Overview of STREAMS

STREAMS is a flexible communication subsystem framework

- Originally developed by Dennis Ritchie for Research UNIX

STREAMS provides a uniform infrastructure for developing and configuring character-based I/O

- e.g., networks, terminals, local IPC

STREAMS supports the addition and removal of processing components at installation-time or run-time

- Via user-level or kernel-level commands

STREAMS Overview (cont'd)

- The STREAMS paradigm is data-driven, not demand-driven
  - i.e., asynchronous in the kernel, yet synchronous at the application

- Supports both immediate and deferred processing

- Internally, data are transferred by passing pointers to messages
  - Goal is to reduce memory-to-memory copying overhead

STREAMS Benefits

- STREAMS provides an integrated environment for developing kernel-resident networking services

- STREAMS promotes definition of standard service interfaces
  - e.g., TPI and DLPI

- STREAMS supports dynamic "service substitution" controlled by user-level commands
STREAMS Benefits (cont’d)

- Message-based interfaces enable off-board protocol migration
- Permits layered and de-layered multiplexing
- More recent implementations take advantage of parallelism in the operating system and hardware

A Simple Stream

![Diagram of a simple stream](image)

The Stream Head

- A “stream head” exports a uniform service interface to other layers of the UNIX kernel
  - Including the application “layer” running in user-space

- General stream head services include
  1. Queueing
     - Provides a synchronous interface to asynchronous devices
  2. Datagram- and stream-oriented data transfer
  3. Segmentation and reassembly of messages
  4. Event propagation
     - *i.e.*, signals
Stream Head (cont'd)

- Stream head operations include
  - `stropen()`
    - called from the file system layer to open a Stream
  - `strclose()`
    - called from the file system layer to dismantle a Stream
  - `strread()`
    - called from the file system layer to retrieve data messages coming *upstream*
  - `strwrite()`
    - called from the file system layer to send data messages *downstream*
  - `stroctl()`
    - called from the file system layer to perform control operations
  - `strgetmsg()`
    - called from the system call layer to get a protocol or data message coming *upstream*
  - `strputmsg()`
    - called from the system call layer to send a protocol or data message *downstream*
  - `strpoll()`
    - called from the file system layer to check if pollable events are satisfied

Messages

- In STREAMS, all information is exchanged via messages
  - *i.e.*, both data and control messages of various priorities
- A multi-component message structure is used to reduce the overhead of
  1. Memory-to-memory copying
    - *i.e.*, via "reference counting"
  2. Encapsulation/de-encapsulation
    - *i.e.*, via "composite messages"
- Messages may be queued at STREAM modules
Message Structure

Composite Message

Message Buffer Sharing

STREAMS Message Types

- Normal priority messages
  - M_DATA, M_PROTO, M_PASSFP, and M_IOCTL may be generated from user-level
  - Typically subject to flow control
STREAMS Message Types

(cont’d)

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_PROTO</td>
<td>protocol information</td>
</tr>
<tr>
<td>M_FLUSH</td>
<td>flush queues</td>
</tr>
<tr>
<td>M_IOCTL</td>
<td>acknowledge ioctl() request</td>
</tr>
<tr>
<td>M_IOCTLNAK</td>
<td>fail ioctl() request</td>
</tr>
<tr>
<td>M_COPYIN</td>
<td>request to copyin ioctl() data</td>
</tr>
<tr>
<td>M_COPYOUT</td>
<td>request to copyout ioctl() data</td>
</tr>
<tr>
<td>M_IOCTLDATA</td>
<td>reply to M_COPYIN and M_COPYOUT</td>
</tr>
<tr>
<td>M_READ</td>
<td>signal process group</td>
</tr>
<tr>
<td>M_READ_NOTICE</td>
<td>read notification</td>
</tr>
<tr>
<td>M_HANGUP</td>
<td>line disconnect</td>
</tr>
<tr>
<td>M_ERROR</td>
<td>fatal error</td>
</tr>
<tr>
<td>M_STOP</td>
<td>stop output immediately</td>
</tr>
<tr>
<td>M_START</td>
<td>restart output</td>
</tr>
<tr>
<td>M_STOPI</td>
<td>stop input immediately</td>
</tr>
<tr>
<td>M_STARTI</td>
<td>restart input</td>
</tr>
<tr>
<td>M_PCRSE</td>
<td>reserved for RSE use</td>
</tr>
</tbody>
</table>

- **High** priority messages

  * Typically not flow controlled
  * M_PROTO may be generated from user-level
  * Others passed between STREAM components

Queue Structure

- **queue_t**
  - Primary data structure
    - Each module contains a write queue and a read queue
    - Stores information on flow control, scheduling, max/min interface sizes, linked messages, private data

- **qinit**
  - Contains put(), service(), open(), close() subroutines

- **qband**
  - Contains information on each additional message band > 0 and < 256

- **module_info**
  - Stores default flow control information

Queue and Message Linkage
Queue Subroutines

- Four standard subroutines are associated with queues in a module or driver, e.g.,

  - `open(queue_t *q, dev_t *devp, int oflag, int sflag, cred_t *cred_p);`
    - Called when Stream is first opened and on any subsequent opens
    - Passed a pointer to the new read queue
    - Also called any time a module is "pushed" onto the Stream

  - `close(dev_t dev, int flag, int otyp, cred_t *cred_p);`
    - Called when last reference to a Stream is closed
    - Also called when a module is "popped" off the Stream

Queue Subroutines (cont'd)

- Standard subroutines (cont'd)
  - `put(queue_t *q, mblk_t *mp)`
    - Performs immediate processing
    - Supports synchronous communciation services
      - i.e., further queue processing is blocked until `put()` returns

  - `service(queue_t *q)`
    - Performs deferred processing
    - Supports asynchronous communication services
      - Uses the message queue available in a `queue`
    - Runs as a "weightless" process...

Queue Flow Control

Flow Control and the `service()` Procedure

- Typical non-multiplexed example

  ```c
  int service (queue_t *q) {
    mblk_t *mp;

    while ((mp = getq (q)) != 0) {
      if (queclass (mp) == QPCTL ||
          canputnext (q)) {
        /* Process message */
        putnext (q, mp);
      } else {
        putbq (q, mp);
        return 0;
      }
    }
  }
  ```

- Flow control is more complex with multiplexers and concurrency
Flow Control and the `canput()` Procedure

- `canputnext()` is used by `put()` and `service()` routines to test advisory flow control conditions

- *e.g.*,

```c
int canputnext (queue_t *q)
{
    find closest queue with a service() procedure
    if (queue is full) {
        set flag for "back-enabling"
        return 0;
    }
    return 1;
}
```

- Note that non-MP systems may use `canput()`...

Flow Control and the `put()` Procedure

- Typical `put()` example

```c
int put (queue_t *q, mblk_t *mp)
{
    if (queclass (mp) == QPCTL)
        canputnext (q); /* Process message */
    else
        putq (q, mp); /* Enables service routine */
    return 0;
}
```

putq()

- The `int putq(queue_t *, mblk_t *)` function enqueues a message on a queue
  - It is typically called by a queue's `put()` procedure when it can no longer proceed...

- putq() automatically handles
  1. priority-band allocation
  2. priority-band message insertion
  3. flow control

- Enqueueing a high priority message automatically schedules the queue's `service()` procedure to run at some point
  - Differs on MP vs. non-MP system

getq()

- The `mblk_t *getq(queue_t *)` function dequeues a message from a queue
  - It is typically called by a queue's `service()` procedure

- Messages are dequeued in priority order
  - *i.e.*, higher priority messages are stored first in the queue!

- getq() handles
  1. Flow control
  2. Back-enabling

- getq() returns 0 when there are no available messages
Multiplexing

- STREMS provides rudimentary support for multiplexing via multiplexor drivers
  - Unlike modules, multiplexors occupy a file-system node that can be "opened"
    - Rather than "pushed"

- Multiplexors may contain one or more upper and/or lower connections to other STREAM modules and/or multiplexors
  - Enables support for layered network protocol suites
    - e.g., "/dev/tcp"

- Note there is no automated support for propagating flow control across multiplexors

---

**Multiplexor Links (before)**

```
fd1 = open("/dev/ip");
fd2 = open("/dev/eth");
```

---

**Multiplexor Links (after)**

```
iocet(fd1, IPC_LINK, fd_2);
```

---

**Internetnetworking Multiplexor**
Pipes and FIFOs

- In SVR4 and Solaris, the traditional local IPC mechanisms such as pipes and FIFOs have been reimplemented using STREAMS.

- This has broadened the semantics of pipes and FIFOs in the following ways:

1. Pipes are now bidirectional (STREAM pipes)

2. Pipes and FIFOs now support both byte stream-oriented and message-oriented communication.

   - ioctl()s exist to modify the behavior of STREAM descriptors to enable or disable many of the new features.

3. Pipes can be explicitly named and exported into the file system.

4. Pipes and FIFOs can be extended by having modules pushed onto them.
Mounted Streams and CONNLД

- In earlier generations of UNIX, pipes were restricted to allowing multiplexed communication
  - This is overly complex for many client/server-style applications

- SVR4 UNIX and Solaris provide a mechanism known as “Mounted Streams” that permits non-multiplexed communication
  - This provides semantics similar to UNIX domain sockets
  - However, Mounted Streams are more flexible since they incorporate other features of STREAMS

### Layered Multiplexing

- **Advantages**
  - Share resources such as control blocks
  - Supports standard OSI and Internet layering models

- **Disadvantages**
  - More processing involved to demultiplex in deep protocol stacks
  - May be more difficult to parallelize due to locks and shared resource contention
  - Hard to propagate flow control and QoS info across muxer
De-layered Multiplexing

Advantages
- Less processing for deep protocol stacks
- Potentially increased parallelism due to less contention and locking required
- Easier to propagate flow control and QOS info

Disadvantages
- Violates layering (e.g., need a packet filter)
- Replicates resources such as control blocks

STREAM Concurrency
- Modern versions of STREAMS support multiprocessing
  - Since modern UNIX systems have multi-threaded kernels

  Different levels of concurrency support include
  1. Fine-grain
     * Queue-level
     * Queue-pair-level
  2. Coarse-grain
     * Module-level
     * Module-class-level
     * Stream-level

  Note, developers must use kernel locking primitives to provide mutual exclusion and synchronization

Concurrency Alternatives
- Layer Parallelism
Concurrency Alternatives (cont’d)

- Connectional Parallelism
Concurrent Alternatives (cont’d)

- Message Parallelism
STREAMS Evaluation

• **Advantages**
  
  – Portability, availability, stability
  
  – Kernel-mode efficiency

• **Disadvantages**
  
  – Stateless process architecture
    
    ▶ *i.e.*, cannot block!
  
  – Lack of certain support tools
    
    ▶ *e.g.*, standard demultiplexing mechanisms
  
  – Kernel-level development environment
    
    ▶ Limited debugging support…
  
  – Lack of real-time scheduling for STREAMS processing…
    
    ▶ Timers may be used for “isochronous” service
Summary

- STREAMS provides a flexible communication framework that supports dynamic configuration and reconfiguration of protocol functionality

- Module interfaces are well-defined and reasonably well-documented

- Support for multi-processing exists in Solaris 2.x