

20

Программирование сокетов

В предыдущей главе мы обсудили локальное межпроцессное взаимодействие и познакомились с концепцией программирования сокетов. Здесь же завершим наше введение и подробно рассмотрим тему программирования сокетов на примере реального клиент-серверного приложения: проекта «Калькулятор».

Порядок, в котором представлены темы, может показаться немного необычным, но это сделано для того, чтобы вы лучше ориентировались в различных типах сокетов и понимали, как они себя ведут в реальном проекте. В рамках данной главы мы:

- для начала подытожим материал, представленный в предыдущей главе. Это будет всего лишь краткое повторение пройденного материала. Вы обязательно должны прочитать вторую часть главы 19, посвященную программированию сокетов;
- в процессе повторения обсудим разные типы сокетов, датаграммные и потоковые последовательности, а также некоторые другие темы, которые понадобятся нам при рассмотрении примера с калькулятором;
- опишем и полностью проанализируем пример клиент-серверного проекта «Калькулятор». Это позволит нам перейти к обсуждению различных его компонентов и представить код на языке C;
- разработаем один из важнейших компонентов нашего примера — библиотеку сериализации/десериализации. Она реализует главный протокол взаимодействия клиентской и серверной частей калькулятора;
- необходимо понимать, что клиентская и серверная части калькулятора должны уметь взаимодействовать с помощью сокетов любых типов. Поэтому в нашем примере рассмотрим интеграцию с разного рода сокетами, начиная с *сокетов домена Unix* (Unix domain sockets, UDS);
- увидим в нашем примере, как с их помощью установить клиент-серверное соединение в рамках одного компьютера;
- вслед за этим рассмотрим сетевые сокеты; узнаем, как TCP- и UDP-сокеты можно интегрировать в проект «Калькулятор».

Для начала повторим все, что уже знаем о сокетах и программировании сокетов в целом. Прежде чем продолжать, крайне желательно ознакомиться со второй частью предыдущей главы. Я буду исходить из того, что вы уже обладаете необходимыми знаниями.

Краткий обзор программирования сокетов

В данном разделе мы еще раз поговорим о том, что такое сокеты, каких типов они бывают и что мы в целом понимаем под программированием сокетов. Это будет краткий обзор, но он станет основой для дальнейшего углубленного обсуждения в последующих разделах.

Если помните, в предыдущей главе мы говорили о наличии двух категорий межпроцессного взаимодействия, которые позволяют двум и более процессам общаться и обмениваться данными. К первой категории относятся *активные* (pull-based) методики, требующие наличия доступного *носителя* (такого как разделяемая память или обычный файл) для хранения и извлечения данных. Вторая категория охватывает *пассивные* (push-based) методики и подразумевает создание *канала*, доступного всем процессам, участвующим во взаимодействии. Главное различие между этими категориями связано с тем, как именно данные извлекаются из носителя (при активном подходе) или канала (при пассивном подходе).

Говоря простым языком, при использовании активных методик данные должны извлекаться или считываться с носителя, а в пассивном подходе данные доставляются читающему процессу автоматически. В первом случае процессы извлекают данные из разделяемого носителя, и если несколько из них могут туда записывать, то это чревато состояниями гонки.

Если быть более точным, то 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.

Мы уже знаем, что программирование сокетов — особая разновидность межпроцессного взаимодействия, относящаяся ко второй категории. Поэтому все методы IPC, основанные на сокетах, являются активными. Однако основная характеристика, которая отличает программирование сокетов от других активных методов IPC, заключается в использовании *сокетов*. Сокеты — это специальные объекты в Unix-подобных и других операционных системах, включая даже Microsoft Windows, представляющие *двунаправленные каналы*.

Иными словами, один объект сокета можно использовать для чтения и записи в один и тот же канал. Таким образом, два процесса, находящиеся на разных концах одного канала, могут участвовать в *двунаправленном взаимодействии*.

В предыдущей главе мы видели, что сокеты представлены дескрипторами по аналогии с тем, как файловые дескрипторы представляют файлы. Оба вида дескрипторов имеют некоторые общие аспекты, такие как операции ввода/вывода и возможность их *запрашивать*, однако на самом деле это разные вещи. Дескриптор сокета всегда представляет канал, а вот файловый дескриптор может представлять такие носители, как обычный файл или POSIX-канал. В связи с этим дескрипторы сокетов не поддерживают определенные операции с файлами, например `seek`; то же самое можно сказать даже о файловых дескрипторах, которые представляют канал.

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.

Таким образом, мы имеем два типа каналов: *потоковые* и *датаграммные*. Каждый потоковый канал в программе представлен *потоковым сокетом*, а каждый датаграммный — *датаграммным*. При подготовке канала мы должны сделать его либо потоковым, либо датаграммным. Чуть позже вы увидите, что наш пример с калькулятором поддерживает оба вида каналов.

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.

Для сравнения, сетевые сокеты можно применять в почти любой конфигурации, независимо от того, как именно развернуты процессы и где они находятся. Они могут размещаться на одном компьютере или быть распределены по сети. В случае с локальным развертыванием более предпочтительны сокеты UDS, поскольку они более быстрые и имеют меньше накладных расходов по сравнению с сетевыми сокетами. В рамках нашего примера с калькулятором мы реализуем поддержку как UDS, так и сетевых сокетов.

UDS и сетевые сокеты могут представлять как потоковые, так и датаграммные каналы. Следовательно, мы имеем четыре разные комбинации: UDS поверх потокового канала, UDS поверх датаграммного канала, сетевой сокет поверх потокового канала и, наконец, сетевой сокет поверх датаграммного канала. Все эти четыре варианта будут реализованы в нашем примере.

Сетевой сокет, предоставляющий потоковый канал, обычно работает по TCP. Дело в том, что TCP — самый распространенный транспортный протокол для этого вида сокетов. С другой стороны, сетевой сокет, предоставляющий датаграммный канал,

обычно работает по UDP. Это объясняется тем, что в большинстве случаев для такого рода сокетов используется транспортный протокол UDP. Обратите внимание: сокеты UDS, предоставляющие потоковые или датаграммные каналы, не имеют каких-то специальных названий, поскольку не применяют никаких транспортных протоколов.

Все указанные разновидности сокетов и каналов лучше всего показать на реальном примере. Вот, собственно, почему мы пошли таким необычным путем. Это позволит вам обратить внимание на общие аспекты разных типов сокетов и каналов и оформить их в виде блоков кода, пригодных к повторному использованию. В следующем разделе мы обсудим проект «Калькулятор» и его внутреннюю структуру.

Проект «Калькулятор»

Мы выделили целый раздел на то, чтобы объяснить назначение проекта «Калькулятор». Это масштабный пример, поэтому, прежде чем углубляться в него, будет полезно получить четкое понимание того, что он собой представляет. 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.

С учетом всего вышесказанного этот проект станет нашим главным примером в данной главе. Мы будем разрабатывать его шаг за шагом, и я проведу вас через различные этапы, кульминацией которых станет завершенное и рабочее приложение.

Для начала следует разработать относительно простой и полноценный прикладной протокол. Он будет использоваться для взаимодействия клиента и сервера. Как уже объяснялось ранее, две стороны не могут общаться между собой без четко определенного протокола прикладного уровня. Они могут быть соединены и передавать данные, поскольку эту возможность предоставляет программирование сокетов, но не будут понимать друг друга.

Вот почему мы должны уделить немного времени обсуждению прикладного протокола, который будет использоваться в проекте «Калькулятор». Но прежде, чем переходить к самому протоколу, рассмотрим иерархию исходного кода, которую можно увидеть в кодовой базе проекта. Это существенно облегчит поиск прикладного протокола и библиотеки сериализации/десериализации.

Иерархия исходного кода

С точки зрения программиста, API для программирования POSIX-сокетов обращается со всеми потоковыми каналами одинаково, независимо от того, что лежит в их основе: UDS или сетевой сокет. Как вы можете помнить из материалов предыдущей главы, для потоковых каналов предусмотрены определенные последовательности шагов, которые выполняются на стороне слушателя и соединителя, и эти последовательности не зависят от типа сокетов.

Таким образом, если вы собираетесь поддерживать разные типы сокетов в сочетании с разными типами каналов, то лучше определить их общие аспекты и реализовать их отдельно, чтобы не повторяться. Именно такой подход мы станем применять в проекте «Калькулятор», и он отражен в исходном коде. Поэтому вам будут встречаться различные библиотеки, и некоторые из них будут содержать общий код, повторно использующийся разными компонентами.

Теперь пришло время взяться за кодовую базу. Прежде всего, исходный код проекта находится по адресу <https://github.com/PacktPublishing/Extreme-C/tree/master/ch20-socket-programming>. Если пройти по данной ссылке и взглянуть на код, то можно увидеть ряд каталогов с исходными файлами. Очевидно, что анализ каждого файла занял бы слишком много времени, поэтому мы сосредоточимся на важных участках кода. Вы можете сами пройтись по кодовой базе, собрать ее и запустить полученную программу; благодаря этому вы получите общее представление о том, как разрабатывался наш пример.

Отмечу, что весь код, относящийся к сокетам UDS, UDP и TCP, находится в одном каталоге. Далее мы рассмотрим иерархию кодовой базы.

Если перейти в корень проекта и выполнить команду `tree`, то можно увидеть дерево файлов и каталогов, представленное в терминале 20.1. В нем же показано, как клонировать репозиторий этой книги на GitHub и перейти в корневой каталог данного примера.

Терминал 20.1. Клонирование кодовой базы проекта «Калькулятор» и вывод его файлов и каталогов

```
$ git clone https://github.com/PacktPublishing/Extreme-C
Cloning into 'Extreme-C'...
...
```

```
Resolving deltas: 100% (458/458), done.
$ cd Extreme-C/ch20-socket-programming
$ tree
.
├── CMakeLists.txt
├── calcser
...
├── calcsvc
...
├── client
│   ├── CMakeLists.txt
│   └── clicore
...
├── tcp
│   ├── CMakeLists.txt
│   └── main.c
├── udp
│   ├── CMakeLists.txt
│   └── main.c
└── Unix
    ├── CMakeLists.txt
    ├── datagram
    │   ├── CMakeLists.txt
    │   └── main.c
    └── stream
        ├── CMakeLists.txt
        └── main.c
└── server
    ├── CMakeLists.txt
    └── srvcore
...
├── tcp
│   ├── CMakeLists.txt
│   └── main.c
├── udp
│   ├── CMakeLists.txt
│   └── main.c
└── Unix
    ├── CMakeLists.txt
    ├── datagram
    │   ├── CMakeLists.txt
    │   └── main.c
    └── stream
        ├── CMakeLists.txt
        └── main.c
└── types.h

18 directories, 49 files
$
```

Как можно видеть в этом списке файлов и каталогов, наш проект состоит из ряда компонентов, в том числе и библиотек. Все компоненты находятся в отдельных каталогах, каждый из которых описан ниже.

- Библиотека сериализации/десериализации в каталоге `/calcser` содержит соответствующие исходные файлы. 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*. Это не то же самое, что серверный процесс. Данный сервис предоставляет основные функции калькулятора и не привязан к серверному процессу, и потому его можно использовать отдельно в качестве самостоятельной библиотеки С. В результате сборки этого каталога получается статическая библиотека `libcalcsvc.a`.
- Библиотека в каталоге `/server/srvcore` содержит исходники, общие для потоковых и датаграммных процессов, независимо от типа сокета. Поэтому ее могут использовать все серверные процессы калькулятора, включая те, которые основаны на UDS и сетевых сокетах и работают с потоковыми и датаграммными каналами. Итоговым результатом сборки этого каталога будет статическая библиотека `libsrvcore.a`.
- Каталог `/server/unix/stream` содержит исходники серверной программы, которая использует потоковые каналы внутри сокетов UDS. Она будет собрана в исполняемый файл `unix_stream_calc_server`. Это одна из нескольких программ, которые можно применять в качестве серверной части калькулятора. Этот конкретный сервер прослушивает сокет UDS для установления потоковых соединений.
- Каталог `/server/unix/datagram` содержит исходники серверной программы, которая использует датаграммные каналы внутри сокетов UDS. Она будет собрана в исполняемый файл `unix_datagram_calc_server`. Это одна из нескольких программ, которые можно задействовать в качестве серверной части калькулятора. Этот конкретный сервер прослушивает сокет UDS для приема датаграммных сообщений.
- Каталог `/server/tcp` содержит исходники серверной программы, которая использует потоковые каналы внутри сетевых сокетов TCP. Она будет собрана в исполняемый файл `tcp_calc_server`. Это одна из нескольких программ, которые можно применять в качестве серверной части калькулятора. Этот конкретный сервер прослушивает сокет TCP для установления потоковых соединений.
- Каталог `/server/udp` содержит исходники серверной программы, которая использует датаграммные каналы внутри сетевых сокетов UDP. Она будет собрана в исполняемый файл `udp_calc_server`. Это одна из нескольких программ,

которые можно задействовать в качестве серверной части калькулятора. Этот конкретный сервер прослушивает сокет UDP для приема датаграммных сообщений.

- Библиотека в каталоге `/client/clicore` содержит исходники, общие для потоковых и датаграммных клиентских процессов, независимо от типа сокета. Поэтому ее могут использовать все клиентские процессы калькулятора, включая те, которые основаны на UDS и сетевых сокетах и работают с потоковыми и датаграммными каналами. Итоговым результатом сборки этого каталога будет статическая библиотека `libclicore.a`.
- Каталог `/client/unix/stream` содержит исходники клиентской программы, которая использует потоковые каналы внутри сокетов UDS. Она будет собрана в исполняемый файл `unix_stream_calc_client`. Это одна из нескольких программ, которые можно применять для запуска клиентской части калькулятора. Этот конкретный клиент подключается к конечной точке UDS и устанавливает потоковое соединение.
- Каталог `/client/unix/datagram` содержит исходники клиентской программы, которая использует датаграммные каналы внутри сокетов UDS. Она будет собрана в исполняемый файл `unix_datagram_calc_client`. Это одна из нескольких программ, которые можно задействовать в целях запуска клиентской части калькулятора. Этот конкретный клиент подключается к конечной точке UDS и отправляет датаграммные сообщения.
- Каталог `/client/tcp` содержит исходники клиентской программы, которая использует потоковые каналы внутри сокетов TCP. Она будет собрана в исполняемый файл `tcp_calc_client`. Это одна из нескольких программ, которые можно применять для запуска клиентской части калькулятора. Этот конкретный клиент подключается к конечной точке TCP-сокета и устанавливает потоковое соединение.
- Каталог `/client/udp` содержит исходники клиентской программы, которая использует датаграммные каналы внутри сокетов UDP. Она будет собрана в исполняемый файл `udp_calc_client`. Это одна из нескольких программ, которые можно задействовать для запуска клиентской части калькулятора. Этот конкретный клиент подключается к конечной точке UDP-сокета и отправляет датаграммные сообщения.

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.

Чтобы собрать проект, выполните следующие команды в корневом каталоге главы (терминал 20.2).

Терминал 20.2. Программы для сборки проекта «Калькулятор»

```
$ mkdir -p build  
$ cd build  
$ cmake ..  
...  
$ 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. Соответствующая команда приводится ниже (терминал 20.3).

Обратите внимание: перед вводом этой команды вы должны перейти в каталог `build`, который был создан в рамках предыдущего раздела.

Терминал 20.3. Запуск потокового сервера, который прослушивает сокет UDS

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

Убедитесь в том, что сервер работает. Запустите во втором терминале потоковый клиент, собранный для использования UDS (терминал 20.4).

Терминал 20.4. 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.  
$
```

Как видите, у клиентского процесса есть собственная командная строка. Она принимает команды от пользователя, превращает их в запросы, соответствующие прикладному протоколу, и отправляет их серверу для дальнейшей обработки. Затем клиент ждет ответа и сразу после того, как тот будет получен, выводит результат. Отмечу, что данная командная строка является частью общего кода, написанного для всех клиентов, поэтому вы всегда сможете работать с ней, независимо от типа канала или сокета.

Теперь пришло время углубиться в детали прикладного протокола и посмотреть на то, как выглядят запросы и ответы.

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. В добавок следует сказать, что протокол будет текстовым. То есть запросы и ответы могут состоять только из алфавитно-цифровых и нескольких других символов. Это позволит сделать со-общения 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. В листинге 20.1 показан заголовочный файл `calcser/calc_proto_req.h`, который описывает структуру запроса в нашем протоколе.

Листинг 20.1. 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

```

Как видите, в рамках нашего протокола определено девять методов. Любой приличный калькулятор должен иметь внутреннюю память, и потому у нас есть операции с памятью, относящиеся к сложению, вычитанию и умножению.

Например, метод ADD просто складывает два числа с плавающей запятой, а ADDM — его разновидность, которая прибавляет к этим двум числам значение, хранящееся во внутренней памяти, и заносит результат в ту же память для дальнейшего использования. Это аналогично применению кнопки памяти в настольном калькуляторе; вы можете найти ее по надписи +M.

У нас также есть специальный метод для чтения и сброса внутренней памяти калькулятора. Операцию деления с внутренней памятью выполнять нельзя, поэтому других вариаций не предусмотрено.

Представьте, что клиент хочет создать запрос с ID 1000, методом ADD и двумя operandами: 1.5 и 5.6. В языке C для этого нужно создать объект calc_proto_req_t (данная структура объявлена в предыдущем заголовке в листинге 20.1) и заполнить его нужными значениями. В листинге 20.2 показано, как это делается.

Листинг 20.2. 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;

```

Как уже объяснялось в предыдущей главе, 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 (листинг 20.3).

Листинг 20.3. Сериализованное сообщение, эквивалентное объекту `req`, который был определен в листинге 20.2

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. *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. Вы можете видеть его в листинге 20.4.

Листинг 20.4. Определение объекта ответа (`calcser/calc_proto_resp.h`)

```
#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
```

По аналогии с объектом запроса из листинга 20.2, `req`, серверный процесс создает *объект ответа*, выполняя следующие инструкции (листинг 20.5).

Листинг 20.5. Создание объекта ответа для объекта `req`, который был определен в листинге 20.2

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

Результат сериализации этого объекта выглядит так (листинг 20.6).

Листинг 20.6. Сериализованное ответное сообщение, эквивалентное объекту `resp`, который был создан в листинге 20.5

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

И снова мы используем символы # для разделения полей и \$ для разделения сообщений. Обратите внимание: поле `status` является числовым и сигнализирует об успешном или неуспешном запросе. В случае неудачи оно имеет ненулевое значение, описанное в заголовочном файле ответа (или, точнее, в протоколе калькулятора).

Теперь более подробно поговорим о библиотеке сериализации/десериализации и посмотрим, как она выглядит внутри.

Библиотека сериализации/десериализации

В предыдущем подразделе было показано, как выглядят сообщения запроса и ответа. 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.

Я уже упоминал о том, что эти возможности предоставляются другим частям проекта в виде статической библиотеки `libcalcser.a`. В листинге 20.7 вы можете видеть публичный API класса `serializer`, который находится в файле `calcser/calc_proto_ser.h`.

Листинг 20.7. Публичный интерфейс класса `serializer` (`calcser/calc_proto_ser.h`)

```
#ifndef CALC_PROTO_SER_H
#define CALC_PROTO_SER_H

#include <types.h>

#include "calc_proto_req.h"
#include "calc_proto_resp.h"

#define ERROR_INVALID_REQUEST          101
#define ERROR_INVALID_REQUEST_ID       102
#define ERROR_INVALID_REQUEST_METHOD   103
#define ERROR_INVALID_REQUEST_OPERAND1 104
#define ERROR_INVALID_REQUEST_OPERAND2 105

#define ERROR_INVALID_RESPONSE         201
#define ERROR_INVALID_RESPONSE_REQ_ID 202
```

```
#define ERROR_INVALID_RESPONSE_STATUS 203
#define ERROR_INVALID_RESPONSE_RESULT 204

#define ERROR_UNKNOWN 220

struct buffer_t {
    char* data;
    int len;
};

struct calc_proto_ser_t;

typedef void (*req_cb_t)(
    void* owner_obj,
    struct calc_proto_req_t);

typedef void (*resp_cb_t)(
    void* owner_obj,
    struct calc_proto_resp_t);

typedef void (*error_cb_t)(
    void* owner_obj,
    const int req_id,
    const int error_code);

struct calc_proto_ser_t* calc_proto_ser_new();
void calc_proto_ser_delete(
    struct calc_proto_ser_t* ser);

void calc_proto_ser_ctor(
    struct calc_proto_ser_t* ser,
    void* owner_obj,
    int ring_buffer_size);

void calc_proto_ser_dtor(
    struct calc_proto_ser_t* ser);

void* calc_proto_ser_get_context(
    struct calc_proto_ser_t* ser);

void calc_proto_ser_set_req_callback(
    struct calc_proto_ser_t* ser,
    req_cb_t cb);

void calc_proto_ser_set_resp_callback(
    struct calc_proto_ser_t* ser,
    resp_cb_t cb);
void calc_proto_ser_set_error_callback(
    struct calc_proto_ser_t* ser,
    error_cb_t cb);

void calc_proto_ser_server_deserialize(
    struct calc_proto_ser_t* ser,
```

```

    struct buffer_t buffer,
    bool_t* req_found);

struct buffer_t calc_proto_ser_server_serialize(
    struct calc_proto_ser_t* ser,
    const struct calc_proto_resp_t* resp);

void calc_proto_ser_client_deserialize(
    struct calc_proto_ser_t* ser,
    struct buffer_t buffer,
    bool_t* resp_found);

struct buffer_t calc_proto_ser_client_serialize(
    struct calc_proto_ser_t* ser,
    const struct calc_proto_req_t* req);

#endif

```

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.

Теперь поближе рассмотрим функции сериализации/десериализации для серверной стороны.

Функции сериализации/десериализации для серверной стороны

Для серверного процесса предусмотрено две функции: одна сериализует объект ответа, а другая десериализует сообщение с запросом. Начнем с функции сериализации.

В листинге 20.8 представлен код функции `calc_proto_ser_server_serialize`, которая сериализует ответ.

Листинг 20.8. 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;
}
```

Элемент `resp` — это указатель на объект ответа, который нужно сериализовать. Функция возвращает объект `buffer_t`, который объявлен в заголовочном файле `calc_proto_ser.h` и выглядит так (листинг 20.9).

Листинг 20.9. Определение объекта `buffer_t` (calcser/calc_proto_ser.h)

```
struct buffer_t {
    char* data;
    int len;
};
```

Сериализатор имеет простой код, основная часть которого — инструкция `sprintf`, предназначенная для создания строки с ответным сообщением. Теперь взглянем на функцию десериализации запроса. Десериализатор обычно сложнее реализовать, и если найти в кодовой базе вызовы следующих функций, то можно увидеть, насколько сложными они бывают.

В листинге 20.10 показана функция для десериализации запроса.

Листинг 20.10. 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);
}
```

Приведенная выше функция выглядит просто, однако на самом деле использует приватные методы `_deserialize` и `_parse_req_and_notify`, которые определены в файле `calc_proto_ser.c`, где содержится сама реализация класса `Serializer`.

Мы не станем углубляться в код упомянутых приватных методов, поскольку это уже выходит за рамки данной книги. Но чтобы вы имели общее представление, особенно если вам хочется почитать исходный код, отмечу: десериализатор использует *кольцевой буфер* фиксированной длины и пытается найти символ `$`, который служит разделителем сообщений.

При нахождении `$` десериализатор применяет указатель, который в нашем случае ссылается на функцию `_parse_req_and_notify` (третий аргумент, переданный функции `_deserialize`). Она пытается извлечь поля и воссоздать объект запроса. Затем использует функции обратного вызова, отправляя уведомления зарегестрированному *наблюдателю*, роль которого в данном случае исполняет объект сервера, а тот уже приступает к обработке запроса.

Теперь рассмотрим функции, применяемые на стороне клиента.

Функции сериализации/десериализации для клиентской стороны

Как и в случае с серверной стороной, для стороны клиента предусмотрено две функции: одна для сериализации объекта запроса, а другая для сериализации входящего ответа.

Начнем с сериализатора запроса. Его определение можно видеть в листинге 20.11.

Листинг 20.11. 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", 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;
}
```

Данная функция принимает объект запроса и возвращает объект `buffer`; это очень похоже на сериализацию ответа на серверной стороне. Здесь даже применяется тот же подход: использование инструкции `sprintf` для создания сообщения с запросом.

В листинге 20.12 показана функция десериализации ответа.

Листинг 20.12. 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);
}
```

Здесь применяется тот же механизм и задействуются похожие приватные методы. Я настоятельно рекомендую должностным образом прочесть эти исходники, чтобы лучше понять, как разные части кода были собраны вместе и максимально хорошо подготовлены для повторного использования существующих компонентов.

Мы не станем углубляться в класс `Serializer`; можете сами пройтись по коду и посмотреть, как он работает.

Итак, у нас есть библиотека сериализации. Теперь мы можем написать клиентскую и серверную программы. Разработка библиотеки, которая сериализует и десериализует сообщения в соответствии с согласованным прикладным протоколом, — обязательный шаг при написании многопроцессного программного обеспечения. Заметьте, что это не зависит от того, как развернута ваша система: на одном или нескольких компьютерах. Процессы должны понимать друг друга, и для этого должны быть определены подходящие протоколы прикладного уровня.

Прежде чем переходить к коду, относящемуся к программированию сокетов, необходимо объяснить еще кое-что: сервис калькулятора. Он лежит в основе серверного процесса и выполняет сами вычисления.

Сервис калькулятора

Сервис калькулятора — основная логика нашего примера. Стоит отметить, что он должен работать независимо от того или иного механизма межпроцессного взаимодействия. В листинге 20.13 показано объявление его класса.

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.

Листинг 20.13. Публичный интерфейс класса, принадлежащего сервису калькулятора (`calcsvc/calc_service.h`)

```
#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
```

Как видите, в этом классе даже предусмотрены отдельные типы ошибок. Входные аргументы имеют стандартные типы языка C, которые никоим образом не зависят от классов или структур, связанных с сериализацией. Поскольку это изолированная и самостоятельная логика, мы компилируем ее в виде отдельной статической библиотеки `libcalcsvc.a`.

Каждый серверный процесс должен использовать объекты данного сервиса для выполнения вычислений. Эти объекты обычно называются *сервисными* или *служебными*. И потому итоговая серверная программа должна быть скомпонована с данной библиотекой.

Прежде чем двигаться дальше, необходимо сделать следующее замечание: если клиенту не требуется определенный контекст, то мы можем ограничиться одним служебным объектом. То есть если сервис на клиентской стороне не требует от нас сохранения какого-либо состояния из предыдущих запросов данного клиента, то служебный объект можно сделать *синглтоном*. Это значит, что у объекта *нет состояния*.

И наоборот, если для обработки текущего запроса нужна какая-либо информация о предыдущих запросах, то в каждом клиенте необходимо использовать отдельный служебный объект. Именно это происходит в нашем проекте. Как вы уже знаете, у калькулятора есть внутренняя память, уникальная для каждого клиента. Поэтому мы не можем задействовать один и тот же объект в разных клиентах. Это значит, у объекта *есть состояние*.

Если подытожить вышесказанное, то для каждого клиента нам нужно создавать новый объект сервиса. Благодаря этому у каждого клиента будет свой калькулятор с отдельной внутренней памятью. Служебные объекты калькулятора имеют состояние (значение во внутренней памяти), которое им нужно загружать.

Теперь мы готовы перейти к обсуждению различных типов сокетов с примерами в контексте проекта «Калькулятор».

Сокеты домена Unix

Мы уже знаем по предыдущей главе, при установлении соединения между двумя процессами, размещенными на одном компьютере, одним из лучших решений являются сокеты домена Unix (Unix domain sockets, UDS). В данной главе мы продолжили наше обсуждение и поговорили чуть более подробно о пассивных методах межпроцессного взаимодействия, а также о потоковых и датаграммных каналах. 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, представленных нами в предыдущей главе. Для начала поговорим о слушающем процессе, работающем с потоковым каналом. Это будет *потоковый сервер*.

Потоковый сервер на основе UDS

Как вы помните, в предыдущей главе мы рассматривали разные последовательности действий, которые выполняют слушатель и соединитель, взаимодействуя с помощью транспортного канала. 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.

Следующий фрагмент кода находится в главной функции серверной программы. Как видно в листинге 20.14, процесс сначала создает объект `socket`.

Листинг 20.14. 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.

В предыдущей главе мы уже объясняли, что 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.

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. Листинг 20.15 используется в проекте «Калькулятор» в целях привязки объекта сокета к файлу, расположенному по заранее известному пути. Этот путь указан в виде массива символов `sock_file`.

Листинг 20.15. 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.



В Linux UDS можно привязать к абстрактному адресу сокета. Это в основном полезно в ситуациях, когда для размещения файла сокета нельзя задействовать файловую систему. В приведенном выше листинге в целях инициализации структуры адреса, `addr`, можно применять строку, начинаяющуюся с нулевого символа, `\0`, в результате чего предоставленное имя будет привязано к объекту сокета внутри ядра. Данное имя должно быть уникальным в рамках всей системы, и к нему не должен быть привязан никакой другой объект.

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 приступить к следующему шагу в последовательности потокового слушателя: настройке размера *очереди отставания*.

В коде, представленном в листинге 20.16, показана процедура настройки очереди отставания для потокового сервера, прослушивающего сокет UDS.

Листинг 20.16. 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. Как объяснялось в предыдущей главе, 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. Это важный этап подготовки потокового сокета перед приемом клиентских запросов.

Согласно последовательности действий потокового слушателя, вслед за привязкой потокового сокета и настройки размера его очереди отставания мы можем приступить к приему новых клиентских запросов. В листинге 20.17 показано, как это делается.

Листинг 20.17. 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,зывающий функцию `accept`, блокируется на ней.

Теперь пришло время собрать все перечисленные выше шаги в единое целое. В листинге 20.18 показан потоковый сервер из проекта «Калькулятор», который прослушивает сокет UDS.

Листинг 20.18. Главная функция потокового сервиса калькулятора, прослушивающая конечную точку UDS (server/unix/stream/main.c)

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

#include <sys/socket.h>
#include <sys/un.h>

#include <stream_server_core.h>

int main(int argc, char** argv) {
    char sock_file[] = "/tmp/calc_svc.sock";
```

```
// ----- 1. Создаем объект сокета -----
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);
}

// ----- 2. Привязываем файл сокета -----
// Удаляем ранее созданный файл сокета, если таковой имеется
unlink(sock_file);

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

// ----- 4. Начинаем принимать клиентов -----
accept_forever(server_sd);

return 0;
}
```

Найти блоки кода, ответственные за выполнение вышеупомянутых действий по инициализации серверного сокета, должно быть довольно легко. Здесь не хватает лишь кода, который принимает запросы от клиентов. Он вынесен в отдельную функцию с именем `accept_forever`. Обратите внимание: данная функция является блокирующей, поэтому главный поток будет заблокирован до завершения работы сервера.

В листинге 20.19 показано определение функции `accept_forever`. Она является частью общей серверной библиотеки, которая находится в каталоге `srvcore`.

Подобное размещение объясняется тем, что ее определение используется другими потоковыми сокетами, включая работающие по TCP. Таким образом, мы можем повторно задействовать имеющуюся логику, вместо того чтобы создавать ее заново.

Листинг 20.19. Функция, принимающая новые клиентские запросы на потоковом сокете, который прослушивает конечную точку UDS (server/srvcore/stream_server_core.c)

```
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);
        }
    }
}
```

В данном листинге видно, что в момент приема нового клиентского запроса мы создаем новый поток выполнения, который отвечает за работу с соединением. Это фактически сводится к чтению байтов из клиентского канала, передаче прочитанных байт десериализатору и генерации подходящих ответов при обнаружении запроса.

Создание нового потока выполнения для каждого клиента — стандартный подход, который обычно используется серверными процессами, работающими с блокирующим потоковым каналом, независимо от типа сокета. Поэтому в таких ситуациях важнейшую роль играет многопоточность и все связанные с ней темы.



Что касается неблокирующих потоковых каналов, то для них обычно используется другой подход, известный как цикл событий.

Полученный объект сокета можно использовать для чтения и записи на клиентской стороне. Если следовать принципу, по которому разрабатывалась библиотека

`srvcore`, то следующим шагом будет анализ функции-компаньона в клиентском потоке, `client_handler`. В исходных текстах эту функцию можно найти сразу за `accept_forever`. В листинге 20.20 показано ее определение.

Листинг 20.20. Функция-компаньон потока выполнения, который работает с клиентом (server/srvcore/stream_server_core.c)

```
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;
}
```

У этого кода есть много разных аспектов, но среди них я хочу выделить несколько особенно важных. Как вы сами можете видеть, для чтения блоков информации, передаваемой клиентом, используется функция `read`. Если помните, данная функция

принимает файловый дескриптор, однако здесь ей передается дескриптор сокета. Это демонстрация того, что, несмотря на разницу этих двух видов дескрипторов, для работы с ними можно использовать одни и те же функции ввода/вывода.

В этом коде мы читаем блоки байтов из ввода и передаем их десериализатору, вызывая функцию `calc_proto_ser_server_deserialize`. Прежде чем ответ полностью десериализуется, данную функцию, возможно, придется вызывать три или четыре раза. Это во многом зависит от размера блоков, которые вы читаете из ввода, и длины сообщений, передаваемых по каналу.

Вдобавок следует отметить: у каждого клиента есть свой объект-сериализатор. То же самое касается объекта сервиса. Эти объекты создаются и уничтожаются вместе со своим потоком выполнения.

Заключительным аспектом, на который стоит обратить внимание, является то, что ответы клиенту возвращаются с помощью функции `stream_write_response`, предназначеннной для работы с потоковым сокетом. Эту функцию можно найти в том же файле, что и код из предыдущих листингов. В листинге 20.21 представлено ее определение.

Листинг 20.21. 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);
    }
}
```

В данном листинге видно, что для записи сообщения, которое возвращается клиенту, мы используем функцию `write`. Как мы уже знаем, она может принимать файловые дескрипторы, но, по всей видимости, ей можно передавать и дескрипторы

сокетов. Это наглядная демонстрация того, что API ввода/вывода POSIX совместим с обоими видами дескрипторов.

То же самое относится и к функции `close`. Мы использовали ее для разрыва соединения. Как известно, она умеет работать с файловыми дескрипторами, поэтому ей смело можно передавать дескрипторы сокетов.

Итак, мы прошлись по некоторым важнейшим аспектам потокового сервера UDS и получили общее представление о том, как он работает. Теперь пришло время перейти к обсуждению потокового клиента UDS. Конечно, многие участки кода остались без внимания, но вы можете проанализировать их самостоятельно.

ПОТОКОВЫЙ КЛИЕНТ НА ОСНОВЕ UDS

Как и серверная программа, описанная в предыдущем подразделе, клиент должен первым делом создать объект сокета. Вы помните, что 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. Дальше нам нужно подключиться к процессу сервера, указав конечную точку UDS, аналогично тому, как это сделал сервер. После создания потокового канала клиентский процесс может читать и записывать в него с помощью открытого дескриптора сокета.

В листинге 20.22 показана функция `main` потокового клиента, который соединяется с конечной точкой UDS.

Листинг 20.22. 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. Создаем объект сокета -----

    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. Подключаемся к серверу -----

    // Подготавливаем адрес
    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;
}

```

Первая часть этого листинга очень похожа на код сервера, но дальше вместо `bind` клиент вызывает `connect`. Обратите внимание: код подготовки адреса ничем не отличается от того, который используется сервером.

Успешное возвращение вызова `connect` означает, что дескриптор сокета `conn_sd` был привязан к открытому каналу. С этого момента `conn_sd` можно использовать для взаимодействия с сервером. Мы передаем данный дескриптор функции `stream_client_loop`, которая выводит командную строку клиента и выполняет все остальные действия, которые будут инициированы клиентской стороной. Это блокирующая функция, выполняющаяся, пока клиент не завершит работу.

Клиент также использует функции `read` и `write` для передачи и получения сообщений от сервера. Листинг 20.23 содержит определение функции `stream_client_loop`, которая входит в состав общей клиентской библиотеки и используется всеми потоковыми клиентами, независимо от типа сокета — UDS или TCP. Как видите, она вызывает функцию `write` для отправки серверу сообщения с сериализованным запросом.

Листинг 20.23. Функция, выполняющая потоковый клиент (`client/clicore/stream_client_core.c`)

```

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) {

```

```

scanf("%s", buf);
int brk = 0, cnt = 0;
struct calc_proto_req_t req;
parse_client_input(buf, &req, &brk, &cnt);
if (brk) {
    break;
}
if (cnt) {
    continue;
}
struct buffer_t ser_req =
    calc_proto_ser_client_serialize(context.ser, &req);
int ret = write(context.sd, ser_req.data, ser_req.len);
if (ret == -1) {
    fprintf(stderr, "Error while writing! %s\n",
            strerror(errno));
    break;
}
if (ret < ser_req.len) {
    fprintf(stderr, "Wrote less than anticipated!\n");
    break;
}
printf("The req(%d) is sent.\n", req.id);
}
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");
}

```

Данный код демонстрирует, что все клиентские процессы имеют один общий объект-сериализатор, и это логично. Для сравнения, на серверной стороне у каждого клиента был свой сериализатор.

Более того, клиентский процесс создает новый поток выполнения для чтения ответов, которые присыпает сервер. Это вызвано тем, что чтение из серверного процесса — блокирующая операция, поэтому ее следует выполнять в отдельном потоке.

В рамках главного потока мы выводим командную строку клиента, которая принимает пользовательский ввод через терминал. Во время завершения работы главный поток присоединяет поток чтения и ждет, когда тот завершится.

В данном листинге также следует обратить внимание на то, что клиентский процесс использует тот же API ввода/вывода для чтения и записи в потоковый канал. Как уже говорилось ранее, для этого предусмотрены функции `read` и `write`, и пример их использования показан в листинге 20.23.

В следующем подразделе мы поговорим о датаграммных каналах, но с применением все тех же сокетов UDS. Начнем с датаграммного сервера.

Датаграммный сервер на основе UDS

Возможно, из предыдущей главы вы помните, что датаграммные процессы, слушатель и соединитель, имеют собственные последовательности действий по передаче данных. Пришло время показать, как может выглядеть датаграммный сервер на основе UDS.

Согласно последовательности датаграммного слушателя, процесс должен сначала создать объект сокета. Это показано в листинге 20.24.

Листинг 20.24. 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, и потому я не стану его здесь приводить. Можете взглянуть на него в листинге 20.15.

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.

В листинге 20.25 показана функция `main` датаграммного сервера, который прослушивает конечную точку UDS в рамках проекта «Калькулятор».

Листинг 20.25. Главная функция датаграммного сервера, который прослушивает конечную точку UDS (server/unix/datagram/main.c)

```
int main(int argc, char** argv) {
    char sock_file[] = "/tmp/calc_svc.sock";

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

```
        strerror(errno));
    exit(1);
}

// ----- 2. Привязываем файл сокета -----

// Удаляем ранее созданный файл сокета, если таковой существует
unlink(sock_file);

// Подготавливаем адрес
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. Начинаем обслуживать запросы -----
serve_forever(server_sd);

return 0;
}
```

Вы уже знаете, что 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, доступная по следующей ссылке, более подробно объясняет, как выбрать корректный размер: <https://stackoverflow.com/questions/21856517/whats-the-practical-limit-on-the-size-of-single-packet-transmitted-over-domain>.

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. Упомянутые выше функции способны запомнить нужный процесс и отправить ему датаграмму.

Ниже вы можете видеть определение функции `serve_forever`, которая использовалась в конце функции `main` в листинге 20.25. Она входит в состав общей серверной библиотеки и предназначена для датаграммных серверов, независимо от типа сокета. В листинге 20.26 наглядно показано, как работает операция `recvfrom`.

Листинг 20.26. Функция для обработки датаграмм, принадлежащая общей серверной библиотеке и предназначенная для датаграммных серверов
(server/srvcore/datagram_server_core.c)

```
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);
        if (read_nr_bytes == -1) {
            close(server_sd);
            fprintf(stderr, "Could not read from datagram socket: %s\n",
                strerror(errno));
            exit(1);
        }
        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);

if (!req_found) {
    struct calc_proto_resp_t resp;
    resp.req_id = -1;
    resp.status = ERROR_INVALID_RESPONSE;
    resp.result = 0.0;
    context.write_resp(&context, &resp);
}

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->sockaddr);
free(context.addr);
}
}

```

Как показано в этом листинге, датаграммный сервер представляет собой программу с одним потоком выполнения и без какой-либо многопоточности. Более того, данная программа работает с каждой датаграммой отдельно и независимо. Она получает датаграмму, десериализует ее содержимое, создает объект запроса, обрабатывает запрос с помощью объекта сервиса, сериализует объект ответа, помещает его в новую датаграмму и отправляет процессу, который послал исходный запрос. Эта процедура повторяется по кругу снова и снова для каждой входящей датаграммы.

Отмечу: у каждой датаграммы есть собственные объект-сериализатор и объект сервиса. Мы могли бы спроектировать приложение так, чтобы для всех датаграмм использовались одни и те же сериализатор и сервис. Подумайте, благодаря чему это возможно и почему такой подход может оказаться несовместимым с проектом «Калькулятор». Это спорное решение, и у разных людей могут быть разные мнения на сей счет.

Обратите внимание: в листинге 20.26 при получении датаграммы мы сохраняем ее клиентский адрес. Позже данный адрес можно использовать для записи напрямую в клиентский процесс. Вам стоит ознакомиться с тем, как датаграмма возвращается исходному клиенту. Как и в случае с потоковым сервером, мы используем для этого функцию. В листинге 20.27 показано определение функции `datagram_write_resp`, которая находится в общей библиотеке датаграммного сервера сразу за функцией `serve_forever`.

Листинг 20.27. Функция, возвращающая датаграммы клиентам
(server/srvcore/datagram_server_core.c)

```
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);
    }
}
```

Как вы можете видеть, мы берем адрес клиента и передаем его функции `sendto` вместе с сериализованным ответом. Все остальное делает за нас операционная система, в результате чего датаграмма возвращается клиенту, который послал запрос.

Мы уже имеем достаточно хорошее представление о датаграммном сервере и о том, как следует использовать сокеты. Теперь обратим внимание на датаграммный клиент, который основан на сокетах того же типа.

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. Но, как вы вскоре увидите, 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.

Если не считать этих различий, то код в целом выглядит довольно похоже. В листинге 20.28 вы можете видеть функцию `main` датаграммного клиента.

Листинг 20.28. Функция, возвращающая датаграммы клиентам
(server/srvcore/datagram_server_core.)

```
int main(int argc, char** argv) {
    char server_sock_file[] = "/tmp/calc_svc.sock";
    char client_sock_file[] = "/tmp/calc_cli.sock";

    // ----- 1. Создаем объект сокета -----

    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. Привязываем файл клиентского сокета -----
    // Удаляем ранее созданный файл сокета, если таковой существует
    unlink(client_sock_file);

    // Подготавливаем клиентский адрес
    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. Подключаемся к серверу -----  
  
// Подготавливаем серверный адрес  
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;  
}
```

Как уже объяснялось ранее и как мы можем видеть в данном листинге, клиент обязан привязаться к файлу сокета. И конечно, чтобы начать клиентский цикл, в конце `main` следует вызвать другую функцию. В данном случае это `datagram_client_loop`. В ней по-прежнему много общего между потоковым и датаграммным клиентами. Основное различие заключается в использовании функций `recvfrom` и `sendto` вместо `read` и `write`. Объяснение, которое приводилось в предыдущем подразделе, актуально и для датаграммного клиента.

Теперь пришло время поговорить о сетевых сокетах. Как вы увидите, все различия при переходе с UDS на сетевые сокеты будут находиться в функциях `main` клиентской и серверной программ.

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.

В следующих разделах мы увидим, как разрабатываются программы на основе TCP- и UDP-сокетов, и рассмотрим реальные примеры в рамках проекта «Калькулятор».

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.

Возвращаясь к проекту «Калькулятор», все отличия следует искать в функциях `main`, где мы создаем объект сокета и привязываем его к конечной точке. Остальной код должен оставаться без изменений. В листинге 20.29 вы можете видеть функцию `main`, принадлежащую TCP-серверу.

Листинг 20.29. Главная функция TCP-сервера (`server/tcp/main.c`)

```
int main(int argc, char** argv) {

    // ----- 1. Создаем объект сокета -----
    int server_sd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_sd == -1) {
        fprintf(stderr, "Could not create socket: %s\n",
                strerror(errno));
        exit(1);
    }

    // ----- 2. Привязываем файл сокета -----
    // Подготавливаем адрес
    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. Подготавливаем резерв -----
    ...

    // ----- 4. Начинаем принимать клиентов -----
    accept_forever(server_sd);

    return 0;
}
```

Если сравнить этот код с функцией `main`, которую мы видели в листинге 20.17, то можно заметить упомянутые ранее отличия. Для привязанного адреса конеч-

ной точки теперь применяется структура `sockaddr_in`, а не `sockaddr_un`. Функция `listen` используется тем же образом, а для обработки входящих соединений вызывается та же функция `accept_forever`.

В завершение отмечу кое-что относительно операций ввода/вывода: будучи потоковым, TCP-сокет обладает теми же свойствами; следовательно, его можно использовать так же, как и любой другой потоковый сокет. Иными словами, мы можем вызывать все те же функции `read`, `write` и `close`.

Теперь обсудим TCP-клиент.

TCP-клиент

Здесь тоже все должно быть похоже на потоковый клиент, работающий с UDS. Отличия, упомянутые выше, актуальны и для TCP-сокета на стороне соединителя и ограничены функцией `main`.

В листинге 20.30 вы можете видеть главную функцию TCP-клиента.

Листинг 20.30. Главная функция TCP-клиента (`client/tcp/main.c`)

```
int main(int argc, char** argv) {  
  
    // ----- 1. Создаем объект сокета -----  
  
    int conn_sd = socket(AF_INET, SOCK_STREAM, 0);  
    if (conn_sd == -1) {  
        fprintf(stderr, "Could not create socket: %s\n",  
                strerror(errno));  
        exit(1);  
    }  
  
    // ----- 2. Подключаемся к серверу -----  
  
    // Находим IP-адрес, которому принадлежит это сетевое имя  
    ...  
  
    // Подготавливаем адрес  
    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);  
  
    ...  
  
    stream_client_loop(conn_sd);  
  
    return 0;  
}
```

Изменения очень похожи на те, которые мы видели в TCP-сервере. Здесь используется другое семейство адресов и другая структура для хранения адреса сокета. Остальной код не претерпел изменений, поэтому подробно обсуждать TCP-клиент нет нужды.

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». Теперь вы знаете, зачем мы создали две общие библиотеки: одну для клиентской программы, а вторую для серверной. Это было сделано для того, чтобы сократить объем кода. Если один и тот же код подходит для двух разных ситуаций, то его всегда лучше вынести в библиотеку, которую можно будет использовать повторно.

Рассмотрим серверную и клиентскую UDP-программы; как вы сами сможете убедиться, они более или менее похожи на то, что мы уже видели в TCP-программах.

UDP-сервер

UDP-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. Листинг 20.31 содержит главную функцию UDP-сервера.

Листинг 20.31. Главная функция UDP-сервера (`server/udp/main.c`)

```
int main(int argc, char** argv) {

    // ----- 1. Создаем объект сокета -----
    int server_sd = socket(AF_INET, SOCK_DGRAM, 0);
    if (server_sd == -1) {
        fprintf(stderr, "Could not create socket: %s\n",
                strerror(errno));
        exit(1);
    }

    // ----- 2. Привязываем файл сокета -----

    // Подготавливаем адрес
    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. Начинаем обслуживать запросы -----
serve_forever(server_sd);

return 0;
}

```

Обратите внимание: UDP-сокеты являются датаграммными. Поэтому весь код, написанный для датаграммных сокетов домена Unix, актуален и для них. Например, работа с UDP-сокетами требует применения функций `recvfrom` и `sendto`. В связи с этим для обслуживания входящих датаграмм была использована та же функция `serve_forever`. Она входит в состав общей серверной библиотеки, в которой собран код, относящийся к датаграммам.

О коде UDP-сервера было сказано достаточно. Посмотрим, как выглядит UDP-клиент.

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. В листинге 20.32 вы можете видеть функцию `main`.

Листинг 20.32. Главная функция UDP-клиента (client/udp/main.c)

```

int main(int argc, char** argv) {

// ----- 1. Создаем объект сокета -----

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

// ----- 2. Подключаемся к серверу -----
...

// Подготавливаем адрес
...

datagram_client_loop(conn_sd);

return 0;
}

```

Это была последняя концепция данной главы. Мы рассмотрели разные общеизвестные типы сокетов и показали, как на языке С реализовать последовательности действий слушателя и соединителя в контексте потоковых и датаграммных каналов.

Проект «Калькулятор» имеет много аспектов, о которых я даже не упомянул. Поэтому настоятельно рекомендую вам пройтись по коду, найти эти места и попытаться их понять. Наличие полностью рабочего примера поможет вам исследовать концепции, которые можно встретить в реальных приложениях.

Резюме

В этой главе мы:

- в рамках обзора методов IPC познакомились с различными типами взаимодействия, каналов, носителей и сокетов;
- исследовали проект «Калькулятор», рассмотрев его прикладной протокол и алгоритм сериализации, который он использует;
- увидели, как с помощью сокетов UDS установить клиент-серверное соединение и как их задействовать в проекте «Калькулятор»;
- по очереди обсудили потоковые и датаграммные каналы, создаваемые с применением сокетов домена Unix;
- рассмотрели, как с помощью TCP- и UDP-сокетов, которые использовались в примере с калькулятором, можно создать клиент-серверный канал межпроцессного взаимодействия.

Следующая глава посвящена интеграции С с другими языками программирования. Такая интеграция позволяет загрузить библиотеку, написанную на С, в среду выполнения другого языка, такого как Java. В рамках следующей главы мы также поговорим об интеграции с C++, Java, Python и Golang.