Содержание:

Введение

Индустрия разработки автоматизированных информационных систем появилась в 1950-х – 1960-х годах. Область применения ИС с тех пор постоянно расширялась, а сами они становятся все сложнее. Многие ИС вырастают и усложняются настолько, что приобретают глобальный характер, и от их правильного и надежного функционирования начинает зависеть деятельность десятков или даже сотен тысяч людей. В силу требований глобальности, необходимо обеспечивать взаимодействие из территориально разнесенных между собой точек.

Но, не следует считать, что распределенные системы - изобретение последних лет. Три-четыре десятилетия назад была популярной модель "хост-компьютер + терминалы". Реализованная на мэйнфреймах, например IBM-360/370 или их советских аналогов - компьютеров серии ЕС ЭВМ.

Рост индустрии персональных компьютеров сперва ничего не изменил в идеологии построения информационных систем. Как и ранее, в большинстве своем программы имели дело с локальными ресурсами (память, дисковое пространство, локальные периферийные устройства). Но, надо отметить, что некоторая часть этих ресурсов была уже "псевдолокальной", например, файлы на сетевом диске. Однако по-прежнему файл обрабатывался непосредственно самим узлом, предварительно скачиваясь по сети.

Начали выявляться проблемы: блокировка ресурсов, поддержание логической целостности для вносимых изменений, резкое увеличение объемов сетевого трафика и т.д. Стало очевидно, что необходимо пересматривать подходы к проектированию и реализации информационных систем.

С определенным увеличением объема перерабатываемых данных, а также по мере возрастания их стоимости, стало ясно, что доверять их обработку клиентским машинам нежелательно. Любая ошибка на них (а чем больше клиентов, тем больше вероятность ошибки) приводит либо к потере данных, либо к их блокировкам в процессе работы, а, стало быть, к снижению общей производительности системы. Следующим ключевым шагом стало распространение идеологии клиент-серверной обработки. Это были "двух-ролевые" системы: клиент несет ответственность за

отображение пользовательского интерфейса и выполнение кода приложения, а роль сервера обычно поручалась СУБД. Сервер обрабатывает запрос и возвращает клиенту результат, который, к примеру, отображается на экране или выводится на печатающее устройство.

Переход на технологию "клиент-сервер" решал много старых проблем, но и создал много новых. Одна из основных трудностей - проведение границы между функционалом клиента и сервера. Часто решение о переносе части задач на сервер пагубно сказывается на общей производительности системы, и наоборот, перенос части нагрузки на клиента может привести к потере централизации и связанных с этим преимуществ.

Архитектура “клиент-сервер” определяет общие принципы организации взаимодействия в сети, где имеются сервера, узлы-поставщики специализированных функций (сервисов) и клиенты, потребители этих функций (сервисов). Реализацию данной архитектуры называется клиент-серверной технологией. Каждая технология определяет собственные или использует имеющиеся правила взаимодействия между клиентом и сервером, которые называются протоколом обмена (протоколом взаимодействия).

По мере роста популярности систем "клиент-сервер" развивалась технология объектно-ориентированного программирования, которая предлагала следующую системную архитектуру: слой представления отводится пользовательскому интерфейсу, слой предметной области предназначен для описания основных функций приложения, необходимых для достижения поставленной перед ним цели, а третий слой представляет источник данных.

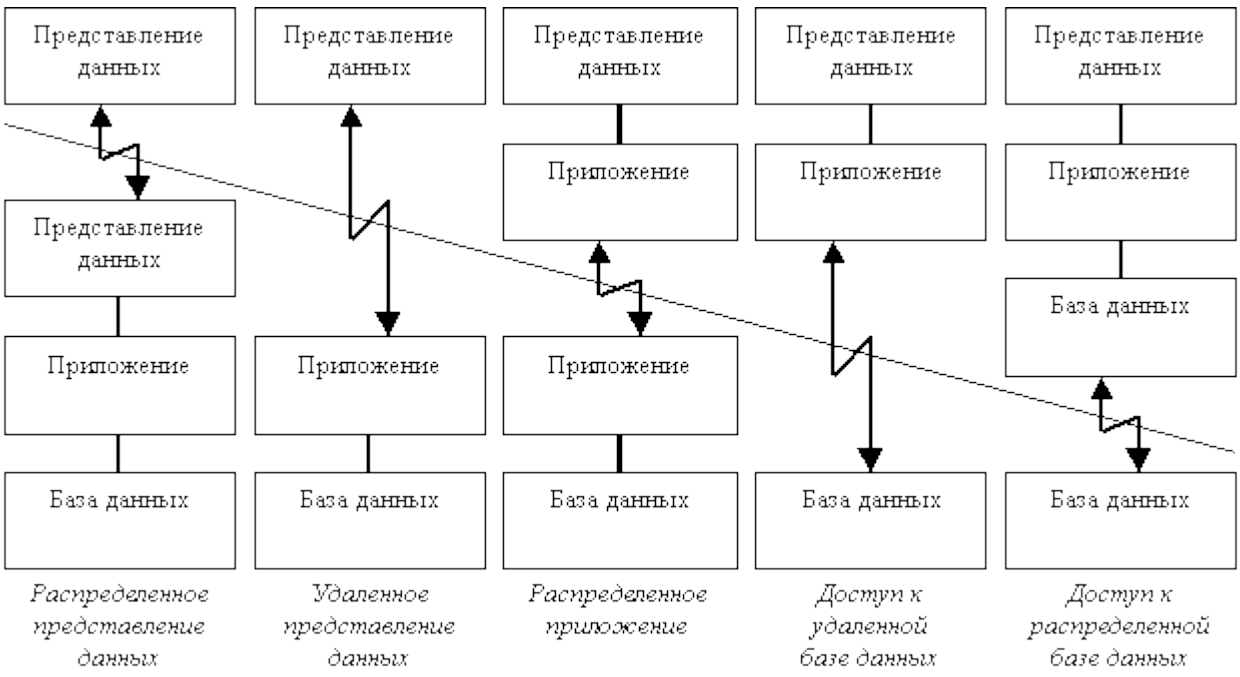
Развитие интернета привело к тому, что стала необходимостью архитектура “клиент-сервер”, где в качестве клиента выступает интернет-браузер. В настоящее время можно считать, что бум технологий, связанных с клиент-серверной архитектурой, все еще продолжается - множество работающих в настоящее время информационных систем выполнено в этой технологии. Однако актуальными являются направления, связанные с развитием этой идеи - так называемые трехслойные и многослойные, а также децентрализованные приложения, а двух-уровневые клиент-серверные приложения если еще остались, то только в тех местах, где их замена нецелесообразна или невозможна по различным причинам.

Глава I. Основные понятия клиент-серверной архитектуры.

1.1 Двухуровневая модель

Использование двухзвенной архитектуры клиент-сервер, подразумевает организацию сети и выполнение каким-либо компьютером в ней роли сервера. Компания Gartner Group предлагает следующие варианты двухзвенной архитектуры:

Рис.1. Варианты двухзвенной архитектуры по Gartner Group



Основные модели взаимодействия клиента и сервера в рамках двухзвенной архитектуры:

- файл-сервер — доступ к удаленной базе данных и файловым ресурсам;
- сервер терминалов — распределенное представление данных;
- сервер базы данных — удаленное представление данных;
- сервер приложений — удаленное приложение.

Первой была реализована модель сервера терминалов. Она базировалась на мэйнфрейме, выступавшего в роли сервера, с подключенными терминалами. Пользователи выполняли ввод данных с периферийный устройств (клавиатуры

терминала, устройства считывания перфокарт или перфоленты), которые затем передавались на мэйнфрейм и там выполнялась их обработка, включая формирование «картинки» с результатами. Эта «картинка» и возвращалась пользователю на экран терминала или на принтер.

С массовым распространением персональных компьютеров и локальных вычислительных сетей, была реализована модель файлового сервера, представлявшего доступ файловым ресурсам и к удаленной базе данных. В этом случае выделенный компьютер являлся файловым сервером, на котором были размещены файлы базы данных. На клиентах запускались приложения, использующие подключенную удаленную базу как локальный файл.

При росте количества пользователей такая модель архитектуры показала свою неэффективность. При активной работе с таблицами БД возникает большая нагрузка на локальную сеть, так как к каждому пользователю отправлялась вся таблица целиком при обращении к базе данных. А так как таблиц было не одна и не две, то все это и увеличивало нагрузку. Также большой проблемой стало внесение одновременных изменений полученных от разных пользователей в один момент времени.

Вся тяжесть вычислительной нагрузки при доступе к БД ложится на приложение клиента, так как при выдаче запроса на выборку информации из таблицы вся таблица БД копируется на клиентскую машину и выборка осуществляется на клиенте. Таким образом, очень не оптимально расходуются ресурсы и сервера и клиентской машины. В результате возрастает сетевой трафик и увеличиваются требования к аппаратным мощностям пользовательского компьютера, так как базы данных могут содержать в себе таблицы с большим объемом данных, что приводит к переполнению оперативной памяти локальных станций, к переполнению места на диске для временных файлов базы данных. Для решения данной проблемы, был разработан подход, работающий с отдельными строками (записями) или участками таблицы, а не со всей таблицей целиком.

В файл-серверной архитектуре, под базой данных зачастую понимался набор слабо связанных таблиц, и изменения в них можно было вносить из инструментальных средств (например, из Database Desktop фирмы Borland для файлов Paradox и dBase). Все это говорит о низком уровне безопасности с точки зрения внесения ошибочных изменений.

Транзакции, в рамках работы логики тех систем, служат потенциальным (и реальным) источником ошибок в плане логической и ссылочной целостности данных при одновременном внесении изменений в одну и ту же запись.

С проектированием и выходом на рынок специализированных СУБД появилась возможность реализации другой модели доступа к удаленной базе данных — сервера баз данных. При такой архитектуре решения, система управления базой данных запускается на сервере, прикладная программа на клиенте, а протокол обмена описан языком SQL (Structured Query Language). Такой подход к архитектуре, в сравнении с файл-серверным, уменьшает загрузку локальной сети. Также, унифицируется интерфейс между клиентом и сервером. Однако, сетевой трафик остается достаточно высоким, кроме того, по прежнему невозможно удовлетворительное администрирование приложений, поскольку в одной программе совмещаются различные функции.

С разработкой и внедрением на уровне серверов баз данных механизма хранимых процедур появилась концепция активного сервера БД. В этом случае часть функций прикладного компонента реализованы в виде хранимых процедур, выполняемых на стороне сервера. Остальная прикладная логика выполняется на клиентской стороне. Протокол взаимодействия — соответствующий диалект языка SQL. Преимущества очевидны:

- возможно централизованное администрирование прикладных функций;
- снижение стоимости владения системой (TOC, total cost of ownership) за счет аренды сервера, а не его покупки;
- значительное снижение сетевого трафика (т.к. передаются не SQL-запросы, а вызовы хранимых процедур).

Основной недостаток — ограниченность средств разработки хранимых процедур по сравнению с языками высокого уровня. Реализация прикладного компонента на стороне сервера представляет следующую модель — сервер приложений. Перенос функций прикладного компонента на сервер снижает требования к конфигурации клиентов и упрощает администрирование, но представляет повышенные требования к производительности, безопасности и надежности сервера.

1.2 Многоуровневая архитектура

Следующий шаг в развитии двухзвенной архитектуры - “разделение обязанностей”. Появление языка программирования Java привело к возникновению апплетов - промежуточного звена, которое можно было вынести на клиентскую машину, тем самым перенеся часть вычислений с сервера на клиентскую машину.

Проблемы двухзвенной архитектуры следующие:

- при тонком клиенте возникают проблемы с производительностью и масштабируемостью системы;
- при толстом клиенте возникают проблемы с управляемостью системы;

Поэтому следующий шаг очень логичен, хоть и появился не сразу:

- Представление (отображение) данных, обработка действий пользователя остается на стороне клиентского приложения;
- Логика работы системы (бизнес-логика) остается на сервере приложений;
- Работа с данными отдается СУБД.

Таким образом каждый слой остается независимым и легко разделяемым между инфраструктурой. Клиентскому приложению остается небольшая часть функций и можно рассматривать (а в настоящий момент не иметь web-интерфейс считается моветоном) web-браузер в качестве клиента. Клиентская часть не знает о том, сколько и каких серверов обрабатывает ее запросы или хранит данные за отображением которых она обращается. Например, клиент может находиться в Европе, а данные которые он запросит могут легко находиться в ЦОДе через океан. Сервер приложений тоже не знает, сколько физически серверов будет обрабатывать и хранить данные за которыми он обращается. Для него это логическая прослойка “Сервер базы данных” к которой будут направлены его запросы.

Многие информационные системы сейчас сохранили “толстого” клиента, но это скорее для сохранения “обратной совместимости” или необходимости проведения интенсивных вычислений или визуализации на клиентском компьютере (например CAD/CAM системы). Некоторые крупные информационные системы имеют только web-интерфейс, например NetSuite. Одна из крупнейших платформ имеет в своем составе множество отраслевых решений, имеет только web-интерфейс и предоставляется только по модели SaaS.

Преимущества подобного архитектурного подхода трудно переоценить, но конечно есть и недостатки:

- б**о**льшая сложность разработки информационной системы, а следовательно и большая стоимость в том числе и стоимость поддержки и развития;
- низкие требования к клиентскому “железу” формируют высокие требования к инфраструктуре сервера приложений и сервера баз данных, а следовательно к их стоимости;
- также низкие требования к каналу связи между клиентом и сервером приложений, формируют требования к высокопроизводительному каналу связи между сервером приложений и сервером баз данных, а следовательно к стоимости их реализации и поддержки.

Чтобы превратить трехуровневую архитектуру в многоуровневую, достаточно добавить дополнительные уровни в необходимые места. Например, сервисы агрегации данных в распределенных базах данных. Сервер приложений будет обращаться к данным сервисам не подозревая, что это еще один слой между ним и СУБД. Многие современные информационные системы разрабатываются сразу в распределенной архитектуре, что позволяет им иметь большую масштабируемость, гибкость, отказоустойчивость в целом.

Почему же так долго системы эволюционировали до казалось бы столь ясного и логичного подхода к архитектуре? Во первых необходимо сюда отнести инерцию человеческого мышления. Во вторых трудозатраты на разработку двухзвенной и трехзвенной архитектуры могут отличаться на порядки, а в самом начале и отличались ввиду полного отсутствия стандартизации, готовых фрейворков разработки и собственно сами языки программирования не давали такой широкой возможности. Она появилась наверное только с распространением объектно-ориентированных языков программирования. Отсутствие стандартов в данных областях, подразумевало, что каждый будет выдумывать свои протоколы общения между СУБД и сервером приложений (например). Современные платформы для построения информационных систем такого уровня могут использовать в качестве СУБД, например, Oracle, IBM DB2, MS SQL Server, PostgreSQL Server, что называется “из коробки”. Языки программирования, например C#, Java, имеют готовые фреймворки для работы с СУБД. Методология программирования также стандартизировала паттерны проектирования, например, MVC - Model, View, Controller. Где, Model - данные, View - отображение, Controller - отражает действия пользователя из View в Model. При этом бизнес-логика приложения может находиться как в Model, так и в Controller.

Требования к распределенным системам можно свести в следующий перечень:

- Локальность автономии. Это означает, что функционирование данного узла сети управляется этим узлом и не зависит от функционирования другого узла сети. Под локальной автономией подразумевается также, что все узлы сети рассматриваются как равные.
- Непрерывность функционирования. Подразумевается, что даже в случае неисправности отдельного узла работа системы продолжается, хотя и на более низком уровне.
- Независимость от расположения. Пользователям не следует знать, в каком физическом месте хранятся данные, наоборот, с логической точки зрения пользователям следует обеспечить такой режим, при котором создается впечатление, что все данные хранятся на их собственном локальном узле.
- Независимость от аппаратного обеспечения и операционной системы. Данные должны интегрироваться на компьютерах с различными техническими характеристиками, архитектурами и операционными системами, чтобы для пользователя создавалось представление единой системы.
- Независимость от сети. Система должна поддерживать не только узлы с разным аппаратным обеспечением и разными операционными системами, но и разные типы сетей.
- Независимость от СУБД. Различные СУБД на различных узлах сети должны быть интегрируемы друг с другом.

Глава II. Современные реализации клиент-серверной архитектуры

2.1 Стандартизация

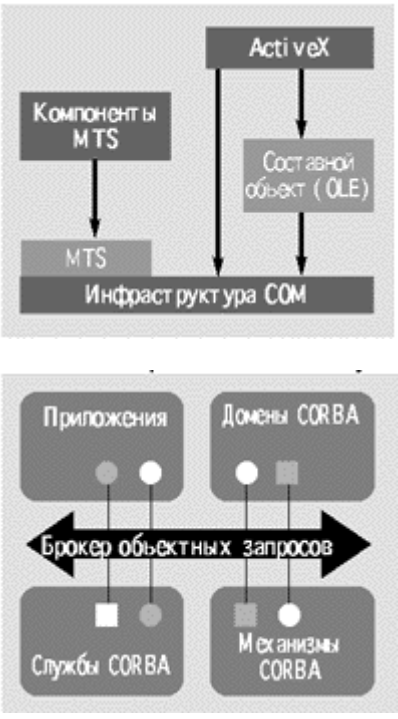
Для реализации модели необходимо обеспечить прозрачность взаимодействия между различными компонентами системы, а следовательно, иметь стандарт такого взаимодействия. Любая прикладная система, вне зависимости от выбранной модели (двух или трехуровневая), требует наличия инструментов, которые могли бы ускорить сам процесс создания системы. А также, одновременно с ускорением обеспечить прозрачность и масштабируемость системы. В практике разработки и внедрения систем корпоративного масштаба явно присутствует тенденция использования объектно-ориентированных компонентных средств разработки. Соответственно, полноценное применение объектов в распределенной клиент-

серверной среде требует и распределенного объектно-ориентированного взаимодействия, то есть возможности обращения к удаленным объектам.

Таким образом, мы приходим к анализу существующих распределенных объектных моделей. На настоящий момент наибольшей проработанностью отличаются COM/DCOM/ActiveX и CORBA/DCE/Java. И COM, и CORBA — современные программные технологии, которые могут быть использованы для создания крупных корпоративных систем.

CORBA (Common Object Request Broker Architecture) — это набор открытых спецификаций интерфейсов, определяющий архитектуру технологии межпроцессного и платформонезависимого манипулирования объектами. Разработчиками данных интерфейсов являются OMG и X/Open. Реализовать технологию в соответствии со спецификациями может кто угодно.

Рис. 2. Модели COM и CORBA



Функции CORBA и COM — это функции промежуточного программного обеспечения объектной среды. Для того чтобы обеспечить взаимодействие объектов и их интеграцию в цельную систему, архитектура промежуточного уровня должна реализовать несколько базовых принципов.

- Независимость от физического размещения объекта. Компоненты программного обеспечения не обязаны находиться в одном исполняемом

файле, выполняться в рамках одного процесса или размещаться на одной аппаратной системе.
- Независимость от платформы. Компоненты могут выполняться на различных аппаратных и операционных платформах, взаимодействуя друг с другом в рамках единой системы.
- Независимость от языка программирования. Различия в языках, которые используются при создании компонентов, не препятствуют их взаимодействию друг с другом.

.Net же сам по себе есть программный коммерческий продукт, ориентированный, в первую очередь, на практическое использование. По этой причине в .Net кроме собственно архитектуры манипулирования объектами присутствует еще и вся инфраструктура, обеспечивающая распределенную обработку объектов в реальном программном и аппаратном окружении. Разработчиком .Net Framework является фирма Microsoft.

Одно из основных отличий CORBA и .Net заключается в подходе к разработке и развитию этих технологий. Условно можно назвать подход, принятый в CORBA, как "движение сверху", а подход .Net как "движение снизу". То есть, разработка и реализация в CORBA, в первую очередь, ведется от спецификаций OMG, которые хотя и основываются на практическом опыте разработчиков, но в своей основе, теоретические проекты интерфейсов, рассчитанные на практическую реализацию кем угодно.

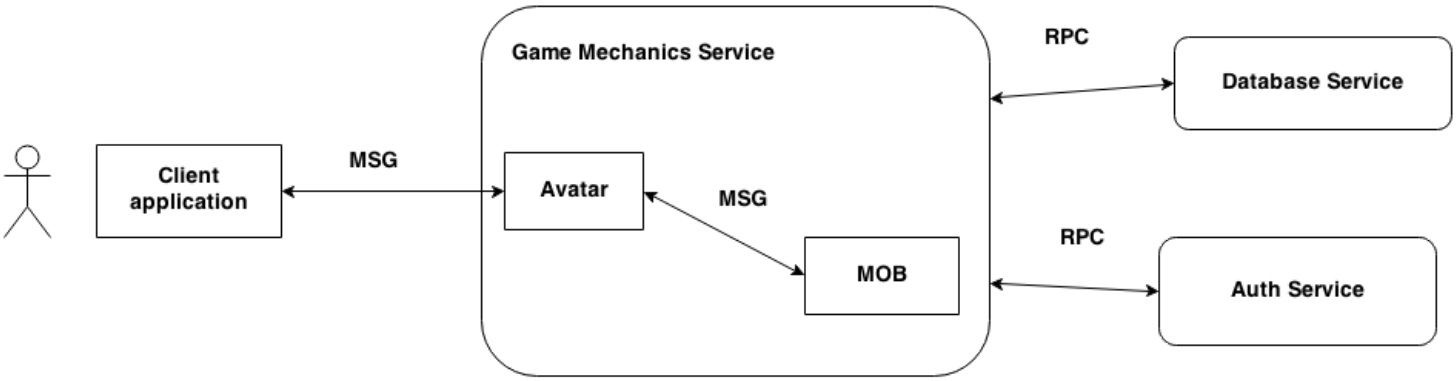
В отличие от CORBA, .Net возникла эволюционным путем, т.е. путем проб и ошибок фирмы Microsoft, которая создавала свои продукты, продвигаясь мелкими шагами, отталкиваясь от опыта использования их миллионами пользователей. Конечно, это не обошлось без крови и слез данных пользователей, но зато есть гораздо большая уверенность, что этот путь не заведет в тупик. Возможно, в некоторых отношениях .Net выглядит более примитивной, но зато есть уверенность, что это реальная работающая технология. Эта примитивность может оказаться мнимой, так как CORBA — открытый стандарт и все ее внутренности предоставлены ко всеобщему обозрению, в .Net же внутренняя структура в основном скрыта, а то что находится снаружи действительно смотрится довольно просто.

2.2 Реализация клиент-серверной архитектуры в масштабных проектах

В современном мире, наиболее сложными и нагруженными являются пожалуй не корпоративные информационные системы, охватывающие тысячи пользователей, а представители игровой индустрии, такие как MMO-игры охватывающие миллионы пользователей по всему миру. Архитектура таких проектов должна быть масштабируемой. Ведь на старте проекта нагрузки как правило невелики, а впоследствии могут вырасти очень быстро. Также, нагрузка очень не регулярная, и характеризуется всплесками активности пользователей. Соответственно, архитектура должна уметь быстро наращивать мощности и распределять нагрузку таким образом, чтобы выдержать пик активности пользователей. Например в проекте “Eve-online”, при массовом присутствии (несколько тысяч) игроков в одном секторе, включается замедление игрового времени. В проекте “World of Tanks” используется блокировка входа избыточного количества игроков и физическое распределение серверов по географии присутствия пользователей.

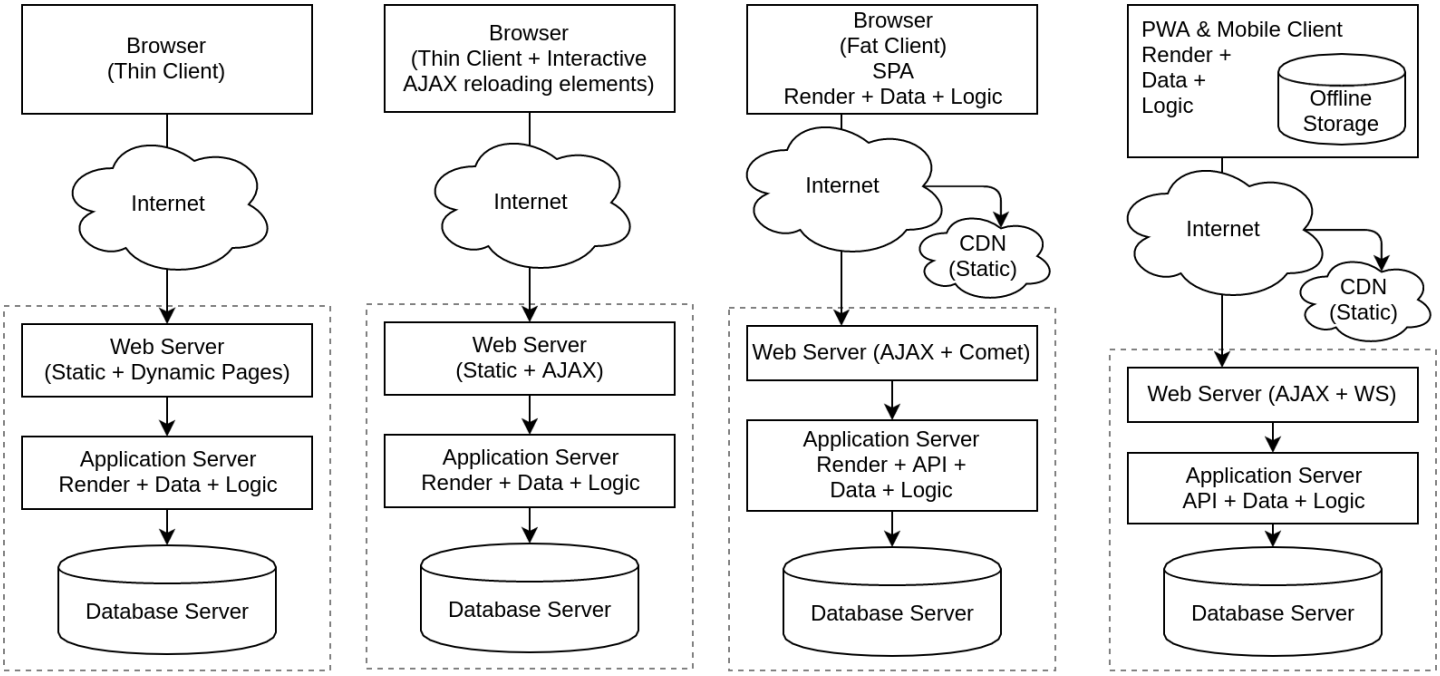
В целом, архитектура MMO-проекта выглядит следующим образом:

Рис 3. Архитектура MMO



В целом, выглядит как классическая трехуровневая архитектура. Вообще, можно отметить, что современные информационные системы становятся похожи на набор сервисов, которые взаимодействуют как с пользователем так и между собой.

Рис 4. Текущие архитектуры современных информационных систем



Большая часть текущих информационных систем построены по архитектуре #5, сейчас очень активно развитие систем по #8. Хотя все остальные архитектуры также присутствуют. Например #6 это всякого рода почтовые службы типа mail.ru, gmail и т.д.

Если обобщать, то можно прикинуть векторы развития информационных систем:

Высоконагруженное облако.

Преимущества: Облака хорошо справляются с высокими нагрузками, хорошо масштабируются.

Недостатки: Плохая интерактивность с пользователем.

DB-центрический подход.

Преимущества: СУБД встроенное в приложение, работа с локальными данными.

Недостатки: сведение всего общения между приложениями на уровень данных.

Одноранговое взаимодействие (Peer-to-peer).

Преимущества: Сервера будут выполнять только функции брокера, локальное взаимодействие гораздо продуктивнее.

Недостатки: Вопрос доверия соседнему пиру. Отсутствие стандартизированных протоколов, библиотек, фреймворков для разработки ПО (весь инструментарий придется писать с нуля).

ЗАКЛЮЧЕНИЕ

Технологии не стоят на месте и по прежнему развиваются с высокой скоростью. Также как всегда, львиная доля решений не идет в ногу с технологическим лидером. Этому конечно есть множество совершенно обоснованных объяснений и причин. Даже сейчас встречаются информационные системы разработанные на FoxPro или Clipper под DOS (да, да, тот самый файл-серверный вариант) и самое главное, что они работают и востребованы. Но это в основном связано с утерей компетенций по постановке задачи, с дороговизной разработки аналогичного решения на современных технологиях и т.д.

В общем, если проанализировать и сгруппировать текущие потребности рынка, то получится следующий перечень:

- Highload — обработка интенсивного потока запросов;
- Big data — возможность обрабатывать большие объемы данных;
- Big memory и in-memory DB — высокая утилизация объемов памяти или перенос баз данных полностью в оперативную память;
- High connectivity — большое количество конкурентных (одновременных и долгоживущих) соединений клиентов к серверу.

Очень многие из данных запросов уже должны быть реализованы на уровне госпрограмм в рамках развития таких систем как “умный город”, в рамках развития цифровизации правительства, например портал “Госуслуги”.

Естественно, большинство общения сейчас идет по каналам широкополосного доступа. По данным Ростелекома, ШПД у абонентов, по территории России в 2014 году было 75% включая мобильные каналы связи. А мобильных номеров по статистике было по два-три у каждого. Таким образом, основным каналом связи остается интернет.

Информационные системы, судя по тенденциям развития, будут строить костяк и в дальнейшем обвешиваться сервисами как елка - гирляндами. С точки зрения разработки конкретного сервиса, так выйдет дешевле и быстрее. Но надо помнить,

что тут становится сильной роль системного архитектора, который должен держать в голове всю эту гирлянду и не допускать проблем во взаимодействии различных сервисов между собой.

Также, данные сервисы начинают активное проникновение в нашу повседневную жизнь уже в явном виде, например в виде интернета вещей. И скоро идиома “позвонить по холодильнику“ из нашего детства может обернуться суровой действительностью и даже больше, можно будет “позвонить холодильнику”. Уже сейчас можно строить клиент-серверную архитектуру своего умного дома.

БИБЛИОГРАФИЯ

1. http://www.netsuite.com/portal/home.shtml
2. https://habr.com/post/326016/
3. Гради Буч. Объектно-ориентированный анализ и проектирование с примерами приложений на C++
4. Мартин Фаулер. Архитектура корпоративных программных приложений.
5. Елманова Н. Перенос приложений C++ Builder в архитектуру клиент/сервер
6. Кузин А.В., Левонисова С.В. - Базы данных (5-е изд.) (Высшее профессиональное образование. Бакалавриат) - 2012
7. http://kunegin.com/ref3/corba5/12.htm
8. https://www.intuit.ru/studies/courses/1145/214/lecture/5513