TCP Support for Sensor Networks

Torsten Braun
University of Bern, Switzerland
braun@iam.unibe.ch

Thiemo Voigt and Adam Dunkels
Swedish Institute of Computer Science
thiemo@sics.se, adam@sics.se

***Abstract*—Many applications of wireless sensor networks require external network connectivity to enable communication between monitoring / controlling entities and sensors. By using the TCP/IP protocols inside the sensor network, external connectivity can be achieved to any other IP node at the edge or outside the sensor network. TCP can then be used for remote management and reprogramming of sensor nodes. However, high bit error rates lead to energy inefficiencies that reduce the lifetime of a sensor network. This paper introduces an approach to support energy-efficient operation of TCP in sensor networks. The concept called TCP Support for Sensor nodes (TSS) allows intermediate sensor nodes to cache TCP segments and to perform local retransmissions in case of errors. TSS does not forward a cached segment until it knows that the previous segment has been successfully received by the next hop node. This forms a kind of congestion control and reduces the total number of packets in the sensor network. Simulations show that TSS significantly reduces the number of TCP segment and acknowledgement transmissions compared to TCP without TSS.**

Keywords-Sensor Networks, Transport Control Protocol

I. INTRODUCTION

A. Motivation and Application Scenarios

Wireless sensor networks are composed of a large number of radio-equipped sensor devices that autonomously form networks through which sensor data is transported. The devices are typically severely resource-constrained in terms of energy, processing power, memory, and communication bandwidth. Many applications of wireless sensor networks require an external connection to monitoring and controlling entities (sinks) that consume sensor data and interact with the sensor devices.

Running TCP/IP in the sensor network allows connecting the sensor network directly to IP-based network infrastructures without proxies or middle-boxes. To deploy a sensor network, we just need Internet connectivity, which can also be achieved by sensor nodes with GPRS or WLAN interfaces. This even allows putting the sink into the fixed part of the IP network. In particular for disaster recovery, deployment would be simpler and faster, when only a single type of sensor node but no other nodes such as protocol proxies or sinks needs to be deployed.

In such a scenario each sensor device is able to communicate via TCP/IP. A single standard protocol suite can then be used. Data to and from the sensor network can be routed via any device with Internet connectivity rather than via sinks or protocol proxy nodes only. This also simplifies to maintain connectivity of the sensor network. Routing all traffic via a single or a few sinks could easily lead to network partitioning due to the heavy load put on nodes close to the sinks. On the other hand, sensor nodes with IP connectivity not only allow distributing the forwarding load but also deploying sensor nodes without having gateways at many smaller areas.

Data transport in IP-based sensor networks can be performed using UDP and TCP. UDP is used for sensor data and other information that do not use reliable byte-stream unicast transmission. TCP should be used for administrative tasks that require reliability and compatibility with existing application protocols. Examples of such tasks are configuration and monitoring of individual sensor nodes as well as download of binary code and data aggregation descriptions to sensor nodes. In particular, downloading code to designated nodes such as cluster heads in a certain geographical region requires a reliable unicast protocol.

Figure 1 shows possible TCP connections in a sensor network. The sink has two TCP connections for configuring the black sensor nodes, while a kind of multicast overlay tree consisting of several (grey) sensor nodes can be established for code or query distribution based on TCP connections. Reliable multicast might be required, if a group of sensor nodes but not all sensor nodes need to be configured or reprogrammed. A subset of homogeneous nodes in a heterogeneous environment may form a group and all group members need to receive the same binary code image. Moreover, members of a group may perform the same task such as object tracking in a certain area. Other sensor nodes may be responsible for other tasks such as temperature monitoring. Sensor nodes with the same task belong to the same group and need to be configured appropriately.

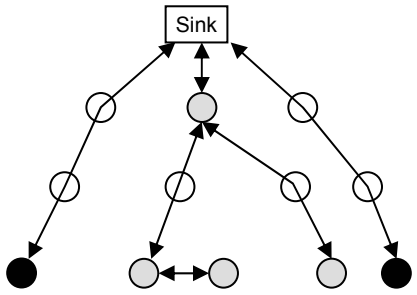

Figure 1: TCP connections in a sensor network

B. Challenges with TCP/IP in Wireless Sensor Networks

On might argue that TCP/IP implementations consume too many sensor network resources. However, recent work showed that TCP/IP can be implemented on sensor nodes with limited processing power and memory [1].

TCP/IP may result in relatively large headers that may add significant overhead in case of short packets. However, we assume that TCP is mainly used for configuration and programming tasks, where a rather high amount of data is transferred and packets become rather large. Moreover, we propose to develop a TCP/IP header compression scheme for sensor networks. Due to the stateful approach proposed in this paper such a scheme should be feasible, but we leave this issue for future work.

Other problems to be solved for TCP/IP in sensor networks are related to addressing. While in traditional IP networks IP addresses are assigned to each network interface based on the network topology, IP-based sensor networks may use spatial IP address assignment based on node locations, which might be relative to a base station location [2].

While in traditional IP networks, packets are transparently routed through the network based on the network topology, data centric routing mechanisms are often preferable in wireless sensor networks [3]. To implement data centric routing in IP-based sensor networks, application overlay networks might be used.

It is also well known that TCP has serious performance problems in wireless networks [4]. One problem is that TCP, which has been designed for wired networks with low bit error rates, interprets packet loss as an indication of congestion and decreases its transmission rate in case of a lost packet. This results in low throughput. The main problem for sensor networks operating autonomously with constrained power supply is the energy-inefficiency of TCP. This is caused by TCP's end-to-end retransmission scheme requiring that lost packets are retransmitted by the original sender of the packet. In a multi-hop network, the retransmitted packets must be forwarded by all intermediate nodes from sender to receiver, thus consuming valuable energy at every hop. In general, end-to-end error recovery is not a good approach for reliable transport in sensor networks, because the per-hop packet loss rate may be in the range of 5% to 10% or even higher [5].

C. TCP Support for Sensor Networks

In this paper we introduce an approach that overcomes the energy efficiency and performance problems: TCP Support for Sensor nodes (TSS). TSS lets intermediate nodes cache TCP data segments and perform local retransmissions after packet loss has been detected. TSS does not require any changes to TCP or the TCP implementations at end points. An intermediate node caches a TCP segment until it knows that the next-hop node has received it. A TCP segment with the next higher sequence number than the one in the cache will not be forwarded but stored in a buffer. The buffer can only store a single packet and can be used for the packet with the next sequence number. This mechanism causes a kind of backpressure at the nodes along the path towards the sender and is intended to reduce the number of packet transmissions. Furthermore, TSS uses an aggressive TCP acknowledgement recovery mechanism to repair TCP acknowledgement loss, because TCP acknowledgements are important for several TSS mechanisms. Our results show that TSS significantly enhances TCP performance both in terms of the overall number of transmitted TCP segments and throughput.

D. Overview

While TSS focuses on TCP support, several protocols for reliable data transfer in sensor networks introduce new transport protocols and do not attempt to support TCP operation. Section II gives an overview about related work in this area. We propose to use TCP for reliable data transfer in sensor networks and introduce TSS in Section III. Section IV describes performance results for reliable TCP data transfer across a multi-hop wireless sensor network using TSS. Section V concludes the paper.

II. Related Work

A. Reliable Data Transport

TSS extends ideas that have been introduced by Distributed TCP Caching (DTC) [6]. DTC aims to avoid energy-costly end-to-end retransmissions by caching TCP segments inside the network and retransmitting segments locally, i.e. from the intermediate sensor nodes' caches, when packet loss occurs. DTC assumes limited memory resources available for caching and proposes to cache a single segment per node. Nodes try to cache segments that have presumably not been received by the next node. To achieve that each segment may be cached somewhere in the sensor network, nodes cache TCP segments with the highest segment number seen based on a certain probability. DTC uses implicit or explicit link layer acknowledgements in order to detect packet loss at the next hop. Segments are locked in the cache indicating that it should not be overwritten by a TCP segment with a higher sequence number. A locked segment is removed from the cache only when a TCP acknowledgement acknowledging the cached segment is received, or when the segment times out. Each node measures the round-trip time (rtt) to the receiver and sets the retransmission timeout to 1.5 * rtt. Since these rtt values are lower than those estimated by the TCP end points, the intermediate nodes are able to perform retransmissions earlier than the TCP end points. DTC uses the TCP selective acknowledgement (SACK) option to detect packet loss and to inform other nodes about the segments locked in the cache. DTC does not cache and retransmit TCP acknowledgements, but locally regenerates a TCP acknowledgement when an intermediate node sees a TCP data segment, for which it has already received and forwarded a TCP acknowledgement. TSS mainly differs from DTC by the backpressure mechanism that keeps segments in the cache until a node knows that the previous segments have been received by the next hop node. This allows implementing some kind of congestion control mechanism based on backpressure signals at the sender. TSS does not use TCP options such as selective acknowledgements

and retransmissions and hence requires less re-sequencing buffers at the receiver.

TSS extends DTC, while DTC has been inspired by the Snoop [4] protocol. Snoop has been developed for supporting TCP over wireless access networks. The Snoop agent is deployed at an intermediate system between the wireless and wired part of the network. The agent buffers TCP segments that have not yet been acknowledged by the receiver and detects TCP segment loss by analysing TCP acknowledgements. The agent can perform local retransmissions and suppress TCP acknowledgements in order to avoid duplicate acknowledgements at the sender. Duplicate acknowledgements might cause end to end retransmissions for packets that could also be recovered locally by the agent.

Reliable Multi-Segment Transport (RMST) [7] has been designed for its use together with directed diffusion. RMST is used for sensor data transfer but not for control data transfer such as TSS. It can provide a caching mechanism within the intermediate nodes, but requires additional negative acknowledgement (NACK) messages. These are sent by an intermediate node to its upstream neighbour, when it detects, e.g. using timeouts, holes in the data flow. As a reaction on NACK messages, an upstream node can retransmit cached packets. The authors assume a limited number of bytes in flight (< 5 KB) and that the intermediate nodes can completely cache this amount of data. For packet loss rates below 10 % the combined caching and NACK mechanism is more efficient than pure link level ARQ approaches. On the other hand, processing of NACK messages in end points only is extremely inefficient for packet loss rates above 10 %. These results are consistent with design principles of TSS. Similar as RMST, TSS uses caching and local retransmissions by intermediate nodes without introducing pure link level ARQ, but relies on information from transport packets only.

Pump Slowly Fetch Quickly (PSFQ) [5] is a reliable transport protocol for re-tasking and re-programming of sensor nodes. The main PSFQ idea is to pump data rather slowly towards the receiving sensor nodes, but to recover missing data locally from intermediate nodes. The pump operation aims to support quick forwarding in case of no errors and behaves like a store and forward approach in situations with a high number of errors. The pump operation is based on broadcasting packets hop-by-hop from source to destination. Segment numbers are used to discover duplicates. Nodes receiving a packet add random delays before re-broadcasting in order to avoid collisions. While packet forwarding based on re-broadcasting have significant advantages in dynamic environments such as mobile ad-hoc networks and networks with unsynchronized sleep cycles [8], simulation experiments have shown that already a low number of packet losses due to congestion or bit errors can cause a significant number of duplicated packets. Duplicates, however, cause unnecessary packet reception, processing and transmissions, which should be avoided in energy-efficient sensor networks. The fetch operation is based on proactively requesting retransmissions from neighbour nodes using NACK messages. If the last message of a packet sequence is lost, a fetch operation is triggered by a timeout. Multiple lost messages can be recovered in a single fetch operation. In addition to NACK messages, PSFQ introduces report messages for reporting the reception status at the destination to the source. The backpressure mechanism of TSS has a similar effect as the pump operation: Packet forwarding will be slowed down as soon as errors are detected by the intermediate nodes. A TSS node stops forwarding a packet, if previous packets have not been forwarded by successor nodes. PSFQ is focusing on code distribution using broadcast. Broadcasting is efficient and feasible to program and configure homogeneous sensor nodes. TSS rather focuses on communication with single nodes or smaller groups of nodes. PSFQ introduces NACK messages, while TSS supports standard mechanisms based on TCP acknowledgements and timeouts.

Event-to-Sink Reliable Transport (ESRT) [9] aims to support reliable sensor data transport in wireless networks. It includes congestion control and mechanisms to achieve reliability. Reliability is controlled by adapting a rate at which the sink sends state reports back to the source. The frequency of the reports depends on the observed and desired reliability as well as the needs from congestion control. As in the case of PSFQ, a special protocol has been proposed, while no transport protocol extensions are required in TSS.

B. Congestion Control

Congestion control is very important in wireless sensor networks, because overloading a wireless network by too many transmissions can increase the collision probability. Collisions lead to packet losses and unnecessary retransmissions, which make sensor network operation energy-inefficient. TCP congestion control limits the maximum window size according to the slow start congestion control algorithm. However, it even might make sense to further limit the window dependent on the number of intermediate hops in a wireless multi-hop network, because the optimal window size in terms of throughput might be below the window size of standard TCP [10]. For example, it has been proposed to limit the maximum congestion window size to h/4 (h: number of hops) in a chain of nodes that are 200 m away from each other and have 250 m transmission range and 550 m interference range. This result shows that it might be beneficial to limit the TCP congestion window in wireless multi-hop networks. The backpressure mechanism used in TSS (cf. Section III.A.4) has a similar effect and can limit the packets in transit to an appropriate number, if the TCP source implementation makes use of the backpressure signal from the TSS implementation on the local node.

Congestion Detection and Avoidance (CODA) [11] is based on congestion detection by monitoring channel utilization and buffer occupancy at the receiver. Detected congestion situations are signalled towards the source using backpressure signals (open-loop). Nodes receiving backpressure signals throttle down their transmission. In addition, a closed-loop mechanism operates on a longer time-scale. Based on acknowledgements received from the sink, sources regulate themselves. Lost acknowledgements result in reducing the rate

at the source. Again, in contrast to TSS, new signalling messages need to be introduced into CODA.

C. Caching for Recovery from Disconnection

While several approaches perform packet caching for local retransmissions in case of packet loss due to congestion or lossy channels, other related works apply caching to recover from more serious errors such as disconnection of networks or route breaks.

The design of a smart link layer is proposed in [11]. Packets might be re-received after a disconnection in order to re-trigger TCP after a longer disconnection period by putting TCP packets such as acknowledgements again into the input TCP queue. Re-sending packets to the peer can also facilitate restart of TCP in such a case. The proposed mechanisms are rather orthogonal to the concepts proposed in this paper.

In [13] it is also proposed to hold copies of forwarded packets in a cache. When a downstream node encounters an error with packet forwarding, a route error message might be sent to the upstream node. The cached packet can then be retransmitted possibly on multiple alternative routes in order to repair the route break.

TCP with BUffering capability and Sequence information (TCP-BUS) [14] proposes to buffer packets during route disconnection and re-establishment. After a route becomes available again, buffered packets are retransmitted by intermediate nodes. Special control messages are used to indicate route breaks and re-establishments. TCP can adapt its behaviour dependent on the knowledge that packets have been lost for other reasons than congestion. TSS does not explicitly focus on route breaks but can be applied to such failures as well.

III. TCP Support for Sensor Nodes

A. Protocol Mechanisms

TCP Support for Sensor nodes (TSS) aims to support energy-efficient operation of sensor nodes and forms a layer between TCP and the routing layer to be implemented in a communication protocol stack of sensor nodes. TSS should ideally be implemented in TCP sensor nodes with senders and receivers as well as in intermediate sensor nodes that relay TCP (data) segments and acknowledgements of a TCP connection. TSS tries to reduce the number of transmissions by several mechanisms:

- Caching of packets that might not have been received by the successor node (next node) based on overhearing and TCP acknowledgement spoofing.
- Local retransmission of TCP segments based on round trip time estimation.
- TCP acknowledgement regeneration and recovery based on forwarding delay estimation and overhearing.
- A backpressure mechanism avoiding that a node forwards a packet if the successor node might not have received all previous packets.

The TSS mechanisms do not require explicit link or MAC level acknowledgements, but TCP segments and acknowledgements are the only packets that are needed. This approach further reduces the amount of transmissions and can be used on top of any kind of sensor network MAC layer. By ensuring in sequence arrival of TCP segments at the destination, TSS avoids any re-sequencing buffer and selective acknowledgement / retransmission extensions in TCP.

1) Caching

An intermediate node caches a segment until it is sure that the successor node towards the destination has received the segment. A node knows this when it detects that the successor node has forwarded the segment (implicit acknowledgement) or when it spoofs a TCP acknowledgement that has been sent from the destination toward the source of the TCP segment. Nodes are assumed to listen to packet transmissions of their neighbour nodes in order to be able to detect whether the neighbour nodes have forwarded TCP segments. One might argue that forcing sensor nodes to overhear packets does not support energy efficient operation. On the other hand, a forwarding node should only listen to other's transmissions for a very short time. An alternative would be explicit link level acknowledgements. However, this would not only require the node to listen and receive but also the successor node to transmit an additional acknowledgement packet. Typically, a packet will be forwarded immediately by the successor node and only in case of packet loss a node must overhear for the whole retransmission timeout interval (cf. subsection IV.B.4)). A packet that is known to be received by the successor node will be removed from the cache. In addition to the cache, TSS requires another packet buffer (simply called buffer hereafter) for temporarily storing the next packet that is waiting to be forwarded to the successor node.

2) Local Retransmissions of TCP Segments

All intermediate nodes are able to perform local retransmissions, when they assume that a cached segment has not been received by the successor node towards the destination. Retransmissions are triggered by timeouts, which requires intelligent setting of timeout values. The retransmission timeout is set to 1.5 * rtt and allows to repair even multiple packet losses before an end-to-end retransmission timeout is triggered. TSS simulations showed that a retransmission timeout of 2 * rtt performs slightly worse. The maximum number of local retransmissions has been limited to four. It might happen that a node's retransmission timeout expires, if it has received an overheard packet header with an error and dropped that implicit acknowledgement. Then, the node retransmits a TCP data segment although that one has already been received and forwarded by the successor node. In this case, the already correctly forwarded TCP segment should not be forwarded again. Forwarding should be prevented by a small history list consisting of the last few (here: ten) forwarded packets to filter out all segments that have been forwarded previously. Retransmitted TCP segments can be uniquely identified by the source address and the IP identification field. Of course, end-to-end retransmissions should not be filtered in order to support end-to-end recovery in serious error situations.

3) Regeneration and Recovery of TCP Acknowledgements

TCP acknowledgements are extremely important for TSS, since several mechanisms such as round-trip-time estimation, retransmission, and caching depend on it. Experiments have shown that loss of acknowledgements may have a severe impact on the amount of TCP segment transmissions. TSS deploys two mechanisms for retransmissions of TCP acknowledgements that help to decrease the number of TCP segment transmissions significantly: a local acknowledgement regeneration mechanism and an aggressive recovery mechanism. The local acknowledgement regeneration mechanism becomes active when a node receives a TCP data segment, which has already been acknowledged by the destination. In that case, the TCP segment is dropped and a TCP acknowledgement with the highest acknowledgement number is regenerated and transmitted toward the source. The aggressive recovery mechanism recovers TCP acknowledgements, if a node has not discovered the forwarding of the TCP acknowledgements by the successor node. Since TCP acknowledgements should usually be forwarded without significant delay towards the sender of TCP segments, each node measures the time between its own TCP acknowledgement transmission to the successor node and the overhearing of the TCP acknowledgement transmission from the successor node towards the TCP segment sender (source). Similar as for the rtt estimation we use exponential averaging. We set the TCP acknowledgement retransmission timeout to the double average value. After timeout expiration, a TCP acknowledgement is recovered using the highest acknowledgement number.

4) Backpressure Mechanism

If the successor of a node has not forwarded all received packets, there might be a problem in the network. For example, the network might be congested or packet forwarding does not make progress, because a previous TCP segment with bit error needs to be recovered first. If a node would continue with packet forwarding in such a case, the risk of unnecessary transmissions would be rather high. In a congestion situation, a forwarded segment might easily get lost then. The same is true in case of a lost packet due to bit errors. In such a situation all caches on subsequent nodes are occupied and the transmission of a new packet would not be protected by caching. For that reason, a TSS node stops any forwarding of subsequent packets until it knows that all earlier packets have been received and forwarded by its successor. Successful forwarding can be detected by overhearing the forwarded packet or by detecting a TCP acknowledgment for that TCP segment. If packet forwarding stops at some point, all other nodes in the chain behind the stopping node will also stop their transmissions until progress is detected at their respective successor nodes. In case of a lost packet (due to congestion or bit errors) packet loss should be recovered by the node that forwarded the packet at last. In that case, we have to avoid that retransmissions are triggered by nodes behind the recovering node, i.e. the nodes closer to the sender. This can be achieved by increasing the retransmission timeouts at the nodes closer to the sender. For that reason, the mechanism ensuring that the retransmission timeouts increase along the nodes from the receiver to the sender as explained in subsection III.A.2) perfectly fits to the backpressure mechanism. The backpressure mechanism should also be implemented at the sender end point. We propose to not increase the TCP congestion window as long as there are a certain (here: three) number of packets waiting at the sender for transmission.

B. Example Operation

Figure 2 illustrates the operation of TSS. The first segment is forwarded without error from sender to receiver, while the second segment is lost between nodes 4 and 3. We assume here that node 5 overhears the forwarded packet from 4 to 3 and that node 5 therefore assumes that node 4 has successfully forwarded the segment to 3. This situation can easily occur, if node 4 is closer to node 5 than to node 3 or if the transmission from 4 to 3 is disturbed by another transmission such as from 1 to 2, while the latter one does not disturb the transmission from 4 to 5. In our example, node 4 caches the second segment and will time out. In order to avoid that node 4 has to drop the third segment sent by node 5, we have to provide a buffer for the third segment. This segment will not be forwarded by node 4 and, therefore, node 5 will stop forwarding subsequent packets. Assuming the nodes have measured the rtt as described in the previous section, node 4 times out before node 5, retransmits the second segment to node 3 and will continue with transmitting the third segment. Node 5 will overhear the transmission of the third segment and continue with forwarding the fourth packet. In general, the timeouts (resulting from the measured rtts) at nodes closer to the TCP receiver must be smaller than the timeouts at nodes closer to the sender. If we assume the minimum round trip measured for the first segment, we see that node 4 times out before node 5. More severe problems result from multiple packet losses. For example, if in our scenario the retransmission of the second segment by node 4 would be unsuccessful again, nodes 5 would time out too early and retransmit unnecessarily.

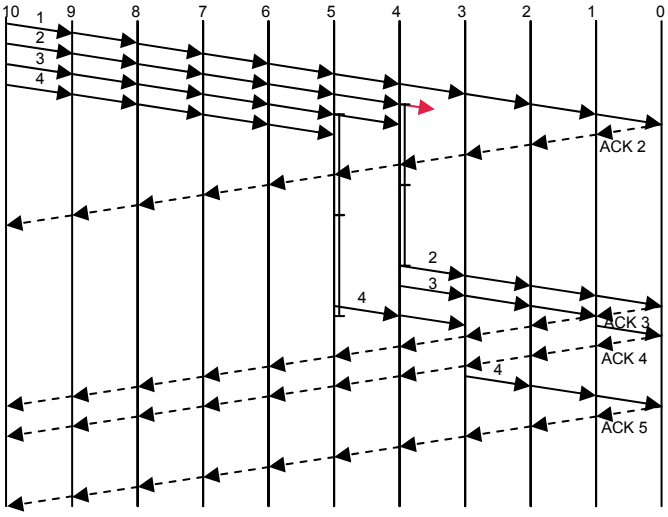

Figure 2: TSS operation in case of a lost TCP segment

In another scenario shown in Figure 3 an error might occur with the second segment between nodes 4 and 3. The segment is correctly received by node 4 and node 3 forwards it correctly to node 2. However, node 4 does not even receive the packet header of the second packet forwarded from node 3 to node 2. Therefore, node 4 assumes that the second packet has not been received by node 3 and stops the transmission of new packets. After a while the TCP acknowledgement for segment 2 arrives at node 4, which can then continue to forward the third segment.

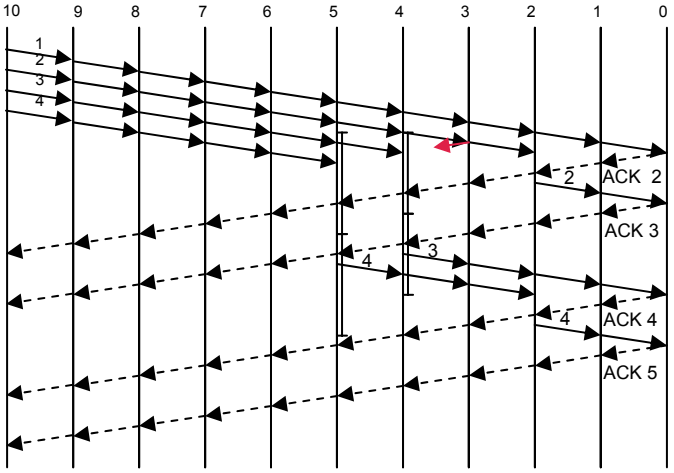

Figure 3: TSS operation in case of an overhearing error

C. Pseudo Code

The operation of a TSS node in an intermediate system is presented in more detail by the C-like pseudo code below. The first part (1) describes acknowledgement timeout processing, i.e. when the node has not detected the forwarding of an acknowledgement by the next node towards the source. This implements the aggressive recovery scheme for TCP acknowledgements.

The second part (2) shows processing in case of a TCP segment retransmission timeout. Retransmissions are only performed, if the data to be retransmitted have not been confirmed by an implicit acknowledgement or by an TCP acknowledgement.

The main part (3) describes processing of received acknowledgement and data segment packets. Part 3a describes normal processing, when a TCP data segment or TCP acknowledgement has been received for forwarding. A newly received acknowledgement might confirm that some data have been received by the successor node. In that case, a segment waiting in the buffer might be forwarded by the node. The received acknowledgement might also stop an ongoing rtt measurement. If the acknowledgement acknowledges previously acknowledged data again, we drop it, but forward it towards the source otherwise. Data processing in part 3a is applied to packets that need to be forwarded towards the destination. If there is a gap between the packet's sequence number and the sequence number that the successor node has received, the packet needs to be stored in the buffer before it can be forwarded. The packet may also include data that has all been acknowledged by the destination. In that case, it is not forwarded further, but an acknowledgement is regenerated and sent towards the source. If all transmitted data have been confirmed and the packet contains the next unconfirmed byte, the packet can be forwarded immediately and a new rtt measurement might be started if such a measurement is not yet going on.

Part 3b shows processing of an overheard packet. In case of an acknowledgement, the acknowledgement timer is cancelled and the time needed by the upstream node to forward an acknowledgement is measured for calculating the acknowledgement retransmission timeout. For an overheard data packet that has been cached, the retransmission timer is cancelled as well and the cache is released. If there is another packet waiting in the buffer, it will be forwarded if it is eligible. However, the forwarding must be delayed in order to reduce the risk of collisions. Simulations have shown that immediate forwarding significantly increases the collision probability.

```
switch(event){
 case ack_timeout: // -1-
  retransmit_ack(acknowledged);
  start_timer(ack_timer, acknowledged,
    γ^attempts++ * ack_forwarding_time);
  break;
 case retransmission_timeout: // -2-
  sequence_no =
   sequence_number_of_packet_to_be_retransmitted;
  if ((sequence_no + length > confirmed){
   retransmit_data(sequence_no);
   if (number_of_retransmissions > limit)
    delete(cache);}
  break;
 default: // -3-
  if (packet_has_bit_error || ttl_expired ||
   (own_address != next_address) &&
   (own_address != previous_address))
     delete(packet);
  else if (next_address == own_address) { // -3a-
   switch (type_of_packet){
    case ack:
     acknowledged = max(ack_no - 1, acknowledged);
     if ((acknowledged > confirmed) &&
      ((byte[acknowledged+1]∩buffered_packet)≠ Ø)){
       forward(buffered_packet);
       move(buffered_packet, cache);
       transmitted =
        sequence_number_of_buffered_packet +
         length-1;
       start_timer(retransmission_timer,
        sequence_no, β * rtt);
       confirmed = acknowledged;}
     if (ongoing_rtt_measurement &&
      (ack_no > rtt_sequence_no)){
        rtt= (1-α) * rtt + α *
         (current_time-start_of_measurement);
        ongoing_rtt_measurement = FALSE;}
     if (ack_no <= ack_forwarded)
      delete(packet);
     else {
      forward(packet);
      start_timer(ack_timer, ackno,
       γ * ack_forwarding_time);
      attempts = 1;}
     break;
```

```
  case data:
   if (sequence_no > transmitted + 1)
    delete(packet);
   else if ((sequence_no > confirmed + 1) &&
    (buffer_is_empty ||
    (sequence_no < seqno_of_buffer)))
      move(packet, buffer);
   else if (sequence_no + length - 1
    <= acknowledged){
     retransmit_ack(acknowledged);
     start_timer(ack_timer,acknowledged,
      γ*ack_forwarding_time);
     attempts = 1;
     delete(packet);}
   else if ((transmitted == confirmed) &&
    (byte[confirmed + 1] ∩ packet) ≠ Ø)){
      if (! ongoing_rtt_measurement){
       ongoing_rtt_measurement = TRUE;
       rtt_sequence_no = sequence_no;
       start_of_measurement = current_time;}
      forward(packet);
      transmitted = sequence_no + length - 1;
      move(packet, cache);
      start_timer(retransmission_timer,
       sequence_no, β * rtt);}
   else
     delete(packet);}
else if (own_address == previous_address){ //-3b-
 switch (type_of_packet){
  case ack:
   ack_forwarding_time =
    (1 - α) * ack_forwarding_time + α *
    (current_time - transmission_time(ack_no));
   cancel(ack_timer, ack_no);
   ack_forwarded = ackno;
   break;
  case data:
   if (sequence_no + length - 1 > confirmed){
    cancel(retransmission_timer, sequence_no);
    delete(cache);
    confirmed = sequence_no + length - 1;
    if (byte[confirmed + 1]∩buffered_packet ≠ Ø){
     forward_delayed(buffer);
     transmitted = sequence_no_of_buffered_packet
      + length - 1;
     move(buffer, cache);
     start_timer(retransmission_timer,
      sequence_no_of_buffer, β * rtt);}}}
delete(packet);}}
```

IV. PERFORMANCE EVALUATION

A. Simulation Scenarios and Parameters

TSS has been evaluated using simulations with Omnet++ [15], because of its power and simplicity and because DTC has been evaluated by this tool too. The used simulation scenario is depicted in Figure 4. The TCP sender implementation at node 10 and the TCP receiver implementation at node 0 exchange 1000 TCP segments with a payload size of 1000 bits plus TCP/IP and MAC header (= 20 + 20 + 12 bytes = 416 bits). Two end points and nine intermediate nodes (nodes 1-9) with a distance of 200 m between each node are interconnected in a chain structure. A transmission range of 200 m is feasible with various sensor nodes such as ESB [16] or WiseNet [17] nodes in outdoor environments. The chain scenario in Figure 4 shows a rather typical scenario in sensor networks, when a sink needs to configure a single node. Cross traffic does not occur, if there is a single sink communicating with a single node or a group of nodes at one instant. For multiple TCP connections in a multicast overlay we expect interferences rather at the sink or branch nodes. Moreover, TCP connections may compete with sensor data flows from sources to sinks. Interference issues are left for future work.

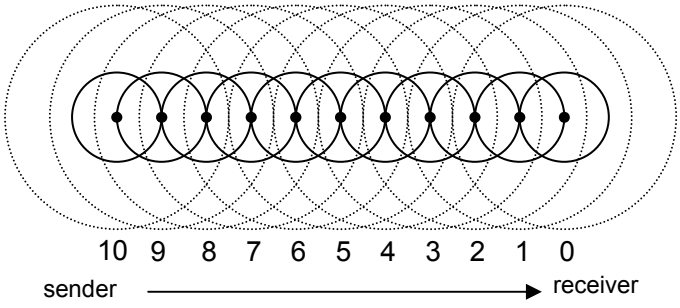

Figure 4: Simulation scenario

The TSS implementation running on each node includes a CSMA MAC implementation, which senses the transmission medium and backs off in case of a busy medium. In order to save energy we back off without sensing the medium for a random time between 1τ and 3τ with τ = time to transmit a 1000 bit payload TCP segment. Furthermore, we assume equal transmission power of all senders. A receiver can correctly receive a packet from a sender if it is not further away than 200 m and the signal to noise ratio is less than 10 dB. A receiver can detect an ongoing transmission if it receives a signal that is equivalent to a sender 500 m away.

Intentionally, we did not implement an RTS/CTS collision avoidance scheme, since such a scheme may be very costly, create a 40 % overhead and may not avoid all collision situations [18], in particular when RTS/CTS packets can not be received correctly. Note that RTS/CTS doubles the number of packet / acknowledgement transmissions. We rather propose to avoid collisions on a higher layer than MAC level. For example, if a node has recently forwarded a segment to the receiver, subsequent segments should not be forwarded immediately but slightly delayed. We implemented such a collision avoidance scheme in TSS by the function `forward_delayed` used in the last case statement of the pseudo code at Section III.C. This approach is somewhat similar to the adaptive rate control scheme proposed in [18].

We also assume that the MAC layer does not use explicit acknowledgements. Again explicit acknowledgements are considered as too costly. The bit rate of the wireless network is 100 kbps. Moreover, we assume that a node considered an overheard TCP segment as correctly received, if the TCP/IP and MAC header (416 bits) has been received without error. We investigated certain uniformly distributed bit error rates [19], in particular no (0), low (10^{-6}), medium (10^{-5}) and high (10^{-4}) bit error rates. Such error models are rather disadvantageous for our scheme, since a single bit error temporarily stops packet forwarding in a chain of nodes. The bit error rates used result in up to 15 % packet error rates. Similar packet error rates have been used in [7] and measured for connected networks in [20]. For throughput and packet

transmission measurements we performed 100 simulation runs per experiment with 1000 TCP segment transmissions from source to destination. For local rtt measurements, overhearing time evaluation, and congestion control considerations we performed a single simulation run with medium bit error rate.

B. Performance Results

1) Packet transmissions

The number of packet transmissions is the most important metric, because the energy efficiency strongly depends on it. Table 1 shows the number of TCP data segment and acknowledgement transmissions for different bit error rates.

Bit error rate	0	10^{-6}	10^{-5}	10^{-4}
TCP				
transmitted TCP segments	1067600	1081090	1197001	3499974
transmitted TCP ACKs	1001000	1003015	1019395	1217739
total number of packets	2068600	2084105	2216396	4717713
e2e retransmissions	33300	34494	45991	474776
throughput [bps]	1955	1811	831	7
TSS (backpressure in end point)				
transmitted TCP segments	1002061	1016829	1058486	1231501
transmitted TCP ACKs	1001600	1000467	1002887	1075384
total number of packets	2003661	2017296	2061373	2306885
e2e retransmissions	0	146	233	1552
throughput [bps]	4997	4412	2969	465
TSS (maximum congestion window = 3)				
transmitted TCP segments	1002061	1011693	1046849	1200717
transmitted TCP ACKs	1001600	1003210	1015297	1092203
total number of packets	2003661	2014903	2062146	2292920
e2e retransmissions	0	199	417	1909
throughput [bps]	4997	4309	2626	288
Optimal number of transmitted TCP segments	1001500	1002919	1015782	1153852

Table 1: Packet transmissions and throughput

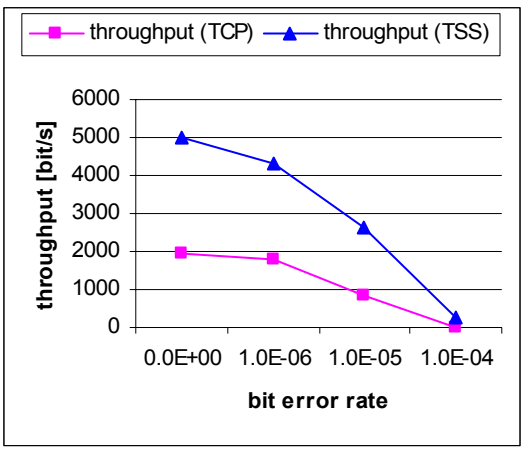

Figure 5: Throughput of TCP and TSS

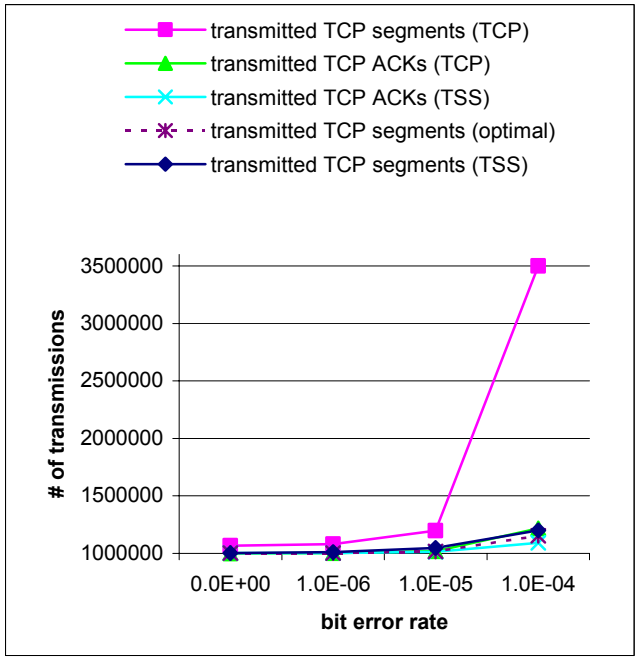

Figure 6: Packet transmissions of TCP and TSS

For TSS we used two variants: In the first variant (backpressure in end point) we combined the backpressure mechanism with the TCP congestion control. The congestion window is increased after receiving a TCP acknowledgement, if there are more than a certain number (here: three) TCP segments waiting for transmission at the source node. In the second variant (maximum congestion window = 3), we limited the maximum congestion window dependent on the number of hops according to [10].

The first TSS variant resulted in better throughput performance, but required slightly more packet transmissions. Note that the first variant is independent from the topology, but has a similar effect as the limitation used in the second variant. In particular for higher bit error rates the throughput improvement is higher. For high bit error rates several packets might be cached and buffered in the sensor network, but might wait for forwarding due to the timeout based retransmission mechanisms. Therefore, we consider the first variant as a better choice than the second.

We see that due to the CSMA MAC layer, there are always a certain but low number of collisions that result in corrupted packets. Therefore, TSS already performs better than TCP for no and low bit error rates. In particular for medium and high bit error rates, the difference in packet transmissions between TSS and TCP becomes evident (Figure 6). The main reasons for the high number of packet transmissions required for TCP without running TSS in the sensor nodes are the many end to end transmissions. The optimal number of TCP segment transmissions is calculated by

$$\frac{1}{1-PER} \bullet (number\ of\ packets\ for\ PER = 0),$$

$PER = 1-(1-BER)^n$, BER: bit error rate,

PER: packet error rate.

The difference between the TCP segment transmissions using TSS and the optimal number is rather low for all bit error rates. Note that retransmissions due to collisions for calculating the optimal number of transmitted TCP segments are not considered. Considering this fact suggests that TSS performs nearly optimal for all investigated bit error rates. The performance of DTC and TSS has been compared in a previous publication using a collision free TDMA MAC layer [21]. For packet error rates below 5 % DTC and TSS have a similar number of total packet transmissions, while the total number of packet transmissions is somewhat lower for TSS in case of packet error rates above 10 %. TSS results in a lower number of TCP data segments, while the number of acknowledgements is always higher compared to DTC. This results from the aggressive acknowledgement recovery scheme implemented in TSS, while DTC does not implement such a scheme.

2) Throughput

Table 1 shows also the resulting throughput of TCP with and without TSS. Note that we did not optimize TSS for throughput, because the main goal was to keep the number of transmissions as low as possible. Nevertheless, the throughput with TSS is always significantly higher than for TCP only. For no or low bit error rates we achieve a throughput of nearly 5 kbps. Compared with the network bandwidth of 100 kbps, this is a reduction by a factor of 20. First, we have to take into account that TCP acknowledgements consume a rather high fraction of the capacity and the TCP/IP/MAC header overhead is rather high. Each payload byte causes nearly another byte to be transmitted in the header part of the TCP segment or the TCP acknowledgement. This could be improved by TCP/IP header compression. Second, packets need to be forwarded 10 times and spatial reuse is rather limited in our investigated scenario. Typically two nodes can send simultaneously. Based on these investigations, we can not expect a total throughput of more than 10 kbps. A further reduction of the throughput is caused by the delay of the CSMA MAC scheme, the occurring collisions, and TCP congestion control. We see in Table 1 that TSS has rather low throughput decrease up to the medium bit error rate, while TCP without TSS drops significantly already for medium bit error rates. For high bit error rates the packet error rate is approximately 14 % per link. In that case, nearly every packet is dropped for TCP without TSS on the path from source to destination. The TCP throughput is therefore close to 0, while TSS can at least achieve some low throughput (Figure 5). For such high bit error rates, the packet sizes could be decreased in order to decrease the packet error rate for a given bit error rate.

3) Local RTT Measurements

The local retransmission scheme deployed at the TSS nodes depends on the estimation of the round trip time between the node and the destination. The retransmission timeout is set to 1.5 * rtt, while the rtt is calculated using exponential averaging of rtt samples. To support fast convergence, we initialize the rtt value by the delay measured during a SYN/SYNACK exchange during TCP connection establishment. Figure 7 shows that the average rtt values used for retransmission timeout calculation decrease at the nodes that are closer to the destination and further away from the source. This is exactly the behaviour we need for the local retransmission and backpressure schemes as explained in Section III.A.

Figure 8, Figure 9, and Figure 10 show the rtt development at nodes 1 (close to destination), 5 (in the middle between source and destination), and node 9 (close to source) for a single simulation run (exchange of 1000 packets) and medium bit error rate. Despite a few spikes the rtt values are rather close to an average value. All simulations for rtt measurements have been performed using TSS with backpressure in the end point. The same variant has been used in the following subsections too.

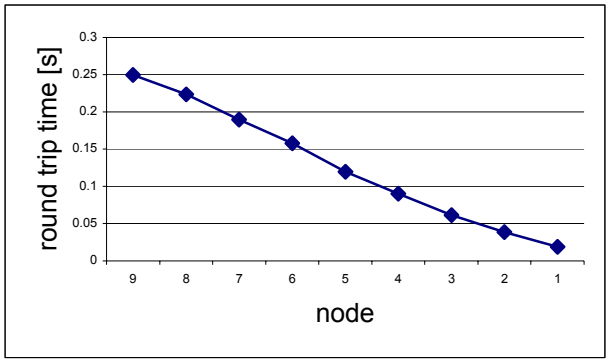

Figure 7: average round trip times per node

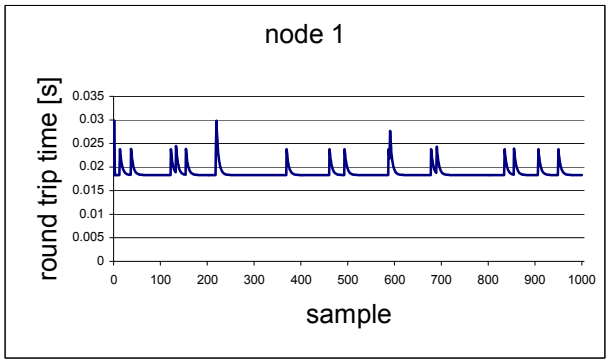

Figure 8: exponential average round trip time at nodes 1

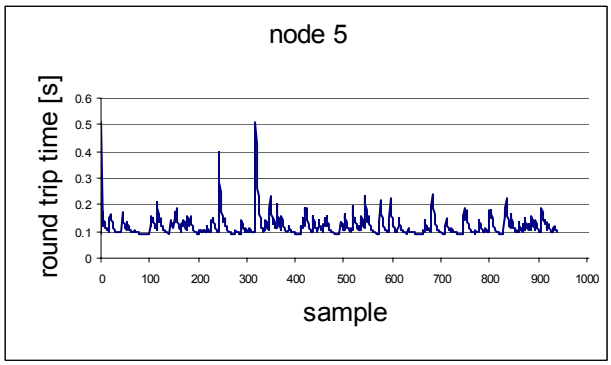

Figure 9: exponential average round trip time at node 5

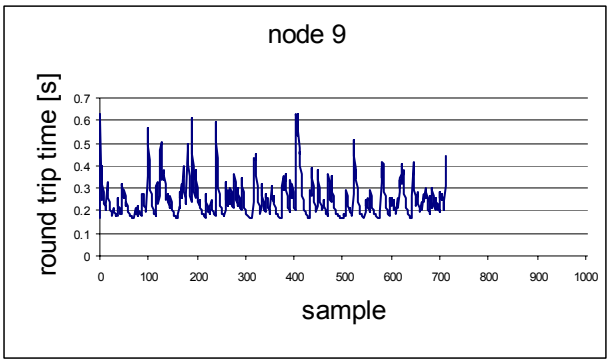

Figure 10: exponential average round trip time at node 9

4) Overhearing Time

Another issue to be investigated is the problem that has been caused by using implicit acknowledgements as discussed in Section III.A. After a node has forwarded a packet it needs to overhear its successor's transmission. This requires a node to stay in idle state and prevents it from going into any sleep state. In the worst case, a node needs to listen for the time interval for which a packet is stored in the cache. This time is limited by the retransmission timeout interval. Figure 11 measures the time a packet is stored in the retransmission buffer until the transmitted packet is either overheard or the retransmission timeout expires. These time values include at least two packet transmissions, i.e. the transmission from the first node to its successor and the transmission by the successor node. For a packet size of 1416 bits and 100 kbps link, this time must be at least 28 ms plus a small back-off time. Figure 11 plots the cumulative distribution function for these times. To get the results we performed again a single simulation run transmitting 1000 packets with medium bit error rate and measured the time values at node 5. We see that in 97 % of the cases, the packet is overheard after approximately 28 ms. However, due to packet loss and retransmission timeout expirations, the time values go up to 280 ms, but in average a node must store the packet 33.5 ms only, which is less than 20 % above the minimum value.

Link level acknowledgements may be an alternative to overhearing. However, if we assume that transmitting a bit is 50 % more costly than receiving or overhearing a bit, transmission of link level acknowledgements with 24 bytes (= 1416 bits * 20 % / 1.5 / 8) costs more energy than overhearing. This simple calculation does even not consider startup costs for turning on the transmitter in case of link level acknowledgement transmissions.

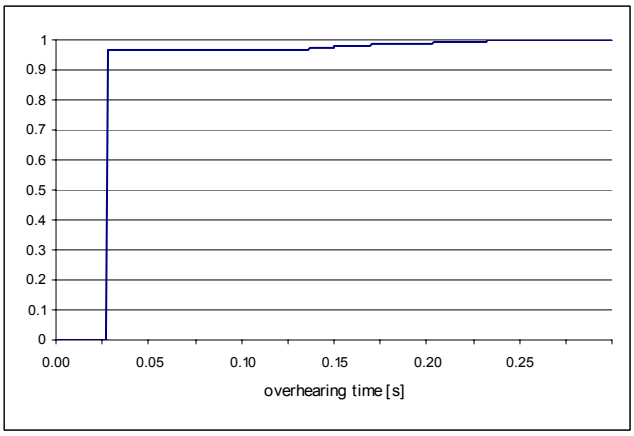

Figure 11: Overhearing time (cumulative distribution function)

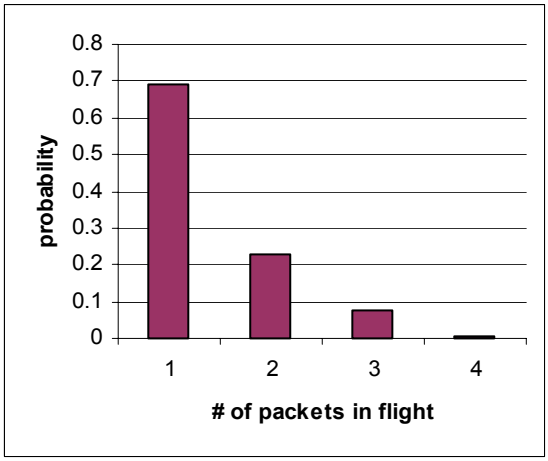

Figure 12: Number of packets in flight for 11 hops

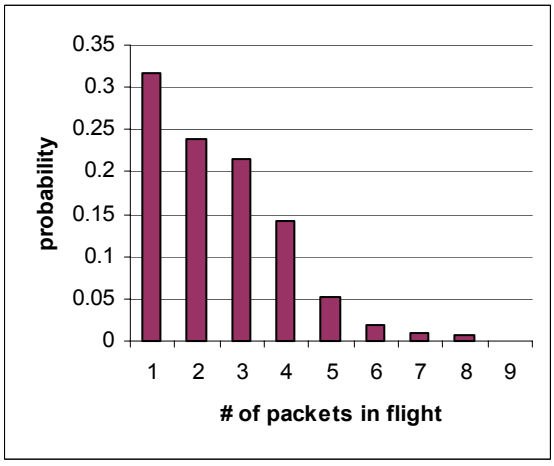

Figure 13: Number of packets in flight for 21 hops

5) Congestion Control Issues

The backpressure based congestion control limits the maximum congestion window for TSS to approximately 15 in all investigated TSS scenarios. However, the number of packets in flight is much lower. We measure the number of packets in flight after each segment has been sent by determining how many segments did not yet arrive at the receiver. Figure 12 and Figure 13show the probability for the number of packets in flight between the sender and the receiver for a scenario with 11 and 21 hops respectively. The values for the packets in flight are in most cases lower than h/4 (h = number of hops) as proposed by [10]. The average values are 1.4 and 2.5 respectively. This shows that the backpressure mechanism effectively limits the number of packets in flight to a similar number that has been proposed by other related work on congestion control in multi-hop wireless networks as discussed in Section II.B. Note that in our case, we do not have to know the number of hops between sender and receiver, but the backpressure mechanism adapts automatically to an appropriate value.

V. CONCLUSIONS

TCP support in wireless sensor networks is desirable to allow direct communication of sensor nodes with other systems for various purposes such as configuration, re-programming or management. This paper showed that even in scenarios with high error rates, TCP can be used and implemented in an energy-efficient way. This requires some protocol support in intermediate nodes that are able to store TCP segments for possible local retransmissions. The mechanisms presented in this paper drastically reduce the number of TCP segment transmissions that are needed to transfer a certain amount of data across a wireless sensor network with relatively high bit / packet error rates. Moreover, a novel congestion control mechanism has been proposed, which is effective as well as simple to implement and deploy.

Future work will analyse the performance in more complex network scenarios such as tree structures and consider background data traffic from sensors to the sink as well as multiple TCP connections originating at the sink. Additional work needs to be done for considering more complex bit error patterns [22]. We also plan to integrate scheduling mechanisms for sleep cycles and consider real implementation on available sensor node hardware. Further reduction of transmissions might be achieved by combining data and acknowledgement transmissions. Also, packet sizes need to be adapted in case of very high bit error rates. Another issue is the application of header compression in order to reduce the header overhead of TCP segments and acknowledgements. Since the TSS nodes store some state for a TCP connection, this seems to be a rather logical extension.

REFERENCES

1. A. Dunkels: Full TCP/IP for 8-bit Architectures, ACM MobiSys, pp. 85-98, San Francisco, May 2003.
2. A. Dunkels, T. Voigt, J. Alonso: Making TCP/IP Viable for Wireless Sensor Networks, Work in Progress Session at 1st European Workshop on Wireless Sensor Networks (EWSN 2004), Berlin, January 2004
3. D. Estrin, R. Govidan, J. Heidemann and S. Kumar. Next century challenges: scalable coordination in sensor networks, Mobile Computing and Networking, pp. 263-270, 1999.
4. H. Balakrishnan, S. Seshan, E. Amir, and R. H. Katz. Improving TCP/IP performance over wireless networks. ACM MobiCom, pp. 2-11, November 1995.
5. C.-Y. Wan, A. Campbell, L. Krishnamurthy: PSFQ: A Reliable Transport Protocol for Wireless Sensor Networks, 1st ACM International Workshop on Wireless Sensor Networks and Applications, Atlanta, September 28, 2002
6. A. Dunkels, T. Voigt, H. Ritter, and J. Alonso: Distributed TCP Caching for Wireless Sensor Networks. Annual Mediterranean Ad Hoc Networking Workshop, Bodrum, Turkey, June 2004.
7. F. Stamn, J. Heidemann: RMST: Reliable Data Transport in Sensor Networks, 1st IEEE International Workshop on Sensor Net Protocols and Applications, Anchorage, May 11, 2003
8. M. Heissenbüttel, T. Braun, Th. Bernoulli, and M. Waelchli: BLR: Beacon-Less Routing Algorithm for Mobile Ad-Hoc Networks, Computer Communications Journal, Elsevier, Vol. 27, No. 11, pp. 1076-1086, July 2004
9. Y. Sankarasubramanian, Ö. Ankan, I. F. Akyildiz: ESRT: Event-to-Sink Reliable Transport in Wireless Sensor Networks, ACM MobiHoc, Anaheim, June 1-3, 2003
10. Z. Fu, P. Zerfos, H. Luo, S. Lu, L. Zhang, M. Gerla: The Impact of Multihop Wireless Channel on TCP Throughput and Loss, IEEE Infocom, San Francisco, March 30 - April 3, 2003
11. C.-Y. Wan, S. Eisenman, A. Campbell: CODA: Congestion Detection and Avoidance in Sensor Networks, ACM SenSys, Los Angeles, November 3-5, 2003
12. J. Scott, G. Mapp: Link Layer-Based TCP Optimisation for Disconnecting Networks, ACM SigComm Computer Communications Review, Vol. 33, No. 5, October 2003.
13. A. Valera, W. Seah, S. Rao: Cooperative Packet Caching and Shortest Multipath Routing in Mobile Ad hoc Networks, IEEE Infocom, San Francisco, March 30 - April 3, 2003
14. D. Kim, C-K Toh and Y. Choi: TCP-BuS: Improving TCP Performance in Wireless Ad Hoc Networks, Journal of Communications and Networks, Vol. 3, No. 2 June 2001
15. Omnet++: Discrete Event Simulation System, web page, visited 2004-11-21, http://www.omnetpp.org
16. J. Schiller, A. Liers, H. Ritter, R. Winter, T. Voigt: ScatterWeb - Low Power Sensor Nodes and Energy Aware Routing, Hawaii International Conference On System Sciences (HICSS 2005), Hawaii, USA, January 2005

17. C. Enz, A. El-Hoiydi, J.-D. Decotignie, V. Peiris: WiseNET: An Ultra-Low Power Wireless Sensor Network Solution, IEEE Computer, August 2004, pp. 62
18. A. Woo, D. Culler: A Transmission Control Scheme for Media Access in Sensor Networks, ACM Mobicom, Rome, 2001
19. A. Gurtov, Sally Floyd: Modeling Wireless Links for Transport Protocols, ACM SIGCOMM CCR, Vol. 34, No. 2, April 2004
20. A. Woo, T. Tong, D. Culler: Taming the Underlying Challenges of Reliable Multihop Routing in Sensor Networks, ACM SenSys, 2003, November 5-7, 2003, Los Angeles
21. T. Braun, Th. Voigt, A. Dunkels: Energy-Efficient TCP Operation in Wireless Sensor Networks, Praxis der Informationsverarbeitung und Kommunikation (PIK), special issue on Wireless Sensor Networks, No. 2, 2005.
22. A. Köpke, A. Willig, and H. Karl: Chaotic Maps as Parsimonious Bit Error Models of Wireless Channels, IEEE INFOCOM, San Francisco, California,USA, March 2003