TCP over Wireless Ad-hoc Networks: Adaptive Snoop Protocol over Ad-hoc Networks (A-SPAN)

Ammar Tahir ammart2@illinois.edu

Daniel Kiv dkiv2@illinois.edu

GitHub: https://github.com/ammartahir24/TCPoverAdhoc

Abstract

TCP was originally designed as a reliable communication protocol for wired networks. Although it has scaled reasonably well to wireless networks, it does come with its own set of challenges and limitations. Particularly, high packet loss and bit error rate in wireless networks stops TCP from utilizing network bandwidth well enough. In this project, we looked at different strategies to improve TCP's performance: fast retransmit, SACKs and Snooping. TCP Snoop is a cross-layer mechanism where packets are cached in base-station/router and retransmitted on receival of dup-acks. Originally proposed TCP Snoop only caches packets at a single base station but in ad-hoc networks, we have multiple options. We define this problem formally and through experimentation, and then we present our Adaptive Snoop Protocol over Ad-hoc Networks (A-SPAN). We evaluate our protocol with multiple performance metrics like packet-loss, power consumption and throughput.

Introduction

Wireless Ad-hoc networks find their applications in a wide range of settings, from IoTs to robotics. Some of these applications require a reliable transport layer protocol and the default protocol used is TCP. However, TCP, a protocol originally developed for wired networks, does not work very well in a much more noisy environment natural to wireless networks. Over the years, many improvements have been proposed in the original TCP protocol to make it more efficient in lossy conditions. One of the major problems with TCP is how it perceives packet losses. The original version of TCP was not designed keeping in mind high packet loss because of the underlying wired medium. Thus, packet drop, which can easily be confused with packet loss, is used as a sign of congestion in the network. As a result, TCP not only backs off on the sending rate but it also shifts its send window backwards and ends up sending many packets again. TCP SACK helps improve throughput following a packet loss, by having the sender send only the lost packets instead of all the packets following the lost packet as well. However, the other problem of perceiving packet loss incorrectly as intentionally dropped packet is still there.

We have two strategies to solve this problem, either we can disambiguate packet loss from packet drops somehow, or we can mask packet losses to make the network appear less lossy to TCP. The first can be achieved by using Explicit Congestion Notification (ECN), Ad-hoc TCP [1] does not depend on packet losses to determine congestion in the network. Instead packet losses are treated as packet losses, i.e. these packets are simply retransmitted without changing the size of the congestion window.

The size of the congestion window is configured when packets with ECN bit on are received at the sender. ECN bit is turned on by the router when it realises queues are building up. The second scenario, i.e. making the network appear less lossy, can be achieved by caching packets at the switches/nodes. TCP Snoop [2], proposed back in 1995, does this while preserving end-end semantics of TCP. However, TCP Snoop was proposed mostly for cellular networks, where the packets from flow are cached on a base station between client device and the network. This design does not automatically extend to Wireless Ad-hoc Networks because in an ad-hoc setting we usually have a lot more choices to cache packets at.

In this project, we explored the design space experimentally and formally to answer questions like what is the ideal node in a path between the sender and receiver to snoop packet at? Should we do it at the first node since the purpose of snoop is to mask packet loss and snooping at first node can do that for the rest of the network. However, we also need to take into consideration that nodes in ad-hoc networks

are low-power. This way in case of failure, packets snooped at the first node would result in higher number of transmissions resulting in higher power consumption. Additionally these nodes also have limitations on memory, so it is only wise to find a way to spread the packets to snoop between different nodes. It is also wasteful to snoop every packet passing through the network, we should snoop only when network condition is deteriorating. We design experiments to verify these intuitions.

Based on our analytical and experimental analysis, we present the design of an Adaptive Snooping Protocol for Ad-hoc Networks. We capture the probability of a packet being dropped in a packet header field. The value of this probability is defined in terms of the reliability of links that the packet has already traversed. When the value of this probability falls below a certain user defined threshold, it is snooped at the node before being transmitted to the next hop. We discuss more details about our design later.

We evaluate our design with metrics like throughput, packet loss rate, memory consumption and number of transmissions per packet. We compare our design against our implementation of TCP Reno and TCP Sack, and show multifold improvement in throughput and packet loss. Lastly, we discuss some limitations of our design and implementation, and suggest future research directions.

Motivation

TCP famously performs very poorly on wireless networks. The reasons for its performance degradation have been widely studied and these include misinterpretation of packet loss, multipath routing and unreliability due to long paths with lossy links to name a few [1]. Particularly, since TCP is designed to process a packet drop as a signal for congestion in the network and as a result it ends up backing off on sending rate. Random packet loss is common in wireless settings and since there is no way for TCP to differentiate between a random packet loss and an intentional packet drop. As a result, TCP ends up backing off on sending rate in case of a random packet loss, even though it shouldn't really do it. Moreover, in case of packet loss, TCP not only backs off on sending rate but it also shifts back its sending window. As a result, even if only one packet was lost and many packets following it were successfully received at the receiver, TCP will end up resending all these packets again unnecessarily.

Over the years, there have been many modified versions of TCP to address its performance over wireless links. To address the latter problem of resending already received packets, we have SACK TCP [4], which acknowledges packets received out of order so those are not spuriously retransmitted. To solve the problem of misinterpretation of packet loss , we have two possible approaches: use something other than packet loss to signal congestion, or mask the packet losses in lower layers to make the network appear not as lossy. We have a number of solutions that aim to do something like the first approach. Ad-hoc TCP [1] relies on Explicit Congestion Notification (ECN) to make decisions about sending rate instead of packet loss. The ECN bit in the packet header is updated by the router node when it experiences queue build-up. Thus at the TCP sender, when packet loss is observed, the packet is simply retransmitted and there is no effect on the sending rate. TCP Vegas [3] and BBR [13] make decisions about sending rate based on their estimate of Bandwidth Delay Product. Similarly Sprout [5] uses a stochastic model to predict the bandwidth of the network in the near future. This again cuts their dependence on packet loss resulting in better performance over wireless.

For the second approach, where we aim to mask packet losses with the help of lower layers, we have TCP Snoop [2]. TCP Snoop is an improvement over Split TCP [14] since it aims to preserve

end-end semantics of TCP. In Split TCP, the original TCP connection between a sender and receiver is split between one (or more) TCP connections. This essentially breaks TCP connection over a long path with high packet loss probability into multiple TCP connections over smaller paths of comparably lower packet loss probability. There is one major flaw with Split TCP's design, if an intermediate node goes down after acknowledging a packet to the preceding connection but without successfully forwarding ahead, it breaks TCP's end-end semantic. TCP Snoop on the other hand caches packets at the routing layer in intermediate nodes. When a dupack is received and if the packet is cached, it is retransmitted and that dupack is dropped. This way the packet loss is masked from the sender.

TCP Snooping is quite an old concept, the original paper by Hari Balakrishnan came back in the late 90s [2]. However, the idea of snooping presented in the paper was only for a single hop network. The paper explores the scenario that only the last hop in the path i.e. the link between client and the base station is lossy. Thus the packets are snooped at base stations. However what happens if all the links in the path are lossy. We want to answer the question of how does one go about doing TCP Snooping on a multi-hop network. Where should the packets be snooped? How often the packets should be snooped?

We particularly focus on the first question i.e. where should the packets be snooped in a multihop network? We can make our decision based on two things, power consumption and packet losses masked. We use the number of transmissions per packet as a proxy for power consumption. Ideally, in a case where all nodes have similar link quality, snooping packets at the middle most nodes would result in the best power efficiency. Assuming that probability of packet loss is the same at all links, snooping at a node before the middle node would incur more packet transmissions when a packet is lost between snooping node and destination node. Whereas snooping at a node following the middle most node means the probability of packet loss becomes higher before the snooping node.

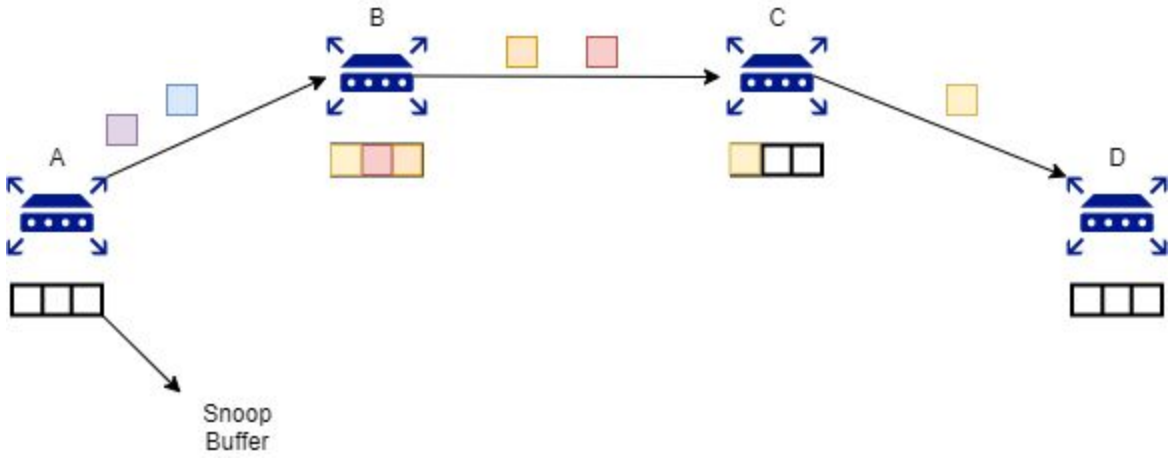

Figure 1: *Node B and C are snooping packets for a flow from node A to node D*

We devised an experiment to verify these intuitions. Our setup consists of 4 Raspberry Pi's connected via an ad-hoc network and by using a static routing table, they are configured to have a topology similar to one shown in Figure 1. 6,000 MTU sized packets are sent from node A to D and we see the performance gains of snooping at node B and C. Due to the close proximity of Raspberry Pi's the packet loss was observed to be very small so we added an artificial packet loss rate of 0.5% at each

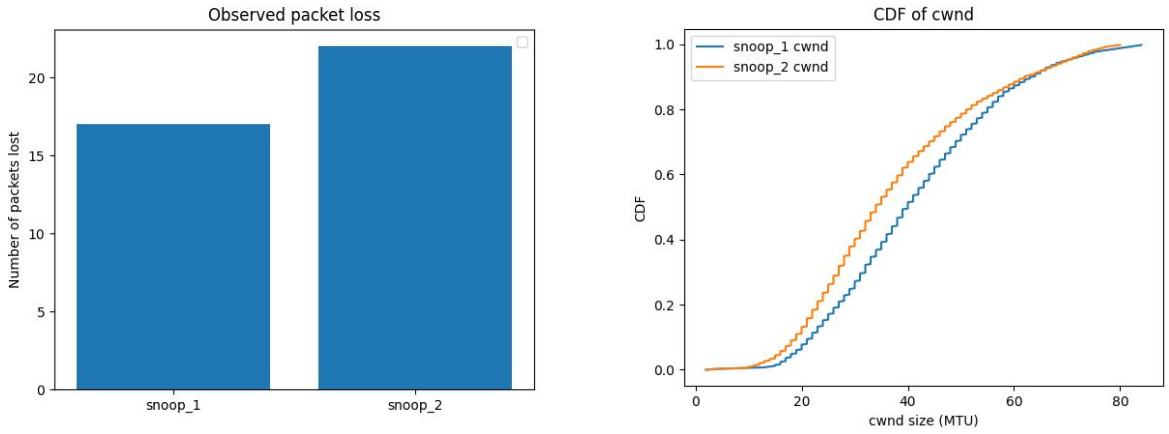

Figure 2: *Comparison of snooping at a node B (snoop_1) compared to a later node C in a 4 node static route (snoop_2).* ***(a)*** *Observed packet loss at the sender.* ***(b)*** *Congestion window size at the sender*

routing node. This means the link quality of each hop is almost similar. We measure the observed packet loss at sender, congestion window size as a proxy for throughput and lastly the number of transmissions done for each packet on the whole network.

We expect to see lower packet loss and ultimately higher sending rate for node B and lower number of transmissions per packet for node C. Our experiment results are consistent with our hypothesis. Figure 2(a) shows lower packet loss for when packets were snooped earlier (snoop_1) i.e. on node B and this naturally translates to higher congestion window since TCP only backs off on congestion window on detection of packet drops. Figure 3 shows the number of transmissions per packet in both the approaches, here we can see that when we snoop later we get fewer numbers of transmissions per

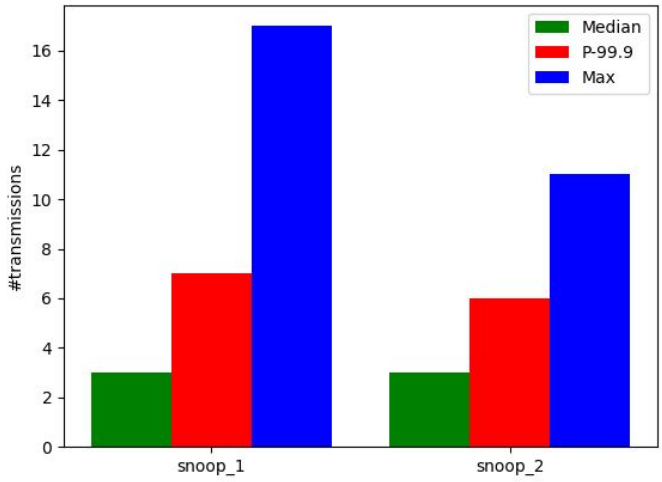

Figure 3: *Number of transmissions per packet for different snooping positions*

packet. The reason we see the difference only in higher percentiles and not in the median case is because packet loss in our experiment is rather rare and we get to retransmit only on packet loss. Thus in the normal case we have 3 transmissions per packet, equal to the total number of hops in our topology.

One last observation we make is that the hits at snoop caches in case of packet loss is 76 and 44 for when packets were snooped at node B and C respectively whereas the most number of packets cached at a point were 338 and 349 respectively. Devices constituting ad-hoc networks are usually equipped with sparse memory, so it is not wise to cache every packet that passes through the network since at some point we will start getting diminishing returns. It makes sense to cache packets only when the network conditions are bad and packet losses are more probable.

Approach

At the start of this project we had two options; either to work on finding better signals than packet loss for congestion control like in Ad-hoc TCP and TCP Vegas or advancing TCP Snoop to multi-hop wireless networks. For the first option, we considered implementing ECN in the routing layer. We even implemented the WRED algorithm for queue size estimation in our routing layer. However, we soon realized that ECN is not very efficient when it comes to wireless ad-hoc networks because queue sizes are typically small and the WRED algorithm does not capture congestion very efficiently. Moreover, there are plenty of other solutions to not using packet loss as an indicator for congestion, as discussed in the previous section. There hasn't been enough work on extending TCP Snoop to a multihop wireless environment though. We also think that a lot of applications can benefit from a multihop TCP Snoop, from IoTs to mobile ad-hoc networks. It does not require advanced algorithms to be run at the routing layer either giving it a much desired simplicity.

For the design of multihop TCP Snoop, we formulated a bunch of questions concerning requirements from our design. Based on the analysis we discussed in the previous section, we knew we wanted an adaptive protocol which aims to distribute packets to snoop between different nodes. We also knew we wanted our solution to not cache every packet but cache packets only when network conditions deteriorate. This meant we needed to measure the link quality of each hop. If we know the link quality and we can translate it to the probability of a successful transmission, we can calculate the probability of successful transmission of the packet over the complete path. Let p_i be the probability of successful transmission across i^{th} hop, then probability of successful transmission across the whole path, P, is given by:

$$P = p_1 p_2 \dots p_n$$

If this probability falls below a certain threshold value at any point, we should cache the packet. For example let's say the threshold value is 0.95 and for the 3rd hop the probability falls below 0.95, i.e. $p_1 p_2 p_3 < 0.95$. Thus we should snoop the packet before sending over the 3rd hop at the 3rd node.

The next design choice we needed to make was about where to make the decision about where to snoop? Essentially should we decide which node to snoop at, on the end host or in network? To do it at the end host, we will need the link quality information about each hop in the network. This way the sender can decide at the end host where they want to snoop each packet. The benefit of this approach is that it gives great control to the sender, they can choose to snoop some packets earlier and others later depending on the packet's QoS requirements. For example if a packet is delay sensitive, the sender could

decide to snoop that packet more often to ensure faster retransmit. The drawback of this approach is that firstly it requires fields in the packet header, the size of which increases depending on the number of hops in the path. Secondly we wanted our design to be easily extensible to dynamic topologies. Having to know all the nodes in path in advance limits us to only static route topologies. Therefore, we shifted the logic to decide which node to snoop packet at, in the network.

Another important decision was about deciding a metric for link quality. We initially decided to go with RSSI but it is ineffective against interference fluctuations [12]. On the other hand ETX is not only very effective, it is easier to implement as well as it does not rely on lower layer functionalities or specific hardware [11]. We implement a slightly modified version of ETX, details of which are discussed in the next section.

Design

We call our design Adaptive Snooping Protocol for Ad-hoc Networks (A-SPAN) and it builds on top of TCP Sack. Below we discuss major components of our design:

ETX Algorithm for Link Quality. Each node in our ad-hoc network measures link quality with its neighbours by measuring the number of probe packets it receives in a time period. When a node observes its neighbour is up, it starts sending probe packets periodically which have monotonically increasing sequence numbers. At the same time, each node also spins a listener thread, which listens for incoming probe packets. If a node does not receive a packet or receives some packet out of order it counts the missed packets as lost. Every node maintains a window of 1000 where it records how many packets were received and how many were lost. The number of packets received divided by 1000 is the link quality of that hop. This algorithm is a little different from the original ETX algorithm where the quality of the hop is equal to the product of packet ratio of the node multiplied by the product of packet ratio for the probe packet sender node. The reason for this is that ETX algorithm assumes that a successful transmission constitutes successful packet delivery followed by successfully receiving acknowledgement. Since in our case, we simply use UDP packets to send data and do not depend on acknowledgement, we do not incorporate it in our ETX algorithm.

Packet Header Fields for A-SNAP. We add the following packet header fields in our packets for A-SNAP to function.

- *Pkt_effort:* This field captures the probability of successful transmission, *P* discussed earlier, for the hops that packet has travelled so far. pkt_effort signifies the estimated effort it took for a packet to arrive at the given node, it indirectly carries information about how many hops it has taken and also how unreliable the previous links have been. It essentially tells the routing node that if the packet is not snooped at that node and is consequently dropped, how much effort will it take for the packet to arrive at the current node upon retransmission.
- *Pkt_qos:* This field is set by the end user and it is basically the threshold value after which the packet is snooped. We keep this field to give sender control over what kind of QoS they want for their packets or flows. If they want lower delay they can set pkt_qos to a higher value which will result in the packet being snooped more often. If they are more concerned about saving memory of the nodes and delay is not that important, pkt_qos can be set to a higher number.
- *Snoop_after:* This field is set by the sender and it can be used to explicitly skip the first few nodes as the sender desires. We do not make use of this field in our measurements but it can be useful for some cases.

- *Snoop_limit:* If the sender wants to limit the number of times a packet can be snooped, this field can be set.

Routing Nodes on Receiving Data Packets. When the routing nodes receive data packets, they do following things:

1. Update the value of *pkt_effort* by multiplying it with the ETX value of the next hop.
2. If the new value of *pkt_effort* falls below *pkt_qos*, the *pkt_effort* field is updated to 1 and the packet's copy is cached. The pkt_effort field is again updated by multiplying it with the ETX of next hop before transmitting.

Routing Nodes on Receiving Ack Packets. For each flow, the routing nodes maintain the last ACK they have seen for it. When a new ack packet arrives, the node does the following:

1. If the acknowledgement number is greater than the last acknowledgement number, remove all the data packets uptil that acknowledgement number from the snoop cache. Update the last acknowledgment number's value with the newer acknowledgement.
2. If the acknowledgement number is equal to the last acknowledgement number and if the packet corresponding to it is available in the cache, retransmit it and drop the ACK packet.

Implementation

This project is written completely in about 1500 lines of Python 3 code, which is available on our Github. Details about how to run specific experiments are available on the repository's readme. In addition, the code is capable of logging data during test runs and plotting the results on a variety of graphs, allowing for easy experimental comparisons.

We first establish an ad-hoc network four Raspberry Pi that will be used in the experiments. This can be done by configuring the Pi's network interfaces and setting the wireless network interface to ad-hoc mode. A unique static IP address is defined for each Pi. For our project we configured them to be 10.2.1.1, 10.2.1.2, 10.2.1.3, and 10.2.1.4. These Pi's are able to ping each other directly within the ad-hoc network. Then, we built IP forwarding capability on top of Python's UDP sockets, and packet generation code. In the forwarding layer we use static routes. While writing code for the routing layer, we also took into account the support we will need from this layer for snooping. We verified this implementation by writing a UDP like application.

Next, we proceeded to build TCP on top of this routing layer. We started by implementing an acknowledgement mechanism, followed by receiver window based flow control and finally congestion control. We implemented TCP Reno since it is the most popular congestion control algorithm. Slow start phase starts in case of an RTO, whereas in case of 3 dup-acks, ssthresh is decreased and we go directly into congestion avoidance phase. Lastly, connection setup consists of three-way handshake, and teardown also happens gracefully after exchanging fin packets.

We have also implemented SACK functionality. Upon receiving a packet with higher sequence number than expected, receiver buffers the packet instead of discarding it and sends back a dup-ack but containing a SACK for all the buffered packets. This way the sender does not retransmit the SACKed packets. TCP SACK is compared as a baseline against our implementation of A-SPAN, which also has the ability to do SACK.

The most important part of our project is adaptive snooping protocol. We have a snooping mechanism enabled in the routing layer i.e. based on packet header fields, IP layer can buffer a packet and retransmit it on seeing a dupack for it. Since we have a static route, the packet is removed from the buffer only when an ack with a larger number is seen. Each switch/routing entity keeps track of its link quality with neighbouring nodes, this is done by running an ETX measurement algorithm [11]. The value for ETX is normalised between 0 and 1, where higher value means higher link reliability. A value of 1 means perfect link. The calculation of the ETX value is slightly different from what is reported in the paper, since the ETX packet is sent via UDP and does not receive an ACK for that sent packet.

Each data packet contains an IP header field pkt_effort, which is initiated by the sender to 1. Before the packet is forwarded by any router it updates the pkt_effort field to pkt_effort * ETX. If the new value of pkt_effort falls below a threshold value, the packet is snooped and pkt_effort's value is set to 1. In our setup, the ETX packet is sent to a neighboring node almost every tenth of a second and has a buffer size of about 1000 to calculate the ETX value. That is, if the ETX packet arrives at a neighboring node with the timeout period, it is considered a successful packet and is marked as received. If all ETX packets are received within the timeout period, then the ETX value would be a 1. If 50% of the ETX packets make the timeout period, then the ETX value would be 0.5, indicating a weaker link, which A-SNAP will recognize if it's lower than another possible snoop node.

Evaluation and Results

We evaluate our design with respect to three main metrics: throughput captured via the sending rate, observed packet losses, number of transmissions per packet as a proxy for power consumption, and the memory consumed in snooping. Packet loss is recorded at the sender for whenever three dupacks are received or a RTO happens. Similarly the sending rate in the form of congestion window is also logged from the sender size. We log it everytime we update it i.e. after completion of each roundtrip or packet loss. The information about the number of transmissions per packet is logged separately at each of the nodes and aggregated after the completion of the experiment to find the cumulative number of transmissions per packet in our ad-hoc network. Lastly, memory consumed and snoop cache's hits are logged at each of the routing nodes.

We use packet losses and cwnd size to measure the performance gains of our design. We particularly use cwnd size for measurement of throughput as it is a much more nuanced and noise free metric as compared to receiver side throughput. The cwnd size is at granularity of RTTs and thus it is able to capture a lot more detail than say receiver side throughput. The number of transmissions per packet is another important metric that evaluates effectiveness of our design in power constrained wireless ad-hoc networks. Lastly, we use snoop memory and snoop hits to quantify the cost of our algorithm. Especially because the devices in wireless ad hoc networks tend to be limited on the memory.

The first results we present are for the experiment settings we discuss in the previous section. We connected four RPi's via a static route as shown in Figure 1, and sent data from node A to node D. Thus our algorithm can adaptively snoop at node B or C. We set the pkt_qos equal to 0.99 for this experiment. Since we had limited space in our apartment, we had to place RPi's in close proximity to each other and thus we observed random packet loss to be very rare. Thus, the performance of our approach is comparable to our baselines i.e. TCP Reno and SACK TCP. Figure 4(a) shows the observed

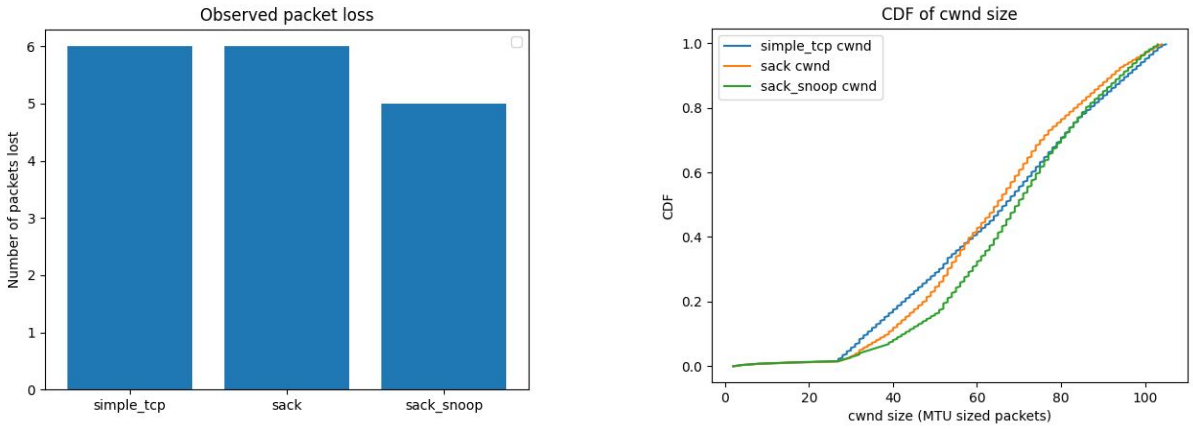

Figure 4: *Comparison of TCP Reno, TCP SACK, and A-SPAN.* ***(a)*** *Observed packet loss at the sender.* ***(b)*** *Congestion window size at the sender.*

packet loss at the sender and Figure 4(b) shows the CDF of congestion window's size. Our approach either observed one less packet loss or masked a random packet loss which results in slightly higher cwnd size as well.

Thus, to measure the gains due to our approach, we induced artificial packet loss at a rate of 0.5% in each of the nodes. Which means the listener at each of the nodes would randomly drop a packet with a probability of 0.5%. The rest of the results we discuss in this paper have this artificial loss. Let's revisit the above experiment with this new setting.

Figure 5 shows the packet loss and cwnd size for this experiment. This time we can clearly see the benefits that we get as a result of snooping. The packet losses are decreased by almost 3-4x and the resulting increase in congestion window is significantly higher as well. In the median case, we have almost 2.5x improvement over TCP Reno and TCP SACK. The reason we still see a small amount of packet losses in A-SPAN's case is that firstly pkt_qos was set to 0.99 which means a few packets may have been missed. But this is not the likely reason, in fact, it's by choice that we let a few packet drops/losses happen. Notice that in our topology, shown in Figure 1, when we snoop at node B, the packet drops that happen on the link A-B are never masked by A-SPAN. This gives room for routing node B to drop packets in case of congestion to signal congestion to the sender. We discuss this further later on when we argue against perfect snoop.

As we see in Figure 6, A-SPAN has one limitation compared to TCP Reno and SACK TCP. In the worst case, it ends up having more number of transmissions per packet. We investigated this further to find out that it is due to a general limitation in TCP Snoop's design. Typically when a packet is lost, the receiver ends up sending multiple dupacks instead of a few ones. The routing node has no way of knowing whether the packet, it retransmitted, has successfully reached the receiver. Thus it has to resend packets for every dupack or at least after a few number of dupacks. In TCP Snoop's case, since the base station did not have power saving requirements, it was not that big of a problem. But in case of wireless ad-hoc networks, these extra transmissions can be significant. This needs further investigation in the future.

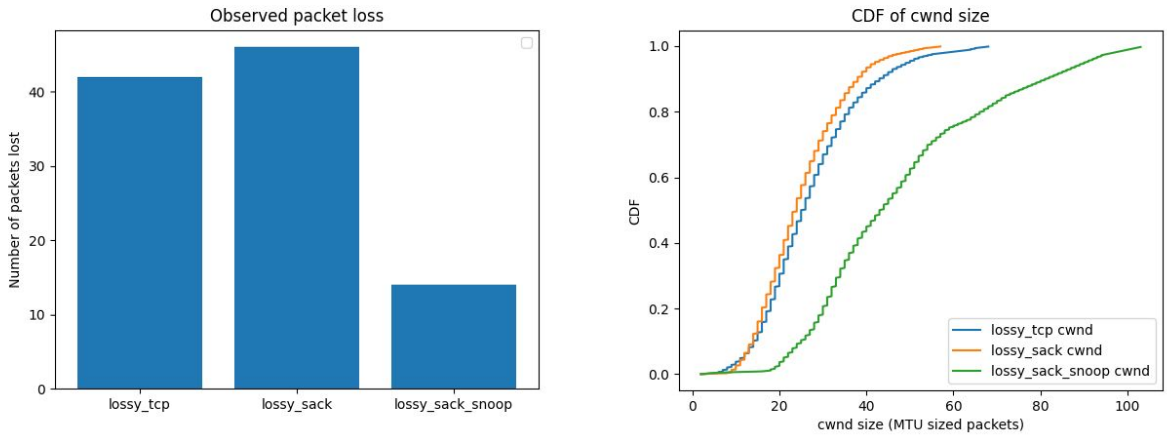

Figure 5: *Comparison of TCP Reno, TCP SACK, and A-SPAN over a network with loss rate of 0.5% per node.* ***(a)*** *Observed packet loss at the sender.* ***(b)*** *Congestion window size at the sender.*

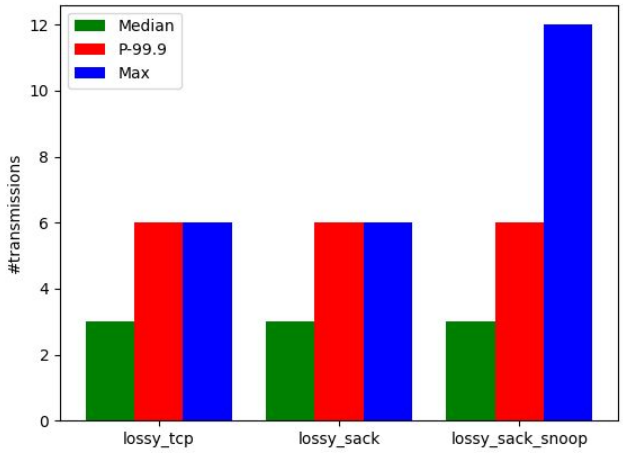

Figure 6*: A-SPAN has a higher number of transmissions per packet in the worst case than TCP Reno and TCP SACK.*

The value of *pkt_qos*

Next we evaluate our design with varying values of *pkt_qos* and show what impact this has on different metrics. We keep everything the same in our experiments i.e. loss rate is 0.5% per node, flow size is 16,000 MTU packets and TCP metadata like receive buffer size are all the same. Then we vary the value of *pkt_qos* from 0.90 to 1 and see its effect on throughput, packet loss, number of transmissions and memory consumed.

We observe that packet loss is substantially lower for *pkt_qos* values of 1.00 and 0.99, which ultimately translates in a higher congestion window. Again, the reason we still have packet loss despite *pkt_qos* being 1 is that since we start snooping at the first node after the sender, there is still a chance of loss happening at the first link. This first link also regulates congestion control by dropping the packets when

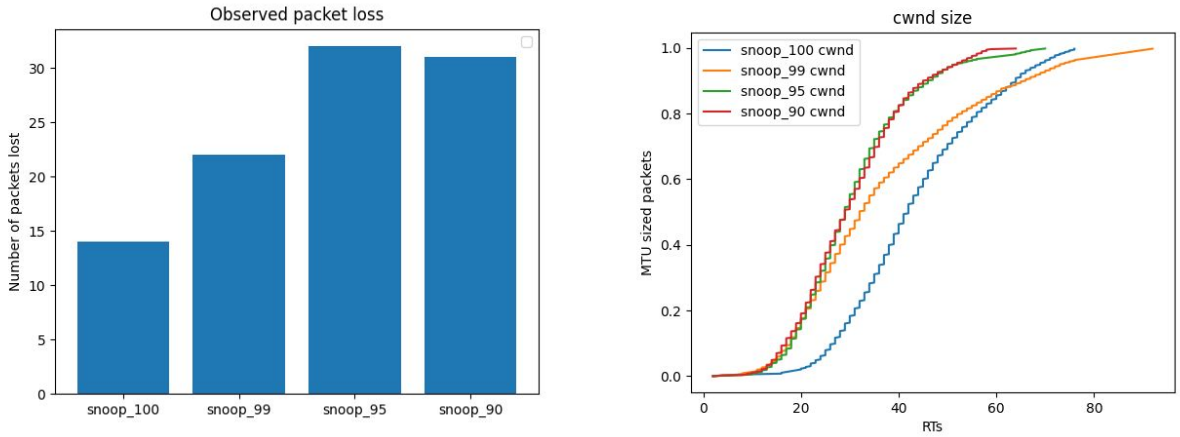

Figure 7: *Different values of pkt_qos (1.00, 0.99, 0.95, 0.9).* ***(a)*** *Observed packet loss at the sender.* ***(b)*** *Congestion window size at the sender.*

queues build up through drop tail policy. If packets are dropped at some later node, the signal of congestion will never reach the sender because some intermediate node will end up retransmitting the packet.

While the *pkt_qos* value of 1 has slightly better performance compared to 0.99, it comes at the cost of higher memory consumption (Figure 8) and higher number of transmissions per packet (Figure 9). If we put all these results in perspective, we can see how *pkt_qos* gives us an all-round good performance. It has packet loss and sending rate comparable to that of *pkt_qos* of 1 but it does so while utilizing much smaller amounts of memory and fewer transmissions per packet. The reason we see good performance at 0.99 is because this value aims to snoop packets closer to the middle of the path. Recall that we had a loss rate of 0.005 per node, which translates to $P = (0.995)^3 = 0.985$, where P is the probability of successful transmission across the complete path.

This motivated us to find a way to learn the optimal value of *pkt_qos* by learning the value of P over time. However, we would still want to give control to the end user on how they want to use the learnt value of P to assign *pkt_qos* values. This is because as we discussed, different packets or flows could have different QoS requirements.

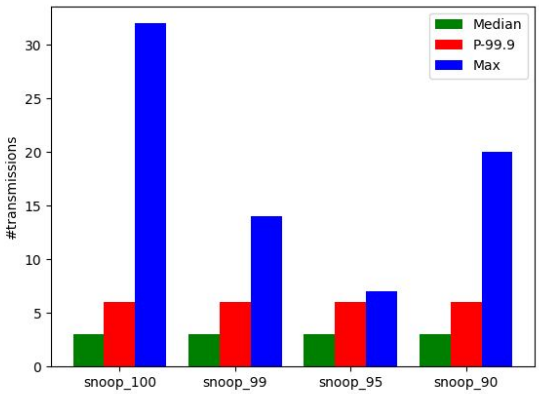

Figure 8: *Higher number of transmissions per packet in the worst case for pkt_qos of 1.00*

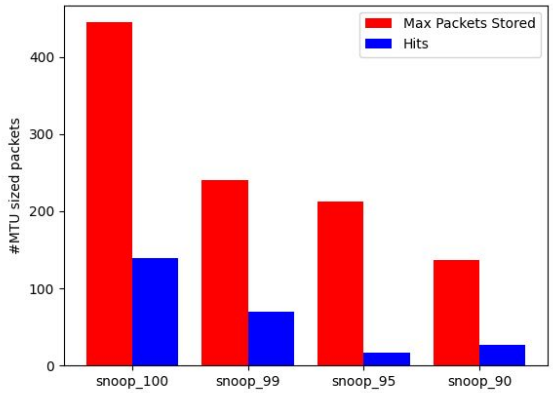

Figure 9: *Higher memory consumption for pkt_qos of 1.00*

Why not the perfect snoop?

During our presentation, someone raised the question of why don't we go for a perfect snoop that masks every packet loss from the network. This essentially means that we snoop packets at the routing layer on the sending node as well. As we discussed, the only loss not masked by the earlier discussed version of perfect snoop (*pkt_qos = 1*) was the loss that happens at the first link. We can mask this too by also snooping the packets at the routing layer on the sending node. This way when a dupack arrives, the routing layer retransmits the snooped packet from snoop cache and does not pass dupack up to the transport layer. This gives us a practically perfect snoop but now we ask the question do we really need a perfect snoop? Recall that the purpose of doing snooping was to mask packet losses from TCP, so when the TCP sender observes packet loss, it can be sure that it was an intentional packet drop to signal condition. By doing a perfect snoop, the TCP sender never gets the signal for congestion and it keeps increasing the congestion window. Until a point comes when the sending rate is way above what router queues can sustain and most of the packets sent are being dropped. The receiver receives a lot of packets out of order and it keeps sending dupacks which are entertained by snoop caches. The newer packets are already congesting the queues, so retransmitted packets are being dropped again and again. This results in an explosion of transmissions per packet and an ultimate meltdown of TCP connection where the transmission of newer packets is halted by older packets and the sender starts experiencing timeouts.

We implemented this setting and tested it out as well. Although we did not observe timeouts, we noticed a massive increase in the number of transmissions per packet. As cwnd keeps increasing, timeouts are inevitable as well but the bigger concern here is the number of transmission per packet being in order of 100s. Based on this we argue that even for doing a perfect snoop, it is good to have first link's packet losses unmasked for regulation of the congestion window. We pay a small price in terms of packet losses at first link for efficient utilization of other resources. These results are shown in Figure 10.

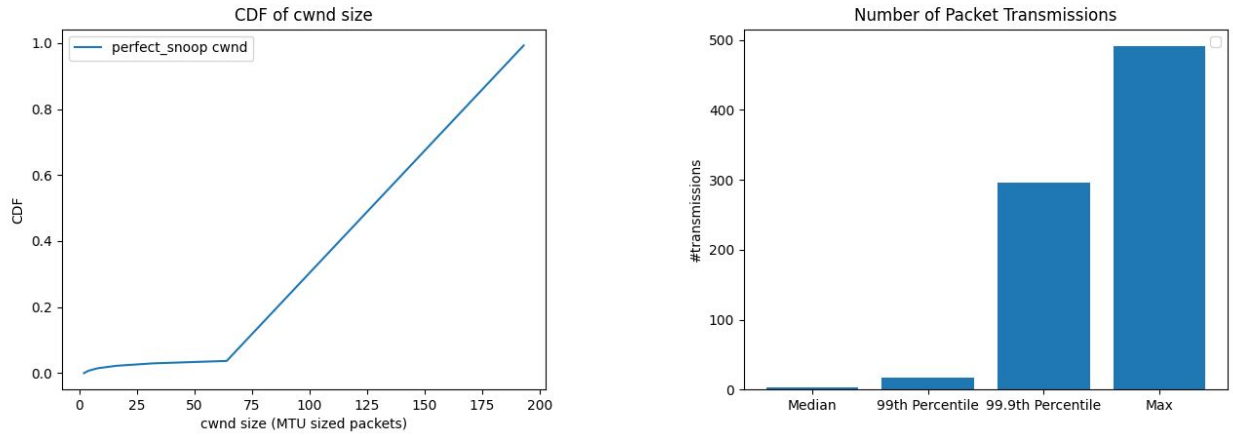

Figure 10: *Perfect snoop and its cost. (a) congestion window is constantly increasing since packet loss is never observed. (b) transmissions per packet increasing as queues are congested*

Future Work

Following are a few research directions to extend our work:

Configuration of *pkt_qos*
As observed in our evaluation, our algorithm gives the best performance when the value of *pkt_qos* is configured to be close to the probability of successful transmission across the complete path. An extension to this project could be to learn this value based on feedback from the network. We are actually working on this already. We have a weighted learning algorithm which gets the probability of successful transmission from the network in an ack packet. The sender does a weighted average with the previous value of probability to get newer value. We are still pruning this algorithm for efficiency and looking for baselines to compare it against.

Extending to Dynamic Topologies
Currently we have tested our implementation on a static route but we made many design choices keeping in mind easier extension to dynamic topologies. For example, calculation of *pkt_effort* happens in the network which eliminates the need for requiring the path in advance, something not feasible for dynamic topologies. One challenge that we will have in dynamic routing is that acks can go on a different route than the one followed by data packets. This not only means we will not get snoop hits but also we will not know when to evict packets from snoop buffer. One workaround could be to use source routing to force the ack packets to follow the path traversed by data packets. This area needs more research.

Conclusion

We used formal and experimental analysis to study the case of extending TCP Snoop to a multihop wireless ad-hoc network. Our analysis showed that we could get different benefits by snooping packets at

nodes on different positions in topologies. Snooping packets earlier gives us lower packet loss whereas snooping later is optimal with respect to number of transmissions per packet. This motivated us to design an adaptive algorithm to decide where to snoop packets based on the link quality of hops in the network. Adaptive Snoop Protocol for Ad-hoc Networks (A-SPAN) can be used to select optimal nodes to snoop packets based on expected transmission count. A-SPAN can be used in environments where multiple connections in the route are weak and lossy. Our setup involved a static route multi-hop ad-hoc network of 4-Raspberry Pi in a small apartment living room. Results from our experiment with A-SPAN were compared against TCP Reno and TCP SACK, with our adaptive snoop algorithm having notable performance gains in a packet loss multi-hop environment.

Bibliography

[1] B.S. Manoj and C. Siva Ram Murthy. “Transport Layer and Security Protocols for Ad Hoc Wireless Networks.” 2005. https://www.informit.com/articles/article.aspx?p=361984&seqNum=5

[2] Balakrishnan, Hari, Srinivasan Seshan, Elan Amir, and Randy H. Katz. "Improving TCP/IP performance over wireless networks." In *Proceedings of the 1st annual international conference on Mobile computing and networking*, pp. 2-11. 1995.

[3] L. S. Brakmo and L. L. Peterson, "TCP Vegas: end to end congestion avoidance on a global Internet," in *IEEE Journal on Selected Areas in Communications*, vol. 13, no. 8, pp. 1465-1480, Oct. 1995, doi: 10.1109/49.464716.

[4] S. Waghmare, P. Nikose, A. Parab and S. J. Bhosale, "Comparative analysis of different TCP variants in a wireless environment," *2011 3rd International Conference on Electronics Computer Technology*, Kanyakumari, 2011, pp. 158-162, doi: 10.1109/ICECTECH.2011.5941878.

[5] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. 2013. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation* (*nsdi'13*). USENIX Association, USA, 459–472.

[6] Purdue, E-Pubs & Fahmy, Sonia & Prabhakar, Venkatesh & Avasarafa, Srinivas & Younis, Ossama. (2003). TCP over Wireless Links: Mechanisms and Implications. Computer Science Technical Reports. Paper.

[7] Jianzhen Sun, Yuan'an Liu, Hefei Hu and Dongming Yuan, "Link stability based routing in mobile ad hoc networks," *2010 5th IEEE Conference on Industrial Electronics and Applications*, Taichung, 2010, pp. 1821-1825, doi: 10.1109/ICIEA.2010.5515377.

[8] R. Dube, C. D. Rais, Kuang-Yeh Wang and S. K. Tripathi, "Signal stability-based adaptive routing (SSA) for ad hoc mobile networks," in *IEEE Personal Communications*, vol. 4, no. 1, pp. 36-45, Feb. 1997, doi: 10.1109/98.575990.

[9] Srinivasan, Kannan, and Philip Levis. "RSSI is under appreciated." *Proceedings of the third workshop on embedded networked sensors (EmNets)*. Vol. 2006. 2006.

[10] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center TCP (DCTCP). *SIGCOMM Comput. Commun.* Rev. 40, 4 (October 2010), 63–74. DOI:https://doi.org/10.1145/1851275.1851192

[11] Richard Draves, Jitendra Padhye, and Brian Zill. 2004. Comparison of routing metrics for static multi-hop wireless networks. In *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '04). Association for Computing Machinery*, New York, NY, USA, 133–144. DOI:https://doi.org/10.1145/1015467.1015483

[12] A. Vlavianos, L. K. Law, I. Broustis, S. V. Krishnamurthy and M. Faloutsos, "Assessing link quality in IEEE 802.11 Wireless Networks: Which is the right metric?," *2008 IEEE 19th International Symposium on*

Personal, Indoor and Mobile Radio Communications, Cannes, 2008, pp. 1-6, doi: 10.1109/PIMRC.2008.4699837.

[13] Cardwell, Neal & Cheng, Yuchung & Gunn, C. & Yeganeh, Soheil & Jacobson, Van. (2017). BBR: Congestion-based congestion control. Communications of the ACM. 60. 58-66. 10.1145/3009824.

[14] Hari Balakrishnan, Venkata N. Padmanabhan, Srinivasan Seshan, and Randy H. Katz. 1996. A comparison of mechanisms for improving TCP performance over wireless links. *SIGCOMM Comput. Commun.* Rev. 26, 4 (Oct. 1996), 256–269. DOI:https://doi.org/10.1145/248157.248179