Socket Programming

Christian Grothoff
christian@grothoff.org
http://grothoff.org/christian/

"The major advances in speed of communication and ability to interact took place more than a century ago. The shift from sailing ships to telegraph was far more radical than that from telephone to email!" – Noam Chomsky

BSD Sockets

- BSD's approach to a C API for network programming
- Standardized in POSIX
- Today: `PF_INET` & `AF_INET`-only
- Socket data type: `int`
- Address data types: `struct sockaddr`, `struct sockaddr_in`, `struct sockaddr_storage`

Keeping it short...

- No declarations of variables unrelated to networking
- No error handling code
- Minor details ignored

$\Rightarrow$ Read man-pages to easily fill the gaps

Parsing addresses

```
int parse(const char * in,
          struct in_addr * out) {
  int ret = inet_pton(AF_INET, in, out);
  if (ret < 0)
    fprintf(stderr, "AF_INET not supported!\n");
  else if (ret == 0)
    fprintf(stderr, "Syntax error!\n");
  else
    return 0;
  return -1;
}
```

Creating UDP Sockets

```
int s;

s = socket (PF_INET, SOCK_DGRAM, IPPROTO_UDP);
// ...
close (s);
```

UDP: Sending Data

```
struct sockaddr *sa;
struct sockaddr_in sin;
int flags = 0;

sin.sin_family = AF_INET;
sin.sin_port = htons (7 /* ECHO */);
inet_pton (AF_INET, "127.0.0.1",
           &sin.sin_addr);
sa = (struct sockaddr *) &sin;
sendto (s, "Hello World", strlen ("Hello World"),
        flags, sa, sizeof (sin));
```

UDP: Receiving Data

```
char buf[65536];
struct sockaddr *sa;
struct sockaddr_in sin;
socklen_t addrlen = sizeof (sin);
int flags = 0;

sa = (struct sockaddr *) &sin;
recvfrom (s, buf, sizeof (buf),
          flags, sa, &addrlen);
```

FreeBSD

```
   memset (&addr, 0, sizeof (addr));
#if HAVE_SOCKADDR_IN_SIN_LEN
   addr.sin_len = sizeof (addr);
#endif
   addr.sin_family = AF_INET;
   ...
```

Example: minimal TCP client

Functionality:

- Connect to server on port 5002
- Transmit file to server

System Calls for TCP client

1. `socket`
2. `connect`
3. (`recv`|`send`)*
4. [`shutdown`]
5. `close`

IPv4 TCP Client Example

```
struct sockaddr_in addr;
int s = socket (PF_INET, SOCK_STREAM, 0);
memset (&addr, 0, sizeof (addr));
addr.sin_family = AF_INET;
addr.sin_port = htons (5002);
addr.sin_addr.s_addr = htonl (INADDR_LOOPBACK);
connect (s, (const struct sockaddr*) &addr,
         sizeof (addr));
process(s);
close (s);
```

Client Example: processing

```
static void process (int s) {
  char buf[4092];
  int f = open (FILENAME, O_RDONLY);
  while ( (-1 != (n = read (f, buf, sizeof (buf)))) &&
          (n != 0) ) {
    pos = 0;
    while (pos < n) {
      ssize_t got = write (s, &buf[pos], n - pos);
      if (got <= 0) goto END;
      pos += got;
    }
  }
END:
  close (f);
}
```

Example: minimal TCP server

Functionality:

- Listen to port 5002
- Write incoming TCP stream to disk
- Support multiple clients in parallel using `pthreads`

System Calls for TCP server

1. `socket`
2. `bind`
3. `listen`
4. "forever":
   (a) `accept`
   (b) (`recv`|`send`)*
   (c) [`shutdown`]
   (d) `close`
5. `close`

Server Example: processing

```
static void * process (struct T * t) {
  ssize_t n;
  char buf[4092];

  int f = creat (filename, S_IRUSR | S_IWUSR);
  while ( (-1 != (n=read (t->a, buf, sizeof (buf)))) &&
          (n != 0) )
    write (f, buf, n);
  close (f);
  close (t->a);
  return NULL;
}
```

Server Example: `struct T`

```
struct T
{
  int a;
};
```

Server Example: accepting

```
struct sockaddr addr;
int s = socket (PF_INET, SOCK_STREAM, 0);
memset (&addr, 0, sizeof (addr));
struct sockaddr_in * ia = (struct sockaddr_in*) &addr;
ia->sin_family = AF_INET; ia->sin_port = htons (5002);
bind (s, &addr, sizeof (struct sockaddr_in));
listen (s, 5);
while (1) {
  memset (&addr, 0, sizeof (addr));
  socklen_t alen = sizeof (struct sockaddr);
  t->a = accept (s, &addr, &alen);
  pthread_create (&pt, NULL, &process, t);          }
```

Threads?

- Need to "clean up" handle `pt` (use `struct T`)
- Can cause dead-locks, data races
- Do not exist on all platforms
- Use **at least** one page of memory per thread, often more
- How scalable is your thread-scheduler?

`select`

- Do everything in one “thread”, no parallel execution needed
- Event-based $\Rightarrow$ tricky API, but fewer tricky bugs!
- Exists on pretty much all network-capable platforms
- Has some issues with UNIX signals, but mostly “safe”
- Scales with $O(n)$

`select` API

- `FD_ZERO(fd_set *set)`
- `FD_SET(int fd, fd_set *set)`
- `FD_ISSET(int fd, fd_set *set)`
- `int select(int n, fd_set *rs, fd_set *ws, fd_set *es, struct timeval *timeout)`

Homework: Read `select_tut` man-page and try it!

Example (1/3)

```
int pi[2];
pipe(pi);
if (fork() == 0) {
  close(pi[0]);
  close(0); close(1); close(2);
  while (1) { write(pi[1], "Hello", 5); sleep(5); }
} else {
  close(pi[1]);
  while (1) { merge(pi[0], 0, 1); }
}
```

Example (2/3)

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
void merge(int in1, int in2, int out) {
  fd_set rs, ws;
  FD_ZERO(&rs);         FD_ZERO(&ws);
  FD_SET(in1, &rs); FD_SET(in2, &rs);
  select(1 + MAX(in1, in2), &rs, &ws, NULL, NULL);
  if (FD_ISSET(in1, &rs)) copy(in1, out);
  if (FD_ISSET(in2, &rs)) copy(in2, out);
}
```

Example (3/3)

```
void copy(int in, int out) {
  size_t num;
  char buf[1024];

  num = read(in, buf, sizeof(buf));
  write(out, buf, num);
}
```

`epoll`

- Select scales with $O(n)$
- Can (theoretically) do the same with $O(1)$
- Linux does this using `epoll`
- Key different to `select`: you must have drained the buffers before `epoll` will trigger again!

epoll API

- `int epoll_create(int size)`
- `int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)`
- `int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout)`

Homework: Read `epoll` man-page and try it!

Other possibilities

- Forking
- Pre-Forking
- Multi-threaded with `select` or `epoll`
- `kqueue` (FreeBSD, NetBSD, OS X)
- Asynchronous IO (W32, z/OS), Signals (Linux)

Further reading: `http://kegel.com/c10k.html`.

`connect` revisited

- `select` works fine for `read` and `write`
- `connect` also blocks!

$\Rightarrow$ Need non-blocking `connect`!

Non-blocking connect

```
struct sockaddr_in addr;
int s = socket (PF_INET,  SOCK_STREAM, 0);
int ret;
int flags = fcntl (s, F_GETFL);
flags |= O_NONBLOCK;
fcntl (s, F_SETFL, flags);
ret = connect (s, (const struct sockaddr*) &addr,
                  sizeof (addr));
if ( (ret == -1) && (errno == EAGAIN) )
  { /* wait in "select" for "write" */ }
```

DNS request

```
int
resolve_old (const char * hostname,
             struct in_addr * addr) {
  struct hostent * he;
  struct sockaddr_in *addr;
  he = gethostbyname(hostname);
  assert (he->h_addrtype == AF_INET);
  assert (hp->h_length == sizeof (struct in_addr));
  memcpy (addr, hp->h_addr_list[0], hp->h_length);
  return OK;
}
```

`gethostbyname` issues

- Synchronous
- IPv4 only

$\Rightarrow$ `gethostbyname2`

`gethostbyname` issues

- Synchronous
- IPv4 only

⇒ `gethostbyname2`

- Not reentrant

⇒ both are obsolete!

IPv4 DNS request with `getaddrinfo`

```
int resolve (const char * hostname,
             struct sockaddr_in * addr) {
  struct addrinfo hints;
  struct addrinfo *result;
  memset (&hints, 0, sizeof (struct addrinfo));
  hints.ai_family = AF_INET;
  getaddrinfo (hostname, NULL, &hints, &result);
  assert(sizeof(struct sockaddr_in) ==result->ai_addrlen);
  memcpy (addr, result->ai_addr, result->ai_addrlen);
  freeaddrinfo (result);
  return OK;
}
```

Reverse Lookup: getnameinfo

```
char *
reverse_resolve (const struct sockaddr_in *ip) {
  char hostname[256];
  if (0 != getnameinfo ((const struct sockaddr*) ip,
                        sizeof (struct sockaddr_in),
                        hostname, sizeof(hostname),
                        NULL, 0, 0))
    return NULL;
  return strdup (hostname);
}
```

Data Transmission

All well-designed protocols transmit data in **network byte order**:

```
uint32_t data;

data = htonl (42);
do_transmit ((const char*) &data,
             sizeof (data));
```

Receiving Data

When receiving data, it must be converted back:

```
char buf[2];
uint16_t *nbo_data;
uint16_t sdata;

do_receive (buf, sizeof (buf));
nbo_data = (uint16_t *) buf;
sdata = ntohs (*nbo_data);
```

Diagnostics

On a GNU/Linux system, run:

- `$ netstat -nl`
- `$ netstat -nt`
- `valgrind --track-fds=yes`

"Happy hacking." – RMS

SCTP

Christian Grothoff
christian@grothoff.org
http://grothoff.org/christian/

"TCP works very hard to get the data delivered in order without errors and does retransmissions and recoveries and all that kind of stuff which is exactly what you want in a file transfer because so you don't want any errors in your file." – John Postel

UDP Semantics

- Message-oriented
- Unreliable, best-effort
- Out-of-order
- No congestion control
- No flow control
- No sessions / streams
- No termination signals

TCP

- Stream-oriented
- Reliable
- In-order
- Congestion control
- Flow control
- Single stream
- Half-closed operation, end-of-transmission signals

SCTP

- Message-oriented
- Reliable or unreliable
- In-order or out-of-order (configurable)
- Congestion control
- Flow control
- Multi-streaming
- No half-closed operation, end-of-transmission signals
- Multi-homing

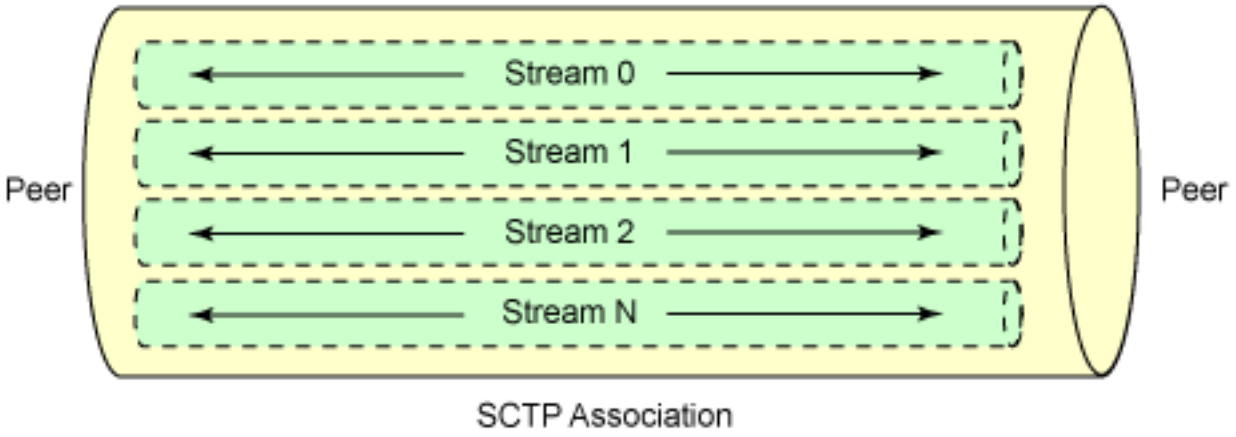
SCTP Availability

- Specified in RFCs 2960, 3286, 4960
- GNU/Linux, BSD, Solaris
- On W32 third-party commercial add-ons exist
- Not supported by many (most?) cheap NAT boxes (!)

SCTP Application Domains

- (real-time) voice & video streaming
- High-performance computing (MPI)
- Transmission over Lossy-channel (WLAN, Satellite) with ECC
- Alternative to TCP (if multi-stream or multi-homing are needed)

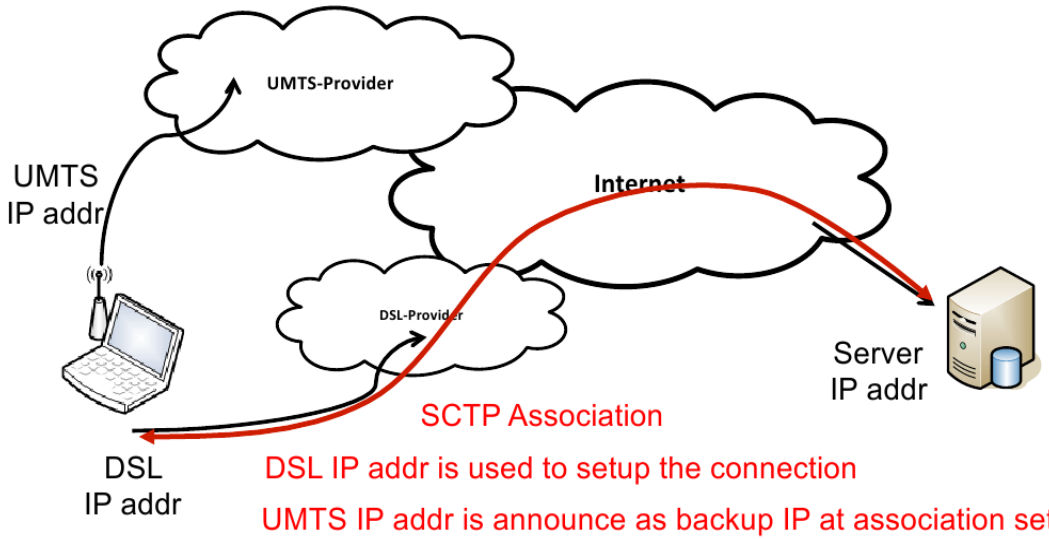
SCTP Associations



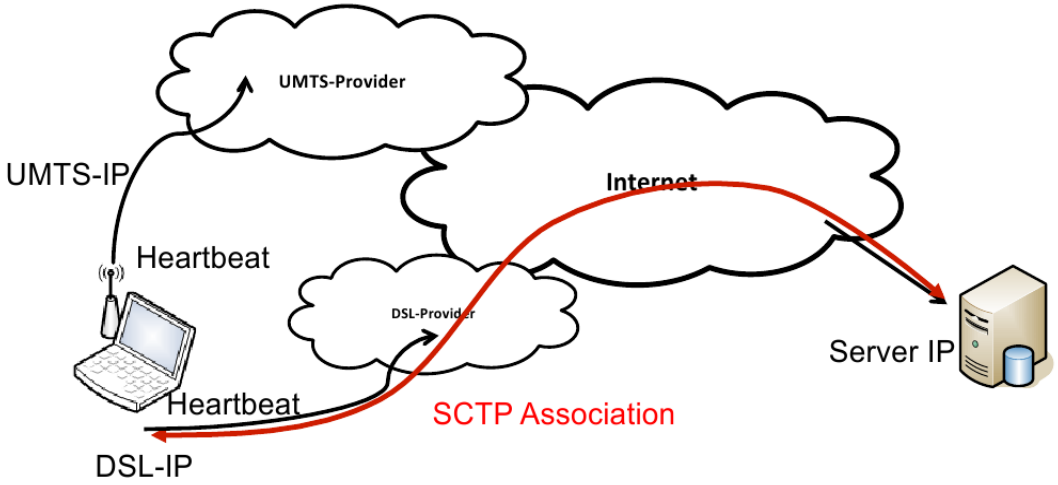
Multiple Streams

- Solves head of line blocking
- Different reliability levels can be mixed:
  - Fully reliable (TCP like)
  - Unreliable (UDP like)
  - Partial reliable (lifetime specifies how long to try to retransmit)

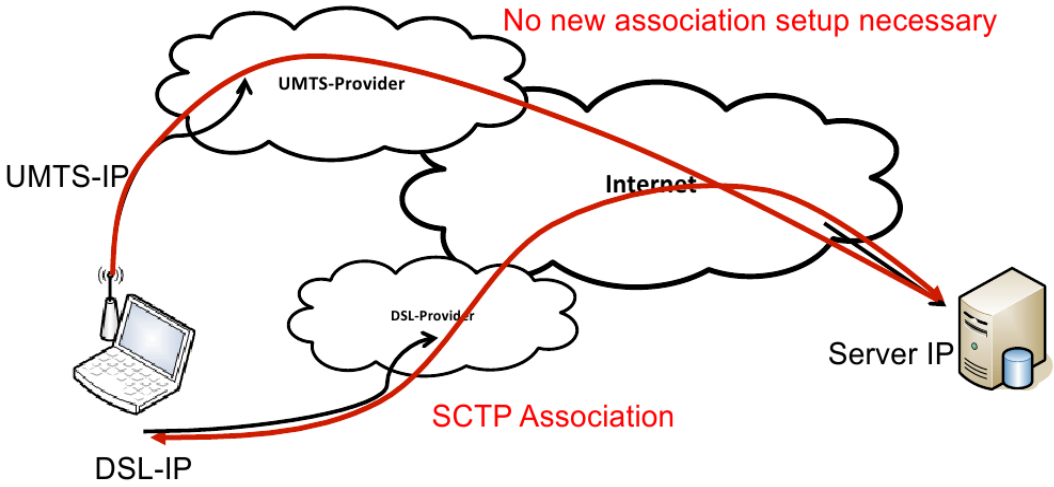
Multi-Homing (1/3)



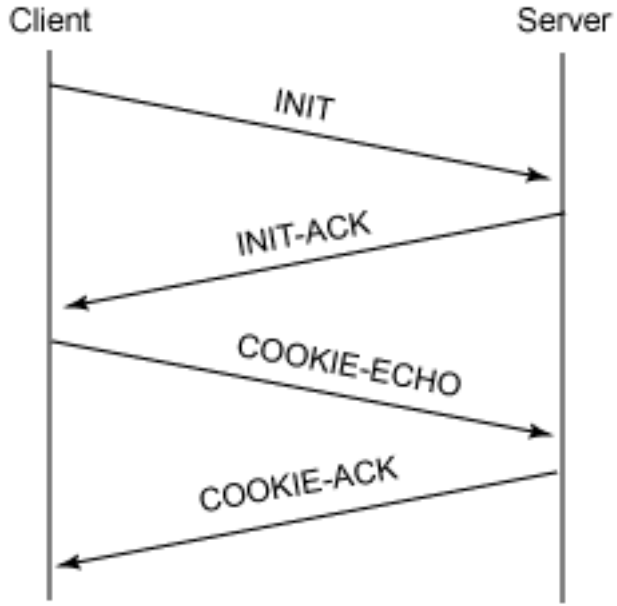
Multi-Homing (2/3)



Multi-Homing (3/3)



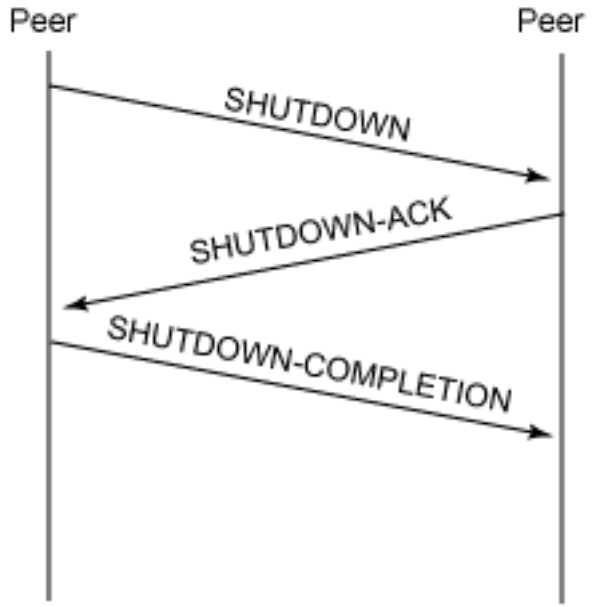
SCTP Setup: Cookies against SYN-Floods[1]



SCTP 4-way handshake

[1] ... at least not with spoofed sender IP.

SCTP Shutdown: No Half-Closed



SCTP Connection Termination

SCTP Packet Structure: Chunks

Source Port		Destination Port
Verification tag		
Checksum		
Chunk 1 type	Chunk 1 flags	Chunk 1 length
Chunk 1 data		
...		
Chunk 2 type	Chunk 2 flags	Chunk 2 length
Chunk 2 data		
...		

Chunk Types

- INIT, INIT-ACK
- COOKIE, COOKIE-ACK
- DATA, SACK
- SHUTDOWN, SHUTDOWN-ACK, SHUTDOWN-COMPLETION

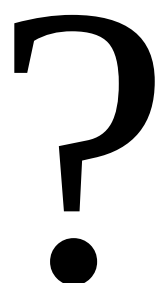
SCTP Data Transmission

Type	Flags	Length
Transport Sequence Number (TSN)		
Stream Identifier S		Stream Sequence Number n
Payload Protocol Identifier		
User Data		

Congestion Control

- Based on Congestion Control in TCP
- SCTP can ACK out-of-order blocks, but those do not count for CC
- Separate CC parameters kept for each destination address
- Parameters for unused destinations “decay” over time
- Each destination address begins with slow-start

Questions



"But there's so much kludge, so much terrible stuff, we are at the 1908 Hurley washing machine stage with the Internet. That's where we are. We don't get our hair caught in it, but that's the level of primitiveness of where we are. We're in 1908. " – Jeff Bezos

SCTP APIs

RFC 6458 defines two APIs:

- UDP-style interface
- TCP-style interface

Preparations

- `# apt-get install libstcp-dev`
- `$ man 7 sctp`

UDP-Style API

- Outbound association setup implicit
- Typical server: socket, bind, listen, recvmsg, sendmsg, close
- Typical client: socket, sendmsg, recvmsg, close
- Here, all associations share a socket (but `man sctp_peeloff`)

SCTP UDP-style Server

```
s = socket (PF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
sctp_bindx (s, addrs, num_addrs, flags);
```

`sctp_bindx` allows binding to multiple addresses for multihoming!

flags can be `SCTP_BINDX_ADD_ADDR` or `SCTP_BINDX_REM_ADDR`.

UDP-style API: listen

- `listen` marks socket to be able to accept new associations
- UDP does not have associations — no `listen` call
- SCTP servers call `listen`, SCTP clients (typically) do not
- UDP-style SCTP associations are implicit — no `accept` call

SCTP Events

- `recvmsg` can return SCTP events (`MSG_NOTIFICATION` in flags)
- Which events are enabled controlls `setsockopt (s, IPPROTO_SCTP, ...)`
- Events include: association, data meta data, address changes, errors, shutdown
- Note that association events are enabled by default

SCTP TCP-style API

- Outbound association explicit
- Typical server: `socket`, `bind`, `listen`, `accept` (`recv`, `send`, `close`), `close`
- Typical client: `socket`, `connect`, `send`, `recv`, `close`
- New socket for each association

SCTP TCP-style Server

```
s = socket (PF_INET, SOCK_STREAM,
            IPPROTO_SCTP);
sctp_bindx (s, addrs, num_addrs, flags);
```

SCTP TCP-style API

- `send`/`recv` use primary address
- `sendmsg`/`recvmsg` can be used to send to alternative addresses and to specify which stream to use
- `sctp_sendmsg`/`sctp_recvmsg` are more convenient wrappers, use those!

Example: SCTP TCP-Style Server

```
ls = socket (PF_INET, SOCK_STREAM, IPPROTO_SCTP);
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
servaddr.sin_port = htons (MY_PORT_NUM);
bind (ls, &servaddr, sizeof (servaddr));
listen(ls, 5);
while (1) {
  connSock = accept (ls, NULL, NULL);
  sctp_sendmsg (connSock,
                message1, message1_size,
                NULL, 0, 0, 0, STREAM_NUMBER1, 0, 0);
  sctp_sendmsg (connSock,
                message2, message2_size,
                NULL, 0, 0, 0, STREAM_NUMBER2, 0, 0);
  close(connSock); }
```

Example: SCTP TCP-Style Client

```
struct sctp_sndrcvinfo sndrcvinfo;
struct sctp_event_subscribe events;

connSock = socket (PF_INET, SOCK_STREAM, IPPROTO_SCTP);
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons (MY_PORT_NUM);
servaddr.sin_addr.s_addr = inet_addr ("127.0.0.1");
connect (connSock, &servaddr, sizeof (servaddr));
memset (&events, 0, sizeof (events));
events.sctp_data_io_event = 1; /* ``on'' */
setsockopt (connSock, SOL_SCTP, SCTP_EVENTS,
            &events, sizeof (events));
while (1) {
  in = sctp_recvmsg (connSock, buffer, sizeof(buffer),
                     NULL, 0, &sndrcvinfo, &flags);
  if (sndrcvinfo.sinfo_stream == STREAM_NUMBER1) { // ... }
  if (sndrcvinfo.sinfo_stream == STREAM_NUMBER2) { // ... } }
```

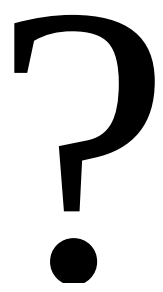
Controlling Stream Semantics

- `sctp_opt_info` is used to set SCTP socket options
- Can set default send parameters (`SCTP_DEFAULT_SNDINFO`) for stream
- For details, see RFC 6458, section 8

Important SCTP RFCs

- RFC 6458 – Sockets API Extensions for SCTP
- RFC 5062 – Attacks on SCTP and Countermeasures
- RFC 5061 – Dynamic Address Reconfiguration
- RFC 4960 – SCTP protocol (replaces RFC 3309 and RFC 2960)
- RFC 4895 – Authenticated Chunks for SCTP
- RFC 3758 – SCTP Partial Reliability Extension
- RFC 3554 – TLS over SCTP
- RFC 3286 – Introduction to SCTP

Questions



"Talk is cheap. Show me the code." – Linus Torvalds

RTFL