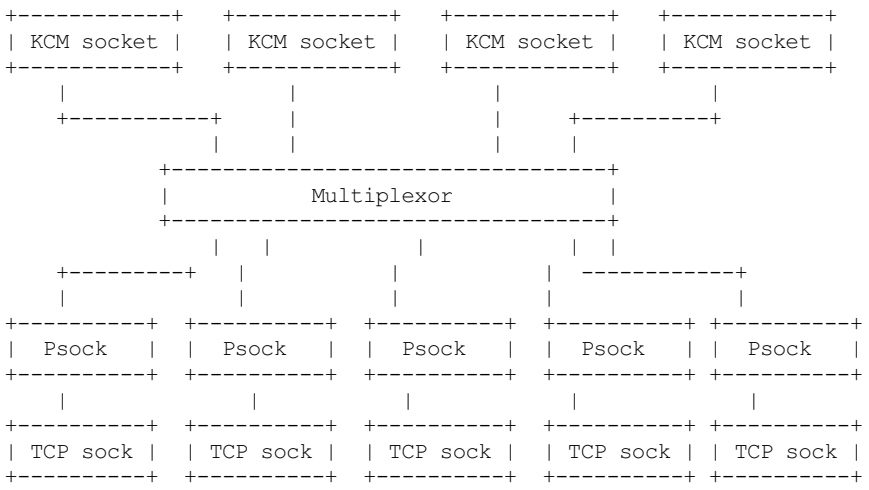
Kernel Connection Multiplexor

Kernel Connection Multiplexor (KCM) is a mechanism that provides a message based interface over TCP for generic application protocols. With KCM an application can efficiently send and receive application protocol messages over TCP using datagram sockets.

KCM implements an NxM multiplexor in the kernel as diagrammed below:



KCM sockets

The KCM sockets provide the user interface to the multiplexor. All the KCM sockets bound to a multiplexor are considered to have equivalent function, and I/O operations in different sockets may be done in parallel without the need for synchronization between threads in userspace.

Multiplexor

The multiplexor provides the message steering. In the transmit path, messages written on a KCM socket are sent atomically on an appropriate TCP socket. Similarly, in the receive path, messages are constructed on each TCP socket (Psock) and complete messages are steered to a KCM socket.

TCP sockets & Psocks

TCP sockets may be bound to a KCM multiplexor. A Psock structure is allocated for each bound TCP socket, this structure holds the state for constructing messages on receive as well as other connection specific information for KCM.

Connected mode semantics

Each multiplexor assumes that all attached TCP connections are to the same destination and can use the different connections for load balancing when transmitting. The normal send and recv calls (include sendmmsg and recvmmsg) can be used to send and receive messages from the KCM socket.

Socket types

KCM supports SOCK_DGRAM and SOCK_SEQPACKET socket types.

Message delineation

Messages are sent over a TCP stream with some application protocol message format that typically includes a header which frames the messages. The length of a received message can be deduced from the application protocol header (often just a simple length field).

A TCP stream must be parsed to determine message boundaries. Berkeley Packet Filter (BPF) is used for this. When attaching a TCP socket to a multiplexor a BPF program must be specified. The program is called at the start of receiving a new message and is given an skbuff that contains the bytes received so far. It parses the message header and returns the length of the message. Given this information, KCM will construct the message of the stated length and deliver it to a KCM socket.

TCP socket management

When a TCP socket is attached to a KCM multiplexor data ready (POLLIN) and write space available (POLLOUT) events are

handled by the multiplexor. If there is a state change (disconnection) or other error on a TCP socket, an error is posted on the TCP socket so that a POLLERR event happens and KCM discontinues using the socket. When the application gets the error notification for a TCP socket, it should unattach the socket from KCM and then handle the error condition (the typical response is to close the socket and create a new connection if necessary).

KCM limits the maximum receive message size to be the size of the receive socket buffer on the attached TCP socket (the socket buffer size can be set by SO_RCVBUF). If the length of a new message reported by the BPF program is greater than this limit a corresponding error (EMSGSIZE) is posted on the TCP socket. The BPF program may also enforce a maximum messages size and report an error when it is exceeded.

A timeout may be set for assembling messages on a receive socket. The timeout value is taken from the receive timeout of the attached TCP socket (this is set by SO_RCVTIMEO). If the timer expires before assembly is complete an error (ETIMEDOUT) is posted on the socket.

User interface

Creating a multiplexor

A new multiplexor and initial KCM socket is created by a socket call:

```
socket(AF_KCM, type, protocol)
```

- type is either SOCK_DGRAM or SOCK_SEQPACKET
- protocol is KCMPROTO_CONNECTED

Cloning KCM sockets

After the first KCM socket is created using the socket call as described above, additional sockets for the multiplexor can be created by cloning a KCM socket. This is accomplished by an ioctl on a KCM socket:

```
/* From linux/kcm.h */
struct kcm_clone {
      int fd;
};

struct kcm_clone info;

memset(&info, 0, sizeof(info));

err = ioctl(kcmfd, SIOCKCMCLONE, &info);

if (!err)
  newkcmfd = info.fd;
```

Attach transport sockets

Attaching of transport sockets to a multiplexor is performed by calling an ioctl on a KCM socket for the multiplexor. e.g.:

```
/* From linux/kcm.h */
struct kcm_attach {
      int fd;
      int bpf_fd;
};

struct kcm_attach info;

memset(&info, 0, sizeof(info));

info.fd = tcpfd;
info.bpf_fd = bpf_prog_fd;

ioctl(kcmfd, SIOCKCMATTACH, &info);
```

The kcm_attach structure contains:

- fd: file descriptor for TCP socket being attached
- bpf_prog_fd: file descriptor for compiled BPF program downloaded

Unattach transport sockets

Unattaching a transport socket from a multiplexor is straightforward. An "unattach" ioctl is done with the kcm_unattach structure as the argument:

```
/* From linux/kcm.h */
struct kcm_unattach {
      int fd;
```

```
};

struct kcm_unattach info;

memset(&info, 0, sizeof(info));

info.fd = cfd;

ioctl(fd, SIOCKCMUNATTACH, &info);
```

Disabling receive on KCM socket

A setsockopt is used to disable or enable receiving on a KCM socket. When receive is disabled, any pending messages in the socket's receive buffer are moved to other sockets. This feature is useful if an application thread knows that it will be doing a lot of work on a request and won't be able to service new messages for a while. Example use:

```
int val = 1;

setsockopt(kcmfd, SOL_KCM, KCM_RECV_DISABLE, &val, sizeof(val))
```

BFP programs for message delineation

BPF programs can be compiled using the BPF LLVM backend. For example, the BPF program for parsing Thrift is:

```
#include "bpf.h" /* for __sk_buff */
#include "bpf_helpers.h" /* for load_word intrinsic */

SEC("socket_kcm")
int bpf_prog1(struct __sk_buff *skb)
{
     return load_word(skb, 0) + 4;
}

char _license[] SEC("license") = "GPL";
```

Use in applications

KCM accelerates application layer protocols. Specifically, it allows applications to use a message based interface for sending and receiving messages. The kernel provides necessary assurances that messages are sent and received atomically. This relieves much of the burden applications have in mapping a message based protocol onto the TCP stream. KCM also make application layer messages a unit of work in the kernel for the purposes of steering and scheduling, which in turn allows a simpler networking model in multithreaded applications.

Configurations

In an Nx1 configuration, KCM logically provides multiple socket handles to the same TCP connection. This allows parallelism between in I/O operations on the TCP socket (for instance copyin and copyout of data is parallelized). In an application, a KCM socket can be opened for each processing thread and inserted into the epoll (similar to how SO_REUSEPORT is used to allow multiple listener sockets on the same port).

In a MxN configuration, multiple connections are established to the same destination. These are used for simple load balancing.

Message batching

The primary purpose of KCM is load balancing between KCM sockets and hence threads in a nominal use case. Perfect load balancing, that is steering each received message to a different KCM socket or steering each sent message to a different TCP socket, can negatively impact performance since this doesn't allow for affinities to be established. Balancing based on groups, or batches of messages, can be beneficial for performance.

On transmit, there are three ways an application can batch (pipeline) messages on a KCM socket.

1. Send multiple messages in a single sendmmsg.
2. Send a group of messages each with a sendmsg call, where all messages except the last have MSG_BATCH in the flags of sendmsg call.
3. Create "super message" composed of multiple messages and send this with a single sendmsg.

On receive, the KCM module attempts to queue messages received on the same KCM socket during each TCP ready callback. The targeted KCM socket changes at each receive ready callback on the KCM socket. The application does not need to configure this.

Error handling

An application should include a thread to monitor errors raised on the TCP connection. Normally, this will be done by placing each TCP socket attached to a KCM multiplexor in epoll set for POLLERR event. If an error occurs on an attached TCP socket, KCM sets an EPIPE on the socket thus waking up the application thread. When the application sees the error (which may just be a

disconnect) it should unattach the socket from KCM and then close it. It is assumed that once an error is posted on the TCP socket the data stream is unrecoverable (i.e. an error may have occurred in the middle of receiving a message).

TCP connection monitoring

In KCM there is no means to correlate a message to the TCP socket that was used to send or receive the message (except in the case there is only one attached TCP socket). However, the application does retain an open file descriptor to the socket so it will be able to get statistics from the socket which can be used in detecting issues (such as high retransmissions on the socket).