19.05.2025

Программирование сокетов

Филиппов Михаил Витальевич

m.filippov@g.nsu.ru

89232283872

Императивное программирование, 2024-2025

Новосибирский государственный университет

*НАСТОЯЩАЯ НАУКА

Давайте познакомимся

Филиппов Михаил Витальевич

- Окончил магистратуру ФФ НГУ
- Окончил аспирантуру ИТ СО РАН
- Являюсь м.н.с. ИТ СО РАН
- 7+ лет опыт в программировании C/C++

Адженда

Локальные сокеты и IPC

45 минут

Программирование сокетов

45 минут

Адженда

Локальные сокеты и IPC	Программирование сокетов
45 минут	45 минут

Введение

Мы поговорим о методах IPC, которые предусматривают некий обмен сообщениями или сигналами между разными процессами. Передающиеся сообщения не хранятся ни в каком общем месте наподобие файла или разделяемой памяти; вместо этого процессы их генерируют и принимают.

Методы межпроцессного взаимодействия

К методам IPC обычно относят любые средства взаимодействия и передачи данных между процессами.

- разделяемая память;
- файловая система (как на диске, так и в памяти);
- POSIX-сигналы;
- POSIX-каналы;
- очереди сообщений POSIX;
- сокеты домена Unix;
- интернет-сокеты (или сетевые сокеты).

С точки зрения программирования, методики разделяемой памяти и файловой системы определенно чем-то схожи, поэтому их можно отнести к одной категории, известной как активные методы IPC. Остальные подходы выделяются на их фоне, и мы будем называть их пассивными.

Обратите внимание: все методы IPC сводятся к передаче сообщений между двумя процессами. Поскольку термин «сообщение» активно используется в следующих абзацах, его сначала стоит определить.

Каждое сообщение содержит последовательность байтов, которые собираются вместе в соответствии с четко определенным интерфейсом, протоколом или стандартом. Структура сообщения должна быть известна обоим процессам, работающим с ним, и обычно описывается в рамках коммуникационного протокола.

Методы межпроцессного взаимодействия

Различия между активными и пассивными методиками.

- Активные методики подразумевают наличие разделяемого ресурса, или носителя, который доступен в пользовательском пространстве и является внешним по отношению к обоим процессам. Роль разделяемого ресурса может играть файл, разделяемая память или даже сетевой сервис, такой как NFS-сервер. Эти носители служат основным средством хранения сообщений, создаваемых и потребляемых процессами. В пассивных методах вместо таких разделяемых ресурсов, или носителей, используется канал, через который процессы отправляют и принимают сообщения. Сами сообщения при этом нигде не хранятся.
- При активном подходе все процессы должны сами извлекать из носителя доступные сообщения. При пассивном — входящие сообщения передаются (доставляются) принимающей стороне.
- Активные методики подразумевают наличие разделяемого ресурса или носителя, конкурентный доступ к которому необходимо синхронизировать. ***Обратите внимание:*** в случае с пассивными методиками синхронизация не требуется.
- При использовании активных методик процессы могут работать независимо друг от друга. Это объясняется тем, что сообщения могут храниться в разделяемом ресурсе, и их извлечение можно отложить. Иными словами, процессам доступно выполнение в асинхронной манере. Для сравнения, в случае применения пассивных методов IPC оба процесса должны находиться в активном состоянии. Кроме того, ввиду мгновенного поступления сообщений принимающий процесс может потерять какую-то их часть, если не запущен в это время. То есть процессы выполняются в синхронной манере.

Коммуникационные протоколы

Одного лишь коммуникационного канала или носителя недостаточно. Две стороны, желающие взаимодействовать по общему каналу, должны понимать друг друга! Общий язык IPC формально он называется протоколом.

Процессы могут передавать только байты. Это называется сериализацией или маршалингом. В контексте межпроцессного взаимодействия это означает, что сообщения, передаваемые между двумя процессами, представляют собой байты, упорядоченные строго определенным образом.

С другой стороны, когда процесс получает последовательность байтов по каналу IPC, он должен уметь воссоздать из них исходный объект. Это называется десериализацией или демаршалингом.

Поговорим о том, как происходит сериализация и десериализация. Когда процесс хочет отправить объект по какому-либо имеющемуся каналу IPC, он сначала сериализует его в байтовый массив и передает результат другой стороне. Принимающий процесс десериализует поступившие байты и воссоздает отправленный объект. Любая технология межпроцессного взаимодействия (RPC, RMI и т. д.) активно задействует сериализацию и десериализацию различных объектов. Пока под сериализацией мы будем понимать совокупность этих двух операций.

Коммуникационные протоколы

Обратите внимание: сериализация применяется не только в пассивных методах IPC, которые рассматривались до сих пор. Активные методы IPC, такие как файловая система или разделяемая память, тоже нуждаются в сериализации. Дело в том, что носители в этих методах могут хранить последовательности байтов, и если процесс хочет записать некий объект, скажем, в разделяемый файл, то должен сначала сериализовать его.

Выбор коммуникационного протокола сам по себе определяет способ сериализации, так как в рамках протокола детально описывается порядок, в котором размещаются байты. Это очень важно, поскольку сериализованный объект должен быть десериализован обратно принимающей стороной. Следовательно, механизмы сериализации и десериализации должны подчиняться правилам, которые диктует протокол. Если эти механизмы несовместимы между собой, то стороны фактически теряют всякую возможность взаимодействовать, просто ввиду того, что получатель не может воссоздать переданный объект.

Иногда в качестве синонима десериализации используется термин «разбор» (parsing), однако на самом деле это совершенно разные понятия.

Пример: Веб-сервер и веб-клиент взаимодействуют по протоколу передачи гипертекста (Hyper Text Transfer Protocol, HTTP).

Протокол системы доменных имен (Domain Name Service, DNS). DNS-клиент и DNS-сервер должны использовать для взаимодействия совместимые средства сериализации и десериализации. DNS является двоичным протоколом.

Коммуникационные протоколы

Сериализация может проводиться в разных компонентах программного проекта, потому ее обычно реализуют в виде библиотек, которые можно подключать там, где они нужны. Для таких распространенных протоколов, как HTTP, DNS и FTP существуют широко известные и легко доступные сторонние библиотеки. Но если у вас есть собственные протоколы, то библиотеки сериализации для них придется писать самостоятельно.

Такие распространенные протоколы, как HTTP, FTP и DNS, — это стандарты, описанные в публичных документах, которые называются рабочими предложениями (request for comments, RFC). Например, протокол HTTP/1.1 описан в RFC-2616. В Google можно легко найти страницу соответствующего документа.

Вдобавок следует отметить, что библиотеки сериализации могут быть доступны в разных языках программирования. ***Обратите внимание:*** сам алгоритм сериализации не зависит ни от какого языка, поскольку всего лишь описывает порядок размещения байтов и способ их интерпретации. В результате нам необходим один и тот же алгоритм сериализации, реализованный с помощью разных технологий. Например, существуют HTTP-сериализаторы для C, C++, Java, Python и т. д. Двум сторонам, которые хотят общаться друг с другом, необходим четко определенный протокол. **Протокол IPC** — это стандарт, который диктует, как в целом должно осуществляться взаимодействие и какие правила относительно порядка следования байтов в сообщениях должны соблюдаться. Для передачи объектов по байтовым каналам IPC требуются алгоритмы сериализации.

Характеристики протоколов

Протоколы IPC имеют различные характеристики. Если коротко, то любой протокол должен определять разные виды содержимого для сообщений, передающихся по IPC-каналу. В одних протоколах сообщения могут иметь фиксированную длину, а в других — переменную. Некоторые протоколы вынуждают использовать предоставляемые ими операции в синхронной манере; есть и те, что поддерживают асинхронную работу. Существующие протоколы можно разделить на разные категории в зависимости от характеристик.

Тип содержимого

Сообщения, отправляемые по IPC-каналам, могут иметь текстовое, двоичное или гибридное содержимое. Двоичные данные состоят из байтов со значениями в диапазоне от 0 до 255. Текстовые представляют собой символы, используемые в тексте.

Иными словами, в текстовом содержимом допускаются только алфавитно-цифровые и некоторые дополнительные символы.

Текстовые данные можно считать частным случаем двоичных, но мы стараемся их разделять и работать с ними по-разному. Например, текстовые сообщения имеет смысл сжимать перед отправкой, а вот у двоичных сообщений плохой коэффициент сжатия (реальный размер, поделенный на размер сжатых данных). ***Следует понимать:*** одни протоколы — сугубо текстовые (например, JSON), а другие — полностью двоичные (такие как DNS). Но протоколы наподобие BSON и HTTP позволяют хранить в сообщениях сочетание из текстовых и двоичных данных; то есть итоговое сообщение может быть смесью обычных байтов и текста.

Отмечу, что двоичное содержимое может передаваться в текстовом виде. Существуют разные кодировки, которые позволяют представлять двоичные данные с помощью текстовых символов. Один из самых известных алгоритмов кодирования двоичных данных в текст, который выполняет такое преобразование, — Base64. Подобные алгоритмы широко используются в сугубо текстовых протоколах, таких как JSON, для отправки двоичной информации.

Длина сообщений

Сообщения, сгенерированные в соответствии с протоколом IPC, могут иметь либо фиксированную, либо переменную длину. Прием сообщений фиксированной или переменной длины оказывает непосредственное влияние на то, как их содержимое будет десериализовано принимающей стороной. Чтение из IPC-канала сообщений фиксированной длины, которые устроены внутри по одному и тому же принципу, дает нам отличную возможность с помощью структур языка C обратиться к их содержимому, используя заранее подготовленные поля.

Если протокол может генерировать сообщения разной длины, то определить, где заканчивается отдельно взятое сообщение, может быть непросто. ***Стоит отметить:*** для получения всего сообщения иногда необходимо прочитать несколько блоков и один из них может содержать данные, принадлежащие двум смежным сообщениям. Методы, с помощью которых разные протоколы позволяют различать или разделять сообщения с разной длиной:

- Использование разделителя. Разделитель — это последовательность байтов (в двоичных протоколах) или символов (в текстовых протоколах), которые обозначают конец сообщения.
- Обрамление фиксированной длины. В таких протоколах у каждого сообщения есть префикс фиксированной длины. Примеры— протоколы TLV (Tag-Value-Length) и ASN (Abstract Syntax Notation).
- Использование конечного автомата. Такие протоколы имеют формальную грамматику, которую можно смоделировать с помощью конечного автомата.

Последовательность

В большинстве протоколов общение между процессами проходит по принципу «запрос — ответ». Одна сторона шлет запрос, а другая на него отвечает. Такая схема обычно используется в клиент-серверных системах. Процесс-слушатель (которым зачастую выступает серверный процесс) ожидает сообщения и, когда оно приходит, отвечает соответствующим образом.

Если протокол синхронный или последовательный, то отправитель (клиент) подождет, пока слушатель (сервер) не завершит обработку запроса и не вернет ответ. То есть, пока слушатель не ответит, отправитель находится в заблокированном состоянии. В асинхронном протоколе процесс-отправитель не блокируется и может заниматься другими делами, в то время как запрос обрабатывается слушателем. Это значит, что во время подготовки ответа отправитель не будет заблокирован.

В асинхронном протоколе должен быть предусмотрен активный или пассивный механизм, позволяющий отправителю проверять наличие ответа. Активный подход означает, что отправитель будет регулярно запрашивать результат у слушателя. В случае с пассивным подходом слушатель сам передаст ответ отправителю по тому же или другому каналу взаимодействия.

Последовательность протокола не ограничена сценариями вида «запрос — ответ». Приложения для обмена сообщениями обычно используют эту методику для обеспечения максимальной отзывчивости как на серверной, так и на клиентской стороне.

Взаимодействие в рамках одного компьютера

Четыре формальных метода, с помощью которых могут общаться процессы, находящиеся в одной и той же системе:

- POSIX-сигналы;
- POSIX-каналы;
- очереди сообщений POSIX;
- сокеты домена Unix.

POSIX-сигналы, в отличие от методик, рассмотренных ранее, не создают канал связи между процессами, но могут служить для уведомления процессов о событиях. В определенных сценариях процессы могут использовать такие сигналы, чтобы сообщать друг другу о тех или иных событиях, происходящих в системе.

Прежде чем переходить к первому методу IPC — POSIX-сигналам, — обсудим файловые дескрипторы. Их так или иначе применяет любой механизм межпроцессного взаимодействия (если не считать POSIX-сигналов).

Файловые дескрипторы

Два взаимодействующих процесса могут выполняться как на одном, так и на двух разных компьютерах, соединенных по сети. Далее основное внимание уделяется первому варианту, в котором процессы находятся в одной системе. И здесь очень важную роль играют файловые дескрипторы. Они применяются и в распределенном IPC, но там называются сокетами.

Файловый дескриптор — абстрактная ссылка на локальный объект, который можно использовать для чтения и записи данных. Несмотря на свое название, файловые дескрипторы могут обозначать широкий спектр различных механизмов, предназначенных для чтения и изменения байтовых потоков.

Естественно, в число объектов, на которые могут ссылаться файловые дескрипторы, входят обычные файлы, размещенные в файловой системе (либо на жестком диске, либо в памяти).

С помощью файловых дескрипторов можно обращаться и к устройствам. У каждого устройства есть свой файл, который обычно находится в каталоге /dev.

Что касается пассивных методов IPC, то файловый дескриптор может представлять IPC-канал с возможностью чтения и записи. Вот почему первый этап подготовки IPC-канала заключается в определении ряда файловых дескрипторов.

POSIX-сигналы

В POSIX-системах процессы и потоки могут отправлять и принимать ряд заранее определенных сигналов, каждый из которых может быть отправлен процессом, потоком или даже самим ядром. На самом деле сигналы предназначены для уведомления процессов или потоков о событиях или ошибках. Например, когда система должна перезагрузиться, она рассылает всем процессам сигнал SIGTERM, уведомляя их о том, что им следует немедленно завершить работу. Процесс, получивший сигнал, должен действовать соответствующим образом. В некоторых случаях ему ничего не нужно делать, но иногда он должен сохранить текущее состояние.

У Linux есть собственные сигналы, которые не входят в стандарт POSIX. Большинство сигналов относятся к общеизвестным событиям, но два из них может определить пользователь. Обычно это нужно в ситуациях, когда во время выполнения программы должны быть совершены определенные действия.

POSIX-сигналы

Сигнал	Стандарт	Действие	Комментарий
SIGABRT	P1990	Core	Сигнал о прекращении, посланный abort(3)
SIGALRM	P1990	Term	Сигнал таймера от alarm(2)
SIGBUS	P2001	Core	Ошибка шины (некорректный доступ к памяти)
SIGCHLD	P1990	Ign	Дочерний процесс остановлен или прерван
SIGCLD	-	Ign	Синоним SIGCHLD
SIGCONT	P1990	Cont	Продолжить в случае остановки
SIGEMT	-	Term	Ловушка эмулятора
SIGFPE	P1990	Core	Неправильная операция с плавающей запятой
SIGHUP	P1990	Term	Обнаружено закрытие управляющего терминала
SIGILL	P1990	Core	Некорректная инструкция от процессора
SIGINFO	-	-	Синоним SIGPWR
SIGINT	P1990	Term	Прерывание с клавиатуры
SIGIO	-	Term	Теперь возможен ввод/вывод (4.2 BSD)

POSIX-сигналы

Сигнал	Стандарт	Действие	Комментарий
SIGIOT	-	Core	Ловушка IOT. Синоним SIGABRT
SIGKILL	P1990	Term	Сигнал принудительного завершения
SIGLOST	-	Term	Не действует блокировка файла
SIGPIPE	P1990	Term	Запись в канале, не имеющем считывающих процессов
SIGPOLL	P2001	Term	Событие, которое можно отложить (Sys V). Синоним SIGIO
SIGPROF	P2001	Term	Закончилось время профилирующего таймера
SIGPWR	-	Term	Отказ системы питания (System V)
SIGQUIT	P1990	Core	Прекратить работу с клавиатурой
SIGSEGV	P1990	Core	Некорректное обращение к памяти
SIGSTKFLT	-	Term	Ошибка в стеке сопроцессора (не используется)
SIGSTOP	P1990	Stop	Процесс остановлен
SIGTSTP	P1990	Stop	Остановка с помощью клавиатуры
SIGSYS	P2001	Core	Некорректный системный вызов (SVr4); см. также seccomp(2)

POSIX-сигналы

Сигнал	Стандарт	Действие	Комментарий
SIGTERM	P1990	Term	Сигнал снятия
SIGTRAP	P2001	Core	Ловушка отладки
SIGTTIN	P1990	Stop	Запрос на ввод с терминала для фонового процесса
SIGTTOU	P1990	Stop	Запрос на вывод с терминала для фонового процесса
SIGUNUSED	-	Core	Синоним SIGSYS
SIGURG	P2001	Ign	Приоритетные данные в сокете (4.2 BSD)
SIGUSR1	P1990	Term	Определяемый пользователем сигнал № 1
SIGUSR2	P1990	Term	Определяемый пользователем сигнал № 2
SIGVTALRM	P2001	Term	Виртуальный таймер (4.2 BSD)
SIGXCPU	P2001	Core	Превышено время работы процессора (4.2 BSD); см setrlimit(2)
SIGXFSZ	P2001	Core	Превышен размер файла (4.2 BSD); см. setrlimit(2)
SIGWINCH	-	Ign	Сигнал изменения размера окна (4.3 BSD, Sun)

POSIX-сигналы

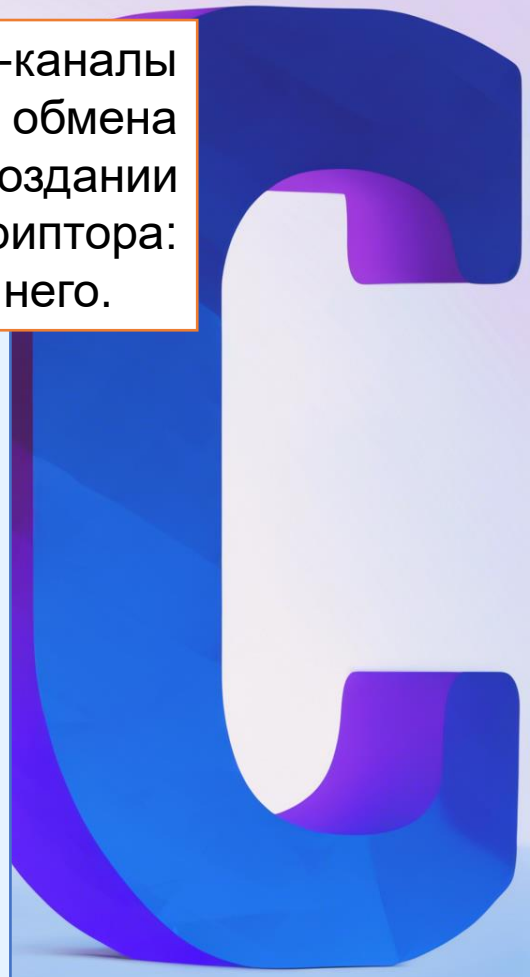
```c
void handle_user_signals(int signal) {
    switch (signal) {
    case SIGUSR1:
        printf("SIGUSR1 received!\n");
        break;
    case SIGUSR2:
        printf("SIGUSR2 received!\n");
        break;
    default:
        printf("Unsupported signal is received!\n");
    }
}
void handle_sigint(int signal) {
    printf("Interrupt signal is received!\n");
}
void handle_sigkill(int signal) {
    printf("Kill signal is received! Bye.\n");
    exit(0);
}
int main(int argc, char **argv) {
    signal(SIGUSR1, handle_user_signals);
    signal(SIGUSR2, handle_user_signals);
    signal(SIGINT, handle_sigint);
    signal(SIGKILL, handle_sigkill);
    while (1);
    return 0;
}
```

```
$ gcc main.c -o a.out
$ ./a.out &
[1] 979
$ kill -SIGUSR2 979
SIGUSR2 received!
$ kill -SIGUSR1 979
SIGUSR1 received!
$ kill -SIGINT 979
Interrupt signal is received!
$ kill -SIGKILL 979
$
[1]+  Killed                    ./a.out
$
```

POSIX-каналы

```c
int main(int argc, char **argv) {
    int fds[2];
    pipe(fds);
    int childpid = fork();
    if (childpid == -1) {... }
    if (childpid == 0) {
        close(fds[0]); // Потомок закрывает файловый дескриптор для чтения
        char str[] = "Hello Daddy!";
        // Потомок записывает в файловый дескриптор, открытый для записи
        fprintf(stdout, "CHILD: Waiting for 2 seconds ...\n");
        sleep(2);
        fprintf(stdout, "CHILD: Writing to daddy ...\n");
        write(fds[1], str, strlen(str) + 1);
    }
    else {
        close(fds[1]); // Родитель закрывает файловый дескриптор для записи
        char buff[32];
        // Родитель читает из файлового дескриптора, открытого для чтения
        fprintf(stdout, "PARENT: Reading from child ...\n");
        int num_of_read_bytes = read(fds[0], buff, 32);
        fprintf(stdout, "PARENT: Received from child: %s\n", buff);
    }
    return 0;
}
```

Unix поддерживает однонаправленные POSIX-каналы (pipes), которые можно использовать для обмена сообщениями между двумя процессами. При создании POSIX-канала вы получаете два файловых дескриптора: один для записи в канал, а второй для чтения из него.



POSIX-каналы

```
$ gcc main.c -o a.out
$ ./a.out
PARENT: Reading from child ...
CHILD: Waiting for 2 seconds ...
CHILD: Writing to daddy ...
PARENT: Received from child: Hello Daddy!
$
```

В пассивном межпроцессном взаимодействии файловый дескриптор указывает на байтовый канал, что позволяет использовать функции для работы с файловыми дескрипторами. Функции read и write принимают файловый дескриптор и производят с ним действия независимо от того, какой IPC-канал он представляет.

В примере новый процесс порождался с помощью функции fork. Но представьте ситуацию с двумя разными процессами, созданными независимо друг от друга. Возникает вопрос: каким образом они могут взаимодействовать через разделяемый канал? Процесс, который хочет иметь доступ к объекту канала, должен иметь соответствующий файловый дескриптор. Этого можно добиться двумя путями:

- один из процессов должен подготовить канал и передать соответствующие файловые дескрипторы другому процессу;
- процесс должен использовать именованный канал.

POSIX-каналы

В первом случае процессы должны применять для обмена файловыми дескрипторами канал на основе сокета домена Unix. Но проблема в том, что если между процессами уже установлен такой канал, то они могут использовать его для дальнейшего взаимодействия. В результате отпадает необходимость в POSIX-канале, у которого к тому же менее дружественный API, по сравнению с сокетами домена Unix.

Второй вариант выглядит более разумным. Один из процессов может использовать функцию mkfifo и создать файл очереди по определенному пути. Затем второй процесс может взять путь к уже созданному файлу и открыть его для последующего общения. Стоит отметить: канал по-прежнему однонаправленный и в некоторых ситуациях один из процессов должен открывать файл только для чтения, а другой — только для записи.

Пример имеет еще одну особенность, на которую стоит обратить внимание. Прежде чем записывать в канал, дочерний процесс ждет 2 секунды. А родительский процесс тем временем заблокирован на функции read. Поэтому, когда в канал ничего не записывается, читающий процесс блокируется.

В заключение напомню, что POSIX-каналы относятся к пассивным методам IPC. Как уже объяснялось ранее, пассивные методы подразумевают наличие в ядре буфера для хранения входящих сообщений, и POSIX-каналы не исключение. Записанные сообщения хранятся в ядре, пока их не прочитают. Если же процесс владелец завершает работу, то объект канала и его буфер в ядре уничтожаются.

Очереди сообщений POSIX

Очереди сообщений, размещенные в ядре, — часть стандарта POSIX. Они во многом отличаются от POSIX-каналов. Ниже перечислены некоторые фундаментальные отличия.

- В канале хранятся байты, а в очереди — сообщения. Каналу ничего не известно о структуре записываемых данных, очередь же хранит отдельные сообщения, которые добавляются при каждом вызове функции write. Очередь соблюдает границы между записанными сообщениями. ***Пример:*** у нас есть три сообщения размером 10, 20 и 30 байт и каждое из них записывается как в POSIX-канал, так и в очередь сообщений POSIX. Канал знает лишь то, что внутри у него 60 байт, и позволяет программе считывать 15-байтные фрагменты. А вот очередь сообщений знает только то, что у нее есть три сообщения, и, поскольку ни одно из них не занимает 15 байт, программа не может прочитать 15-байтный фрагмент.
- Каналы и очереди сообщений имеют ограниченный размер, который исчисляется в байтах и сообщениях соответственно. Кроме того, каждое сообщение имеет максимальный размер, измеряемый в байтах.
- Каждая очередь сообщений, такая как именованная разделяемая память или именованный семафор, открывает файл. Это не обычные файлы, но другие процессы могут использовать их для доступа к одному и тому же экземпляру очереди сообщений.
- Сообщения в очереди могут иметь разный приоритет, тогда как в канале все байты равны.

Очереди сообщений POSIX

Эти два механизма имеют и некоторые сходства:

- оба однонаправленные; для двунаправленного взаимодействия необходимо создать два экземпляра очереди или канала;
- оба имеют ограниченную емкость; вы можете записать только определенное количество байтов или сообщений;
- в большинстве POSIX-систем оба механизма представлены файловыми дескрипторами, поэтому для работы с ними можно использовать функции ввода/вывода, такие как read и write;
- обе методики работают без соединения. Иными словами, если два разных процесса запишут два разных сообщения, то один из них может прочитать сообщение другого. У сообщений нет владельца, и их могут читать любые процессы. Это чревато проблемами, особенно если с одним и тем же каналом или очередью сообщений работают несколько конкурентных процессов.

Очереди сообщений POSIX не следует путать с брокерами сообщений, применяемыми в архитектуре MQM (Message Queue Middleware — связующее ПО для работы с очередями сообщений).

В Интернете есть разные ресурсы, в которых объясняется, как работают очереди сообщений POSIX.

Очереди сообщений POSIX 1/2

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <mqueue.h>
int main(int argc, char **argv) {
    mqd_t mq; // Обработчик очереди сообщений
    struct mq_attr attr;
    attr.mq_flags = 0;
    attr.mq_maxmsg = 10;
    attr.mq_msgsize = 32;
    attr.mq_curmsgs = 0;
    int childpid = fork();
    if (childpid == -1) {
        fprintf(stderr, "fork error!\n");
        exit(1);
    }
    if (childpid == 0) {
        sleep(1); // Потомок ждет, пока родитель создает очередь
        mqd_t mq = mq_open("/mq0", O_WRONLY);
        char str[] = "Hello Daddy!";
        // Потомок записывает в файловый дескриптор, открытый для записи
        fprintf(stdout, "CHILD: Waiting for 2 seconds ...\n");
        sleep(2);
        fprintf(stdout, "CHILD: Writing to daddy ...\n");
        mq_send(mq, str, strlen(str) + 1, 0);
        mq_close(mq);
    }
```

Очереди сообщений POSIX 1/2

```
else {
        mqd_t mq = mq_open("/mq0", O_RDONLY | O_CREAT, 0644, &attr);
        char buff[32];
        fprintf(stdout, "PARENT: Reading from child ...\n");
        int num_of_read_bytes = mq_receive(mq, buff, 32, NULL);
        fprintf(stdout, "PARENT: Received from child: %s\n", buff);
        mq_close(mq);
        mq_unlink("/mq0");
    }
    return 0;
}
```

```
$ gcc main.c -o a.out
$ ./a.out
PARENT: Reading from child ...
CHILD: Waiting for 2 seconds ...
CHILD: Writing to daddy ...
PARENT: Received from child: Hello Daddy!
```

POSIX-каналы и очереди сообщений имеют ограниченный размер буфера в ядре. Следовательно, запись в канал или очередь, из которой никто не читает, может привести к блокировке любых операций записи. Иными словами, любой вызов функции write будет оставаться заблокированным, пока потребитель не прочитает сообщение из очереди или байты из канала.

Сокеты домена Unix

Еще один подход, с помощью которого разные процессы могут взаимодействовать на одном компьютере, состоит в использовании сокетов домена Unix. Это особая разновидность сокетов, работающих только в рамках одной системы. Этим они отличаются от сетевых сокетов, позволяющих двум процессам, находящимся на разных компьютерах, общаться друг с другом по сети. Сокеты домена Unix имеют различные характеристики, которые делают их важными и нетривиальными в использовании по сравнению с POSIX-каналами и очередями сообщений POSIX. Самая главная характеристика — тот факт, что они двунаправленные. Благодаря этому одного сокета достаточно для чтения и записи в связанный с ним канал. То есть каналы, основанные на сокетах домена Unix, являются полнодуплексными. Кроме того, такие сокеты могут работать как с сеансами, так и с отдельными сообщениями. Это делает их еще более гибкими. Мы вернемся к поддержке сеансов и сообщений в следующих разделах.

Поскольку сокеты домена Unix нельзя обсуждать, не имея понимания основных принципов программирования сокетов, данная тема в этой главе больше рассматриваться не будет. В следующих разделах вы познакомитесь с областью программирования сокетов и сопутствующими концепциями.

Введение в программирование сокетов

Программированием сокетов можно заниматься и локально, и на разных компьютерах. В первом случае используются сокеты домена Unix. А вот взаимодействие разных систем требует создания и применения сетевых сокетов. Оба вида сокетов имеют примерно одинаковые API и принцип работы, поэтому вполне логично, что в следующей главе они рассматриваются бок о бок.

Прежде чем начинать задействовать сетевые сокеты, необходимо знать, как работают компьютерные сети.

Компьютерные сети

Задача — дать вам базовое понимание того, что происходит внутри компьютерной сети, особенно между двумя процессами. Мы попытаемся взглянуть на это с точки зрения программиста. И главными «действующими лицами» нашего обсуждения будут процессы, а не компьютеры.

Модель OSI

Модель взаимодействия открытых систем (Open Systems Interconnection, OSI) принята в качестве стандарта Международной организацией по стандартизации (ISO) в 1983 году.

Определения:

Открытая система – система, построенная в соответствии с открытыми спецификациями (пример: Windows)

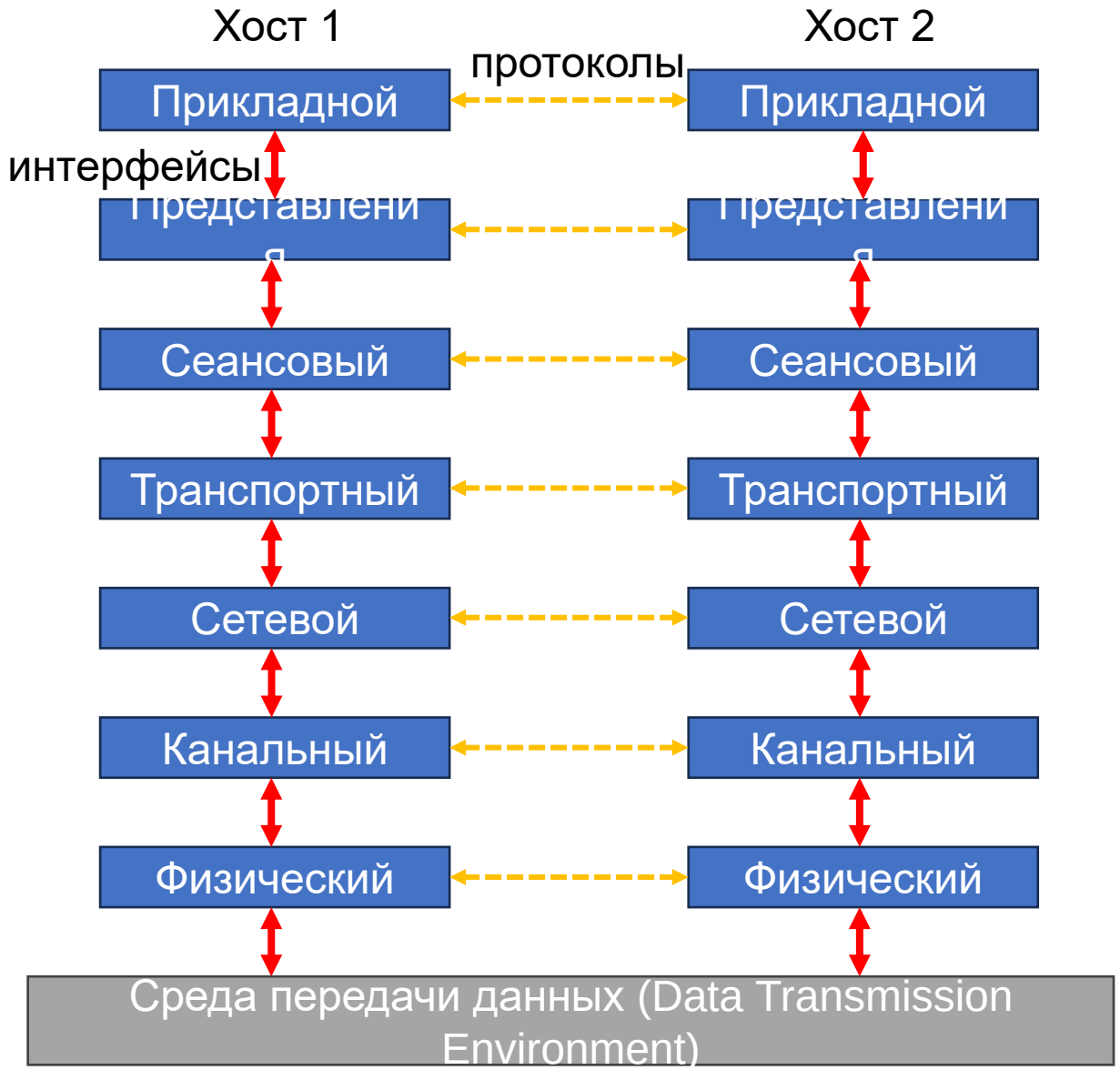
Открытая спецификация – общедоступная спецификация, соответствующая стандартам

Преимущества открытых систем:

- Возможность строить сети из оборудования разных производителей
- «Безболезненная» замена компонентов сети
- Легкость объединения нескольких сетей

OSI:

- Модель OSI описывает семь уровней организации сети и назначения каждого уровня
- Модель не является сетевой архитектурой, так как не включает описание протоколов (протоколы описаны в отдельных стандартах)
- Модель OSI используется в качестве «общего языка» для описания разных сетей



Модель OSI

Физический (physical) уровень – передачи битов по физическому каналу связи
- не вникает в смысл передаваемой информации

Задача: представить биты информации в виде сигналов, передаваемых по среде

Канальный (Data Link) уровень:
- передача сообщений по каналу связи (определение начала и конца сообщения в потоке бит)
- обнаружение и коррекция ошибок
- в широковещательной сети: управление доступом к среде передачи данных, физическая адресация

Сетевой (Network) уровень – объединяет сети, построенные на основе разных технологий. ***Задачи:***
- Создание составной сети, согласование различий в сетях
- Адресация (сетевые и глобальные адреса)
- Определение маршрута пересылки пакетов в составной сети (маршрутизация)

Транспортный (Transport) уровень – обеспечивает передачу данных между процессами на хостах.

Особенности:
- Управление надежностью (может предоставить надежность выше сети)
- Транспортный уровень – сквозной уровень (сообщения доставляются от источнику к адресату)

Сеансовый (Session) уровень – позволяет устанавливать сеансы связи. ***Задачи:***
- управление диалогом (очередность передачи сообщений)
- управление маркерами (предотвращение одновременного выполнения критической операции)
- синхронизация (метки в сообщениях для возобновления передачи в случае сбоя)

Уровень **(presentation)** представления (пример: Transport Layer Security (TLS)/Secure Sockets Layer (SSL)):
- обеспечивает согласование синтаксиса и семантики передаваемых данных (форматы представления символов, форматы чисел)
- шифрование и дешифрование

Прикладной (application) уровень – набор приложений, полезных пользователям: гипертекстовые web-страницы, социальные сети, видео и аудио связь, электронная почта, доступ к разделяемым файлам и т. д.

Единицы предаваемых данных

Уровень	Название единицы
Прикладной	Сообщение
Представления	Сообщение
Сеансовый	Сообщение
Транспортный	Сегмент/ Дейтаграмма
Сетевой	Пакет
Канальный	Кадр
Физический	Бит

Сетевое оборудование

Уровень модели OSI	Оборудование
Сетевой	Маршрутизатор
Канальный	Коммутатор, точка доступа
Физический	Концентратор

Модель и стек протоколов TCP/IP

Модель TCP/IP:

- Фактический стандарт на основе стека протоколов TCP/IP
- Описывает, как нужно строить сети на основе разных технологий, чтобы в них работал стек TCP/IP

Уровень сетевых интерфейсов – интерфейс, обеспечивающий взаимодействие с сетевыми технологиями (например, wi-fi)

Интернет = сетевой – обеспечивает поиск маршрута в составной сети, объединяющий сети, построенных на разных технологиях

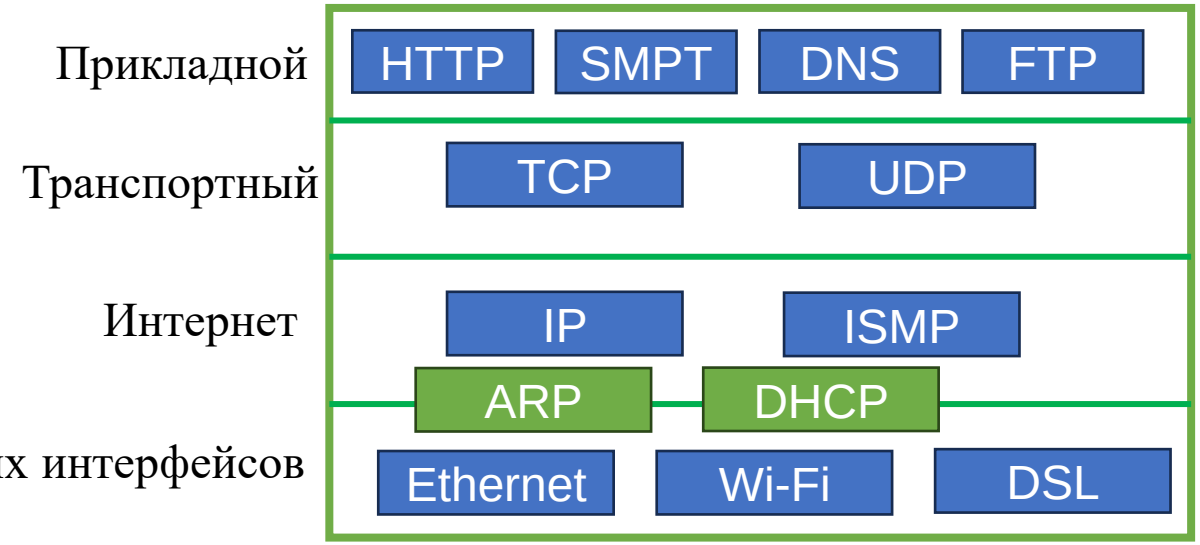
Транспортный - обеспечивает связь между двумя процессами на разных компьютерах

Прикладной = 3*OSI

Сравнение OSI и TSP/IP:

- *OSI теоретически проработана,*
- *TCP/IP широко используется, но ограничена и подходит для описания сетей на основе стека TCP/IP*
- *OSI применяется для описания разных типов сетей (Fibre Channel, Infiniband, телефонная сигнализация SS7), а TCP/IP для протоколов на основе Ethernet*

Модель OSI	Модель TCP/IP
Прикладной	Прикладной
Представления	
Сеансовый	
Транспортный	Транспортный
Сетевой	Интернет
Канальный	Сетевых интерфейсов
Физический	



Физический уровень

Для начала забудем о процессах и сосредоточимся на компьютерах. Прежде чем продолжать, отмечу, что компьютер в сети может называться по-разному: устройство, хост, узел или даже система. Конечно, чтобы понять настоящее значение того или иного термина, нужен контекст.

Первый шаг на пути к созданию распределенной системы — наличие нескольких компьютеров, соединенных по сети, или, если быть более точным, компьютерной сети. Пока ограничимся двумя компьютерами, которые нужно соединить, — двумя физическими устройствами. Соединить их между собой, очевидно, следует с помощью некой физической промежуточной среды наподобие кабеля или беспроводного соединения.

Без этой среды (которая не обязательно должна быть видимой, как в случае с беспроводной сетью) соединение было бы невозможным. Такие физические соединения похожи на дороги между городами

Любое аппаратное оборудование, необходимое для физического подключения двух устройств, относится к физическому уровню. Это первый и самый низкий уровень, который мы исследуем. Без него невозможно было бы передавать данные между двумя компьютерами и считать их соединенными. Все остальные уровни, находящиеся выше, являются программными и представляют собой набор различных стандартов, определяющих то, как должны передаваться данные.

Канальный уровень

Организация дорожного движения между городами требует не только дорог. То же самое относится и к физическому соединению между компьютерами. Чтобы пользоваться дорогами, нам нужны законы и правила, регулирующие транспортные средства, дорожные знаки, строительные материалы, бровки, скорость, разметку, направление движения и т. д. Похожие правила требуются и для прямого физического соединения между двумя компьютерами.

Если все аппаратные компоненты и устройства, необходимые для соединения разных компьютеров, принадлежат к физическому уровню, то обязательные нормы и протоколы, определяющие способ передачи данных, относятся к более высокому, канальному уровню.

Стоит упомянуть, что сеть может быть установлена между любыми двумя вычислительными устройствами, не обязательно компьютерами.

Существует множество стандартов, описывающих такие канальные соединения, — например, как настольный компьютер можно подключить к промышленному устройству. Один из самых известных канальных протоколов, предназначенный для проводного соединения компьютеров, — Ethernet. Он описывает все правила и нормы относительно передачи данных по компьютерным сетям. Еще один широко распространенный протокол, определяющий работу беспроводных сетей, называется IEEE 802.11.

Канальный уровень

Организация дорожного движения между городами требует не только дорог. То же самое относится и к физическому соединению между компьютерами. Чтобы пользоваться дорогами, нам нужны законы и правила, регулирующие транспортные средства, дорожные знаки, строительные материалы, бровки, скорость, разметку, направление движения и т. д. Похожие правила требуются и для прямого физического соединения между двумя компьютерами.

Если все аппаратные компоненты и устройства, необходимые для соединения разных компьютеров, принадлежат к физическому уровню, то обязательные нормы и протоколы, определяющие способ передачи данных, относятся к более высокому, канальному уровню.

Стоит упомянуть, что сеть может быть установлена между любыми двумя вычислительными устройствами, не обязательно компьютерами.

Существует множество стандартов, описывающих такие канальные соединения, — например, как настольный компьютер можно подключить к промышленному устройству. Один из самых известных канальных протоколов, предназначенный для проводного соединения компьютеров, — Ethernet. Он описывает все правила и нормы относительно передачи данных по компьютерным сетям. Еще один широко распространенный протокол, определяющий работу беспроводных сетей, называется IEEE 802.11.

Канальный уровень

Сеть, состоящая из компьютеров (или любых других вычислительных машин/устройств одного типа), соединенных физически с помощью определенного канального протокола, называется локальной вычислительной сетью (local area network, LAN).

Обратите внимание: любое устройство, которое подключается к LAN, должно обладать физическим компонентом под названием «сетевой адаптер» или «контроллер сетевого интерфейса» (Network Interface Controller, NIC). Например, у компьютера, который мы подключаем к сети Ethernet, должен иметься Ethernet NIC.

У компьютера может быть несколько сетевых адаптеров, каждый из которых подключен к отдельной локальной сети. Таким образом, компьютер с тремя NIC способен работать с тремя локальными сетями одновременно. Кроме того, возможно, что все три сетевых адаптера используются для подключения к одной сети.

Каждый NIC имеет уникальный адрес, который задается управляющим канальным протоколом. Этот адрес используется для передачи данных между узлами внутри LAN. Протоколы Ethernet и IEEE 802.11 назначают каждому совместимому сетевому адаптеру MAC-адрес (media access control — управление доступом к сети). Таким образом, адаптеру Ethernet или IEEE 802.11 Wi-Fi следует иметь уникальный MAC-адрес, иначе он не сможет подключиться к LAN. MAC-адреса не должны повторяться внутри одной локальной сети. В идеале им надлежит быть совершенно уникальными и неизменяемыми. Однако в реальности это не так; вы даже можете установить MAC-адрес сетевого адаптера вручную.

Сетевой уровень

Но что, если соединить нужно компьютеры из двух разных LAN, которые, к слову, могут быть несовместимы между собой?
Например, одна из них может быть проводной сетью Ethernet, а другая — FDDIсетью (fiber distributed data interface — волоконно-оптический распределенный интерфейс передачи данных), которая на физическом уровне в основном использует оптическое волокно. Еще одним примером могут быть промышленные устройства, подключенные к сети IE (Industrial Ethernet — промышленный Ethernet) и взаимодействующие с компьютерами операторов, которые обычно находятся в локальной сети Ethernet.
Сетевой уровень работает с пакетами точно так же, как канальный с кадрами. Сетевой протокол заполняет пробел между разными локальными сетями, соединяя их между собой. У каждой локальной сети могут быть свои отдельные стандарты и протоколы физического и канального уровней, но все они имеют общий управляющий сетевой протокол. В противном случае неоднородные (несовместимые) LAN не могли бы соединяться друг с другом. Самый известный сетевой протокол на текущий момент — IP (Internet Protocol — интернет-протокол). Он активно используется в крупных компьютерных сетях, которые обычно состоят из более мелких LAN на основе Ethernet или Wi-Fi. У IP есть две версии с разной длиной адресов: IPv4 и IPv6.

Сетевой уровень

Но как соединить два компьютера из двух разных LAN? Ответ кроется в механизме маршрутизации. Получение данных из внешней локальной сети требует наличия узла-маршрутизатора. Представьте, что мы хотим соединить две разные сети: LAN1 и LAN2. Маршрутизатор — обычный узел, который находится в обеих сетях благодаря наличию двух сетевых адаптеров. Один принадлежит LAN1, а другой — LAN2. Затем специальный алгоритм маршрутизации определяет, какие пакеты следует передавать между сетями и как это делать.

У каждого узла в IP-сети есть IP-адрес. Как уже было сказано ранее, существует два вида IP-адресов: IP версии 4 (IPv4) и IP версии 6 (IPv6). В IPv4 адреса состоят из четырех сегментов, каждый из которых может содержать числовое значение от 0 до 255. То есть IPv4-адреса находятся в диапазоне от 0.0.0.0 до 255.255.255.255. Как видите, для хранения адресов этого формата достаточно 4 байт (или 32 бит). Если взять IPv6-адреса, то они могут достигать 16 байт (128 бит). Кроме того, IP-адреса бывают приватными и публичными, но подробности этого выходят далеко за рамки темы, обсуждаемой в данной главе. Нам достаточно знать о том, что каждый узел в IP-сети имеет уникальный IP-адрес.

В отдельно взятой локальной сети каждый узел имеет как адрес канального уровня, так и IP-адрес, но для соединения с узлами мы будем использовать только последний. В итоге множество отдельных LAN можно объединить в одну громадную сеть. На самом деле такая сеть уже существует; мы называем ее Интернетом.

Сетевой уровень

Как и в любой другой сети, у узла, подключенного к Интернету, должен быть IP-адрес. Но главное отличие между узлом с выходом в Интернет и без него состоит в том, что первый должен иметь публичный IP-адрес, а второй может обойтись приватным.
Простейший инструмент, который позволяет убедиться в том, что два хоста (узла) в одной или разных локальных сетях могут обмениваться данными или «видеть» друг друга, — утилита ping. Она отправляет ICMP-пакеты (Internet Control Message Protocol — протокол межсетевых управляющих сообщений); если они возвращаются обратно, то это значит, другой компьютер работает, подключен к сети и отвечает на запросы.
ICMP — еще один протокол сетевого уровня, который в основном используется для мониторинга и администрирования сетей на основе IP в ситуациях, когда возникают проблемы с соединением или с качеством обслуживания.

```
$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=108 time=63.4 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=108 time=63.4 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=108 time=63.4 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=108 time=63.6 ms
^C
--- 8.8.8.8 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 63.412/63.467/63.604/0.079 ms
```

Транспортный уровень

Итак, мы уже знаем, что два компьютера можно соединить с помощью стека из трех уровней: физического, канального и сетевого. Межпроцессное взаимодействие требует, чтобы соединение и общение происходило на уровне процессов. Однако на каждом из соединенных компьютеров подобных процессов может быть много, и мы должны иметь возможность установить соединение между любыми двумя процессами, размещенными на разных компьютерах. Поэтому соединение, действующее лишь на сетевом уровне, будет слишком общим для поддержки взаимодействия нескольких отдельных процессов.

Транспортный уровень закрывает эту потребность. Если компьютеры общаются на сетевом уровне, то запущенные на них процессы могут соединяться на транспортном уровне, который функционирует поверх сетевого.

Модель «слушатель — соединитель» в контексте телефонных сетей

Использование ТСОП требует наличия телефонного аппарата. Точно так же компьютерный узел должен иметь сетевой адаптер. Помимо этого, существуют разные уровни, состоящие из различных протоколов. Эти уровни, на которые опирается инфраструктура, делают возможным создание транспортного канала.

ТСОП. Один из телефонных аппаратов ждет звонка. Пусть это будет слушающая сторона. Отмечу, что телефон, подключенный к ТСОП, всегда ожидает сигнала вызова и звонит при его получении.Теперь поговорим о другой стороне, которая инициирует звонок. Это эквивалент создания транспортного канала. Здесь тоже используется телефонный аппарат.

Транспортный уровень

Транспортное взаимодействие с соединениями и без

В рамках взаимодействия, ориентированного на соединения, для соединителя создается отдельный канал. Следовательно, если один слушатель общается с тремя соединителями, то для этого требуется три разных канала. Неважно, насколько велико передаваемое сообщение, оно дойдет до получателя в корректном виде и без какой-либо потери данных внутри канала. Если одному адресату послать несколько сообщений, то порядок их отправки будет сохранен и принимающий процесс не заметит никаких неполадок в работе инфраструктуры.

Если один из пакетов потеряется во время передачи, то операционная система получателя запросит его снова, чтобы восстановить все сообщение целиком. Например, TCP (Transport Control Protocol — протокол управления передачей) — это протокол транспортного уровня, который ведет себя в точности так.

Взаимодействие можно проводить и без соединений. Наличие соединения гарантирует два фактора: доставку отдельных пакетов и их упорядоченность. Такие протоколы, как TCP, обеспечивают одновременное выполнение этих двух условий. А вот транспортные протоколы, которые не устанавливают соединение, их не гарантируют.

Вы не можете гарантировать, что будет доставлен каждый отдельный пакет в сообщении или что все пакеты дойдут в правильном порядке. И то и другое возможно по отдельности и вместе! Например, UDP (User Datagram Protocol — протокол пользовательских датаграмм) не гарантирует доставку пакетов и их упорядоченность.

Транспортный уровень

Поток данных — последовательность байтов, передающаяся по каналу, ориентированному на соединения. Единица данных, передающихся по каналу без соединения — датаграмма. Это фрагмент данных, который можно доставить целиком в условиях отсутствия соединения. Если фрагмент данных превышает максимальный размер датаграммы, то нам не под силу гарантировать его доставку и итоговая последовательность может оказаться неверной.

Последовательность инициализации транспортного протокола

Процесс-слушатель всегда привязывается к конечной точке (обычно это сочетание IP-адреса и порта), а процесс-соединитель к ней подключается.

Последовательности инициализации при отсутствии соединения

Чтобы создать канал связи без соединения, процесс-слушатель делает следующее.

1. Привязывается к порту на одном из имеющихся сетевых интерфейсов.
2. Процесс ждет и считывает сообщения, которые становятся доступными в созданном канале, и отвечает на них, выполняя запись в тот же канал.

А процесс-соединитель выполняет следующие действия.

1. Он должен знать IP-адрес и номер порта, принадлежащие процессу-слушателю.
2. Если соединение установлено успешно, то процесс-соединитель может записывать в канал и читать из него почти таким же способом (то есть используя тот же API), что и процесс-слушатель.

Помимо выполнения описанных выше шагов, процесс-слушатель и процесс-соединитель должны использовать один и тот же транспортный протокол.

Транспортный уровень

Последовательности инициализации при использовании соединения

При использовании подхода, ориентированного на соединения, процесс-слушатель инициализируется, выполняя такую последовательность.

1. Он привязывается к порту, как и в предыдущем сценарии, в котором не было соединений. Порт ничем не отличается от описанного в прошлом разделе и подвержен тем же ограничениям.
2. Процесс слушателя задает размер очереди отставания. Она содержит ожидающие соединения, еще не принятые процессом. При взаимодействии, ориентированном на соединения, сторона слушателя получает возможность отправлять данные только после приема входящих соединений. Настроив очередь отставания, процесс-слушатель переходит в режим прослушивания.
3. Теперь процесс-слушатель начинает принимать входящие соединения. Это обязательный этап создания транспортного канала. Только приняв входящее соединение, слушатель может передать данные. Если процесс-соединитель инициирует соединение со слушателем, но тот не может его принять, то данное соединение будет оставаться в очереди отставания, пока не будет принято или пока не истечет время ожидания. Это может произойти, если процесс-слушатель слишком занят обработкой других соединений. В таком случае входящие соединения будут накапливаться в очереди отставания, и когда та заполнится, соединения начнут немедленно отклоняться операционной системой.

Прикладной уровень

Транспортный канал соединяет два процесса, находящихся на разных его концах, и позволяет им общаться друг с другом. Под общением мы понимаем передачу последовательности байтов, которую могут понять обе стороны. Как уже объяснялось в начальных разделах данной главы, для этого требуется коммуникационный протокол. Поскольку он находится на прикладном уровне и используется процессами (или приложениями, которые выполняются в виде процессов), этот протокол называется прикладным.

На канальном, сетевом и транспортном уровнях существует не так уж много протоколов, и большинство из них хорошо известны. Протоколов прикладного уровня намного больше. Это опять же аналогично ситуации в телекоммуникационных сетях. Для телефонных сетей существует всего несколько стандартов, однако люди общаются между собой на множестве языков, которые могут сильно разниться. В компьютерных сетях каждому приложению, запущенному в виде процесса, нужен прикладной протокол, позволяющий взаимодействовать с другими процессами.

Следовательно, программист использует либо общеизвестный прикладной протокол, такой как HTTP или FTP, либо спроектированный и реализованный внутри своей команды.

Набор интернет-протоколов

Ежедневно мы имеем дело с широко распространенной сетевой моделью IPS (Internet Protocol Suite — набор интернет-протоколов). Она в основном используется в Интернете, и, поскольку доступ к нему поддерживают практически все компьютеры, мы можем наблюдать ее повсеместное присутствие. Тем не менее IPS не является официальным стандартом ISO. Стандартной моделью для компьютерных сетей считается OSI (Open System Interconnections — модель взаимодействия открытых систем), которая носит скорее теоретический характер и почти никогда не развертывается и не используется в публичных сетях. Ниже перечислены уровни, из которых состоит IPS, а также известные протоколы, которые применяются на каждом из них:

- физический уровень;
- канальный уровень — Ethernet, IEEE 802.11 Wi-Fi;
- сетевой уровень — IPv4, IPv6 и ICMP;
- транспортный уровень — TCP, UDP;
- прикладной уровень — многочисленные протоколы наподобие HTTP, FTP, DNS, DHCP и многие другие.

Причина в том, что в IPS распространенными сетевыми протоколами являются только IPv4 и IPv6. В остальном к IPS применимо все представленное нами ранее.

Что такое программирование сокетов

Программирование сокетов — метод межпроцессного взаимодействия, который позволяет соединить два процесса, размещенных на одном или разных узлах с установленным между ними сетевым соединением. Если мы говорим о сценарии с двумя узлами, то они должны быть подключены к рабочей сети. Строго говоря, программирование сокетов происходит в основном на транспортном уровне.
Сокеты выступают главным средством прокладывания транспортного канала.

Что такое сокет

В ядре должна существовать абстракция, напоминающая соединение. Более того, одно ядро может инициировать и принимать много соединений, поскольку в его операционной системе может выполняться несколько процессов, которым нужен доступ к сети.
Роль этой абстракции играет сокет. Для любого соединения в системе, уже существующего или устанавливаемого, выделяется сокет, который его идентифицирует. Для отдельно взятого соединения между двумя процессами нужно по одному сокету на каждом конце. Ранее уже объяснялось, что один из этих сокетов принадлежит стороне соединителя, а другой — стороне слушателя. API, который позволяет нам определять сокеты и работать с ними, описывается библиотекой сокетов, предоставляемой операционной системой.
Поскольку нас в основном интересуют POSIX-системы, мы можем ожидать, что в составе API POSIX такая библиотека есть, и это действительно так.

Что такое программирование сокетов

Библиотека сокетов POSIX

У каждого объекта сокета есть три атрибута: домен, тип и протокол.

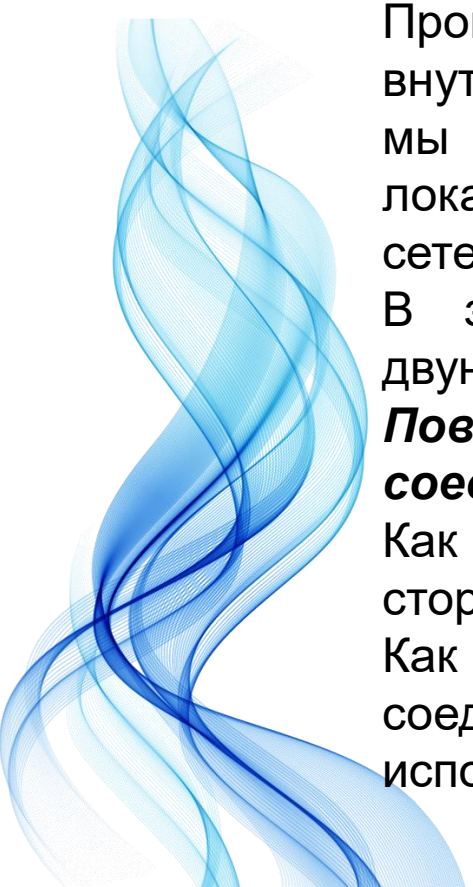
Начнем с домена, иначе называемого семейством адресов (address family, AF) или семейством протоколов (protocol family, PF).

- Сокеты AF_LOCAL или AF_UNIX — локальные; работают, только если процессы соединителя и слушателя находятся на одном компьютере.
- Сокеты AF_INET позволяют двум процессам соединяться друг с другом по IPv4.
- Сокеты AF_INET6 дают возможность двум процессам соединяться друг с другом по IPv6.

В некоторых POSIX-системах константы, которые используются в качестве домена, могут иметь префикс PF_ вместо AF_. Самые распространенные значения, которые используются в качестве типа сокет:

- Тип сокетов SOCK_STREAM представляет транспортный канал, ориентированный на соединения, с гарантией доставки, корректности и упорядоченности отправляемых данных.
- Тип сокетов SOCK_DGRAM представляет транспортный канал без поддержки соединений.
- Сокет типа SOCK_RAW может представлять каналы с соединениями и без. Основное отличие между SOCK_RAW и SOCK_DGRAM или SOCK_STREAM в том, что ядро на самом деле знает о том, какой транспортный протокол используется внутри (UDP или TCP). Однако с сокетами типа SOCK_RAW ядро этого не делает.

Что такое программирование сокетов



Третий атрибут определяет протокол, который должен использоваться для объекта сокета. Этот атрибут может быть выбран операционной системой в момент создания сокета, поскольку в большинстве случаев его можно определить по сочетанию семейства адресов и типа. Если же существует несколько потенциальных протоколов, то данный атрибут необходимо установить вручную.

Программирование сокетов предлагает решения для межпроцессного взаимодействия и внутри одной системы, и между разными компьютерами. То есть два процесса, которые мы соединяем, могут вполне находиться как на разных хостах и даже в разных локальных сетях, так и в одной и той же системе; в первом случае используются сетевые сокеты, а во втором — сокеты домена Unix.

В заключение стоит отметить, что соединения на основе сокетов являются двунаправленными и полнодуплексными.

Повторное рассмотрение последовательности инициализации слушателя и соединителя

Как уже упоминалось ранее, в почти любом соединении в компьютерной сети одна из сторон всегда ожидает входящих соединений, а другая пытается к ней подключиться. Как вы помните, ранее мы обсуждали последовательность действий слушателя и соединителя отдельно для взаимодействия с соединениями и без. Здесь будет использоваться тот же подход.

Что такое программирование сокетов

Потоковая последовательность на стороне слушателя. Процесс, который хочет принимать новые потоковые соединения (слушатель), должен выполнить следующие шаги.

1. Процесс должен создать объект сокета, используя функцию socket.
2. Теперь сокет нужно привязать к конечной точке, доступной процессу соединителя. Для этого предусмотрена функция bind.

3. Сокет должен быть сконфигурирован для прослушивания. Здесь используется функция listen.

4. После настройки очереди отставания можно приступать к приему входящих соединений. Для каждого входящего соединения должна быть вызвана функция accept. В связи с этим вызов данной функции зачастую помещают в бесконечный цикл.

Потоковая последовательность на стороне соединителя. Когда процесс соединителя хочет подключиться к процессу слушателя.

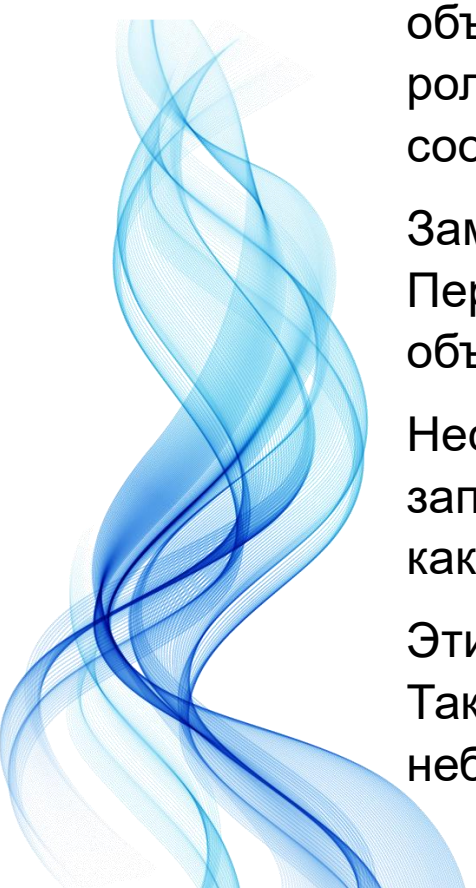
1. Процесс соединителя должен создать сокет, вызвав функцию socket.
2. Затем нужно вызвать функцию connect и передать ей аргументы, которые однозначно идентифицируют конечную точку слушателя.

Датаграммная последовательность на стороне слушателя. При инициализации датаграммного процесса слушателя выполняются следующие шаги.

1. Датаграммный слушатель, как и потоковый, создает объект сокета, вызывая функцию socket. Однако на этот раз в качестве типа сокета следует указать SOCK_DGRAM.

2. После создания слушающего сокета процесс слушателя должен привязать его к конечной точке.

У сокетов есть собственные дескрипторы!



В отличие от других пассивных методов межпроцессного взаимодействия, которые работают с файловыми дескрипторами, методики на основе сокетов имеют дело с объектами сокетов. Каждый объект имеет целочисленный идентификатор, играющий роль дескриптора сокета внутри ядра. С помощью этого дескриптора можно ссылаться на соответствующий канал.

Заметьте, что файловые дескрипторы и дескрипторы сокетов — это разные вещи. Первые указывают на обычные файлы или файлы устройств, а вторые ссылаются на объекты сокетов, созданные путем вызова функций socket, accept и connect.

Несмотря на разницу файловых дескрипторов и дескрипторов сокетов, для чтения и записи в них используется один и тот же API (или набор функций). Поэтому с сокетами, как и с файлами, можно работать с помощью функций read и write.

Эти дескрипторы имеют нечто общее: их API позволяет сделать их неблокирующими. Такие дескрипторы можно использовать для работы с файлом или сокетом в неблокирующей манере.

Адженда

Локальные сокеты и IPC	Программирование сокетов
45 минут	45 минут

Краткий обзор программирования сокетов

Две категорий межпроцессного взаимодействия:

- активные (pull-based) методики, требующие наличия доступного носителя
- пассивные (push-based) методики и подразумевает создание канала, доступного всем процессам

В активных методиках данные всегда доставляются в буфер внутри ядра, который доступен принимающему процессу с помощью дескриптора (файла или сокета). Затем принимающий процесс может либо заблокироваться, пока не станут доступными какие-то новые данные, либо запросить дескриптор и проверить, появилось ли в буфере ядра нечто новое, и в случае отрицательного ответа заняться чем-то другим. Первый подход называется блокирующим вводом/выводом, а второй — неблокирующим, или асинхронным, вводом/выводом. В этой главе все активные методики используют блокирующий подход.

Сокеты — это специальные объекты в Unix-подобных и других операционных системах, включая даже Microsoft Windows, представляющие двунаправленные каналы.

Взаимодействие, основанное на сокетах, может работать как с соединениями, так и без. В первом случае канал представляет поток байтов, передающийся между двумя определенными процессами, а во втором по каналу могут передаваться датаграммы, и при этом никакого соединения между процессами нет. Несколько процессов могут использовать один и тот же канал для разделения состояния или обмена данными.

Сокеты бывают разных типов. Каждый тип предназначен для определенных задач и ситуаций. В целом сокеты можно разделить на две категории: UDS и сетевые. Как вы уже, наверное, знаете из предыдущей главы, UDS можно использовать в случаях, когда все процессы, желающие участвовать в межпроцессном взаимодействии, находятся на одном компьютере. Иными словами, UDS подходит только для проектов, развернутых в рамках одной системы. Сетевой сокет, предоставляющий потоковый канал, обычно работает по TCP. С другой стороны, сетевой сокет, предоставляющий датаграммный канал, обычно работает по UDP.

Проект «Калькулятор»

Данный проект должен помочь вам достичь следующих целей:

- рассмотреть полнофункциональный пример с рядом простых и четко определенных возможностей;
- извлечь общие аспекты различных типов сокетов и каналов и оформить их в виде библиотек, пригодных к повторному использованию. Это существенно уменьшит количество кода, который нужно будет написать, и продемонстрирует вам границы, пролегающие между разными видами сокетов и каналов;
- организовать взаимодействие с помощью четко определенного прикладного протокола. Обычным примерам программирования сокетов недостает этого очень важного свойства. Они, как правило, демонстрируют очень простые и зачастую одноразовые сценарии взаимодействия клиента и сервера;
- поработать над примером, имеющим все характеристики полнофункциональной клиент-серверной программы, такой как прикладной протокол с поддержкой разных видов каналов, возможностью сериализации/десериализации и т. д. Это позволит вам по-новому взглянуть на программирование сокетов.

Иерархия исходного кода

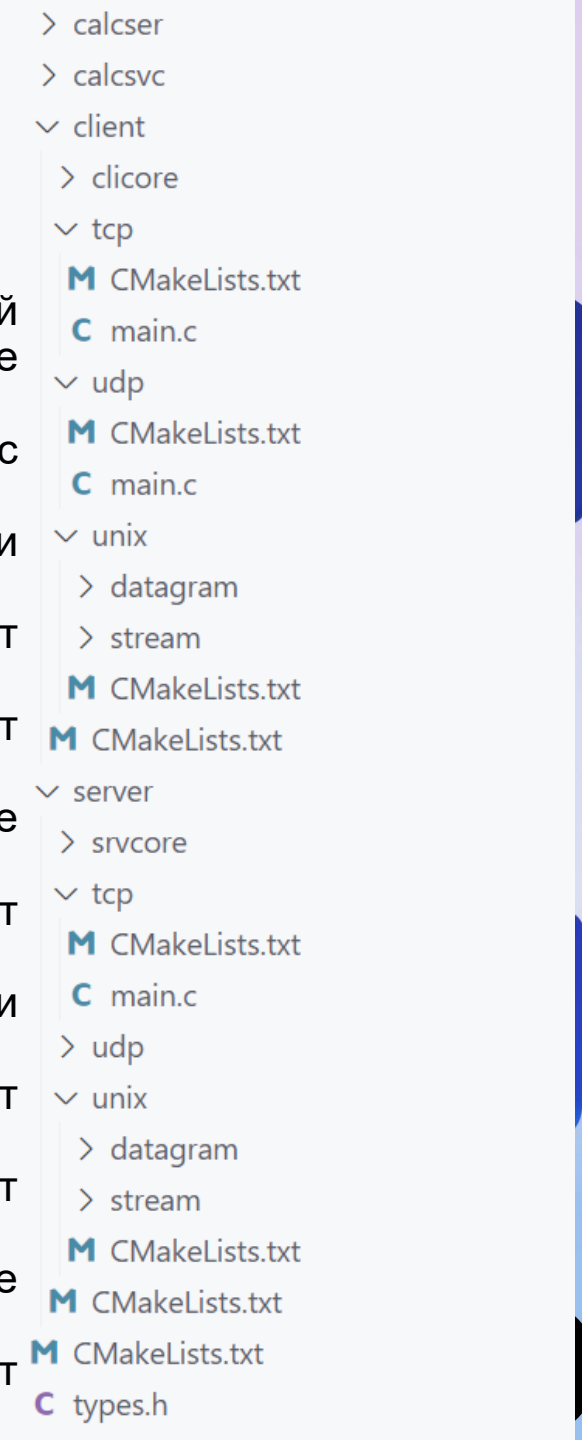
Данный проект должен помочь вам достичь следующих целей:

- рассмотреть полнофункциональный пример с рядом простых и четко определенных возможностей;
- извлечь общие аспекты различных типов сокетов и каналов и оформить их в виде библиотек, пригодных к повторному использованию. Это существенно уменьшит количество кода, который нужно будет написать, и продемонстрирует вам границы, пролегающие между разными видами сокетов и каналов;
- организовать взаимодействие с помощью четко определенного прикладного протокола. Обычным примерам программирования сокетов недостает этого очень важного свойства. Они, как правило, демонстрируют очень простые и зачастую одноразовые сценарии взаимодействия клиента и сервера;
- поработать над примером, имеющим все характеристики полнофункциональной клиент-серверной программы, такой как прикладной протокол с поддержкой разных видов каналов, возможностью сериализации/десериализации и т. д. Это позволит вам по-новому взглянуть на программирование сокетов.

Иерархия исходного кода

Проект состоит из ряда компонентов, в том числе и библиотек:

- Библиотека сериализации/десериализации в каталоге /calcser. Она описывает прикладной протокол, по которому общаются клиентская и серверная часть калькулятора. В итоге собирается в статическую библиотеку libcalcser.a.
- Библиотека в каталоге /calcsvc содержит исходники вычислительного сервиса. Данный сервис предоставляет основные функции калькулятора и не привязан к серверному процессу.
- Библиотека в каталоге /server/srvcore содержит исходники, общие для потоковых и датаграммных процессов, независимо от типа сокета.
- Каталог /server/unix/stream содержит исходники серверной программы, которая использует потоковые каналы внутри сокетов UDS.
- Каталог /server/unix/datagram содержит исходники серверной программы, которая использует датаграммные каналы внутри сокетов UDS.
- Каталог /server/tcp содержит исходники серверной программы, которая использует потоковые каналы внутри сетевых сокетов TCP.
- Каталог /server/udp содержит исходники серверной программы, которая использует датаграммные каналы внутри сетевых сокетов UDP.
- Библиотека в каталоге /client/clicore содержит исходники, общие для потоковых и датаграммных клиентских процессов, независимо от типа сокета.
- Каталог /client/unix/stream содержит исходники клиентской программы, которая использует потоковые каналы внутри сокетов UDS.
- Каталог /client/unix/datagram содержит исходники клиентской программы, которая использует датаграммные каналы внутри сокетов UDS.
- Каталог /client/tcp содержит исходники клиентской программы, которая использует потоковые каналы внутри сокетов TCP.
- Каталог /client/udp содержит исходники клиентской программы, которая использует датаграммные каналы внутри сокетов UDP.



Сборка проекта

Итак, мы прошлись по всем каталогам проекта. Теперь нам нужно показать, как он собирается. Проект использует систему CMake, поэтому, прежде чем переходить к сборке, убедитесь в том, что она у вас установлена. Пример 32.4

Чтобы собрать проект, выполните следующие команды в корневом каталоге главы:

```
$ sudo apt-get update
$ sudo apt-get install libcmocka-dev
$ mkdir -p build
$ cd build
$ cmake -G "Unix Makefiles" ..
...
$ make
...
$
```

Запуск проекта

Ничто так не помогает убедиться в работоспособности проекта, как его самостоятельный запуск. Поэтому, прежде чем переходить к техническим подробностям, я хочу, чтобы вы по очереди запустили серверную и клиентскую части калькулятора и понаблюдали за тем, как они общаются друг с другом.

Перед запуском процессов необходимо открыть два отдельных терминала (или командные оболочки), чтобы ввести два разных набора команд. В первом терминале мы запустим потоковый сервер, прослушивающий сокет UDS. Обратите внимание: перед вводом этой команды вы должны перейти в каталог build, который был создан в рамках предыдущего раздела.

Соответствующая команда приводится ниже:

```
$ ./server/unix/stream/unix_stream_calc_server
```

Убедитесь в том, что сервер работает. Запустите во втором терминале потоковый клиент, собранный для использования UDS. Запуск клиентской части калькулятора и отправка нескольких запросов:

```
$ ./client/unix/stream/unix_stream_calc_client
? (type quit to exit) 3++4
The req(0) is sent.
req(0) > status: OK, result: 7.000000
? (type quit to exit) mem
The req(1) is sent.
req(1) > status: OK, result: 7.000000
? (type quit to exit) 5++4
The req(2) is sent.
req(2) > status: OK, result: 16.000000
? (type quit to exit) quit
Bye.
$
```

Прикладной протокол

Любые два процесса, которые хотят общаться друг с другом, должны соблюдать общий прикладной протокол. Он может быть написан специально для проекта «Калькулятор», но мы можем воспользоваться и общеизвестным стандартом, таким как HTTP. Наш протокол будет называться протоколом калькулятора.

Сообщения в протоколе калькулятора имеют переменную длину. То есть длина сообщений может отличаться, и между ними должны находиться разделители. У нас будет один тип запросов и один тип ответов. Вдобавок следует сказать, что протокол будет текстовым. То есть запросы и ответы могут состоять только из алфавитно-цифровых и нескольких других символов. Это позволит сделать сообщения калькулятора понятными человеку.

Запрос состоит из четырех полей: идентификатора, метода, первого и второго операнда. Каждый запрос обладает уникальным идентификатором, благодаря которому сервер знает, кому отправлять соответствующий ответ.

Метод — операция, выполняемая сервисом калькулятора.

Определение объекта запроса (calcser/calc_proto_req.h)

```
#ifndef CALC_PROTO_REQ_H
#define CALC_PROTO_REQ_H

#include <stdint.h>

typedef enum {
  NONE,
  GETMEM, RESMEM,
  ADD, ADDM,
  SUB, SUBM,
  MUL, MULM,
  DIV
} method_t;

struct calc_proto_req_t {
  int32_t id;
  method_t method;
  double operand1;
  double operand2;
};

method_t str_to_method(const char*);
const char* method_to_str(method_t);

#endif
```

Прикладной протокол

Представьте, что клиент хочет создать запрос с ID 1000, методом ADD и двумя операндами: 1.5 и 5.6. В языке C для этого нужно создать объект calc_proto_req_t и заполнить его нужными значениями. Создание объекта запроса на C:

```
struct calc_proto_req_t req;
req.id = 1000;
req.method = ADD;
req.operand1 = 1.5;
req.operand2 = 5.6;
```

Объект req в этом листинге можно отправить серверу только после сериализации и превращения его в запрос. Иными словами, нам нужно сериализовать данный объект запроса в соответствующее сообщение запроса. В соответствии с нашим прикладным протоколом результат сериализации объекта req будет выглядеть следующим образом:

```
1000#ADD#1.5#5.6$
```

Символ # служит для разделения полей, а символ $ играет роль разделителя сообщений. Кроме того, у каждого сообщения есть ровно четыре поля.

Прикладной протокол

Десериализатор на другом конце канала использует эти факты для разбора входящих байтов и воссоздания оригинального объекта.

С другой стороны, серверный процесс, который отвечает на запрос, должен сериализовать объект ответа. Ответ состоит из трех полей: идентификатора запроса, статуса и результата. Идентификатор уникален и указывает на запрос, на который хочет ответить сервер.

Заголовочный файл calcser/calc_proto_resp.h описывает то, как должен выглядеть ответ.

По аналогии с объектом запроса req, серверный процесс создает объект ответа, выполняя следующие инструкции:

```
struct calc_proto_resp_t resp;
resp.req_id = 1000;
resp.status = STATUS_OK;
resp.result = 7.1;
```

Результат сериализации этого объекта выглядит так:

```
1000#0#7.1$
```

```
#ifndef CALC_PROTO_RESP_H
#define CALC_PROTO_RESP_H

#include <stdint.h>

#define STATUS_OK               0
#define STATUS_INVALID_REQUEST  1
#define STATUS_INVALID_METHOD   2
#define STATUS_INVALID_OPERAND  3
#define STATUS_DIV_BY_ZERO      4
#define STATUS_INTERNAL_ERROR   20

typedef int status_t;

struct calc_proto_resp_t {
  int32_t req_id;
  status_t status;
  double result;
};

#endif
```

Библиотека сериализации/десериализации

Здесь же поговорим об алгоритмах сериализации и десериализации, которые используются в проекте «Калькулятор». Предоставление соответствующих операций будет выполняться с помощью класса serializer с calc_proto_ser_t в качестве структуры атрибутов.
Публичный API класса serializer находится в файле calcser/ calc_proto_ser.h.
Помимо конструктора и деструктора, которые нужны для создания и уничтожения объекта serializer, у нас есть две пары функций: первая пара для серверного процесса, а вторая — для клиентского.
На клиентской стороне мы сериализуем объект запроса и десериализуем сообщение с ответом. А на серверной стороне десериализуем сообщение с запросом и сериализуем объект ответа.
Помимо операций сериализации и десериализации, у нас также есть три функции обратного вызова:

- обратный вызов для получения объекта запроса, десериализованного из соответствующего канала;
- обратный вызов для получения объекта ответа, который был десериализован из соответствующего канала;
- обратный вызов для получения ошибки в случае провала сериализации или десериализации.

Эти обратные вызовы используются клиентскими и серверными процессами для получения входящих запросов и ответов, а также ошибок, которые могут возникнуть при сериализации или десериализации сообщения.

Функции сериализации/десериализации для клиентской стороны

Функция сериализации ответа для серверной стороны (calcser/calc_proto_ser.c)

```
struct buffer_t calc_proto_ser_server_serialize(
    struct calc_proto_ser_t* ser,
    const struct calc_proto_resp_t* resp) {
  struct buffer_t buff;
  char resp_result_str[64];
  _serialize_double(resp_result_str, resp->result);
  buff.data = (char*)malloc(64 * sizeof(char));
  sprintf(buff.data, "%d%c%d%c%s%c", resp->req_id,
          FIELD_DELIMITER, (int)resp->status, FIELD_DELIMITER,
      resp_result_str, MESSAGE_DELIMITER);
  buff.len = strlen(buff.data);
  return buff;
}
```

Функция десериализации запроса для серверной стороны (calcser/calc_proto_ser.c)

```
void calc_proto_ser_server_deserialize(
    struct calc_proto_ser_t* ser,
    struct buffer_t buff,
    bool_t* req_found) {
  if (req_found) {
    *req_found = FALSE;
  }
  _deserialize(ser, buff, _parse_req_and_notify,
          ERROR_INVALID_REQUEST, req_found);
}
```

Функции сериализации/десериализации для серверной стороны

Функция сериализации запроса для клиентской стороны (calcser/calc_proto_ser.c)

```
struct buffer_t calc_proto_ser_client_serialize(
    struct calc_proto_ser_t* ser,
    const struct calc_proto_req_t* req) {
  struct buffer_t buff;
  char req_op1_str[64];
  char req_op2_str[64];
  _serialize_double(req_op1_str, req->operand1);
  _serialize_double(req_op2_str, req->operand2);
  buff.data = (char*)malloc(64 * sizeof(char));
  sprintf(buff.data, "%d%c%s%c%s%c%s%c", req->id, FIELD_DELIMITER,
          method_to_str(req->method), FIELD_DELIMITER,
          req_op1_str, FIELD_DELIMITER, req_op2_str,
          MESSAGE_DELIMITER);
  buff.len = strlen(buff.data);
  return buff;
}
```

Функция десериализации ответа для клиентской стороны (calcser/calc_proto_ser.c)

```
void calc_proto_ser_client_deserialize(
    struct calc_proto_ser_t* ser,
    struct buffer_t buff, bool_t* resp_found) {
  if (resp_found) {
    *resp_found = FALSE;
  }
  _deserialize(ser, buff, _parse_resp_and_notify,
          ERROR_INVALID_RESPONSE, resp_found);
}
```

Сервис калькулятора

Сервис калькулятора — основная логика нашего примера. Стоит отметить, что он должен работать независимо от того или иного механизма межпроцессного взаимодействия. Как видите, этот сервис спроектирован таким образом, что его можно использовать даже в простейшей программе, состоящей из одной лишь функции main и не имеющей никакого отношения к IPC.

```
#ifndef CALC_SERVICE_H
#define CALC_SERVICE_H
#include <types.h>
static const int CALC_SVC_OK = 0;
static const int CALC_SVC_ERROR_DIV_BY_ZERO = -1;
struct calc_service_t;
struct calc_service_t* calc_service_new();
void calc_service_delete(struct calc_service_t*);
void calc_service_ctor(struct calc_service_t*);
void calc_service_dtor(struct calc_service_t*);
void calc_service_reset_mem(struct calc_service_t*);
double calc_service_get_mem(struct calc_service_t*);
double calc_service_add(struct calc_service_t*, double, double b, bool_t mem);
double calc_service_sub(struct calc_service_t*, double, double b, bool_t mem);
double calc_service_mul(struct calc_service_t*, double, double b, bool_t mem);
int calc_service_div(struct calc_service_t*, double, double, double*);
#endif
```

Сокеты домена Unix

При установлении соединения между двумя процессами, размещенными на одном компьютере, одним из лучших решений являются сокеты домена Unix (Unix domain sockets, UDS). Теперь пришло время объединить эти знания и посмотреть на UDS в действии.

Текущий раздел состоит из четырех частей, посвященных разным видам процессов, находящимся на стороне слушателя или соединителя и использующим потоковые или датаграммные каналы. Все эти процессы работают с сокетами UDS. Мы рассмотрим все шаги, которые они должны выполнить в целях создания канала с учетом последовательностей.

Потоковый сервер на основе UDS

Сервер играет роль слушателя, поэтому должен выполнять его последовательность. В частности, поскольку в данном подразделе речь идет о потоковом канале, ему следует использовать последовательность потокового слушателя. В рамках этой последовательности сервер должен сначала создать объект сокета. В нашем проекте потоковый сервер, желающий принимать соединения по UDS, должен выполнять те же шаги.

Создание потокового объекта UDS (server/unix/stream/main.c)

```
int server_sd = socket(AF_UNIX, SOCK_STREAM, 0);
if (server_sd == -1) {
  fprintf(stderr, "Could not create socket: %s\n",
          strerror(errno));
  exit(1);
}
```

Как видите, объект сокета создается с помощью функции socket. Она подключается из заголовка <sys/socket.h>, который входит в стандарт POSIX.

Обратите внимание: мы пока не знаем, каким будет данный объект: клиентским или серверным. Это смогут определить только последующие функции.

У каждого объекта сокета есть три атрибута, которые определяются тремя аргументами, переданными функции socket. Эти аргументы указывают семейство адресов, тип и протокол, которые будет использовать данный объект.

Потоковый сервер на основе UDS

Согласно последовательности инициализации потокового слушателя, особенно той ее части, которая относится к UDS после создания объекта сокета, серверная программа должна привязаться к файлу сокета. Это будет наш следующий шаг. Привязка потокового объекта UDS к файлу сокета, заданному с помощью массива символов sock_file (server/unix/stream/main.c):

```
struct sockaddr_un addr;
  memset(&addr, 0, sizeof(addr));
  addr.sun_family = AF_UNIX;
  strncpy(addr.sun_path, sock_file, sizeof(addr.sun_path) - 1);
  int result = bind(server_sd, (struct sockaddr*)&addr, sizeof(addr));
  if (result == -1) {
    close(server_sd);
    fprintf(stderr, "Could not bind the address: %s\n", strerror(errno));
    exit(1);
  }
```

Данный код состоит из двух этапов. На первом создается экземпляр типа struct sockaddr_un с именем addr, который после инициализации указывает на файл сокета. На втором этапе объект addr передается функции bind, чтобы она знала, какой файл следует привязать к объекту сокета. Вызов данной функции завершается успешно, только если к заданному файлу не привязан никакой другой объект. Следовательно, в случае с UDS два объекта сокетов, которые, вероятно, принадлежат разным процессам, не могут быть привязаны к одному и тому же файлу.

Потоковый сервер на основе UDS

Продолжая говорить о пути к файлу сокета, следует отметить, что в большинстве систем Unix он не может превышать 104 байтов. Однако в системах Linux его длина составляет 108 байт. Обратите внимание: строковая переменная, которая хранит этот путь в виде массива типа char, всегда содержит в конце дополнительный нулевой символ. Поэтому путь к файлу сокета фактически имеет длину 103 или 107 байт в зависимости от операционной системы.

Если функция bind возвращает 0, то это значит, что привязка прошла успешно и вы можете приступать к следующему шагу в последовательности потокового слушателя: настройке размера очереди отставания.

Настройка размера очереди отставания для привязанного потокового сокета (server/unix/stream/main.c)

```
result = listen(server_sd, 10);
  if (result == -1) {
    close(server_sd);
    fprintf(stderr, "Could not set the backlog: %s\n",
            strerror(errno));
    exit(1);
  }
```

Функция listen задает размер очереди отставания для уже привязанного сокета. Когда занятый серверный процесс не в состоянии принимать от клиентов дальнейшие входящие запросы, некоторое количество этих запросов может подождать в очереди отставания, пока у программы не появится возможность их обработать.

Потоковый сервер на основе UDS

Согласно последовательности действий потокового слушателя, вслед за привязкой потокового сокета и настройки размера его очереди отставания мы можем приступить к приему новых клиентских запросов. Принятие новых клиентских запросов с помощью сокета потокового слушателя (server/unix/stream/main.c)

```
while (1) {
  int client_sd = accept(server_sd, NULL, NULL);
  if (client_sd == -1) {
    close(server_sd);
    fprintf(stderr, "Could not accept the client: %s\n", strerror(errno));
    exit(1);
  }
  ...
}
```

Все волшебство происходит в функции accept, которая возвращает новый сокет при получении нового запроса со стороны клиента. Возвращаемый объект сокета указывает на соответствующий потоковый канал, созданный между сервером и клиентом, запрос которого был принят. Обратите внимание: у клиента есть свой потоковый канал и, следовательно, собственный дескриптор сокета.

Отмечу: если потоковый слушающий сокет является блокирующим (что происходит по умолчанию), то функция accept блокирует выполнение, пока не поступит новый клиентский запрос. То есть при отсутствии новых запросов поток выполнения, вызывающий функцию accept, блокируется на ней.

Потоковый сервер на основе UDS

```
void accept_forever(int server_sd) {
  while (1) {
    int client_sd = accept(server_sd, NULL, NULL);
    if (client_sd == -1) {
      close(server_sd);
      fprintf(stderr, "Could not accept the client: %s\n", strerror(errno));
      exit(1);
    }
    pthread_t client_handler_thread;
    int* arg = (int *)malloc(sizeof(int));
    *arg = client_sd;
    int result = pthread_create(&client_handler_thread, NULL, &client_handler, arg);
    if (result) {
      close(client_sd);
      close(server_sd);
      free(arg);
      fprintf(stderr, "Could not start the client handler thread.\n");
      exit(1);
    }
  }
}
```

Потоковый сервер на основе UDS

```
void* client_handler(void *arg) {
  struct client_context_t context;
  context.addr = (struct client_addr_t*)malloc(sizeof(struct client_addr_t));
  context.addr->sd = *((int*)arg);
  free((int*)arg);
  context.ser = calc_proto_ser_new();
  calc_proto_ser_ctor(context.ser, &context, 256);
  calc_proto_ser_set_req_callback(context.ser, request_callback);
  calc_proto_ser_set_error_callback(context.ser, error_callback);
  context.svc = calc_service_new();
  calc_service_ctor(context.svc);
  context.write_resp = &stream_write_resp;
  int ret;
  char buffer[128];
  while (1) {
    int ret = read(context.addr->sd, buffer, 128);
    if (ret == 0 || ret == -1)
      break;
    struct buffer_t buf;
    buf.data = buffer; buf.len = ret;
    calc_proto_ser_server_deserialize(context.ser, buf, NULL);
  }
  calc_service_dtor(context.svc);
  calc_service_delete(context.svc);
  calc_proto_ser_dtor(context.ser);
  calc_proto_ser_delete(context.ser);
  free(context.addr);
  return NULL;
}
```

Потоковый сервер на основе UDS

Заключительным аспектом, на который стоит обратить внимание, является то, что ответы клиенту возвращаются с помощью функции stream_write_response, предназначенной для работы с потоковым сокетом. Функция, которая используется для возвращения ответов клиенту (server/srvcore/stream_server_core.c):

```
void stream_write_resp(struct client_context_t* context, struct calc_proto_resp_t* resp) {
  struct buffer_t buf = calc_proto_ser_server_serialize(context->ser, resp);
  if (buf.len == 0) {
    close(context->addr->sd);
    fprintf(stderr, "Internal error while serializing response\n");
    exit(1);
  }
  int ret = write(context->addr->sd, buf.data, buf.len);
  free(buf.data);
  if (ret == -1) {
    fprintf(stderr, "Could not write to client: %s\n", strerror(errno));
    close(context->addr->sd);
    exit(1);
  } else if (ret < buf.len) {
    fprintf(stderr, "WARN: Less bytes were written!\n");
    exit(1);
  }
```

Потоковый клиент на основе UDS

Клиент должен первым делом создать объект сокета. Мы должны выполнить последовательность шагов потокового соединителя. Здесь используется такой же фрагмент кода, что и на серверной стороне, включая те же аргументы, которые сигнализируют о работе с UDS. Главная функция потокового клиента, соединяющегося с конечной точкой UDS (client/unix/stream/main.c)

```
int main(int argc, char** argv) {
  char sock_file[] = "/tmp/calc_svc.sock";
  // ----------- 1. Create socket object -----------------
  int conn_sd = socket(AF_UNIX, SOCK_STREAM, 0);
  if (conn_sd == -1) {
    fprintf(stderr, "Could not create socket: %s\n", strerror(errno));
    exit(1);
  }
  // ----------- 2. Connect to server --------------------
  struct sockaddr_un addr;
  memset(&addr, 0, sizeof(addr));
  addr.sun_family = AF_UNIX;
  strncpy(addr.sun_path, sock_file, sizeof(addr.sun_path) - 1);
  int result = connect(conn_sd, (struct sockaddr*)&addr, sizeof(addr));
  if (result == -1) {
    close(conn_sd);
    fprintf(stderr, "Could no connect: %s\n", strerror(errno));
    exit(1);
  }
  stream_client_loop(conn_sd);
  return 0;
}
```

Потоковый клиент на основе UDS

Данный код демонстрирует, что все клиентские процессы имеют один общий объект-сериализатор, и это логично. Более того, клиентский процесс создает новый поток выполнения для чтения ответов, которые присылает сервер. Это вызвано тем, что чтение из серверного процесса — блокирующая операция, поэтому ее следует выполнять в отдельном потоке.

```
void stream_client_loop(int conn_sd) {
  struct context_t context;
  context.sd = conn_sd;
  context.ser = calc_proto_ser_new();
  calc_proto_ser_ctor(context.ser, &context, 128);
  calc_proto_ser_set_resp_callback(context.ser, on_response);
  calc_proto_ser_set_error_callback(context.ser, on_error);
  pthread_t reader_thread;
  pthread_create(&reader_thread, NULL, stream_response_reader, &context);
  char buf[128];
  printf("? (type quit to exit) ");
  while (1) {
    ...
    struct buffer_t ser_req = calc_proto_ser_client_serialize(context.ser, &req);
    int ret = write(context.sd, ser_req.data, ser_req.len);
  }
  shutdown(conn_sd, SHUT_RD);
  calc_proto_ser_dtor(context.ser);
  calc_proto_ser_delete(context.ser);
  pthread_join(reader_thread, NULL);
  printf("Bye.\n");
}
```

Датаграммный сервер на основе UDS

датаграммные процессы, слушатель и соединитель, имеют собственные последовательности действий по передаче данных. Пришло время показать, как может выглядеть датаграммный сервер на основе UDS. Создание объекта UDS, предназначенного для работы с датаграммным каналом (server/unix/datagram/main.c):

```
int server_sd = socket(AF_UNIX, SOCK_DGRAM, 0);
 if (server_sd == -1) {
   fprintf(stderr, "Could not create socket: %s\n", strerror(errno));
   exit(1);
 }
```

Вместо SOCK_STREAM мы используем SOCK_DGRAM. Это значит, объект сокета будет работать с датаграммным каналом. Остальные два аргумента остаются без изменений.

Второй шаг в последовательности датаграммного слушателя состоит в привязке сокета к конечной точке UDS. Как уже говорилось ранее, это файл сокета. Данный шаг ничем не отличается от того, который мы выполнили в потоковом сервере.

Это все действия, которые выполняет слушающий датаграммный процесс; датаграммному сокету не назначается очередь отставания. Более того, здесь нет этапа приема клиентских запросов, поскольку у нас не может быть клиентских соединений с выделенным каналом между двумя процессами.

Датаграммный сервер на основе UDS

```
int main(int argc, char** argv) {
  char sock_file[] = "/tmp/calc_svc.sock";
  // ----------- 1. Create socket object -----------------
  int server_sd = socket(AF_UNIX, SOCK_DGRAM, 0);
  if (server_sd == -1) {
    fprintf(stderr, "Could not create socket: %s\n", strerror(errno));
    exit(1);
  }
  // ----------- 2. Bind the socket file -----------------
  // Delete the previously created socket file if it exists.
  unlink(sock_file);
  // Prepare the address
  struct sockaddr_un addr;
  memset(&addr, 0, sizeof(addr));
  addr.sun_family = AF_UNIX;
  strncpy(addr.sun_path, sock_file, sizeof(addr.sun_path) - 1);
  int result = bind(server_sd, (struct sockaddr*)&addr, sizeof(addr));
  if (result == -1) {
    close(server_sd);
    fprintf(stderr, "Could not bind the address: %s\n", strerror(errno));
    exit(1);
  }
  // ----------- 3. Start serving requests ---------
  serve_forever(server_sd);
  return 0;
}
```

Датаграммный сервер на основе UDS

Датаграммные каналы не поддерживают соединения и работают не так, как потоковые каналы. Иными словами, мы не можем установить выделенное соединение между двумя процессами. Поэтому процессы передают по каналу только отдельные фрагменты данных. Клиент отправляет отдельные и независимые друг от друга датаграммы, а сервер их принимает и в свою очередь возвращает другие датаграммы в качестве ответа.

Таким образом, важнейшим аспектом датаграммного канала является то, что сообщение с запросом или ответом должно влезать в одну датаграмму. В противном случае его нельзя разделить между двумя датаграммами, и ни сервер, ни клиент не смогут его обработать. К счастью, сообщения в нашем проекте в основном достаточно короткие.

Размер датаграммы во многом зависит от канала, по которому она проходит. Например, датаграммы UDS являются довольно гибкими, поскольку проходят через ядро. А вот при использовании UDP-сокетов все зависит от конфигурации сети. Что касается UDS, то информация более подробно объясняет, как выбрать корректный размер.

Еще одно различие между датаграммными и потоковыми сокетами, заслуживающее внимания, состоит в том, что для передачи данных по ним используются разные API ввода/вывода. Для работы с датаграммным сокетом, как и с потоковым, можно применять операции read и write, однако для чтения и отправки данных в датаграммный канал мы обычно используем другие функции: recvfrom и sendto.

Это вызвано тем, что потоковые сокеты имеют выделенный канал и при записи в него мы знаем, что находится на обоих его концах. Касательно датаграммных сокетов, один и тот же канал используется множеством разных сторон.

Датаграммный сервер на основе UDS

```
void serve_forever(int server_sd) {
  char buffer[64];
  while (1) {
    struct sockaddr* sockaddr = sockaddr_new();
    socklen_t socklen = sockaddr_sizeof();
    int read_nr_bytes = recvfrom(server_sd, buffer, sizeof(buffer), 0, sockaddr, &socklen);
…
    struct client_context_t context;
    context.addr = (struct client_addr_t*) malloc(sizeof(struct client_addr_t));
    context.addr->server_sd = server_sd;
    context.addr->sockaddr = sockaddr;
    context.addr->socklen = socklen;

    context.ser = calc_proto_ser_new();
    calc_proto_ser_ctor(context.ser, &context, 256);
    calc_proto_ser_set_req_callback(context.ser, request_callback);
    calc_proto_ser_set_error_callback(context.ser, error_callback);

    context.svc = calc_service_new();
    calc_service_ctor(context.svc);

    context.write_resp = &datagram_write_resp;

    bool_t req_found = FALSE;
    struct buffer_t buf;
    buf.data = buffer;
    buf.len = read_nr_bytes;
    calc_proto_ser_server_deserialize(context.ser, buf, &req_found);
…
  }
}
```

Датаграммный сервер на основе UDS

```
void datagram_write_resp(struct client_context_t* context,
        struct calc_proto_resp_t* resp) {
  struct buffer_t buf = calc_proto_ser_server_serialize(context->ser, resp);
  if (buf.len == 0) {
    close(context->addr->server_sd);
    fprintf(stderr, "Internal error while serializing object.\n");
    exit(1);
  }
  int ret = sendto(context->addr->server_sd, buf.data, buf.len,
      0, context->addr->sockaddr, context->addr->socklen);
  free(buf.data);
  if (ret == -1) {
    fprintf(stderr, "Could not write to client: %s\n", strerror(errno));
    close(context->addr->server_sd);
    exit(1);
  } else if (ret < buf.len) {
    fprintf(stderr, "WARN: Less bytes were written!\n");
    close(context->addr->server_sd);
    exit(1);
  }
}
```

Датаграммный клиент на основе UDS

С технической точки зрения потоковые и датаграммные клиенты очень похожи. Это значит, что их общая структура должна быть почти идентичной, а различия будут связаны с тем, что мы передаем датаграммы, вместо того чтобы работать с потоковым каналом.

Однако существует одна важная особенность, довольно уникальная и присущая датаграммным клиентам, подключающимся к конечным точкам UDS.

Дело в том, что датаграммный клиент, как и серверная программа, обязан привязаться к файлу сокета, чтобы получать направляемые ему датаграммы. Это не относится к датаграммным клиентам, которые используют сетевые сокеты. Отмечу, что клиент и сервер должны привязываться к разным файлам сокетов.

Главная причина этого различия состоит в том, что серверной программе нужен адрес, по которому можно вернуть ответ, и если датаграммный клиент не привязан к файлу сокета, то данный файл не будет иметь никакого отношения к конечной точке. Если же говорить о сетевых сокетах, то у клиента всегда есть соответствующий дескриптор, привязанный к IP-адресу и порту, и потому подобной проблемы не возникает.

```c
int main(int argc, char** argv) {
  char server_sock_file[] = "/tmp/calc_svc.sock";
  char client_sock_file[] = "/tmp/calc_cli.sock";
  // ----------- 1. Create socket object -----------------
  int conn_sd = socket(AF_UNIX, SOCK_DGRAM, 0);
  if (conn_sd == -1) {
    fprintf(stderr, "Could not create socket: %s\n", strerror(errno));
    exit(1);
  }
  // ----------- 2. Bind the client socket file ----------
  unlink(client_sock_file);
  struct sockaddr_un addr;
  memset(&addr, 0, sizeof(addr));
  addr.sun_family = AF_UNIX;
  strncpy(addr.sun_path, client_sock_file, sizeof(addr.sun_path) - 1);
  int result = bind(conn_sd, (struct sockaddr*)&addr, sizeof(addr));
  if (result == -1) {
    close(conn_sd);
    fprintf(stderr, "Could not bind the client address: %s\n", strerror(errno));
    exit(1);
  }
  // ----------- 3. Connect to server --------------------
  memset(&addr, 0, sizeof(addr));
  addr.sun_family = AF_UNIX;
  strncpy(addr.sun_path, server_sock_file, sizeof(addr.sun_path) - 1);
  result = connect(conn_sd, (struct sockaddr*)&addr, sizeof(addr));
  if (result == -1) {
    close(conn_sd);
    fprintf(stderr, "Could no connect: %s\n", strerror(errno));
    exit(1);
  }
  datagram_client_loop(conn_sd);
  return 0;
}
```

Сетевые сокеты

Еще одно широко используемое семейство адресов сокетов — AF_INET. К нему относятся любые каналы, создаваемые поверх сетевого соединения. В отличие от потоковых и датаграммных сокетов UDS, не имеющих никаких протоколов, для сетевых сокетов существует два общеизвестных протокола. TCP-сокеты создают потоковый канал между двумя процессами, а UDP-сокеты — датаграммный канал, который может применять любое количество процессов.

Как разрабатываются программы на основе TCP и UDP-сокетов?

TCP-сервер

Программа, использующая TCP-сокет для прослушивания и приема разных запросов (то есть TCP-сервер), отличается от потокового сервера, который прослушивает конечную точку UDS. Этих отличий два: во-первых, при вызове функции socket указывается другое семейство адресов, AF_INET вместо AF_UNIX, и, во-вторых, адрес, который она использует для привязки, имеет другую структуру.

В остальном, если говорить об операциях ввода/вывода, то TCP-сокет ведет себя так же, как и UDS. Следует отметить, что TCP-сокет является потоковым, поэтому для него должен подойти код, рассчитанный на потоковые сокеты домена Unix.

```
int main(int argc, char** argv) {
  // ----------- 1. Create socket object ------------------
  int server_sd = socket(AF_INET, SOCK_STREAM, 0);
  // ----------- 2. Bind the socket file ------------------
  // Prepare the address
...
  struct sockaddr_in addr;
  memset(&addr, 0, sizeof(addr));
  addr.sin_family = AF_INET;
  addr.sin_addr.s_addr = INADDR_ANY;
  addr.sin_port = htons(6666);
...
  // ----------- 3. Prepare backlog ------------------
  // ----------- 4. Start accepting clients ---------
}
```

TCP-клиент

Изменения очень похожи на те, которые мы видели в TCP-сервере. Здесь используется другое семейство адресов и другая структура для хранения адреса сокета.
Поскольку TCP-сокеты потоковые, тот же общий код позволяет обрабатывать новые клиентские запросы. Это можно видеть на примере функции stream_client_loop, которая является частью общей клиентской библиотеки в проекте «Калькулятор».

```
int main(int argc, char** argv) {
  // ----------- 1. Create socket object -----------------
  int conn_sd = socket(AF_INET, SOCK_STREAM, 0);
  // ------------ 2. Connect to server-- -----------------
  // Find the IP address behind the hostname
  // Prepare the address
...
  struct sockaddr_in addr;
  memset(&addr, 0, sizeof(addr));
  addr.sin_family = AF_INET;
  addr.sin_addr = *((struct in_addr*)host_entry->h_addr);;
  addr.sin_port = htons(6666);
  int result = connect(conn_sd, (struct sockaddr*)&addr, sizeof(addr));
...
}
```

UDP-сервер

UPD-сокеты являются сетевыми и датаграммными. Поэтому мы можем ожидать высокой степени сходства с кодом, написанным для TCP-сервера и для датаграммного сервера, который использовал UDS.
Кроме того, главное различие между UDP и TCP-сокетами, независимо от того, в какой программе они применяются: клиентской или серверной, состоит в том, что UDP-сокет имеет тип SOCK_DGRAM. Семейство адресов остается тем же, поскольку оба вида сокетов являются сетевыми. Главная функция UDP-сервера (server/udp/main.c)

```
int main(int argc, char** argv) {
  // ----------- 1. Create socket object ------------------
  int server_sd = socket(AF_INET, SOCK_DGRAM, 0);
  // ----------- 2. Bind the socket file ------------------
  // Prepare the address
...
struct sockaddr_in addr;
  memset(&addr, 0, sizeof(addr));
  addr.sin_family = AF_INET;
  addr.sin_addr.s_addr = INADDR_ANY;
  addr.sin_port = htons(9999);


...
  // ----------- 3. Prepare backlog ------------------
  // ----------- 4. Start accepting clients ---------
}
```

UDP-клиент

Код UDP и TCP-клиентов очень похож, однако они используют сокеты разных типов и различные функции для обработки входящих сообщений; в UDP-клиенте задействована функция из датаграммного клиента, основанного на UDS.

```
int main(int argc, char** argv) {
  // ----------- 1. Create socket object -----------------
  int conn_sd = socket(AF_INET, SOCK_DGRAM, 0);
  // ------------ 2. Connect to server-- -----------------
  // Find the IP address behind the hostname
  // Prepare the address
...
  struct sockaddr_in addr;
  memset(&addr, 0, sizeof(addr));
  addr.sin_family = AF_INET;
  addr.sin_addr = *((struct in_addr*)host_entry->h_addr);;
  addr.sin_port = htons(9999);
  int result = connect(conn_sd, (struct sockaddr*)&addr, sizeof(addr));
...
}
```