Лекция 9

Протокол SCTP

Протокол передачи с управлением потоком (Stream Control Transmission Protocol, SCTP) − это надежный транспортный протокол, который обеспечивает стабильную, упорядоченную (с сохранением порядка следования пакетов) передачу данных между двумя конечными точками (подобно TCP) [5-7]. Кроме того, протокол обеспечивает сохранение границ отдельных сообщений (подобно UDP). Однако в отличие от протоколов TCP и UDP протокол SCTP имеет дополнительные возможности, такие как поддержка множественной адресации (multihoming) и многопоточности (multi-streaming).

Протокол SCTP изначально разрабатывался для передачи телефонных сигналов и был определен в (уже устаревшем) RFC2960. Текущая спецификация RFC4960 отменяет RFC2960. Подробности отличий приведены в RFC4460.

Итак, согласно спецификации RFC 4960 (http://www.protocols.ru/WP/rfc4960/), SCTP представляет собой протокол транспортного уровня с гарантированной доставкой, работающий в пакетных сетях без явной организации соединений таких, как IP. Протокол обеспечивает пользователям следующие типы сервиса:

- передача пользовательских данных с корректировкой ошибок, подтверждением доставки и отсутствием дубликатов;
- фрагментирование данных в соответствии с определенным для пути значением MTU;
- упорядоченная доставка пользовательских сообщений внутри множества потоков с возможностью управления порядком доставки отдельных пользовательских сообщений;
- возможность группировки пользовательских сообщений в один пакет SCTP;
- устойчивость к отказам на сетевом уровне за счет поддержки многодомных хостов на обеих сторонах соединения.

Протокол SCTP включает механизмы предотвращения насыщения и устойчивости к атакам с использованием лавины пакетов (flooding) или маскированием адресов (masquerade).

3.3.1. Заголовок пакета SCTP

SCTP пакеты имеют более простую структуру, чем пакеты TCP (см. рис. 3.5).

Биты	**0–7**	**8–15**	**16–23**	**24–31**
+0	Порт источника		Порт назначения	
32	Тэг проверки			
64	Контрольная сумма			
96	Тип 1 блока	Флаги 1 блока	Длина 1 блока	
128	Данные 1 блока			
…	…			
…	Тип N блока	Флаги N блока	Длина N блока	
…	Данные N блока			

Рис.3.5 Формат заголовка SCTP-пакета

Каждый пакет состоит из двух основных разделов [6, 7]:

1. Общий заголовок занимает первые 12 байт. Поля *порт источника*, *порт назначения* и *контрольная сумма* описывались при рассмотрении заголовка протокола TCP и UDP. Поле *Тэг проверки* (проверочная метка) имеет длину 32 бита и предназначено для проверки отправителя пакета. Его значение определяется на этапе установления соединения. Благодаря этому механизму улучшается защищенность информации от возможного искажения.
2. Блоки данных занимают оставшуюся часть пакета. Пакет делится на блоки. Каждый блок имеет идентификатор типа (*тип блока, см. табл.1*), размером один байт. Если данное поле имеет нулевое значение, то это говорит о передаче полезной информации (payload data); в других случаях блок «несет» служебные сведения. RFC 4960 определяет список видов блоков, всего на данный момент определено 15 типов. Второе поле блока, содержит *флаги*; его использование определяется типом блока. Третье поле заполняется суммарным значением длины блока с учетом полей заголовка (максимальный размер 65535 байт). Если *длина блока* не кратно 4, то оставшееся пространство (*блок данных* выравнивается до кратности 4) заполняется нулями.

Таблица 1 Типы блоков SCTP

ID	Тип блока	ID	Тип блока
0	Пользовательские данные (DATA)	12	Зарезервировано для Explicit Congestion Notification Echo (ECNE)
1	Инициализация (INIT)	13	Зарезервировано для Congestion Window Reduced (CWR)
2	Подтверждение инициирования (INIT ACK)	14	Процедура окончания работы завершена (SHUTDOWN COMPLETE)
3	Выборочное подтверждение (SACK)	15–62	Доступны
4	Запрос Heartbeat (HEARTBEAT)	63	Резерв для определенных IETF расширений
5	Подтверждение Heartbeat (HEARTBEAT ACK)	64–126	Доступны
6	Прерывание (ABORT)	127	Резерв для определенных IETF расширений
7	Завершение работы (SHUTDOWN)	128– 90	Доступны
8	Подтверждение завершения (SHUTDOWN ACK)	191	Резерв для определенных IETF расширений
9	Ошибка при операции (ERROR)	192–254	Доступны
10	State Cookie (COOKIE ECHO)	255	Резерв для определенных IETF расширений
11	Подтверждение Cookie (COOKIE ACK)		

Для блока **Пользовательские данные (DATA)** используется формат, показанный на рисунке 2.

Рис.2 Формат блока – Пользовательские данные

Строка	0 7	8 15	16 31
0	Type =0	**Флаги**: Reserved, U, B, E	Length
1	TSN		
2	Stream Identifier S		Stream Sequence Number n
3	Payload Protocol Identifier		
4	User Data (последовательность n потока S)		

Тип этого блока *Type* =0.

Флаги. Reserved (5 бит); это поле заполняется нулями, а получатель – игнорирет его.

U – 1 бит; Флаг разупорядочивания, устанавливаемый для блоков DATA, которые могут передаваться без сохранения порядка. Для таких блоков значение поля *Stream Sequence Number* не устанавливается, следовательно, получатель должен его игнорировать. После сборки (если она требуется) неупорядоченные блоки DATA должны диспетчеризоваться получателем на вышележащий уровень без попыток восстановления порядка, если флаг *U* имеет значение 1. Если неупорядоченное пользовательское сообщение фрагментируется, для каждого фрагмента должен устанавливаться флаг *U* = 1.

B – 1 бит. Флаг первого фрагмента пользовательского сообщения.

E – 1 бит. Флаг последнего фрагмента пользовательского сообщения.

В нефрагментированных пользовательских сообщениях нужно устанавливать (1) оба флага *B* и *E* (см. табл. 2). Нулевые значения обоих флагов устанавливаются в средних (не первом и не последнем) фрагментах пакета.

Таблица 2: Состояния флагов B и E

B	E	**Описание**
1	0	Первый фрагмент
0	0	Один из средних фрагментов
0	1	Последний фрагмент
1	1	Нефрагментированное сообщение

При фрагментировании пользовательского сообщения на множество блоков получатель использует для сборки номера *TSN*. Это означает, что фрагменты должны передаваться в строгой последовательности.

Length – 16 битов (целое число без знака). Это поле показывает размер блока DATA от начала поля типа и до конца пользовательских данных (без учета байтов заполнения). Для блока DATA с одним байтом пользовательских данных *Length* = 17 (17 байтов). Блок DATA с полем User Data размера L будет иметь поле *Length* со значением (16 + L) и L должно быть больше 0.

TSN – 32 бита (целое число без знака). Это поле содержит порядковый номер *TSN* для блока DATA. Значения номеров *TSN* могут находиться в диапазоне от 0 до (2^32 – 1). После достижения *TSN* максимального значения 4294967295 нумерация продолжается с 0.

Stream Identifier S – 16 битов (целое число без знака). Идентификатор потока *S*, к которому относится блок данных.

Stream Sequence Number n – 16 битов (целое число без знака). Это значение представляет порядковый номер *n* пользовательских данных в потоке *S*. Допустимые значения лежат в диапазоне от 0 до 65535. При фрагментировании пользовательского сообщения протоколом SCTP в каждом фрагмент должен указываться одинаковый порядковый номер в потоке.

Payload Protocol Identifier – 32 бита (целое число без знака). Это поле содержит заданный приложением (или вышележащим протоколом) идентификатор протокола. SCTP получает идентификатор от вышележащего уровня и передает его

удаленному партнеру. Значение идентификатора не используется протоколом SCTP, но может быть использовано некоторыми сетевыми объектами и приложениями на удаленной стороне для идентификации типа информации, передаваемой в блоке DATA. Это поле должно передаваться даже для фрагментированных блоков DATA (чтобы обеспечить доступность информации для агентов в сети). Отметим, что реализации SCTP не работают с этим полем, поэтому в нем не требуется использовать порядок байтов big endian. За преобразования порядка байтов в этом поле отвечает вышележащий уровень. Нулевое значение показывает что протокол вышележащего уровня не указал идентификатор протокола для этого блока.

User Data – переменный размер. Это поле переменной длины содержит пользовательскую информацию. Реализация протокола должна обеспечивать выравнивание размеров поля по 4-байтовой границе путем добавления от 1 до 3 байтов с нулевым значением. Недопустимо учитывать байты заполнения в поле размера блока. Для отправителя недопустимо использование более 3 байтов заполнения.

Подробное описание блоков SCTP представлено в RFC 4960. Здесь приведем краткое пояснение к ним.

Подтверждение инициализации (INIT ACK). Блок INIT ACK используется для подтверждения инициирования ассоциации SCTP.

Селективное подтверждение (SACK). Этот блок передается партнеру для подтверждения приема блоков DATA и информирования партнера о пропусках в порядковых номерах блоков DATA, представленных в TSN. В частности, блок содержит поле Advertised Receiver Window Credit (a_rwnd, 32 бита), которое показывает обновленное значение размера (в байтах) приемного буфера на стороне отправителя данного блока SACK.

Запрос Heartbeat (HEARTBEAT). Конечной точке следует передавать такой запрос своему партнеру для проверки его доступности через тот или иной транспортный адрес, указанный для данной ассоциации.

Подтверждение Heartbeat (HEARTBEAT ACK). Конечной точке следует передавать такой блок в ответ на запрос HEARTBEAT. Блок HEARTBEAT ACK всегда передается по тому адресу IP, который был указан в заголовке дейтаграммы, содержавшей HEARTBEAT.

Разрыв ассоциации (ABORT). Блок ABORT передается партнеру для разрыва ассоциации. Блок ABORT может содержать параметры Error Cause, информирующие партнера о причинах разрыва ассоциации.

Если конечная точка получает блок ABORT с некорректным форматом или TCB (блок контроля передачи) не найдено, такой блок должен быть отброшен без уведомления. Более того, в некоторых случаях для получившей такой блок ABORT конечной точки недопустимо передавать в ответ свой блок ABORT.

Завершение ассоциации (SHUTDOWN). Конечная точка ассоциации должна использовать этот блок для аккуратного завершения работы ассоциации. Формат блока описан ниже.

Подтверждение закрытия ассоциации (SHUTDOWN ACK). Этот блок должен использоваться для подтверждения приема блока SHUTDOWN при завершении процесса закрытия.

Закрытие ассоциации завершено (SHUTDOWN COMPLETE). Этот тип блоков должен использоваться для подтверждения приема блока SHUTDOWN ACK при завершении ассоциации

Ошибка при работе (ERROR). Конечные точки передают блоки этого типа для уведомления партнера о некоторых типах ошибок (например, некорректный идентификатор потока, нехватка ресурсов, нераспознанный тип блока и др.). Блок содержит один или несколько кодов ошибок. Прием сообщения об ошибке не обязывает партнера разрывать ассоциацию, однако такие блоки могут передаваться вместе с блоком ABORT в качестве отчета о причине разрыва.

Cookie Echo (COOKIE ECHO). Этот блок используется только в процессе создания ассоциации. Он передается инициатором ассоциации удаленному партнеру для завершения процесса инициирования ассоциации. Такой блок должен предшествовать любому блоку DATA, передаваемому через ассоциацию, но может быть сгруппирован с одним или несколькими блоками DATA в один пакет.

Подтверждение Cookie (COOKIE ACK). Этот тип блоков используется только при создании ассоциации и служит подтверждением приема блока COOKIE ECHO.

3.3.2. Множественная адресация

По сравнению с протоколом TCP поддержка протоколом SCTP *множественной адресации* обеспечивает приложениям повышенную готовность. Хост, подключенный к нескольким сетевым интерфейсам и потому имеющий несколько IP адресов, называется multi-homed хост. В протоколе TCP *соединением* называется канал между двумя конечными точками (в данном случае сокет между интерфейсами двух хостов). Протокол SCTP вводит понятие *ассоциации*, которая устанавливается между двумя хостами и в рамках которой возможна организация взаимодействия между несколькими интерфейсами каждого хоста.

На рисунке 3.6 показано отличие соединения протокола TCP и ассоциации протокола SCTP.

В верхней части рисунка показано соединение протокола TCP. Каждый хост имеет один сетевой интерфейс, соединение устанавливается между одним интерфейсом на клиенте и одним интерфейсом на сервере. Установленное соединение привязано к конкретному интерфейсу.

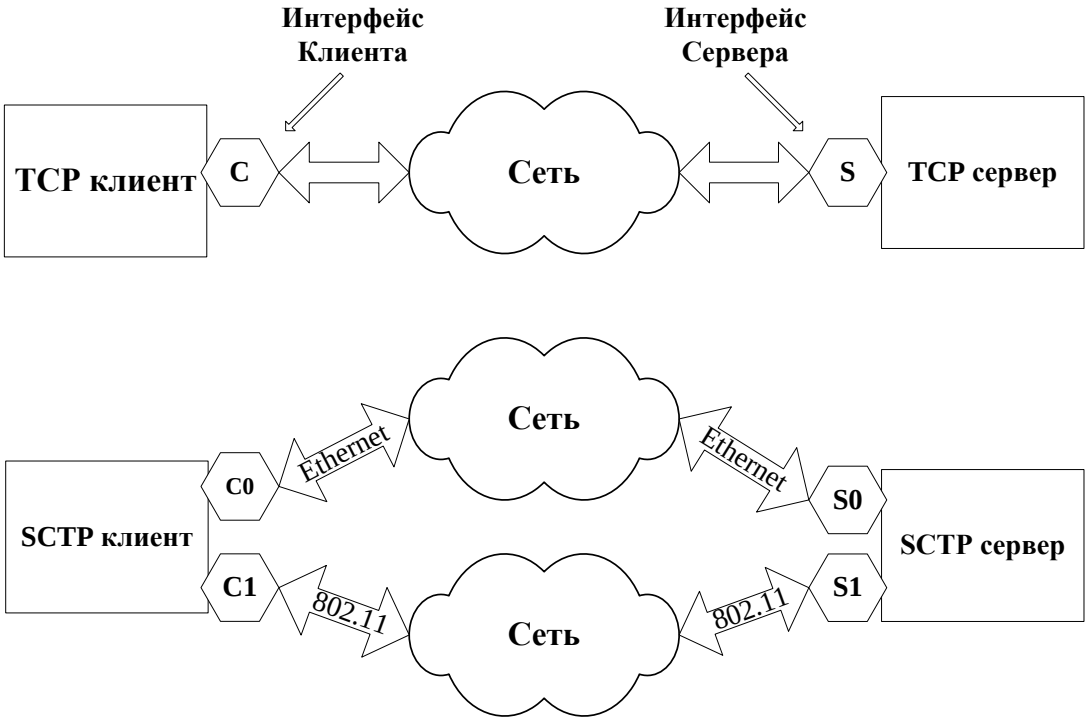


Рис.3.6 Отличие между соединением протокола TCP и ассоциацией протокола SCTP

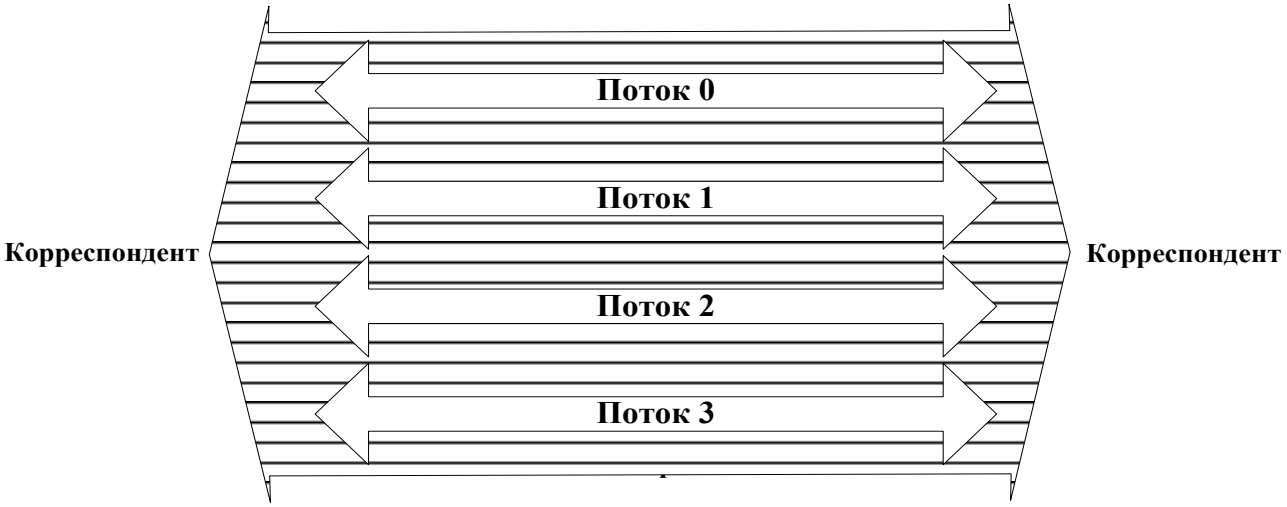
В нижней части рисунка представлена архитектура, в которой каждый хост имеет два сетевых интерфейса. Обеспечивается два маршрута по независимым сетям: один от интерфейса C0 к интерфейсу S0, другой — от интерфейса C1 к интерфейсу S1. В протоколе SCTP два этих маршрута объединяются в ассоциацию.

Протокол SCTP отслеживает состояние маршрутов в ассоциации с помощью встроенного механизма контрольных сообщений; при нарушении маршрута передача данных продолжается по альтернативному маршруту. При этом приложению даже не обязательно знать о фактах нарушения и восстановления маршрута.

Переключение на резервный канал также может обеспечивать непрерывность связи для сетевых приложений. Для примера рассмотрим ноутбук, который имеет беспроводный интерфейс 802.11 и интерфейс Ethernet. Пока ноутбук подключен к док-станции, предпочтительнее использовать более скоростной интерфейс Ethernet (в протоколе SCTP используется термин *основной адрес*); при нарушении этого соединения (в случае отключения от док-станции) будет использоваться соединение по беспроводному интерфейсу. При повторном подключении к док-станции будет обнаружено соединение по интерфейсу Ethernet, через который будет продолжен обмен данными. Таким образом, протокол SCTP реализует эффективный механизм, обеспечивающий высокую готовность и повышенную надежность.

3.3.3. Многопотоковая передача данных

В некоторой степени ассоциация протокола SCTP похожа на соединение протокола TCP [6, 7]. Отличие состоит в том, что протокол SCTP поддерживает несколько потоков в рамках одной ассоциации. Все потоки являются независимыми, но принадлежат одной ассоциации (см. рис. 3.7).



Р

Рис. 3.7 Взаимосвязь между ассоциацией протокола SCTP и потоками

Каждому потоку ассоциации присваивается номер, который включается в передающиеся пакеты SCTP. Важность многопотоковой передачи обусловлена тем, что блокировка какого-либо потока (например, из-за ожидания повторной передачи при потере пакета) не оказывает влияния на другие потоки в ассоциации. В общем случае данная проблема получила название *head-of-line blocking (блокировка очереди*). Протокол TCP уязвим для подобных блокировок. Блокирование возникает при потере сегмента TCP при передачи и приходе следующего за ним сегмента, который удерживается до тех пор, пока утраченный сегмент не будет передан повторно и получен адресатом.

Поток SCTP - это вовсе не поток байтов, как в TCP. Это последовательность сообщений упорядоченных в пределах ассоциации. Потоки с собственным порядком используются для того, чтобы обойти блокирование очереди.

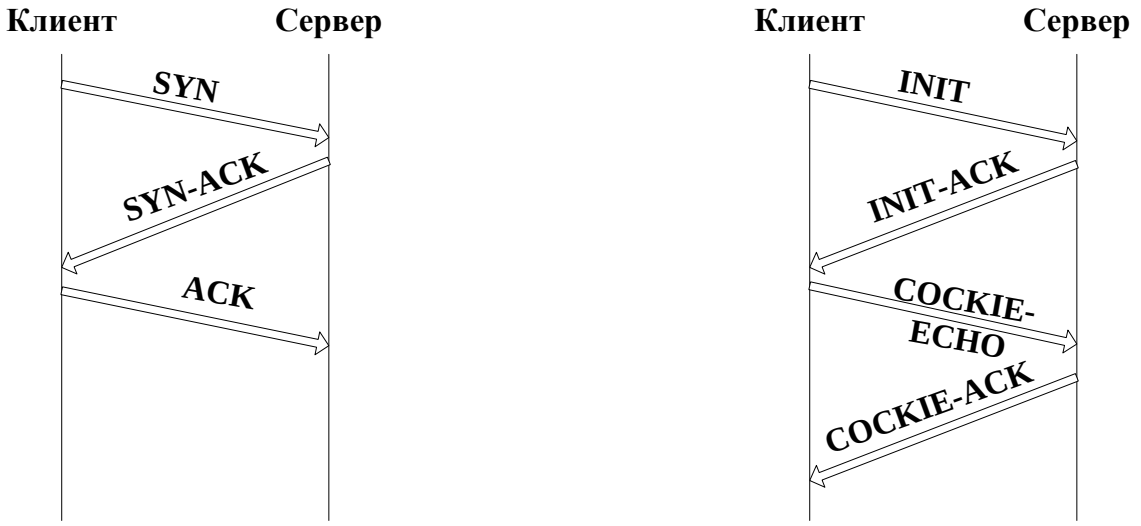
Каким образом множество потоков обеспечивают лучшую оперативность при передаче данных? Например, в протоколе HTTP данные и служебная информация передаются по одному и тому же сокету. Web-клиент запрашивает файл, и сервер посылает файл назад к клиенту по тому же самому соединению. Многопотоковый HTTP-сервер сможет обеспечить более быструю передачу, так как множество запросов может обслуживаться по независимым потокам одной ассоциации. Такая возможность позволяет распараллелить ответы сервера. Это если и не повысит скорость отображения страницы, то позволит обеспечить ее лучшее восприятие благодаря одновременной загрузке кода HTML и графических изображений.

Многопотоковая передача − это важнейшая особенность протокола SCTP, особенно в том, что касается одновременной передачи данных и служебной информации в рамках протокола. В протоколе TCP данные и служебная информация передаются по одному соединению. Это может стать причиной проблем, так как служебные пакеты из-за передачи данных будут передаваться с задержкой. Если служебные пакеты и пакеты данных передаются по независимым потокам, то служебная информация будет обрабатываться своевременно, что, в свою очередь, приведет к лучшему использованию доступных ресурсов.

3.3.4. Безопасность устанавливаемого подключения

Создание нового подключения в протоколах TCP и SCTP происходит при помощи механизма подтверждения (квитирования) пакетов [6, 7]. В протоколе TCP

данная процедура получила название *трехэтапное квитирование (three-way handshake*). Клиент посылает пакет SYN (сокр. *Synchronize*). Сервер отвечает пакетом SYN-ACK (*Synchronize-Acknowledge*). Клиент подтверждает прием пакета SYN-ACK пакетом ACK. На этом процедура установления соединения (показана на рисунке 3.8) завершается.



Процедура установления соединения (треэтапное квитирование) в протоколе TCP

Процедура установления соединения (четерехэтапное квитирование) в протоколе SCTP

Рис.3.8 Обмен пакетами при установлении соединения по протоколам TCP и SCTP

Протокол TCP имеет потенциальную уязвимость, обусловленную тем, что нарушитель, установив фальшивый IP-адрес отправителя, может послать серверу множество пакетов SYN. При получении пакета SYN сервер выделяет часть своих ресурсов для установления нового соединения. Обработка множества пакетов SYN рано или поздно займет все ресурсы с невозможностью обработки новых запросов сервером. Такая атака получила название «отказ в обслуживании» (*Denial of Service* (DoS)).

Протокол SCTP защищен от подобных атак с помощью механизма четырехэтапного квитирования (four-way handshake) и вводом маркера (cookie). По протоколу SCTP клиент начинает процедуру установления соединения посылкой пакета INIT. В ответ сервер посылает пакет INIT-ACK, который содержит маркер (уникальный ключ, идентифицирующий новое соединение). Затем клиент отвечает посылкой пакета COOKIE-ECHO, в котором содержится маркер, посланный сервером. Только после этого сервер выделяет свои ресурсы новому подключению и подтверждает это отправлением клиенту пакета COOKIE-ACK.

Для решения проблемы задержки пересылки данных при выполнении процедуры четырехэтапного квитирования в протоколе SCTP допускается включение данных в пакеты COOKIE-ECHO и COOKIE-ACK.

3.3.5. Формирование кадров сообщения

При формировании кадров сообщения обеспечивается сохранение границ сообщения в том виде, в котором оно передается сокету; это означает, что если клиент посылает серверу 100 байт, за которыми следуют 50 байт, то сервер воспринимает 100 байт и 50 байт за две операции чтения [6, 7]. Точно так же

функционирует протокол UDP, это является особенностью протоколов, ориентированных на работу с сообщениями.

В противоположность им протокол TCP обрабатывает неструктурированный поток байт. Если не использовать процедуру формирования кадров сообщения, то узел сети может получать данные по размеру больше или меньше отправленных. Такой режим функционирования требует, чтобы для протоколов, ориентированных на работу с сообщениями и функционирующих поверх протокола TCP, на прикладном уровне был предоставлен специальный буфер данных и выполнялась процедура формирования кадров сообщений (что потенциально является сложной задачей).

Протокол SCTP обеспечивает формирование кадров при передаче данных. Когда узел выполняет запись в сокет, его корреспондент с гарантией получает блок данных того же размера (см. рис.3.9).

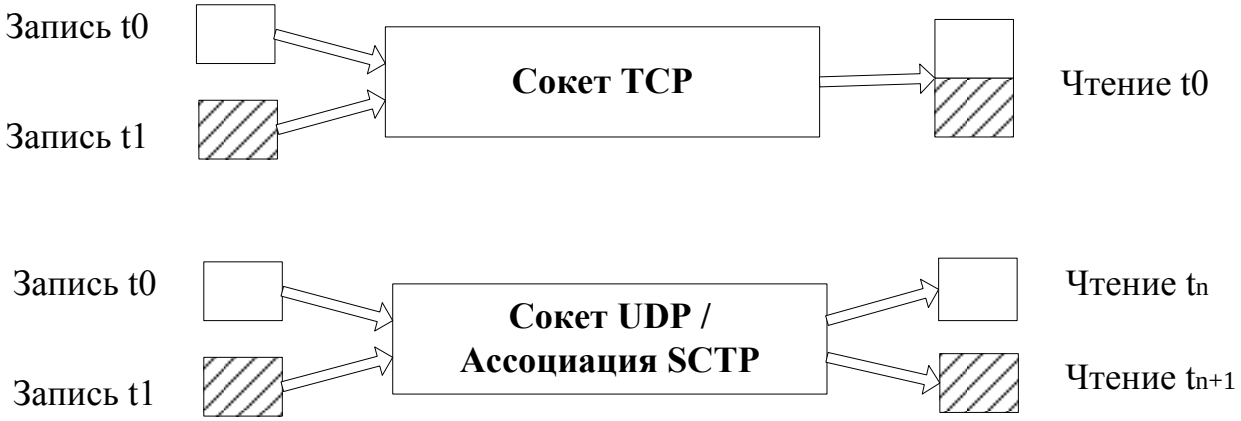


Рис.3.9 Формирование кадров сообщений по протоколу UDP/SCTP
и по протоколу, обрабатывающему неструктурированный поток байт

При передаче потоковых данных (аудио- и видеоданных) процедура формирования кадров необязательна.

3.3.6. Настраиваемая неупорядоченная передача данных

Сообщения по протоколу SCTP передаются с высокой степенью надежности, но необязательно в нужном порядке [6, 7]. Протокол TCP гарантирует, что данные будут получены именно в том порядке, в котором они отправлялись (это хорошо, учитывая, что TCP – это потоковый протокол). Протокол UDP не гарантирует упорядоченную доставку. При необходимости вы можете настроить потоки в протоколе SCTP так, чтобы они принимали неупорядоченные сообщения.

Эта особенность может быть востребована в протоколах, ориентированных на работу с сообщениями, так как запросы в них независимы и последовательность поступления данных не очень важна. Кроме того, вы можете настроить использование неупорядоченной передачи по номеру потока в ассоциации, т.е. принимая сообщения по потокам.

3.3.7. Поэтапное завершение передачи данных

В отличие от протокола UDP, функционирование которого не предполагает установления соединения, протоколы TCP и SCTP являются протоколами с установлением соединения [6, 7]. Оба эти протокола требуют выполнения

процедуры установления и разрыва соединения между корреспондентами. Рассмотрим отличия между процедурой закрытия сокетов протокола SCTP и процедурой частичного закрытия (*half-close*) протокола TCP.
На рисунке 3.10 показана последовательность разрыва соединения в протоколах TCP и SCTP.

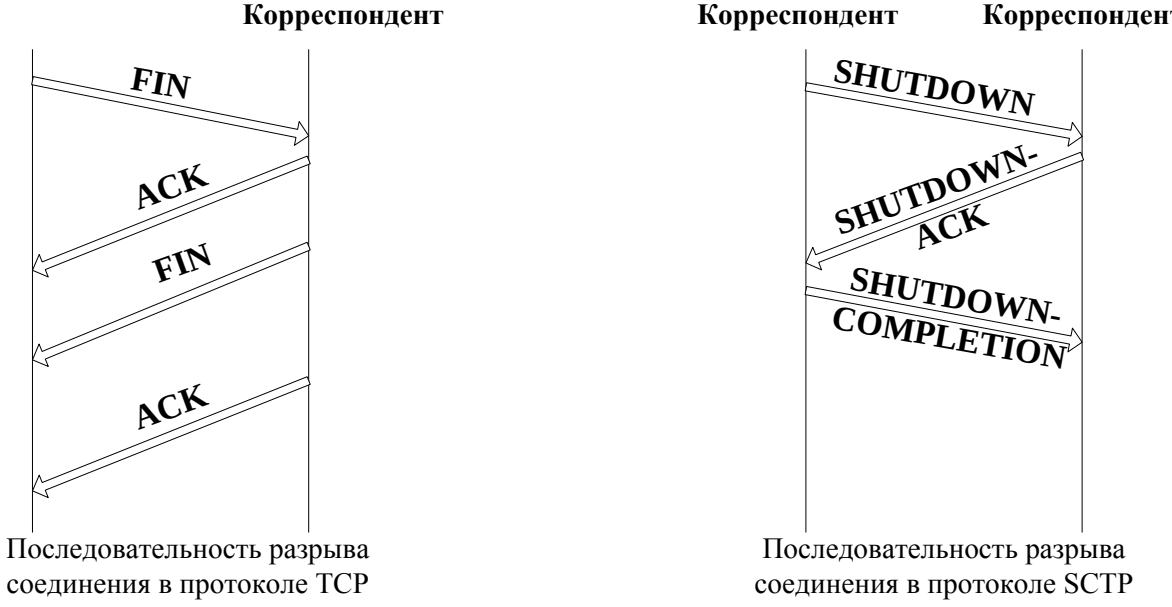


Рис.3.10 Последовательность разрыва соединения
по протоколам TCP и SCTP

В протоколе TCP возможна ситуация, когда узел закрывает у себя сокет (выполняя посылку пакета `FIN`), но продолжает принимать данные. Пакет `FIN` указывает корреспонденту на отсутствие данных для передачи, однако до тех пор, пока корреспондент не закроет свой сокет, он может продолжать передавать данные. Состояние частичного закрытия используется приложениями крайне редко, поэтому разработчики протокола SCTP посчитали нужным заменить его последовательностью сообщений для разрыва существующей ассоциации. Когда узел закрывает свой сокет (посылает сообщение `SHUTDOWN`), оба корреспондента должны прекратить передачу данных, при этом разрешается лишь обмен пакетами, подтверждающими прием ранее отправленных данных.

5.1.1. Функция sctp_bindx

Функция sctp_bindx предоставляет возможность связывать сокет SCTP с заданными адресами [5].

#Include <netinet/sctp.h>

*int sctp_bindx(int sid, const struct sockaddr *addrs, int addrcnt, int flags);*

Аргумент sid представляет собой дескриптор сокета, возвращаемый функцией *socket*. Второй аргумент — указатель на упакованный список адресов. Каждая структура адреса сокета помещается в буфер непосредственно после

предшествующей структуры, без всяких дополняющих нулей (например, см. рисунок 5.3).

Количество адресов, передаваемых *sctp_bindx*, указывается в параметре *addrcnt*. Параметр *flags* сообщает функции *sctp_bindx* о необходимости выполнения действий:

SCTP_BINDX_ADD_ADDR – добавляет адреса к уже определенным для сокета;

SCTP_BINDX_REM_ADDR – удаляет адреса из списка адресов сокета.

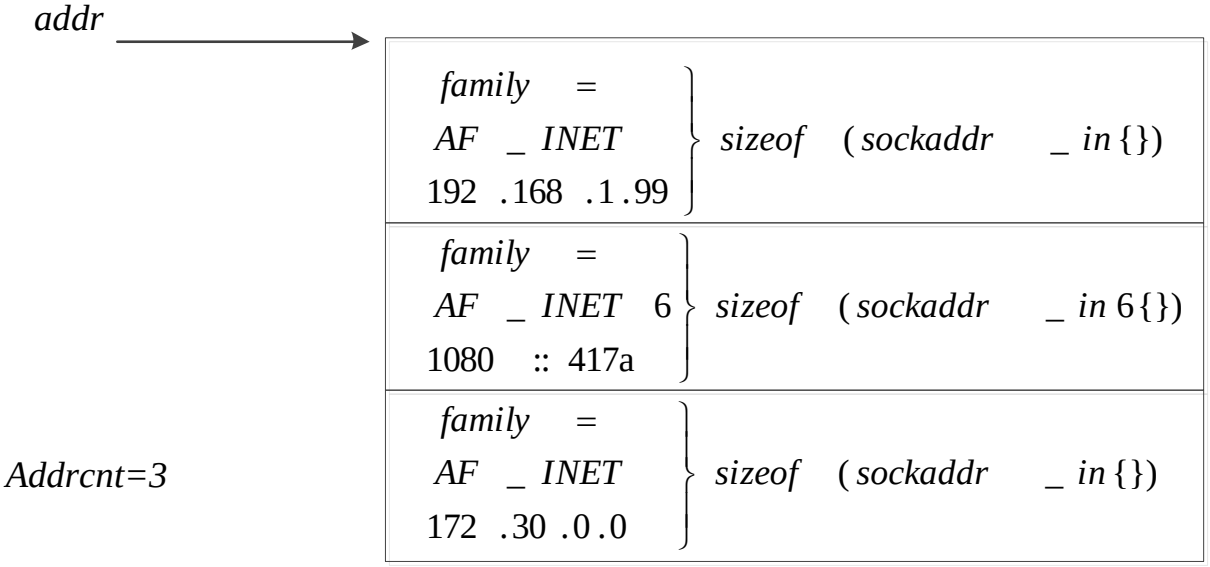


Рис. 5.3 Пример упакованного списка адресов

Функцию *sctp_bindx* можно вызывать независимо от того, привязан ли сокет к каким-нибудь адресам. Для несвязанного сокета вызов *sctp_bindx* приведет к привязке указанного набора адресов. При работе с уже связанным сокетом указание флага *SCTP_BIND_ADD_ADDR* позволяет добавить адреса к данному дескриптору. Флаг *SCTP_BINDX_REM_ADDR* предназначен для удаления адресов из списка связанных с данным дескриптором. Если *sctp_bindx* вызывается для прослушиваемого сокета, новая конфигурация будет использоваться только для новых ассоциаций; вызов никак не затронет уже установленные ассоциации. Флаги *sctp_bindx* взаимно исключают друг друга: если указать оба, функция вернет ошибку *EINVAL*. Номер порта во всех структурах адреса сокета должен быть одним и тем же. Он должен совпадать с тем номером порта, который был связан с данным сокетом ранее. В противном случае *sctp_bindx* тоже вернет ошибку *EINVAL*.

5.1.2. Функция sctp_connectx

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

int sctp_connectx(int sid, const struct sockaddr *addrs, int addrcnt);
```

Функция *sctp_connectx* используется для соединения с многоинтерфейсным узлом. При ее вызове должны быть указаны адреса собеседника в параметре *addrs* (количество адресов определяется параметром *addrcnt*). Формат структуры *addrs* представлен на рисунке 5.3. Стек SCTP устанавливает ассоциацию, используя один

или несколько адресов из переданного списка. Все адреса *addrs* считаются действующими и подтвержденными.

Возвращает: 0 в случае успешного завершения. -1 в случае ошибки.

5.1.3. Функция sctp_sendmsg

Приложение может управлять параметрами SCTP, используя функцию *sendmsg* со вспомогательными данными [5]. Однако из-за неудобств, связанных с применением вспомогательных данных, многие реализации SCTP предоставляют дополнительный библиотечный вызов (который на самом деле может быть и системным вызовом), упрощающий обращение к расширенным функциям SCTP. Вызов функции должен иметь следующий формат:

*ssize_t sctp_sendmsg (int sid, const void * buff, size_t bufflen, const struct sockaddr *to, socklen_t tolen, uint32_t ppid, uint32_t flags, uint16_t stream, uint32_t timetolive, uint32_t context);*

Использование *sctp_sendmsg* значительно упрощает отправку параметров, но требует указания большего количества аргументов. В поле *sid* помещается дескриптор сокета, возвращаемый системным вызовом *socket*. Аргумент *buff* указывает на буфер размера *bufflen*, содержимое которого должно быть передано собеседнику. В поле *tolen* помещается длина адреса, передаваемого через аргумент *to*. В поле *ppid* помещается идентификатор протокола, который будет передан вместе с порцией данных. Поле *flags* передается стеку SCTP. Например, некоторыми разрешенными значениями флагов являются [5], *MSG_ABORT* - вызывает аварийное завершение ассоциации, *MSG_ADDR_OVER* – заставляет SCTP использовать указанный адрес вместо адреса по умолчанию и др.

Номер потока SCTP указывается вызывающим приложением в аргументе *stream*. Процесс может указать время жизни сообщения в миллисекундах в поле *lifetime*. Значение 0 соответствует бесконечному времени жизни. Пользовательский контекст, при наличии такового, может быть указан в поле *context*. Пользовательский контекст связывает неудачную передачу сообщения (о которой получено уведомление) с локальным контекстом, имеющим отношение к приложению. Например, чтобы отправить сообщение в поток 1 с флагом отправки *MSG_PR_SCTP_TTL*, временем жизни равным 2000 мс, идентификатором протокола 24 и контекстом 52, процесс должен сделать следующий вызов:

ret - sctp_sendmsg(sid, data, datasz, &dest, sizeof(dest), 24, MSG__PR_SCTP_TTL, 1, 2000, 52);

Этот подход значительно проще выделения памяти под необходимые вспомогательные данные и настройки структур, входящих в *msghdr*. Обратите внимание, что если функция *sctp_sendmsg* реализована через вызов *sendmsg*, то поле *flags* в последнем устанавливается равным 0.

Возвращает количество записанных байтов в случае успешного завершения, -1 в случае ошибки

5.1.4. Функция sctp_recvmsg

Функция *sctp_recvmsg* принимает сообщение через гнездо, указанное в аргументе *sid* и подобно *sctp_sendmsg*, предоставляет удобный интерфейс к

расширенным возможностям SCTP. С ее помощью пользователь может получить не только адрес собеседника, но и поле *msg_flags*, которое обычно заполняется при вызове *recvmsg* (например, *MSG_NOTIFICATION*, *MSG_EOR* и так далее). Кроме того, функция дает возможность получить структуру *sctp_sndrcvinfo*, которая сопровождает сообщение, считанное в буфер. Если приложение хочет получать информацию, содержащуюся в структуре *sctp_sndrcvinfo*, оно должно быть подписано на событие *sctp_data_io_event* с параметром сокета *SCTP_EVENTS* (по умолчанию эта подписка включена).

*ssize_t sctp_recvmsg(int sid, void *buff, size_t bufflen, struct sockaddr *from, socklen_t *fromlen, struct sctp_sndrevinfo *sinfo, int *msg_flags);*

По возвращении из этого вызова аргумент *buff* оказывается заполненным не более, чем *bufflen* байтами данных. Адрес отправителя сообщения помещается в аргумент *from*, а размер адреса — в аргумент *fromlen*. Флаги сообщения будут помещены в аргумент *msg_flags*. Если уведомление *sctp_data_io_event* включено (а по умолчанию это так и есть), структура *sctp_sndrcvinfo* заполняется подробными сведениями о сообщении. Если функция *sctp_recvmsg* реализована через вызов *recvmsg*, то поле *flags* в последнем устанавливается равным нулю.

Количество адресов, передаваемых *sctp_bindx*, указывается в параметре *addrcnt*. Параметр *flags* сообщает функции *sctp_bindx* о необходимости выполнения действий, перечисленных в п.5.1.11.

Возвращает количество считанных байтов в случае успешного завершения, -1 в случае ошибки

Функция getsockname

#include <sys/socket.h>

*int getsockname(int sid, struct sockaddr *name, socklen_t *namelen);*

Функция *getsockname* возвращает данные указанного сокета в параметре *name* (*struct sockaddr*). В параметре *namelen* указывается, сколько места (размер в байтах) выделено под *name*.

В случае успеха возвращается ноль. При ошибке возвращается -1, а значение *errno* устанавливается должным образом.

Замечание. Третий аргумент функции *getsockname* в действительности имеет тип int *. Определенное недопонимание привело к тому, что в стандарте POSIX появился тип socklen_t.

Функция getpeername

#include <sys/socket.h>

*int getpeername(int sid, struct sockaddr *name, socklen_t *namelen);*

Функция *getpeername* возвращает данные абонента (*struct sockaddr*), подключившейся к сокету *sid*. Параметр *namelen* – размер, который занимает *name*. По возвращении он содержит размер памяти, занимаемый данными (адрем собеседника) об абоненте в байтах. Эти данные не считываются, если буфер окажется слишком мал.

При удачном завершении возвращается ноль. При ошибке возвращается -1 , а переменной *errno* присваивается соответствующее значение.

Замечание. Третий аргумент функции *getpeername* в действительности является `int *'. В результате недоразумения в POSIX появилось socklen_t.

Функция sctp_getpaddrs

Функция *getpeername* не предназначена для использования протоколом, рассчитанным на работу с многоинтерфейсными узлами. Для сокетов SCTP она способна вернуть лишь основной адрес собеседника. Если нужны все адреса, следует вызывать функцию *sctp_getpaddrs*.

#include <netinet/sctp.h>

*int sctp_getpaddrs(int sid, sctp_assoc_t id, struct sockaddr *addrs);*

Возвращает: 0 в случае успешного завершения, -1 в случае ошибки.

Аргумент *sid* представляет собой дескриптор сокета, возвращаемый функцией socket. Второй аргумент задает идентификатор ассоциации для сокетов типа «один-ко-многим». Для сокетов типа «один-к-одному» этот аргумент игнорируется, *addrs* — адрес указателя, который функция *sctp_getpaddrs* заполнит упакованным списком адресов, выделив под него локальный буфер (см. рис. 1и листинг 1). Для освобождения буфера, созданного *sctp_getpaddrs*, следует использовать вызов *sctp_freepaddrs*.

Функция sctp_freepaddrs

Функция *sctp_freepaddrs* освобождает ресурсы, выделенные вызовом *sctp_ getpaddrs*.

#inc1ude <netinet/sctp.h>

*void sctp_freepaddrs(struct sockaddr *addrs);*

Здесь аргумент *addrs* – указатель на масив, возвращаемый *sctp_getpaddrs*.

Функция sctp_getladdrs

Функция *sctp_getladdrs* может использоваться для получения списка локальных адресов, относящихся к определенной ассоциации. Эта функция бывает необходима в тех случаях, когда приложению требуется узнать, какие именно локальные адреса оно использует (набор адресов, может быть произвольным подмножеством всех адресов системы).

#include <netinet/sctp.h>

*int sctp_getladdrs(int sid, sctp_assoc_t id, struct sockaddr *addrs);*

Возвращает: количество локальных адресов, помещенных в addrs, или -1 в случае ошибки.

Здесь *sid* — дескриптор сокета, возвращаемый функцией socket. Аргумент *id* — идентификатор ассоциации для сокетов типа «один-ко-многим». Поле *id* игнорируется для сокетов типа «один-к-одному». Параметр представляет собой адрес указателя на буфер, выделяемый и заполняемый функцией *sctp_getladdrs*. В этот буфер помещается упакованный список адресов. Структура списка представлена на рис.1 и в листинге 1. Для освобождения буфера процесс должен вызвать функцию sctp_freeladdrs.

Функция sctp_freeladdrs

Функция *sctp_freeladdrs* освобождает ресурсы, выделенные при вызове *sctp_getladdrs*.

#include <netinet/sctp.h>

*void sctp_freeladdrs(struct sockaddr *addrs);*

Здесь *addrs* указывает на список адресов, возвращаемый *sctp_getl addrs*.

Проверка типа уведомления

9-13 Функция преобразует буфер приема к типу универсального указателя на уведомления, чтобы определить тип полученного уведомления. Из всех уведомлений нас интересуют только уведомления об изменении ассоциации, а из них — уведомления о создании или перезапуске ассоциации (*SCTP_COMM_UP* и *SCTP_RESTART*). Все прочие уведомления нас не интересуют.

Получение и вывод адресов собеседника

14-17 Функция *sctp_getpaddrs* возвращает нам список удаленных адресов, которые мы выводим при помощи функции *sctp_print_addresses*, представленной в листинге

1. После работы с ней мы освобождаем ресурсы, выделенные *sctp_getpaddrs*, вызывая функцию *sctp_freeladdrs*.

Получение и вывод локальных адресов

18-21 Функция *sctp_getladdrs* возвращает нам список локальных адресов, которые мы выводим на экран вместе с их общим количеством. После завершения работы с адресами мы освобождаем память вызовом *sctp_freeladdrs*.

Последняя из новых функций называется *sctp_print_addresses*. Она выводит на экран адреса из списка, возвращаемого функциями *sctp_getpaddrs* и *sctp_getladdrs*. Текст функции представлен в листинге 1.

Листинг 1. Вывод списка адресов

```
//sctp/sctp_print_addrs c
1 #include "unp.h"
2 void
3 sctp_print_addresses(struct sockaddr_storage *addrs, int
num)
4 {
5      struct sockaddr_storage *ss;
6      int i.salen;

7       ss = addrs;
8      for(i=0; i<num; i++){
9          printf("%s\n", Sock_ntop()SA *)ss.salen));
10   #ifdef HAVE_SOCKADDR_SA_LEN
11           salen = ss->ss len;

12  #else
13        swilch(ss->ss_family) {
14        case AF_INET;
15        salen = sizeof(struct sockaddr_in);
16        break:

17  #ifdef IPV6
18       case AF_INET6
19        salen = sizeof(struct sockaddr_in6);
20        break,
21   #endif
22         default:
23        err_quit("sctp_print_addresses: unknown AF");
24        break;
25   }
26   #endif
27     ss = (struct sockaddr_storage *)((char *)ss + salen);
28   }
```

```
29      }
```

Пример реализации многопотоковой передачи по протоколу SCTP

Рассмотрим пример реализации серверного и клиентского приложений [7]. Исходный текст программы демонстрирует возможности протокола SCTP по многопотоковой передаче данных.

В этом примере сервер реализует одну из форм протокола точного времени, сообщая текущее время подключенному клиенту. Однако для демонстрации возможностей протокола SCTP, передается местное время по потоку 0 и время по Гринвичу (GMT) в потоке 1. Этот простой пример демонстрирует интерфейс API потокового взаимодействия.

На рисунке 5.4 приведен весь процесс взаимодействия: показан не только поток данных приложения по API сокетов, но и взаимосвязи между клиентом и сервером.

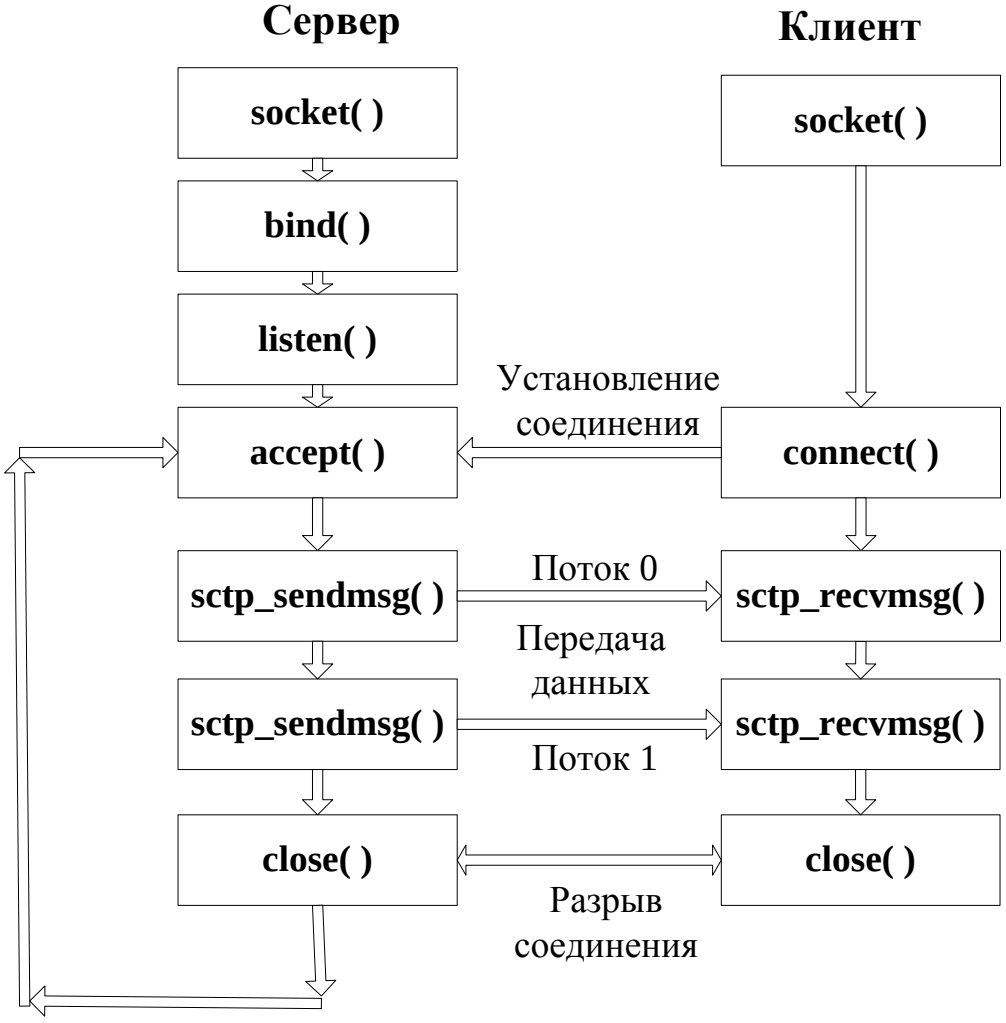


Рис.5.4 Функции сокетов, используемые при многопотоковой реализации сервера и клиента точного времени

Сервер точного времени

Исходный код приложения многопоточного сервера точного времени показан в листинге 3.

Листинг3. Сервер точного времени для протокола SCTP, использующий многопотоковую передачу

```
int main()
{
  int lSock, cSock, ret;
struct sockaddr_in servaddr;
  char buffer[MAX_BUFFER+1];
time_tcurrentTime;

  /* Создание сокета SCTP в стиле TCP */
lSock = socket( AF_INET, SOCK_STREAM, IPPROTO_SCTP );

bzero( (void *)&servaddr, sizeof(servaddr) );
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = htonl( INADDR_ANY );
  servaddr.sin_port = htons(MY_PORT_NUM);

ret = bind( lSock, (struct sockaddr *)&servaddr,
    sizeof(servaddr) );

listen( lSock, 5 );

  while( 1 ) {

cSock = accept( lSock, (struct sockaddr *)NULL, (int *)NULL
    );

    /* Выясняется текущее время */
currentTime = time(NULL);

    /* Посылается текущее время по потоку 0 (поток для
    локального времени) */
snprintf( buffer, MAX_BUFFER, "%s\n", ctime(&currentTime) );

   ret = sctp_sendmsg(cSock, (void *)buffer,
    (size_t)strlen(buffer), NULL, 0, 0, 0, LOCALTIME_STREAM,
    0, 0 );

    /* Посылается GMT по потоку 1 (поток для GMT) */
snprintf( buffer, MAX_BUFFER, "%s\n",
              asctime( gmtime( &currentTime ) ) );

    ret = sctp_sendmsg( cSock, (void *)buffer,
    (size_t)strlen(buffer), NULL, 0, 0, 0,
     GMT_STREAM, 0, 0 );

close(cSock );
```

```
  }

  return 0;
}
```

Обратите внимание на то, что протокол SCTP использует многие из тех же сокетов API, что и протоколы TCP и UDP. Для создания сокета SCTP прямого соединения используется `IPPROTO_SCTP`.

Серверная программа ожидает в цикле нового подключения клиента. При возвращении из функции `accept` создается новое клиентское подключение, определяемое сокетом `cSock`. С помощью функции `time` выясняется текущее время и переводится в строку функцией `snprintf`. С помощью функции `sctp_sendmsg` (нестандартный вызов сокета) строка посылается клиенту по заданному потоку (`LOCALTIME_STREAM`). После того как строка с локальным временем послана, текущее время переводится в формат GMT и посылается по потоку `GMT_STREAM` [7].

Клиент протокола точного времени

Реализация многопоточного клиента приводится в листинге 4.

Листинг 4. Реализация клиента точного времени для протокола SCTP, использующего многопотоковую передачу

```
int main()
{
  int cSock, in, i, flags;
  struct sockaddr_in servaddr;
  struct sctp_sndrcvinfo sndrcvinfo;
  struct sctp_event_subscribe events;
  char buffer[MAX_BUFFER+1];

  /* Созданиесокета SCTP в стиле TCP */
  cSock = socket(AF_INET, SOCK_STREAM, IPPROTO_SCTP );

bzero((void *)&servaddr, sizeof(servaddr) );
  servaddr.sin_family = AF_INET;
  servaddr.sin_port = htons(MY_PORT_NUM);
  servaddr.sin_addr.s_addr = inet_addr( "127.0.0.1" );

connect(cSock, (struct sockaddr *)&servaddr,
    sizeof(servaddr) );

/* Ожидается получение данных SCTP Snd/Rcv с помощью функции
    sctp_recvmsg */
memset( (void *)&events, 0, sizeof(events) );
```

```
  events.sctp_data_io_event = 1;
setsockopt(cSock, SOL_SCTP, SCTP_EVENTS,
               (const void *)&events, sizeof(events) );

for (i = 0 ; i< 2 ; i++) {

in = sctp_recvmsg(cSock, (void *)buffer, sizeof(buffer),
    (struct sockaddr *)NULL, 0, &sndrcvinfo, &flags );

/* Завершающий символ строки - 0 */
buffer[in] = 0;

if(sndrcvinfo.sinfo_stream == LOCALTIME_STREAM) {
      printf("(Local) %s\n", buffer);
    } else if (sndrcvinfo.sinfo_stream == GMT_STREAM) {
      printf("(GMT  ) %s\n", buffer);
}

  }

  /* Закрытие сокета и выход */
close(cSock);

return 0;
}
```

В клиентском приложении создается сокет протокола SCTP, а затем структура `sockaddr`, содержащая информацию о конечной точке подключения.

Для того чтобы получить номер потока сообщений по протоколу SCTP, необходимо указать параметр сокета `sctp_data_io_event`.

При получении сообщения с помощью функции API `sctp_recvmsg` дополнительно принимается структура `sctp_sndrcvinfo`, содержащая номер потока. Этот номер позволяет различать сообщения потока 0 (местное время) и потока 1 (GMT).