Глава 4. Прикладные протоколы клиент-серверных приложений. Примеры реализации

4.1. Проектирование прикладных протоколов

Преамбула. Существуют ли нерешенные проблемы?

Любая система, состоящая из клиентской и серверных частей, использует протокол прикладного уровня. Он может быть построен на основе универсального сетевого соединения (TCP/IP), либо встроен в один из прикладных протоколов (например, http).

Если полагать, что надежность базовой сети достаточна (если не абсолютна), то разработчику остается только определить форматы сообщений и набор примитивов взаимодействия типа «запрос-ответ». Что обычно и делается. Тем не менее, «чисто протокольные» проблемы имеют место и на прикладном уровне. Приведем несколько примеров:

- программные ошибки могут приводить к нарушению форматов сообщений и потере синхронизации. Случай из практики: сервер использует на каждое соединение единственный буфер для ответного сообщения, а также отдельный поток для его передачи и поток для приема. Если клиент будет посылать сообщение, не дождавшись ответа на предыдущее, то он может получить сообщение, содержащее начало первого и окончание второго (процесс записи в буфер второго ответа «обгонит» передачу первого);
- реальная сеть может быть ненадежна (например, мертвые зоны в мобильной связи), поэтому на прикладном уровне необходимо предусмотреть процедуры восстановления не только самих соединений, но и последовательности принимаемых и передаваемых сообщений, а также средства отложенной передачи в состоянии «временно недоступен»;
- при длительном или предполагаемом отсутствии трафика соединение можно закрывать с сохранением контекста у клиента и сервера, а при появлении трафика – восстанавливать без повторной авторизации;
- обычной практикой является собственный keep alive, поскольку при отсутствии трафика в соединении «падение» одного из участников (например, крах операционной системы) или промежуточного звена не будут замечены остальными достаточно долго;
- клиент может посылать сообщения, не дожидаясь ответа на предыдущее – требуется установление соответствия запросов и ответов;
- аналогично клиенту, сервер может являться инициатором взаимодействия, например, обновлять или запрашивать данные клиента – требуется распознавание сообщений во встречных потоках.

Специфика программирования протокольных процессов заключается в самой природе протокола. Напомним его определение [2.1]:

Протокол – описание правил взаимодействия параллельных независимых (асинхронных) процессов путем обмена сообщениями через ненадежную инертную пространственную среду.

Принципиальным моментом является то, что любой протокол не обеспечивает 100% надежности доставки сообщений с учетом перечисленных выше факторов, поэтому дублирование функций протоколов низших уровней вполне оправдано.

Аспекты описания протокола и его реализации

Программная реализация протоколов содержит много специфических моментов, связанных с особенностями протокольных процессов (параллелизм, синхронизация, производительность). Здесь мы обсудим их на общем уровне, соответствующие иллюстрации можно найти далее. Прикладной протокол и реализующие его процессы можно рассматривать в нескольких аспектах:

- **функциональный** – функциональность протокола – перечень примитивов взаимодействия (запросов-ответов), наличие механизмов восстановления, повышения надежности, синхронизации состояний;
- **структурный** – все, что связано с передаваемыми данными - форматы сообщений, программные интерфейсы;
- **алгоритмический** – принципы и алгоритмы реализации функционала – общие механизмы взаимодействия, процедуры обмена, состояния протокола, автоматные модели, процедуры синхронизации состояний, восстановления, поддержки сессий;
- **процессный** – особенности реализации протокола как системы процессов (потоков) - параллелизм, потоки, синхронизация, производительность, задержки.

Прикладной протокол как элемент взаимодействия слоев клиент-серверной системы

Стандартным способом реализации прикладного протокола в клиент-серверной системе является его встраивание в интерфейс между функциональными уровнями. Интерфейс «расшивается» следующим образом:

- функциональный слой клиента соединяется через интерфейс с **клиентской компонентой протокола;**
- клиент протокола взаимодействует через сеть с **серверной компонентой протокола;**
- в серверной компоненте протокола создается **агент,** от имени которого удаленный клиент вызывает необходимые в следующем слое через «расшитый» интерфейс;
- наличие **агента** как отдельной компоненты объясняется тем, что для выполнения запросов к следующему уровню на сервере необходимо создавать, передавать от клиента или постоянно поддерживать необходимое окружение (**контекст)** (C), в котором выполняются эти запросы.

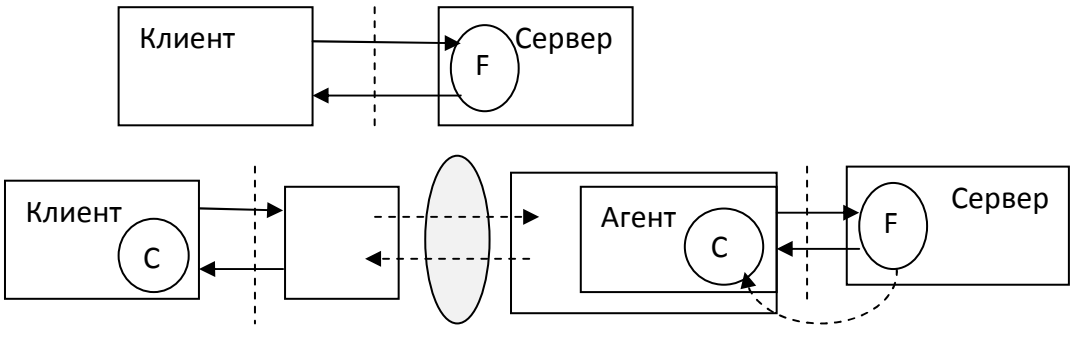

Рис.4.1. Классическая реализация удаленного доступа с "расшивкой" интерфейса

Данный вариант соответствует наиболее распространенной стратегии взаимодействия - **синхронные запросы клиента к серверу** (см.1.3). Например, при двунаправленной синхронных или асинхронных запросах каждая сторона должна иметь своего агента, а компоненты протокола – идентифицировать сообщения во встречных потоках.

Параллелизм, синхронизация

Протокольный процесс обычно включает в себя несколько параллельно исполняемых компонент. Необходимость их параллельного исполнения вызвана разными причинами:

- в компоненте происходит ожидание каких-либо событий (например, приема данных) с блокировкой потока;
- по самой природе протокола компоненты работают одновременно;
- исполнение компоненты достаточно продолжительно, и ее исполнение в основном потоке приведет к зависанию графического интерфейса приложения;
- компонента может иметь более высокий или более низкий приоритет исполнения.

Перечислим наиболее часто встречающиеся:

- поток приема – является необходимым, если используется синхронный прием данных, который сопровождается ожиданием с блокировкой текущего потока, например в классах Socket (TCP/IP) или URLConnection (HTTP). Иногда при приеме с блокированием реализуются более «хитрые» механизмы:
    - поток «засыпает», а при пробуждении проверяет наличие принятых данных (метод available), если они присутствуют, то начитается цикл приема, иначе поток снова «засыпает»;
    - после начала приема устанавливается тайм-аут ожидания приема, который сбрасывается по окончании приема сообщения. Это исключает зависание потока, если по каким-то причинам сообщение не было передано до конца.
- дополнительный поток передачи/приема при использовании механизма отложенного опроса (см.1.3);
- поток для поддержки дополнительных соединений, например, в FTP для передачи/приема файла вне основного потока сообщений;
- поток передачи – является необходимым, если время блокирования при синхронной передаче данных является существенным. Кроме того, чтобы исключить паузы в самом процессе передачи, данные сообщения сначала пишутся в байтный буфер (ByteArayOutputStream), а затем извлекаются единым массивом;
- тайм-ауты. Протокольный процесс выставляет интервалы времени, в течение которых должны завершиться инициированные им взаимодействия или процессы. Если они завершаются, то тайм-аут отменяется. В противном случае по истечении тайм-аута выполняется некоторая восстановительная процедура. Для реализации тайм-аутов могут использоваться как обычные потоки, так и специальные классы, поддерживающие отложенные по времени события (Timer,Handler). В протокольном процессе тайм-ауты могут иметь место везде, где возможны зависания и блокировки:
    - тайм-аут замыкания петли «запрос-ответ» или получения подтверждения приема (тайм-аут передачи);

- тайм-аут попытки восстановления соединения при его временном пропадании;
- тайм-аут проверки активности соединения (keep alive);
- тайм-аут закрытия соединения (shutdown);
- тайм-аут ожидания приема – исключает зависание на приеме сообщения после начала приема (обрыв передачи).

Примечание. Следует заметить, что обычное TCP/IP соединение не является исключительно реактивным по отношению к состоянию соединения. Т.е. если, например, на другом конце соединения происходит крах операционной системы, либо аварийное завершение процесса, то другая сторона может «не догадываться» об этом в течение достаточно долгого времени. Поэтому указанные тайм-ауты не являются чем-то вычурным. Скорее наоборот, все, что может зависнуть, зависает. Аналогичный эффект производят ошибки программирования, когда ожидаемая последовательность данных может нарушаться, либо могут возникать блокировки (клинчи).

Буферизация, очереди

Очереди сообщений и, соответственно, необходимость их буферизации, возникает в нескольких случаях:

- клиент содержит несколько потоков, создающих запросы к серверу, либо несколько независимых источников запросов, вызываемых событиями, т.е. мультиплексирует запросы от разных источников в одно соединение;
- клиент может посылать запрос, не дождавшись завершения предыдущего.

Примечание. Иногда явная буферизация отсутствует, но в потоке приема имеется синхронизация к основному потоку, которая на каждое принятое сообщение вызывает метод, который фактически ставит дальнейший исполняемый код (Runnable) в очередь, а с ней уже работает основной поток. В любом случае желательно, чтобы данные принимались в динамически выделяемые буферы (объекты).

Пример дефекта буферизации. В процессе взаимодействия с сервером с закрытым кодом обнаружилось, что он нарушает формат ответных сообщений, которой, вроде бы, избыточно контролируется форматом. Видимо, для каждого соединения был выделен отдельный буфер передачи, и посылка подряд двух или более команд могла привести к тому, что буфер перезаписывался ответом на второе сообщение в процессе передачи первого. Все это осталось лишь догадкой, проблема была решена с помощью создания очереди запросов и выдачи их драйверу с заданным интервалом.

Диспетчеризация, планирование

Иногда вместо обычной буферизации сообщений применяется «буферизация кода» запроса-ответа в целом. Она заключается в последовательном исполнении в одном потоке всех поступающих запросов и реализуется шаблоном проектирования **планировщик (scheduler)**. Интерфейс объекта-запроса на планирование имеет методы, в которых прописывается:

- код, исполняемый планировщиком в собственном потоке;
- код, исполняемый в основном потоке, вызываемый при завершении предыдущего;
- код обработки исключений, возникших при исполнении указанных выше;
- код реакции на отмену планирования и т.д..

Планировщик поддерживает очередь объектов-запросов и собственный поток, который выбирает из очереди запросы и последовательно выполняет передаваемый в них код, а также передает код завершения основному потоку.

Использование планировщика в клиенте позволяет бесконфликтно выполнять запросы к серверу, даже если они поступают параллельно.

Диаграмма состояний протокола

Для того, чтобы логически связать воедино все параллельно исполняемые компоненты протокольного процесса и исключить ошибки их взаимодействия, необходима единая модель описания поведения протокольного процесса. В этом качестве часто используются конечные автоматы и их графическое представление в виде диаграмм состояний (см.2.2).

С технологической точки зрения все события, которые изменяют или проверяют состояние конечного автомата (команды верхнего уровня, тайм-ауты, принятые сообщения), должны быть синхронизированы между собой. Объектом синхронизации может быть сам автомат.

В простейших случаях диаграмма состояний описывает поведение отдельно клиентской или серверной компоненты. Например, диаграмма состояний соединения в прикладном протоколе со стороны клиента. По диаграмме состояний легко анализируется и тестируется корректность поведения автомата при различных последовательностях событий, определяются возможные блокировки.

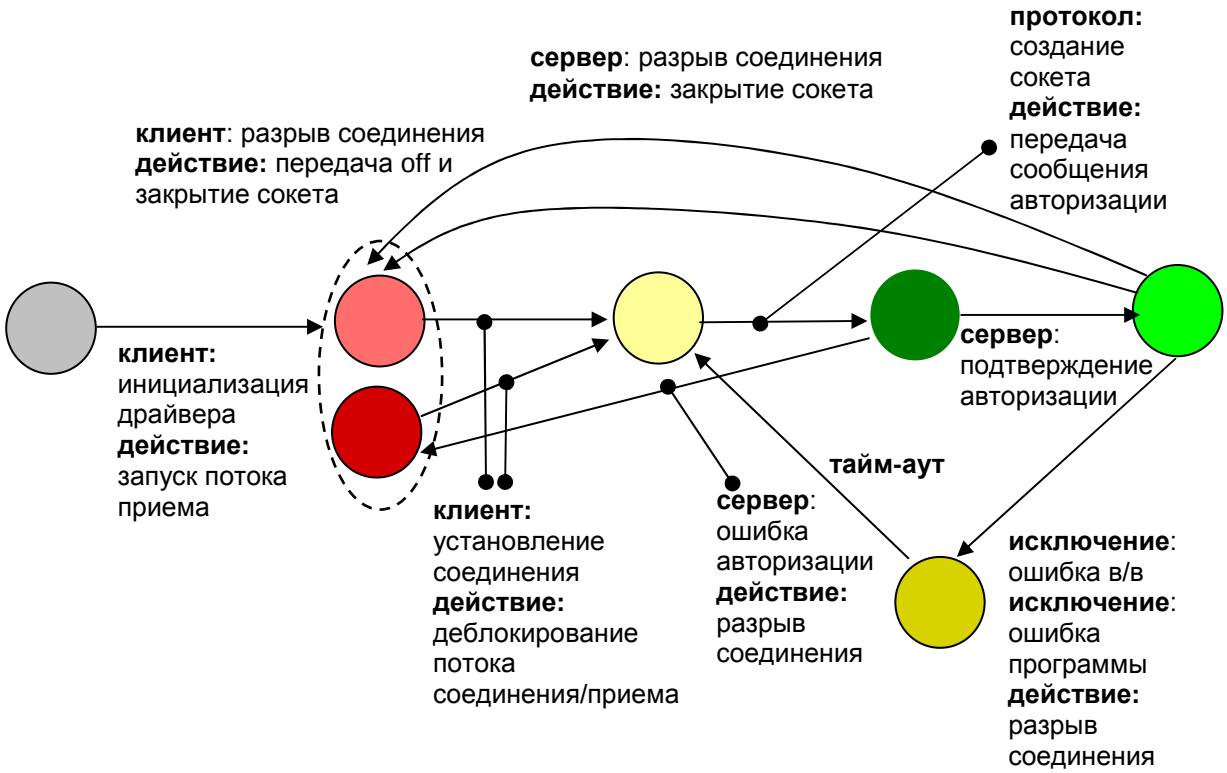

Рис.4.2. Диаграмма состояния соединения с сервером (прикладной уровень)

В случае, если конечный автомат используется для описания состояния обоих сторон соединения (например, диаграмма состояний для процедур установления и разрыва соединений TCP/IP), то фактически речь идет о взаимодействии двух автоматов, которое может приводить к их взаимной блокировке, условия возникновения которой не

очевидны из самой диаграмм. Анализ корректности таких моделей – тема для отдельного обсуждения [17].

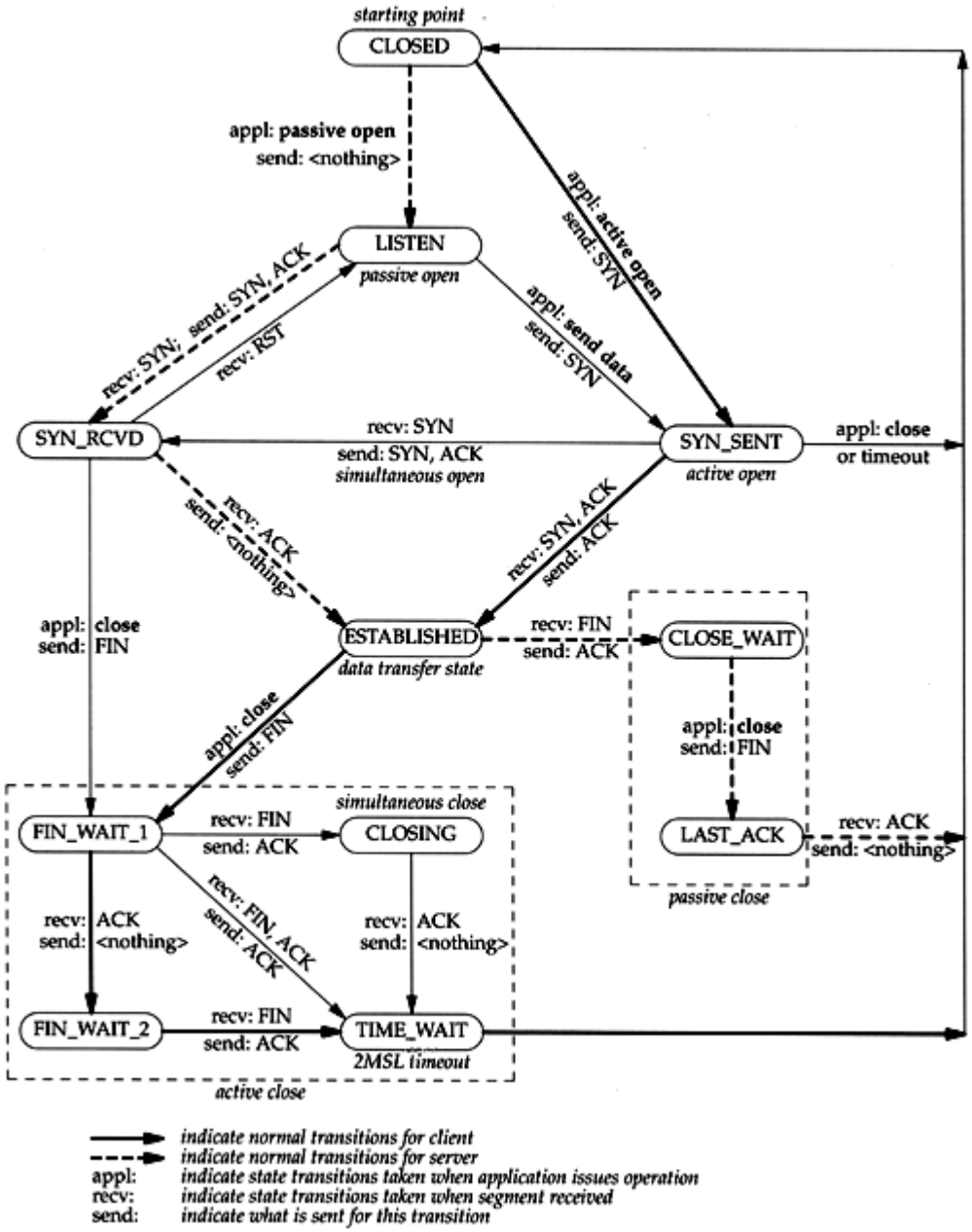

Рис.4.3. Диаграмма состояний TCP-соединения

Установление, восстановление и закрытие соединения, сессия, авторизация

В протокольном процессе могут использоваться собственные средства повышения надежности и эффективности соединения, например:

- при частых низкой разрывах соединения (например, мертвые зоны в мобильной связи, перегрузка сервера) в протоколе присутствует состояние «временно недоступен» с отложенной передачей запросов серверу. При этом с точки зрения клиента соединение считается работоспособным;
- при длительном или предполагаемом отсутствии трафика соединение закрывается с сохранением контекста у клиента и сервера, а при появлении трафика – восстанавливается без повторной авторизации;

- передача собственного keep alive, поскольку при отсутствии трафика в соединении «падение» одного из участников (например, крах операционной системы) или промежуточного звена не будут замечены остальными достаточно долго.

Процедура авторизации клиента на сервере также может быть составной частью протокольного процесса, а не вышележащих уровней, т.к. их интерфейсы могут быть прозрачны по отношению к локальному и удаленному соединению. К тому же состояние доступности сервера для протокола определяется не только создание соединения, но и успешной авторизацией.

В протокольный процесс также может быть включен механизм сессий. Если последовательность исполнения запросов логически связана между собой, т.е. на сервере сохраняются некоторые данные, связанные с исполнением запросов (начиная с учетных данных авторизации), то при закрытии соединения удобно сохранять его данные в течение некоторого времени с возможностью быстрого восстановления без авторизации с сохранением накопленного контекста на сервере.

Шаблон проектирования «сессия» предполагает, что при авторизации клиент и сервер получают и запоминают уникальный идентификатор сессии. При восстановлении соединения клиент может предъявить идентификатор сессии для получения доступа к старому контексту.

Канальная и прикладная компоненты протокольного процесса

Прикладной протокол поддерживает множество примитивов взаимодействия клиент-сервер. При этом происходит обмен сообщениями, формат которых соответствует этим примитивам. Если в протоколе принимается **единый формат** представления всех сообщений и единый способ их передачи, то функции оформления упаковки запросов/ответов в сообщения, их буферизация и непосредственная передача в сетевое соединение может быть реализована **канальной компонентой** (**линейным драйвером**). Это позволяет разделить содержательную и транспортную компоненты протокола.

Стандартным способом унификации с использованием технологии ООП является создание отдельного класса сообщений для линейного драйвера:

- сообщение содержит код, соответствующий команде или ответу;
- специальные коды сообщений могут использоваться для общепринятых ответов (положительное завершение без параметров, ошибка, исключение в серверной компоненте) и для собственных нужд протокола;
- сообщение может содержать несколько полей для наиболее часто используемых типов параметров;
- преобразование запросов/ответов в сообщение может происходить по-разному:
    - параметры запросов/ответов передаются как данные класса сообщений;
    - запросы/ответы являются производными классами, которые сериализуются при передаче и приеме;
    - объекты запросов-ответов «прицепляются» (делегируются) к объекту класса сообщения и сериализуются по ссылке вместе с ним, тогда в протоколе прикрепленный объект следует за основным как отдельное сообщение.

Процедурная и объектно-ориентированная реализация протокола

Обычная реализация набора примитивов взаимодействия выглядит следующим образом:

- команды и ответы примитивов кодируются уникальными кодами;
- клиентская компонента протокола для каждой команды создает сообщение с соответствующим кодом, заполняет параметры и передает линейному драйверу;
- серверная компонента, приняв сообщение, выполняет переключатель (**switch**) по коду сообщения, в каждой ветке переключателя пишется код обработки;
- в зависимости от результата исполнения запроса серверная компонента создает сообщение с соответствующим кодом;
- клиентская компонента при приеме ответа выполняет ветвление (или переключение) по коду ответного сообщения, в каждой ветке пишется свой код завершения.

Такая реализация является немодульной, поскольку код обработки всех запросов «свален в одну кучу», то же самое касается обработки ответов. Альтернативным решением является использование технологии ООП:

- все сообщения имеют общего предка – базовый класс, содержащий данные протокола, необходимые для работы с потоком сообщений, например, они могут последовательно нумероваться, если есть необходимость сопоставлять запросы и ответы при выполнении клиентом множественных запросов;
- выполняется принцип – «уникальный запрос/ответ – производный класс»;
- в классе сообщения имеются полиморфные методы onServer и onClient, которым передается необходимый контекст для обработки сообщения на сервере (запрос) и на клиенте (ответ), в них прописывается соответствующий код обработки сообщения.

Таким образом, работа прикладной компоненты протокола состоит в передаче клиентом объекта класса, соответствующего запросу, в его методе onServer прописана процедура его обработки, создания и передачи объекта класса, соответствующего ответу. В методе ответного сообщения onClient прописана процедура обработки ответа на клиенте. Код становится максимально модульным, сам протокол необходимо снабдить диаграммой классов с указанием зависимостей «порождения» одних классов другими.