1. INTRODUCTION

The performance and fairness problems of TCP in 802.11 multi-hop wireless mesh networks (WMN) are well known [1, 2, 3]. Approaches to solve them fall roughly into two different categories: (i) those which maintain legacy TCP compatibility, and (ii) clean-slate protocols which fundamentally redefine WMN transport, breaking TCP compatibility.

Clean-slate approaches such as block-switched and cross-layer transport protocols (e.g. HOP [4] and ATP [5]) break the end-to-end principle, utilizing intermediate WMN nodes for intelligent caching and pacing, demonstrating significant throughput and flow fairness improvements. Clean-slate approaches however, exclude TCP-speaking clients from the network and as such, their practical impact as well as real-world network evaluation potential is severely limited.

We believe that the strengths of these two ostensibly opposed categories may in-fact be exploited by uniting them. We propose TCPSpeaker, a transport-protocol translator to “bridge” TCP flows entering and leaving the WMN at its edges. TCPSpeaker differs from other split-TCP approaches [3, 6], as it does not merely “split” TCP flows into multiple segments, but rather, it enables the removal and replacement of TCP entirely from within the WMN.

To the mesh network users, TCPSpeaker presents a transparent, fully TCP-compliant interface. Inside the WMN, TCPSpeaker introduces the freedom to utilize a new transport protocol more suited to the broadcast nature of WMN forwarding (e.g. supporting opportunistic forwarding).

TCPSpeaker may further prove useful beyond the scope of mesh networks, allowing any clean-slate transport to interact with legacy TCP end-hosts. We implement TCPSpeaker as a fully-functional Click Element [7] and evaluate its performance and behavior as an interface to TCP. We next discuss our design, implementation, and evaluation, identifying open questions and directions for further work.

2. DESIGN AND IMPLEMENTATION

The primary design goal of TCPSpeaker is to act as an interface between TCP and a semantically equivalent intra-WMN transport protocol, which we will call L4. For the purposes of TCPSpeaker design, we consider a protocol to be semantically equivalent to TCP if it exhibits the same properties of TCP: bi-directional, connection-oriented, in-order, reliable bytestream with multiplexing, and flow-control. TCPSpeaker is operated in conjunction with its semantically equivalent partner, with which it is paired directly back-to-back on the same wireless node instance. Two separate back-to-back pairs are operated at each respective edge of the WMN where TCP traffic enters or exits, as illustrated in Figure 1.

Figure 1 depicts TCP traffic entering the WMN, where it is intercepted by a TCPSpeaker (1) and then passed to an L4 translator (2), both translators running back-to-back on the same WMN node. The traffic then moves through the WMN until it is intercepted at the gateway, where it is converted from L4 (3) back into TCP (4), at which point it leaves the WMN for its final destination. For evaluation and development purposes, TCPSpeaker may be paired back-to-back with another TCPSpeaker, effectively translating from TCP to TCP in the fashion of a split-TCP proxy.

2.1 Flow Dispatching and Handling

TCPSpeaker handles TCP traffic on a per-flow basis, where a flow is made up of TCP segments belonging to the same bi-directional TCP connection sharing the same entry and exit nodes of the wireless network. This design feature is crucial for preserving the connection-oriented semantics of TCP on both edges of the WMN. To achieve per-flow handling, TCPSpeaker features a layered architecture given in Figure 2. As a TCP flow is intercepted by one of the paired TCPSpeakers, it is dispatched to its appropriate flow-handler. The flow-handler realizes the actual protocol translation process by translating the incoming stateful TCP flow to a stateless intermediate bytestream. This stateless bytestream is fed directly to the paired L4 (or TCPSpeaker) protocol translator where its respective intra-WMN state is kept. This stateful-to-stateless-to-stateful transition is realized in our implementation via a zero-copy, intra-process communication, safeguarding the stateless bytestream in all but the most extreme scenarios of total node failure.

To realize the communication of per-flow TCP-to-L4 se-
manics (for purposes of flow-control) TCPSpeaker provides an interface for throttling the rate at which data is passed to its paired L4 neighbor and vice-versa. The throttling is realized by inspection of the free space in its neighbor’s OUT queue. A TCPSpeaker will only push data to its neighbor when its neighbor is able to send that data itself. This realizes the flow-control mechanism of TCP via a backpressure effect.

2.2 Preserving Macroscopic TCP Behavior

In order to preserve outward TCP compatibility, the TCPSpeaker takes special care to preserve the macroscopic behavioral attributes of TCP, preserving end-to-end flow control and outward in-order, reliable transport robust against packet loss and reordering. TCPSpeaker also takes special care to ensure throughput fairness across multiple flows under its control.

In the event of total node failure, a curious scenario arises, as some data which has been ACKed to the sender as having been received may fail to reach its final destination. We believe that in practice, this limitation should prove to be manageable. The backpressure mechanism limits the amount of data which could be potentially lost in this scenario to the bandwidth-delay product of the WMN. Additionally, an ACK only means that the segment was received by the TCP subsystem—it does not guarantee processing by the application. The delivery guarantee issue of the TCPSpeaker under this pathological condition deserves investigation and poses an important question for future investigation.

2.3 TCPSpeaker as a Click Element

We choose to implement TCPSpeaker as an Element of the Click Modular Router, a modular C++ packet processing framework, which affords us exceptional flexibility in development, evaluation, and deployment. Our Click TCPSpeaker implementation allows us to run the same code-base on any system capable of running Linux (among other platforms), in either user-space, kernel-space, or even within an ns-2 simulation environment, with no code-level modifications required. With only minor modifications to ensure memory position-independent code, we run and evaluate TCPSpeaker on the Gateworks Avila (an ARM architecture embedded device capable of running Linux), the chosen wireless hardware platform of the Berlin Open Wireless Laboratory [8] tested.

3. PERFORMANCE EVALUATION

In order for our TCPSpeaker implementation to prove usable in its intended role, we demonstrate that its performance and macroscopic behavioral characteristics closely match those of the native system TCP implementation. The metrics we choose include throughput and RTT under conditions of packet loss and reordering, and flow-fairness under simultaneous multiple flow.

We want every flow handled by TCPSpeaker to achieve an equal share of the TCPSpeaker and network’s overall capacity for TCP throughput. In order to measure the unfairness over multiple flows, we employ Equation 1 [9], where $k$ is the number of simultaneous flows, $t_i$ is the throughput achieved by flow $i$, and $T_f$ is the throughput that flow $i$ would have achieved had it been the only flow to traverse the TCPSpeaker.

$$Unfairness = \sqrt{\frac{\sum_{i=1}^{k} \left(\frac{t_i}{T_f}\right)^2 - \left(\frac{1}{k}\right)}{k - 1}}$$

3.1 Experimental Design

To evaluate our implementation according to the previously given metrics, we design a series of experiments based on the test topology given in Figure 3. All experiments consistently utilize the same traffic generator, traffic sink, and traffic shaper for packet loss and reordering. These roles are filled by servers Loadgen 1, Loadgen 2, and Loadgen 3 respectively, as given in our topology diagram. Servers Loadgen 1 and Loadgen 2 run the Linux 2.6.18-6 kernel on a VIA Nehemiah x86-compatible processor running at 1 GHz with 512 Mbytes of RAM. Loadgen 3 runs a slightly newer 2.6.28-17 kernel.

As our system under test running the Click TCPSpeaker, we utilize the Gateworks Avila GW 2348-4 network platforms. Each Avila runs the Linux 2.6.26.8 kernel on an Intel IXP425 XScale CPU running at 533MHz with 64Mbytes of addressable SDRAM. This system is given by host Avila 2 in our diagram. As two TCPSpeakers run back-to-back on Avila 2, our measurement results reflect two separate TCP connections, measured from side A and B according to the diagram.

In our preliminary experiments, we establish TCP connections and transfer 20 MBytes of data from Loadgen 1 to Loadgen 2 between which, a pair of TCPSpeakers intercept and translate each TCP connection from TCP to TCP (effectively operating as a split-TCP proxy). We subject the flows traversing our TCPSpeaker implementation to varying conditions of packet loss and reordering via our shaper. We also establish varying numbers of simultaneous concurrent flows, and measure the per-flow throughput and RTT characteristics on both sides of the TCPSpeakers. We then com-

Figure 2: Architectural layers of two back-to-back TCPSpeakers, paired on the same host.

Figure 3: Preliminary experimental setup.
The results from our preliminary evaluation show that TCPSpeaker macroscopic behavior is comparable to the Linux Kernel TCP implementation and within acceptable and practically usable bounds. Essentially, we see that the TCPSpeaker responds resiliently on both sides of the split connection to both packet loss (see Table 1 and reordering.

For instance, under 10% packet reordering, TCPSpeaker Side A exhibits a median 92.5% of in-order throughput while side B exhibits median 52.5%. This reflects the need for buffer gaps introduced by out-of-order packets to be filled from side A before the data can be passed to side B. By comparison, the Linux kernel TCP Reno implementation exhibits roughly 24% median throughput under the same 10% packet reordering conditions.

Our measurements show that TCPSpeaker contributes positively to per-flow fairness, as well. As compared with the kernel implementation, the flow-scheduling and back-pressure mechanisms of the paired TCPSpeakers result in a more equal distribution of network bandwidth capacity to its flows. This can be seen as a reduction in our unfairness metric as flows pass from side A to side B of our network, illustrated in figure 4.

In summary, our measurements show that TCPSpeaker (i) exhibits operational correctness and macroscopic behavior of Linux Kernel TCP (ii) is performant and robust to loss and reordering and (iii) is fair.

4. OUTLOOK

This work describes a transport protocol translator, enabling clean-slate transport protocols to support legacy TCP end-hosts. Despite a rigorous development process, areas of implementation remain for further optimization and refinement. These include simple but delicate optimizations like rewriting the TCP options code for proper four-byte alignment on the ARM CPU architecture. For example, one of the most appealing features to be included in the TCPSpeaker implementation is support for the selective acknowledgement option. The ongoing work by the Click development team to bring Click into the unpatched Linux kernel may offer crucial performance gains, necessary for bringing the TCPSpeaker head-to-head with the performance of the kernel TCP implementation.

TCPSpeaker also raises significant questions at a protocol-design level. It remains to be investigated to what extent we impact upper-layer protocols which may receive acknowledgements for data that is never delivered. Accordingly, we plan for further measurements and investigation into the behavior of more complex use cases. The next steps involve an evaluation of the TCPSpeaker performance over multi-hop wireless when used in conjunction with our L4 protocol.

5. REFERENCES