Using Time Division Multiplexing to support Real-time Networking on Ethernet

Hariprasad Sampathkumar

25th January 2005

Master's Thesis Defense

Committee

Dr. Douglas Niehaus, Chair

Dr. Jeremiah James, Member

Dr. David Andrews, Member

Outline

- Introduction
- Related Work
- Objectives
- Background
- Implementation
- Evaluation
- Conclusion
- Future work

Introduction

- Ethernet dominant LAN technology in office and educational environment
- Advantages – Low cost and ease of installation
- Ideal technology for industrial automation, if it can support applications with time constrained QoS
- Traditional Ethernet based on CSMA/CD
- Disadvantages – Collisions and exponential back-off causing random delay in packet transmission
- Unable to support real-time applications due to non-determinism in packet transmission

Related work

- Hardware Approaches
  - Expensive, require specialized hardware and software
- Token bus and Token Ring Architectures
  - – Token passing protocol, collision free, deterministic transmission
- Switched Ethernet
  - – Private collision domain for machines on destination port
- SCRAMNet – Shared Common Random Access Memory Network
  - – Write to shared memory to transmit, reflects data throughout the network in bounded time

Related Work

- Software Approaches
  - RTnet – Hard Real-Time Networking for Linux/RTAI
    - TDMA based collision free transmission
    - separate network stack for real-time processes
  - RETHER Protocol
    - Timed-token protocol suitable for video transmission
  - Traffic Shaping
    - statistical guarantees for collision-free transmission
    - controls rate of transfer of non-real-time packets
  - Master/Slave Protocols
    - Master controls transmission of packets

Objectives

- Make Ethernet suitable to support real-time applications by providing collision-free packet transmission
- Solution should support existing Ethernet Hardware
- Modifications need to be minimal without affecting existing network and transmission protocols
- Proposed Solution
  - Implement Time Division Multiplexing on Ethernet
  - Use the framework provided by KURT-Linux

Background

- UTIME – High Resolution Timers
- Datastreams Kernel Interface (DSKI)
- Group Scheduling Framework
- Time Synchronization in a Distributed Network
- Control Flow of a Packet through the Linux kernel during transmission and reception
- NetSpec

UTIME

- Standard Linux notion of time is jiffies – timing resolution of 10ms in 2.4.20 kernel – not sufficient for real-time applications
- UTIME modifications to support subjiffy resolution, typically in microseconds
- UTIME offset timers take into account timer interrupt overhead and schedule accurate timer events
- UTIME provides a privileged timer that allows timer handling routines to be executed in interrupt context

Datastreams Kernel Interface (DSKI)

- Method to gather data relating to operating system's state or performance
- Used to log and timestamp events as they happen inside the kernel
- Data collected as events, counters or histograms
- Data is presented in a standard XML format
- Post-processing applied on the collected data to generate graphs
- Supports visualization of events collected from a distributed network on a global timescale
- Accessed by standard device driver conventions and allows to collect only events the user is interested in

Group Scheduling

- Unified scheduling model used to control scheduling and execution semantics of different computational components
- Computational components are processes, hardirqs, softirqs, tasklets and bottom halves
- Components represented in a hierarchic decision structure
- Groups – nodes in scheduling hierarchy that direct the decision path
- Each group has a name and scheduler associated with it
- Groups contain members which are computational components or other groups
- Associated scheduler determines scheduling semantics imposed by a group on its members

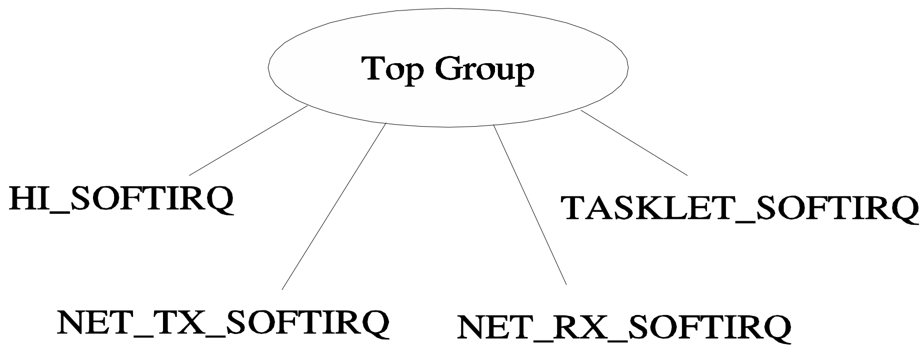
Group Scheduling

- Scheduler associated with root group is invoked, which recursively invokes schedulers of member groups, if any
- Decisions of member groups propagated to the root of the hierarchy which decides the member to be scheduled next
- Can be used to achieve customized scheduling and execution semantics for computational components
- Framework defines function pointer hooks to scheduling and execution routines of different computational components, that map to Vanilla Linux semantics by default
- Define custom routines that map on to these function pointers to have custom semantics
- Used in defining TDM Model

Group Scheduling

- Vanilla Linux Softirq semantics
  - Linux 2.4.20 kernel has following softirqs:
    - HI_SOFTIRQ – Handle high priority tasklets and bottom halves
    - NET_TX_SOFTIRQ – Process transmission of network packets
    - NET_RX_SOFTIRQ – Process reception of network packets
    - TASKLET_SOFTIRQ – Handle low priority tasklets
  - Maintains pending softirq flags for each CPU
  - A snapshot of pending softirqs is taken and are executed in decreasing order of priority
  - Invokes a kernel thread to perform the processing in case of large number of softirqs

Group Scheduling



- Linux Softirq model under Group Scheduling
  - Top group with the 4 softirqs as its members
  - Softirqs added in decreasing order of priority
  - Members are selected sequentially
  - If the pending bit of the selected softirq is enabled, the member is selected for execution

Time Synchronization in a Distributed Network

- Time synchronization among nodes needed to gather real-time data in a distributed network
- Modified Network Time Protocol (NTP) support under KURT-Linux offers synchronization on order of microseconds
- Precision is about ± 5 μs on an average and ± 16μs in the worst case
- Provides time synchronization for supporting TDM

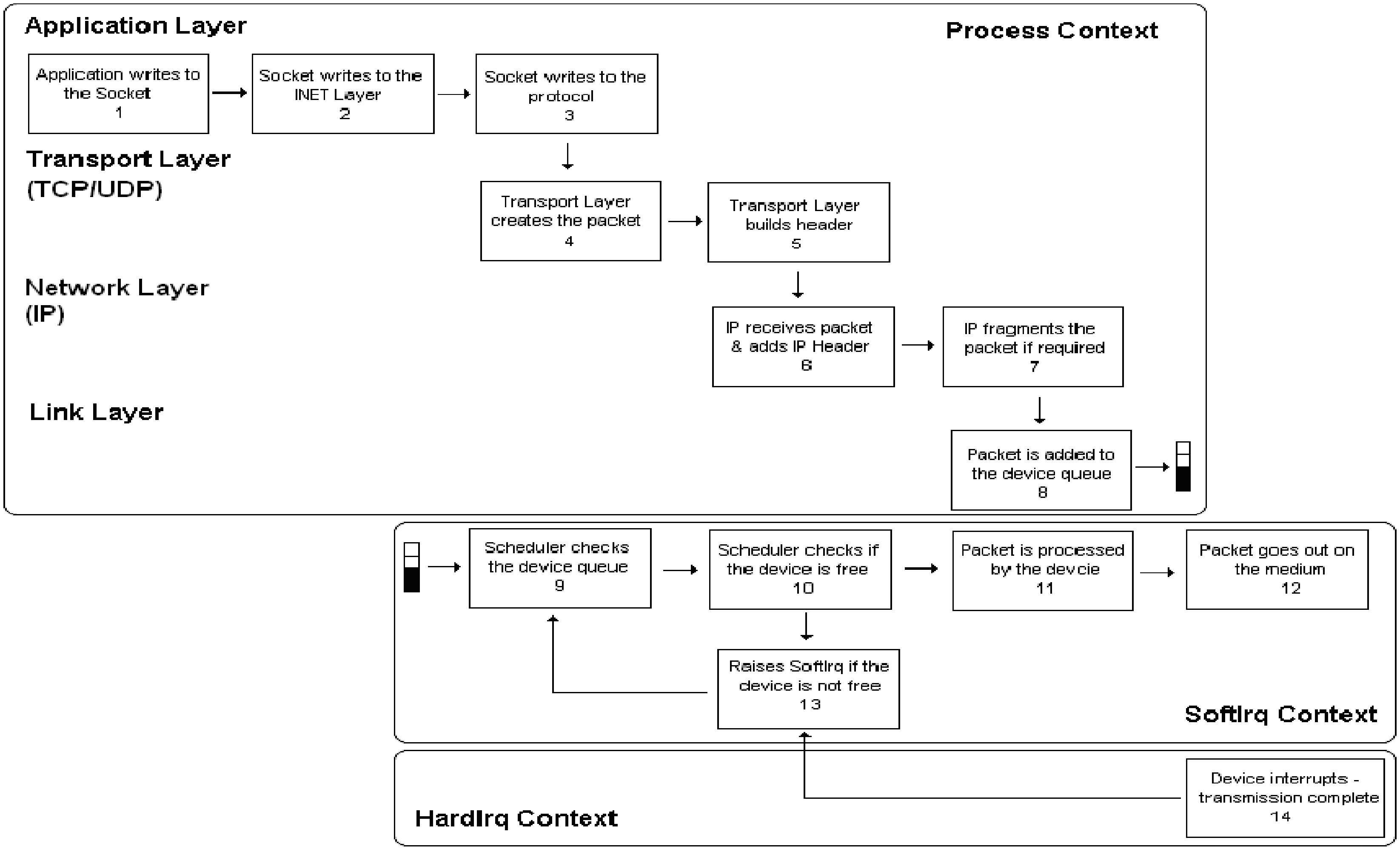
Linux Network Stack

- Data structures : Socket buffer (sk_buff) and Socket (sock)
- sk_buff represents a packet in the network stack and contains pointers to different headers of the protocol stack
- Processing of a packet in a layer is manipulation of the corresponding header in the socket buffer structure
- Movement of a packet between layers is achieved by simply passing a pointer to socket buffer
- Sock is created when a socket is created in user space
- Sock maintains state of a TCP or a virtual UDP socket connection

Linux Network Stack – Packet Transmission

- Packet Transmission
  - Starts from the application in process context
  - Packet gets queued in the net-device layer
  - If device is free packet transmission occurs in process context
  - If not, NET_TX_SOFTIRQ is enabled to carry out transmission in Softirq context

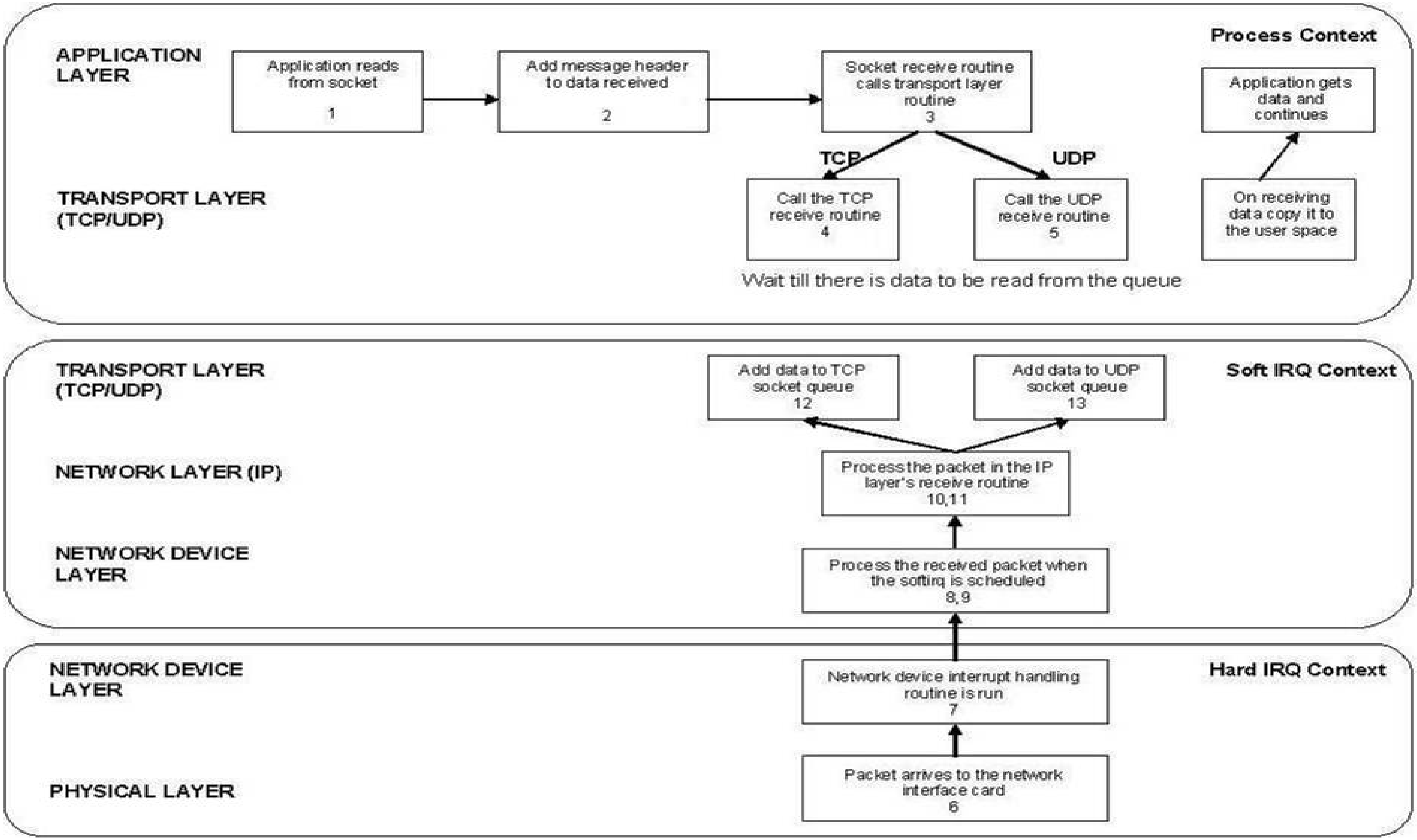
Linux Network Stack – Packet Transmission



Linux Network Stack – Packet Reception

- Packet Reception
  - Has two flows of execution
  - Application layer to Transport Layer
    - Process blocks for incoming packets
    - Execution carried out in process context
  - Physical layer to Transport layer
    - Packet received from network is sent up to the queue in transport layer
    - Execution carried out in both hardirq and softirq contexts

Linux Network Stack – Packet Reception



NetSpec

- Tool used to automate schedule of experiments involving several machines in a distributed network
- Daemons run in machines that are part of experiment
- NetSpec controller passes experiment schedules to the daemons, which carry out the experiment
- Experiments specified in script file
- Supports transfer of configuration files and collection of output files

Implementation

- Kernel Modifications
  - Reduce latency in packet transmission
  - Packet transmission in softirq context
  - TDM Model under Group Scheduling
  - TDM Scheduler
- User Interface
  - TDM Master-Slave configuration
  - User space programs to configure TDM

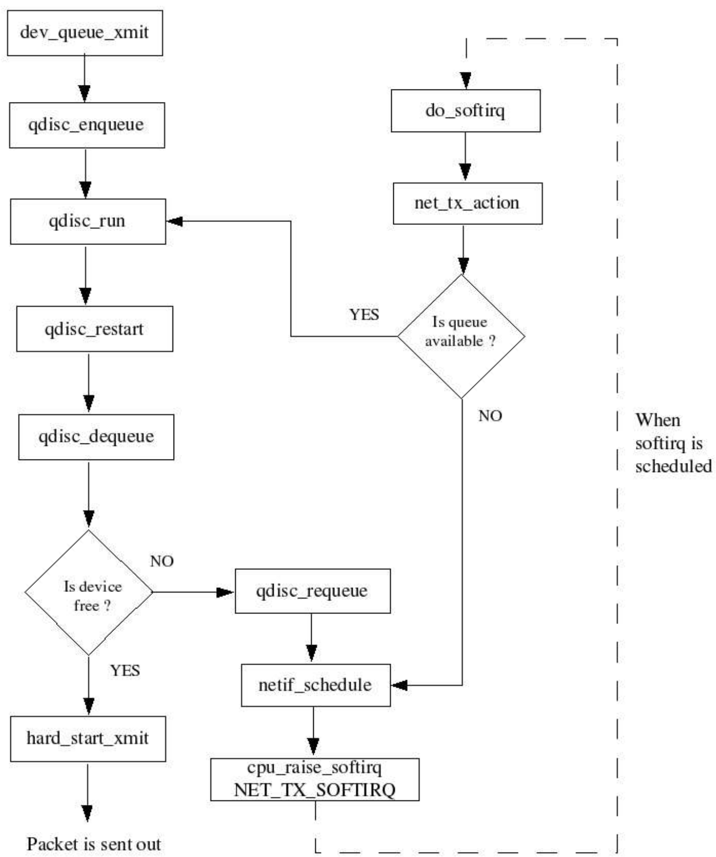
Reducing Latency in Transmission

- Perform only transmission during time-slot, delay all non-critical operations
- NET_TX_SOFTIRQ handling routine first frees socket buffers of packets that have been transmitted and then starts packet transmission
- Handling routine modified to just perform packet transmission
- Create new low-priority NET_KFREE_SKB_SOFTIRQ that performs the garbage collection

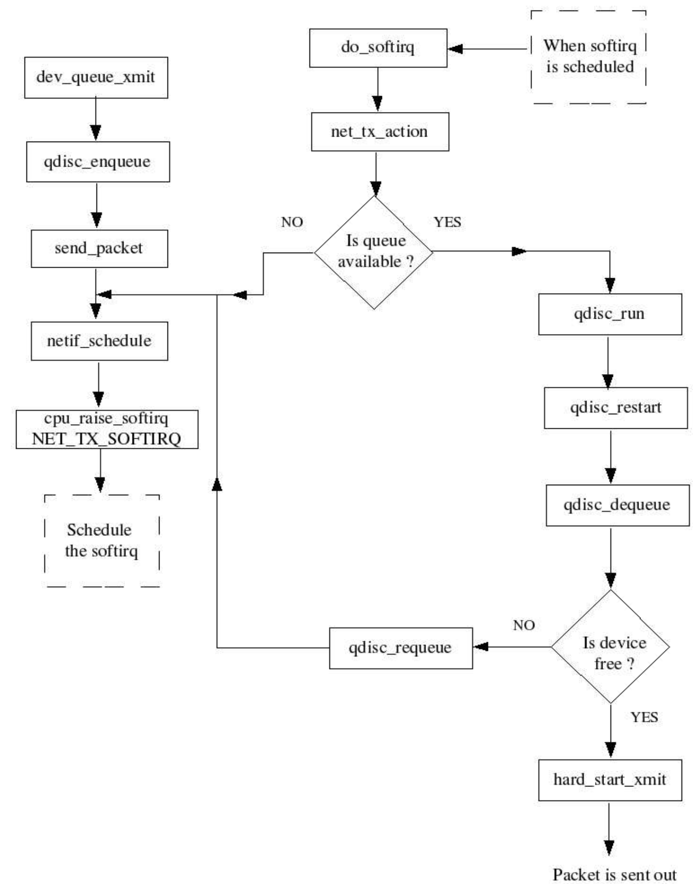
Packet transmission in Softirq Context

- Transmission can occur in both Process or Softirq context
- Time-triggered transmissions require control over computation performing the transmission
- Force transmissions to occur in softirq context beyond the net device layer
- Packet is added to queue and NET_TX_SOFTIRQ is enabled to transmit the packet

Packet transmission in Softirq Context



Normal Execution Flow

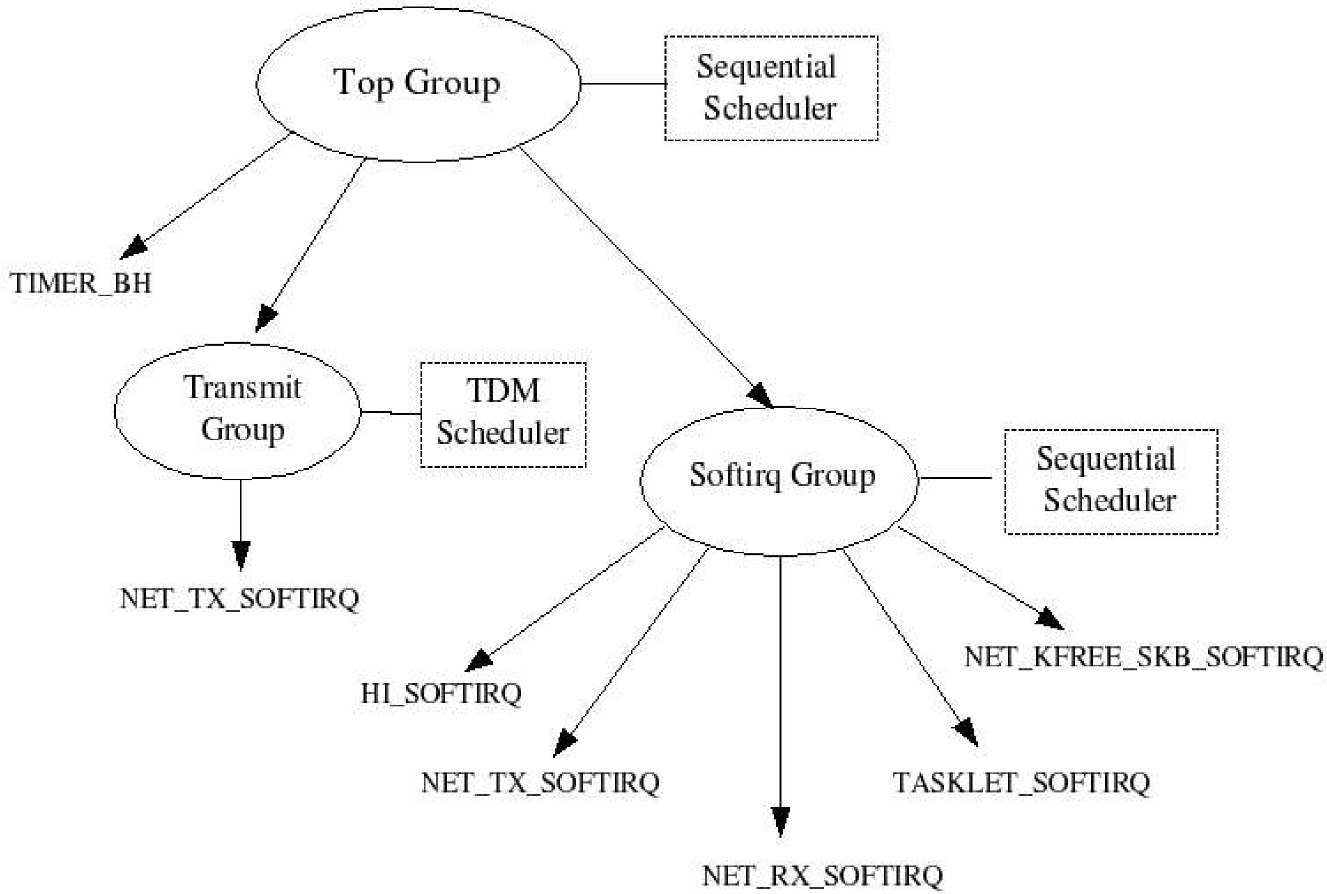


Transmission in softirq context

TDM Model under Group Scheduling

- Time related updates must be provided to the machine's clock immediately.
- Transmission must take place at scheduled intervals of time when TDM is enabled - higher priority for NET_TX_SOFTIRQ
- When TDM is not enabled NET_TX_SOFTIRQ has default priority
- NET_KFREE_SKB_SOFTIRQ to have the lowest priority

TDM Model under Group Scheduling



Time Division Multiplexing Scheduler

- Creates a privileged UTIME kernel timer
- time_to_transmit flag denotes the transmission slot
- Two timer handling routines for the start and end of transmission intervals
- Timer handling routine for start of time slot
  - Sets time_to_transmit to true
  - Sets kernel timer to expire for the end of time-slot
- Timer handling routine for end of time slot
  - Sets time_to_transmit to false
  - Sets kernel timer to expire for the start of time-slot
  - calculates start and end expirations for next cycle

TDM Scheduling Decision Function

Program 4.3 Pseudo-Code for the TDM Scheduling Decision Function

```
group_member tdm_scheduler (previous_task_struct, this_cpu,
                                                    group_member){
    group_member = get_member_from_member_list();
    if(group_member == NET_TX_SOFTIRQ){
       if(tdm_status == TDM_ENABLED){
          if(time_to_transmit == TRUE){
             if(net_tx_softirq_is_pending){
               return group_member;
             }
          }
       }
    }
    return global_pass_member;
}
```

TDM Master –Slave Configuration

- Any machine can be configured as TDM Master
- TDM Daemon started in remaining machines which act as slaves
- Determine number of machines in setup – initial handshake between the master and slaves
- Broadcasts a ‘hello’ message to all machines in LAN segment
- Slave machines part of TDM reply for the broadcast message
- Master computes the schedule for each machine
- Each slave machine is provided with its TDM schedule through a new connection
- Slaves submit schedule to the kernel to start TDM

Calculating Transmission Schedules

- Total Transmission Period = T
- Number of Machines = N
- Ideal Time Slot size = $TS_{ideal\text{-}size}$ = T/N
- Buffer Period between timeslots = B
- For a machine of ordinality ‘n’
  - Time slot Begin time

    $$TS_{begin} = ((n-1) * TS_{ideal\text{-}size}) + (B/2)$$

  - Time slot End time

    $$TS_{end} = (n * TS_{ideal\text{-}size}) - (B/2)$$

User Space Commands

- Interface through standard device driver conventions
- To submit TDM schedule
  - tdm master <broadcast address> <minutes> <seconds> <total transmission cycle>

Where

<broadcast address> - broadcast address of LAN segment where TDM is to be enabled

<minutes> - time in minutes from now when TDM is to be started

<seconds> - time in seconds from now when TDM is to be started

<total transmission cycle> - time in nanoseconds including the transmission time-slots of all the machines in the TDM network

- To stop TDM schedule
  - tdm stop

Evaluation

- Determine the transmission times of packets of varying sizes
- Selection of a suitable buffer period based on the time synchronization achieved
- Setting up TDM schedule based on the transmission time and buffer period determined
- Testing TDM schedules for collisions for packet transmissions of various sizes

Determining Packet Transmission Time

- Total Theoretical Transmission Time

  $T_{\text{total}} = T_{\text{convert bits into signals}} + T_{\text{Propagation delay}}$ where

  $T_{\text{convert bits into signals}} = M / L$ and

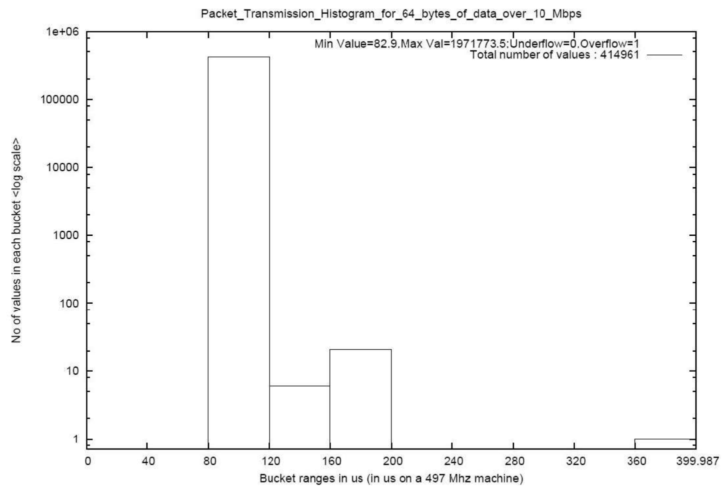
  $T_{\text{Propagation delay}} = D / C$ where

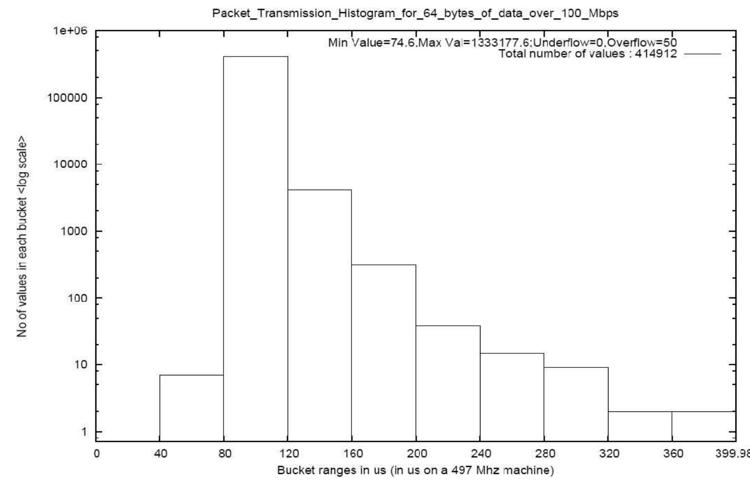
  - M – Message size in bits
  - L – Link Capacity in Mbps
  - D – length of physical link in meters
  - C – Speed of light in the physical medium in m/s
- Propagation delay is negligible as D is small
- Therefore $T \approx T_{\text{convert bits into signals}}$

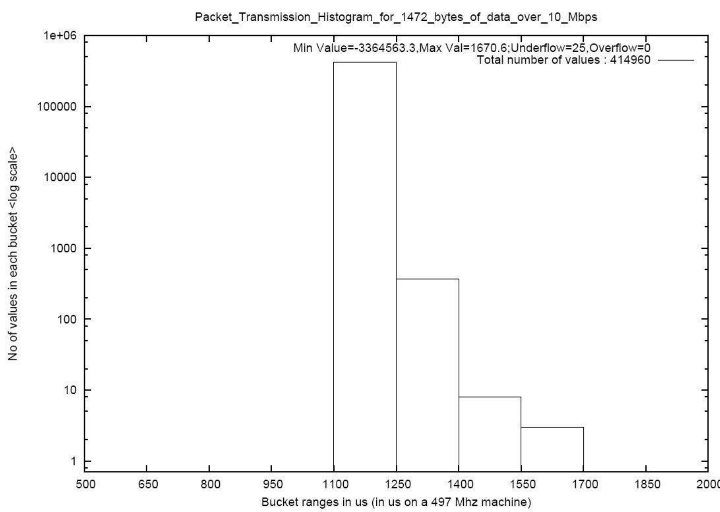
Determining Packet Transmission Time

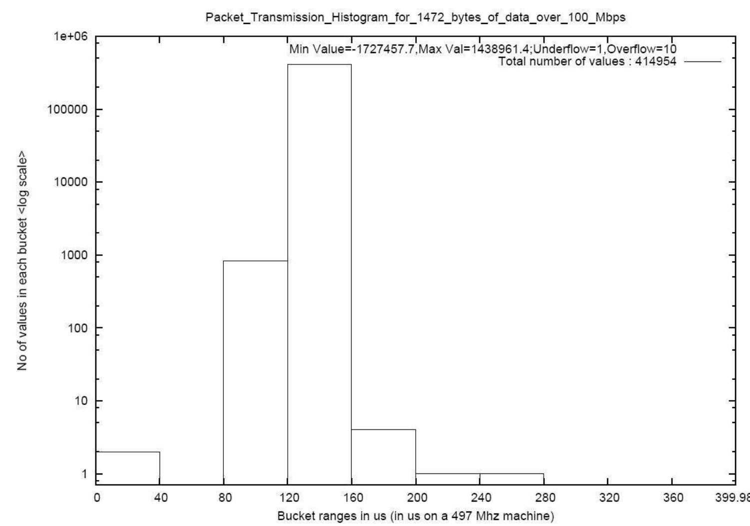
- Two 500 MHz machines running KURT-Linux without any TDM modifications
- Measure time intervals between successive reception of packets using DSKI histograms
- A stream of about 400,000 packets were transmitted from a UDP application
- The transmission times were recorded for varying message sizes.
- Tests performed for both 10 and 100 Mbps Ethernet

Determining Packet Transmission Time









Determining Packet Transmission Time

Size of Data (in bytes)	Total Packet Size (in bytes)	10Mbps Ethernet (in µs)	100Mbps Ethernet (in µs)
upto 18	64	51.2	5.12
64	110	88	8.8
128	174	139.2	13.92
256	302	241.6	24.16
512	558	446.4	44.64
1024	1070	856	85.6
1472	1518	1214.4	121.44

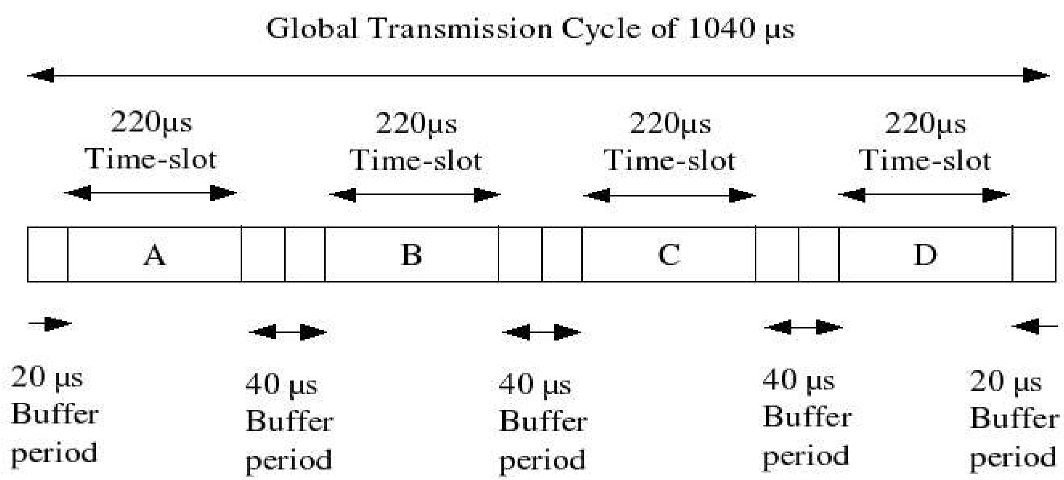
Table 5.1: Theoretical Transmission times for 10Mbps and 100Mbps Ethernet

Size of Data (in bytes)	Total Packet Size (in bytes)	10Mbps Ethernet (in µs)	100Mbps Ethernet (in µs)
upto 18	64	100.05	95.17
64	110	100.49	94.94
128	174	140.11	96.31
256	302	274.95	96.01
512	558	450.03	98.87
1024	1070	850.04	99.64
1472	1518	1275.14	122.51

Table 5.2: Observed Average Transmission times for 10Mbps and 100Mbps Ethernet

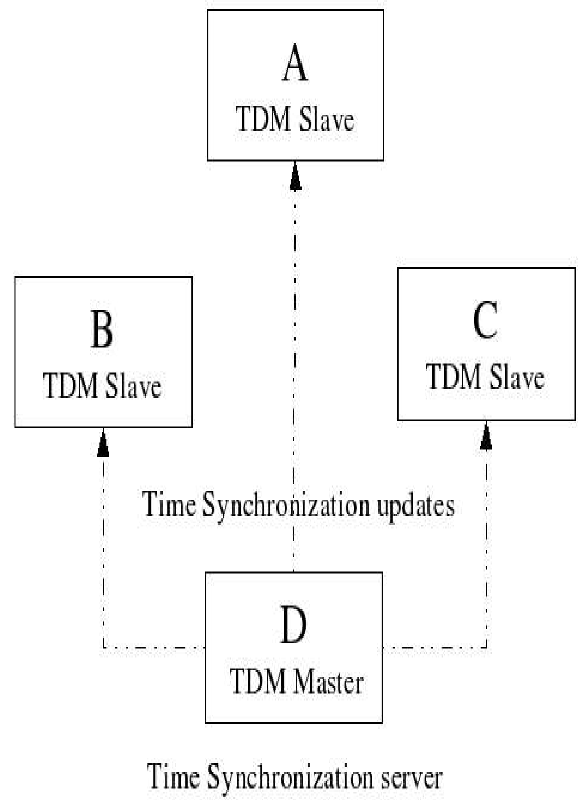
Buffer Period between time-slots

- Precision of time synchronization from NTP modification scheme is ±5 μs on average and ±16 μs in worst case
- Machines can be as far apart as 32 μs
- We settle on a value of 40 μs for buffer period



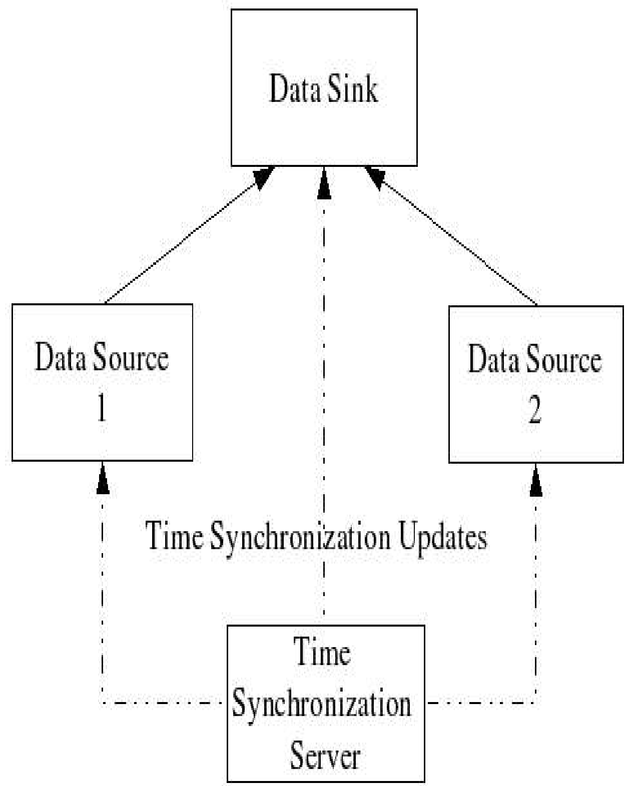
Setting up TDM Ethernet

- Four 500Mhz machines with TDM modifications on KURT-Linux
- Achieving time synchronization
  - Clock Calibration
  - Clock Synchronization
- Time Server sends updates every 5 minutes
- Start TDM Daemons and submit schedule from the TDM Master

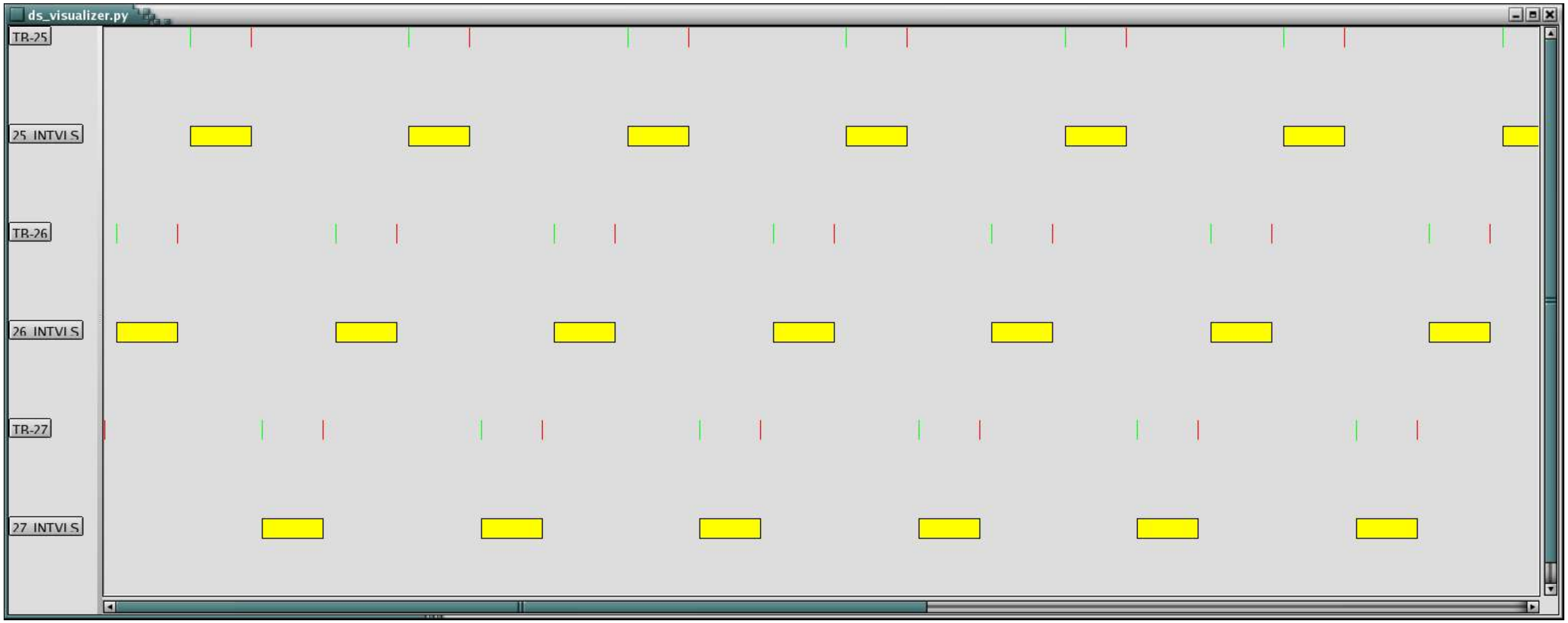


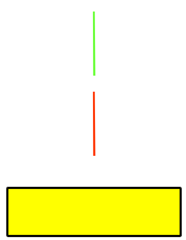
Two Sources and a Sink

- TCP application transferring about 10,000 packets from both sources to the sink
- Time slot of 1300μs for 1500 bytes of data on 10Mbps
- Number of collisions observed to be zero
- DSKI events were collected on Sink



Visualization of Transmission Time-slots in TDM Ethernet





Event denoting start of transmission time-slot

Event denoting end of transmission time-slot

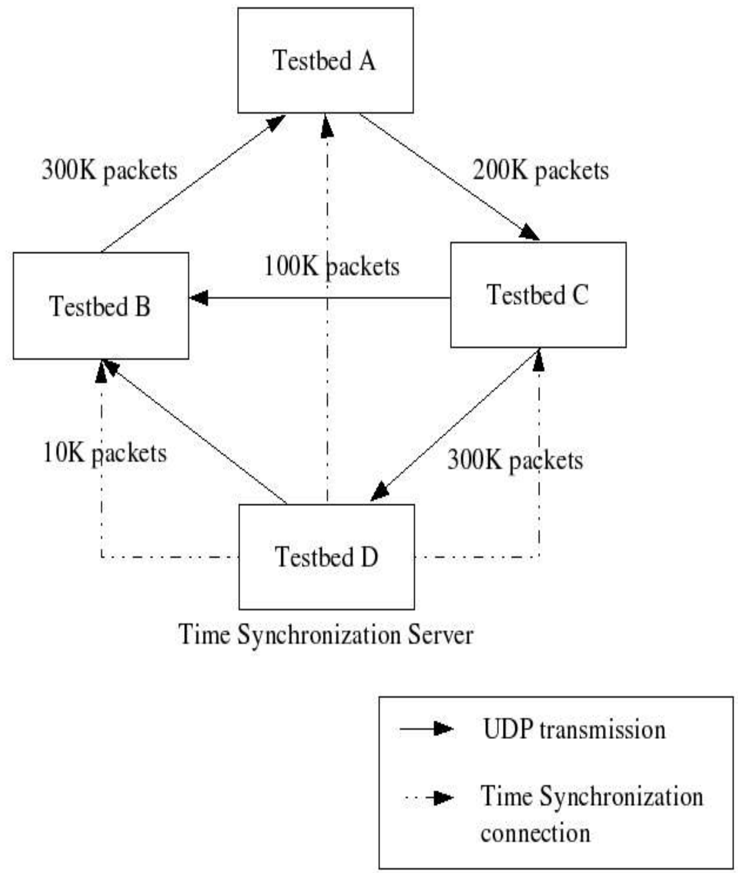
Interval denoting transmission time-slot

TDM Schedules for varying packet sizes

- Multiple UDP transmissions generating over a million packets
- Transmission times measured for data of 64, 256 and 1472 bytes
- Tested with 100Mbps Hub
- MTU was varied based on the different packet sizes
- Suitable time-slots obtained when there were no collisions or packet loss

Size of Data (in bytes)	Total Packet Size (in bytes)	Time slot in 100Mbps Ethernet (in μs)
64	110	220
256	302	260
1472	1518	440

Table 5.3: Transmission time-slots for 100Mbps Ethernet



Conclusion

- Time Division Multiplexing can be employed to achieve collision-free deterministic transmission on Ethernet
- Suitable time-slots for transmission for different packet sizes have been measured for 100Mbps Ethernet
- Accomplished with minimal modifications to network stack
- It is a software solution, will support any common Ethernet Hardware
- Suitable for Industrial Automation applications requiring periodic transmission
- Can be used even on Switched Ethernet to avoid packet loss and queuing latency

Future work

- Port to Linux 2.6 kernel
  - Has 2 additional softirqs
  - Delayed Timer bottom half handling is a softirq
  - Modification to TDM Group Scheduling hierarchy
- Creation of a TDM Schedule Server
  - Creates TDM schedules taking more constraints into account
  - Machine with larger volume of data is given a larger time slot for transmission

Thank You