

The COP (connection oriented reliable packet stream) protocol

COP is a packet oriented transport protocol whose objective is to tame some aspects of the bad behavior of UDP.

Application

===Socket API - User/ OS interface ===

COP

IP

Link

COP data structures

Ring buffers -

Ring buffers are used to hold pointers to

- (1) buffers that have been transmitted but are awaiting acknowledgement
- (2) buffers containing packets that have been received out of order.

```
typedef struct cop_oge_type
{
    struct sk_buff *skb;          /* buffer address */
    unsigned long  txtime;       /* in jiffies      */
} cop_oge_t;
```

The ring buffers are stored in an array indexed by packet sequence number.

The cop_sock

The *cop_sock* is the principal structure controlling protocol operations. It is created in response to an application call to *socket(AF_INET, SOCK_COP, IPPROTO_COP)*; It is destroyed when the socket is closed.

```
typedef struct cop_sock
{
    struct inet_sock inet;
    struct sock      *sk;
    unsigned int     state;    /* Type of connection AD, FCD, etc */

    /* Network and transport header skeletons for fast copy */

    struct iphdr     iph;
    struct cop_hdr   coph;
}
```

```

/* Connection state data */

unsigned char  nxtsnd;          /* Next send seq num */
unsigned char  nxtrcv;        /* Next expected pkt */
unsigned char  lastack;       /* Last ack received */
unsigned char  win;           /* Offered window */
unsigned char  uwin;          /* Usable window */
unsigned char  dup_acks;      /* Duplicate ack ctr */
unsigned char  wuppend;       /* persist timer armed */
unsigned char  maxrxqlen;     /* Max offered window */
unsigned int   disable;       /* Prevents redundant fast retx */
struct timeval lasttx;        /* Time of last tx for wup */

/* These ring buffers are indexed by packet sequence numbers */
/* and used to hold pointers to transmitted buffers pending */
/* acks and out of order received packets. For 8 bit seq */
/* SEQ_SPACE_SIZE = 256 */

cop_wqe_t  ack_pend[SEQ_SPACE_SIZE];
cop_wqe_t  rcv_pend[SEQ_SPACE_SIZE];

```

```

/* Performance metrics */

unsigned long  timeouts;      /* Timeouts */
unsigned long  fastretx;     /* Fast retransmissions */
unsigned long  txdrops;      /* Synthesized drops */
unsigned long  rxdrops;      /* Drops for rcv queue */
unsigned long  txttotal;     /* Total tx data packets */
unsigned long  rxttotal;     /* Total rx data packets */
unsigned long  txwups;       /* WUP requests sent */
unsigned long  rxwups;       /* WUP acks received */
unsigned long  txacks;       /* ACKs sent */
unsigned long  rxacks;       /* ACKs received */
unsigned long  ooacks;       /* out of order acks */
unsigned long  tototal;      /* Total timeouts */
unsigned long  restarts;

/* Token bucket flow control data */

unsigned int   maxrate;       /* max number of packets per jiffy */
unsigned int   maxcredit;     /* max accumulated credits */
unsigned int   millicredits;  /* current number of millicredits */
int            tickintvl;     /* in jiffies */
struct timeval lasttick;     /* time of last tick... */

/* Timers */

struct timer_list resend_timer; /* timer for send retransmits */
struct timer_list token_timer;  /* timer token bucket timer */
struct timer_list persist_timer; /* persist timer */

/* Locks */

spinlock_t    cop_txlock;
spinlock_t    cop_loglock;
spinlock_t    cop_rxlock;
} cop_sock_t;

```

```
#define MAX_PORTS (64*1024)
#define START_PORT (32*1024)
#define COP_HTABLE_SIZE 64

unsigned char port_map[MAX_PORTS/8];

struct hlist_head cop_hash[NTP_HTABLE_SIZE];
rwlock_t cop_hash_lock = RW_LOCK_UNLOCKED;
```

The COP state machine

- The operation of COP (like other protocols) is controlled by a finite state machine
- Each COP socket exists in exactly one state at any point in time.
- There are *two sources of input* to the state machine:
 - The API functions (*bind()*, *listen()*, *connect()*, *sendto()*, *recvfrom()*, *connect()*, *close()*)
 - Receipt of specific *packet types*.

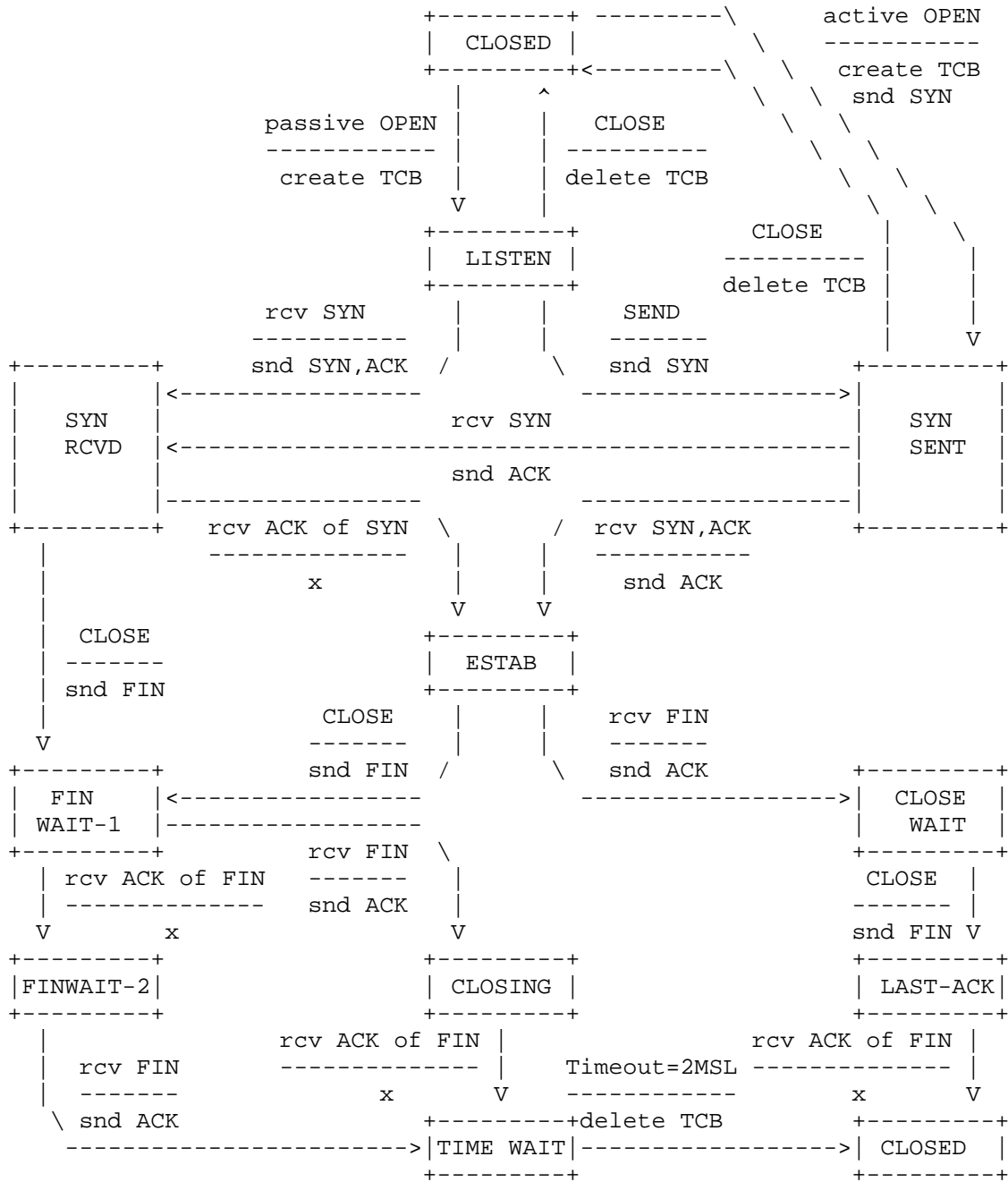
Associated with *each state* are a set of rules that determine

- What inputs (API function calls, or received packet types) are legal in the current state.
 - How to respond to legal inputs
 - How to respond to illegal inputs
- Response rules include
 - The identity of the next state (which can be the current state).
 - Packet type (if any) to send to the other end of the connection
 - Value that an API function should return to its caller.

The total number of rules for a fully specified system is:

- **number of states x number of inputs**

The TCP state machine



TCP Connection State Diagram

```
/* Header flag bits */

#define CPF_CONNECT    1    /* Connect request or connect ACK */
#define CPF_ACK        2    /* ACK number is valid */
#define CPF_FIN1       4    /* Close request */
#define CPF_FIN2       8    /* Close ACK */
#define CPF_WUP        16   /* Window update probe */
#define CPF_RESET      32
#define CPF_SUPV       64
```

Operation of the COP state machine -- The API as input source

One source of input to the state machine is the set of API functions.

- The table below summarizes the collection of *valid* inputs and actions.
- A robust system needs to handle *invalid* inputs as well.

<i>API call</i>	<i>Present state</i>	<i>Output state</i>	<i>Action</i>
<i>socket()</i>	<i>N/A</i>	<i>TCP_CLOSED</i>	no action
<i>listen()</i>	<i>TCP_CLOSED</i>	<i>TCP_LISTEN</i>	wait for connect
<i>connect()</i>	<i>TCP_CLOSED</i>	<i>TCP_SYN_SENT</i>	sendsupv(CPF_CONNECT) wait for connect ack.
<i>close()</i>	<i>TCP_ESTABLISHED</i>	<i>TCP_FIN_WAIT1</i>	drain output queue purge receive queue sendsupv(CPF_FIN1) wait for fin-ack. invoke common release
<i>send/rcv()</i>	<i>TCP_ESTABLISHED</i>	<i>TCP_ESTABLISHED</i>	no action
<i>rcv()</i>	<i>TCP_FIN_WAIT2</i>	<i>TCP_FIN_WAIT2</i>	if receive queue empty return(0)
<i>send()</i>	<i>TCP_FIN_WAIT2</i>	<i>TCP_FIN_WAIT2</i>	return(0)

State transitions and actions triggered by calls to the COP API

<i>Pkt Type</i>	<i>Present state</i>	<i>Output state</i>	<i>Action</i>
<i>CPT_DATA</i>	<i>TCP_ESTABLISHED</i>	<i>TCP_ESTABLISHED</i>	no action
<i>CPT_SYN1</i>	<i>TCP_LISTEN</i>	<i>TCP_ESTABLISHED</i>	sendsupv <i>CPF_CONNECT</i> <i>CPF_ACK</i>
<i>CPT_SYN2</i>	<i>TCP_SYN_SENT</i>	<i>TCP_ESTABLISHED</i>	no action
<i>CPT_ACK</i>	<i>TCP_ESTABLISHED</i>	<i>TCP_ESTABLISHED</i>	no action
<i>CPT_FIN1</i>	<i>TCP_ESTABLISHED</i>	<i>TCP_FIN_WAIT2</i>	send ACK of FIN if (recv_queue empty) set sk_shutdown() wakeup interruptible
<i>CPT_FIN2</i>	<i>TCP_FIN_WAIT1</i>	<i>TCP_CLOSE</i>	proceed with <i>sk_common_release()</i>
<i>CPT_RESET</i>	<i>TCP_ESTABLISHED</i>	<i>TCP_CLOSED</i>	drain rx and tx queue and shutdown socket.
<i>CPT_RESET</i>	<i>TCP_LISTEN</i>	<i>TCP_LISTEN</i>	no action

State transitions and actions triggered by calls to the COP API

Exception handling

We have addressed the handling of the normal sequence of events in the connection protocol. However, we also need to be able to handle abnormal/erroneous inputs from both the API above and unexpected packet types from below.

For incorrect API calls it is first necessary to determine if the call will be delivered to the COP layer or will be handled in the AF_INET layer. For those incorrect calls that are delivered to COP, a proper errno, must be returned.

	<i>_CLOSE</i>	<i>_LISTEN</i>	<i>_SYN_SENT</i>	<i>_ESTABLISHED</i>
bind()	1	2	3	4
connect()	5	1	2	3
listen()	4	5	1	2
send/rcv()	3	4	5	1

When incorrect or unexpected packet types are received, it may or may not be necessary to change state and it may or may not be necessary to emit an CFP_SUPV reply.

	<i>_CLOSE</i>	<i>_LISTEN</i>	<i>_SYN_SENT</i>	<i>_ESTABLISHED</i>
syn1	2	3	4	5
syn2	1	2	3	4
data	5	1	2	3
reset	4	5	A	A

Responsibility for defining a consistent and correct state machine will be distributed among the teams in as shown above. We will have a design review next thursday in which each team proposes its solutions for the 5 situations it is responsible for. Each team should prepare a written pdf document describing in detail the actions that should be taken in each situation for which they are responsible.