Оглавление

1 Базовые механизмы

Для взаимодействия распределенных приложений в среде ОС UNIX наибольшее распространение получили следующие три средства:

1. **socket** - интерфейс прикладной программы с модулем ОС, реализующим сетевое взаимодействие;
2. **интерфейс транспортного уровня** (TLI - Transport Level Interface);
3. **средства удаленного вызова процедур** (RPC - Remote Procedure Call).

Примечание. *Перечисленные средства программирования универсальны, и могут быть использованы для работы с различными протоколами сетевого взаимодействия, однако в данном случае они рассматриваются применительно только к протоколам TCP/IP.*

Рассматриваемые средства предназначены для создания распределенных приложений, функционирующих, в первую очередь, согласно модели взаимодействия "клиент-сервер". Одна программа-сервер может обслуживать (последовательно) несколько клиентов, организуя очередь запросов от них.

Двумя наиболее общими режимами взаимодействия прикладных

программ в вычислительной сети являются:

1. **режим с установлением соединения;**

2. **режим без установления соединения.**

Режим с установлением соединения подразумевает, что взаимодействие осуществляется в три этапа:

- **установление логической связи между прикладными программами (по инициативе клиента);**
- **двусторонний, последовательный (потоковый, без учета каких-либо границ записей), надежный (без потери и дублирования) обмен данными;**
- **закрытие связи (экстренное или с доставкой буферизованных на передающей стороне данных).**

В режиме без установления соединения обмен информацией ведется отдельными блоками данных, часто называемых дейтаграммами и содержащими в себе помимо собственно данных еще и **адрес их получателя** (соответственно, клиента или сервера). В этом режиме, как правило, не гарантируется надежность доставки данных (они могут быть потеряны или продублированы), может быть нарушена правильная последовательность дейтаграмм на принимающей стороне, но, очевидно, явно присутствуют границы в передаваемых данных.

Программист имеет возможность выбрать режим взаимодействия, отвечающий специфике создаваемого приложения.

В сетях **TCP/IP** для организации режима взаимодействия с логическим соединением используется протокол транспортного уровня **TCP,** а режима без установления соединения -протокол **UDP.**

1.1 Socket-интерфейс

Данное средство было первоначально разработано для обеспечения прикладным программистам в среде ОС UNIX доступа к транспортному уровню стека протоколов TCP/IP. Позже оно было адаптировано для использования иных протоколов

Socket (гнездо, разъем) - абстрактное программное понятие, используемое для обозначения в прикладной программе конечной точки канала связи с коммуникационной средой, образованной вычислительной сетью.

При использовании протоколов **TCP/IP** можно говорить, что **socket** является средством подключения прикладной программы к порту локального узла сети.

Socket-интерфейс представляет собой просто набор системных вызовов и/или библиотечных функций **языка программирования СИ**, разделенных на четыре группы:

1. **локального управления;**
2. **установления связи;**
3. **обмена данными (ввода/вывода);**
4. **закрытия связи.**

1.2 Интерфейс транспортного уровня

Интерфейс транспортного уровня (**TLI - Transport Level Interface**) был разработан как альтернатива более раннему **socket-интерфейсу**. Он базируется на средстве ввода-вывода **STREAMS**, первоначально реализованном в версиях **System V** операционной системы **UNIX**.

Основное достоинство **STREAMS** заключается в гибкой, управляемой пользователем многослойности модулей, по конвейерному принципу обрабатывающих информацию, передаваемую от прикладной программы к физической среде хранения/пересылки и обратно. Это делает STREAMS удобным инструментом для реализации стеков протоколов сетевого взаимодействия различной архитектуры (**OSI, TCP/IP, DECnet, SNA, XNS и т.п.**).

Хотя все современные реализации версии **ОС UNIX** поддерживают **socket-интерфейс** по крайней мере для **TCP/IP**, для вновь разрабатываемых сетевых приложений настоятельно рекомендуется использовать **TLI** что обеспечит их **независимость от используемых сетевых протоколов**.

С точки зрения прикладного программиста логика **TLI** очень похожа на

логику **socket-интерфейса**.

TLI реализован в виде библиотеки функций языка программирования СИ, разделенных (как и в случае с **socket-интерфейсом**) на четыре группы:

1. **локального управления;**
2. **установления связи;**
3. **обмена данными (ввода/вывода);**
4. **закрытия связи.**

Основу концепции **TLI** составляют три базовых понятия:

- **поставщик транспортных услуг**
- **пользователь транспорта**
- **транспортная точка.**

Поставщиком транспортных услуг (**transport provider**) называется набор модулей, реализующих какой-либо конкретный стек протоколов сетевого взаимодействия (в данном случае - **TCP/IP**) и обеспечивающий **сервис транспортного уровня модели OSI.**

Пользователем транспорта (transport user) является любая прикладная программа, использующая сервис, предоставляемый **ПТС** на локальном узле сети.

Транспортная точка (transport endpoint) -абстрактное понятие (аналогичное **socket'у**), используемое для обозначения канала связи между пользователем транспорта и поставщиком транспортных услуг на локальном узле сети.

Транспортная точка имеет уникальный для всей сети транспортный адрес (для сетей **TCP/IP** этот адрес образуется триадой:

адрес узла сети,

номер порта,

используемый протокол транспортного уровня.

Для ссылки на транспортные точки в функциях TLI используются их дескрипторы, подобные дескрипторам обычных файлов и socket'oB ОС UNIX.

1.3 Вызов удаленных процедур

Средства вызова удаленных процедур **(RPC)** является составной частью более общего средства, называемого **Open Network Computing (ONC)**, разработанного фирмой **Sun Microsystems**, и получившего всеобщее признание в качестве промышленного стандарта.

ONC помимо **RPC** включает в себя средства внешнего представления данных **(XDR)**, необходимые для организации обмена информацией в гетерогенных сетях, включающих в себя ЭВМ различной архитектуры, и средства монтирования сетевых файловых систем **(NFS)**, обеспечивающее доступ пользователям локального узла сети к файлам, физически расположенным на удаленных узлах, как к файлам локальным.

Средство **RPC** реализует модель **"клиент-сервер"**, где роль клиента играет прикладная программа, обращающаяся к набору процедур (функций), исполняемых на удаленном узле в качестве сервера.

RPC предоставляет прикладным программистам сервис более высокого уровня, чем ранее рассмотренных два, т.к. обращение за услугой к удаленным процедурам выполняется в привычной для программиста манере вызова "локальной" функции языка программирования СИ.

RPC реализовано, как правило, на базе **socket** интерфейса и/или **TLI**.

Средство RPC предоставляет программистам сервис трех уровней:

1. препроцессор rpcgen, преобразующий исходные тексты "монолитных" программ на языке программирования СИ в исходные тексты программы-клиента и программы сервера по спецификации программиста;
2. библиотека функций вызова удаленных процедур по их идентификаторам;
3. библиотека низкоуровневых функций доступа к внутренним механизмам RPC и нижележащим протоколам транспортного уровня.

2 Программный интерфейс высокого уровня. Удаленный

вызов процедур

2.1 Концепция

При программном интерфейсе низкого уровня по существу программа взаимодействовала непосредственно с транспортным протоколом, самостоятельно реализуя некоторый протокол верхнего уровня при обмене данными. Значительная часть кода этих программ посвящена созданию коммуникационных узлов, установлению и завершению связи.

С точки зрения разработчика программного обеспечения, более перспективным является подход, когда используется прикладной программный интерфейс более высокого уровня, изолирующий программу от специфики сетевого взаимодействия.

Одним из таких подходов, на базе которого, в частности, разработана файловая система NFS, является подход, получивший название удаленный вызов процедур (Remote Procedure Call, RPC).

Использование подпрограмм в программе - традиционный способ структурировать задачу, сделать ее более ясной. Наиболее часто используемые подпрограммы собираются в библиотеки, где могут использоваться различными программами. В данном случае речь идет о локальном (местном) вызове, т. е. и вызывающий, и вызываемый объекты работают в рамках одной программы на одном компьютере.

В случае удаленного вызова процесс, выполняющийся на одном компьютере, запускает процесс на удаленном компьютере (т. е. фактически запускает код процедуры на удаленном компьютере). Очевидно, что удаленный вызов процедуры существенным образом отличается от традиционного локального, однако с точки зрения программиста такие отличия практически отсутствуют, т. е. архитектура удаленного вызова процедуры позволяет сымитировать вызов локальной.

Однако если в случае локального вызова программа передает параметры в вызываемую процедуру и получает результат работы через стек или общие области памяти, то в случае удаленного вызова передача параметров

превращается в передачу запроса по сети, а результат работы находится в пришедшем отклике.

Данный подход является возможной основой создания распределенных приложений, и хотя многие современные системы не используют этот механизм, основные концепции и термины во многих случаях сохраняются. При описании механизма RPC мы будем традиционно называть вызывающий процесс — клиентом, а удаленный процесс, реализующий процедуру, — сервером.

2.2 Алгоритм удаленного вызова процедуры:

1. Программа-клиент производит локальный вызов процедуры, называемой заглушкой (stub). При этом клиенту "кажется", что, вызывая заглушку, он производит собственно вызов процедуры-сервера. И действительно, клиент передает заглушке необходимые параметры, а она возвращает результат. Однако дело обстоит не совсем так, как это себе представляет клиент. Задача заглушки - принять аргументы, предназначаемые удаленной процедуре, возможно, преобразовать их в некий стандартный формат и сформировать сетевой запрос. Упаковка аргументов и создание сетевого запроса называется сборкой (marshalling).

2. Сетевой запрос пересылается по сети на удаленную систему. Для этого в заглушке используются соответствующие вызовы. При этом могут быть использованы различные транспортные протоколы, причем не только семейства TCP/IP.

3. На удаленном хосте все происходит в обратном порядке. Заглушка сервера ожидает запрос и при получении извлекает параметры - аргументы вызова процедуры. Извлечение (unmarshalling) может включать необходимые преобразования (например, изменения порядка расположения байтов)

4. Заглушка выполняет вызов настоящей процедуры-сервера, которой адресован запрос клиента, передавая ей полученные по сети аргументы.

5. После выполнения процедуры управление возвращается в заглушку сервера, передавая ей требуемые параметры. Как и заглушка клиента, заглушка сервера преобразует возвращенные процедурой значения формируя сетевое

сообщение-отклик, который передается по сети системе, от которой пришел запрос.

6. Операционная система передает полученное сообщение заглушке клиента, которая, после необходимого преобразования, передает значения (являющиеся значениями, возвращенными удаленной процедурой) клиенту, воспринимающему это как нормальный возврат из процедуры.

Таким образом, с точки зрения клиента, он производит вызов удаленной процедуры, как он это сделал бы для локальной.

То же самое можно сказать и о сервере: вызов процедуры происходит стандартным образом, некий объект (заглушка сервера) производит вызов локальной процедуры и получает возвращенные ею значения.

Клиент воспринимает заглушку как вызываемую процедуру-сервер, а сервер принимает собственную заглушку за клиента.

Таким образом, заглушки составляют ядро системы RPC, отвечая за все аспекты формирования и передачи сообщений между клиентом и удаленным сервером (процедурой), хотя и клиент, и сервер считают, что вызовы: происходят локально. В этом-то и состоит основная концепция RPC — полностью спрятать распределенный (сетевой) характер взаимодействия в коде заглушек. Преимущества такого подхода очевидны: и клиент и сервер являются независимыми от сетевой реализации, оба они работают в рамках некой распределенной виртуальной машины, и вызовы процедур имеют стандартный интерфейс.

2.3 Передача параметров

Передача параметров-значений не вызывает особых трудностей. В этом случае заглушка клиента размещает значение параметра в сетевом запросе, возможно, выполняя преобразования к стандартному виду (например, изменяя порядок следования байтов). Гораздо сложнее обстоит дело с передачей указателей, когда параметр представляет собой адрес данных, а не их значение. Передача в запросе адреса лишена смысла, так как удаленная процедура выполняется в совершенно другом адресном пространстве. Самым простым

решением, применяемым в RPC, является запрет клиентам передавать параметры иначе, как по значению, хотя это, безусловно накладывает серьезные ограничения.

2.4 Связывание (binding)

Прежде чем клиент сможет вызвать удаленную процедуру, необходимо связать его с удаленной системой, располагающей требуемым сервером Таким образом, задача связывания распадается на две:

- Нахождение удаленного хоста с требуемым сервером
- Нахождение требуемого серверного процесса на данном хосте

Для нахождения хоста могут использоваться различные подходы.

Возможный вариант - создание некоего централизованного справочника, в котором хосты анонсируют свои серверы, и где клиент при желании может выбрать подходящие для него хост и адрес процедуры.

Каждая процедура RPC однозначно определяется номером программы и процедуры. Номер программы определяет группу удаленных процедур, каждая из которых имеет собственный номер. Каждой программе также присваивается номер версии, так что при внесении в программу незначительных изменений (например, при добавлении процедуры) отсутствует необходимость менять ее номер. Обычно несколько функционально сходных процедур реализуются в одном программном модуле, который при запуске становится сервером этих процедур, и который идентифицируется номером программы.

Таким образом, когда клиент хочет вызвать удаленную процедуру, ему необходимо знать номера программы, версии и процедуры, предоставляющей требуемый сервис.

Для передачи запроса клиенту также необходимо знать сетевой адрес хоста и номер порта, связанный с программой-сервером, обеспечивающей требуемые процедуры.

Для этого используется демон portmap(lM) (в некоторых системах он называется rpcbind(lM)). Демон запускается на хосте, который предоставляет сервис удаленных процедур, и использует общеизвестный номер порта.

При инициализации процесса-сервера он регистрирует в portmap(lM) свои процедуры и номера портов. Теперь, когда клиенту требуется знать номер порта для вызова конкретной процедуры, он посылает запрос на сервер portmap(lM), который, в свою очередь, либо возвращает номер порта, либо перенаправляет запрос непосредственно серверу удаленной процедуры и после ее выполнения возвращает клиенту отклик.

В любом случае, если требуемая процедура существует, клиент получает от сервера portmap(lM) номер порта процедуры, и дальнейшие запросы может делать уже непосредственно на этот порт.

2.5 Обработка особых ситуаций (exception)

Обработка особых ситуаций при вызове локальных процедур не представляет особой проблемы.

UNIX обеспечивает обработку ошибок процессов, таких как деление на ноль, обращение к недопустимой области памяти и т. д.

В случае вызова удаленной процедуры вероятность возникновения ошибочных ситуаций увеличивается. К ошибкам сервера и заглушек добавляются ошибки, связанные, например, с получением ошибочного сетевого сообщения.

Например, при использовании UDP в качестве транспортного протокола производится повторная передача сообщений после определенного тайм-аута. Клиенту возвращается ошибка, если, спустя определенное число попыток, отклик от сервера так и не был получен. В случае, когда используется протокол TCP. клиенту возвращается ошибка, если сервер оборвал TCP-соединение.

2.6 Семантика вызова

Вызов локальной процедуры однозначно приводит к ее выполнению, после чего управление возвращается в головную программу.

Иначе дело обстоит при вызове удаленной процедуры.

Невозможно установить, когда конкретно будет выполняться процедура, будет ли она выполнена вообще, а если будет, то какое число раз?

Например, если запрос будет получен удаленной системой после аварийного завершения программы сервера, процедура не будет выполнена вообще.

Если клиент при неполучении отклика после определенного промежутка времени (тайм-аута) повторно посылает запрос, то может создаться ситуация, когда отклик уже передается по сети, а повторный запрос вновь принимается на обработку удаленной процедурой. В этом случае процедура будет выполнена несколько раз.

Таким образом, выполнение удаленной процедуры можно характеризовать следующей семантикой:

Один и только один раз. Данного поведения (в некоторых случаях наиболее желательного) трудно требовать ввиду возможных аварий сервера.

Максимум раз. Это означает, что процедура либо вообще не была выполнена, либо была выполнена только один раз. Подобное утверждение можно сделать при получении ошибки вместо нормального отклика.

Хотя бы раз. Процедура наверняка была выполнена один раз. но возможно и больше. Для нормальной работы в такой ситуации удаленная процедура должна обладать свойством идемпонентности (от англ idemponent). Этим свойством обладает процедура, многократное выполнение которой не вызывает кумулятивных изменений. Например чтение файла идемпонентно, а добавление текста в файл - нет.

2.7 Представление данных

Когда клиент и сервер выполняются в одной системе на одном компьютере, проблем с несовместимостью данных не возникает. И для клиента v для сервера данные в двоичном виде представляются одинаково.

В случае удаленного вызова дело осложняется тем. что клиент и сервер могут выполняться на системах с различной архитектурой, имеющих различное представление данных (например, представление значения с плавающей точкой, порядок следования байтов и т. д.)

Большинство реализаций системы RPC определяют некоторые стандартные виды представления данных, к которым должны быть преобразованы все

значения, передаваемые в запросах и откликах.

Например, формат представления данных в RPC фирмы Sun Microsystems следующий:

Порядок следования байтов **Старший — последний**

Представление значений с плавающей точкой **IEEE**

Представление символа **ASCII**

2.8 Сеть

По своей функциональности система RPC занимает промежуточное место между уровнем приложения и транспортным уровнем.

В соответствии с моделью OSI этому положению соответствуют **уровни представления и сеанса.**

Таким образом. RPC теоретически независим от реализации сети, в частности, от сетевых протоколов транспортного уровня.

Программные реализации системы, как правило, поддерживают один или два протокола.

Например, система RPC разработки фирмы Sun Microsystems поддерживает передачу сообщений с использованием протоколов TCP и UDP. Выбор того или иного протокола зависит от требований приложения.

Выбор протокола UDP оправдан для приложений, обладающих следующими характеристиками:

- Вызываемые процедуры идемпотентны
- Размер передаваемых аргументов и возвращаемого результата меньше размера пакета UDP — 8 Кбайт.
- Сервер обеспечивает работу с несколькими сотнями клиентов. Поскольку при работе с протоколами TCP сервер вынужден поддерживать соединение с каждым из активных клиентов, это занимает значительную часть его ресурсов. Протокол UDP в этом отношении является менее ресурсоемким

С другой стороны, TCP обеспечивает эффективную работу приложений со следующими характеристиками:

- Приложению требуется надежный протокол передачи

- Вызываемые процедуры неидемпонентны
- Размер аргументов или возвращаемого результата превышает 8 Кбайт

Выбор протокола обычно остается за клиентом, и система по-разному организует формирование и передачу сообщений.

Так, при использовании протокола TCP, для которого передаваемые данные представляют собой поток байтов, необходимо отделить сообщения друг от друга. Для этого, например, применяется протокол маркировки записей, при котором в начале каждого сообщения помещается 32-разрядное целое число, определяющее размер сообщения в байтах.

По-разному обстоит дело и с семантикой вызова.

Например, если RPC выполняется с использованием ненадежного транспортного протокола (UDP), система выполняет повторную передачу сообщения через короткие промежутки времени (тайм-ауты). Если приложение-клиент не получает отклик, то с уверенностью можно сказать, что процедура была выполнена ноль или большее число раз. Если отклик был получен, приложение может сделать вывод, что процедура была выполнена хотя бы однажды.

При использовании надежного транспортного протокола (TCP) в случае получения отклика можно сказать, что процедура была выполнена один раз. Если же отклик не получен, определенно сказать, что процедура выполнена не была, нельзя.

2.9 Реализация RPC

Система RPC является встроенной в программу-клиент и программу-сервер.

При разработке распределенных приложений, не придется вникать в подробности протокола RPC или программировать обработку сообщений. Система предполагает существование соответствующей среды разработки, которая значительно облегчает жизнь создателям прикладного программного обеспечения.

Одним из ключевых моментов в RPC является то, что разработка распределенного приложения начинается с определения интерфейса объекта -

формального описания функций сервера, сделанного на специальном языке.

На основании этого интерфейса затем автоматически создаются заглушки клиента и сервера. Единственное, что необходимо сделать после этого. - написать фактический код процедуры.

Пример. Система RPC фирмы Sun Microsystems. состоит из трех основных частей:

- rpcgen(l) - RPC-компилятор. который на основании описания интерфейса удаленной процедуры генерирует заглушки клиента и сервера в виде программ на языке С.
- Библиотека XDR (eXternal Data Representation), которая содержит функции для преобразования различных типов данных в машинно-независимый вид, позволяющий производить обмен информацией между разнородными системами.
- Библиотека модулей, обеспечивающих работу системы в целом.

3 Сетевая подсистема BSD UNIX (Berkley Software Design, Inc)

3.1 Архитектура подсистемы

- **Транспортный уровень Обмен данными между процессами**
- **Сетевой уровень Маршрутизация сообщений**
- **Уровень сетевого интерфейса Передача данных по физической сети**

Два верхних уровня представляют собой модули коммуникационных протоколов, а нижний уровень по существу является драйвером устройства.

Представленные уровни соответствуют транспортному, сетевому уровням и уровню канала данных модели OSI.

Транспортный уровень является самым верхним в системе и призван обеспечить необходимую адресацию и требуемые характеристики передачи данных, определенных коммуникационным узлом процесса, которым является **сокет.**

Например, сокет потока предполагает надежную последовательную доставку данных, и в семействе TCP/IP модуль данного уровня реализует протокол TCP.

Сетевой уровень обеспечивает передачу данных, адресованных удаленному сетевому или транспортному модулю.

Для этого модуль данного уровня должен иметь доступ к информации о маршрутах сети (таблице маршрутизации).

Уровень сетевого интерфейса отвечает за передачу данных хостам, подключенным к одной физической среде передачи (например, находящимся в одном сегменте Ethernet).

Внутренняя структура сетевой подсистемы изолирована от непосредственного доступа прикладных процессов.

Единым (и единственным) интерфейсом доступа к сетевым услугам является интерфейс сокетов.

Для обеспечения возможности работы с конкретным коммуникационным протоколом соответствующий модуль экспортирует интерфейсу сокетов функцию пользовательского запроса.

При этом данные от прикладного процесса передаются от интерфейса сокетов требуемым транспортным модулям с помощью соответствующих вызовов экспортированных функций. И наоборот, данные, полученные из сети, проходят обработку в соответствующих модулях протоколов и помещаются в очередь приема сокета-адресата.

Движение данных вниз (т. е. от верхних уровней к нижним) обычно инициируется системными вызовами и может иметь синхронный характер.

Принимаемые данные из сети поступают в случайные моменты времени и передаются сетевым драйвером в очередь приема соответствующего протокола.

При этом функции модуля протокола и обработка данных не вызываются непосредственно сетевым драйвером.

Вместо этого последний устанавливает бит соответствующего программного прерывания, в контексте которого система позднее и запускает необходимые функции.

Если данные предназначены протоколу верхнего уровня (транспортному), его функция обработки будет вызвана непосредственно модулем сетевого уровня.

Если же сообщение предназначено другому хосту, и система выполняет функции шлюза, сообщение будет передано уровню сетевого интерфейса для последующей передачи.

3.2 Сокеты.

Были введены понятия:

Коммуникационный домен (communication domain)

Коммуникационный домен обладает некоторым набором коммуникационных свойств(характеристик взаимодействия), определяющих

- способ именования сетевых узлов и ресурсов,
- характеристики сетевых соединений (надежные, дейтаграммные, упорядоченные),
- способы синхронизации процессов и т. п.

Одним из наиболее популярных доменов является домен Интернета с протоколами стека TCP/IP.

Сокет (socket)

Сокет (socket) - специальный объект для обозначения коммуникационного узла, обеспечивающего прием и передачу данных для объекта (процесса).

Сокет — это точка, через которую сообщения уходят в сеть или принимаются из сети. Сетевое соединение между двумя процессами осуществляется через пару сокетов.

Каждый процесс пользуется своим сокетом, при этом сокеты могут находится как на разных компьютерах, так и на одном (в этом случае сетевое межпроцессное взаимодействие сводится к локальному).

Сокет может иметь как высокоуровневое символьное имя (адрес), так и низкоуровневое, отражающее специфику адресации определенного коммуникационного домена. Например, в домене Интернета низкоуровневое имя представлено парой (IP-адрес, порт).

Сокеты создаются в рамках определенного коммуникационного домена, подобно тому как файлы создаются в рамках файловой системы.

Сокеты имеют соответствующий интерфейс доступа в файловой системе UNIX, и так же как обычные файлы, адресуются некоторым целым числом — дескриптором. Однако в отличие от обычных файлов, сокеты представляют собой виртуальный объект, который существует, пока на него ссылается хотя бы один из процессов.

В BSD UNIX реализованы следующие основные типы сокетов:

Сокет датаграмм (datagram socket) - теоретически ненадежная, несвязная передача пакетов.

Сокет потока (stream socket) - надежная передача потока байтов без сохранения границ сообщений. Этой тип сокетов поддерживает передачу экстренных данных.

Сокет пакетов (packet socket) - надежная последовательная передача данных без дублирования с предварительным установлением связи. При этом сохраняются границы сообщений.

Сокет низкого уровня (raw socket) - непосредственный доступ к коммуникационному протоколу.

Для сокетов должно быть определено пространство имен. Имя сокета имеет смысл только в рамках коммуникационного домена, в котором он создан.

4 Поддержка сети в UNIX System V

Многие из аспектов реализации поддержки сети в **BSD UNIX** справедливы и для архитектуры сетевых протоколов **UNIX System V**. Однако сам механизм обеспечения взаимодействия модулей существенно отличается.

Для поддержки сети в **UNIX System V** используется подсистема **STREAMS**.

4.1 Архитектура STREAMS

Подсистема STREAMS предоставляет интерфейс обмена данными, основанный на сообщениях, и обеспечивает стандартные механизмы буферизации, управления потоком данных и различную приоритетность

обработки.

В STREAMS дублирование кода сводится к минимуму, поскольку однотипные функции обработки реализованы в независимых модулях, которые могут быть использованы различными драйверами.

В настоящее время подсистема STREAMS поддерживается большинством производителей операционных систем UNIX и является основным способом реализации сетевых драйверов и модулей протоколов. Использование STREAMS охватывает и другие устройства, например терминальные драйверы в UNIX SVR4.

Подсистема STREAMS обеспечивает создание потоков - полнодуплексных коммуникационных каналов между прикладным процессом и драйвером устройства.

Сам поток полностью располагается в пространстве ядра, соответственно и все функции обработки данных выполняются в системном контексте.

Типичный поток состоит из головного модуля, драйвера и, возможно, одного или более модулей.

Головной модуль взаимодействует с прикладными процессами через интерфейс системных вызовов.

Драйвер, замыкающий поток, взаимодействует непосредственно с физическим устройством или псевдоустройством, в качестве которого может выступать другой поток. Модули выполняют промежуточную обработку данных.

Процесс взаимодействует с потоком, используя стандартные системные вызовы open(2), close(2), read(2), write(2) и ioctl(2).

Передача данных по потоку осуществляется в виде сообщений, содержащих данные, тип сообщения и управляющую информацию.

Для передачи данных каждый модуль, включая головной модуль и сам драйвер, имеет две очереди — очередь чтения (read queue) и очередь записи (write queue).

Каждый модуль обеспечивает необходимую обработку данных и передает

их в очередь следующего модуля. При этом передача в очередь записи осуществляется вниз по потоку (downstream), а в очередь чтения — вверх по потоку (upstream).

Мультиплексирование

Подсистема STREAMS обеспечивает возможность мультиплексирования потоков с помощью мультиплексора, который может быть реализован только драйвером STREAMS.

Различают три типа мультиплексоров - верхний, нижний и гибридный.

Верхний мультиплексор, называемый также мультиплексором N: 1, обеспечивает подключение нескольких каналов вверх по потоку к одному каналу вниз по потоку.

Нижний мультиплексор, называемый также мультиплексором 1:M, обеспечивает подключение нескольких каналов вниз по потоку к одному каналу вверх по потоку.

Гибридный мультиплексор позволяет мультиплексировать несколько каналов вверх по потоку с несколькими каналами вниз по потоку.

Подсистема STREAMS обеспечивает возможность мультиплексирования, но за идентификацию различных каналов и маршрутизацию данных между ними отвечает сам мультиплексор.

Подсистема ввода/вывода, основанная на архитектуре STREAMS, позволяет в полной мере отразить уровневую структуру коммуникационных протоколов, когда каждый уровень имеет стандартные интерфейсы взаимодействия с другими (верхним и нижним) уровнями, и может работать независимо от конкретной реализации протоколов на соседних уровнях.

Архитектура STREAMS полностью соответствует этой модели, позволяя создавать драйверы, которые являются объединениями независимых модулей.

Обмен данными между модулями STREAMS также соответствует характеру взаимодействия отдельных протоколов:

данные передаются в виде сообщений, а каждый модуль выполняет требуемую их обработку.

Модуль IP является гибридным мультиплексором, позволяя обслуживать несколько потоков, приходящих от драйверов сетевых адаптеров (в данном случае Ethernet и FDDI), и несколько потоков к модулям транспортных протоколов (TCP и UDP)

Модули TCP и UDP являются верхними мультиплексорами, обслуживающими прикладные программы, такие как

сервер маршрутизации routed(lM),

сервер удаленного терминального доступа telnetd(W),

сервер FTP ftpd(W),

программы-клиенты пользователей (например talk(l)).

Анализ программного обеспечения сетевой поддержки показывает, что как правило сетевые и транспортные протоколы, составляющие базовый стек TCP/IP, поставляются одним производителем, в то время как поддержка уровней сетевого интерфейса и приложений может осуществляться продуктами различных разработчиков.

Соответственно, можно выделить два основных интерфейса взаимодействия, стандартизация которых позволяет обеспечить совместную работу различных компонентов программного обеспечения.

Первый интерфейс определяет взаимодействие транспортного уровня и уровня приложений и называется интерфейсом поставщика транспортных услуг (Transport Provider Interface, TPI).

Второй интерфейс устанавливает правила и формат сообщений, передаваемых между сетевым уровнем и уровнем сетевого интерфейса, и называется интерфейсом поставщика услуг канала данных (Data Link Provider Interface, DLPI).

Сетевая архитектура, основанная на архитектуре STREAMS, позволяет обеспечить поддержку любого стека протоколов, соответствующего модели OSI.