Soft-TDMAC: A Software TDMA-based MAC over Commodity 802.11 hardware

Petar Djukic
Department of Systems and Computer Engineering
Carleton University
1152 Colonel By Drive, Ottawa ON, K1S 5B6, Canada
e-mail: djukic@sce.carleton.ca

Prasant Mohapatra
Department of Computer Science
University of California – Davis
One Shields Ave., Davis, CA, 95616, U.S.A.
e-mail: prasant@cs.ucdavis.edu

***Abstract*—We design and implement Soft-TDMAC, a software Time Division Multiple Access (TDMA) based MAC protocol, running over commodity 802.11 hardware. 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 network wide synchronization. With, out-of-band, network wide synchronization Soft-TDMAC can schedule arbitrary TDMA transmission patterns.**

We summarize hundreds of hours of testing Soft-TDMAC on a multi-hop testbed. Our experimental results show that Soft-TDMAC synchronizes multi-hop networks to within a few microsecond sized TDMA slots. Soft-TDMAC can schedule transmissions to take end-to-end demands into account and in a way that decreases end-to-end delay [1], [2]. With no collisions, under good channel conditions, TCP achieves almost the full wireless channel bandwidth.

***Index Terms*—Wireless Mesh Networks, Software TDMA MAC, 802.11**

I. Introduction

New applications of wireless multi-hop networks, such as commercial mesh networks, offering Voice-over-IP and other audio/video streaming services, require guaranteed Quality-of-Service (QoS) in the MAC layer. IEEE 802.11 is a de-facto standard for wireless mesh networks, even though it is well known that its Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) based MAC protocol performs poorly in multi-hop wireless networks [3]. Packet collisions, which are the main cause of 802.11's CSMA/CA problems can be resolved with Time Division Multiple Access (TDMA) based MAC protocols, which schedule interfering links at non-overlapping times.

Despite the advantages, TDMA protocol implementations and testbeds still have a long way to go due to the difficulties of implementing precise packet transmission scheduling. IEEE has ratified and is working on new TDMA-based MAC multi-hop protocols: 802.11s protocol [4], 802.16 mesh protocol [5] and the 802.16 multi-hop relay protocol [6]. It is unlikely that the equipment implementing these standards can be modified to develop new TDMA protocols. The alternative is to use the commodity 802.11 hardware for development of software TDMA based MAC protocols.

We design and implement Soft-TDMAC, a software TDMA based multi-hop MAC protocol, which uses commodity 802.11 hardware under the Linux OS to send and receive wireless packets. Soft-TDMAC intercepts network packets and enqueues them for a period of time, until their wireless link is scheduled for a transmission. To transmit a packet, Soft-TDMAC disables the 802.11 CSMA/CA functionality by setting the appropriate 802.11 QoS parameters, embeds the enqueued packets into 802.11 frames and requests the wireless card to broadcast the packets. Disabling CSMA/CA ensures that wireless transmissions have predictable transmission times, necessary for synchronization.

With the predictable transmission times, Soft-TDMAC synchronizes pairs of nodes to within one, 16μs, TDMA slot. While tight synchronization may be possible in new TDMA protocols [4]–[6], which redesign the wireless hardware, it is more difficult without full access to the hardware [7]–[13]. Building on pairwise synchronization, Soft-TDMAC builds a network synchronization tree, where all node clocks are within a few TDMA slots of each other. With tight synchronization, Soft-TDMAC can use small TDMA guard times to ensure collision-free transmissions.

For further efficiency, Soft-TDMAC can pack smaller network packets into larger wireless transmissions. Soft-TDMAC also provides a layer-2 routing mechanism and forwards packets over multiple-hops.

We summarize hundreds of hours of testing Soft-TDMAC on a multi-hop testbed. Our experimental results show that Soft-TDMAC synchronizes multi-hop networks to within a few microsecond sized TDMA slots. With microsecond synchronization, Soft-TDMAC has no restrictions on scheduling patterns and can schedule transmissions in a way that decreases end-to-end delay. With no collisions, TCP achieves almost the full channel bandwidth, with good channel conditions.

The significance of Soft-TDMAC is twofold. First, even though Soft-TDMAC is built on top of 802.11 commodity hardware, it can achieve microsecond precision network wide synchronization. With tight synchronization, Soft-TDMAC

Work was performed while the first author was a postdoctoral researcher at the University of California, Davis.

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.

uses microsecond sized TDMA slots, which makes it more efficient. Due to the lack of tight synchronization, TDMA MAC protocols, which were built on top of 802.11 hardware had millisecond sized TDMA slots [7]–[13]. Second, since nodes are tightly synchronized, out-of-band, Soft-TDMAC can schedule transmissions with arbitrary transmission patterns. Soft-TDMAC can adjust link allocations based on end-to-end demands and it can schedule transmissions to decrease end-to-end delay.

Soft-TDMAC is built on top of the extensible TDMA Software Abstraction Layer (TDMA-SAL), a system interface for development of new TDMA MAC protocols. TDMA-SAL depends on hardware abstraction layer to send and receive packets. We have previously implemented most of TDMA-SAL and a hardware abstraction layer for the ns-2 network simulator [15], which allowed quickly prototyping of the 802.16 mesh protocol [16]. In this paper, implement a Linux hardware abstraction layer and implement a new software TDMA-based MAC protocol – Soft-TDMAC. The code for the Linux implementation of TDMA-SAL and the SoftTDMAC protocol is available free of charge for academic use [17].

The rest of the paper is organized as follows. Related works are described in Sec. II. We describe the Soft-TDMAC protocol in Sec. III and its hardware and software setup in Sec. IV. In Sec. IV, we also show the results of our TDMA guard time tuning. In Sec. V, we summarize Soft-TDMAC performance on our testbed, followed by the conclusion in Sec. VI.

II. Related Work

TDMA based MAC protocols have been previously proposed for commodity 802.11 networks [7]–[13]. Unlike Soft-TDMAC, these protocols are not tightly synchronized [7]–[9], sometimes use external synchronization [10], [11], or use synchronization independent TDMA schedules [12], [13].

A software overlay TDMA based MAC, with loose TDMA synchronization, for 802.11 hardware is proposed in [7]. With the Atheros "MadWiFi" driver [18], it is also possible to build software MAC research platforms [8]–[11]. We use the driver in this paper, however our dependence on the MadWiFi driver is weaker than [8]–[11]. Our entire MAC is implemented in Linux user space, without the use of any special features of the hardware, e.g. Atheros hardware timers [11]. Soft-TDMAC only relies on the 802.11 QoS features provided by the driver, which are also available in other wireless drivers [19].

Using software platforms [8]–[11], researchers have built TDMA based MAC protocols over commodity 802.11 hardware. In [8], synchronization is achieved with hardware timestamps, provided by the Atheros chipset. A TDMA MAC built on top of the MadWiFi driver is also proposed in [10], [11]. That MAC relies on pairwise synchronization provided through wired connections.

It is possible to implement TDMA based, collision-free, MAC protocols over commodity 802.11 hardware even without perfect synchronization in the network [12], [13]. In 2P [12] and its derivatives [13], each node switches between transmission and reception slots. Nodes re-synchronize after each transmission, so synchronization is done, in-band, through data transmissions. This approach limits the kind of schedules allowed in the protocol. Since Soft-TDMAC uses out-of-band synchronization, it can schedule links in any pattern in the TDMA frame.

Microsecond precision network synchronization, which is essential for development of efficient TDMA MAC protocols, has proven to be a hard problem. In wired networks, NTP achieves synchronization to within about 100μs [20]. Using the so-called post-facto synchronization, which uses extra knowledge of the clock workings, synchronization can be brought to about 1μs [20]. Network synchronization algorithms running on 802.11 based wireless networks are able to guarantee synchronization to within milliseconds [21] or hundreds of microseconds [22]. With customizable Mica Berkeley node wireless hardware, the precision of NTP like single-hop synchronization can be brought to about 20μs [23], however Mica wireless hardware cannot achieve bandwidth required for mesh networks. Over wired connections this protocol synchronizes pairs of nodes with the precision of 25μs [10], [11].

The main problem in achieving precise clock synchronization in 802.11 networks is in estimating the propagation delay between pairs of wireless nodes. In 802.11 networks, this delay is variable due to the 802.11 CSMA/CA collision-avoidance mechanism, which makes the delay unpredictable. One way to deal with this problem is to time-stamp packets just before they are transmitted [13]. On the other hand, we disable the 802.11 CSMA/CA by changing the 802.11 QoS parameters, ensuring that wireless transmissions have predictable transmission times.

III. Soft-TDMAC Multihop TDMA MAC

In this section, we describe the Soft-TDMAC TDMA based MAC protocol, showing its frame structure, neighbour and route discovery, synchronization and network entry procedure. In the next section, we describe the architecture of Soft-TDMAC Linux implementation and how the complete system was tuned to achieve efficient TDMA over commodity 802.11 hardware.

The Soft-TDMAC network boots up when the first node comes online. We use the 802.16 mesh protocol jargon and call this node the "base station". 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.

Soft-TDMAC views the time in terms of slots, T_s seconds long, and groups TDMA slots into fixed size frames. Each frame consists of N_f slots, for a frame size 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 are configurable during the network boot-up, but stay fixed while the network operates.

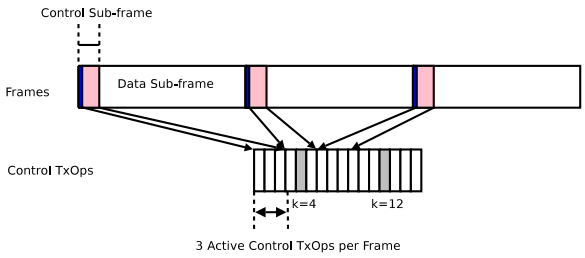


Fig. 1. S-TDMA Framing

Each slot carries a fixed number of bytes, which depends on the physical modulation rate used during the transmission in the slot. The modulation rates are mapped to the available 802.11 hardware modulation rates. In the rest of the paper, we assume that the hardware uses 802.11a and that $T_s = 16\mu$s, corresponding to 4 802.11a Orthogonal Frequency Division Multiplexing (OFDM) symbols [25]. With these settings, a slot can carry 12, 24, 36, 48, 64, 96, or 108 bytes, corresponding to the 802.11a's 6Mbps, 12Mbps, 18Mbps, 24Mbps, 36Mbps, 48Mbps and 54Mbps modulation rates, respectively. Transmissions are padded, if necessary, to make them an integer number of TDMA slots.

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 maximum packet length is 2048 bytes, corresponding to the maximum allowed by 802.11. The link number and the node number uniquely identify a directional link, originating at the sender. Soft-TDMAC is a connection oriented link layer protocol, so nodes discard all packets not addressed for one of their incoming links. 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 contains the information about the next data sub-header, which is packed in the same transmission. So, one data transmission may carry a number of smaller IP packets, which increases the protocol efficiency.

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.

A. Network Beacons

Each Soft-TDMAC wireless node periodically transmits 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, 16μs, TDMA slots for each beacon transmission. So, 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 is silent for all nodes and is used to run the clock synchronization algorithm. The other $\texttt{CTRL\_LEN} - 1$ control TxOps are active and are used to send network beacons according to a fixed schedule. Each of the n nodes in the network, i transmits its network beacon every CTRL_REUSE-th TxOp, k, for which

$$k \pmod{\texttt{CTRL\_REUSE}} = i, \tag{1}$$

where k is the number of the active slot since the network boot-up and $\texttt{CTRL\_REUSE} \geq n$, is a configurable control sub-frame TxOp re-use factor. For example, if $\texttt{CTRL\_LEN} = 4$ and $\texttt{CTRL\_REUSE} = 8$ nodes in the network, node $i = 4$ transmits in control TxOp $k = 4$, corresponding to the 3rd control TxOp in frame 2, adjusted for the empty TxOp (Fig. 1), then again in control TxOp $k = 12$, corresponding to the 2nd control TxOp in frame 5, and so on.

Each beacon contains a list of the sender's neighbours. For each neighbour, the list contains its synchronization error and hop count to the sender. Since each 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 all neighbours are advertised. We note that "neighbour" refers to all of the nodes that the sender is aware of, including the nodes that the sender cannot hear from directly. For indirect neighbours, the synchronization information is considered invalid and the number of hops is used for layer-2 routing.

Before sending each network beacon, Soft-TDMAC timestamps it with the current TDMA frame and control sub-frame TxOp number. The timestamp is used by the synchronization algorithm.

B. Synchronization Algorithm

Highly precise network synchronization is essential for efficient TDMA. Soft-TDMAC synchronizes the network by building a network wide synchronization tree. The synchronization tree is determined independently of the routes found by the Soft-TDMAC routing algorithm. In the network synchronization tree, each node synchronizes to its parent, defining a master/slave synchronization relationship for all pairs of nodes in the synchronization tree. Each node determines its master using the shortest synchronization path to the base station. Optionally, node's master is specified manually.

Pairs of nodes achieve mutual synchronization by exchanging clock offsets in the network beacon messages, similar to the Network time Protocol (NTP) [23], [24]. In one exchange, the first node sends a network beacon with a timestamp $T_1 = C_A(t_1)$, which is its value of the network time at time t_1 (Fig. 2). After receiving the time stamp at time t_2, the second node uses its timestamp, $T_2 = C_B(t_2)$, to find the time difference between the clocks relative to itself $\Delta_{BA} = T_2 - T_1$. The second node records the time difference Δ_{BA} as a part of the neighbour's information. After some time, the second node sends a network beacon at time t_3, with the timestamp $T_3 = C_B(t_3)$ and the previously found time difference Δ_{BA}. After receiving the beacon, at time T_4, the first node calculates its relative time difference $\Delta_{AB} = T_4 - T_3$, where $T_4 = C_B(t_4)$. The exchange finishes when the first node sends a beacon that includes Δ_{AB}.

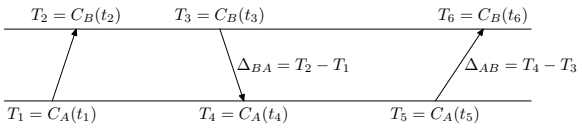


Fig. 2. Pairwise Synchronization

At the end of the exchange, both nodes have each others relative time difference and calculate the pairwise clock offset

$$\theta_{AB} = \frac{1}{2}\left[\Delta_{BA} - \Delta_{AB}\right] \tag{2}$$

and the pairwise round-trip delay

$$d_{AB} = \Delta_{AB} + \Delta_{BA}. \tag{3}$$

Nodes collect the clock offsets to their neighbours between the runs of the synchronization algorithm. The synchronization algorithm first filters the raw clock offsets. The filter detects the values of clock offsets that are out of the valid range and removes them. We describe why some of the values are out of range and how they are detected in the next section. After filtering, the synchronization algorithm averages the raw clock offsets from each neighbour. The final step in the algorithm is to use the averaged clock offset of the node's master to re-synchronize the node's clock. If the node does not have a valid offset to its master, e.g. due to lost network beacons, it synchronizes to the neighbour, which is closest to the base station in the synchronization tree.

The period of the synchronization algorithm depends on the clock drift between the master and the slave. The maximum allowed re-synchronization period is 5s and the minimum period is the minimum time required to receive a network beacon from each neighbour, which depends on the frame duration and the control sub-frame re-use factor, `CTRL_REUSE`. If the synchronization algorithm finds that the clock offset is smaller than half the slot size $T_s/2 = 8\mu$s, the synchronization period is increased by one frame. On the other hand, if the algorithm finds that the clock drift is larger than 8μs, it halves the re-synchronization period. In our experiments, we found that this algorithm quickly converges to a fairly stable re-synchronization period.

C. Network Entry

Since Soft-TDMAC is a collision-free TDMA protocol, before transmitting any packets in the network a node has to be synchronized. Synchronization to the network is achieved in two stages. Initially, a node is unsynchronized. An unsynchronized node collects network beacons for the maximum time of the network synchronization algorithm. During this time, the node estimates the clock offset to the synchronized nodes, by substracting a fixed one-way packet propagation delay. After the first run of the synchronization algorithm, the node is said to be "roughly" synchronized.

Soft-TDMAC assumes a one-way propagation delay of 15 slots (240μs). We show later that this delay is closely related to the network beacon transmission time, which we found empirically. This propagation delay may be off from its true value, however channel accesses by network beacons have about 7 slots of slack, so the roughly synchronized node's clock may be off by as much $7 * T_s = 112\mu$s before full synchronization.

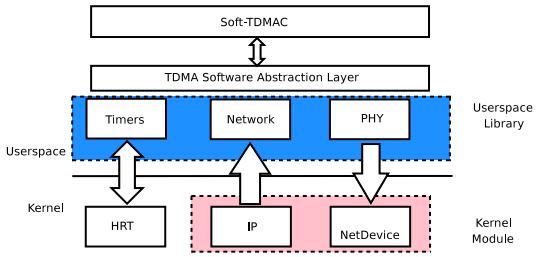


Fig. 3. Soft-TDMAC Architecture

In the second stage, after the node is roughly synchronized, is starts to transmit network beacons and to exchange timestamps for precise synchronization. After 20 runs of the synchronization algorithm, the node declares itself "fully synchronized" and starts sending packets.

We note that a more sophisticated way for the node to transition to the fully synchronized status is to wait until its clock is close to the master's clock. However we found that waiting for 20 runs of the synchronization algorithm is sufficient to synchronize the node to its master to within one TDMA slot. Waiting for 20 runs of the synchronization algorithm limits the total duration of network entry to at most 105s (5s to achieve rough synchronization and at most $20 * 5$s to achieve full synchronization). In practice, the re-synchronization period is always smaller than the maximum 5s – network entry takes $20 - 30$s, depending on the frame duration and the re-use factor in the control sub-frame.

D. Layer-2 Routing

On each node, Soft-TDMAC builds the routing table for the entire network from the network beacon packets. Each network beacon packet contains the information about the number of hops to nodes known by the sender. This information is sufficient to run a distance vector routing in Soft-TDMAC. On each node, Soft-TDMAC maintains a routing table for the nodes it is aware of and it forwards packets based on the routing table.

Including routing in the Soft-TDMAC protocol simplifies its overall implementation. Soft-TDMAC is a connection oriented TDMA MAC protocol that uses pairwise link layer connections to communicate between nodes. Since the Linux networking stack was not designed for a purely connection oriented MAC protocol, it is actually simpler to add routing and forwarding to Soft-TDMAC than it is to change the Linux network stack, or to implement Linux routing table updates in Soft-TDMAC.

IV. Soft-TDMAC's Linux Implementation

Soft-TDMAC's Linux implementation has three major components (Fig. 3). The first component is the kernel module,

which resides in the Linux kernel space and connects Soft-TDMAC with the Linux kernel network services. The second component is the userspace library, which implements a hardware abstraction layer. The hardware abstraction layer hides the implementation details of system timers, Linux network services and 802.11 driver implementation. The third component is the implementaion of the Soft-TDMAC protocol.

When the kernel module receives an IP packet from the the Linux kernel IP networking stack, it forwards it to the userspace library. The userspace library hands the packet to Soft-TDMAC, which process it, enqueues it and requests a timer interrupt from the userspace library for when the packet should be transmitted. After some time, the userspace library creates a timer interrupt and delivers it to Soft-TDMAC, which then forwards the packet to the userspace library as a transmission request. The userspace library hands-off the packet to the kernel module, which uses the 802.11 hardware driver to broadcast the packet.

This architecture has two major benefits. First, since the majority of the software is in userspace, it avoids the difficulties of programming in the Linux kernel, due to the lack of proper debugging tools and the standard C-library system interface for dynamic memory allocation and file access. Due to its location and programming accessibility, Soft-TDMAC presents a more accessible jumping-off point for further TDMA based MAC protocol development, than the software platforms implemented inside the Linux kernel [8]–[11]. Second, by putting the most important components of our software into userspace, we avoid tying our implementation to the restrictive GNU Public License [26].

A. Hardware and System Software Setup

We have tested Soft-TDMAC on Hewlett-Packard nc6000 laptops, which use the Pentium M processors running at around 1.5GHz. The laptops come pre-installed with the wireless cards using the Atheros Communications AR5212 chipset. We use the MadWiFi driver [18], through the Linux networking sub-system. We add profiling software to the driver, which allows us to measure 802.11 transmission times and we add code to change per-packet 802.11 modulation rates. As we show next, Soft-TDMAC can also work with other wireless card, which have 802.11 QoS extensions [19], however the Atheros wireless cards are the only ones we have available.

The laptops run Linux kernel 2.6.23 [27] with the real-time extensions [28]. The Linux real-time extension streamlines the kernel to remove unnecessary software locks and provides 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.

B. The Envelope on Packet Transmission Times

Soft-TDMAC achieves conflict-free TDMA by ensuring all transmissions conform to a given conflict-free link schedule. Before transmitting a packet on a link, Soft-TDMAC calculates the packet's maximum transmission time and ensures that its transmission is over during the link's allocated slots. This check, in addition to tight network wide synchronization, ensures continuous, conflict-free, operation of Soft-TDMAC.

The maximum transmission time for an l byte packet, transmitted at modulation rate m, is bounded by the envelope $\varepsilon_m(l)$, which is measured in TDMA slots. Soft-TDMAC views wireless transmissions as consisting of the active part of the transmission, which is directly related to the packet size, and the guard time, which ensures that the envelope covers software and hardware delays. The envelope on transmission times is calculated with

$$\varepsilon_m(l) = T_g + \left\lceil \frac{l + h_{802.11}}{b_m} \right\rceil * T_s \tag{4}$$

where T_g is the number of "guard slots", reserved for the transmission, $\lceil \cdot \rceil$ is the ceil function, l is the packet size in bytes, $h_{802.11}$ is number of bytes of 802.11 headers, $b_m = \{12, 24, \ldots 108\}$ is the number of bytes carried in a TDMA slot and T_s is the slot size.

The guard time, T_g, ensures that the envelope covers software and hardware delays. Packet transmission time is

$$t_{\text{send}} = t_{\text{proc}} + t_{\text{trans}}, \tag{5}$$

where t_{proc} is software delay, which occurs from the time of the requested timeout, to the time a packet is handed-off to the wireless card and t_{trans} is the time from when the wireless card receives the packet, to the time the card finishes transmitting the packet, which includes hardware delays.

The processing time, t_{proc}, depends on the processor speed and the time taken by the operating system to deliver a timer interrupt. For 802.11 broadcast packets, t_{trans} is

$$t_{\text{trans}} = t_{\text{cont.}} + t_{\text{AIFS}} + t_{\text{tx}}, \tag{6}$$

where $t_{\text{cont.}}$ is the random time the node spends backing-off, if the channel is busy, t_{AIFS} is the pause before the transmission, after determining the channel is free and t_{tx} is the time required to transmit a packet [25]. We note that since Soft-TDMAC ensures that the hardware interface is only handling one packet at a time, t_{trans} has no queueing delay component.

The guard time should also be longer then the maximum synchronization error in the network. For conflict-free transmissions, it is sufficient that the guard time T_g satisfies

$$T_G > \max\left\{t_{\text{cont.}} + t_{\text{AIFS}} + t_{\text{proc}}, \sigma_{\text{sync}}\right\}, \tag{7}$$

where σ_{sync} is the maximum synchronization error.

For a tight envelope it is essential to decrease the guard time. We set set $t_{\text{AIFS}} = 0$, and the limits on random back-off to `CWmin` $= 0$, `CWmax` $= 0$, to eliminate $t_{\text{cont.}}$ and t_{AIFS}. These parameters can be modified with the newer Linux wireless drivers such as the MadWiFi driver [18] for the Atheros wireless chipsets and IPW2200 driver for the Intel wireless chipsets [19]. We decrease t_{proc} with careful code design. As we show later in the paper, the maximum synchronization error, σ_{sync}, is much smaller than the other delays, so we do

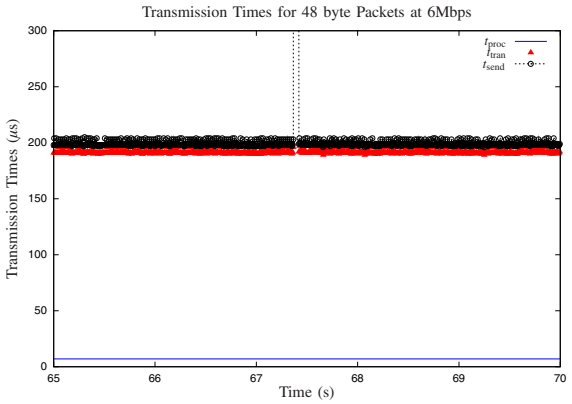


Fig. 4. Transmission Delay

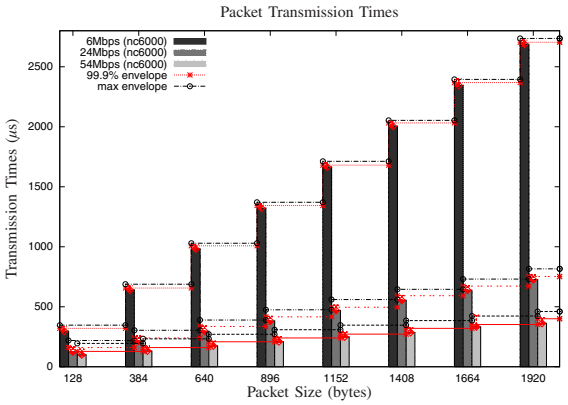


Fig. 5. Fitting the Envelope

not discuss it further in this section.

C. Empirical Tuning of TDMA Guard Times

We derive the envelope on the transmission times from experimental data. We perform an experiment where a node transmits 20,000 packets at 5ms intervals. We use the 802.11a wireless channel 100, operating at 5.5GHz, which we found to be free of other wireless nodes. We repeat the experiment for different packet lengths and modulation rates. At the lowest 802.11a modulation rate of 6Mbps, the longest packet transmission of 1920 bytes takes less than 3.0ms, so all transmissions end before the next transmission begins. We modify the MadWiFi driver to record the hand-off of a userspace packet and the time the card has finished sending the packet.

Fig. 4 shows the packet processing time, t_{proc}, the packet transmission time, t_{tran}, and the total time to send the packet, t_{send}, for the 5 seconds of the run for 48 byte packets at 6Mbps modulation, corresponding to network beacons. All time differences are calculated from the time that the timer was supposed to occur. We note that t_{proc} is almost constant with a negligible amount of variability (less than a few microseconds). Second, enabling the QoS features of the MadWiFi driver also ensures that t_{tran} is almost constant.

We note that sometime around 27s into this experiment the card experienced what we call a "hiccup" – it inexplicably delayed the packet for a long time ($>$ 1ms). 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 function that delays the packets. We 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. Nevertheless, they force us to add a filtering mechanism to the synchronization algorithm to remove all observed clock-offsets, corresponding to round-trip delay times with negative times or times greater than 800μs, which is roughly twice the round-trip delay for network beacons.

Using the measured times for the 48 byte network beacons transmitted at the lowest rate, we find the envelope, $\varepsilon_m(l)$, for $l = 48$ and $m =$ 6Mbps. The MadWiFi driver adds $h_{802.11} = 36$ bytes of overheads to each broadcast packet. So, the wireless card actually broadcasts 84 bytes of data, which at 6Mbps takes 7 TDMA slots. On average, $t_{\text{trans}} = 192\mu$s with the standard deviation of 2μs, so the Atheros hardware adds around 80us of delay to each transmission. We believe 40μs correspond to the 10 802.11a OFDM symbols of preamble specified by the standard [25] and the other 40μs correspond to the Atheros post-transmission back-off [11]. On average, $t_{\text{proc}} = 7$ with the standard deviation of 0.2μs.

Using these values, we get the average sending time for a 48 byte packet

$$t_{\text{send}} = t_{\text{proc}} + t_{\text{trans}} = 7\text{us} + 80\text{us} + 7 * 16\text{us} < 13 * 16\text{us},$$

Generalizing for all packet sizes,

$$\varepsilon_m(l) = 6 * T_s + \left\lceil \frac{l+36}{b_m} \right\rceil * T_s. \tag{8}$$

We test the envelope against a subset of measured transmission times (Fig. 5). Results are similar for other measurements. The bars show the transmission times for 99.9% of the packets of the given size and the error lines show the maximum and minimum values, excluding the hiccups. The envelope, (8), is shown as the envelope "99.9% envelope", since due to due to small variations in processing times, the envelope only covers 99.9% of the packets.

We also obtain an envelope that covers the maximum time for each transmission, excluding the hiccups

$$\hat{\varepsilon}_m(l) = 11 * T_s + \left\lceil \frac{l}{b_m} \right\rceil * T_s, \tag{9}$$

where the MadWiFi overheads are now absorbed by T_g This envelope is shown as "max envelope" in Fig. 5a.

In order to make the Soft-TDMAC protocol robust, we make the network beacon transmissions very robust. We use the the maximum envelope, (9), to find that each network beacon transmission takes 15 TDMA slots. In addition, we also add 5

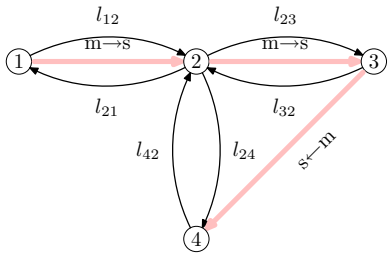


Fig. 6. Logical Topology of the Testbed

extra guard symbols. We note that at least 7 of these slots are not involved in the transmission, so network beacons tolerate synchronization errors of 112μs.

Since the 99.9% envelope, (8), is more efficient than the maximum envelope, we use it for data transmissions. Efficiency is defined as the ratio of the number of slots required by the envelope to the number of slots required to transmit the packet without any overhead. For example, for transmissions at 54Mbps, the 99.9% envelope is 52% more efficient than the maximum envelope for packet size of 128 bytes and 15% more efficient for the maximum packet size of 1920. In addition to using the 99.9% envelope for packet transmissions, Soft-TDMAC further increases efficiency by packing small IP packets into larger Soft-TDMAC packets.

V. Testbed Results

We summarize hundreds of hours of evaluating Soft-TDMAC on our 4 node testbed (Fig. 6). All nodes can hear and accept network beacons from all other nodes. 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 4 ignores all data packets except the ones arriving from node 2, (link l_{24}); if node 4 sends an IP packet to node 1, that packet must traverse node 2. In all experiments, node 1 is the base station.

The software and hardware setup are the same as in Sec. IV-A. The testbed runs on the 802.11a wireless channel 100, operating at 5.5GHz.

A. Evaluation of Soft-TDMAC Synchronization

We setup an experiment on the testbed to measure the clock offset between the nodes in the network. In this experiment, the frame size is 20ms, 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 minimum re-synchronization period is 160ms. We also tried Soft-TDMAC with other control sub-frame sizes and maximum number of nodes, with similar results.

Since all nodes are in the range of each other, using the minimum synchronization tree, each node would normally select node 1 as its clock master. However, we manually set the synchronization tree to $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ (thick arrows in Fig. 6), to test clock synchronization over multiple hops. We run the network continuously for 5 hours. The synchronization algorithm ran on average every 1617ms on the first hop, 1096ms on the second hop and 323ms on the third hop.

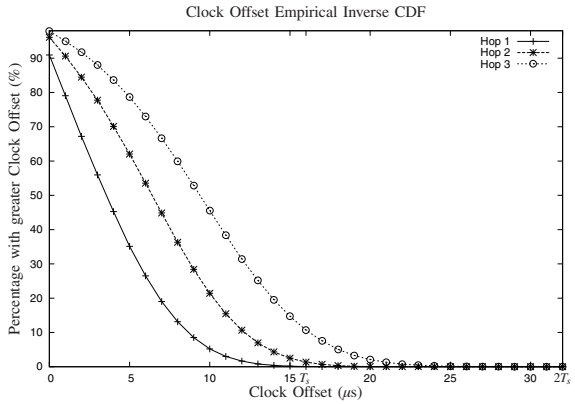


Fig. 7. Multi-hop Synchronization

TABLE I
Absolute Clock Offset Statistics

Statistic	Hop 1	Hop 2	Hop 3
Mean	4.5 μs	7.0 μs	10.0 μs
Std. Dev.	3.3 μs	4.2 μs	5.2 μs
99.9 prctile.	16 μs	20 μs	26 μs
99 prctile.	13 μs	17 μs	22 μs
Maximum	58 μs	84 μs	56 μs

Fig. 7 shows the inverse empirical Cumulative Density Function (CDF) of the absolute value of the filtered clock offsets, measure at node 1 to node 2 ("Hop 1"), node 3 ("Hop 2") and node 4 ("Hop 3"). There were well over a 100,000 clock offsets collected by node 1 to each of the other nodes during this period. We note that 10% of clock offsets on the first hop were 0μs. We also note that most clock offsets at any hop are less than 2 TDMA slots.

Table I shows detailed statistics of the experiment. The mean absolute clock offset for all three hops is less than 10μs. We note that less than 0.1% clock offsets for the first hop are more than 16μs and that at three hops, less than 0.1% of clock offsets are greater than 26μs. The worst case clock offset of 84μs 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.

B. Single-hop TCP Performance

We test Soft-TDMAC single-hop TCP performance by turning off nodes 3 and 4 and only using nodes 1 and 2. We try frame sizes $T_f = 20$ms and $T_f = 50$ms. For both frame durations, we allocate 10% of the frame to the control sub-frame, 20% of the frame to the uplink and 70% of the frame to the downlink and use these to calculate the slot allocations. For the frame duration $T_f = 20$ms, we allow for 4 control

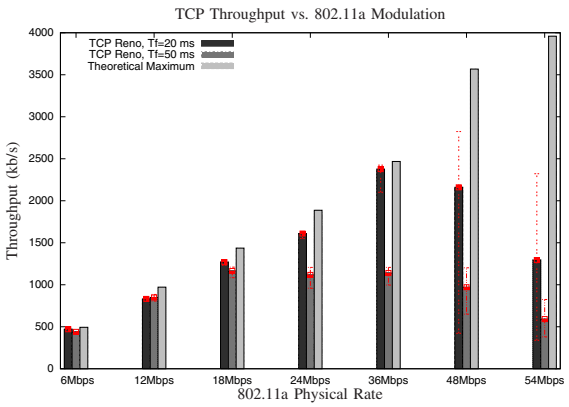
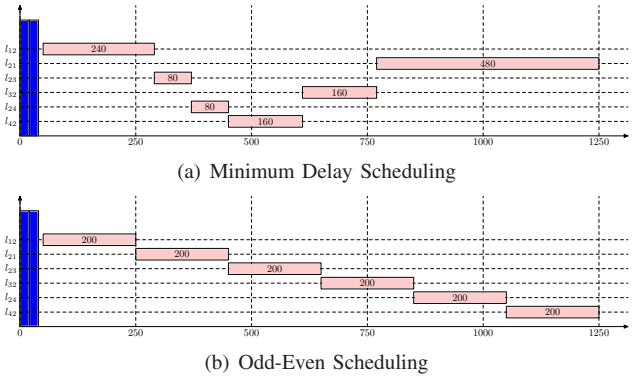


Fig. 8. One-hop TCP Throughput



(a) Minimum Delay Scheduling

(b) Odd-Even Scheduling

Fig. 9. Multi-hop TDMA Schedules

TABLE II
MEASURED ROUND-TRIP DELAY (6M)

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

TxOps, while for $T_f = 50$ms, we allow 15 control sub-frame TxOps. For the purposes of the synchronization algorithm we use control sub-frame re-use factor `CTRL_REUSE` $= 32$.

In order to get statistically valid performance results, we run a series of experiments to test the performance of pairwise synchronization and TCP throughput for two TCP variants. In each experiment, we first start node 1 and then node 2. We let node 1 synchronize and after 30 seconds it initiates a download of 30Mb of data from node 2. We repeat this setup 30 times for each of the 7 available 802.11a physical modulation rates, TCP Reno and TCP Westwood [29], for a total of 420 experiments.

Analysing the data, we discovered a strong corelation 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. Soft-TDMAC does not implement an Automatic Repeat Request (ARQ) mechanism, which would eliminate frame losses at the MAC layer.

Fig. 8 show 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 scenario with the best and the scenario with the worst throughput. The figure also compares the throughput to the theoretical maximum ("Theoretical Maximum"), which we obtain by finding how much throughput the link would have if it transmitted continuously at its available 70% of the given modulation rate, without any overheads. 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 54Mbps). Due to space restrictions, we do not show TCP Westwood performance – it performs slightly worse than TCP Reno, at higher modulations.

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

C. Multi-hop TCP Performance

We evaluate Soft-TDMAC multi-hop performance using our testbed (Fig. 6), with the same TDMA parameters used in Sec. V-A.

We evaluate Soft-TDMAC performance with two types TDMA multi-hop schedules. The first type of schedule, is a minimum TDMA delay schedule [1], [2] (Fig. 9a). Slot allocations to links are shown in the schedule. This schedule minimizes TDMA scheduling delay for all nodes in the network. TDMA scheduling delay occurs if, on the same path, an outbound link on a router is scheduled to transmit before an inbound link on that router. For the path connecting node 1 to node 2, this schedule orders the links $l_{12} \rightsquigarrow l_{23} \rightsquigarrow l_{32} \rightsquigarrow l_{21}$. The schedule allocates twice as much bandwidth on the uplink (traffic to the base station), as it does on the downlink. Allocating higher bandwidth on the uplink simplifies the running of experiments.

The second type of schedule is an "odd-even" TDMA schedule [14] (Fig. 9b), which schedules pairs of nodes alternatively. This type of scheduling is called odd-even scheduling because links are either scheduled in even (l_{21}, l_{32}, l_{24}) or odd slots (l_{12}, l_{23}, l_{24}). We note that despite the fact that 2P [12] 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 2P and slot allocations are generally longer than in Fig. 9b. Consistent with odd-even scheduling [12], [14], all links are allocated the same number of slots.

To measure round-trip delay, we send 1000 ICMP packets from node 2,3, and 4 to the base station. The modulation on each link is fixed at 6Mbps. We use the Linux version of `ping` to find the maximum ICMP flood. Table II shows the mean

TABLE III
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

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 3 and 4 are delayed until the next frame at node 2. The minimum delay schedule has round-trip times, which are always around one frame. We tried this experiment at all available rates, with similar results.

The minimum delay schedule adjusts link bandwidths to take into account that links l_{12} and l_{21} carry traffic of 3 nodes. Since all links have the same time allocation in the odd-even schedule, links l_{12} and l_{21} present a bottleneck for nodes 3 and 4, which have excess bandwidth available on their links to node 2. To compare the two schedules in terms of throughput, we fix the rate on all links in the minimum delay schedule at 6Mbps, while setting the rate on links l_{12} and l_{21} to 18Mbps in the odd-even schedule.

For each schedule, we run an experiment where nodes 2, 3 and 4 start a 3Mb upload to the base station, at roughly the same time, with node 2 starting first. We repeat each experiment 30 times. Table III show the average TCP throughput, over the 30 runs for each node. In all cases, standard deviation was less than 5kb/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 2 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 1 and 2 to 18Mbps. Since node 2 now has extra bandwidth allocated to it, its bandwidth increases substantially ("Min-delay (18/6/6)"). The the bandwidth of the other two nodes stays about the same since their bottleneck links are on the first hop.

VI. CONCLUSION

We have presented Soft-TDMAC a software TDMA based MAC protocol running over commodity 802.11 hardware. Soft-TDMAC implements a pairwise synchronization algorithm, which synchronizes nodes to within a few microsecond sized TDMA slots. The precise synchronization allows us to decrease the overhead of transmissions for a very efficient TDMA protocol, using small frame sizes. We test the protocol on our test-bed and show that when channel conditions are good, TCP can achieve almost full channel bandwidth. We also show that Soft-TDMAC can use schedules that adjust link allocations based on end-to-end demands and that decrease end-to-end delay.

REFERENCES

[1] P. Djukic and S. Valaee, "Link scheduling for minimum delay in spatial re-use TDMA," in *Proceedings of INFOCOM*, 2007.

[2] ——, "Delay aware link scheduling for multi-hop TDMA wireless networks," *IEEE/ACM Trans. Netw.*, to Appear, October 2009.

[3] 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.

[4] "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.

[5] "IEEE standard for local and metropolitan area networks part 16: Air interface for fixed broadband wireless access systems," 2004.

[6] "IEEE Draft Standard P802.16j/D5, part 16: Air interface for fixed and mobile broadband wireless access systems - multihop relay specification," May 2008.

[7] A. Rao and I. Stoica, "An overlay MAC layer for 802.11 networks," in *MobiSys*, 2005, pp. 135–148.

[8] M. Neufeld, J. Fifield, C. Doerr, A. Sheth, and D. Grunwald, "SoftMAC—flexible wireless research platform," in *HotNets*, 2005.

[9] C. Doerr, M. Neufeld, J. Fifield, T. Weingart, D. C. Sicker, and D. Grunwald, "MultiMAC - an adaptive MAC framework for dynamic radio networking," in *IEEE DySPAN*, 2005.

[10] A. Sharma, M. Tiwari, and H. Zheng, "MadMAC: Building a reconfigurable radio testbed using commodity 802.11 hardware," in *IEEE SECON SDR*, 2006.

[11] A. Sharma and E. M. Belding, "FreeMAC: Framework for multi-channel MAC development on 802.11 hardware," in *ACM SIGCOMM PRESTO*, 2008.

[12] B. Raman and K. Chebrolu, "Design and evaluation of a new MAC protocol for long-distance 802.11 mesh networks," in *Mobicom*, 2005.

[13] 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 *USENIX NSDI*, 2007.

[14] G. Narlikar, G. Wilfong, and L. Zhang, "Designing multihop wireless backhaul networks with delay guarantees," in *INFOCOM*, 2006.

[15] http://www.isi.edu/nsnam/ns/.

[16] 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.

[17] http://spirit.cs.ucdavis.edu/SoftTDMAC/.

[18] http://madwifi.org/.

[19] "Intel® PRO/Wireless 2200BG driver for linux," http://ipw2200.sourceforge.net/.

[20] J. Elson and D. Estrin, "Time synchronization for wireless sensor networks," in *IPDPS*, 2001.

[21] M. L. Sichitiu and C. Veerarittiphan, "Simple, accurate time synchronization for wireless sensor networks," in *WCNC*, 2003, pp. 1266–1273.

[22] K. Römer, "Time synchronization in ad hoc networks," in *MobiHoc*, 2001, pp. 173–181.

[23] S. Ganeriwal, R. Kumar, and M. B. Srivastava, "Timing-sync protocol for sensor networks," in *SenSys*, 2003.

[24] D. Mills, "Network Time Protocol (NTP)," RFC 958, Sep. 1985, obsoleted by RFCs 1059, 1119, 1305. [Online]. Available: http://www.ietf.org/rfc/rfc958.txt

[25] "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.

[26] "GNU general public license," http://www.gnu.org/licenses/gpl.html.

[27] "Linux kernel," http://www.kernel.org/, 2006.

[28] I. Molnar, "Real-time patches for Linux 2.6 kernel," http://www.kernel.org/pub/linux/kernel/projects/rt/.

[29] 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.