hops. We run the network continuously for about 5 hours. The synchronization algorithm has run on average every 1617 ms on the first hop, 1096 ms on the second hop and 323 ms on the third hop.

Fig. 11 shows the inverse empirical Cumulative Density Function (CDF) of the absolute value of the filtered clock offsets, measured at the base-station for node n_1 ("Hop 1"), node n_2 ("Hop 2") and node n_3 ("Hop 3"). We show absolute values of clock offsets since both negative and positive synchronization errors affect transmissions. There are well over a $100,000$ clock offsets collected by the base-station to the other nodes during this period. We note that 10% of clock offsets on the first hop were 0 μs - so the minimum synchronization error is 0 μs. We also note that most clock offsets at any hop are less than 32 μs (2 TDMA slots).

Table 1 shows detailed statistics of the experiment (all intervals in the table are 99.9% confidence intervals). The

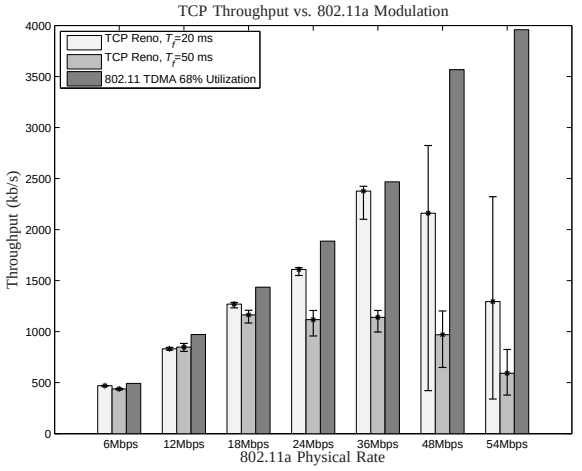


Fig. 12. One-hop TCP Throughput.

mean absolute clock offset for all three hops is less than 10 μs. We note that less than 0.1% clock offsets for the first hop are more than 16 μs and that at three hops, less than 0.1% of clock offsets are greater than 26 μs. The worst case clock offset of 84 μs occurred on the second hop. Out of the $100,000$ collected offsets, there was only one clock offset this high.

We conclude that the synchronization algorithm works well. With high confidence, it synchronizes a pair of nodes to within 1 TDMA slot and that it can synchronize all nodes in a 3 hop network to within 2 TDMA slots.

5.3 Single-hop TCP Performance

We test Soft-TDMAC single-hop TCP performance by turning off nodes n_2 and n_3 and only using nodes n_0 and n_1. We try frame sizes $T_f = 20$ ms and $T_f = 50$ ms. The longer frame size emulates protocols without tight synchronization, which use longer transmissions. For both frame durations, we allocate 12% of the frame to the control sub-frame, 20% of the frame to the uplink and 68% of the frame to the downlink. For the frame duration $T_f = 20$ ms, we allow for 4 control TxOps, while for $T_f = 50$ ms, we allow 15 control sub-frame TxOps. We allocate 850 of 1250 available data slots and 2125 of 3125 available data slots on the downlink, for $T_f = 20$ ms and $T_f = 50$ ms, respectively. The rest for the slots are used for the uplink. The control sub-frame re-use factor is $\texttt{CTRL\_REUSE} = 32$.

In order to get statistically valid performance results, we run a series of experiments to test the performance of TCP throughput for two TCP variants: TCP Reno and TCP Westwood [32]. In each experiment, we first start node n_0 and then node n_1. We let node n_1 synchronize and after 30 seconds it initiates a download of 30 Mb of data from node n_0. We repeat this setup 30 times for each of the 7 available 802.11a physical modulation rates and the two TCP variants for a total of 420 experiments.

Fig. 12 shows the average throughput of TCP Reno over all 30 scenarios, for all modulation rates and for both frame sizes. The error bars show the throughput of the run with the best throughput and the run with the worst throughput. The figure also compares the throughput to the theoretical maximum ("802.11 TDMA 68% Utilization"), which we obtain by finding how much throughput the link would have if it continuously transmitted 68%, corresponding to the portion of time allocated on the downlink. Since the delay-bandwidth product increases with the longer frame duration, TCP throughput decreases with longer frame sizes when the channel has errors ($\sim$ 1% 802.11 frame loss at 54 Mbps). We do not show TCP Westwood performance – it performs slightly worse than TCP Reno, at higher modulations.

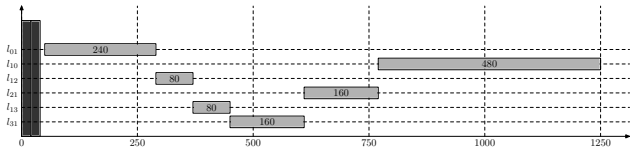
Analyzing the data, we discovered a strong correlation between the decreases in TCP window size and packet losses in the physical layer. This correlation accounts for the fact that the TCP throughput decreases at higher modulation rates. If we only consider the frame duration of 20 ms in Fig. 12, the *average* TCP window size and TCP rate increase until the rate of 36 Mbps. When the rate changes to 48 Mbps, the number of lost 802.11 packets seems to reach a critical point, where the frequency of dropped 802.11 packets severly impacts TCP window size, thus decreasing the average TCP rate. Fig. 12 shows that increasing the frame duration to 50 ms, and consequently round-trip time, makes TCP rate more susceptible to packet losses, accounting for lower average TCP rates. It is unclear to us why this is the case. Soft-TDMAC does not implement an Automatic Repeat ReQuest (ARQ) mechanism, which would decrease frame losses in the link layer and likely improve TCP performance.

We do not believe that the hiccups have affected our results, since we can clearly trace drops in TCP window size to channel errors. We found that the Atheros card hiccups are a relatively rare – the Atheros chipset delays about 1 in every 1200 transmitted 802.11 frames.

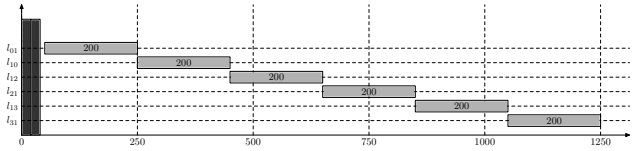
5.4 Multi-hop TCP Performance

We evaluate Soft-TDMAC multi-hop performance using our test-bed (Fig. 8), with the same TDMA parameters used in Sec. 5.2. Since the Soft-TDMAC is a connection oriented link layer protocol, nodes only accept data packets arriving on one of their registered (logical) incoming links. For example, node n_2 ignores all data packets except the ones arriving from node n_1, (link l_{12}); if node n_0 sends an IP packet to node n_2, that packet must traverse node n_1. Node n_0 is the base-station.

We evaluate Soft-TDMAC performance with two types TDMA multi-hop schedules. The first type of schedule, is a minimum TDMA delay schedule [20] (Fig. 13a). This schedule minimizes TDMA scheduling delay for all nodes in the network. TDMA scheduling delay occurs if an outbound link on a router is scheduled to transmit before an inbound link on that router. For the path connecting node n_0 to node n_2, this schedule orders the links $l_{01} \rightsquigarrow l_{12} \rightsquigarrow l_{21} \rightsquigarrow l_{10}$. The schedule maximizes the throughput in the network, but allocates twice as much bandwidth on the uplink (traffic to the base-station), as it does on the downlink to model the asymmetry of traffic in real networks.



(a) Minimum Delay Schedule.



(b) Odd-Even Schedule.

Fig. 13. Multi-hop TDMA Schedules.

TABLE 2
Mean TCP Throughput

Schedule Rates (Mbps)	Min-Delay 6/6/6	Odd-Even 18/6/6	Min-Delay 18/6/6
Node 1	76.9 kb/s	74.3 kb/s	470.1 kb/s
Node 2	67.3 kb/s	68.2 kb/s	68.9 kb/s
Node 3	63.8 kb/s	64.9 kb/s	64.9 kb/s

The second type of schedule is an "odd-even" TDMA schedule [13] (Fig. 13b), which schedules pairs of nodes alternatively. This type of scheduling is called odd-even scheduling because links are either scheduled in even (l_{10}, l_{21}, l_{31}) or odd slots (l_{01}, l_{12}, l_{13}). We note that despite the fact that 2P [10] uses odd-even scheduling, results obtained with Soft-TDMAC using odd-even scheduling cannot be directly compared to 2P. 2P uses multiple interfaces, which increases channel capacity. Also, 2P slot allocations are longer than the link durations in Fig. 13b. Consistent with odd-even scheduling [10], [13], all links are allocated the same number of slots.

The minimum delay schedule adjusts link bandwidths to take into account that links l_{01} and l_{10} carry traffic of 3 nodes. Since all links have the same time allocation in the odd-even schedule, links l_{01} and l_{10} present a bottleneck for nodes n_2 and n_3, which have excess bandwidth available on their links to node n_1. To compare the two schedules in terms of throughput, we fix the rate on all links in the minimum delay schedule at 6 Mbps, while setting the rate on links l_{01} and l_{10} to 18 Mbps in the odd-even schedule.

For each schedule, we run an experiment where nodes n_1, n_2 and n_3 start a 3 Mb upload to the base station, at roughly the same time, with node n_1 starting first. We repeat each experiment 30 times. Table 2 show the average TCP throughput, over the 30 runs for each node. In all cases, standard deviation was less than 5 kb/s. The performance of the minimum delay schedules ("Min-Delay (6/6/6)") and the odd-even ("Odd-Even (18/6/6)") is almost the same. The throughput of the three nodes is balanced. Node n_1 gets slightly higher throughput than the other two nodes because it starts first.

Taking advantage of the minimum delay bandwidth adjusted schedule, we also run experiments where we set the rate on the hop between nodes n_0 and n_1 to 18 Mbps. Since node n_1 now has extra bandwidth allocated to it, its bandwidth increases substantially ("Min-delay (18/6/6)"). The bandwidth of the other two nodes stays about the same.

TABLE 3
Measured Round-trip Delay (6 Mbps)

	Min-Delay		Odd-Even	
	Mean	Std. Dev.	Mean	Std. Dev.
Node 1	20.0 ms	0.3 ms	27.0 ms	6.9 ms
Node 2	27.1 ms	7.3 ms	47.0 ms	7.0 ms
Node 3	27.3 ms	7.2 ms	47.0 ms	7.1 ms

To measure the round-trip delay, we send 1000 Internet Control Message Protocol (ICMP) packets from node n_1,n_2, and n_3 to the base-station. The modulation on each link is fixed at 6 Mbps. We use the Linux version of `ping` to find the maximum ICMP flood. Table 3 shows the mean round-trip time, averaged over the 1000 transmitted ICMP packets. We note that the round-trip delay for the odd-even schedule is about two frame sizes for nodes at two hops, due to the fact that packets going from nodes n_2 and n_3 are delayed until the next frame at node n_1. The minimum delay schedule round-trip times are always around about frame duration. We tried this experiment will all available 802.11 physical layer rates and observed similar results.

6 Conclusion

We have implemented the SySI-MAC Linux interface for 802.11 overlay TDMA MAC protocol development, and used it to design and implement Soft-TDMAC, a software-based TDMA MAC protocol. SySI-MAC provides a simple, kernel independent, message based interface for overlay MAC protocol implementations to schedule transmissions, send packets, and receive packets. The key feature of SySI-MAC is that it provides near deterministic timers and transmission times, which allows for implementation of highly synchronized TDMA MAC protocols.

The key feature of Soft-TDMAC is its microsecond synchronization, which enables high TDMA efficiency. Soft-TDMAC has a synchronization mechanism, which synchronizes the network to within a few microsecond-long TDMA slots. The tight synchronization decreases the overhead of transmissions for a very efficient TDMA protocol with short frame durations. We have tested the protocol on our test-bed and shown that when channel conditions are good, TCP can achieve almost full channel bandwidth since there are no packet collisions. Soft-TDMAC's small frame durations help TCP ramp-up to the channel capacity. We have also shown that Soft-TDMAC's flexible scheduling allows for schedules that maximize end-to-end throughput and decrease end-to-end delay.

References

[1] P. Djukic and P. Mohapatra, "Soft-TDMAC: Software TDMA-based MAC over commodity 802.11 hardware," *INFOCOM 2009. The 28th Conference on Computer Communications. IEEE*, pp. 1836–1844, April 2009.

[2] S. Xu and T. Saadawi, "Does the IEEE 802.11 MAC protocol work well in multihop wireless ad hoc networks," *IEEE Commun. Mag.*, vol. 39, no. 6, pp. 130–137, June 2001.
[3] "IEEE P802.11s/D1.01, Draft STANDARD for Information Technology - Telecommunications and information exchange between systems - local and metropolitan area networks- specific requirements- part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications amendment: ESS mesh networking," 2006.
[4] A. Khattab, J. Camp, C. Hunter, P. Murphy, A. Sabharwal, and E. Knightly, "WARP - a flexible platform for clean-slate wireless medium access protocol design," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 12, no. 1, pp. 56–58, January 2008.
[5] A. Rao and I. Stoica, "An overlay MAC layer for 802.11 networks," in *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, 2005, pp. 135–148.
[6] M. Neufeld, J. Fifield, C. Doerr, A. Sheth, and D. Grunwald, "SoftMAC—flexible wireless research platform," in *HotNets*, 2005.
[7] C. Doerr, M. Neufeld, J. Fifield, T. Weingart, D. C. Sicker, and D. Grunwald, "MultiMAC - an adaptive MAC framework for dynamic radio networking," in *First IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN 2005)*, 2005, pp. 548 –555.
[8] A. Sharma, M. Tiwari, and H. Zheng, "MadMAC: Building a reconfigurable radio testbed using commodity 802.11 hardware," in *1st IEEE Workshop on Networking Technologies for Software Defined Radio Networks (SDR '06)*, 2006, pp. 78–83.
[9] A. Sharma and E. M. Belding, "FreeMAC: Framework for multi-channel MAC development on 802.11 hardware," in *PRESTO '08: Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, 2008, pp. 69–74.
[10] B. Raman and K. Chebrolu, "Design and evaluation of a new MAC protocol for long-distance 802.11 mesh networks," in *MobiCom '05: Proceedings of the 11th annual international conference on Mobile computing and networking*, 2005, pp. 156–169.
[11] R. Patra, S. Nedevschi, S. Surana, A. Sheth, L. Subramanian, and E. Brewer, "WiLDNet: Design and implementation of high performance wifi based long distance networks," in *Proceedings of 4th USENIX Symposium on Networked Systems Design & Implementation (NSDI'07)*. USENIX, 2007, pp. 87–100.
[12] D. Koutsonikolas, T. Salonidis, H. Lundgren, P. LeGuyadec, Y. C. Hu, and I. Sheriff, "TDM MAC protocol design and implementation for wireless mesh networks," in *CoNEXT '08: Proceedings of the 2008 ACM CoNEXT Conference*. New York, NY, USA: ACM, 2008, pp. 1–12.
[13] G. Narlikar, G. Wilfong, and L. Zhang, "Designing multihop wireless backhaul networks with delay guarantees," in *Proceedings of 25th IEEE International Conference on Computer Communications (INFOCOM 2006)*, 2006, pp. 1–12.
[14] D. L. Mills, *Computer Network Time Synchronization: the Network Time Protocol*. CRC Press, 2006.
[15] J. Elson and D. Estrin, "Time synchronization for wireless sensor networks," in *IPDPS '01: Proceedings of the 15th International Parallel & Distributed Processing Symposium*. IEEE Computer Society, 2001, p. 186.
[16] M. L. Sichitiu and C. Veerarittiphan, "Simple, accurate time synchronization for wireless sensor networks," in *IEEE Wireless Communications and Networking (WCNC)*, vol. 2, 2003, pp. 1266–1273.
[17] K. Römer, "Time synchronization in ad hoc networks," in *MobiHoc '01: Proceedings of the 2nd ACM international symposium on Mobile ad hoc networking & computing*, 2001, pp. 173–181.
[18] S. Ganeriwal, R. Kumar, and M. B. Srivastava, "Timing-sync protocol for sensor networks," in *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, 2003, pp. 138–149.
[19] http://madwifi.org/.
[20] P. Djukic and S. Valaee, "Delay aware link scheduling for multi-hop tdma wireless networks," *IEEE/ACM Transactions on Networking*, vol. 17, no. 3, pp. 870– 883, June 2009.
[21] http://www.isi.edu/nsnam/ns/.
[22] C. Hedrick, "Routing Information Protocol," RFC 1058 (Historic), Jun. 1988, updated by RFCs 1388, 1723. [Online]. Available: http://www.ietf.org/rfc/rfc1058.txt
[23] D. W. Allan, "Time and frequency (time-domain) characterization, estimation, and prediction of precision clocks and oscillators," *IEEE Trans. Ultrason., Ferroelectr., Freq. Control*, vol. 34, no. 6, pp. 647–654, November 1987.
[24] R. Solis, V. S. Borkar, and P. Kumar, "A new distributed time synchronization protocol for multihop wireless networks," in *45th IEEE Conference on Decision and Control*, December 2006, pp. 2734–2739.
[25] "Linux kernel," http://www.kernel.org/, 2006.
[26] P. Djukic and S. Valaee, "Getting the most of WiFi mesh networks with 802.16 mesh emulation," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 23, no. 6, pp. 1744–5760, 2008.
[27] I. Molnar, "Real-time patches for Linux 2.6 kernel," http://www.kernel.org/pub/linux/kernel/projects/rt/.
[28] "The open group base specifications issue 6," IEEE Std 1003.1, 2004.
[29] "GNU general public license," http://www.gnu.org/licenses/gpl.html.
[30] "http://softtdmac.sourceforge.net/."
[31] "IEEE standard for information technology-telecommunications and information exchange between systems-local and metropolitan area networks-specific requirements - part 11: Wireless lan medium access control (MAC) and physical layer (PHY) specifications," 2007.
[32] L. A. Grieco and S. Mascolo, "Performance evaluation and comparison of Westwood+, New Reno, and vegas TCP congestion control," *ACM SIGMOBILE Mobile Computer Communications Review*, vol. 34, no. 2, pp. 25–37, April 2004.

PLACE PHOTO HERE

Petar Djukic (S '01, M '08) received B.A.Sc., M.A.Sc. and Ph.D. degrees from the University of Toronto in 1999, 2002 and 2008, respectively. From 2008 to 2010 we was a postdoctoral researcher at the Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada. From 2007 to 2008 he was a postdoctoral researcher at the Department of Computer Science, University of California, Davis. In 2010 he founded MeshIntelligence Inc., a leader in TDMA software for 802.11-based wireless networks. His research interests are in wireless multi-hop scheduling and resource allocation and test-bed implementations of new wireless MAC protocols. From 1999 to 2001 he worked as a software designer in Ottawa, Canada.

Prasant Mohapatra (S '01, M '08) is currently the Tim Bucher Family Endowed Chair Professor and the Chairman of the Department of Computer Science at the University of California, Davis. In the past, he has been on the faculty at Iowa State University and Michigan State University. He has also held Visiting Scientist positions at Intel Corporation, Panasonic Technologies, Institute of Infocomm Research (I2R), Singapore, and National ICT Australia (NICTA). He has been a Visiting Professor at the University of Padova, Italy and Yonsei University, South Korea. He was/is on the editorial board of the IEEE Transactions on Computers, IEEE Transactions on Mobile Computing, IEEE Transaction on Parallel and Distributed Systems, ACM WINET, and Ad Hoc Networks. He has been on the program/organizational committees of several international conferences. He served as the Program Vice-Chair of INFOCOM 2004 and the Program Chair of SECON 2004, QShine 2006, and WoWMoM 2009. He has been a Guest Editor for IEEE Network, IEEE Transactions on Mobile Computing, IEEE Communications, IEEE Wireless Communications, and the IEEE Computer.

Dr. Mohapatra received his doctoral degree from Penn State University in 1993, and received an Outstanding Engineering Alumni Award in 2008. He is a Fellow of the IEEE.

Dr. Mohapatra's research interests are in the areas of wireless networks, sensor networks, Internet protocols, and QoS. Dr. Mohapatra's research has been funded though grants from the National Science Foundation, Department of Defense, Intel Corporation, Siemens, Panasonic Technologies, Hewlett Packard, Raytheon, and EMC Corporation.