Implementation Of Micro Transport Protocol (µTP)

Prepared for:

MS Project

Department of Computer Science

University of Kentucky

Prepared by:

Sandip K Shah

Advised by:

Dr. Raphael Finkel

Professor, University of Kentucky

Abstract

This document describes the Micro Transport Protocol (µTP), implemented in Linux version 2.2.10. µTP is a transport layer protocol that implements a simple, reliable, packet stream service on top of IP datagram service. The objective of this project is to provide the user with another reliable transport layer protocol and compare µTP's performance with TCP. Results show that for certain test cases, µTP outperforms TCP.

1 Introduction

The Micro Transport Protocol (µTP) is a connection−oriented end−to−end protocol implementing a reliable, full−duplex packet−stream service between application processes residing on different hosts. µTP was designed by Dr. Kenneth Calvert[1] in 1995. It is designed to fit into a layered hierarchy of protocols that support various network applications.

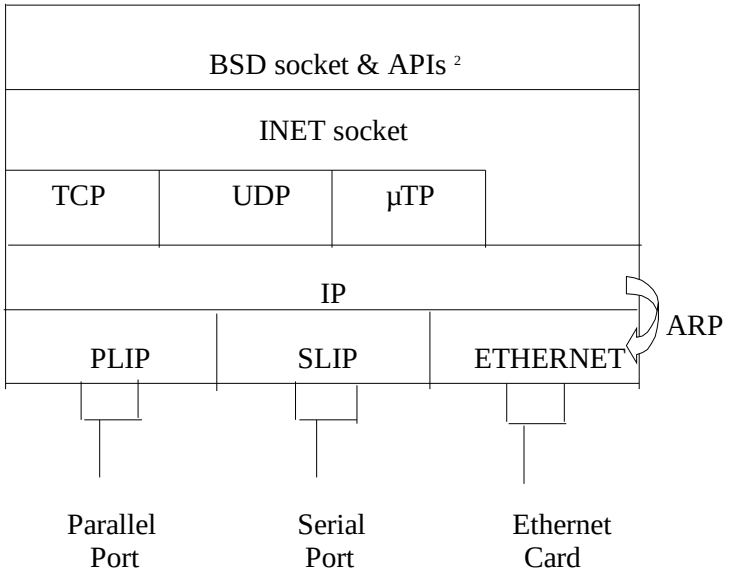


Figure 1: Protocol Layering

When an application process running on top of µTP communicates via the network, it uses functions provided by the BSD socket layer and µTP−specific system calls. As shown in Figure 1, below this layer is the INET socket layer, which manages the communication end−points for the IP−based protocols like TCP, UDP and µTP. µTP assumes it can obtain a simple, potentially unreliable datagram service from the lower−level protocols. µTP fits into the layered protocol architecture just above the Internet Protocol (IP), which provides a way for µTP to send and receive variable−length segments of information enclosed in IP datagrams. The IP datagram provides a means for addressing source and destination hosts in different networks. The internet protocol also performs fragmentation and reassembly of IP datagrams to transport through multiple networks.

Application processes residing on different hosts can use µTP to establish connections and exchange sequences of packets. Similar to TCP, µTP establishes connection using three−way handshake communication. A µTP connection is always established between an active end that initiates the connection and a passive end that waits for the connection requests to arrive from the network.

As long as the connection remains open, the protocol delivers packets in the order they are sent,

1 Faculty member in the College of Computing at Georgia Tech from 1991 to 1998 and now Assistant Professor at University of Kentucky.

2 some APIs (or system calls) are specific to µTP and cannot be used for TCP or UDP.

without modifying or discarding them. But µTP does not provide an orderly `close()` function; all data transfer stops abruptly as soon as either end requests that the connection be terminated. Loss of data is possible during connection termination.

Like TCP, a pair of endpoints identifies a µTP connection. An endpoint is defined as a tuple {IP addr, port number}. But unlike TCP, only the connection−establishment messages carry the originating and destination port numbers. At connect time, each endpoint chooses a reference number for the new connection and informs the other endpoint. After the initial connection−establishment phase, the connection is identified with that reference number, that is, each message has a destination reference number (a single port can have several reference numbers and hence can be involved in different connections). The idea is that the recipient uses the reference number to quickly locate the state information associated with the connection and to reduce the header size.

The receiver uses sequence numbers to detect loss, duplication, and disordering of data packets. The dynamic sliding−window mechanism uses sequence numbers for efficient transmission and flow control. A µTP acknowledgment specifies the sequence number of the next data message that the receiver expects to receive. It also contains the receiver's window size. Acknowledgment information can be piggybacked on data traveling in the opposite direction.

When congestion occurs, delays increase, causing µTP to retransmit PDUs (Protocol Data Units). In the worst case, retransmissions increase congestion and produce an effect known as congestion collapse. Like TCP, µTP uses a slow start multiplicative decrease technique to avoid congestion collapse.

2 µTP State Machine (Connection Management)

Figure 2 shows the state transitions of a connection endpoint during the lifetime of a connection. The following representation is used:

: State of a connection

+X : Receiving Message X
−Y : Sending Message Y

CR : Connect Request
CA : Connect Ack
CC : Connect Confirm
D : Data
XX : Disconnect

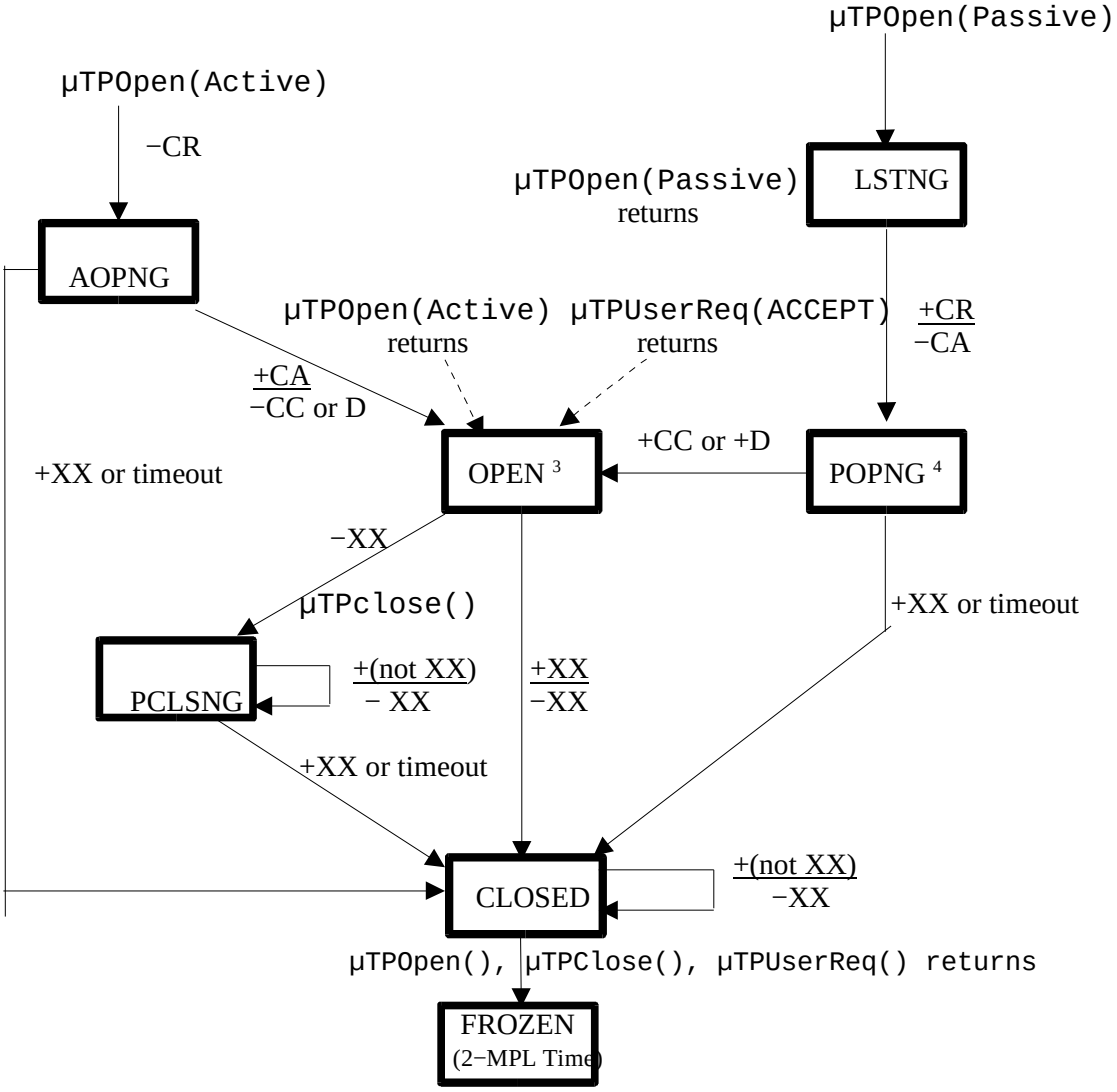


Figure 2: µTP State Machine

3 Data PDUs may be send or received.
4 Data PDUs may be received.

2.1 Establishing a Connection

This section describes the states of the endpoints as they establish a connection. Let us consider two hosts A (running µTP server) and B (running µTP client). The server on host A invokes `µTPOpen()` on a local port, say X, with the active/passive flag set to PASSIVE. A connection endpoint is associated with port X and is placed in the LSTNG (Listening) state. The µTP client on host B invokes `µTPOpen(ACTIVE)` with destination host A and destination port X. The client sends a Connect Request PDU and enters the AOPNG (Active Opening) state. It remains in that state, retransmitting Connect Request periodically, until a reply (either Connect Ack or Disconnect) is received, at which point it enters the OPEN state and sends either Connect Confirm or a Data PDU. When the server in the LSTNG state receives a Connect Request PDU, it sends Connect Ack in reply and enters the POPNG (Passive Opening) state. It remains in that state, retransmitting Connect Ack periodically, until receiving a reply (Connection Confirm or Data), at which point it enters the OPEN state. Figure 3 shows the messages exchanged during connection establishment.

DR: Destination Reference LP: Local Port CR: Connect Request W: Window size
OR:Originating Reference DP: Destination Port CA: Connect Ack
RR: Responding Reference C3: Connect Confirm
XX: Disconnect

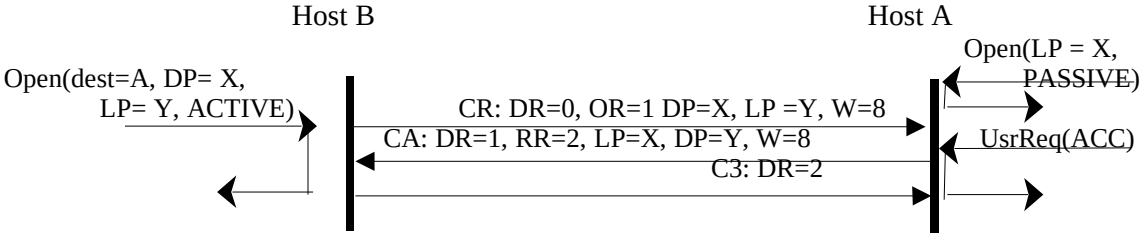


Figure 3: Connection Establishment

It is possible for the PASSIVE end to receive Disconnect in response to Connect Ack when it is in the POPNG state. This situation can happen as follows: The ACTIVE end receives Connect Ack, sends a Connect Confirm PDU that is lost by the network, and enters the OPEN state, whereupon the µTP client immediately calls `µTPClose()`. In this case, the PASSIVE end sends Disconnect and enters the FROZEN state.

2.2 Closing a Connection

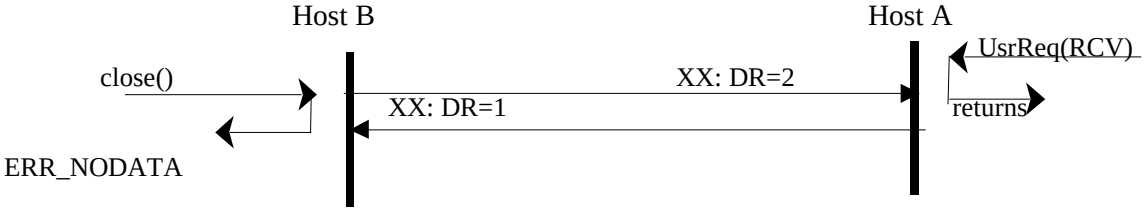


Figure 4: Connection Termination.

The connection can be closed at any time by either endpoint, as shown in Figure 2. When a application process calls `µTPClose()`, the µTP client or server sends a Disconnect PDU, and periodically retransmits it until Disconnect is received or a timer expires. If a Disconnect PDU is received, the close is “clean”, otherwise it is “dirty”. Both connection endpoints may call `µTPClose()`, causing Disconnect PDUs to cross the network, which results in a “clean” close at both ends. When an endpoint has either received Disconnect or sent Disconnect several times without receiving a reply, it enters the FROZEN state. Closing is preemptive: As soon as one end closes, all data transfer stops, any in−transit data are lost, and both ends are frozen for a time period equal to twice the Maximum Packet Lifetime (MPL) of the underlying network service. This algorithm ensures that all messages from the connection have drained from the network before the reference number is reused, and is referred to as “2−MPL timeout”. Figure 4 shows the messages exchanged during “clean” connection termination.

3 Data Transfer

As the state diagram shows, data transfer occurs only in certain states. An endpoint can send data when it is in the OPEN state and receive data only in the OPEN or POPNG state. This section contains a description of procedures and state variables for transfer in one direction from the sender to receiver. However, each end can act simultaneously as sender and receiver. µTP maintains one−to−one correspondence between Service Data Units (SDUs) and Data PDUs. An SDU is a group of bytes submitted by the application. A Data PDU is a data packet transmitted over the network. Each data carrying a PDU contains exactly one SDU. This requirement simplifies the implementation of the protocol, but it limits the size of SDU to MAX_µTP_DATA.

3.1 State Variables

The sender maintains the following state variables per connection for the implementation of the dynamic sliding−window algorithm.

- `send_unack`: least sequence number such that an Ack PDU containing that sequence number has not been received.
- `send_next`: least sequence number that has not been transmitted in a Data PDU.
- `send_lim`: sum of `send_unack` and current send window size.
- `send_acc`: total number of SDUs accepted from the application process.
- `send_ack_subseq`: subsequence number of the Ack message from which `send_lim` was most recently set.

All of the above variables, except `send_lim,` are initialized to zero during connection establishment. `Send_lim` is initialized to the value in the initial window field of the received Connect Ack or Connect Request PDU.

The receiver maintains the following state variables per connection:

- `rcv_next`: least sequence number such that a Data PDU with that sequence number has not been received.
- `rcv_win`: number of PDUs for which buffer space is currently available (includes buffers currently in use to hold PDUs with sequence numbers in the window but greater than `rcv_next`).
- `rcv_ack_cnt`: number of times the current value of `rcv_next` has been transmitted in an Ack PDU.

These variables, except `rcv_win,` are initialized to zero. The value of `rcv_win` is a parameter of µTP.

3.2 Getting the data from process A to process B

Similar to specifying SOCK_DGRAM for UDP and SOCK_STREAM for TCP, both endpoints specify SOCK_µTP_STREAM for µTP as the parameter for `socket()`. We assume that both the processes have already created sockets and are connected to each other via `µTPOpen()`. We restrict our description to one µTP connection. The µTP calls (API) used in this section are explained later in Section 5. The protocol control block (PCB) contains the data structures used

for each connection. The data structures includes a reference number, a pointer to a socket, all the state variables, the state of the connection, and various pointers to buffers. Consider that data is to be sent from process A to process B. Process A contains the following fragment of code:

```
µTP_UserReq (ref_no, SEND, data, length)
```

This code calls the kernel function `sys_µTP_UserReq()`, which tests for a number of conditions including length of data and read−access rights to the memory area referred to by the data. It also transfers the data from user space to the kernel space using `copy_from_user()`. It calls i`net_sendmsg(),` which tests for validity of destination address and calls µTP's send operation `µTP_send()`. `µTP_send()` builds the µTP header and calls the function `ip_queue_xmit()`. `Ip_queue_xmit()` builds the protocol header, slots the packet into the wait queue of packets ready to be transferred, and calls `dev_queue_xmit()`. If we assume that the device is an Ethernet card, `dev_queue_xmit()` calls `ei_start_xmit()`, passes the data to the network adapter, and sends it to the Ethernet.

After receiving the Ethernet packet, the network card triggers an interrupt that is handled by `ei_interrupt()`. If the transfer completes without error, `ei_interrupt()` calls `ei_receive()` with a reference to a network device. E`i_receive()` writes the packet to a newly set−up buffer using `wd_block_input()`. Then e`i_receive()` calls `netif_rx()`, which adds the packet to the backlog list. There is only one list in the entire system that contains all the packets received by the system. All the functions described so far for receiving packets are executed within the interrupt context. The `netif_rx()` function then marks the network implementation's bottom−half routine in the bottom−half mask `bh_mask`. `Net_bh()` calls `ip_rcv()`, which checks the header for correctness. If necessary, `ip_rcv()` executes the handling routines for the IP options. Since µTP is specified in the protocol field of the IP header, `ip_rcv()` calls `µTP_rcv()`. `µTP_rcv()` calls `µTP_lookup()` to determine, by reference to the sender and destination addresses and the sender and destination port numbers, the INET socket to which the µTP segment is addressed. It then calls `µTP_input()`, which enters the buffer in the list of data received for the packet and sends an acknowledgment if the sequence number of this packet is equal to the next expected sequence number (positive acknowledgment).

If process B wishes to receive the data sent by A, it executes a receive operation:

```
µTP_UserReq(ref_no, RECV, data, length)
```

`µTP_UserReq` calls `sys_µTP_UserReq`, tests for various conditions, gets the data from buffer and transfers the data to user space.

3.3 µTP Sender Procedure

The sender maintains two linked lists: `sent_list` and `unsent_list`. The sender accepts the SDU from the application process. The sender constructs a PDU from the SDU by adding a header and computing the checksum. The sender assigns the value `send_acc` as the PDU's sequence number and increments `send_acc`. If the send window is not closed, the sender transmits the PDU, keeping a copy of each transmitted PDU in `sent_list` until acknowledged by the receiver. The send window is the segment of sequence numbers beginning at `send_unack` and extending up to, but not including, `send_lim`. The send window is the sender's view of the receive window, based on the information received in Ack PDUs. The

sender closes send window when send_unack equals send_lim and keeps a copy of each PDU that is not transmitted in unsent_list. Whenever the send window opens, the sender moves PDUs from unsent_list to sent_list and transmits the PDUs. Buffer space to hold PDUs may occasionally become scarce, and the sender may have to stop or delay accepting new PDUs from the application process for transmission.

After receiving an Ack PDU from the receiver, the sender first determines whether it contains new information. An Ack PDU contains new information if it satisfies either of the following conditions:

- The value in its Ack sequence number field is greater than send_unack, or
- Its Ack sequence number is equal to send_unack, and the value in its Ack subsequence number field is greater than send_ack_subseq.

Receiving an Ack that contains new information, the sender updates the following state variables:

- send_unack to the value of the Ack sequence number field,
- send_ack_subseq to the value of the Ack subsequence number field, and
- send_lim to the sum of send_una (the new value) and the value in the window size field of the Ack PDU.

After updating its state variables, the sender deletes all PDUs in sent_list that are no longer pending. A data PDU is pending if its sequence number is greater than or equal to send_unack. If a PDU remains pending for longer than a retransmit timeout period, the sender retransmits it. Round−trip time calculation decides the time to wait for retransmission. Whenever there are pending PDUs and the window is closed, the sender periodically probes the receiver by sending a probe PDU, which elicits an immediate Ack PDU in reply. Probing ensures that the sender receives an updated flow control window information.

3.4 µTP Receiver Procedure

The receive window is a contiguous set of rcv_win sequence numbers beginning with rcv_next. The receiver buffers any data PDU it receives with a sequence number within the receive window. The receiver drops all the PDUs with sequence numbers not in the window. The receiver delivers a buffered PDU to the application process only after receiving and delivering all lower numbered PDUs. The number of buffers available for incoming PDUs may vary with time according to how fast the application process accepts data PDUs. The state variable rcv_win reflects the amount of buffering available; in particular, rcv_win is zero when the receiver has no buffers available for incoming PDUs. The receiver modifies the values of rcv_next and rcv_win in response to two events: receipt of new data PDUs and change of availability of buffers (if data from some buffered PDU has been delivered to the application process). The receiver maintains two lists: deliverable_list and undeliverable_list. When the receiver receives the PDU with sequence number equal to rcv_next, it does the following:

- Check for a valid checksum (includes both data and header),
- Set rcv_next to the lowest sequence number such that a data PDU with that number is not currently buffered,
- Decrease rcv_win to account for the buffers occupied by the recently received PDU plus any PDUs that were received (out of order) before it,
- Insert the PDU in deliverable_list ,

- Transmit an Ack, and
- Check any PDU that can be moved from `undeliverable_list` to `deliverable_list`.

If the sequence number of the PDU is not equal to `rcv_next` but is within the limits of the receive window, the receiver inserts that PDU in `undeliverable_list`. The receiver frees the buffers by delivery of data to the application process and increases `rcv_win`. The receiver modifies the value of `rcv_ack_cnt` whenever it transmits an Ack PDU or `rcv_next` changes. The receiver increments `rcv_ack_cnt` each time it sends an Ack PDU for the same `rcv_next`. When the value of `rcv_next` changes, `rcv_ack_cnt` is set to 0. Thus among Ack PDUs with the same Ack sequence field, the one with the highest Ack subsequence number contains the most recent window information. The sender uses this fact to determine whether an incoming Ack PDU contains new window information. The receiver transmits an Ack in the following cases:

- The arriving PDU's sequence number is equal to `rcv_next` .
- The receiver gets a Probe PDU.
- The window size increases from zero (the window reopens).

3.5 Example Exchanges

Some sample exchanges are shown in this section.
DR: Destination Reference OP: Originating Port AK: Ack Sequence SN: Sequence Number
OR:Originating Reference DP: Destination Port AS: Ack Subsequence W: Window size
RR: Responding Reference

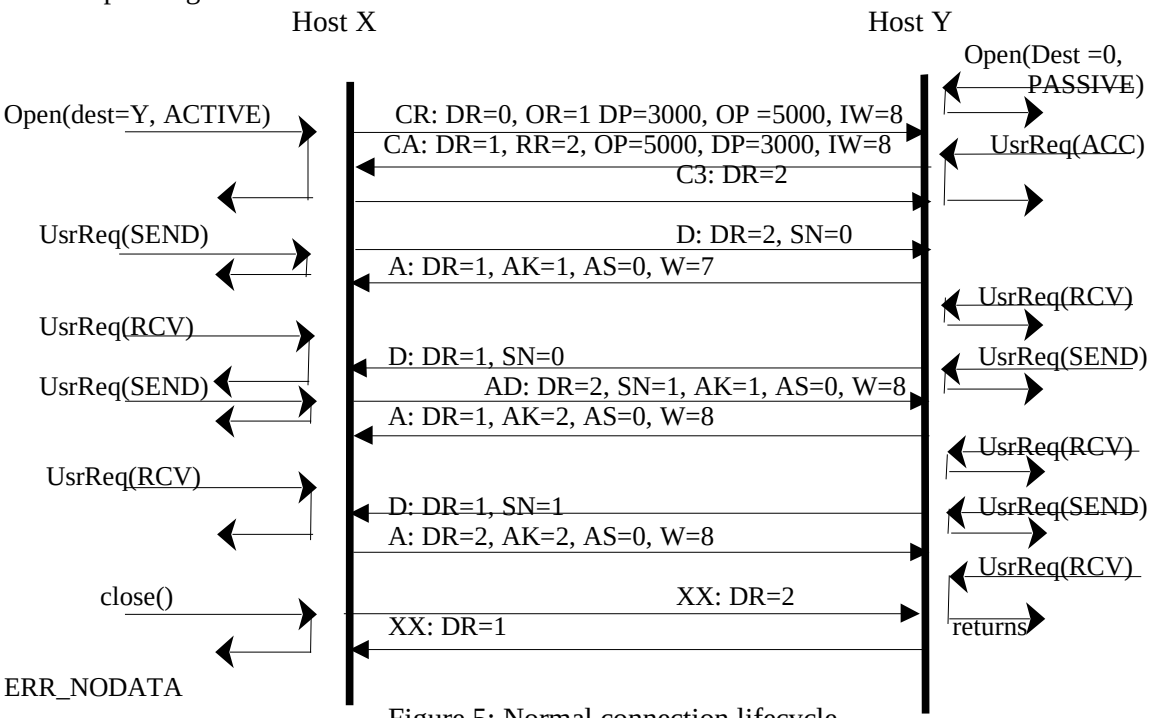


Figure 5: Normal connection lifecycle

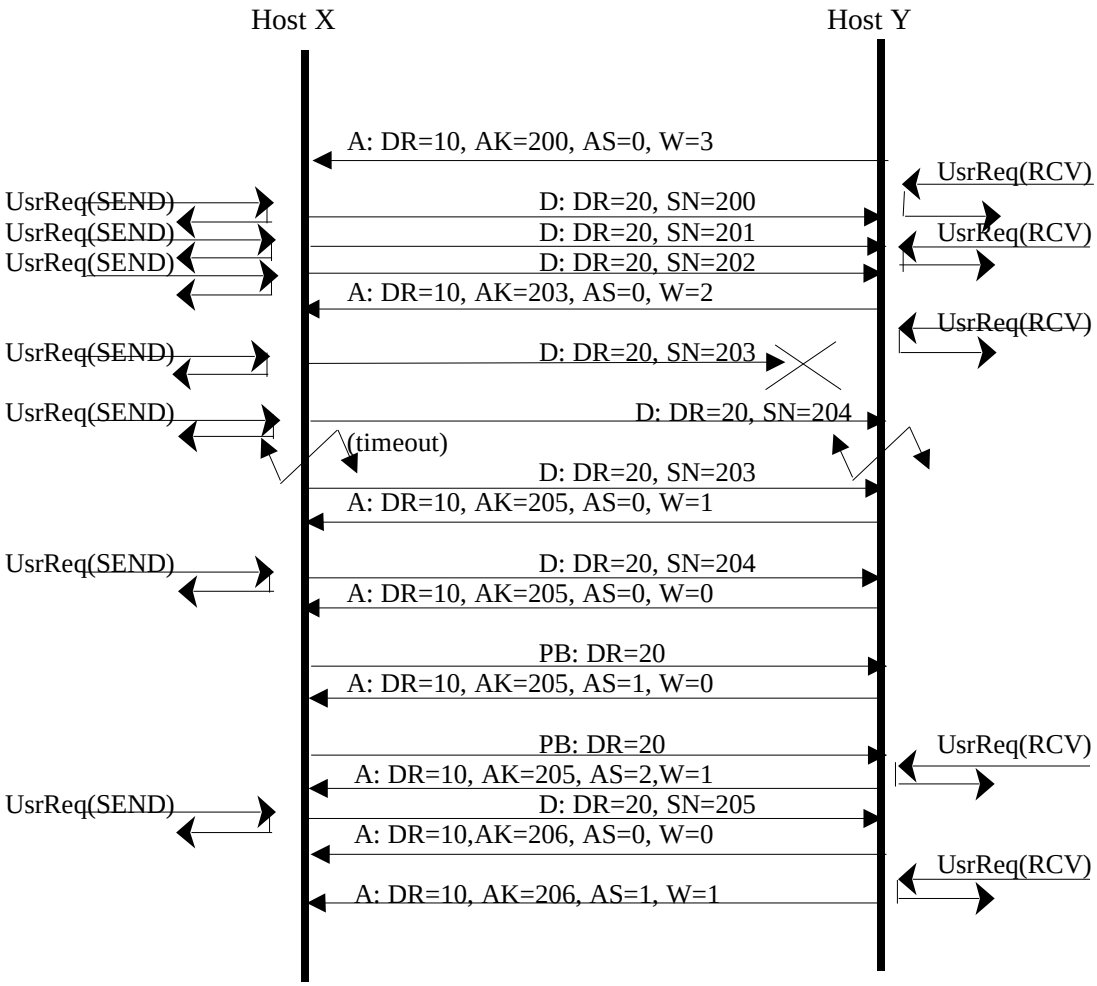


Figure 6: Error recovery and window flow control

4 Interfaces to µTP

This section describes the interface to the application layer (system calls) and to the lower layer (IP calls). It explains the parameters and return values of the interfaces.

4.1 Upper Interface

The Upper Interface to µTP consists of three system calls:

`µTP_Open()`: first call for connection establishment.
`µTP_Close()`: call for terminating the connection.
`µTP_UserReq()`: call for send, receive, accept and other requests.

4.1.1 The Open Function

To request connection−establishment, the application process first calls `µTPOpen()`. `µTPOpen()` returns the connection reference number or error value (if negative). Its prototype is:

```
int µTPOpen(int µTPSocket,     /* socket fd associated with the connection */
        Addr_t *desthost,      /* destination host address */
        µTPPorts_t ports,      /* local and destination µTP port number */
        int apflag)           /* ACTIVE/PASSIVE flag */
```

`µTPOpen()` does the following:

- Allocates a new protocol control block (PCB) for the connection.
- Checks for one of the following conditions:
  - If the active/passive parameter is ACTIVE, both destination address and destination port should be nonzero.
  - If the active/passive parameter is PASSIVE, the local port should be nonzero.
- Copies the destination address and port parameters into the PCB.

The rest of the `µTPOpen` processing depends on the value of the A/P flag:

If the A/P parameter is ACTIVE, `µTPOpen()` sets the state to AOPNG, sends the initial Connect Request PDU, starts the retransmission timer, and blocks until the state becomes OPEN or FROZEN. If the connection state becomes OPEN, it returns an identifier to be used as a parameter in subsequent calls to interface routines. This connection identifier uniquely identifies the protocol control block.

If the A/P parameter is PASSIVE, `µTP_Open()` sets the state to LSTNG and returns the connection identifier. The PCB allocated in response to a PASSIVE Open request is not associated with any particular connection. The application process can subsequently obtain new connections using this PCB. After a PASSIVE Open, the application process calls `UsrReq(ACCEPT)` to obtain the connection identifiers associated with particular connection.

4.1.2 The Close Function

The application process calls µTPClose() to abruptly terminate a connection. Its prototype is:

```
int µTPClose(int connection_id)  /* connection identifier */
```

A call to close an OPEN connection changes the state to PCLSNG and causes a Disconnect PDU to be sent and repeatedly retransmitted. The process blocks until endpoint receives a Disconnect PDU or transmits a Disconnect PDU number of times (equal to MAX_RETRY) with no response. The return value of µTP_Open() indicates whether the termination is "clean" (the endpoint receives a Disconnect PDU in response to the one sent). If the endpoint receives no Disconnect reply, the termination is dirty and µTPClose() returns an error value.

4.1.3 The UserReq Function

This function handles requests from the application process to send and receive data, to inquire about the status of a connection and to accept new connections after a PASSIVE Open. Its prototype is:

```
int UserReq(int connection_id,  /* connection identifier */
            int command,        /* identifies the type of request */
            void *data,         /* pointer to data or status buffer */
            int *dlen)          /* size of data sent/received */
```

The connection identifier is obtained from the earlier µTPOpen call. Command is an integer identifying the operation to be performed on behalf of the application process. Possible values are:

```
ACCEPT      /* accept connection (may block) */
NBACCEPT    /* non blocking accept */
SEND        /* send data (may block) */
RCV         /* receive data (may block) */
NBRCV       /* non blocking receive */
STAT        /* status inquiry */
```

The meanings of pointer to data (*data) and pointer to length (*dlen) parameters vary with the command. The input and output semantics of data and dlen for each command are summarized in the following table:

	***data**		***dlen**	
Command	**Input**	**output**	**Input**	**output**
ACCEPT, NBACCEPT	(ignored)	(undef)	(ignored)	New conn id
SEND	Data to send	= Input	# bytes	# sent
RCV, NBRCV	Input buffer	= Input	Buffer size	# recd
STAT	µTP_stat_struct	= Input	Size of struct	Size of struct

Table 1: Input and Output parameters of UserReq()

The ACCEPT command causes dlen to point to a new connection id that identifies a remotely initiated connection to the local port. If no such connection is pending, the call blocks until a

connection reaches the OPEN state.

The NBACCEPT command similarly returns a connection identifier via `dlen`. If no connection has reached the OPEN state, the call returns ERR_NODATA rather than blocking.

The SEND command requests that an SDU be delivered to the remote application process. `UserReq()` blocks or delays if several PDUs sent are not yet accepted by the remote application process. The number of bytes in the request is limited to maximum data size in the PDU. It returns the number of bytes actually transmitted via `dlen`.

The RCV command places data sent by the remote application process in the `data` buffer. The buffer supplied should be large enough (as indicated by `dlen` on input) to pass the data to the application process. The command may block if no data is currently deliverable. It returns the number of bytes actually received via `dlen`.

The NBRCV command similarly places data sent by the remote application process in a single SEND request in the indicated buffer. However, if no PDU is deliverable at the time of the call, `UserReq` returns ERR_NODATA instead of blocking.

The STAT command provides the application process with a snapshot of the state of the connection, filling in a structure with information about sequence number and state variables. The `µTP_stat` structure is defined as follows:

```
struct µTP_stat {
      int µTP_st,                 /* connection state */
      int µTP_sacc,               /* value of send acc */
      unsigned short µTP_snxt,    /* value of send next */
      unsigned short µTP_suna,    /* value of send una */
      unsigned short µTP_sndlim, /* value of send lim */
      unsigned short µTP_rnxt,    /* value of rcv next */
}
```

4.2 Lower Interface

The interface between µTP and the internet protocol (IP) is defined in terms of two functions:

```
void ip_queue_xmit (struct skbuff *skbuff);
int ip_rcv(struct sk_buff *skb, struct device *dev,
                      struct packet_type *pt);
```

`Ip_queue_xmit()`is an output function used for transmitting data. After adding µTP header to the data, `µTP_send()` invokes `ip_queue_xmit()` which builds the IP header and calculates checksum. It queues the packet to be sent and transmits if necessary.

When the µTP packet arrives, `net_bh()` invokes `ip_rcv()`. It validates the protocol header and determines if the packet is addressed to the local host. It then invokes `µTP_rcv()`.

5 Retransmission Timers and Round Trip Estimation

µTP uses kernel timers for the following purposes :

- To handle retransmission of data PDUs,
- To handle retransmission of Connect Request, Connect Ack or Disconnect PDUs,
- For a 2−MSL delay following the `µTpclose()`, and
- To probe the receiver when the sender is unable to transmit the data.

The timeout value (in jiffies) and the function to be called when the timer expires characterizes the kernel timers. The timer handler receives an argument, which is stored in the data structure, together with a pointer to the handler itself.

The data structure of a timer is defined as:

```
struct timer_list {
      struct timer_list *next;          /* pointer to next struct */
      struct timer_list *prev;          /* pointer to prev struct */
      unsigned long expires;            /* the timeout, in jiffies */
      unsigned long data;               /* argument of the handler */
      void (*function)(unsigned long);/*  timeout handler */
}
```

µTP uses the following timer routines:

`void init_timer(struct timer_list *timer);`
This inline function is used to initialize the timer structure.

`void add_timer(struct timer_list *timer);`
This function inserts a timer into the global list of active timers. As an argument for `add_timer()`, `expires` specifies the timeout value. When the timer expires, the kernel invokes the handler.

`void del_timer(struct timer_list *timer);`
If a timer needs to be removed from the list before it expires, µTP calls `del_timer()`.

Before transmitting the PDU, the sender puts the PDU in the `sent_list` and turns on the retransmission timer. There is only one retransmission timer. If the timer expires, the sender retransmits the PDU and turns on the timer. But if an acknowledgment arrives before the timer expires, the sender checks the `send_list`. If there are pending PDUs in `sent_list` awaiting acknowledgment, the sender turns on the timer. When the timer expires, the sender retransmits the first PDU from the `sent_list`. This algorithm reduces the number of timers and retransmissions.

During normal data transfer, an acknowledgment arrives for each PDU before the retransmission timer expires. The retransmission timeout for the PDU to be transmitted is equal to the round−trip time taken by the previous normal data transfer (without any retransmissions). µTP ignores round−trip measurements of retransmitted PDUs (similar to Karn's algorithm in TCP).

6 Congestion Avoidance and Control

When congestion occurs, delays increase, causing µTP to retransmit PDUs. In the worst case, retransmissions increase congestion and produce an effect known as congestion collapse. To avoid adding to congestion, µTP uses a multiplicative decrease technique and uses slow start during recovery.

The sender side of µTP maintains a variable known as `congestion_window` to restrict the amount of data being sent. When transmitting, µTP uses the minimum of the receiver's window size and the `congestion_window` to determine how much data to send.

The sender initializes the `congestion_window` to the receiver's window size. When congestion begins (a retransmission timer expires), the sender reduces the `congestion_window by` half. Each time retransmission occurs, the sender reduces the `congestion_window` by half until it becomes 1. This technique is known as multiplicative decrease.

Slow start is the reverse of multiplicative decrease. If a communication is successful and an acknowledgment arrives before the retransmission timer expires, the sender increases `congestion_window` by one until it reaches the threshold set by multiplicative decrease. Then the sender increases the `congestion_window` by one for every round trip.

7 Protocol Data Units (PDUs)

PDU NAME	CODE	CONNECT	DATA TRANSFER	DISCONNECT
Connect Request	CR	X		
Connect Ack	CA	X		
Connect Confirm	C3	X		
Data	bD	X	X	
Ack	Ab		X	
Data+Ack	AD		X	
Probe	PB		X	
Disconnect	XX			X

Table 2 : µTP−PDUs and their functions

Table 1 lists all the µTP PDUs and the function with which each is associated. Headers in µTP consist of an eight−byte common part, which has the same format for all PDUs, followed by a PDU−specific part, whose length and format varies. The format of the PDU−specific part is determined by the “PDU type” field in the common part. All µTP headers consist of an integral number of four−byte words, which is useful because the data can be manipulated as a word (four bytes) at a time, particularly to compute checksums.

7.1 Common Header Part of PDU

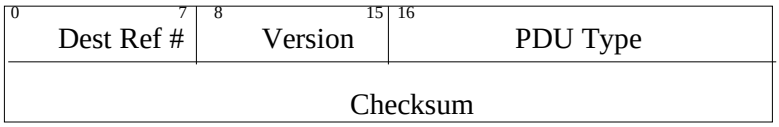


Figure 7: Common Part of PDU Header

The header part common to every µTP PDU is shown in Figure 5. It contains the following four fields:

- Destination Reference Number (8 bits) is the reference number associated with the connection endpoint intended to receive this PDU. It is always nonzero except in the Connect Request PDU, where it is always zero.

- Version (8 bits) identifies the version of µTP implemented. According to the specification, packets for the current implementation must have the value 2 in this field.

- PDU Type (16 bits) identifies the type of the PDU. It is formatted as two one-byte subfields, each containing an ASCII character.

- Checksum (32 bits) is bitwise sum of all the 32-bit words in the PDU (header + data).

7.2 Connection−Establishment PDUs

During connection establishment, the ACTIVE end (client) transmits a Connect Request (CR) PDU; the PASSIVE end (server) replies with Connect Ack (CA). The ACTIVE end completes the three−way handshake by sending either a Connect Confirm (C3) or a Data (bD) PDU.

7.2.1 Connect Request PDU

The Connect Request PDU format is shown in Figure 6.

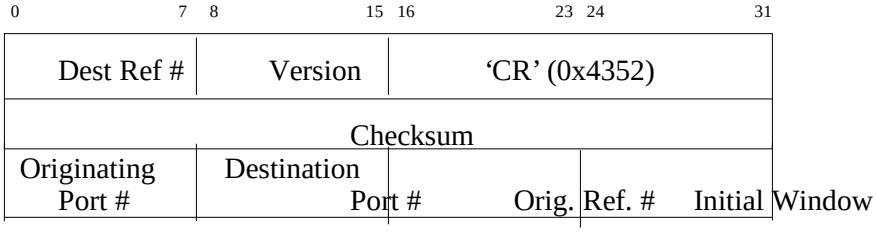


Figure 8: Connect Request PDU format

- The Destination Reference field of the Connect Request PDU always contains 0.
- The PDU Type field contains the ASCII codes for the two characters 'C' and 'R'.
- Originating Port (8 bits) is the port associated with the process ACTIVEly opening the connection.
- Destination Port (8 bits) is the port associated with the process at the PASSIVE end.
- Originating Reference Number (8 bits) is chosen by the ACTIVE end as a local identifier for this connection. It becomes the Destination Reference Number of all PDUs sent on this connection by the PASSIVE end.
- Initial Window Size (8 bits) is used to initialize the window mechanism. It may be zero.

7.2.2 Connect Ack PDU

The PASSIVE end sends the Connect Ack PDU in response to a received Connect Request PDU.

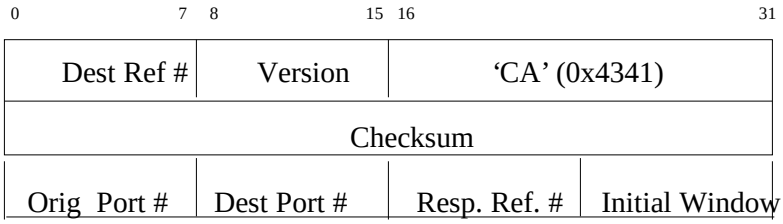


Figure 9: Connect Ack PDU format

- The PASSIVE end sends a Connect Ack PDU in response to a received Connect Request PDU.
- The Destination Reference Number contains the value of the Originating Reference Number field from the requesting PDU.
- The PDU Type field contains the ASCII codes for the two characters ‘C’ and ‘A’.
- Originating Port (8 bits) is the same as in the initiating Connect Request PDU, and so is the Destination Port. Thus the Originating Port always refers to the port on the ACTIVE end, and the Destination Port always refers to the PASSIVE end.
- Responding Reference (8 bits) is a local identifier chosen by the PASSIVE end. It becomes the Destination Reference Number for all PDUs sent by the ACTIVE end on this connection.
- Initial Window (8 bits) is used to initialize the window mechanism. It may be zero.

7.2.3 Connect Confirm PDU

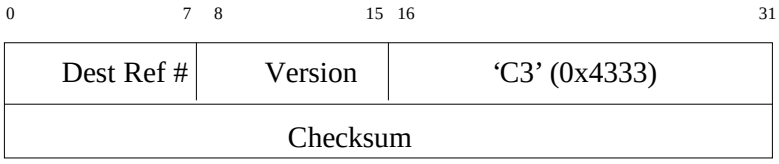


Figure 10: Connect Confirm PDU format

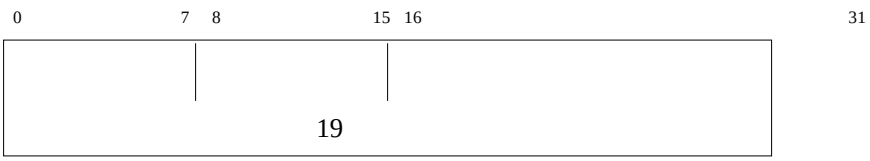
The ACTIVE end sends Connect Confirm PDU in response to a received Connect Ack PDU. It is the third message of the three−way handshake. Whenever the ACTIVE end has no user data to send or the initial flow control window specified (in the Connect Ack PDU) by the PASSIVE end is zero, it sends this message. The Destination Reference Number is the value from the Responding Reference Number field of the received Connect Ack PDU.

7.3 Data Transfer PDUs

Application data is transferred in PDUs of type Data or Data+Ack. The protocol provides for independent data flow in each direction. The description of this section refers to a single direction. Each data carrying PDU has an associated 16-bit sequence number.

7.3.1 Data PDU

The format of the data PDU is shown in Figure 9. The data contained in a Data PDU is a single SDU, that is, the amount of data passed by the application process in a single send request.



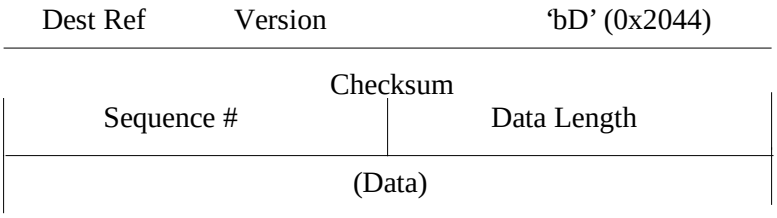


Figure 11: Data PDU format

- The PDU Type field contains the two characters 'bD', i.e. the first byte contains the ASCII character space.

- Sequence Number (16 bits) assigned to Data and Data+Ack PDUs come from the same sequence, e.g. if a Data PDU is sent with sequence number x, and the next PDU is a Data+Ack PDU, its sequence number is x + 1. The maximum amount of application process data is that can be carried in single PDU specified by the constant MAX_µTP_DATA.

- The checksum in a Data (or Data+Ack) PDU includes both header and data.

7.3.2 Ack PDU

The receiver sends the Ack PDU. The Ack PDU contains the sequence number of the next data carrying PDU expected by the receiver. Its format is shown in Figure 10.

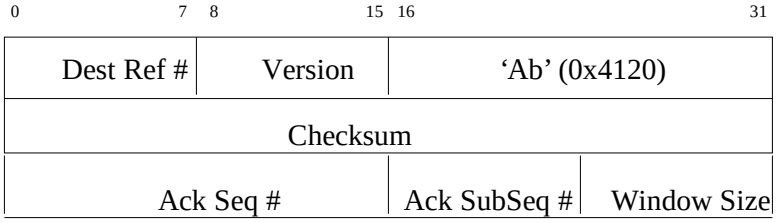


Figure 12: Ack PDU format

- The PDU Type field contains the ASCII characters 'Ab', i.e. the second byte of the field contains the space character.

- The Ack Sequence Number carried in the Ack PDU is the number of the next PDU the receiver expects to receive.

- The Window Size indicates the number of data carrying PDUs for which the receiver has buffer space, including the PDU whose number is in the Ack Sequence Number field. This indication is useful for implementation of the dynamic sliding−window protocol. The sender adjusts its window size according to the receiver's window size.

- The Ack Subsequence Number is used in the flow control function. Its purpose is to enable the sender to distinguish between the flow control parameters contained in different Ack or Data+Ack PDUs that acknowledge the same Data PDU.

7.3.3 Data+Ack PDU

The Data+Ack PDU allows acknowledgments to be piggybacked with data flowing in the opposite direction. The header of the Data+Ack PDU contains the fields of both the Ack and Data PDUs, as shown in Figure 11.

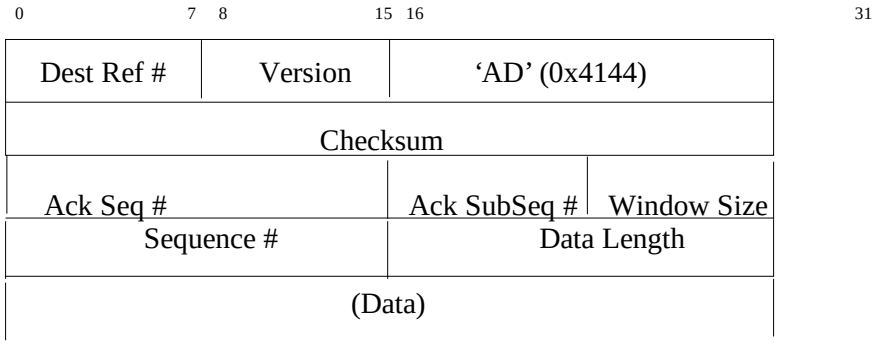


Figure 13: Data+Ack PDU format

7.3.4 Probe PDU

The Probe PDU can be used to elicit an Ack from the receiver. In particular, it is transmitted by a sender when one or more data PDUs are pending, the window is closed, and no Ack PDU has been received for some time. The format of the Probe PDU is shown in Figure 12.

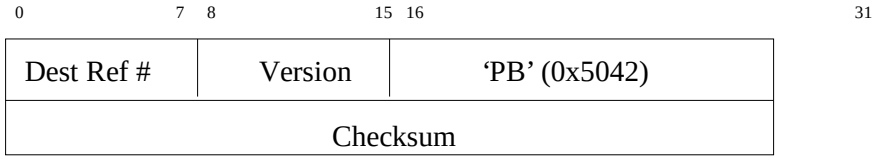


Figure 14: Probe PDU format

The receiver replies to Probe immediately by sending Ack.

7.4 Disconnect PDU

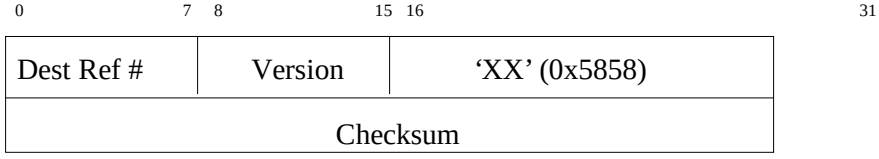


Figure 15: Disconnect PDU format

The Disconnect PDU (Figure 10) causes all communication on a connection to cease, immediately. Whenever an endpoint receives Disconnect PDU on an OPEN connection, it transmits a Disconnect in reply, and the connection enters the FROZEN state. All subsequent requests return an error condition, and all incoming PDUs bearing the destination reference number are discarded. In addition, a Disconnect may be sent under the following conditions:

- In response to a Connect Request PDU when it is not possible to establish the requested connection (if no process is listening on the specified port, or if no reference numbers are available at the host for new connections).

- In response to a Connect Ack PDU when no Connect Request has been sent.

- In response to any PDU (other than Disconnect) whose Destination Reference field refers to a connection in the FROZEN state. The Disconnect PDU must not be sent in reply to a Disconnect received in the FROZEN state, because this behavior, coupled with loss of a Disconnect PDU, can cause the two ends to shoot Disconnects at each other for the entire 2 MPL time.

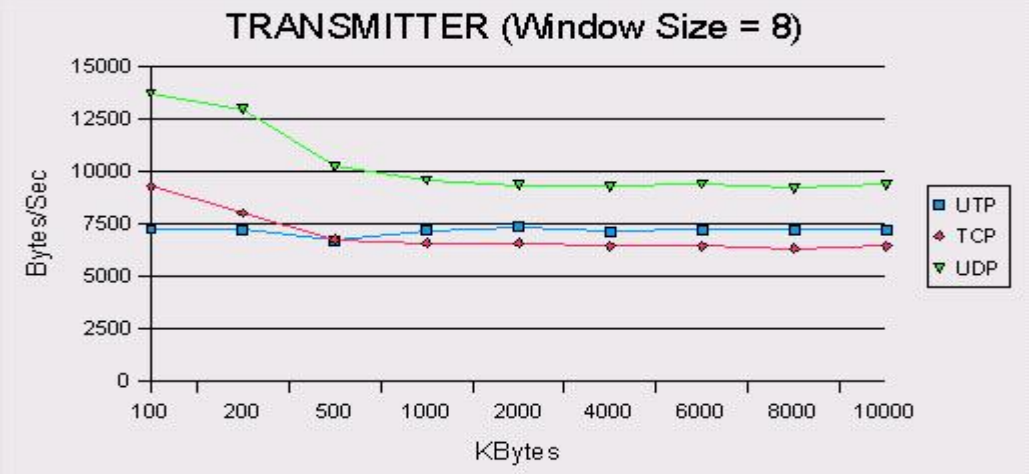
8 Performance Measurement and Comparison

8.1 Test

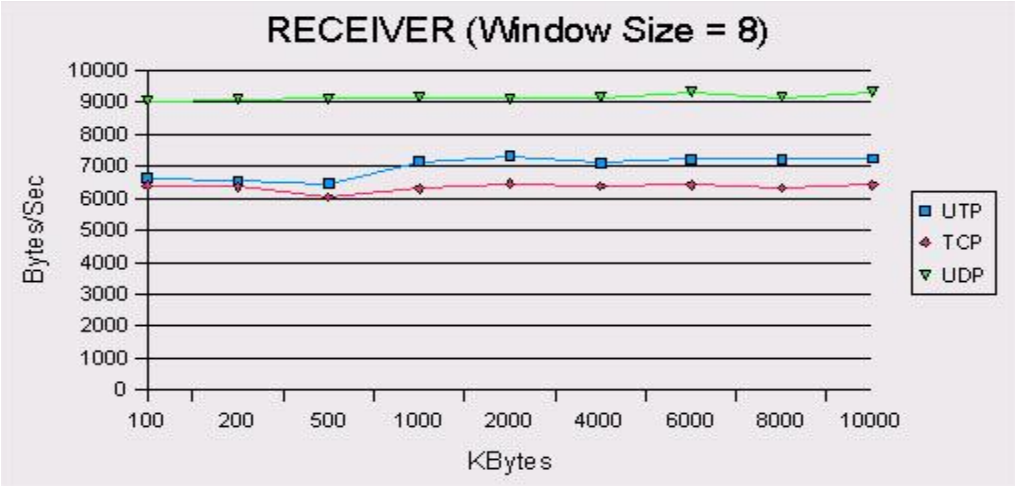
The performance measurement tests measure the throughput between two hosts running µTP on Linux 2.2.10. I implemented TµTP (Test µTP), an application program similar to TTCP (Test TCP), for measuring and comparing the performance. The performance tests measure the throughput for data ranging from 100 KB to 10 MB for window size of 8 and 16. The packet size is constant (4000 bytes). The graphs are plotted to compare the throughput of µTP with TCP and UDP. The performance tests measure the throughput 3 times for each combination of parameters and takes average of the results. To make a valid comparison, the tests adjust the TCP's window size according to µTP's window size. The window size of TCP is 32120 bytes when µTP's window size is 8 (8*4000 = 32000 bytes). It is double that (64240 bytes) when the window size is 16 for µTP (16*4000 = 64000 bytes). Both hosts are on a directly connected network (the connection doesn't pass through a gateway). The clock speed of the transmitter is 133 MHz and that of the receiver is 450 MHz. The following graphs display the performance of µTP, TCP and UDP, plotting throughput (Bytes/Sec) on the Y−axis versus data size (Bytes) on the X−axis.

8.2 Discussion of results

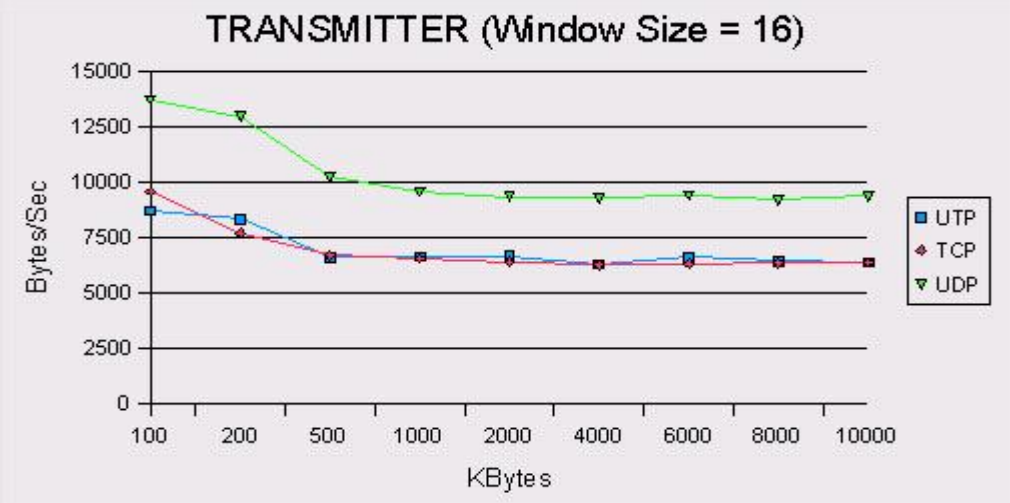
Graph 1 and 2 display the receiver's and transmitter's performance of transport layer protocols for window size 8. Similarly, Graphs 3 and 4 display the performance for window size 16. The graphs show that unreliable UDP's throughput is more than TCP or µTP for window size 8 and 16. For window size 8 (Graphs 1 and 2), µTP outperforms TCP. But for window size of 16 (Graphs 3 and 4), the throughput of TCP is similar to that of µTP. If the hosts are not directly connected on the same network, these results may be different. I expect µTP to outperform TCP if the traffic is not too congested, since µTP minimizes the number of retransmissions and the header size is small. But if there is congestion, TCP might outperform µTP, since it uses exponential backoff and Karn's algorithm to compute round−trip time. Karn and Partridge [August 1997] describe Karn's algorithm and estimation of round−trip times.



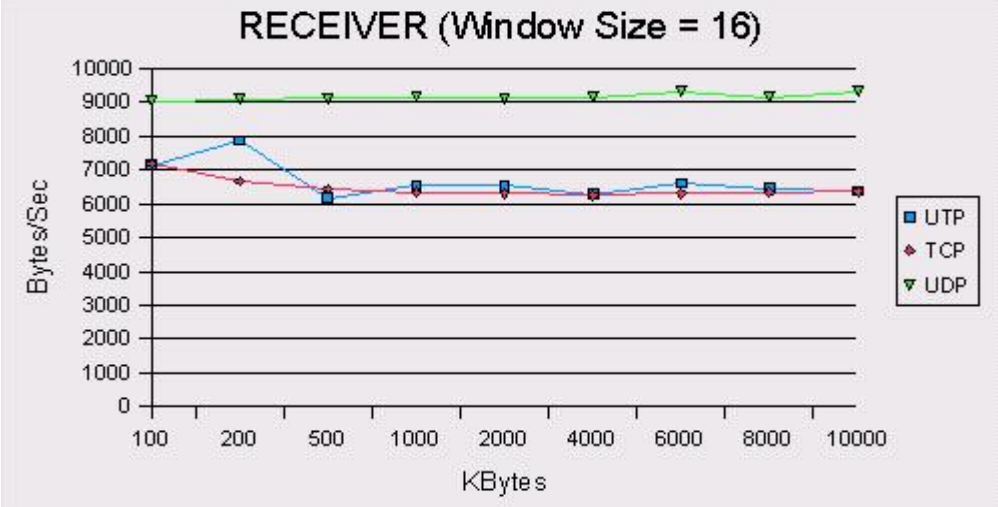
Graph 1: Transmitter (Window Size = 8)



Graph 2: Receiver (Window Size = 8)



Graph 3: Transmitter (Window Size =16)



Graph 4: Receiver (Window Size = 16)

9 Conclusion

I successfully implemented µTP in Linux 2.2.10. µTP provides reliable stream delivery. It provides full−duplex connection between two hosts, allowing them to exchange large volumes of data efficiently. µTP makes efficient use of the network by using a dynamic sliding window protocol and flow control. It uses adaptive round−trip time calculation for retransmission timers. It also uses slow start and multiplicative decrease to avoid congestion collapse. Measurements show that the throughput of µTP is higher if it transmits larger blocks. They also show that µTP performs well at window size 8, where it has higher throughput than TCP on the same network.

References

- Calvert K. L. [1995] Micro Transport Protocol: Version 2 − Specification.
- Postel J. [1981] Transmission Control Protocol: RFC 0793
- Comer, D. E. [1991] Internetworking with TCP/IP: Volume 1: Principles, Protocols, and Architecture, Prentice−Hall.
- M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, D. Verworner [1998] Linux Kernel Internals: Second Edition, Addison−Wesley.
- P. Karn, C. Partridge [August 1987], Improving Round−Trip Time Estimates in Reliable Transport Protocols, Proceedings of ACM SIGCOMM' 87.