MobiCom 2010 Poster: TCPSpeaker: Clean and Dirty Sides of the Same Slate

Dan Levin, Harald Schioeberg
dan,harald@net.t-labs.tu-berlin.de

Ruben Merz, Cigdem Sengul
ruben.merz,cigdem.sengul@telekom.de

Deutsche Telekom Labs/TU-Berlin

I. Introduction

The performance and fairness problems of TCP in wireless mesh networks (WMNs) are well known [1, 2]. Approaches to solve them fall roughly into two classes: *(i)* those which maintain legacy TCP compatibility, and *(ii)* clean-slate protocols which redefine WMN transport, breaking TCP compatibility. Clean-slate approaches such as block-switched and cross-layer transport protocols [3, 4] break the end-to-end principle, using intermediate WMN nodes for intelligent caching and pacing and show 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 live-network evaluation potential is severely limited.

We believe that the strengths of these two approaches may 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 [2], as it does not merely "split" TCP flows into multiple segments, but rather, enables the removal of TCP entirely from the WMN. To the WMN users, TCPSpeaker presents a transparent, fully TCP-compliant interface. Inside the WMN, TCPSpeaker introduces the freedom to use a new transport protocol more suited to the WMNs.

TCPSpeaker may further prove useful beyond the scope of WMNs, allowing any clean-slate transport to interact with legacy TCP end-hosts. We implement TCPSpeaker as a fully-functional Click element [5] 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.

II. 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. L4 is semantically equivalent to TCP if it is: bi-directional, connection-oriented, in-order, reliable bytestream with multiplexing, and flow-control. TCPSpeaker operates together with L4, with which it is paired back-to-back on the same node. Two separate back-to-back pairs operate at each edge of the WMN where TCP traffic enters/exits. Figure 1 depicts TCP traffic entering the WMN, where it is intercepted by a TCPSpeaker (1) and then passed to an L4 translator (2). 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.

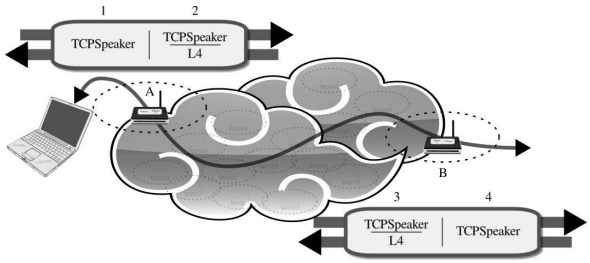

Figure 1: Use Case: TCPSpeaker-to-L4 pairs at the edges of a WMN translating intercepted TCP flows.

II.A. Flow Dispatching and Handling

TCPSpeaker handles TCP traffic on a per-flow basis, where a flow is made up of TCP segments of a bi-directional TCP connection sharing the same entry and exit nodes of the wireless network. This is crucial for preserving the connection-oriented nature 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 translates the incoming stateful TCP flow to a stateless intermediate bytestream, which 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 implemented via a zero-copy, intra-process communication, safeguarding the stateless bytestream in all but the most extreme

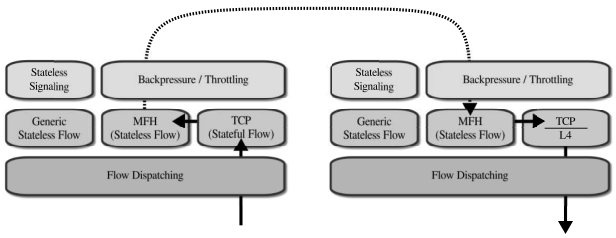

Figure 2: Architectural layers of two back-to-back TCPSpeakers, paired on the same host.

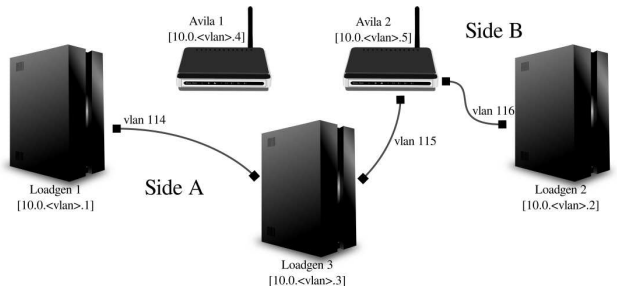

Figure 3: Preliminary experimental setup.

scenarios of total node failure.

For per-flow TCP-to-L4 semantics (i.e, flow-control) TCPSpeaker provides an interface for throttling the data rate 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, which allows a flow-control mechanism similar to backpressure.

II.B. Preserving Macroscopic Behavior

To preserve outward TCP compatibility, the TCPSpeaker preserves the macroscopic behavior of TCP with end-to-end flow control and outward in-order, reliable transport robust against packet loss and reordering. TCPSpeaker also ensures 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 may fail to reach its destination. However, in practice, this limitation should prove to be manageables since the backpressure mechanism limits the amount of data which could be lost to the bandwidth-delay product of the WMN. Additionally, an ACK only means that the segment was received by TCP and does not guarantee processing by the application. Nevertheless, the delivery guarantee issue of the TCPSpeaker under this condition merits future investigation.

II.C. TCPSpeaker as a Click Element

We implemented TCPSpeaker as a Click element. Our implementation allows us to run the same codebase on any system running Linux (among other platforms), in either user-space, kernel-space, or even within ns-2, with no code-level modifications. 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 BOWL testbed [6].

III. Performance Evaluation

To prove that TCPSpeaker implementation is usable in its intended role, we show that its performance and macroscopic behavior closely match those of the native system TCP implementation. The metrics of our evaluation are throughput and RTT under conditions of packet loss and reordering, and flow-fairness [7] under simultaneous multiple flows.

III.A. Experimental Design

We design a series of experiments based on the topology given in Figure 3. All experiments use the same traffic generator, traffic sink, and traffic shaper for packet loss and reordering. These roles are filled by servers `Loadgen 1`, `2` and `3`, respectively. Servers `Loadgen 1` and `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 use 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 the host `Avila 2` in the figure. As two TCPSpeakers run back-to-back on `Avila 2`, our measurement results reflect two separate TCP connections, measured from side A and B in Figure 3.

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 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 compare our measurements to those with the Linux kernel TCP Reno.

Table 1: Median throughput normalized to 0% Packet Loss Case vs. % Packet Loss.

% Loss	TCPS A	TCPS B	Kernel
0.1	96.9	96.9	102
1	77.4	78.0	69.7
5	5.97	5.95	6.3

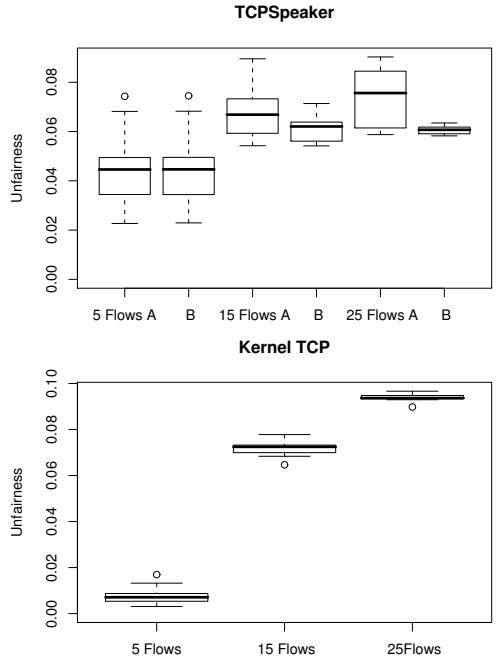

Figure 4: TCPSpeaker (top) vs. Kernel TCP RENO (bottom): Unfairness with varying number of flows. TCPSpeaker improves fairness of its translated flows.

III.B. Preliminary Results

Our results show that TCPSpeaker macroscopic behavior is comparable to the Linux Kernel TCP implementation and within acceptable and practically usable bounds. 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 since the flow-scheduling and backpressure mechanisms of the paired TCPSpeakers result in a more equal distribution of network bandwidth capacity to flows. This can be seen as a reduction in unfairness as flows pass from side A to side B of our network (see 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.

IV. Outlook

TCPSpeaker enables clean-slate transport protocols to support use and evaluation with legacy TCP endhosts. Despite a rigorous development process, areas of implementation remain for further optimization and refinement. An appealing feature for future inclusion in the TCPSpeaker is support for the selective ACK option. Recent efforts on Click with unpatched Linux kernel may also offer crucial performance gains, 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 seen to what extent we impact upper-layer protocols which may receive ACKs for data that is never delivered.

References

[1] P. Z. Z. Fu, H. Luo, "The impact of multihop wireless channel on TCP performance," *IEEE Trans. Mobile Comput.*, vol. 4, no. 2, 2005.

[2] S. Kopparty, S. Krishnamurthy, M. Faloutsos, and S. Tripathi, "Split TCP for mobile ad hoc networks," "UC Riverside", Tech. Rep., 2002.

[3] M. Li, D. Agrawal, D. Ganesan, and A. Venkataramani, "Block-switched networks: a new paradigm for wireless transport," in *NSDI*, 2009, pp. 423–436.

[4] K. Sundaresan, V. Anantharaman, H.-Y. Hsieh, and R. Sivakumar, "ATP: a reliable transport protocol for ad-hoc networks," in *MobiHoc*, 2003, pp. 64–75.

[5] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, 2000.

[6] R. Merz, H. Schiberg, and C. Sengul, "Design of a configurable wireless network testbed with live traffic," in *TridentCom*, May 2010.

[7] R. Jain, *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, 1991.