A TRANSPORT PROTOCOL FOR DEDICATED END-TO-END CIRCUITS

MS Thesis Final Examination

Anant P. Mudambi

Computer Engineering
University of Virginia

December 6, 2005
Outline

Motivation

CHEETAH Background

UDP-based Protocol

TCP-based Protocol

Control Plane Functions
Motivation

- Distributed Scientific Research
- Examples

**SETI@home**
Data Source: Radio Telescope, Peru
Data Processing: Berkeley, CA, USA

**Terascale Supernova Initiative**
Simulations: ORNL, TN
Scientists: NCSU, NC
Motivation

• Networking makes distributed research projects practical
• Large data sets, stringent performance needs
• Connectionless Internet model : not well-suited
• Connection-oriented networks : resources dedicated to a single transfer
Motivation

Problem Statement
Design and implement transport protocol for dedicated circuits.

Solution Approach

- Make use of the fact that resources are reserved for data transfer’s exclusive use
- High circuit utilization is important
Circuit-switched High-speed End-to-End Transport Architecture (CHEETAH)

- Benefits of dedicated circuit to end user
- Ethernet at edges
- Ethernet-over-SONET in core
- Multi Service Provisioning Platform (MSPP): Ethernet ↔ SONET
  1. Use Ethernet NIC
  2. Already deployed in enterprises
  3. Support standard signaling protocols
CHEETAH Features

• Add-on service
  • Applications using CHEETAH and the Internet coexist
  • Internet path provides fall-back

• GMPLS signaling
  • Distributed
  • Dynamic
CHEETAH Testbed

Diagram showing the testbed setup with GbE switches, Sycamore SN16000, and connections to hosts in different locations: NCSU, NC, MCNC, NC, ORNL, TN, and Atlanta, GA.
UDP-based protocol

Why UDP-based?

- Basic data transfer service
- Additional functions with no kernel modifications
- UDP and TCP sockets API widely deployed
Simple Available Bandwidth Utilization Library (SABUL)

- UDP data channel, TCP control channel
- Reliability
- Rate-based congestion control
- Reactive flow control
Fixed Rate Transport Protocol (FRTP)

- Modifications to SABUL
- Maintain constant rate, matched to the circuit rate
- SABUL’s rate altering congestion control disabled
- Support to use separate NICs for data and control channels
Problems with the FRTP Implementation

- Sender unable to maintain a fixed rate
- Fixed inter-packet gap used to maintain rate
- Inter-packet gap small for high rate
  - 12 $\mu$s to maintain 1 Gbps using 1500 B packets
- No OS support; use busy waiting
  - Wastes processor cycles
  - Not reliable
Problems with the FRTP Implementation

- Receiver unable to empty buffers at a rate higher than the sending rate
- User space buffer filling/emptying under user-space process’ control
- Kernel UDP buffer filled asynchronously from network
- Flow control required to avoid UDP buffer overflow or reduce its impact
- Reduce loss: FRTP does not have reactive flow control
Possible Solutions

- Give up CPU after transmitting 1 packet; reclaim when next packet has to be sent
- Maintain rate in longer time frame, not packet-by-packet
- Need access to kernel buffer status
- Overflow avoidance: requires kernel support
TCP-based Protocol

Pros

- Kernel support for flow control
- Self clocking: low-overhead method to maintain steady sending rate
- Widely deployed and used
- Error control

Cons

- Rate-altering congestion control
- Kernel-space modification
Circuit-TCP

- C-TCP requirements adequately met by TCP for
  - Connection establishment: 3-way handshake
  - Multiplexing: using port numbers
  - Flow control: window-based
  - Error control: sequence numbers, retransmission time-out, triple duplicate ACKs

- Congestion control
  - TCP congestion control alters transfer rate
  - C-TCP requires steady transfer rate maintained
Circuit-TCP Congestion Control

Phase 1: Control plane

- Initiate request for a circuit by invoking a signaling protocol client.
- Await circuit set up status.

Circuit set up status:
- SUCCESS: Initiate data transfer
- FAILURE: Handle circuit set up failure
Circuit-TCP Congestion Control

Phase 2 : Data plane

- TCP congestion control varies data transfer rate
  - Slow start, congestion avoidance, reaction to loss
- Rate variation detrimental when resources are reserved
- Maintain maximum rate while avoiding network buffer overflows
Circuit-TCP Implementation

- Implemented using Web100 instrumented stack on Linux 2.6.11

Requirements
- Maintain steady data transfer rate, closely matched to reserved circuit rate
- Co-exist with TCP
Web100

- Instrumented stack
- Read TCP internal state variables
- Set certain TCP parameters
- Interface through /proc
Circuit-TCP Implementation

- 2 new Web100 control parameters *useckt* and *ncap*
  - *useckt*: single protocol stack handles TCP and C-TCP
  - *ncap*: amount of unacknowledged data = $\min(ncap, rwnd)$; *ncap* set to BDP
- Receiver advertised window behavior fixed
EXPERIMENTAL RESULTS
Testbed Configuration
Start-up Behavior for C-TCP and TCP

Graph showing the start-up behavior of C-TCP and TCP, with axes labeled as follows:

- Y-axis: Sequence offset
- X-axis: Relative time (seconds)

The graph compares C-TCP and TCP window and data transmission behaviors over time.
Utility of Disabling Slow Start
(Circuit rate = 1 Gbps)
Sustained Data Transfer: Reno-TCP
(Circuit rate = 500 Mbps, baseline RTT = 13.6 ms)
Sustained Data Transfer: BIC-TCP
(Circuit rate = 500 Mbps, baseline RTT = 13.6 ms)
Sustained Data Transfer: C-TCP
(Circuit rate = 500 Mbps, baseline RTT = 13.6 ms)
Control-Plane Functions

- Control plane functions support the data transfer
  - Initializing sequence numbers
  - Setting up a circuit
- 2 parts to C-TCP control plane functions
  - Selecting the circuit rate
  - Setting up the circuit
Selecting the Circuit Rate

• Trade-off: circuit utilization and transfer delay
• Set up circuit with rate equal to maximum rate end hosts can sustain
• How to determine this rate?
• End-host variability
  • Multitasking
  • Disk access
Throughput Variability for Disk-to-Disk Transfers
(Circuit rate = 1 Gbps, file size = 1.6 GB)
Disk I/O benchmark

- Disk write most likely bottleneck
- Estimate disk write rate beforehand
- Select circuit rate based on estimated disk write rate
- `xdd` benchmark
- High variability in benchmark results
- Affect of interaction with network not captured
Setting up the Circuit

- C-TCP API
- `CTCP_sender_connect()` / `CTCP_receiver_connect()`
- `CTCP_sender_accept()` / `CTCP_receiver_accept()`
- Wrapper around TCP sockets API calls
  - uses RSVP-TE library to set up circuit
  - sets C-TCP parameters using Web100 API
Setting up the Circuit

• Application cannot be modified to use C-TCP API
• CTCP Work Around Daemon (CTCP-WAD)
• \texttt{bwrequestor} informs CTCP-WAD of new circuit request
• CTCP-WAD periodically checks for socket using that circuit
• When found, sets its C-TCP parameters
Conclusions and Future Work

Conclusions

• Match data transfer rate to reserved circuit rate
• High utilization, low transfer delay
• FRTP: user-space, UDP-based
• C-TCP: kernel support for flow control
• Steady data transfer rate

Future Work

• Determining the circuit rate
• Real-time support
Conclusions and Future Work

Conclusions

• Match data transfer rate to reserved circuit rate
• High utilization, low transfer delay
• FRTP: user-space, UDP-based
• C-TCP: kernel support for flow control
• Steady data transfer rate

Future Work

• Determining the circuit rate
• Real-time support

THANK YOU
Self Clocking

![Diagram of Self Clocking mechanism]

- **Sender** sends data through **Network Node** to **Receiver**.
- **Network Nodes** transmit **Data** and receive **ACKs**.
- **Link1**, **Link2**, and **Link3** represent the channels for data transmission.
- **ACKs** acknowledge the receipt of data packets.