Implementing network protocols – A Streams approach

Thomas George

ABSTRACT

Unix Streams by Dennis Ritchie is a suitable pattern for implementing layered communication network protocols. This pattern provides for a flexible implementation, whereby a protocol layer can be dynamically inserted into or removed from the protocol stack. This is a portable implementation of Streams framework.

Keywords

Object Oriented methods

Interface-based programming

.

1.INTRODUCTION

Unix Streams[1], is a popular framework for implementing networking protocols. Though it was intended to be a part of the kernel in Unix OS; the framework uses a pattern that can easily be applied to develop a user-level framework in any OS. We refer to both the pattern and the framework using the term, Streams, when there is no confusion.

Streams is like a pipeline along which modules implementing various protocol behaviors are arranged. As data flows thru the pipeline it is acted upon by the modules. The pipeline also provides buffer management and flow-control. This is different from input-output stream libraries available for various compilers. The following chapters further elaborate this idea and also introduce an implementation of this pattern in OS user-space.

2.Streams

Streams is a full-duplex connection between the application and a device driver containing independent modules thru which data pass. In this design protocol layers are implemented as modules, all of them sharing a common interface defined by Streams.

Each module has two queues – downstream queue for messages headed toward the device driver and upstream queue for messages headed in the opposite direction. The queues buffer messages for processing in a message-list, and also implement a flow-control mechanism. The modules are connected by connecting the queues, in a singly-linked list each way to form a downstream and an upstream. The queues are actually 'interfaces' in OO terminology, containing a set of function pointers which are assigned to fulfill protocol layer specific functionality. Each queue has the following interfaces:

- putp(queue, data) procedure to submit data to a queue. Invoked by previous queue feeding into this queue. The data is either stored in the internal message-list for later processing or passed onto the next queue.
- srvp(queue) a service procedure that processes and transfers data onto next queue. This is used for implementing asynchronous behavior. It is called automatically when the queue needs servicing. For example a blocked queue would be automatically scheduled for service when a queue further along in the stream has its flow-control restrictions lifted.
- open(deviceId) procedure invoked by the Streams framework on pushing a queue pair into place.
- close(streamHead) procedure invoked by the Streams framework on popping a queue pair from its place.

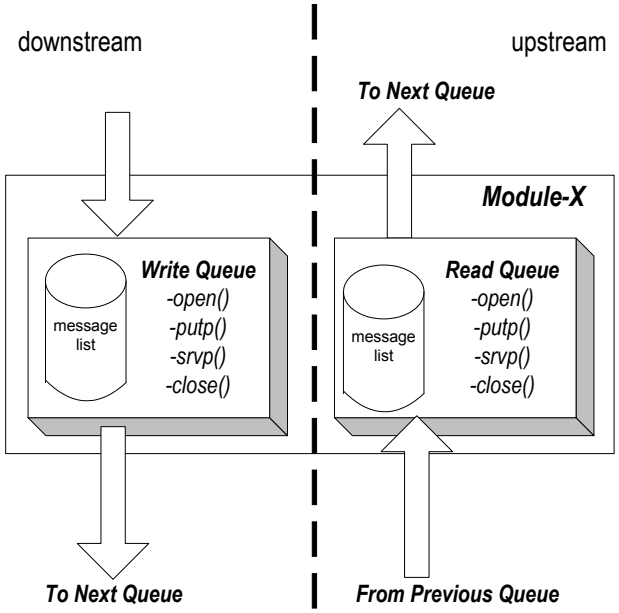

Figure 21. Streams module

Each queue acts as if it has its own thread of control. This is because the service procedures in each queue are invoked by a scheduler as necessary, which gives the illusion that all service procedures are running concurrently. The scheduler could use multiple threads if that sort of concurrency is required.

Communication between layers is done by passing message buffers. These message buffers use a data type that enables linking multiple buffers together. This data type also allows referencing of a single data buffer by multiple messages.

3.Streams Benefits

Streams provides an uniform mechanism for buffering and I/O processing. Layered protocols can be implemented with each layer executing in its own 'virtual' thread. Message buffers allow for avoiding the copying of data as messages pass thru the layers; also for the ease of adding protocol headers or segmenting the data.

Once implemented in the framework this is available for use in all protocol layer implementations and hence promotes re-use. Also, the message-passing mechanism imposes a discipline on the code design that promotes modularization.

The flexibility provided by this pattern allows for protocol layers to be re-configured at run-time. That makes a suitable test bed framework too.

4.PSTREAMS

4.1.Overview

Streams with some enhancements is native to many Unix flavors – where it is usually incorporated into the kernel. As such this is operating system dependant [2]. Our goal was to derive a version following the ideas in Dennis Ritchie's original paper and make it easy to port. Our version – PSTREAMS – is written in C, and supplies its own basic memory manager.

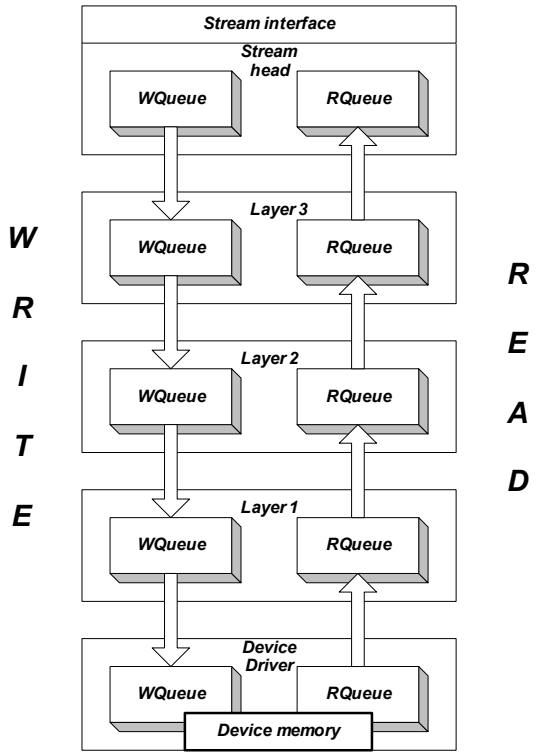

Figure 41. Modules plumbed into PSTREAMS

The following subsections introduce the design of PSTREAMS.

4.2.p_streamhead

This class is the application interface supplied by the framework to the user of the protocol stack. In, C, this is implemented as a struct including a set of function pointers.

- pstreams_open(device ID) creates and returns a stream handle(PstreamHandle) to the calling application. It also sets up connection between the stream head and the device represented by device ID.
- pstreams_push(PstreamHandle, PstreamModule) pushes given developer supplied module into stream. The module is pushed just beneath the stream head.
- pstreams_pop(PstreamHandle) complement of pstreams_push()
- pstreams_putmsg(PstreamHandle, Msg) sends Msg downstream.
- pstreams_getmsg(PstreamHandle, Msg) reads upstream messages into Msg.
- pstreams_close(PstreamHandle) closes the stream

4.3.p_queues

A PSTREAMS module supplied by a developer contains the code that PSTREAMS uses to instantiate the queues that an in-stream module contains. A p_qinit structure contains queue initialization code. Thus a developer module contains two such structures to initialize write and read queues.

```
struct p_qinit
{
    int (*qi_putp)(struct p_queue *, struct p_msgb *);
        /*pointer to put procedure*/

    int (*qi_srvp)(struct p_queue *);
        /*pointer to service procedure*/

    int (*qi_qopen)(struct p_queue *);
 /*pointer to procedure called when module is opened or pushed*/

    int (*qi_qclose)(struct p_queue *);
 /*pointer to procedure called when module is closed or popped*/

    P_MODINFO*qi_minfo;
        /*module specific default values*/

    P_MODSTAT *qi_mstat;
        /* module statistics*/
}
```

PSTREAMS queue is created automatically when the developer module is pushed in; its associated procedures are instantiated using the given p_qinit structure.

```
struct p_queue
{
    p_qinit q_qinfo;
     /*info on processing routines for queue*/

        listhdr *q_msglist;
    /*queue of messages - whence the name*/
        void *strmhead;
    /*pointer to its streamhead*/

        struct p_queue *q_next;
    /*next queue downstream*/
        struct p_queue *q_peer;
    /*pointer to peer queue*/

        void *q_ptr;    /*private data store*/
        ushort q_count; /*count of outstanding bytes queued*/
        ushort q_flag;  /*state of queue - treated as a bit flag*/
        short  q_minpsz; /*minimum packet size*/
        short q_maxpsz; /*maximum packet size*/
        ushort q_hiwat; /*hghi water mark*/
```

```
    ushort q_lowat; /*low water mark*/
};
```

4.4.message buffers

Messages that an application sends to the stream head are converted into PSTREAMS representation consisting of message blocks (P_MSGB) and data blocks (P_DATAB). Each P_MSGB has an associated P_DATAB. A P_DATAB can be associated with multiple P_MSGBs.

```
struct p_daab
{
    P_FREE_RTN      *db_frtnp;/*function that free's this*/
    unsigned char   *db_base; /*pointer to start of data buffer*/
    unsigned char   *db_lim; /*pointer to end of buffer*/
    unsigned char   db_ref;/*reference count for this data block*/
    unsigned char   db_type;/*data type*/
    unsigned char   db_flags;
    struct msgb     *db_msgaddr; /* backptr to MSGB*/

    unsigned char data[MAXDATABSIZE];

/*data buffer  – for simplicity attached to this structure*/
};

struct p_msgb
{
    struct p_msgb *b_cont; /*link to next message block*/
    unsigned char *b_rptr; /*read pointer into data block*/
    unsigned char *b_wptr;/*write pointer into data block*/
    p_datab *b_datap; /*pointer to data block*/
};
```

4.5.Flow-Control

Each pstream queue has a high water level and a low water level. If the byte count in that queue exceeds the high water mark, that queue is marked ‘busy’ and subsequent pstreams_canput() calls into that queue return FALSE until byte count falls below the low water. Any queue waiting to send data to a ‘busy’ queue is marked ‘blocked’. Blocked queues are automatically serviced by the scheduler when the target queue’s ‘busy’ status is removed.

4.6.Scheduling

pstreams_callsrvp() is a pstreams utility function that can be called periodically by pstreams user applications to schedule the various queues. However, in multi-threaded operating systems this chore can be assigned to a task that blocks when idle. It is alos to let each pstream queue’s srvp() function execute in its own thread, providing good parallelism.

4.7.Memory manager

A fast memory manager is essential because message buffers are allocated and released frequently. Our approach is to pre-allocate pools of fixed size memory blocks. Each pool consists of consecutive memory on which a list structure is imposed. The elements of the list are the fixed sized memory blocks that can be allocated. A memory block allocated from a pool has to be released back into that pool.

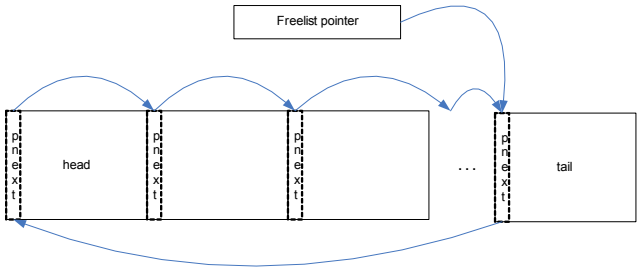

Figure 42. Initial list structure on memory pool

5.CONCLUSION

PSTREAMS is a simple and faithful implementation of the Streams pattern proposed by Dennis Ritchie. This simplicity opens up the opportunity to use this pattern in various platforms, especially light-weight embedded platforms, for implementing layered networking protocols.

The ease of re-configuring the layers allows us to do loop-back tests within the same process, by pushing in a peer-layer composed by switching the read and write queues.

6.REFERENCES

1. Dennis M. Ritchie A Stream Input-Output System AT&TBell Laboratories Technical Journal *, 63 N. 8 Part 2* (Oct. 1984), 1897-1910.
2. *The Magic Garden Explained. The internals of UNIX SYSTEM V RELEASE 4* Berny Goodheart and James Cox