TCP Performance Optimizations for Wireless Sensor Networks

Philipp Hurni, Ulrich Bürgi, Markus Anwander, Torsten Braun

Institute of Computer Science and Applied Mathematics (IAM), University of Bern
hurni, buergi, anwander, braun@iam.unibe.ch

Abstract. Since the appearance of downsized and simplified TCP/IP stacks, single nodes from Wireless Sensor Networks (WSNs) have become directly accessible from the Internet with commonly used networking tools and applications (e.g., Telnet or SMTP). However, TCP has been shown to perform poorly in wireless networks, especially across multiple wireless hops. This paper examines TCP performance optimizations based on distributed caching and local retransmission strategies of intermediate nodes in a TCP connection, and proposes extended techniques to these strategies. The paper studies the impact of different radio duty-cycling MAC protocols on the end-to-end TCP performance when using the proposed TCP optimization strategies in an extensive experimental evaluation on a real-world sensor network testbed.

1 Introduction

The performance of TCP across multiple wireless hops has yet been intensively studied in the context of IEEE 802.11-based (mobile) ad hoc networks [1][2][3], long before sensor networks emerged. The discovered performance degradation is directly associated with the unreliable nature of the wireless channel (higher bit error rates and packet loss), particular properties of and interactions with the underlying wireless channel MAC protocols (exponential backoff mechanisms, hidden node and exposed node problem), and the design of the TCP congestion control, which erroneously interprets packet loss as an indication for congestion.

So far, the issue of optimizing TCP in the context of WSNs has not yet attracted too much attention. The most well-known studies in this field are DTC [4] and TSS [5], which propose mechanisms based on packet inspection in intermediate nodes and distributed TCP segment caching [4] and/or TCP ACK regeneration [5]. Both studies base on evaluations using network simulations with rather simple channel assumptions, and on energy-unconstrained CSMA/CA on the MAC layer, in order to overhear neighboring node's transmissions. The implications of radio duty-cycled MAC protocols on TCP performance, however, have generally not yet been addressed. In this paper, we examine the impact of different MAC protocols on the aforementioned distributed caching techniques of DTC [4] and TSS [5]. We then propose and evaluate a series of extensions. Section 2 discusses the basic concepts of DTC and TSS, which were specifically designed for WSNs. In Section 3, we present various suggestions for optimizations of TCP operating across multiple WSN hops, which we then examine in Section 4 in a series of real-world experiments. Section 5 concludes the paper.

2 Related Work

Distributed TCP Caching for WSNs (DTC) [4] proposes to locally cache TCP segments on sensor nodes in between two TCP endpoints in order to perform *local* retransmissions by the intermediate nodes to recover from random packet loss. DTC can be seen as a generalization of Snoop [3]. Each intermediate node maintains a cache for buffering TCP segments, and continuously updates a round-trip-time (RTT) estimation for each TCP connection. If it takes considerably more time for a TCP ACK to arrive, intermediate nodes consider it to be lost, regenerate the TCP segment and forward it again. DTC further relies on implicit acknowledgements by continuously overhearing neighboring nodes' transmissions, and hence relies on having energy-unconstrained CSMA on the MAC layer. The simulation-based evaluation using in [4] claims a significantly reduced amount of transmissions and an overall throughput increase by 450% in a 6 hop scenario with a 10% overall packet loss rate.

TSS extends DTC [4] by local regeneration of TCP ACKs: an incoming TCP segment that has already been acknowledged is not forwarded, but answered with a locally generated TCP ACK with the highest TCP ACK number ever received. An aggressive TCP acknowledgement recovery mechanism is introduced: each node monitors if the TCP ACKs it has sent to the next node in line are effectively forwarded by the latter. If such a forwarding transmission is discovered to be missing, the TCP ACK is sent out again. The simulation-based evaluation of TSS has shown to reduce the total amount of frame transmissions required to forward 500 TCP segments over a chain of 10 nodes, a reduction of over 70%. Same as DTC, TSS relies on the assumption of an energy-unconstrained CSMA/CA MAC layer for continuous overhearing of the neighboring nodes' transmissions.

3 TCP Performance Optimizations

The starting point of our investigations is formed by the TCP segment caching strategy proposed in DTC [4] and TSS [5], which perform local retransmissions of TCP segments and TCP ACKs. Based on ideas of these two studies, we iteratively designed, tested, and combined a number of TCP performance optimizations, which we discuss in the following. Our optimizations are to a large extent independent from each other, consist in either new functionalities or modification of previous suggestions. We implemented all our TCP optimizations in in a modular manner in our so-called *Caching and Congestion Control (cctrl)* module. The module can be integrated into the µIP stack [6] of the Contiki OS [7], which together with the TelosB [8] nodes serves as our main platform for implementation and evaluation. The transparency to the underlying network stack was a major design goal of the *cctrl* module, since it allows to use all

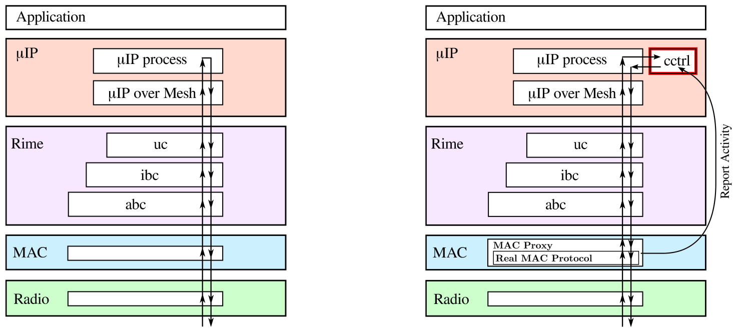

Fig. 1. Contiki Network Stack, *unmodified* (left) and with *cctrl* Module (right)

its implemented features on top of every MAC protocol layer of Contiki. The only prerequisite of the *cctrl* module to effectively interact with the flow of TCP packets is to have *symmetric routes*, such that TCP ACKs are sent across the same paths as the corresponding TCP data segments.

3.1 The Caching and Congestion Control (cctrl) Module

In order to locally cache TCP segments in intermediate nodes, the *cctrl* module be aware of all TCP packets that are forwarded by the node. Figure 1 depicts a packet traversing the different layers of the Contiki networking stack, in case of an *unmodified* Contiki OS network stack (left) and with our *cctrl* module integrated (right). A TCP packet, encapsulated in a Rime [9] packet, is received by the radio and passed to the MAC layer, which copies it to Rime's packet buffer. The node in Figure 1 is an intermediate node. Therefore, the packet is sent out again without passing it to the application layer. The *cctrl* module intercepts the packet flow right before the packet is again passed to the *µIP over mesh* module, which implements the routing functionality. We intercept outbound packets instead of inbound packets, since at this stage packets have already been processed and validated. The *cctrl* module allocates a buffer in which the data is cached. This buffer has to be large enough to hold at least the content of *two* TCP/IP packets per TCP connection, one for each direction. Since at least one host is assumed to be running the µIP stack, which is restrained to only one single unacknowledged segment per TCP connection, the *cctrl* module allocates exactly twice the size of µIP's packet buffer per observed TCP connection, one for each direction.

3.2 Initial Strategy: Segment Caching and Local Retransmissions

In our initial design of the *cctrl* module, we implemented the distributed caching features from DTC [4] and TSS [5] that are *independent* from the underlying MAC protocol. We avoided to rely on the overhearing assumption and to tightly couple our *cctrl* module with specific MAC layer properties, since we intended to examine the former with all available Contiki MAC layers. The TCP segment

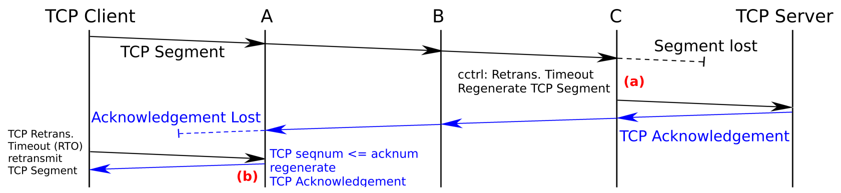

Fig. 2. Local TCP Retransmission (a) and TCP ACK Regeneration (b)

caching and local retransmission strategy of our initial design of the *cctrl* module are illustrated in Figure 2, and will be denoted as *initial cctrl* in the graphs of Section 4.

- In each intermediate node between two TCP endpoints, the *cctrl* module processes each IP packet that is received and forwarded. IP packets without a TCP payload (e.g., UDP packets) are ignored. The *cctrl* module copies the entire content of the µIP buffer to one of its empty buffer slots, and schedules a retransmission timer. This timer is calculated as $t_{rto} = 3 \times RTT_{est}$, with RTT_{est} being the round-trip-time estimation *from* the intermediate node *to* the target node of the segment. RTT_{est} is continuously updated according to the TCP Exponentially Weighted Moving Average (EWMA) filter.
- If the incoming packet is a TCP ACK, only the TCP/IP header is cached, and no retransmission timer is scheduled. Caching is omitted for out-of-sequence packets and retransmissions, such that the connection status always keeps valid sequence and acknowledgement numbers.
- When the retransmission timer of a forwarded TCP segment expires before a TCP acknowledgement has returned, the *cctrl* module assumes that the segment has been lost, releases it from the cache and sends it out again, as displayed with label (a) in Figure 2, with C initiating a local retransmission.
- Each TCP segment is checked according to the state of the TCP connection it belongs to. If the current packet is a TCP ACK of a cached TCP segment, the *cctrl* module checks whether the current segment's acknowledgement number is greater than the cached segment's sequence number, and removes this segment from the cache if this is the case.
- If the packet is a retransmission of a TCP segment, for which a TCP ACK has already been received, it is discarded and a TCP acknowledgement returned towards the packet's sender, as displayed with label (b) in Figure 2, where node A responds to a TCP retransmission with regenerating the previously received TCP acknowledgement. The regeneration of TCP acknowledgements is not linked to a timer, in contrast to TSS [5].

3.3 Channel Activity Monitoring

Numerous WSN studies have proposed to take advantage of the broadcast nature of omnidirectional wireless transmissions, in order to gain additional information about ongoing transmissions in the vicinity. In CODA [10], the channel conditions of the recent past and the amount of buffered packets in the TinyOS internal send queue are used to calculate an indicator for congestion in the network. Most features of CODA, however, rely on the assumption of energy-unconstrained CSMA on the MAC layer, similarly as DTC and TSS. When a radio duty-cycling E^2-MAC protocol is used on the MAC layer, continuous overhearing of every transmission of the neighboring nodes is not possible. Transmissions can only be overheard when the radio is currently turned on, e.g., in a wake-up. However, even the coincidental reception of other nodes' frames can provide valuable information. With preamble-strobing MAC protocols, e.g., X-MAC [11], the overhearing of a strobe targeting another node indicates currently ongoing transmissions in the vicinity. A large amount of overheard packets over the recent past generally indicates a situation in which it would be beneficial to withhold a scheduled transmission to avoid further collisions, independent from the MAC layer. Therefore, we designed a solution that a) remains independent from the MAC protocol, but that b) still allows to feed information about the current channel utilization and channel conditions to the *cctrl* module.

The MAC Protocol Proxy: We implemented a simple hook for the *cctrl* module to the MAC protocol's packet buffer, however, keeping the MAC protocol implementation completely unmodified and replaceable. Our solution consists in a MAC protocol proxy, which implements the MAC layer interface of the Contiki OS, but does not provide any functionality on its own, except for notifying the *cctrl* module upon reception of any packet. The MAC protocol proxy initializes a real MAC protocol (such as X-MAC or LPP), and simply forwards every function call to that of the real protocol module, as depicted in Figure 1. Even in case of an error in a received packet, e.g., a CRC checksum mismatch or a wrong target address, the MAC proxy becomes aware of the overheard but corrupted packet. After each reception, it notifies the *cctrl* module about the overheard packet and stores a timestamp into its newly introduced *activity history*. In this history, the *cctrl* stores the timestamps of the most recent activity notifications received from the MAC proxy. The *cctrl* module continuously calculates the *channel activity level*, which we defined as the amount of overheard packets registered by the MAC proxy that are not older than one average RTT estimation (RTT_{est}) of the observed TCP connection. Relating the activity level calculation to RTT_{est} compensates for the large differences among the various MAC protocols' latencies.

Idle Channel Periods: In a preliminary evaluation, we investigated whether the calculation of the activity levels has any informative value, especially when overhearing is only possible to a minor extent with radio duty-cycling E^2-MAC protocols. We evaluated the obtained activity values in a scenario using X-MAC, where one TCP sender at the beginning of a 7-nodes linear chain sends TCP segments to the receiver at the end of the chain. Five nodes with *cctrl* modules hence forward the TCP data segments and ACKs between these two nodes. Figure 3 depicts the activity levels registered by the *cctrl* modules of the five intermediate nodes versus the experiment time. Each node is represented by one specific color, the reception of data packets at the end node and the corresponding sequence number are displayed along the x-axis. During most of the time, all nodes register rather high activity levels: 6-8 packets are overheard within each node's RTT_{est} on average, since X-MAC samples the channel each 125ms per default, chances are high that some preamble strobes of currently ongoing transmissions are overheard by non-targeted nodes. Between t=100s-130s and t=250s-300s, the flow of TCP segments and ACKs is continuous. However, some TCP segments (e.g., sequence numbers 27, 31, 45, 47 etc.) need significantly longer to be delivered. During these time periods (e.g., t=150-160s), little or no channel activity is registered by all the nodes. We investigated on the problem for these long interruptions in the end-to-end TCP flow, during which precious bandwidth remained unallocated. An in-depth analysis of the trace files revealed that there are two common problems that caused these rather long *idle periods*, cf. Figure 3. The first problem is the loss of a TCP segment on one of the first hops. In such a situation, the packet has not been cached by many intermediate nodes, maybe even by none. All nodes in the chain hence have to wait for the sending host's or one of its neighbors' RTO to occur. A second encountered problem occurs when a TCP ACK is lost close to its final destination. The retransmission of the ACK can then only be triggered by the reception of a retransmitted TCP segment with an already acknowledged sequence number, which has to come from the original TCP sender, because all intermediate nodes have already emptied their cache. The more nodes the ACK has already passed, the longer is the waiting time, and hence, the idle channel period.

Activity Dependent Early Retransmissions: We investigated on a means to exploit the additional channel knowledge gained with the employed MAC

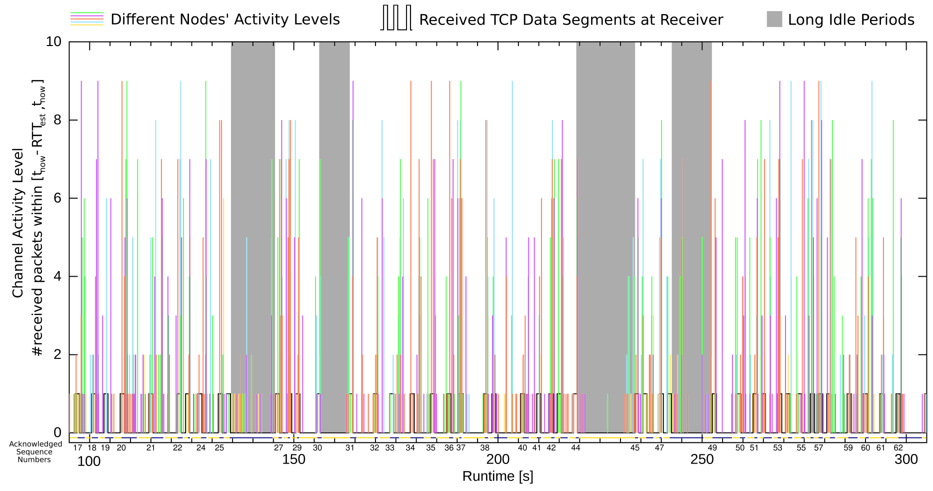

Fig. 3. Different Nodes' Activity Levels with X-MAC vs. Time Dashed Line indicates Traveling Time for Data Segments (*yellow*) and TCP ACKs (*blue*)

protocol proxy and channel activity history, in order to resolve or alleviate the discovered problems of the long idle periods and to further improve the end-to-end TCP throughput. According to our observation that the activity values of most nodes equal to zero in the discovered situations of an idle waiting period, cf. Figure 3, we altered the *cctrl* module's retransmission mechanism to retransmit *earlier* when the channel was found idle for a long time. The retransmission timer of cached TCP segments was hence split into two parts: $t_{rto} = t_{rto1} + t_{rto2}$. The first timer times out at $\frac{2}{3}$ of the usual RTT estimation value RTT_{est}, hence $t_{rto1} = 3 \times RTT_{est} \times \frac{2}{3}$, and the second timer $t_{rto2} = 3 \times RTT_{est} \times \frac{1}{3}$. When the first retransmission timer expires, the *cctrl* module checks its activity history, and initiates an early retransmission if the activity level equals zero. Otherwise the retransmission is deferred again by the second timer t_{rto2}. With this retransmission strategy, we targeted at reducing the occurrence probability and the duration of the discovered long *idle periods*. Since the value of RTT_{est} decreases towards the nodes *closer* to the destination of the TCP data segments, the *channel activity level* value (calculated as the number of packet receptions within $[t_{now} - RTT_{est}, t_{now}]$) decreases as well, and is more likely to equal zero than at the beginning of the chain. Therefore, the outlined strategy triggers the retransmissions of the nodes closer to the destination earlier than those at the beginning of the node chain, which is a desirable property.The absolute value of the *activity level* has no particular deeper meaning. Our outlined retransmission strategy is only triggered when its value is zero, a situation in which it is probable that the channel has not been used and has hence been left idle for $2 \times RTT_{est}$.

3.4 Multiple Connections

Since the Contiki µIP stack only allows to transmit one unacknowledged TCP data segment at any time per TCP connection, precious bandwidth could probably remain unallocated, especially on long routes, where transmissions on one end of the route would not necessarily interfere with transmissions on the far other end. We investigated a means to spatially reuse the wireless channel and to allow the transmission of multiple segments *in flight*. However, sticking to our initially described design decisions, we decided to keep the *cctrl* module modular, MAC-layer-independent and independent from modifications within the established µIP stack.We designed a simple solution that simultaneously establishes multiple TCP connections between TCP client (sender) and TCP server (receiver). This allows an application to send out a new data packet over a second TCP connection, although the previously transmitted packet has not yet been acknowledged, permitting a maximum of two TCP segments or ACKs *in flight*. In the subsequent evaluation, we compare this approach with the mechanism of *activity-dependent early retransmissions* of Section 3.3, as well as both mechanisms combined. The effective implications of this approach were yet unforeseeable at the time of designing it: instead of an improvement of throughput, it could also result in a deterioration, since more TCP segments being transmitted could also lead to congestive situations and an increasing number of collisions.

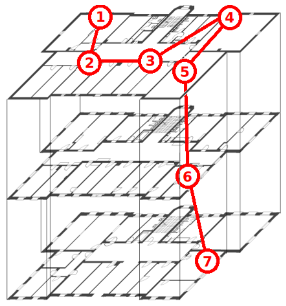

Fig. 4. Seven TelosB [8] Nodes in the Distributed Testbed

4 Experimental Evaluation

4.1 Testbed Platform and Experiment Setup

For all the experiments and evaluations, we used our distributed indoor testbed facilities [12][13]. The topology used for experimentation is depicted in Figure 4. TCP data is transmitted over 2, 3, 4, 5 and 6 hops on the depicted route. In each of these single route experiment configurations, node 1 formed the receiver of the TCP segments, and nodes 3, 4, 5, 6 and 7 were the the senders of the TCP segments. As one can see in Figure 4, nodes are located in different rooms of the building, with the configured route spanning across three floors. The links of the employed routes exhibit a rather high success rate in case of no other ongoing transmissions ($\geq 75\%$). Besides the links displayed in the figure, some close node pairs physically are within each other's transmission range (e.g., nodes 3 and 5). However, for many node pairs, direct communication failed because of the signal attenuation from obstacles (e.g., walls, floors) and/or the distance (e.g., 1, 4 and 7 are definitively out of each other's transmission range, respectively).

By running all experiments with four different wireless channel MAC protocols, we thoroughly investigate the impact of the MAC layer on the end-to-end performance. The examined Contiki OS MAC layers are:

- the *NullMAC* layer, which, combined with the Contiki CSMA layer operates as simple energy-unconstrained CSMA with a random backoff contention,
- the *X-MAC* [11] protocol layer applying asynchronous preamble sampling and preamble strobing,
- the *ContikiMAC* [14] layer, which merges features from a range of asynchronous preamble-sampling E^2-MAC protocols, and which has become the default MAC layer of the Contiki OS v.2.5, and
- the receiver-initiated *Low Power Probing (LPP)* protocol layer, with which nodes periodically send out beacons to indicate reception readiness.

We chose every experiment run to last 10 minutes, during which the TCP sender on one end of the route sent as many segments as possible to the TCP receiver on the other end. All TCP data packets contained a 16 byte character string as payload. Including the TCP/IP and Rime headers, the radio had to transmit 79 bytes per data frame. A TCP ACK, as transmitted by the TCP receiver in response to a data packet, contains only 63 bytes in total. All experiments were run *with* and *without* our proposed *cctrl* module and the extensions proposed in Sections 3.3 and 3.4. The results of the experiment runs obtained without the *cctrl* module, hence, with the same application sending TCP packets but an unmodified Contiki networking stack, are referred-to as *unmodified* in the subsequent figures. In order to reduce the impact of environmental impacts on the experiment results, all experiments were run over night or during weekends, when fewer people were expected to be present in the building. Thus, the channel conditions were comparable to a large extent. Each configuration was repeated 15 times, in order to obtain a data set from which stable statistical measures could be calculated (i.e., mean and standard deviation).

4.2 Initial Strategy: Segment Caching and Local Retransmissions

Figure 6 depicts the number of successfully transmitted TCP segments of the four examined MAC layers dependent on the number of hops. The graphs labeled *initial cctrl* refer to our initial design of the *cctrl* module discussed in Section 3.2, which implements the basic features of DTC/TSS, but without the extensions of Sections 3.3 and 3.4 (*channel activity monitoring* and *multiple connections*). The figure conveys that NullMAC clearly benefits from the caching

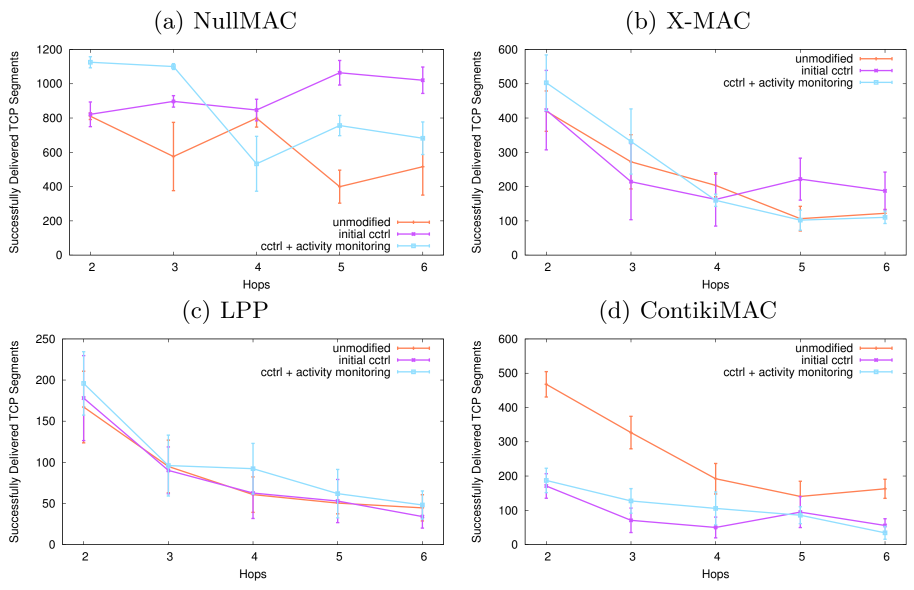

Fig. 6. Throughput with *unmodified* Contiki, the initial *cctrl* Strategy without and with the Activity Monitoring Extension

and retransmission mechanisms introduced with the *initial cctrl* design, most notably when data travels across long routes consisting of 5 and more hops, where it reaches almost twice the throughput of the *unmodified* Contiki network stack. For the shorter routes, the result is less distinctive, but the application of local retransmissions still tends to be an improvement. For X-MAC, throughput remained more or less in the same range with the *initial cctrl* approach as with *unmodified* X-MAC. An analysis of the traces yielded that retransmissions and the duplicate segment dropping features increased throughput for longer routes, but had a slightly adverse impact for shorter routes. For LPP, the introduction of the caching mechanisms had almost no impact on the end-to-end throughput at all. The curves both exhibit an astonishingly similar degradation of the throughput with increasing length of the route. With ContikiMAC, the *initial cctrl* approach dramatically decreased the amount of delivered TCP packets. ContikiMAC suffers more from increased levels of interference and competing medium access, which is probably triggered by the *early* local retransmissions of the *cctrl* module. The degrading effect of such early retransmissions is higher in ContikiMAC than in X-MAC, since ContikiMAC, after knowing the schedule offsets of its neighbors, only transmits the data frames at the announced wake-up time of the targeted receiver. A collision at this point then inevitably results in a transmission failure. In contrast, X-MAC sends out long preamble strobes preceding every frame transmission, where a collision of two strobe packets has no dramatic impact. The preamble strobes further serve to reserve the channel, since they are likely to be heard by nodes checking the channel for transmission,

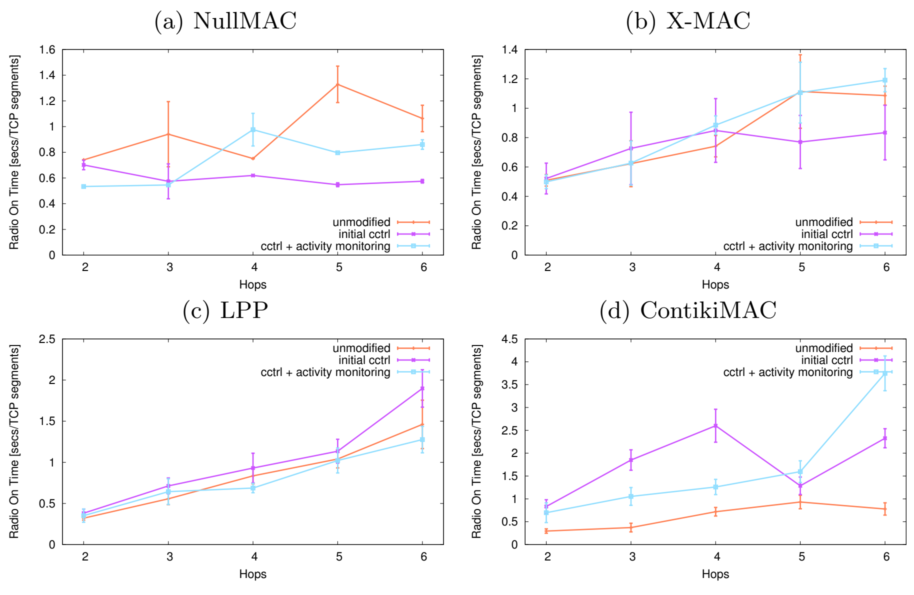

Fig. 7. Energy measured as Radio On-Time t_{on} with *unmodified* Contiki, the initial *cctrl* Strategy without and with the Activity Monitoring Extension

probably even some that are more than one hop away in the testbed topology, which alleviates the *hidden node* problem.

Figure 7 depicts the radio on-time t_{on} of all the nodes in the chain summed up and divided by the number of successfully transmitted and acknowledged TCP segments from the TCP senders to the TCP receivers. Contiki's internal power profiler [15] calculates t_{on} as the combined duration the radio spends in receive and transmit mode. Since the radio is in general by far the most power-hungry component of a WSN node, the estimation t_{on} has recently often been used for estimating the energy consumption, e.g. in [14]. The curves in Figure 7 represent a metric for the *energy-efficiency* of the different configurations, since they denote how much radio on-time has been spent per TCP segment on average. The displayed ratio tends to increase with the number of hops for the E^2-MAC protocols, since longer routes obviously require more energy to be spent per segment. In absolute numbers, all protocols operate in a similar range. The energy-unconstrained NullMAC protocol combined with our *initial cctrl* strategy even outperforms X-MAC and LPP for most route lengths, since it can transmit much more segments within the 10 min experiment time. The introduction of the *initial cctrl* mechanism tends to improve the energy-efficiency of NullMAC and X-MAC, but clearly deteriorates that of ContikiMAC.

4.3 Activity Monitoring

We examined the impact of the channel activity monitoring approach introduced in Section 3.3, where the MAC proxy is introduced between the µIP and the MAC layer, in order to make information regarding the current channel utilization available to the *cctrl* module. Our strategy then consists in making the *cctrl* module's local retransmission timeouts *dependent* of the registered activity level, transmitting *earlier* in situations of low channel activity (to avoid long wasted idle periods of the channel), and prolonging it when transmissions from neighboring are detected. We refer to this strategy as *activity dependent retransmissions*, and label it as *cctrl + activity monitoring* in the figures.

Figure 6 illustrates how this strategy affects the achieved throughput, and compares it with the *initial cctrl* strategy. NullMAC and X-MAC convey a similar behavior: for both, the *activity dependent retransmissions* increase the end-to-end TCP throughput for the 2-hop and 3-hop experiments, but decrease it for longer routes, compared to the *initial cctrl* strategy. With X-MAC the throughput even falls below the *unmodified* Contiki configuration for long routes. The LPP throughput could be improved across all route lengths, which resulted in the highest amount of successfully transmitted TCP segments with this MAC protocol so far. When comparing the maximum throughput in absolute values, LPP remains far behind ContikiMAC and XMAC, which both operate on a similar level. The energy-unconstrained NullMAC, in contrast, reaches a throughput that is at least 2-3 times as high for all route lengths. ContikiMAC also slightly benefits from the activity monitoring approach. Its throughput was increased for shorter routes (2-4 hops), or remained at roughly the same level for longer routes. However, the throughput of ContikiMAC remained remarkably below that of its configuration without any *cctrl* mechanisms. Obviously, concurrent channel activity and competition is particularly harmful for ContikiMAC.

The energy-efficiency metric of the *activity monitoring* approach, calculated as radio on-time per TCP segment, is further depicted in Figure 7. Compared to the *initial cctrl* strategy, no significant changes could be observed. This result was rather expected, since the MAC proxy remains transparent to the MAC protocol, without introducing any energy costs whatsoever.

4.4 Multiple Connections

In Section 3.4, we outlined our proposed concept of establishing a second TCP connection between the TCP client and server, which remains transparent to the application, and through which data can be continuously transmitted. During a disruption in the packet flow of one of the connections, e.g., due to a packet loss on one link, the second connection can still operate, and the channel is not left idle until the TCP retransmission is triggered. We refer to this approach as *dualconnection*. First, we evaluated whether the approach of initiating two connections without the *cctrl* basic mechanisms increases the end-to-end throughput at all. Then, we examined the initial *cctrl* caching strategy *with* a second connection (cf. *cctrl + dualconnection*) but *without* the activity monitoring, and then in combination with the activity monitoring (cf. *cctrl + dualconnection + activity monitoring*).

Figure 8 illustrates their achieved throughput, together with the *unmodified* Contiki examined beforehand. With NullMAC, the availability of a second

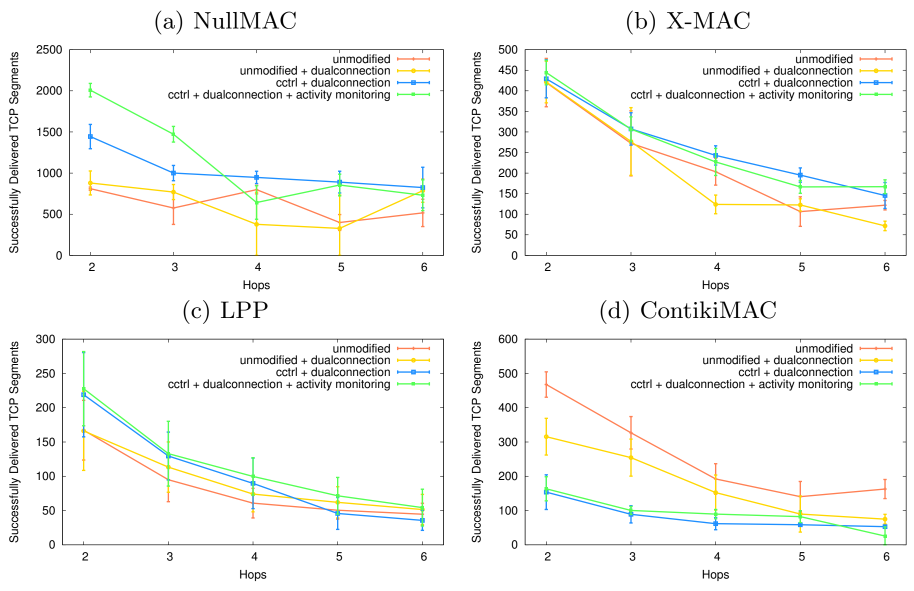

Fig. 8. Throughput with Multiple Connections and combined Strategies

connection and the initial *cctrl* caching and retransmission strategy (cf. *cctrl + dualconnection*) doubled the amount of transmitted TCP segments, across almost all tested routes, compared to the *unmodified* Contiki variant. If the *cc-trl* module applied the *activity dependent retransmission* strategy, together with the dual-connection approach, the throughput was even further increased for the 2-hop and 3-hop routes, and performed only slightly worse than the *cctrl + dualconnection* approach for longer routes. The X-MAC protocol also benefited from a second open connection, but only if the *cctrl* module was active. The X-MAC throughput was increased consistently across all route lengths, in the best case by roughly 37% (cf. 5 hops). Similar results were obtained for the LPP protocol: the experiment configurations relying on two TCP connections significantly increased the throughput, and the best results were obtained with the combination of two connections and the activity dependent retransmissions. ContikiMAC consistently remained the exception: no matter whether it was run with the dual-connection strategy, the activity dependent retransmissions or both extensions combined, its throughput persistently remained at a very low level. We presume that its degradation was mainly caused by early triggered re-transmissions colliding with the original transmissions or TCP ACKs, probably due to the *hidden node* problem, which may occur more often with ContikiMAC due to the lack of preamble strobes in advance of frames, compared to X-MAC.

4.5 Overall Comparison

The previous results provided a detailed insight into the performance of the different *cctrl* module variants for the different route lengths. Since the results sometimes vary heavily across the different configurations and route lengths, a

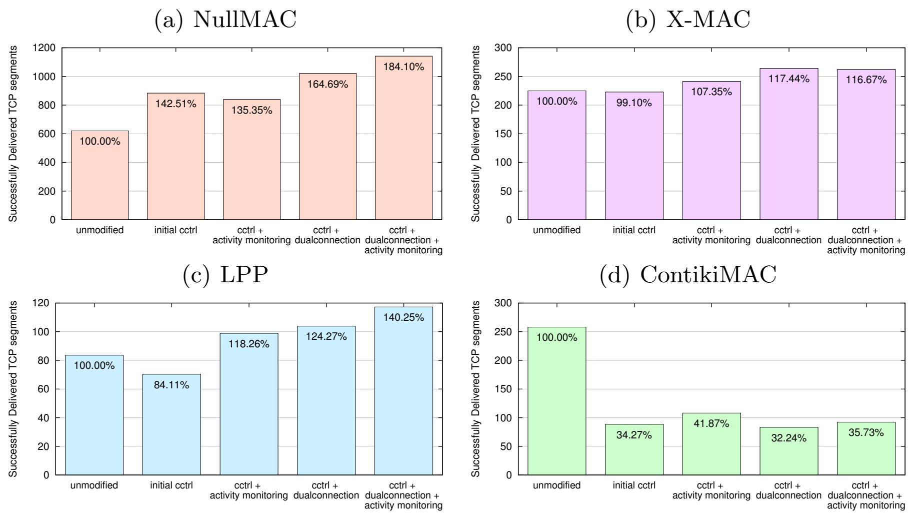

Fig. 9. Single Route Scenario: Overall Comparison of Throughput

general conclusion and guideline remains hard to derive. We therefore averaged the mean values of each different route length and configuration (= mean value of the five hop-specific values per protocol and configuration) to obtain a single one-dimensional number for each examined approach, with which the results of the different strategies become somewhat comparable.

Figure 9 displays the obtained averaged values for each examined approach and each MAC protocol. The application of distributed caching and local retransmission of TCP segments and ACKs, denoted as *initial cctrl* approach, obviously had a positive impact on the NullMAC protocol. Across all route lengths, this approach reached an improvement of 42.51% compared to *unmodified* Contiki. Our proposed extensions of *activity dependent retransmissions* as well as *multiple connections* combined finally reached the best results, with an average increase of 84% compared to the *unmodified* Contiki µIP configuration. With the X-MAC protocol, the improvement is less distinctive. When following to the *initial cctrl* design, the performance remained almost equal to the unmodified Contiki variant. The best results of 17% more transmitted TCP segments could be achieved when combining the initial *cctrl* approach with a second TCP connection. Similar results were obtained by combining this approach with *activity monitoring*. The LPP protocol behaves similarly as X-MAC when applying the different *cctrl* extensions. Again, our modifications of *activity dependent retransmissions* and the *multiple connections* combined achieve the best overall performance with X-MAC, an end-to-end throughput increase of 40%.

Figure 10 depicts the energy-efficiency metric calculated as radio on-time per transmitted TCP segment for the four protocols. The efficiency of NullMAC clearly profits from the *cctrl* module, in particular when combining all the fea-

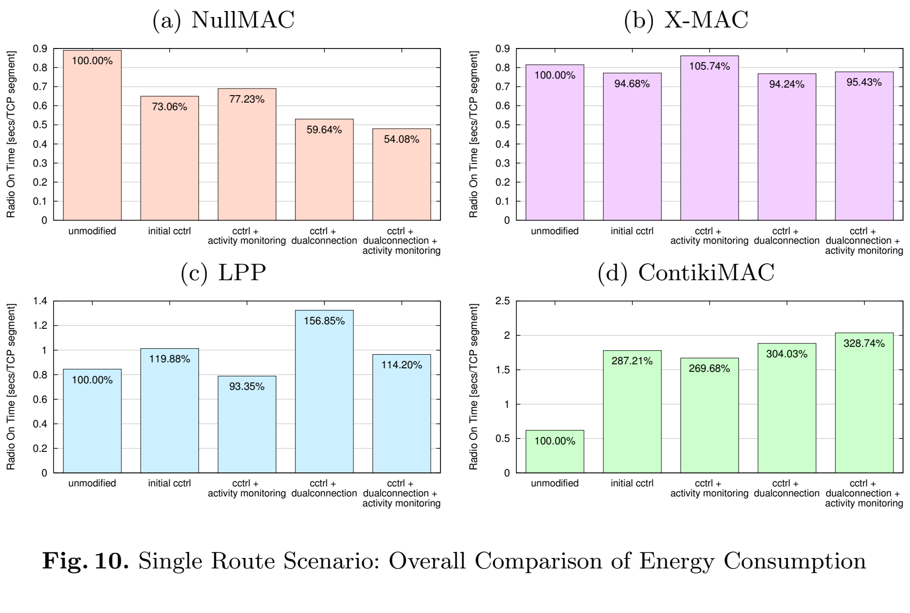

Fig. 10. Single Route Scenario: Overall Comparison of Energy Consumption

tures. Since NullMAC reaches the highest number of transmitted TCP segments over the entire experiment timespan, its efficiency is even better than that of any radio duty-cycled MAC protocol ($<$ 0.5 seconds per TCP segment). The energy-efficiency of X-MAC is slightly improved by most of the mechanisms of the *cctrl* module. LPP remains in the same range as X-MAC in absolute values. However, its efficiency varies rather strongly with the different strategies, but is neither deteriorated nor improved. ContikiMAC again constituted the negative exception among the four evaluated protocols: its energy-efficiency suffers a lot when introducing any caching strategies whatsoever. This degradation can most probably be explained by the increase in collisions and other sources of packet losses due to concurrent activity in the channel. This does not occur when only one TCP segment is in flight at any time and no local retransmissions are triggered, which is only the case with the *unmodified* variant. Since ContikiMAC acknowledges the data packets themselves, and re-attempts to transmit upon failed attempts, as opposed to X-MAC, it already integrates a certain reliability. Comparing the similar, yet even slightly better results of the X-MAC configuration with all our extensions (*cctrl + dualconnection + activity monitoring*) with that of unmodified ContikiMAC, the question whether reliability should rather be ensured on a *hop-by-hop* or *end-to-end* manner can not be answered conclusively.

5 Conclusions

In this paper, we have proposed and evaluated our *cctrl* module, a modular add-on for Contiki's µIP stack, which implements and augments the distributed caching and local retransmission features proposed in DTC [4] and TSS [5] for the Contiki OS network stack. To the best of our knowledge, our evaluation is the first to study distributed TCP caching in a real-world environment. We thoroughly examine the impact of three different radio duty-cycling energy-efficient MAC protocols on the distributed caching mechanisms proposed in [4][5], which turned out to be significant. We tested our implementations in an indoor wireless sensor node testbed using five different route lengths. Some protocols (e.g., NullMAC, LPP) generally provided a good response to the local retransmission strategies, whereas others, i.e., ContikiMAC, performed significantly worse. The *cctrl* module has been shown to increase the throughput of TCP data segments in many of the examined configurations, though it remained impossible to find an approach that maximizes the throughput with every MAC protocol. The application of our *cctrl* module with all our proposed extensions combined achieved the best results with respect to the maximum throughput for energy-unconstrained NullMAC and, also the best energy-efficiency measured as radio on-time per successfully transmitted TCP segment. For the class of E^2-MAC protocols, the configuration of X-MAC with the *cctrl* module and the *dualconnection* option achieved the highest average throughput across the different route lengths, even more than its successor ContikiMAC in any of its configurations.

References

[1] Chandran, K., Raghunathan, S., Venkatesan, S., Prakash, R.: A Feedback Based Scheme For Improving TCP Performance In Ad-Hoc Wireless Networks, Intl. Conf. on Distributed Computing Systems (ICDCS) (1998)

[2] Holland, G., Vaidya., N.: Analysis of TCP Performance over Mobile Ad Hoc Networks, ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom), Seattle, Washington, USA (1999)

[3] Balakrishnan, H., Seshan, S., Amir, E., Katz, R.H.: Improving TCP/IP Performance over Wireless Networks, International Conference on Mobile Computing and Networking (MobiCom), Berkeley, USA (1995)

[4] Dunkels, A., Alonso, J., Voigt, T., Ritter, H.: Distributed TCP Caching for Wireless Sensor Networks, Mediterranean Ad-Hoc Networks Workshop, Bodrum, Turkey (2004)

[5] Braun, T., Voigt, T., Dunkels, A.: TCP Support for Sensor Networks, Wireless On demand Network Systems and Services (WONS) (2007)

[6] Dunkels, A.: Full TCP/IP for 8-Bit Architectures, Intl. Conference on Mobile Systems, Applications, and Services (MobiSys) (2003)

[7] Dunkels, A., Groenvall, B., Voigt, T.: Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors, IEEE Workshop on Embedded Networked Sensors (EmNets), Tampa, Florida (2004)

[8] Polastre, J., Szewczyk, R., Culler, D.: Telos: Enabling Ultra-Low Power Wireless Research, International Conference on Information Processing in Sensor Networks (IPSN), Los Angeles, USA (2005)

[9] Dunkels, A., Österlind, F., He, Z.: An Adaptive Communication Architecture for Wireless Sensor Networks, ACM Conference on Embedded Networked Sensor Systems (SenSys), Sydney, Australia (2007)

[10] Wan, C., Eisenman, S.: CODA: Congestion Detection and Avoidance in Sensor Networks, ACM Conference on Embedded Networked Sensor Systems (SenSys), Los Angeles, USA (2003)

[11] Buettner, M., Gary, V., Anderson, E., Han, R.: X-MAC: A Short Preamble MAC Protocol for Duty-cycled Wireless Sensor Networks, ACM Conference on Embedded Networked Sensor Systems (SenSys), Boulder, USA (2006)

[12] Hurni, P., G.Wagenknecht, Anwander, M., Braun, T.: A Testbed Management System for Wireless Sensor Network Testbeds (TARWIS), European Conference on Wireless Sensor Networks (EWSN), 2010, Demo Session

[13] Hurni, P., Anwander, M., Wagenknecht, G., Staub, T., Braun, T.: TARWIS - A Testbed Management Architecture for Wireless Sensor Network Testbeds, International Conference on Network and Service Management (CNSM), Paris, France (2011)

[14] Dunkels, A., Mottola, L., Tsiftes, N., Osterlind, F., Eriksson, J., Finne, N.: The Announcement Layer: Beacon Coordination for the Sensornet Stack, European Conference on Wireless Sensor Networks (EWSN) (2011)

[15] Dunkels, A., Osterlind, F., Tsiftes, N., He, Z.: Software-based On-line Energy Estimation for Sensor Nodes, IEEE Workshop on Embedded Networked Sensors (EmNets), Cork, Ireland (2007)