SRM INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

18CSC302J- Computer Networks

Unit II

SCTP Sockets
SCTP Services and Features,
Packet Format
SCTP Client/Server

Syllabus - Unit II

- Byte ordering
- Byte ordering conversion functions
- System calls
- Sockets
- System calls used with Sockets
- Iterative and concurrent server
- Socket Interface
- Structure and Functions of Socket
- Remote Procedure Call
- RPC Model, Features
- TCP Client Server Program
- Input, Output Processing Module
- UDP Client Server Program
- UDP Control block table & Module
- UDP Input & Output Module
- SCTP Sockets
- SCTP Services and Features, Packet Format
- SCTP Client/Server

SCTP Sockets, Services, Features, Packet Format, SCTP Client/Server

1. SCTP-Introduction

- SCTP is a newer transport protocol
- It was first designed to meet the needs of the growing IP telephony market; in particular, transporting telephony signaling across the Internet.
- SCTP is a reliable, message-oriented protocol, providing multiple streams between endpoints and transport-level support for multihoming.
- Since it is a newer transport protocol, it does not have the same ubiquity as TCP or UDP;
- It provides some new features that may simplify certain application designs.

Introduction

- There are some fundamental differences between SCTP and TCP,
  - ✔ the *one-to-one* interface for SCTP provides very nearly the same application interface as TCP; allows for trivial porting of applications, but does not permit use of some of SCTP's advanced features.
  - ✔ The *one-to-many interface provides full support for these features, but may require significant* retooling of existing applications.
  - ✔ The one-to-many interface is recommended for most new applications developed for SCTP.

2. SCTP Socket types

- There are two types of SCTP sockets:
  - ✔ a *one-to-one socket*
  - ✔ *a one-to-many socket.*
- *A one-to-*one socket corresponds to exactly one SCTP association.
- SCTP association is a connection between two systems, but may involve more than two IP addresses due to multihoming)
- This mapping is similar to the relationship between a TCP socket and a TCP connection.

SCTP Socket types

- With a one-to-many socket, several SCTP associations can be active on a given socket simultaneously.
- This mapping is similar to UDP socket bound to a particular port by interleaved datagrams from several remote UDP endpoints that are all simultaneously sending data.

SCTP Socket types

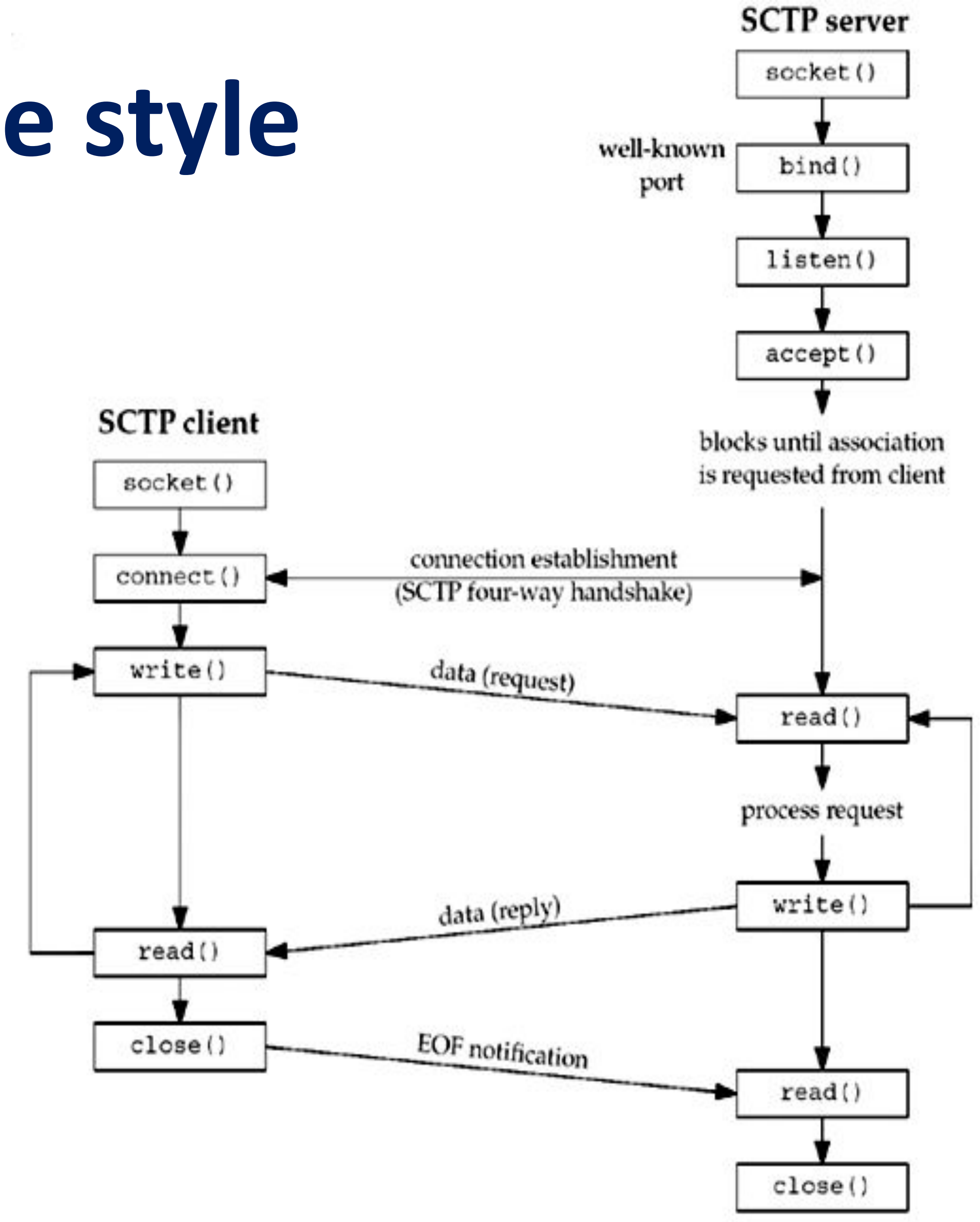
- When deciding which style of interface to use, the application needs to consider several factors, including:
  - ✔ What type of server is being written, *iterative or concurrent?*
  - ✔ How many socket descriptors does the server wish to manage?
  - ✔ Is it important to optimize the association setup to enable data on the third (and possibly fourth) packet of the four-way handshake?
  - ✔ How much connection state does the application wish to maintain?

SCTP Socket types

- Different terminology was used for the two styles of sockets
- The original term for the one-to-one socket was a "TCP-style" socket, and the original term for a one-to-many socket was a "UDP-style" socket.
- These style terms were later dropped because they tended to cause confusion by creating expectations that SCTP would behave more like TCP or UDP, depending on which style of socket was used.
- The current terminology ("one-to-one“ versus "one-to-many") focuses our attention on the key difference between the two socket styles.

a. The one-to-one style

- The one-to-one style was developed to ease the porting of existing TCP applications to SCTP.
- A one-to-one-style SCTP socket is an IP socket (family AF_INET or AF_INET6), with type SOCK_STREAM and protocol IPPROTO_SCTP.



Socket functions for SCTP one-to-one style

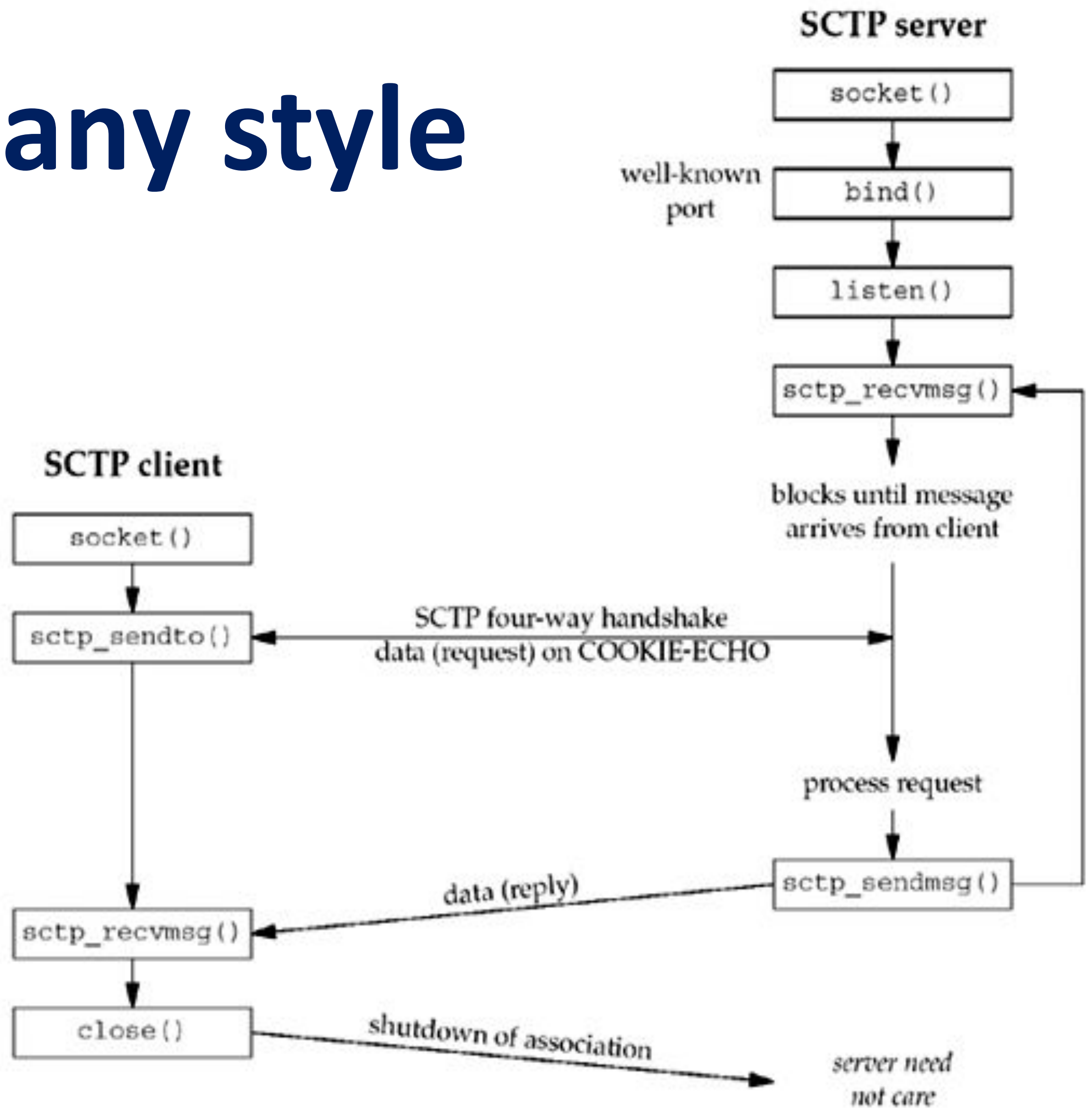
- Any socket options must be converted to the SCTP equivalent. Two common options TCP_NODELAY and TCP_MAXSEG are easily mapped to **SCTP_NODELAY** and **SCTP_MAXSEG**.
- SCTP preserves message boundaries; thus, application-layer message boundaries are not required. Eg. **write()**
- Some TCP applications use a half-close to signal the end of input to the other side. To port such applications in SCTP, the application-layer protocol will need to be rewritten.
- The **send** function can be used in the normal fashion. For the **sendto** and **sendmsg** functions, any address information included is treated as an override of the primary destination address.

Working of SCTP one-to-one style

- When the server is started, it opens a socket, binds to an address, and waits for a client connection with the **accept** system call.
- Sometime later, the client is started, it opens a socket, and initiates an association with the server.
- We assume the client sends a request to the server, the server processes the request, and the server sends back a reply to the client.
- This cycle continues until the client initiates a shutdown of the association.
- This action closes the association, where upon the server either exits or waits for a new association.

b. The one-to-many style

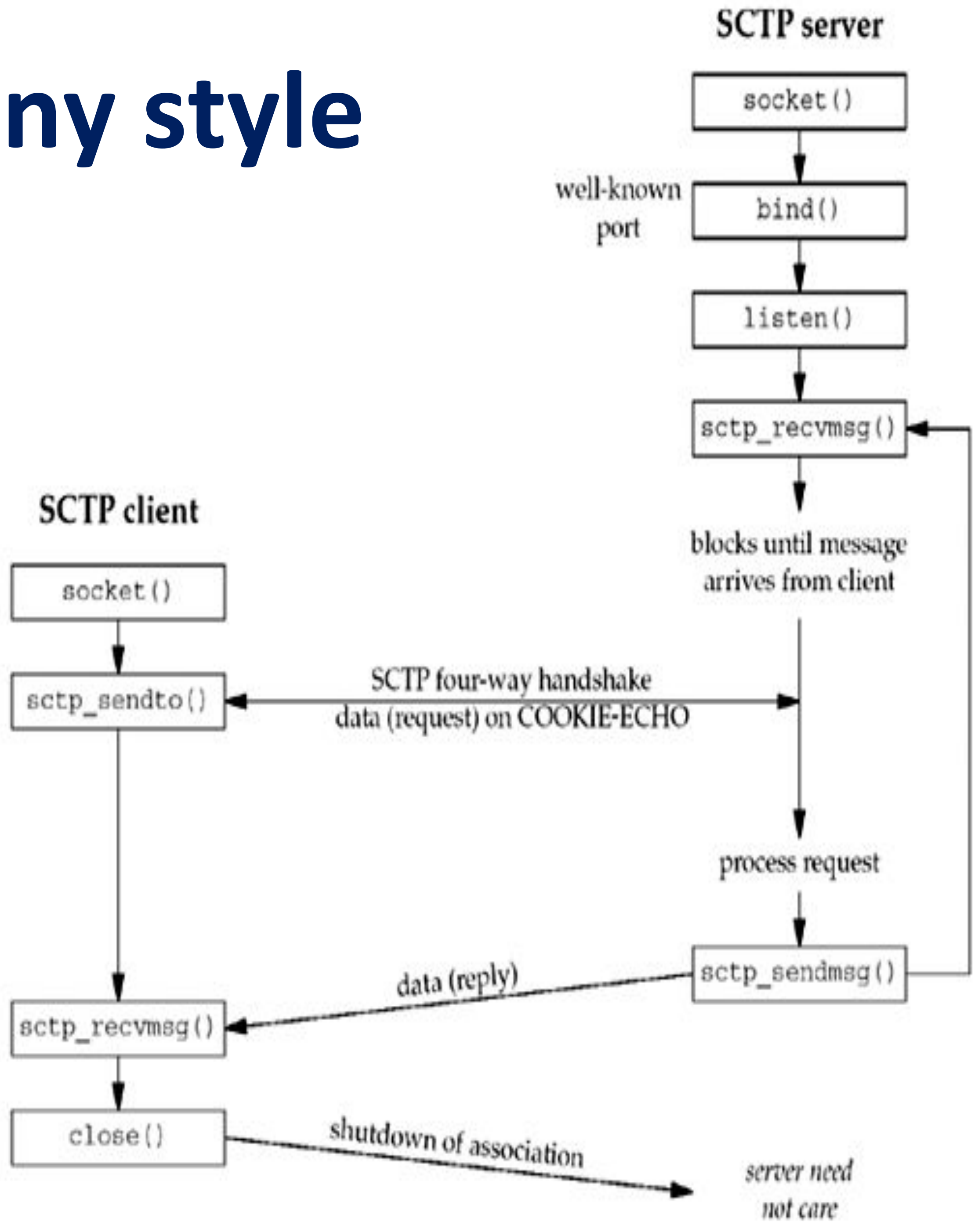
- The one-to-many style provides an application writer the ability to write a server without managing a large number of socket descriptors.
- A single socket descriptor will represent multiple associations, (same way that a UDP socket can receive messages from multiple clients).



- A one-to-many-style SCTP socket is an IP socket (family AF_INET or AF_INET6) with type SOCK_SEQPACKET and protocol IPPROTO_SCTP.

The one-to-many style

- An association identifier is used to identify a single association on a one-to-many-style socket.
- This association identifier is a value of type **sctp_assoc_t**;
- It is normally an integer. It is an opaque value; an application should not use an association identifier that it has not previously been given by the kernel.



Socket functions-SCTP one-to-many style

- When the client closes the association, the server side will automatically close as well
- Using one-to-many style is the only method that can be used to cause data to be piggybacked on the third or fourth packet of the four-way handshake
- Any **sendto, sendmsg**, or **sctp_sendmsg** to an address for which an association does not yet exist will cause an active open to be attempted, thus creating (if successful) a new association with that address. This behavior occurs even if the application doing the send has called the **listen** function to request a passive open.
- The user must use the **sendto, sendmsg**, or **sctp_sendmsg** functions, and may not use the **send** or **write** function.

Socket functions-SCTP one-to-many style

- Anytime one of the send functions is called, the primary destination address that was chosen by the system at association initiation time will be used unless the **MSG_ADDR_OVER** flag is set by the caller in a supplied **sctp_sndrcvinfo** structure.
- Association events may be enabled; if an application does not wish to receive these events, it should disable them explicitly using the **SCTP_EVENTS** socket option.
- By default, the only event that is enabled is the **sctp_data_io_event**, which provides ancillary data to the **recvmsg** and **sctp_recvmsg** call. This default setting applies to both the one-to-one and one-to-many style.

Working of SCTP one-to-many style

- First, the server is started, creates a socket, binds to an address, calls **listen** to enable client associations, and calls **sctp_recvmsg**, which blocks waiting for the first message to arrive.
- A client opens a socket and calls **sctp_sendto**, which implicitly sets up the association and piggybacks the data request to the server on the third packet of the four-way handshake.
- The server receives the request, and processes and sends back a reply.
- The client receives the reply and closes the socket, thus closing the association.
- The server loops back to receive the next message.

Working of SCTP one-to-many style

- This example shows an iterative server, where (possibly interleaved) messages from many associations (i.e., many clients) can be processed by a single thread of control.
- With SCTP, a one-to-many socket can also be used in conjunction with the **sctp_peeloff** the function to allow the iterative and concurrent server models to be combined as follows:
  - ✔ The sctp_peeloff function can be used to peel off a particular association (for example, a long-running session) from a one-to-many socket into its own one-to-one socket.
  - ✔ The one-to-one socket of the extracted association can then be dispatched to its own thread or forked process (as in the concurrent model).
  - ✔ The main thread continues to handle messages from any remaining associations in an iterative fashion on the original socket.

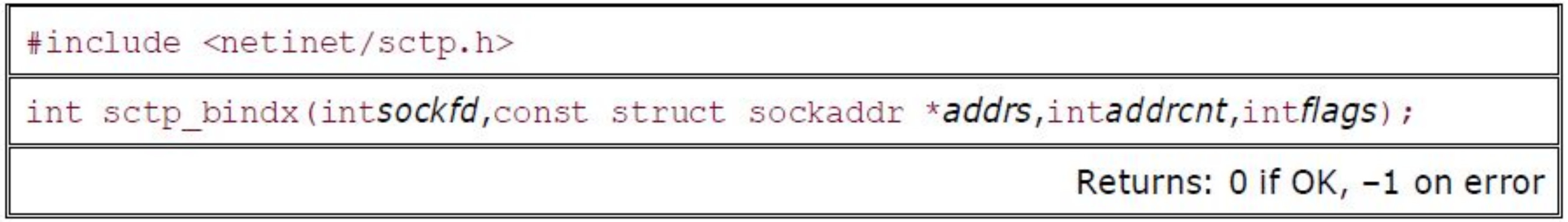
3. SCTP Socket functions

- sctp_bindx Function
- sctp_connectx Function
- sctp_getpaddrs Function
- sctp_freepaddrs Function
- sctp_getladdrs Function
- sctp_freeladdrs Function
- sctp_sendmsg Function
- sctp_recvmsg Function
- sctp_opt_info Function
- sctp_peeloff Function
- shutdown Function

a. sctp_bindx Function

- An SCTP server may wish to bind a subset of IP addresses associated with the host system.
- Traditionally, a TCP or UDP server can bind one or all addresses on a host, but they cannot bind a subset of addresses
- The **sctp_bindx** function provides more flexibility by allowing an SCTP socket to bind a particular subset of addresses.

a. sctp_bindx Function



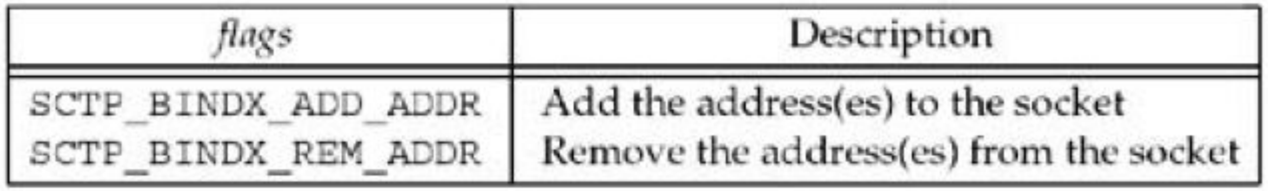
```
#include <netinet/sctp.h>

int sctp_bindx(intsockfd,const struct sockaddr *addrs,intaddrcnt,intflags);

                                                  Returns: 0 if OK, -1 on error
```

- The *sockfd* is a socket descriptor returned by the *socket* function.
- The second argument, *addrs*, is a pointer to a packed list of addresses.
- Each socket address structure is placed in the buffer immediately following the preceding socket address structure, with no intervening padding.

a. sctp_bindx Function

- The number of addresses being passed to sctp_bindx is specified by the addrcnt parameter.
- The flags parameter directs the sctp_bindx call to perform one of the two actions

flags **used with** `sctp_bindx` **function.**



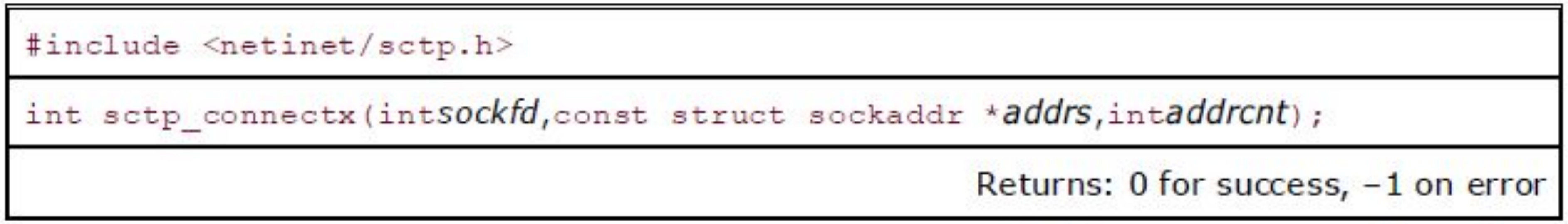
flags	Description
`SCTP_BINDX_ADD_ADDR`	Add the address(es) to the socket
`SCTP_BINDX_REM_ADDR`	Remove the address(es) from the socket

- The sctp_bindx call can be used on a bound or unbound socket.
- For an unbound socket, a call to sctp_bindx will bind the given set of addresses to the socket descriptor.
- If sctp_bindx is used on a bound socket, the call can be used with SCTP_BINDX_ADD_ADDR to associate additional addresses with the socket descriptor or with SCTP_BINDX_REM_ADDR to remove a list of addresses associated with the socket descriptor.

b. sctp_connectx Function

- The sctp_connectx function is used to connect to a multihomed peer.
- We specify addrcnt addresses, all belonging to the same peer, in the addrs parameter.

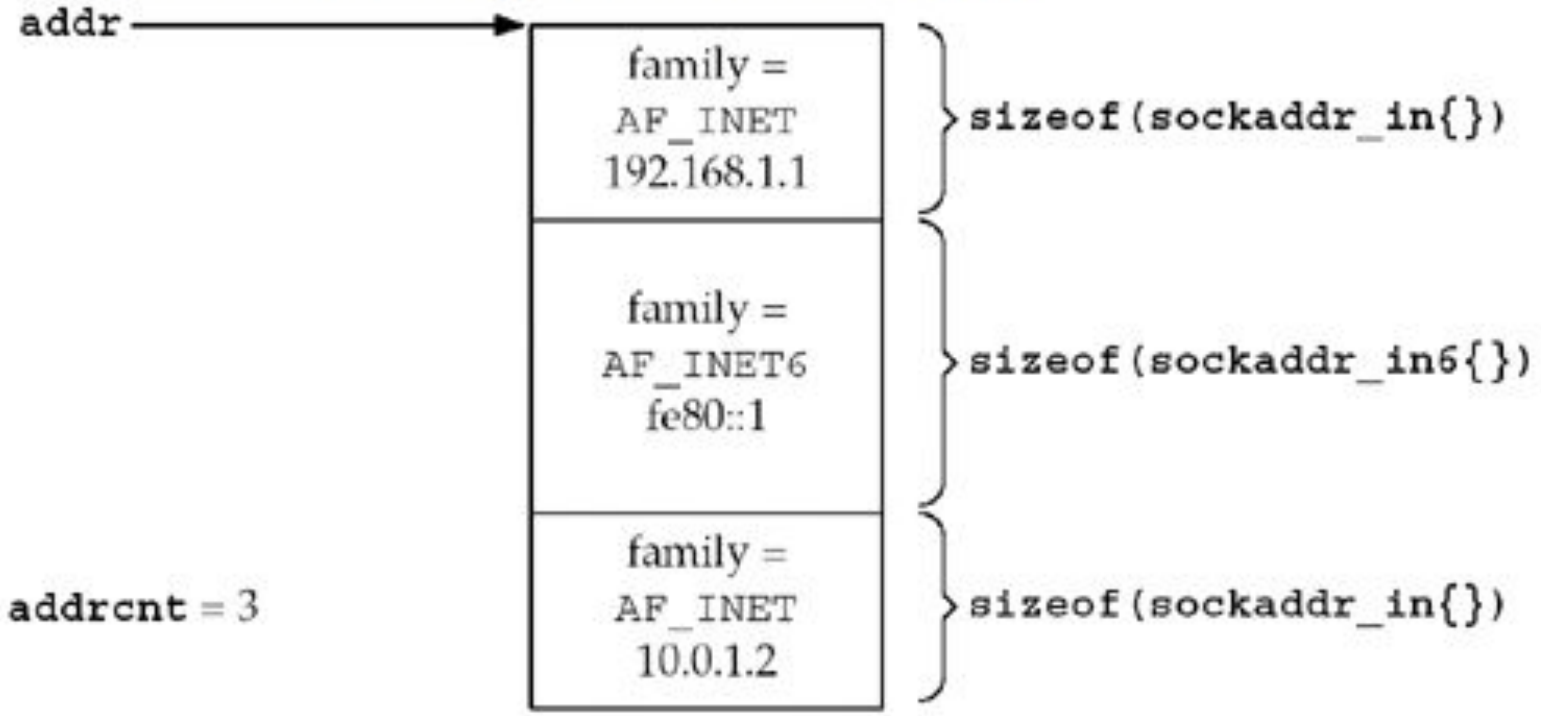
  The addrs parameter is a packed list of addresses.
- The SCTP stack uses one or more of the given addresses for establishing the association.
- All the addresses listed in addrs are considered to be valid, confirmed addresses.



```
#include <netinet/sctp.h>

int sctp_connectx(intsockfd,const struct sockaddr *addrs,intaddrcnt);

                                        Returns: 0 for success, −1 on error
```

b. sctp_connectx Function

Packed address list format for SCTP calls.

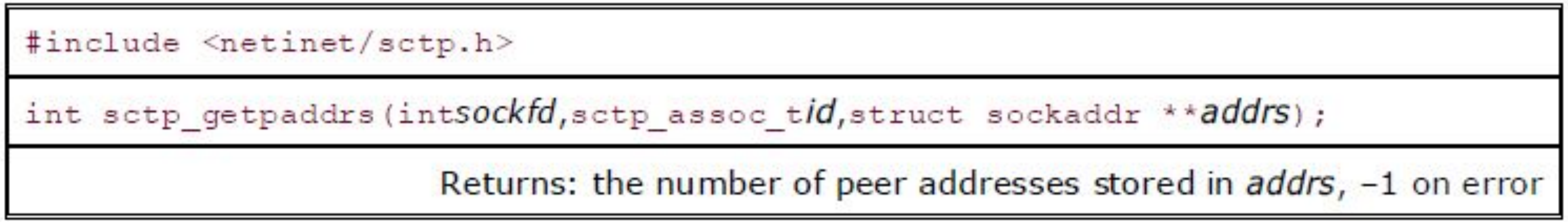


c. sctp_getpaddrs Function

- The getpeername function was not designed with the concept of a multihoming-aware transport protocol;
- when using SCTP, it only returns the primary address.
- When all the addresses are required, the sctp_getpaddrs function provides a mechanism for an application to retrieve all the addresses of a peer.

c. sctp_getpaddrs Function

- The sockfd parameter is the socket descriptor returned by the socket function.
- The id is the association identification for a one-to-many-style socket.
- If the socket is using the one-to-one style, the id field is ignored.
- addrs is the address of a pointer that sctp_getpaddrs will fill in with a locally allocated, packed list of addresses.
- The caller should use sctp_freepaddrs to free resources allocated by sctp_getpaddrs when finished with them.

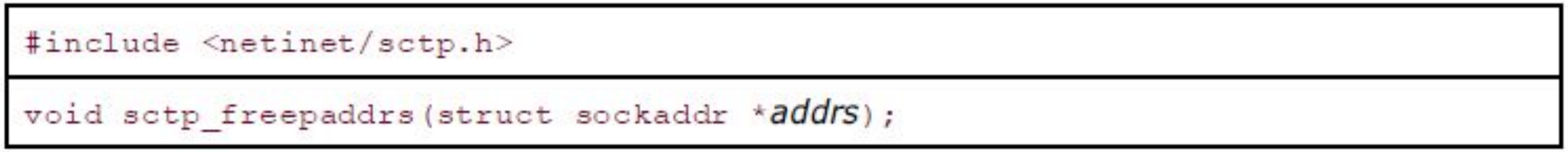


```
#include <netinet/sctp.h>

int sctp_getpaddrs(intsockfd,sctp_assoc_tid,struct sockaddr **addrs);
```

Returns: the number of peer addresses stored in *addrs*, –1 on error

d. sctp_freepaddrs Function

- The sctp_freepaddrs function frees resources allocated by the sctp_getpaddrs function.
- It is called as follows:



```
#include <netinet/sctp.h>

void sctp_freepaddrs(struct sockaddr *addrs);
```

- addrs is the pointer to the array of addresses returned by sctp_getpaddrs.

e. sctp_getladdrs Function

- The sctp_getladdrs function can be used to retrieve the local addresses that are part of an association.
- This function is often necessary when a local endpoint wishes to know exactly which local addresses are in use (which may be a proper subset of the system's addresses).

e. sctp_getladdrs Function

- The sockfd is the socket descriptor returned by the socket function. id is the association identification for a one-to-many-style socket.
- If the socket is using the one-to-one style, the id field is ignored. The addrs parameter is an address of a pointer that sctp_getladdrs will fill in with a locally allocated, packed list of addresses
- The caller should use sctp_freeladdrs to free resources allocated by sctp_getladdrs when finished with them.

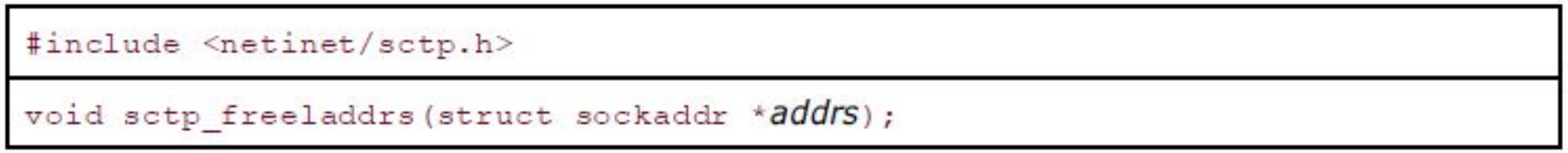
```
#include <netinet/sctp.h>
```

```
int sctp_getladdrs(intsockfd,sctp_assoc_tid,struct sockaddr **addrs);
```

Returns: the number of local addresses stored in *addrs*, –1 on error

f. sctp_freeladdrs Function

- The sctp_freeladdrs function frees resources allocated by the sctp_getladdrs function.
- It is called as follows:



```
#include <netinet/sctp.h>
```

```
void sctp_freeladdrs(struct sockaddr *addrs);
```

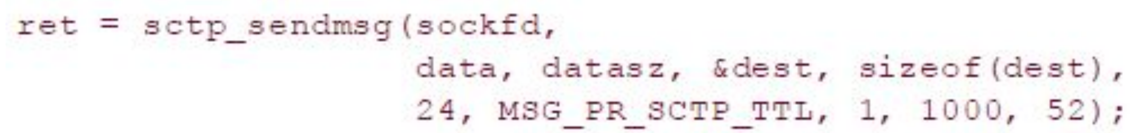
- addrs is the pointer to the array of addresses returned by sctp_getladdrs.

g. sctp_sendmsg Function

- An application can control various features of SCTP by using the sendmsg function along with ancillary data
- However, because the use of ancillary data may be inconvenient, many SCTP implementations provide an auxiliary library call (possibly implemented as a system call) that eases an application's use of SCTP's advanced features.
- The call takes the following form:

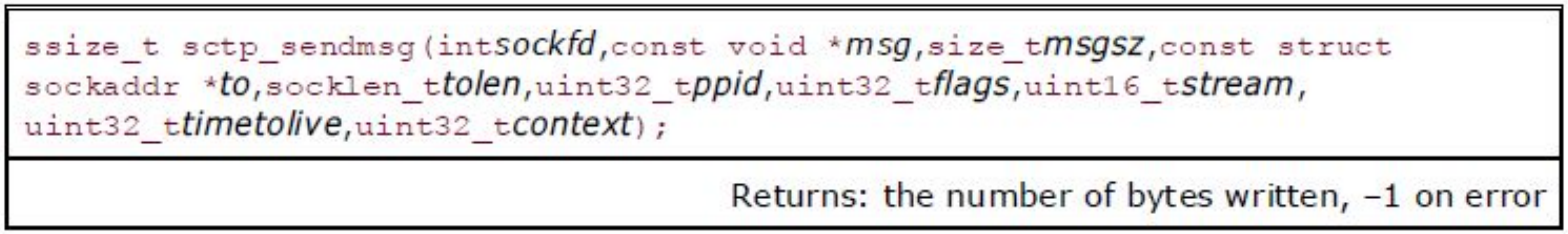
g. sctp_sendmsg Function

- The user of sctp_sendmsg has a greatly simplified sending method at the cost of more arguments.



```
ret = sctp_sendmsg(sockfd,
                   data, datasz, &dest, sizeof(dest),
                   24, MSG_PR_SCTP_TTL, 1, 1000, 52);
```

- The sockfd field holds the socket descriptor returned from a socket system call.
- The msg field points to a buffer of msgsz bytes to be sent to the peer endpoint to.
- The tolen field holds the length of the address stored in to.
- The ppid field holds the pay-load protocol identifier that will be passed with the data chunk. The flags field will be passed to the SCTP stack to identify any SCTP options;



```
ssize_t sctp_sendmsg(intsockfd,const void *msg,size_tmsgsz,const struct
sockaddr *to,socklen_ttolen,uint32_tppid,uint32_tflags,uint16_tstream,
uint32_ttimetolive,uint32_tcontext);
```

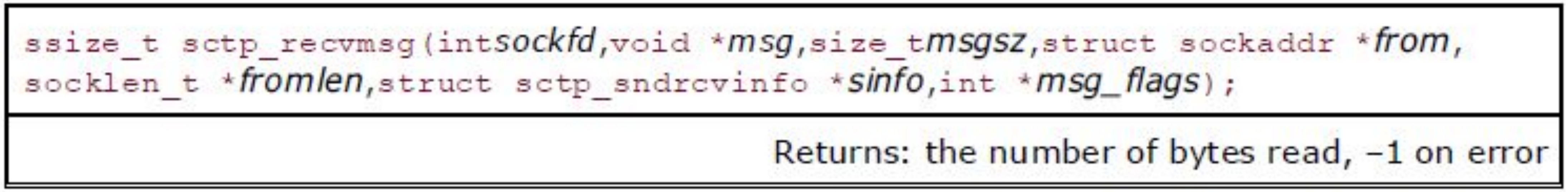
Returns: the number of bytes written, –1 on error

h. sctp_recvmsg Function

- The sctp_recvmsg function provides a more user-friendly interface to the advanced SCTP features.
- Using this function allows a user to retrieve not only its peer's address, but also the msg_flags field that would normally accompany the recvmsg function call (e.g., MSG_NOTIFICATION, MSG_EOR, etc.).
- The function also allows the user to retrieve the sctp_sndrcvinfo structure that accompanies the message that was read into the message buffer.
- If an application wishes to receive sctp_sndrcvinfo information, the sctp_data_io_event must be subscribed to with the SCTP_EVENTS socket option (ON by default).
- The sctp_recvmsg function takes the following form:

h. sctp_recvmsg Function

- eturn from this call, msg is filled with up to msgsz bytes of data.
- The message sender's address is contained in from, with the address size filled in the fromlen argument.
- Any message flags will be contained in the msg_flags argument.
- If the notification sctp_data_io_event has been enabled (the default), the sctp_sndrcvinfo structure will be filled in with detailed information about the message as well.
- If an implementation maps the sctp_recvmsg to a recvmsg function call, the flags field of the call will be set to 0.



```
ssize_t sctp_recvmsg(int sockfd, void *msg, size_t msgsz, struct sockaddr *from,
socklen_t *fromlen, struct sctp_sndrcvinfo *sinfo, int *msg_flags);
```

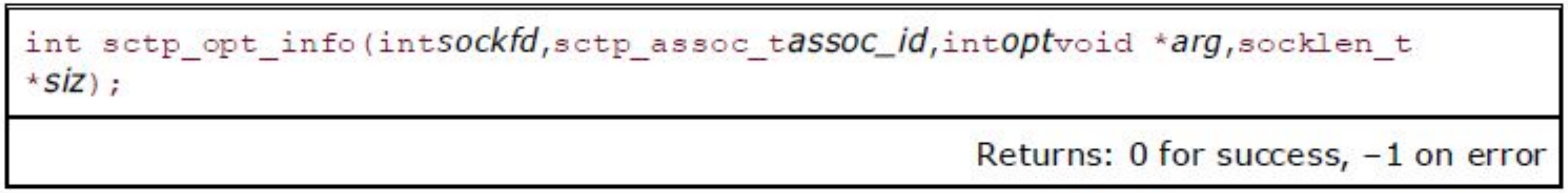
Returns: the number of bytes read, –1 on error

i. sctp_opt_info Function

- The sctp_opt_info function is provided for implementations that cannot use the getsockopt functions for SCTP.
- This inability to use the getsockopt function is because some of the SCTP socket options, for example, SCTP_STATUS, need an in-out variable to pass the association identification.
- For systems that cannot provide an in-out variable to the getsockopt function, the user will need to use sctp_opt_info.

i. sctp_opt_info Function

- sockfd is the socket descriptor that the user would like the socket option to affect.
- assoc_id is the identification of the association (if any) on which the user is performing the option.
- opt is the socket option for SCTP.
- arg is the socket option argument, and siz is a pointer to a socklen_t which holds the size of the argument..

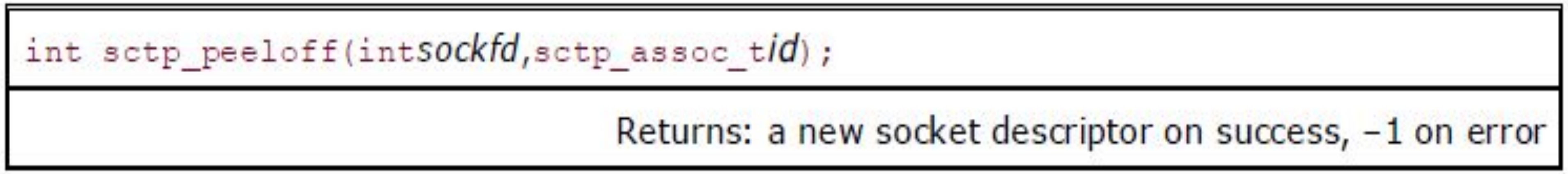


```
int sctp_opt_info(intsockfd,sctp_assoc_tassoc_id,intoptvoid *arg,socklen_t
*siz);
```

Returns: 0 for success, –1 on error

j. sctp_peeloff Function

- It is possible to extract an association contained by a one-to-many socket into an individual one-to-one-style socket.
- The semantics are much like the accept function call with an additional argument.
- The caller passes the sockfd of the one-to-many socket and the association identification id that is being extracted.
- At the completion of the call, a new socket descriptor is returned.
- This new descriptor will be a one-to-one-style socket descriptor with the requested association.



```
int sctp_peeloff(intsockfd,sctp_assoc_tid);
```

Returns: a new socket descriptor on success, –1 on error

k. shutdown Function

- SCTP's design does not provide a half-closed state, an SCTP endpoint reacts to a shutdown call differently than a TCP endpoint.
- When an SCTP endpoint initiates a shutdown sequence, both endpoints must complete transmission of any data currently in the queue and close the association.
- The endpoint that initiated the active open may wish to invoke shutdown instead of close so that the endpoint can be used to connect to a new peer.
- Unlike TCP, a close followed by the opening of a new socket is not required.
- SCTP allows the endpoint to issue a shutdown, and after the shutdown completes, the endpoint can reuse the socket to connect to a new peer.

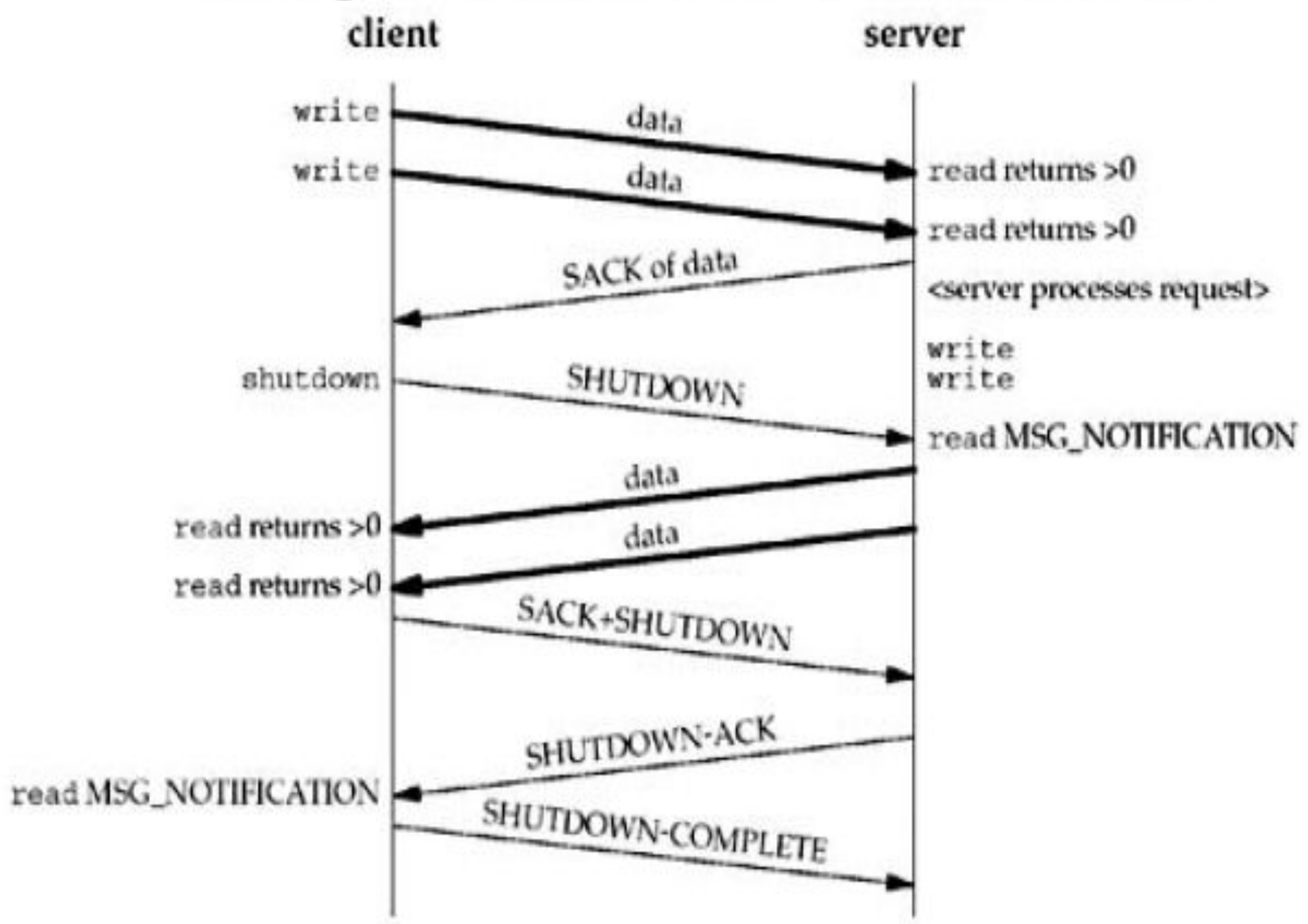
k. shutdown Function

- The new connection will fail if the endpoint does not wait until the SCTP shutdown sequence completes.
- The shutdown function how to holds

the following semantics for SCTP:

- SHUT_RD
- SHUT_WR
- SHUT_RDWR

Calling shutdown to close an SCTP association.



4. SCTP Services

a) Process-to-Process Communication

b) Multiple Streams

c) Multihoming

d) Full-Duplex Communication

e) Connection-Oriented Service

f) Reliable Service

a. ***Process-to-process communication-***SCTP uses all well-known ports in the TCP space

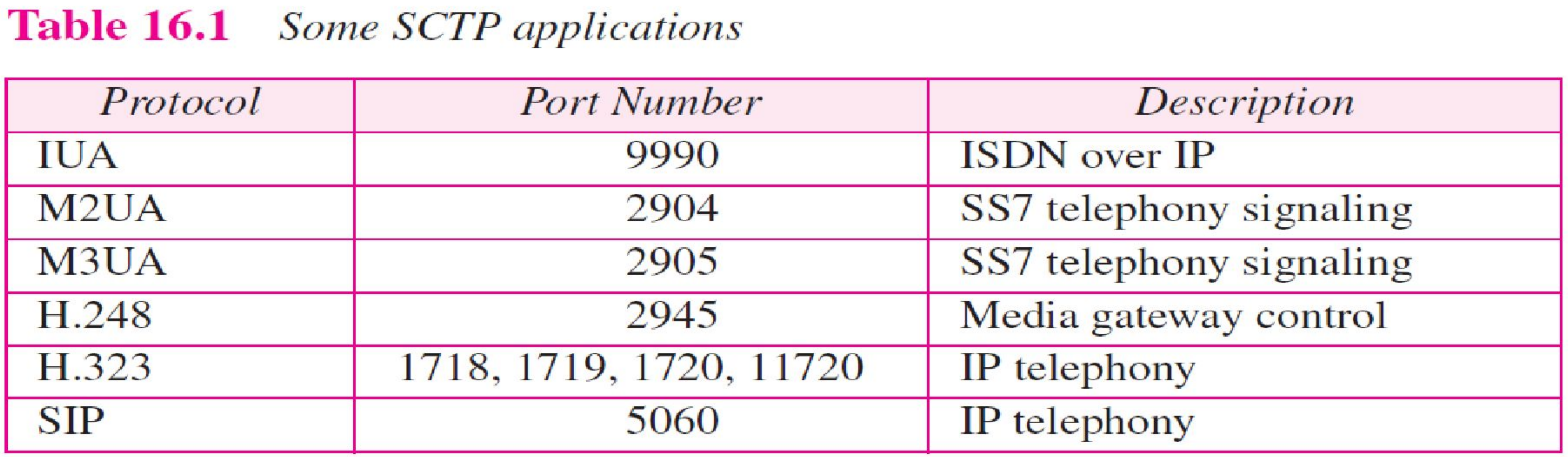


Table 16.1 *Some SCTP applications*

Protocol	*Port Number*	*Description*
IUA	9990	ISDN over IP
M2UA	2904	SS7 telephony signaling
M3UA	2905	SS7 telephony signaling
H.248	2945	Media gateway control
H.323	1718, 1719, 1720, 11720	IP telephony
SIP	5060	IP telephony

b. Multiple streams

- Each connection between a TCP client and a TCP server involves one single stream.
- Problem-A loss at any point in the stream blocks the delivery of rest of data.
- Acceptable when we are transferring text
- Again problem-when we are sending real-time data such as audio or video.
- SCTP allows **multistream service** in each connection, which is called **association** in SCTP terminology**.**
- If one of the streams is blocked, the other streams can still deliver their data.
- For example, multiple lanes on a highway. Each lane can be used for a different type of traffic.

If one of the streams is blocked, the other streams can still deliver their data.

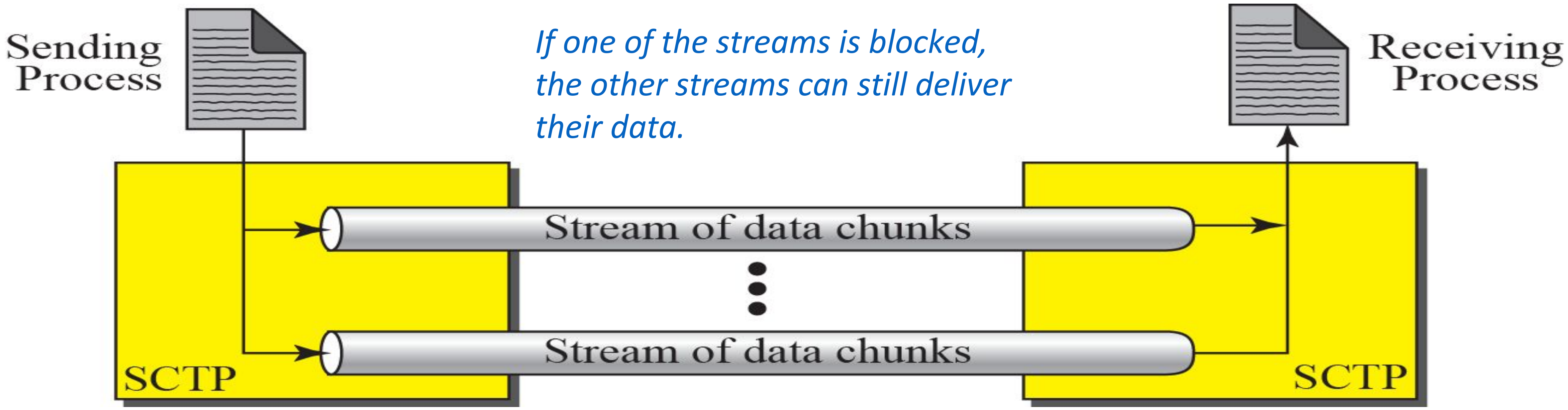


Figure -*Multiple-stream concept*

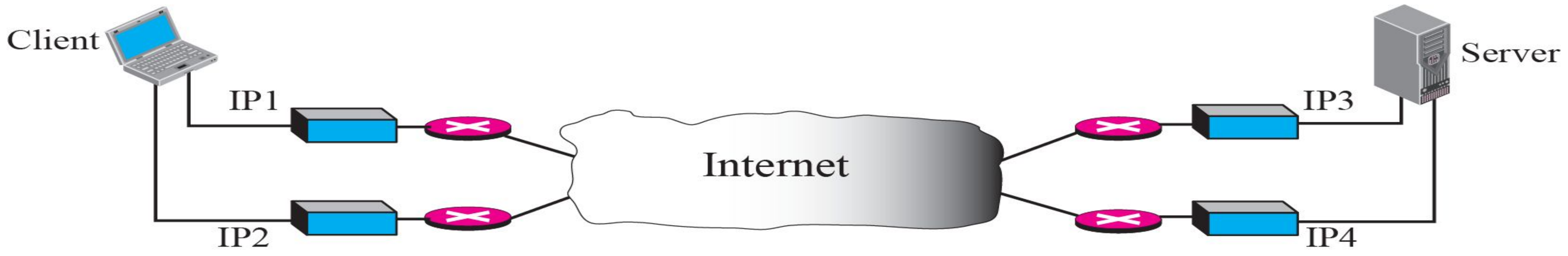
c. Multihoming

- TCP connection involves one source and one destination IP address.
- This means that even if the sender or receiver is a multihomed host (connected to more than one physical address with multiple IP addresses), only one of these IP addresses per end can be utilized during the connection.
- An SCTP association, on the other hand, supports **multihoming service.** The sending and receiving host can define multiple IP addresses in each end for an association.
- In this fault-tolerant approach, when one path fails, another interface can be used for data delivery without interruption.
- This fault-tolerant feature is very helpful when we are sending and receiving a real-time payload such as Internet telephony.

Figure 16.3 *Multihoming concept*

At present, SCTP does not allow load sharing between different path.

Currently, it is only for fault-tolerance.



d. Full-Duplex Communication

- Like TCP, SCTP offers **full-duplex service,** where data can flow in both directions at the same time.
- Each SCTP then has a sending and receiving buffer and packets are sent in both directions

e. Connection-Oriented Service

- Like TCP, SCTP is a connection-oriented protocol.
- In SCTP, a connection is called an **association.**
- When a process at site A wants to send and receive data from another process at site B, the following occurs:
  1. The two SCTPs establish an association between each other.
  2. Data are exchanged in both directions. (Data transfer)
  3. The association is terminated.

f. Reliable Service

- SCTP, like TCP, is a reliable transport protocol.
- It uses an acknowledgment mechanism to check the safe and sound arrival of data.

5. SCTP Features

a) Transmission Sequence Number (TSN)
b) Stream Identifier (SI)
c) Stream Sequence Number (SSN)
d) Packets
e) Acknowledgment Number
f) Flow Control
g) Error Control
h) Congestion Control

a. Transmission Sequence Number (TSN)

- The unit of data in TCP is a byte.
- Data transfer in TCP is controlled by numbering bytes using a sequence number.
- The unit of data in SCTP is a data chunk, which may or may not have a one-to-one relationship with the message coming from the process because of fragmentation.
- Data transfer in SCTP is controlled by numbering the data chunks.
- SCTP uses a **transmission sequence number (TSN)** to number the data chunks.
- TSNs are 32 bits long and randomly initialized between 0 and $2^{32} – 1$.
- Each data chunk must carry the corresponding TSN in its header.

b. Stream Identifier (SI)

- In TCP, there is only one stream in each connection.
- In SCTP, there may be several streams in each association.
- Each stream in SCTP needs to be identified using a **stream identifier (SI).**
- *E*ach data chunk must carry the SI in its header so that when it arrives at the destination, it can be properly placed in its stream.
- The SI is a 16-bit number starting from 0.

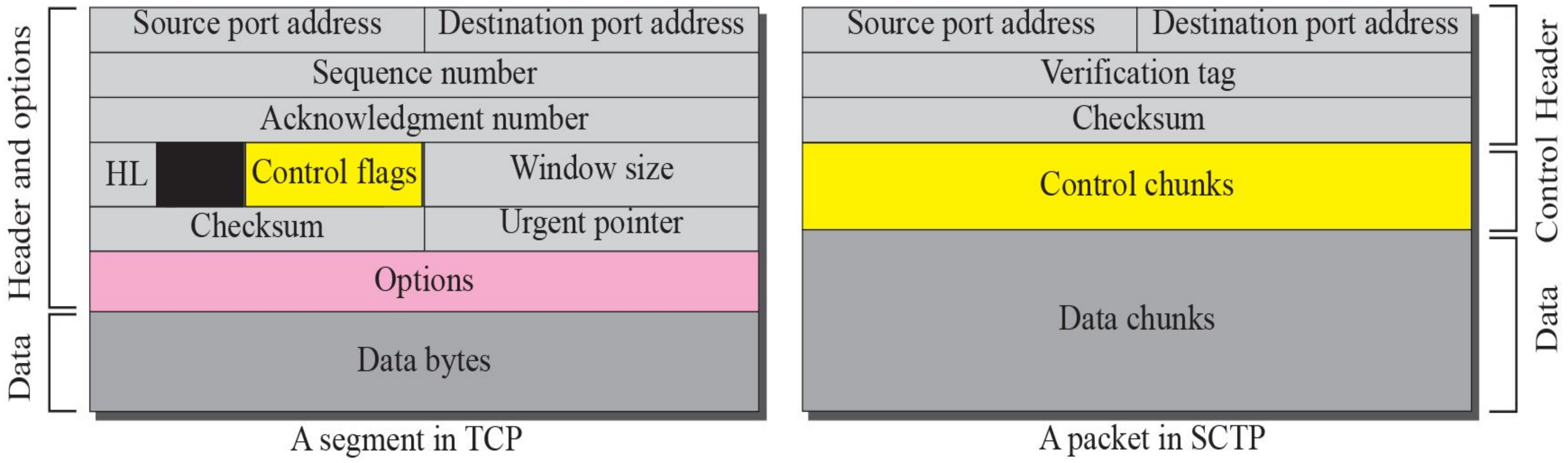
C. Stream Sequence Number (SSN)

- When a data chunk arrives at the destination SCTP, it is delivered to the appropriate stream and in the proper order.
- In addition to an SI, SCTP defines each data chunk in each stream with a **stream sequence number (SSN).**

d. Packets

- In TCP, a segment carries data and control information.
- In TCP, Data are carried as a collection of bytes; control information is defined by six control flags in the header.
- The design of SCTP is totally different: data are carried as data chunks, control information as control chunks.
- Several control chunks and data chunks can be packed together in a packet.
- A packet in SCTP plays the same role as a segment in TCP.

Figure 16.4 *Comparison between a TCP segment and an SCTP packet*



SCTP vs. TCP (1)

- Control information
  - TCP: part of the header
  - SCTP: several types of control chunks
- Data
  - TCP: one entity in a TCP segment
  - SCTP: several data chunks in a packet
- Option
  - TCP: part of the header
  - SCTP: handled by defining new chunk types

SCTP vs. TCP (2)

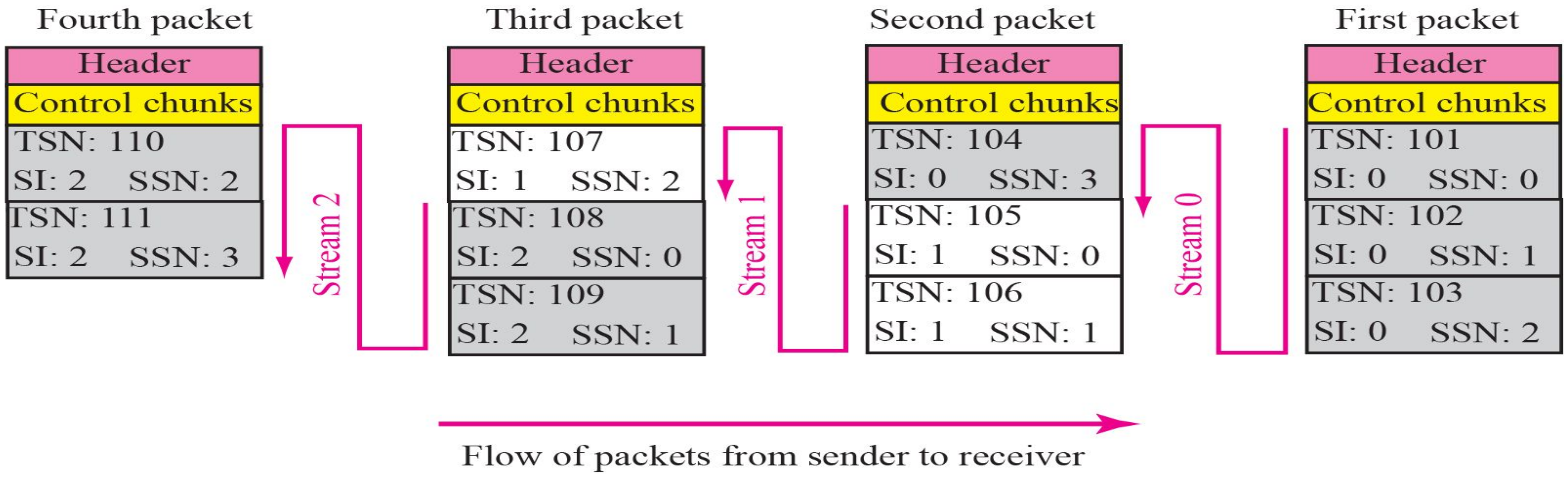
- Mandatory part of the header
  - TCP: 20 bytes, SCTP: 12 bytes
  - Reason:
    - TSN in data chunk's header
    - Ack. # and window size are part of control chunk
    - No need for header length field (∵ no option)
    - No need for an urgent pointer
- Checksum
  - TCP: 16 bits, SCTP: 32 bit

SCTP vs. TCP (3)

- Association identifier
  - TCP: none, SCTP: verification tag
  - Multihoming in SCTP
- Sequence number
  - TCP: one # in the header
  - SCTP: TSN, SI and SSN define each data chunk
  - SYN and FIN need to consume one seq. #
  - Control chunks never use a TSN, SI, or SSN number

Figure 16.5 *Packet, data chunks, and streams*

Data chunks are identified by three identifiers: TSN, SI, and SSN.
TSN is a cumulative number identifying the association;
SI defines the stream to which the chunk belongs;
SSN defines the chunk's order in a particular stream. SSN starts from 0 for each stream



e. Acknowledgment Number

- TCP Ack numbers are byte-oriented and refer to the sequence numbers.
- SCTP Ack numbers are chunk-oriented. They refer to the TSN.
- A second difference between TCP and SCTP Ack is the control information.
- To acknowledge segments that carry only control information, TCP uses a sequence number and Ack number (for example, a SYN segment needs to be acknowledged by an ACK segment).
- In SCTP, however, the control information is carried by control chunks, which do not need a TSN.
- These control chunks are acknowledged by another control chunk of the appropriate type (some need no acknowledgment).
- For example, an INIT control chunk is acknowledged by an INIT-ACK chunk.
- There is no need for a sequence number or an acknowledgment number.

f. Flow Control

•Like TCP, SCTP implements flow control to avoid overwhelming the receiver

g. Error Control

•Like TCP, SCTP implements error control to provide reliability.
•TSN numbers and acknowledgment numbers are used for error control.

h. Congestion Control

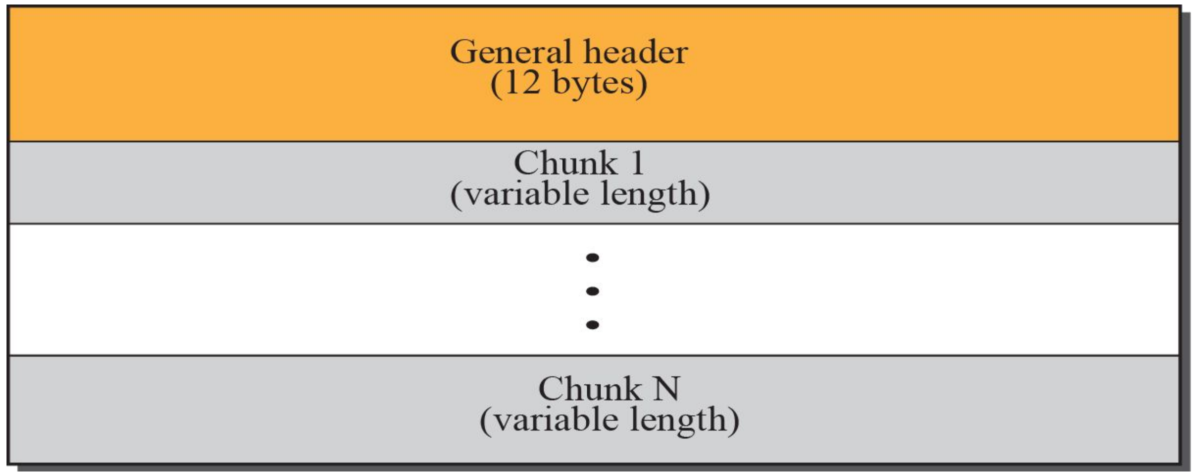
Like TCP, SCTP implements congestion control to determine how many data chunks can be injected into the network.

6. SCTP Packet Format

- An SCTP packet has a mandatory general header and a set of blocks called chunks.
- There are two types of chunks: control chunks and data chunks.
    - A control chunk controls and maintains the association;
    - A data chunk carries user data.
- In a packet, the control chunks come before the data chunks

✔ General Header

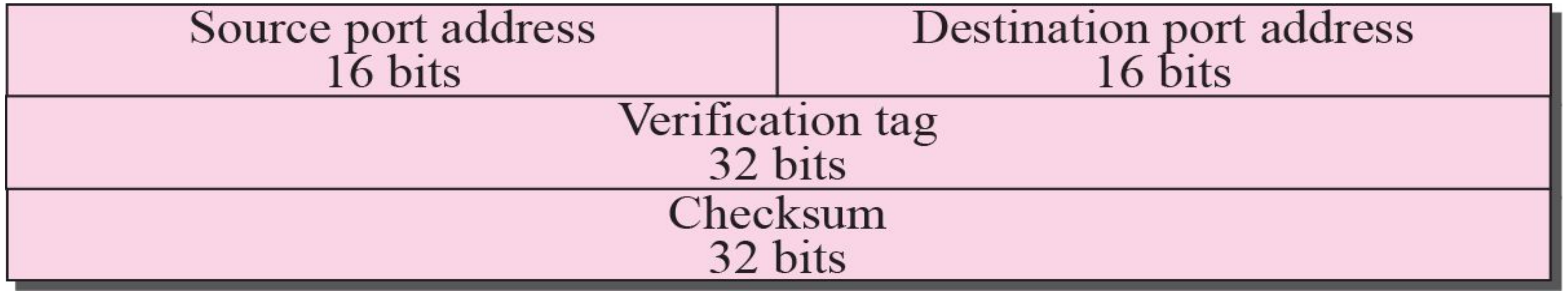
✔ Chunks



1. General Header

The general header (packet header) defines the end points of each association to which the packet belongs, guarantees that the packet belongs to a particular association, and preserves the integrity of the contents of the packet including the header itself

Verification tag□This is a number that matches a packet to an association.



2. Chunks

- Control information or user data are carried in chunks.
- The first three fields are common to all chunks; the information field depends on the type of chunk.
- SCTP requires the information section to be a multiple of 4 bytes; if not, padding bytes (eight 0s) are added at the end of the section.

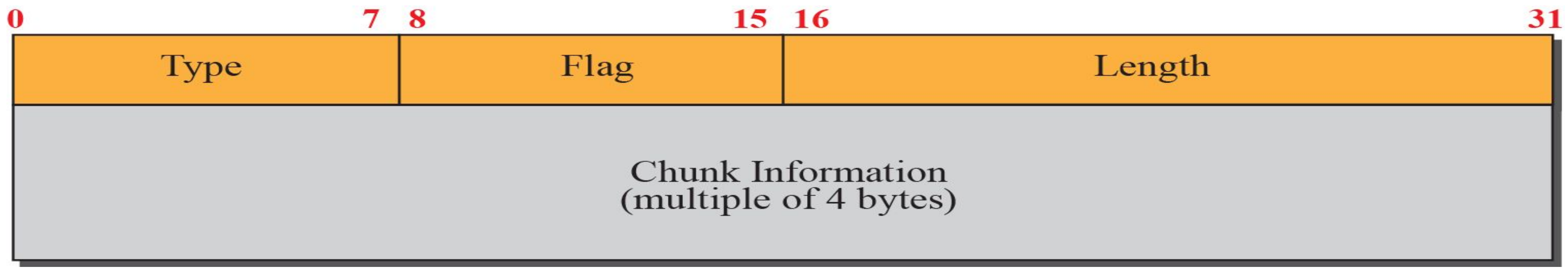
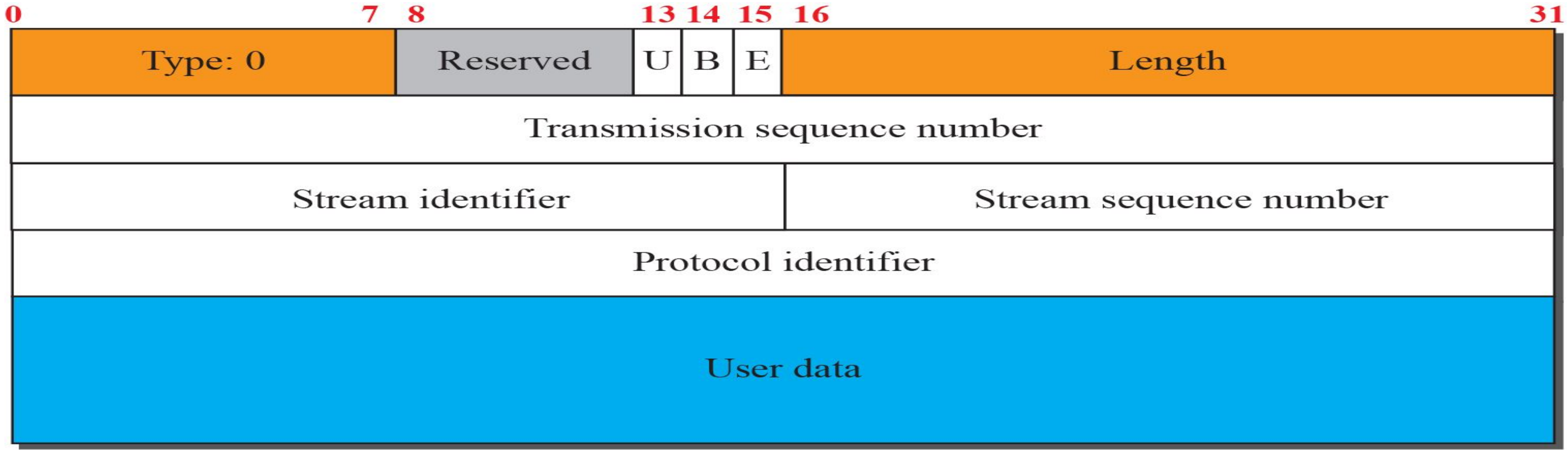


Table 16.2 *Chunks*

Type	*Chunk*	*Description*
0	DATA	User data
1	INIT	Sets up an association
2	INIT ACK	Acknowledges INIT chunk
3	SACK	Selective acknowledgment
4	HEARTBEAT	Probes the peer for liveliness
5	HEARTBEAT ACK	Acknowledges HEARTBEAT chunk
6	ABORT	Abort an association
7	SHUTDOWN	Terminates an association
8	SHUTDOWN ACK	Acknowledges SHUTDOWN chunk
9	ERROR	Reports errors without shutting down
10	COOKIE ECHO	Third packet in association establishment
11	COOKIE ACK	Acknowledges COOKIE ECHO chunk
14	SHUTDOWN COMPLETE	Third packet in association termination
192	FORWARD TSN	For adjusting cumulating TSN

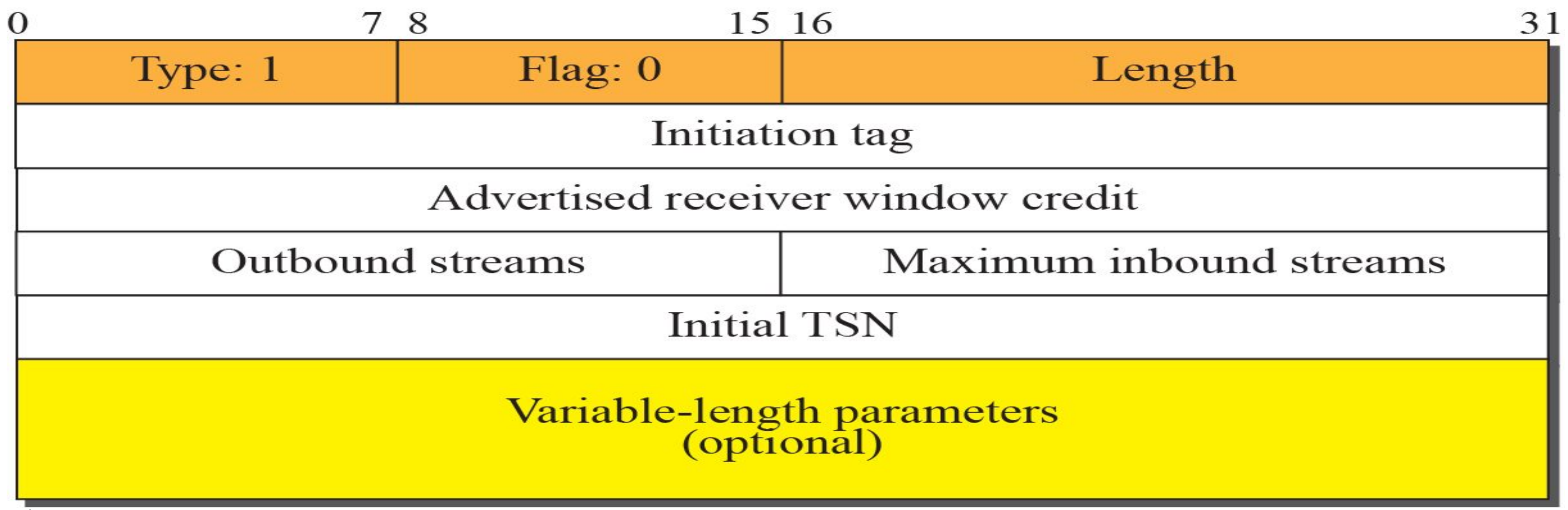
a. DAT A

- The DATA chunk carries the user data. A packet may contain zero or more data chunks.
- A DATA chunk cannot carry data belonging to more than one message, but a message can be split into several chunks. The data field of the DATA chunk must carry at least one byte of data.



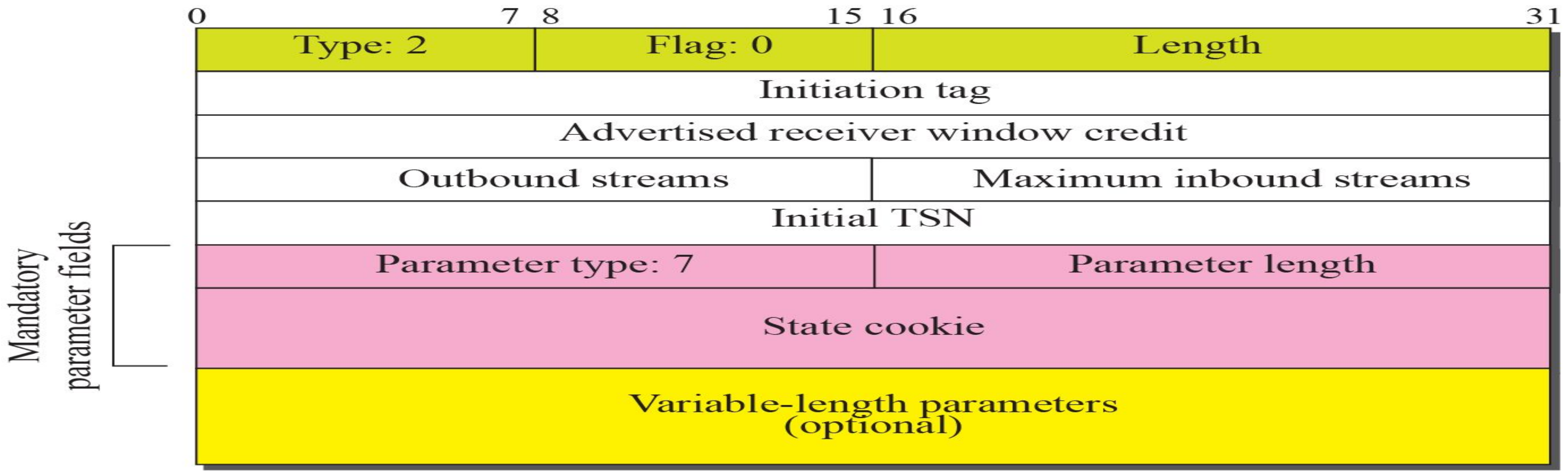
b. INIT

•The INIT chunk (initiation chunk) is the first chunk sent by an end point to establish an association.
•The packet that carries this chunk cannot carry any other control or data
•chunks. The value of the verification tag for this packet is 0, which means no tag has yet been defined.



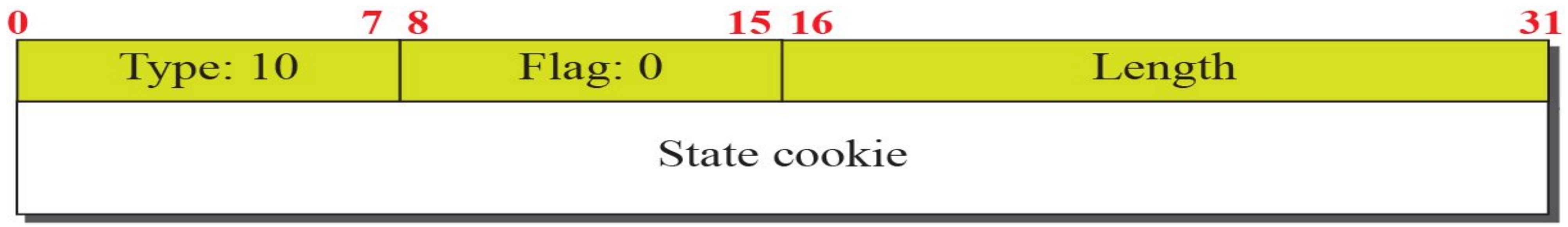
c. INIT ACK

- The INIT ACK chunk (initiation acknowledgment chunk) is the second chunk sent during association establishment.
- The packet that carries this chunk cannot carry any other control or data chunks. The value of the verification tag for this packet (located in the general header) is the value of the initiation tag defined in the received INIT chunk.



d. COOKIE ECHO

The COOKIE ECHO chunk is the third chunk sent during association establishment.
It is sent by the end point that receives an INIT ACK chunk (normally the sender of the INIT chunk).

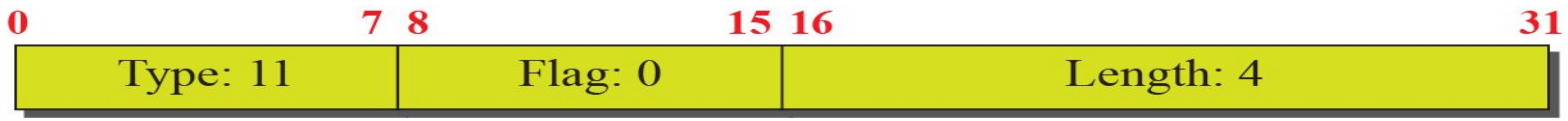


e. COOKIE ACK

•The COOKIE ACK chunk is the fourth and last chunk sent during association establishment.

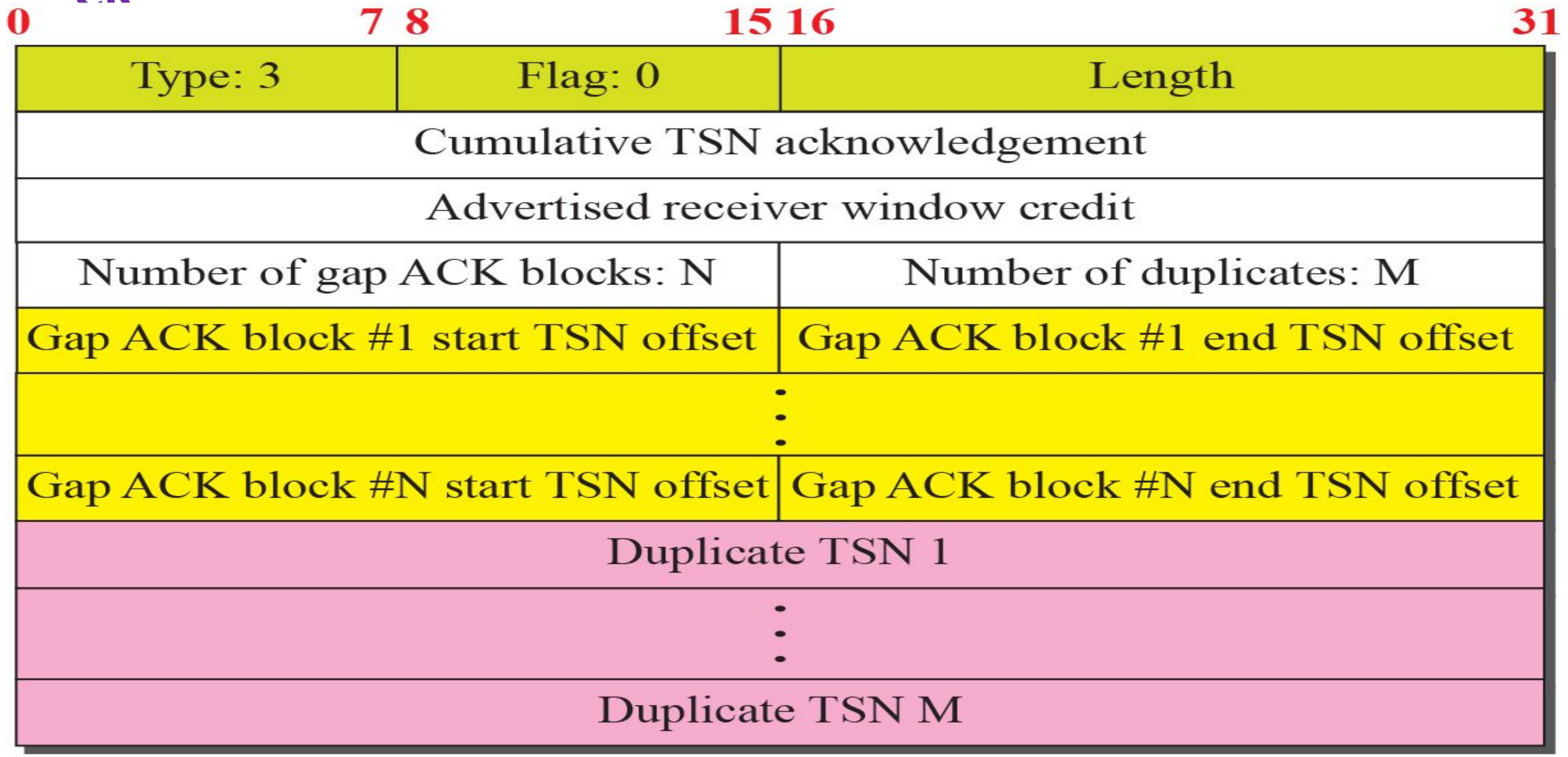
•It is sent by an end point that receives a COOKIE ECHO chunk. The packet

•that carries this chunk can also carry user data.



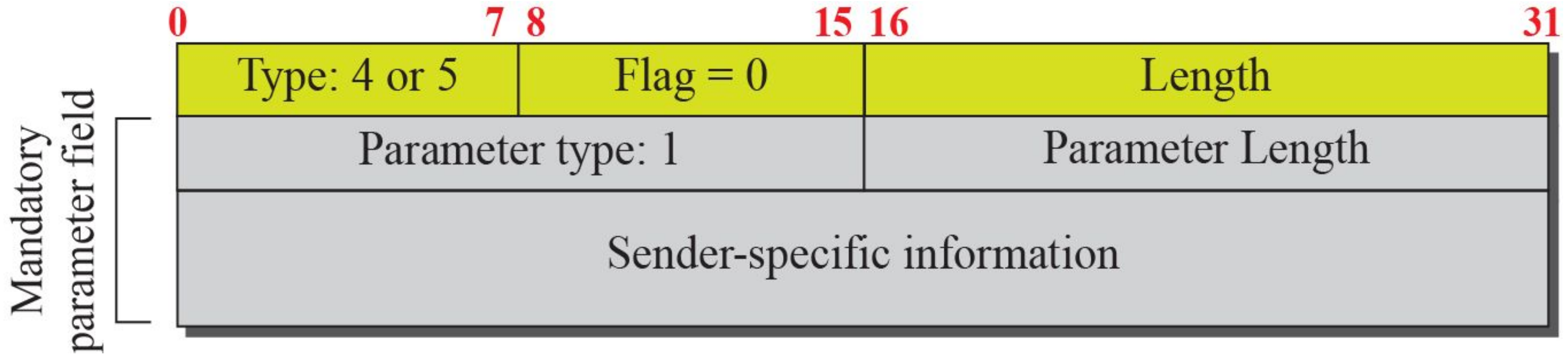
f. SACK

- The SACK chunk (selective ACK chunk) acknowledges the receipt of data packets.


0
7 8
15 16
31
Type: 3
Flag: 0
Length
Cumulative TSN acknowledgement
Advertised receiver window credit
Number of gap ACK blocks: N
Number of duplicates: M
Gap ACK block #1 start TSN offset
Gap ACK block #1 end TSN offset
Gap ACK block #N start TSN offset
Gap ACK block #N end TSN offset
Duplicate TSN 1
Duplicate TSN M

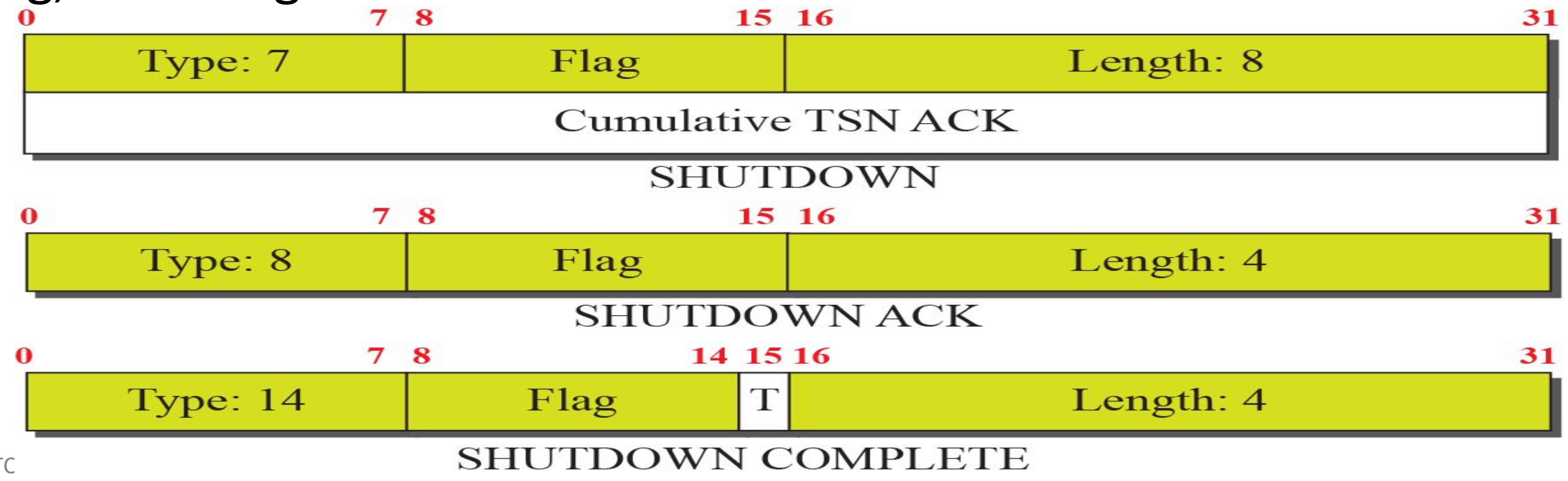
g. HEARTBEAT and HEARTBEAT ACK

The HEARTBEAT chunk and HEARTBEAT ACK chunk are similar except for the type field. These two chunks are used to periodically probe the condition of an association.



h. SHUTDOWN, SHUTDOWN ACK, and SHUTDOWN COMPLETE

- These three chunks (used for closing an association) are similar.
- The SHUTDOWN chunk, type 7, is eight bytes in length; the second four bytes define the cumulative TSN.
- The SHUTDOWN ACK chunk, type 8, is four bytes in length.
- The SHUTDOWN COMPLETE chunk, type 14, is also 4 bytes long, and has a one bit flag, the T flag.



i. ERROR

- The ERROR chunk is sent when an end point finds some error in a received packet.
- Note that the sending of an ERROR chunk does not imply the aborting of the association. (This would require an ABORT chunk.)

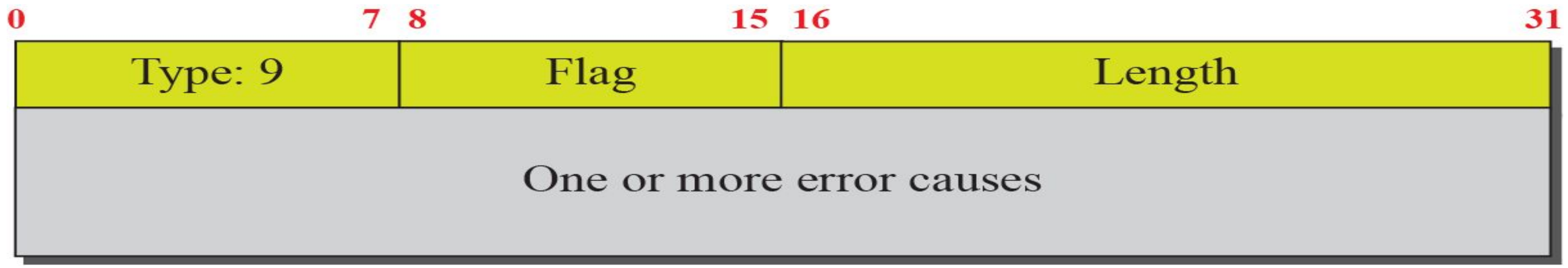
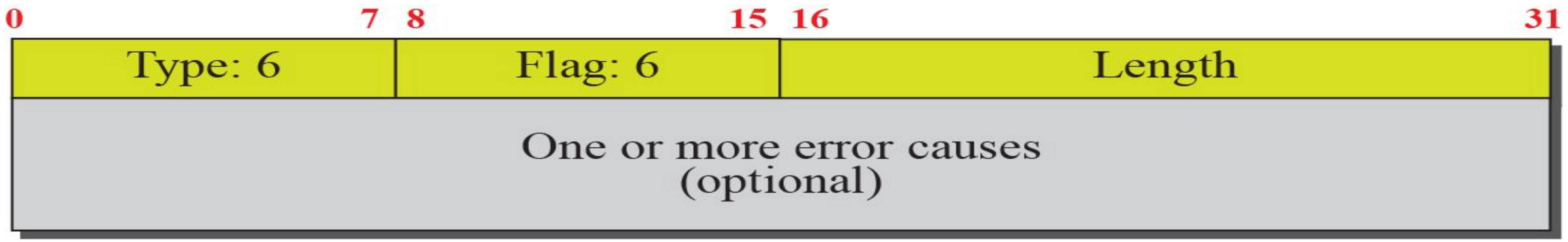


Table 16.3 *Errors*

Code	*Description*
1	Invalid stream identifier
2	Missing mandatory parameter
3	State cookie error
4	Out of resource
5	Unresolvable address
6	Unrecognized chunk type
7	Invalid mandatory parameters
8	Unrecognized parameter
9	No user data
10	Cookie received while shutting down

j. ABORT

- The ABORT chunk is sent when an end point finds a fatal error and needs to abort the association.
- The error types are the same as those for the ERROR chunk



k. Forward TSN Chunk

- This is a chunk recently added to the standard to inform the receiver to adjust its cumulative TSN.
- It provides partial reliable service.

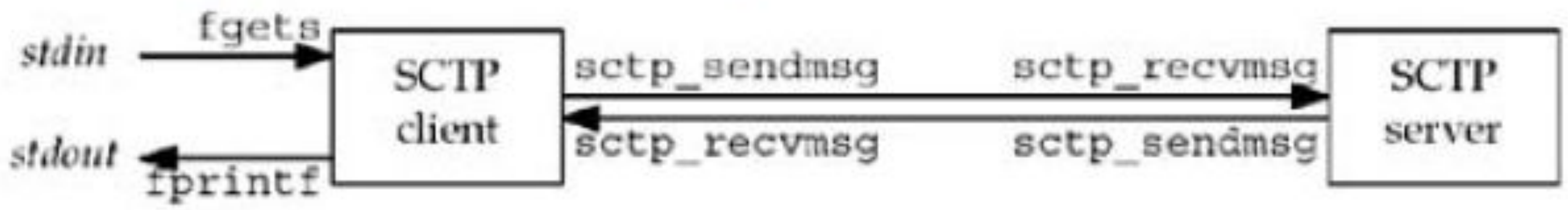
7. SCTP Client/Server

- To write a complete one-to-many SCTP client/server example.
- A client reads a line of text from standard input and sends the line to the server. The line follows the form [#] text, where the number in brackets is the SCTP stream number on which the text message should be sent.
- The server receives the text message from the network, increases the stream number on which the message arrived by one, and sends the text message back to the client on this new stream number.
- The client reads the echoed line and prints it on its standard output, displaying the stream number, stream sequence number, and text string.

SCTP Client/Server

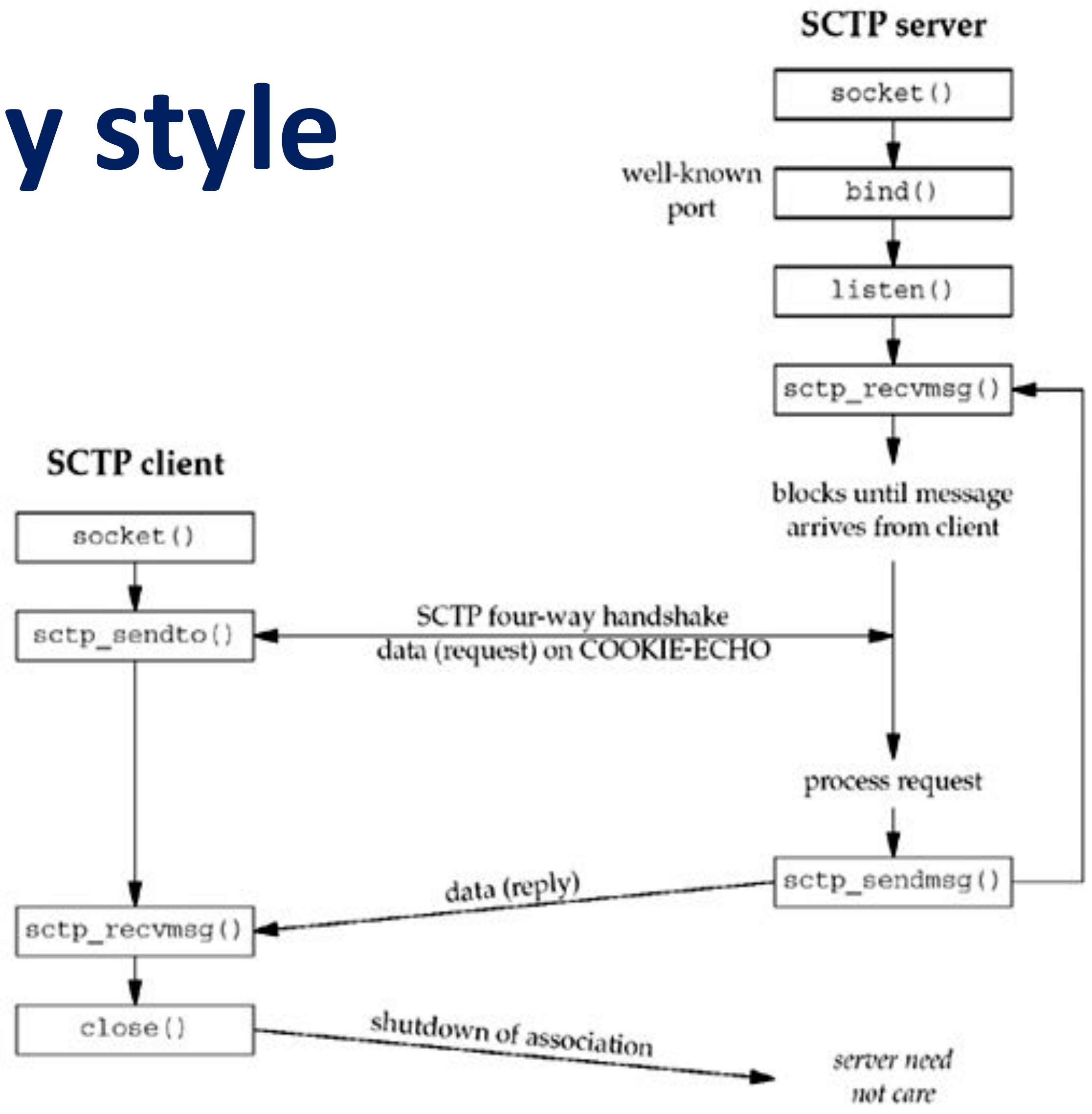
- We show two arrows between the client and server depicting two unidirectional streams being used, even though the overall association is full-duplex.
- The fgets and fputs functions are from the standard I/O library.
- We do not use the writen and readline functions since they are unnecessary.
- Instead, we use the sctp_sendmsg and sctp_recvmsg functions

Simple SCTP streaming echo client and server.



The one-to-many style

- The one-to-many style provides an application writer the ability to write a server without managing a large number of socket descriptors.
- A single socket descriptor will represent multiple associations, (same way that a UDP socket can receive messages from multiple clients).



- A one-to-many-style SCTP socket is an IP socket (family AF_INET or AF_INET6) with type SOCK_SEQPACKET and protocol IPPROTO_SCTP.

SCTP One-to-Many-Style Streaming Echo Server: main Function

```c
#include     "unp.h"

int
main(int argc, char **argv)
{
    int     sock_fd, msg_flags;
    char    readbuf [BUFFSIZE];
    struct sockaddr_in servaddr, cliaddr;
    struct sctp_sndrcvinfo sri;
    struct sctp_event_subscribe evnts;
    int     stream_increment = 1;
    socklen_t len;
    size_t rd_sz;

    if (argc == 2)
        stream_increment = atoi (argv[1]);
    sock_fd = Socket (AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
    bzero (&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
    servaddr.sin_port = htons (SERV_PORT);

    Bind (sock_fd, (SA *) &servaddr, sizeof (servaddr));

    bzero (&evnts, sizeof (evnts)) ;
    evnts.sctp_data_io_event = 1;
    Setsockopt (sock_fd, IPPROTO_SCTP, SCTP_EVENTS, &evnts, sizeof (evnts)) ;

    Listen(sock_fd, LISTENQ) ;
    for ( ; ; ) {
        len = sizeof(struct sockaddr_in) ;
        rd_sz = Sctp_recvmsg(sock_fd, readbuf, sizeof (readbuf) ,
                              (SA *) &cliaddr, &len, &sri, &msg_flags) ;
        if (stream_increment) {
            sri.sinfo_stream++;
            if (sri.sinfo_stream >=
               sctp_get_no_strms (sock_fd, (SA *) &cliaddr, len) )
               sri.sinfo_stream = 0;
        }
        Sctp_sendmsg (sock_fd, readbuf, rd_sz,
                      (SA *) &cliaddr, len,
                      sri.sinfo_ppid,
                      sri.sinfo_flags, sri.sinfo_stream, 0, 0) ;
    }
}
```

SCTP streaming echo server

SCTP One-to-Many-Style Streaming Echo Server: main Function

- Set stream increment option

If command line option is passed to program, program decides whether to increment stream number of incoming messages

```
if (argc == 2)
stream_increment = atoi(argv[1]);
```

- Create an SCTP socket

```
sock_fd = Socket(AF_INET, SOCK_SEQPACKET,
  IPPROTO_SCTP);
```

- Bind an address

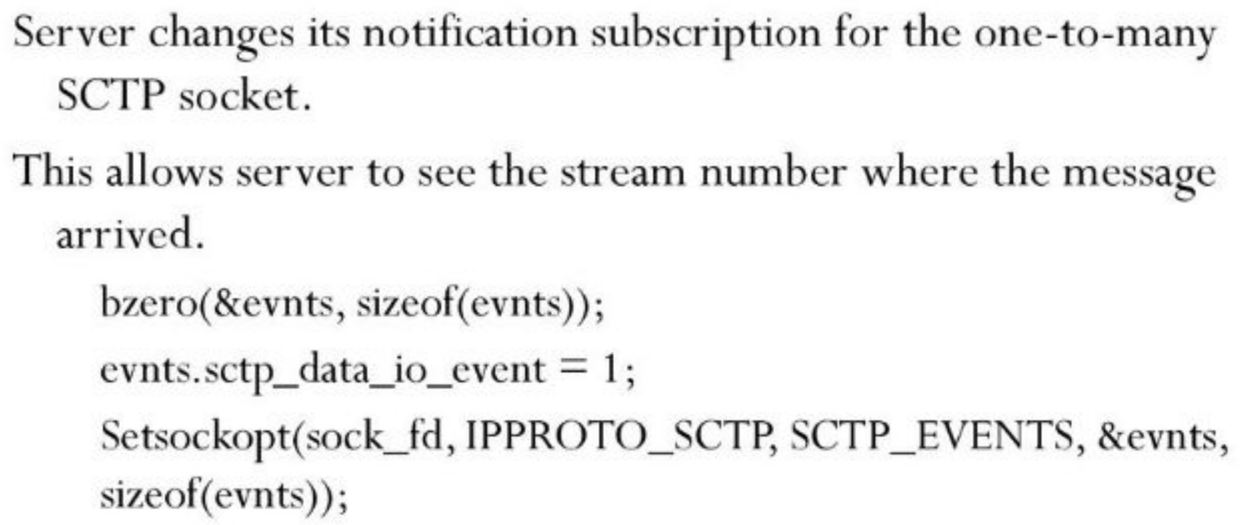
Bind any incoming address to certain server port.

```
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(SERV_PORT);
Bind(sock_fd, (SA *) &servaddr, sizeof(servaddr));
```

Server: Create an SCTP socket

SCTP One-to-Many-Style Streaming Echo Server: main Function

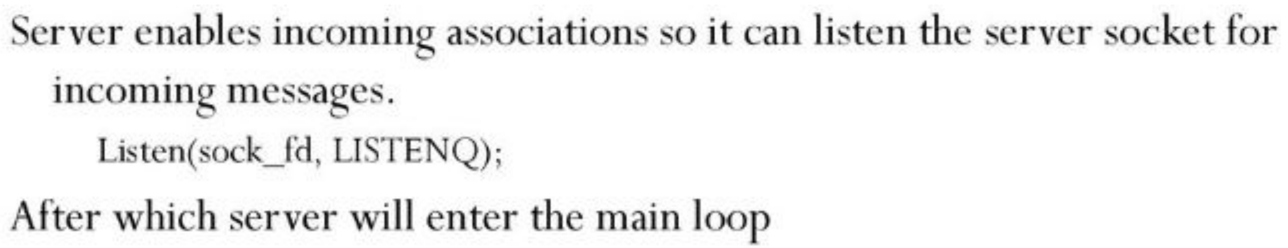
- Set up for notifications of interest


Server changes its notification subscription for the one-to-many SCTP socket.

This allows server to see the stream number where the message arrived.

```
bzero(&evnts, sizeof(evnts));
evnts.sctp_data_io_event = 1;
Setsockopt(sock_fd, IPPROTO_SCTP, SCTP_EVENTS, &evnts,
sizeof(evnts));
```

- Enable incoming associations

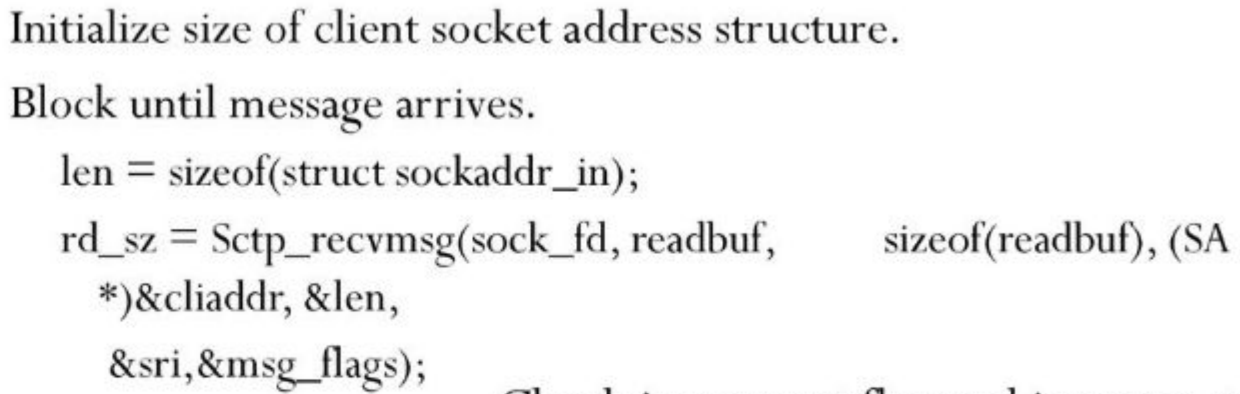

Server enables incoming associations so it can listen the server socket for incoming messages.

```
Listen(sock_fd, LISTENQ);
```

After which server will enter the main loop

SCTP One-to-Many-Style Streaming Echo Server: main Function

- **Wait for message**

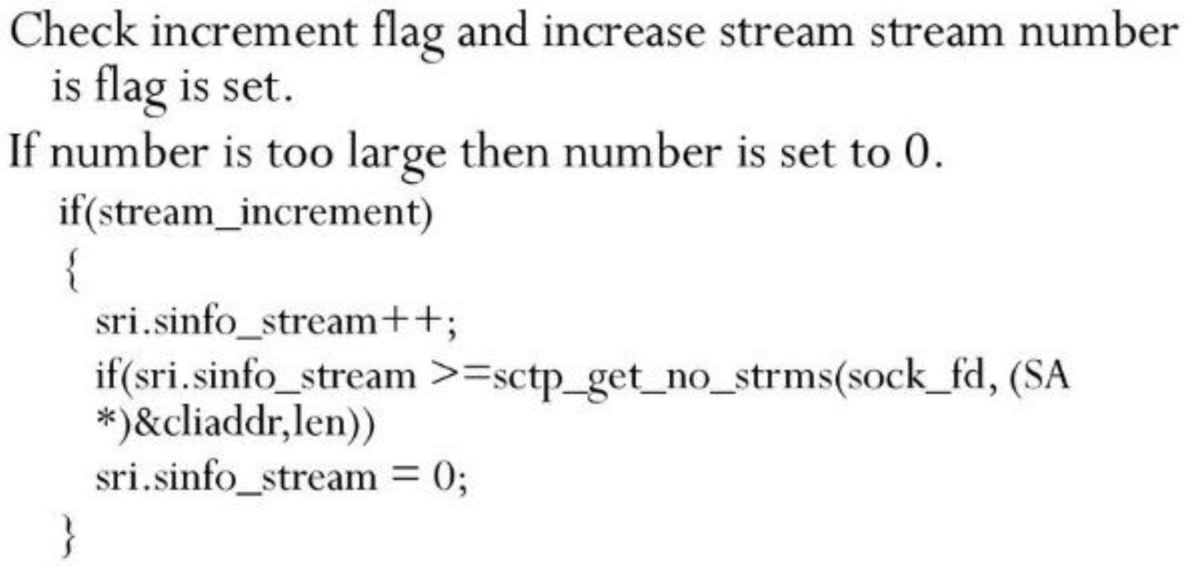


Initialize size of client socket address structure.

Block until message arrives.

```
len = sizeof(struct sockaddr_in);
rd_sz = Sctp_recvmsg(sock_fd, readbuf,        sizeof(readbuf), (SA
   *)&cliaddr, &len,
    &sri,&msg_flags);
```

- **Increment stream number if desired**



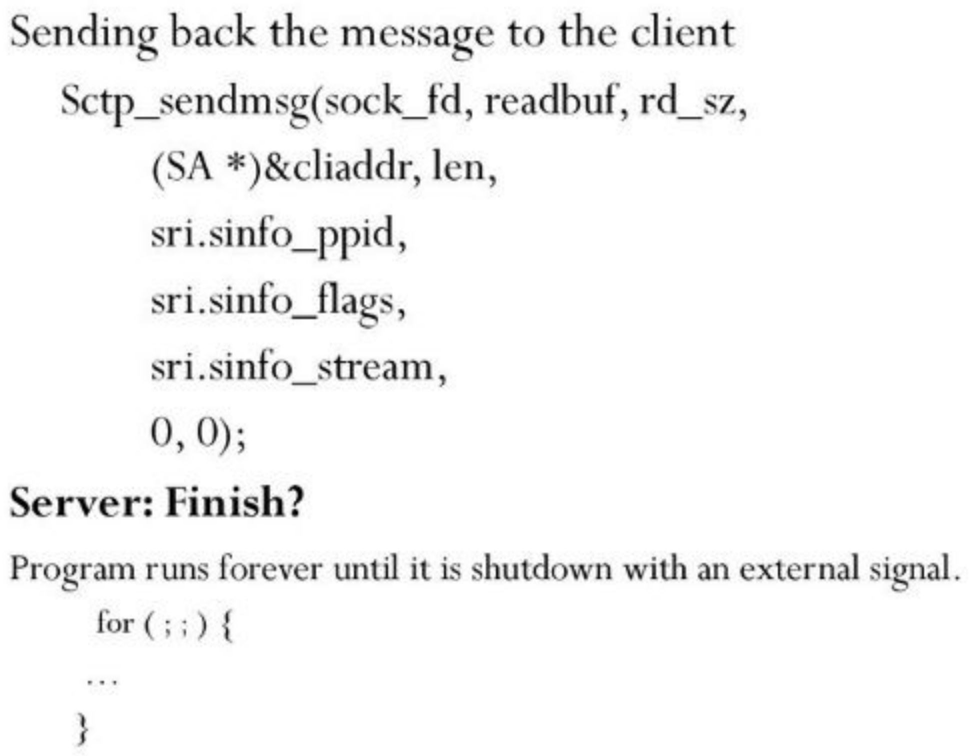
Check increment flag and increase stream stream number is flag is set.

If number is too large then number is set to 0.

```
if(stream_increment)
{
   sri.sinfo_stream++;
   if(sri.sinfo_stream >=sctp_get_no_strms(sock_fd, (SA
   *)&cliaddr,len))
   sri.sinfo_stream = 0;
}
```

SCTP One-to-Many-Style Streaming Echo Server: main Function

- **Send back response**



Sending back the message to the client

```
Sctp_sendmsg(sock_fd, readbuf, rd_sz,
        (SA *)&cliaddr, len,
        sri.sinfo_ppid,
        sri.sinfo_flags,
        sri.sinfo_stream,
        0, 0);
```

Server: Finish?

Program runs forever until it is shutdown with an external signal.

```
for ( ; ; ) {
...
}
```

SCTP One-to-Many-Style Streaming Echo Client: main Function

```c
#include     "unp.h"

int
main(int argc, char **argv)
{
    int      sock_fd;
    struct sockaddr_in servaddr;
    struct sctp_event_subscribe evnts;
    int      echo_to_all = 0;
    if (argc < 2)
        err_quit("Missing host argument - use '%s host [echo] '\n", argv[0]) ;
    if (argc > 2) {
        printf("Echoing messages to all streams\n") ;
        echo_to_all = 1;
    }
    sock_fd = Socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
    bzero(&servaddr, sizeof (servaddr) ) ;
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
    servaddr.sin_port = htons (SERV_PORT);
    Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

    bzero(&evnts, sizeof (evnts)) ;
    evnts.sctp_data_io_event = 1 ;
    Setsockopt(sock_fd, IPPROTO_SCTP, SCTP_EVENTS, &evnts, sizeof (evnts)) ;
    if (echo_to_all == 0)
        sctpstr_cli (stdin, sock_fd, (SA *) &servaddr, sizeof (servaddr)) ;
    else
        sctpstr_cli_echoall(stdin, sock_fd, (SA *) &servaddr,
                            sizeof (servaddr)) ;
    Close (sock_fd) ;
    return (0) ;
}
```

SCTP streaming echo client main

SCTP One-to-Many-Style Streaming Echo Client: main Function

- Validate arguments and create a socket

Arguments: host and echo flag.

```
if(argc < 2)
   err_quit("Missing host argument - use '%s  host [echo]'\n", argv[0]);
if(argc > 2) {
   printf("Echoing messages to all streams\n");
   echo_to_all = 1;
}
```

Create an SCTP socket

```
sock_fd = Socket(AF_INET, SOCK_SEQPACKET,
   IPPROTO_SCTP);
```

- Set up server address

Use given server address and well known port number to construct server address structure using Inet_pton().

```
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(SERV_PORT);
Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
```

SCTP One-to-Many-Style Streaming Echo Client: main Function

- Set up for notifications of interest

Explicitly setting notification subscription for one-to-many socket.

```
bzero(&evnts, sizeof(evnts));
evnts.sctp_data_io_event = 1;
Setsockopt(sock_fd,IPPROTO_SCTP, SCTP_EVENTS, &evnts,
sizeof(evnts));
```

- Call echo processing function

If flag is not set then sctpstrcli() is called.

Else sctpstr_cli_echoall() is called.

```
if(echo_to_all == 0)
    sctpstr_cli(stdin,sock_fd,(SA *)&servaddr,sizeof(servaddr));
else
    sctpstr_cli_echoall(stdin,sock_fd,(SA*)&servaddr,sizeof(servaddr));
```

These functions are shown later.

- Finish up

When all is done, client closes socket and returns zero.

```
Close(sock_fd);
return(0);
```

Thank
You