1. Важнейшие типы архитектур

Итак, как говорилось ранее, в настоящее время накоплен значительный успешный опыт в разработке распределенных приложений, который позволяет говорить о наличии некоторых типов типовых архитектур, некоторые из которых будут рассмотрены в этом разделе.

Клиент-сервер

Первый тип, который будет нами рассмотрен - архитектура "клиент - сервер" (Рис. 1.1). Это в настоящее время наиболее распространенная архитектура, в которой выполнено, пожалуй, большинство работающих информационных систем. Существует даже мнение, что почти все остальные архитектуры могут быть представлены как большая или меньшая вариация этой, базовой.

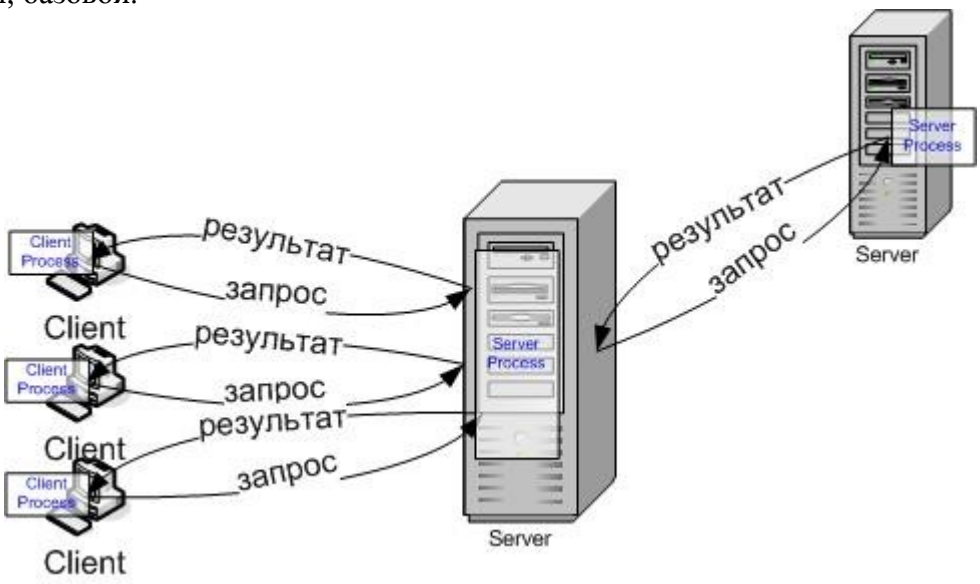

Рис. 1.1. Архитектура "клиент - сервер"

В традиционном понимании система, выполненная в архитектуре клиент-сервер, представляет собой совокупность взаимодействующих компонент двух типов - клиентов и серверов. Обычным также является разнесение этих компонент по узлам двух типов - соответственно узлам - клиентам и узлам - серверам. Клиенты обращаются к серверам с запросами, сервера их обрабатывают и возвращают результат. Клиент, вообще говоря, может обращаться с запросами к нескольким серверам. Сервера также могут обращаться с запросами друг к другу. Таким образом, типичный протокол для одного факта взаимодействия может быть представлен в виде двух обменов - запрос на сервер и ответ сервера.

Наиболее часто встречающийся класс приложений, выполненных в архитектуре клиент-сервер - различные приложения, работающие с базами данных. В таком случае в качестве сервера выступает СУБД, обеспечивающая выполнение запросов клиента, который в свою очередь реализует интерфейс пользователя.

Рассмотренные далее модели систем, по сути, являются вариациями архитектуры

клиент – сервер.

Модель сервиса (один сервис - много серверов)

Модель сервиса (Рис. 1.2) представляет собой реализацию ситуации, при которой одну услугу реализует не один, а несколько серверов, представляемых клиенту как единое целое. Строго говоря, предложенная модель является чистым клиент-сервером, у которого сервер имеет сложную структуру (не монолитен). Этот вариант особенно хорош для критичных сервисов, для которых недопустим приостанов обслуживания. Поскольку сервис реализуется сразу несколькими серверами, останов одного из них не является критическим[1]. Для прекращения обслуживания клиентов необходим останов всех серверов системы. Кроме того, такая схема позволяет реализовывать различные стратегии балансировки нагрузки между серверами системы.

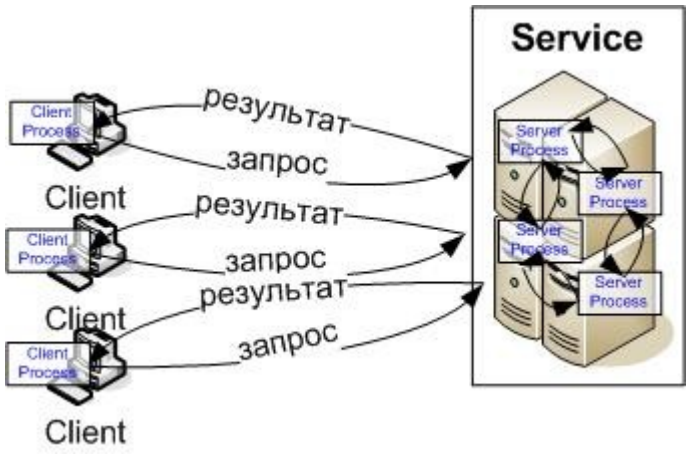

Рис. 1.2. Модель сервиса

Таким образом, может быть существенно увеличена как производительность системы, так и ее устойчивость к сбоям. Однако наряду с плюсами у предложенной модели есть и минусы - самый большой - более сложная реализация по сравнению с базовой архитектурой. В самом деле, поскольку все сервера обеспечивают один и тот же сервис, возникают проблемы либо с репликацией обрабатываемых данных (поддержание на всех серверах системы актуальной копии данных), либо с поддержанием целостности распределенных данных.

Технология подключения через proxy

Примером системы, построенной в такой архитектуре, является привычная система из браузера, прокси-сервера и веб-сервера.

Отличием от ранее рассмотренных моделей является то, что клиент соединяется не с сервером, а с неким промежуточным компонентом (посредником) (Рис. 1.3). Этот посредник, в свою очередь, от имени клиента передает запрос серверу. При этом посредник может сам решать, на какой из серверов послать запрос (возможны стратегии балансировки нагрузки). Кроме того, он может поддерживать кэш последних задаваемых запросов и при очередном запросе пользователя вернуть ему ответ, не обращаясь к серверу[2].

---

[1] Здесь уместно вспомнить про кластеризацию

[2] Это возможно только в специальных случаях, а вообще говоря, посредник обязан обладать механизмом

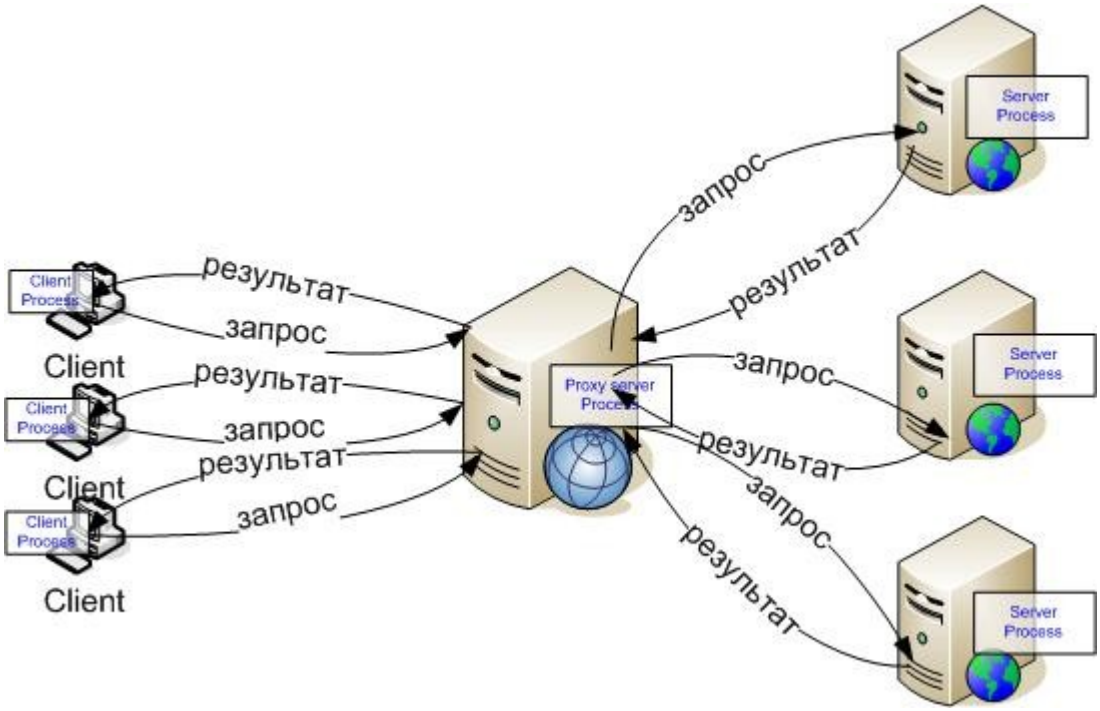

Рис. 1.3. Технология подключения через proxy

Следует отметить, что при неправильном построении посредник может стать узким местом всей системы.

Как уже говорилось ранее, этот подход особенно часто применяют при работе в Internet.

Сервер инициирует соединение

В классической архитектуре клиент-сервер инициатором диалога всегда выступает клиент. Можно, однако, представить и другую ситуацию - диалог инициирует сервер "проталкивая" информацию на клиента. Роль клиента в таком случае сводится к реакции (просмотру) на сообщения сервера. Типичным примером является работа "по подписке". Представим себе, что сервер получает какие-то события из внешнего источника. События имеют тип. Клиент, заинтересованный в получении события определенного типа сообщает о своей заинтересованности серверу. Сервер, получив очередное событие, передает его всем заинтересованным в нем клиентам. Приведенный алгоритм является упрощенным описания работы очень часто используемой службы событий (подобная служба есть практически во всех современных middleware).

поддержания кэша в состоянии непротиворечивом с данными сервера.

Мобильные агенты

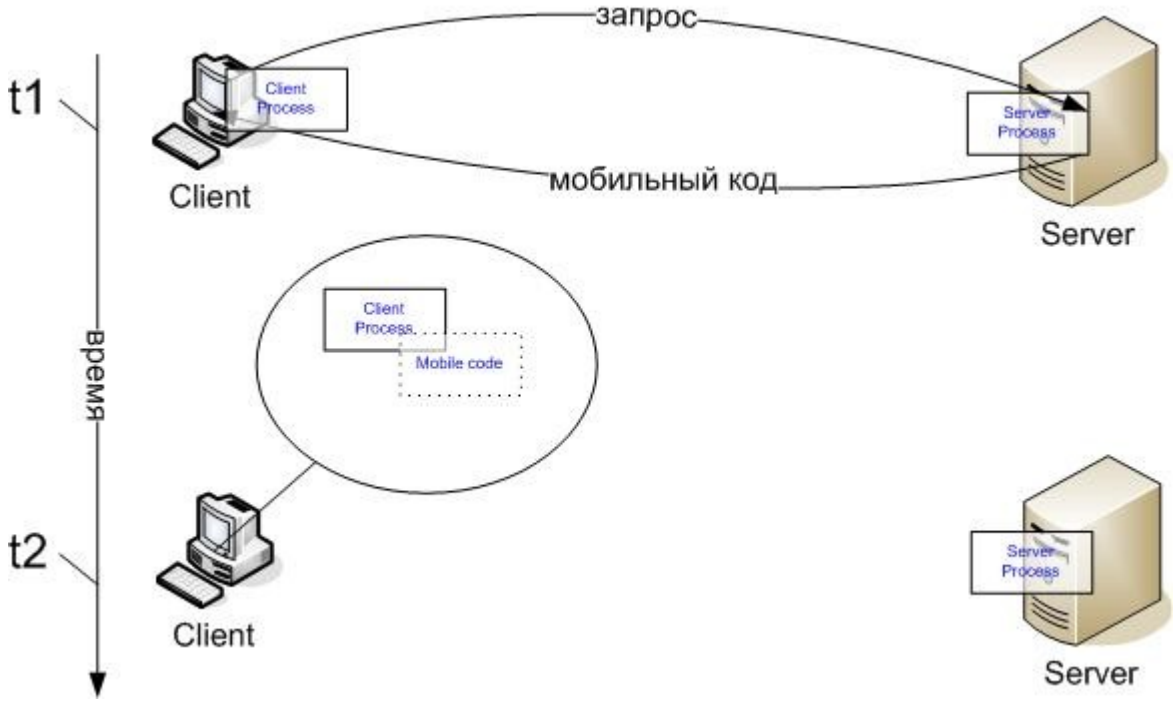

Рис. 1.4. Мобильные агенты

Идея рассматриваемой модели (Рис. 1.4) состоит в том, что зачастую, как это не парадоксально, клиент сам в состоянии выполнить ту задачу, решение которой он запросил у сервера, более того, данные необходимые для решения этой задачи располагаются на клиенте. В таком случае, для разгрузки сервера (и, очень часто, для снижения сетевого трафика) целесообразно решать эту задачу на клиенте. Но как это сделать, если у клиента нет соответствующего программного модуля, содержащего необходимую функциональность? Ответ таков - этот модуль нужно клиенту отправить. Клиент, получив модуль (этот модуль называется мобильным агентом) может выполнить его локально, решив, таким образом, задачу. В качестве примера можно рассмотреть взаимодействие браузера и веб-сервера, возвращающего страницу, содержащую аплет. Аплет также передается клиенту и выполняется в браузере (т.е. на клиенте), выполняя какие-то значимые для пользователя действия. Основная проблема для такого подхода состоит в сложности реализации механизма передачи и выполнения мобильных агентов, а также контроля безопасности[3]. Однако, современные средства middleware (Java, например) такими возможностями обладают.

Тонкий клиент

В течение нескольких последних лет наблюдается постоянное увеличение количества применяемых портативных устройств - сотовых телефонов, PDA, и т.д. Возникает

[3] По поводу сложности обеспечения безопасности для исполнения мобильных агентов - большинство почтовых вирусов, представляющих собой код на VB, внедренный в тело письма и выполняющийся в почтовой программе - по сути являются мобильными агентами.

естественное желание использовать такие устройства как средства для работы с информационными системами - почему бы, например, не зайти wap-сайт туристического агентства и не заказать путевку прямо с сотового телефона? Поскольку в большинстве современных телефонов встроен wap-броузер, в этом нет ничего невозможного.

Однако, интерфейс, предоставляемый браузерами, весьма ограничен. С другой стороны, в силу ограниченной мощности мобильных устройств в них пока не удается размещать приложения со сложной бизнес-логикой.

Рис. 1.5. Тонкий клиент

Решить эту проблему можно используя технологию "тонкого клиента" (Рис. 1.5). Суть этой технологии состоит в том, что клиент выполняет очень ограниченную по функционалу задачу (очень часто - только прием ввода с клавиатуры и других устройств и обработка команд рисования). Схема работы подобных систем в простейшем случае следующая. Клиентская программа передает весь ввод пользователя (нажатия клавиш, движение мыши и т.д.) по сети серверу. Сервер, разбирает и обрабатывает этот ввод и передает клиенту готовые экраны, которые тот просто отображает[4]. Этот принцип используется в системах типа X-Windows уже очень давно. Таким образом, сервер фактически берет на себя не только задачу управления данными, но и вообще все задачи по логике клиентского интерфейса.

Архитектура P2P (Peer - to Peer)

Другой архитектурой, встречающейся в основном в специальных областях, является архитектура P2P (Рис. 1.6). Приложение, выполненное в такой архитектуре, не имеет четкого разделения на "серверные" и "клиентские" модули - все его части равноправны и могут выполняться на любых узлах.

[4] Реальный протокол конечно сложнее. Если передавать все события на сервер, а с сервера передавать целые изображения экранов это вызовет очень большой сетевой трафик, и, в конечном итоге, к низкой скорости работы системы. В реальности события пользователя предобрабатываются на клиенте и серверу передаются только необходимые, сервер в свою очередь передает лишь команды *изменения* экрана пользователя.

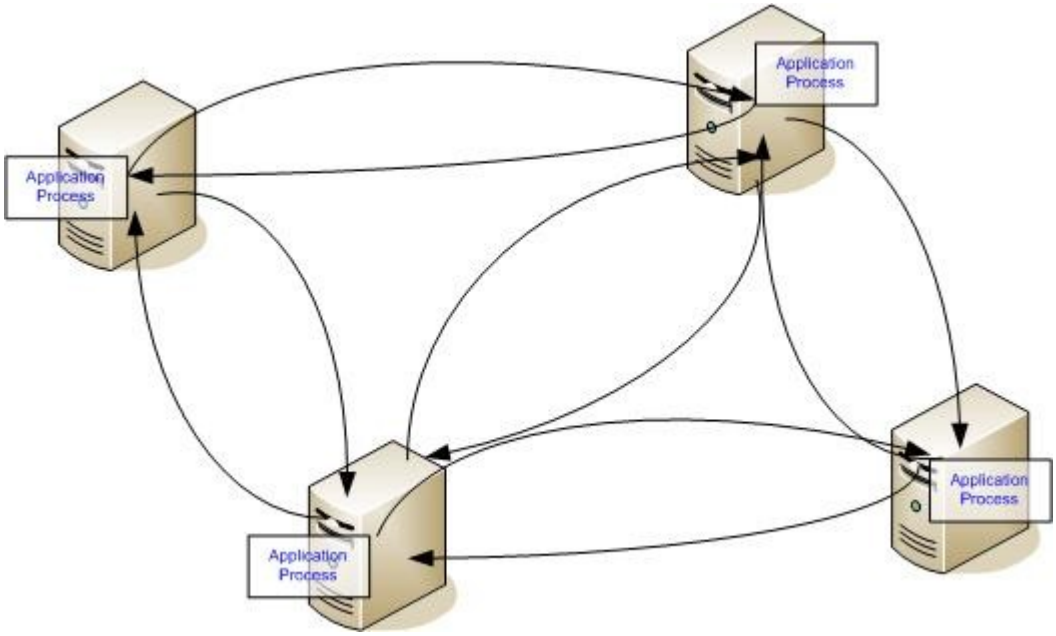

Рис. 1.6. Архитектура P2P

Таким образом, на одном и том же узле в один момент выполняются части системы, обрабатывающие запросы других частей (и узел выполняет "серверную" часть), а в другой момент времени - части системы, посылающие запросы (и узел выполняет "клиентскую" часть). Причем, приложение может быть устроено так, что вызывающая часть не знает, локально или удаленно расположена вызываемая. Построенные таким образом приложения обладают уникальными свойствами - его части никак не привязаны друг к другу и к узлам, на которых они исполняются. Таким образом, от запуска к запуску может меняться состав модулей, расположенных на узле. Это позволяет организовывать очень изощренные политики распределения нагрузки, а также обеспечивать очень хорошие показатели масштабируемости и отказоустойчивости. Такая архитектура активно используется для разработки параллельных вычислительных систем для решения сложных вычислительных задач. Например, базовая техника при программировании с использованием технологии MPI состоит в том, что на нескольких узлах запускается *одна и та же программа,* которая, однако, ведет себя иначе (срабатывают различные ветви условных переходов), в зависимости от порядкового номера машины, на которой она запущена. Совокупность этих процессов представляет собой распределенную систему, решающую поставленную задачу.