1.3. Распределенные системы. Компоненты и взаимодействия

Распределенные системы

Рассмотренная выше клиент-серверная архитектура является самой популярной, но далеко не единственной **распределенной системой.**

Термин **распределенная система** имеет разные толкования. В широком смысле **распределенная система –** система, элементы которой распределены по пространственно разнесенным **узлам.** Т.е. констатируется лишь факт наличия узлов, разнесенных в пространстве.

Если же рассматривать функциональность системы, то исполнение отдельной ее функций может быть как **сосредоточенным** в одном узле, так и распределенным. В последнем случае можно говорить о **функционально распределенной системе.** Если же некоторая функциональность системы сосредоточена в одном узле, то это не функционально распределенная система, а система **с дистанционным (удаленным) доступом.**

И, наконец, если говорить о функциях администрирования, управления и контроля, то можно определить **систему с распределенным управлением –** распределенная система, в которой административные функции управления и контроля функционально распределены по узлам.

Поэтому, прежде чем говорить о распределенной системе и ее «видовой принадлежности», надо описать ее компоненты, распределение функциональности и управления. Рассмотрим несколько примеров.

Компьютерная сеть TCP/IP является функционально распределенной системой и системой с распределенным управлением, поскольку в каждом узле реализуются функции предоставления доступа на уровне логического соединения и сеть не имеет централизованного управления и администрирования.

На прикладном уровне **семейство протоколов HTTP/FTP/SMTP** реализует дистанционный доступ к соответствующим ресурсам (web-сервисам, файлам).

Термины **распределенная операционная система, распределенная файловая система, распределенная БД** справедливы, если основные ресурсы, предоставляемые системой (процессы - операционная среда, файлы, базы данных) не привязаны к конкретному узлу. Для таких систем характерно:

- пользователь не видит отдельных узлов системы, а имеет дело с единым логическим представлением;
- ресурсы (файлы, таблицы БД) могут распределяться по узлам динамически, произвольным образом, иногда ресурс делится на части и размещается по различным узлам;
- имеет место распределенное управление ресурсами.

В противном случае опять-таки имеет место группа систем с дистанционным доступом.

Централизованная система клиенты-сервер».

На практике большинство простых систем и систем средней сложности представляют собой сервер с уникальной функциональностью и набор разнообразных клиентов, связанных с сервером единым протоколом.

Клиент-сервер: роли и программы, взаимодействия

Распределенная система состоит из взаимодействующих процессов. Сама компьютерная сеть не налагает ограничений на характер обмена данными (и взаимодействия) процессов, т.е. процессы могут передавать данные в любое время по любым соединениям. Однако на практике для каждой пары процессов временно или постоянно устанавливаются роли «ведущий-ведомый» или «клиент-сервер». Термин «клиент» подразумевает не обязательно конечного пользователя, а имеет в виду **инициатора взаимодействия.** Важным является то, что в клиенте происходит **событие,** которое вызывает взаимодействие в распределенной системе, и момент его наступления серверу не известен.

В процессе взаимодействия функциональные компоненты могут играть роль клиента или сервера постоянно, могут менять свои роли, создаваться динамически для соответствующих ролей т.п.. Поэтому следует различать **программные компоненты клиент-сервер** и **роли клиент-сервер**, которые обычно закреплены за отдельными потоками в этих программах.

Например, в протоколе FTP при выполнении основных команд роли клиента и сервера распределены естественным образом. При передаче файлов данных и содержимого директорий в пассивном режиме FTP-сервер выступает в роли сервера, а в активном – в роли клиента (а программа-клиент – в роли сервера). В программе-клиенте для этих целей, как правило, создается отдельный поток.

Взаимодействие клиента с сервером может иметь разную степень сложности. В простейшем варианте это **вызов функции – синхронный ответ.** Клиент инициирует выполнение действий на сервере и дожидается результата исполнения. В программной реализации это соответствует обычному вызову функции или метода в объекте-сервере.

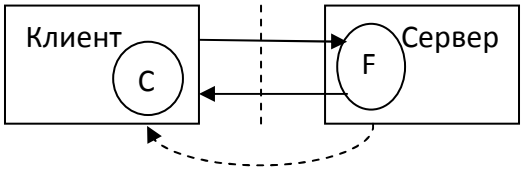

рис.1.5. Синхронный запрос - ответ

Для последующего изложения также важен тот факт, что клиент передает серверу данные своего контекста (C), в котором он выполняется и с которым сервер может работать.

Если ожидание завершения действия клиентом является неприемлемым, то используется схема **вызов функции – асинхронный ответ.** Как правило, исполнение действий в сервере осуществляется с использованием внутреннего параллелизма (см.2.4 - потоки), а ответ возвращается в вызывающий код с помощью **обратного вызова** (см. 2.4, асинхронный обратный вызов). Вызывающий код создает м передает серверу объект-слушатель (L), который выполняет функцию (метод) обратного вызова в контексте вызывающего кода. Также требуются средства синхронизации обратного вызова с потоком вызывающего кода (S). В некоторых языках (**Scala**) функция обратного вызова может быть передана серверу непосредственно, вне объекта.

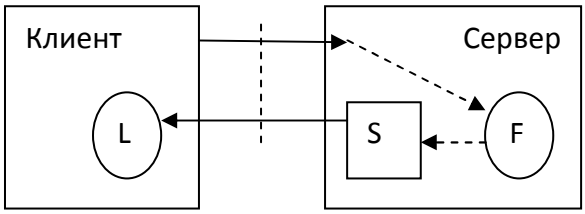

рис.1.6. Асинхронный ответ с обратным вызовом

Развивая схему асинхронного ответа, можно реализовать сервер, который однократно инициируется клиентом, после чего клиент может выполнять в нем команды-функции, получая асинхронные ответы на них, а также на события, происходящие на сервере. Т.е. сервер становится относительно автономной службой с асинхронным внутренним поведением.

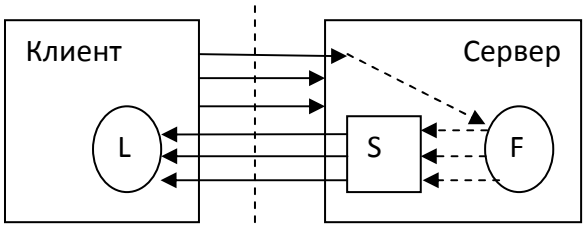

рис.1.7. Множественный асинхронный ответ (асинхронная служба)

В программной реализации роль сервера часто исполняет объект соответствующего класса, в котором код-клиент вызывает соответствующие методы и получает ответы в виде прямого результата, либо в виде обратных вызовов в объекте-слушателе событий.

Интерфейсы и протоколы

Способ взаимодействия между программными компонентами распределенной системы зависит от их взаимного расположения. Если взаимодействие локальное, то оно осуществляется через **программные интерфейсы** (непосредственные вызовы, обращения к службам). Если клиент и сервер расположены в различных узлах сети, то взаимодействие между компонентами осуществляется посредством **протокола.** Далее (2.1) будут рассматриваться все нюансы этого термина. Пока достаточно будет определения:

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

В принципе, компоненты протокола могут быть включены в программный код клиента и сервера. Если же следовать принципам модульности и повторного использования кода, то исходные программные интерфейсы взаимодействия следует сохранить, а протокольные – реализовать отдельно. В схеме появляются две компоненты протокольных процессов, с которыми у клиента и сервера сохраняются исходные интерфейсы.

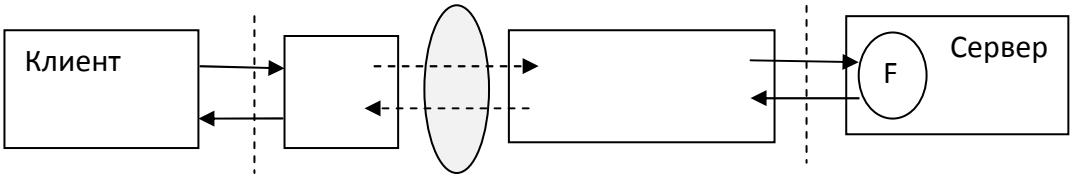

рис.1.8. "Расшивка" локального интерфейса

Чтобы сделать удаленное взаимодействие клиента и сервера окончательно прозрачным, в протокольном процессе сервера создается **агент,** который является клиента, от имени которого выполняется локальное взаимодействие с сервером. При этом агент создает аналогичный контекст или получает его содержимое от удаленного клиента.

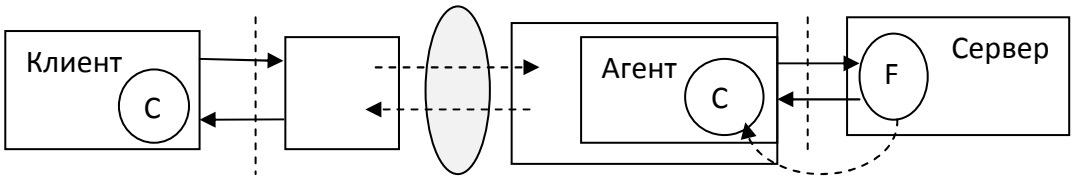

рис.1.9. Клиент и агент протокола

Постоянные соединения и транзакции

При проектировании распределенной системы, прежде всего, необходимо определить характер соединений между его компонентами.

Короткие соединения – транзакции. Наиболее просто реализуется взаимодействие, если оно происходит в виде короткого сеанса обмена сообщениями без значительных пауз, что имеет свое наименование – **транзакция.** При этом сервер может вообще не использовать многопоточность – клиенты выстраиваются к нему в очередь еще на уровне установления соединения (серверный сокет поддерживает очередь запросов на создание соединения). Однако эта простота имеет существенный недостаток – сервер обычно не сохраняет данные о транзакциях, поэтому клиент обязан каждый раз передавать весь набор параметров, начиная с логина/пароля.

Постоянные соединения. В постоянных соединениях сервер может сохранять данные о текущем состоянии клиента, создавая для него некоторый **текущий контекст,** в котором клиент работает (начиная с данных авторизации и т.д.). Эти данные, а также все остальные данные о соединении, хранятся в объекте-потоке, таким образом, сервер является многопоточным. Сами же потоки требуют синхронизации при работе с общими данными серверного приложения.

Другим недостатком постоянного соединения является необходимость проверки его работоспособности – keep-alive (**см.2.2.**). Если это не сделано в самом соединении, то это реализуется на прикладном уровне передачей пустых сообщений, подтверждающих работоспособность соединения. Серверное приложение может закрывать соединения, в которых в течение заданного интервала нет активности. Например, MySQL-сервер делает это, поэтому клиенту рекомендуется периодически посылать в него фиктивные SQL-запросы типа SELECT 1.

Постоянные соединения с ограниченным временем (периодически засыпающие). Значительное количество постоянных соединений может сопровождаться существенными затратами на их поддержание (кроме самих соединений, например, на связанные с ними потоки - thread). Если в течение некоторого времени к серверу нет

обращений, то клиент может временно закрыть соединение. Например, в клиенте может работать поток, который по истечении тайм-аута закрывает соединение, устанавливая необходимый признак. Очередной запрос к серверу, проверяя этот признак, восстанавливает соединение перед выполнением обращения.

Восстанавливаемые соединения. С установленным логическим соединением в протокольных процессах прикладного уровня могут быть ассоциированы некоторые данные. Сервер может кэшировать считанные клиентом данные, создавая для него **текущий контекст**, в котором тот работает. При аварийном разрыве соединения сервер уничтожает поток, контекст клиента, и весь диалог необходимо начинать сначала. Чтобы этого избежать, на прикладном уровне необходимо ввести механизм сессий. При первичном установлении соединения и авторизации клиента сервер передает клиенту уникальный идентификатор (handle) сессии. При аварийном закрытии соединения сервер в течение некоторого времени хранит контекст клиента. Повторное соединение при восстановлении производится клиентом по старому handle. Значения handle должны генерироваться последовательно в таком диапазоне, чтобы избежать повторения значения handle при самом длительном соединении.

Стратегии взаимодействия клиент-сервер

В приведенных выше схемах программа-клиент всегда выступает в роли клиента, а сервер – сервера. Эта тавтология может быть нарушена, если сервер перестанет в процессе взаимодействия выступать в роли пассивного исполнителя запросов клиента. Для этого возможны различные предпосылки, например:

- клиенты интенсивно взаимодействуют между собой;
- оперативное обновление у клиента изменяемых данных сервера;
- сервер демонстрирует достаточно сложное собственное поведение;
- клиенту сложно отфильтровать изменения данных на сервере, которые касаются лично его;
- параметры объектов, циркулирующих в системе, могут обновляться как сервером, так и клиентом, необходима синхронизация изменений.

В таких случаях стандартная схема может оказаться неэффективной, привести к необоснованному увеличению трафика или снижению реактивности системы. На этапе архитектурного проектирования необходимо рассматривать различные стратегии обмена данными между сервером и клиентами. Приведем некоторые из них.

Синхронные запросы клиента к серверу. Единицей взаимодействия в паре программ клиент-сервер является передача команды (запроса) клиента с ожиданием ответа сервера. Технологическим вариантом такого взаимодействия является удаленный вызов процедур (RPC –Remote Procedure Call), когда клиент передает серверу имя и параметры вызываемой процедуры и получает в ответ результаты ее выполнения на сервере;

Циклический опрос состояния сервера (polling). Клиент периодически посылает серверу команды, на которые он отвечает набором данных о событиях, произошедших на сервере и касающихся клиента. Опрос может быть сделан более эффективным за счет повышения частой опроса при увеличении трафика и постепенном его снижении во время паузы;

Отложенный опрос состояния сервера (long polling). При отсутствии данных сервер может задерживать ответ вплоть до установленного значения тайм-аута (например, до 30 сек). Если же данные отсутствуют, то возвращается пустое сообщение. Аналогично сервер может задерживать ответ до накопления нужного количества данных с целью снижения частоты опроса. Недостатком отложенного опроса является необходимость его поддержки отдельным соединением;

Двунаправленная система передачи асинхронных сообщений. Клиент и сервер имеют независимые наборы передаваемых сообщений, на каждое из которых на принимающей стороне предусмотрена обрабатывающая его компонента. В этом случае роли клиент-сервер не закреплены на уровне отдельных сообщений (команд), при необходимости клиент должен самостоятельно идентифицировать ответные сообщения сервера, полученные в ответ на передаваемые;

Двунаправленная система передачи синхронных сообщений. Наиболее сложный вариант, в котором как клиент, так и сервер имеют собственный набор команд, которые обрабатываются противоположной стороной, т.е. имеется две пары связок клиент-сервер. Для идентификации и разделения сообщений из разных связок создается специальный подуровень протокола обмена;

Асинхронное обновление сервером состояния клиента. Для каждого клиента сервер отслеживает изменение состояний объектов, которые касаются только данного клиента, и

с требуемой периодичностью выполняет команды обновления клиента, передавая ему соответствующие данные.

В качестве примера рассмотрим службу заказа такси, имеющую несколько видов клиентских приложений:

- desktop-приложение диспетчера;
- web-приложение или мобильное приложение заказчика;
- мобильное приложение водителя.

Больше всего вопросов возникает к последнему клиенту, т.к.

- оперативное представление данных о свободных заказах с учетом местоположения водителя (автомобиля), установленного им радиуса (т.н. радара), а также других параметров;
- оперативное предоставление сервером заказов «на голосование» водителю;
- периодическая передача данных о местоположении водителя;

Наиболее проблемным с точки зрения реактивности и объема трафика является приложение водителя, к которому предъявляются еще и дополнительные требования:

- значительное количество клиентов, особенно при обслуживании сервером нескольких диспетчерских;
- кэширование данных для сохранения частичной работоспособности приложения при потере сетевого соединения, синхронизация изменений (проведение заказа) при восстановлении связи.

Для снижения трафика при высокой реактивности системы можно предложить следующие стратегии взаимодействия приложений с сервером:

- формирование данных сервером для событий, касающихся данного водителя: предоставление заказов «на голосование», изменение списка свободных заказов с учетом местоположения водителя, состояния исполняемых заказов и сообщения водителю. Очередная порция данных формируется относительно последней переданной;
- отложенный опрос сервера для вышеуказанных данных;
- синхронные запросы для остальных команд;
- двунаправленная система передачи синхронных или асинхронных сообщений с указанной выше стратегией работы сервера.