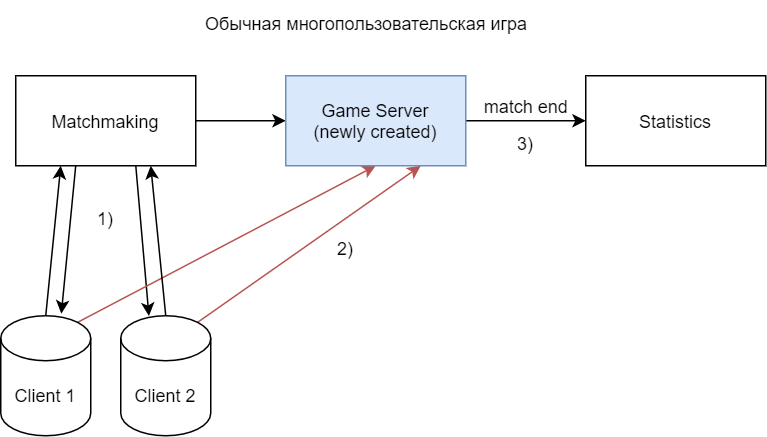
**Архитектура сервера для описания передвижения агентов-клиентов в трехмерном виртуальном пространстве.**

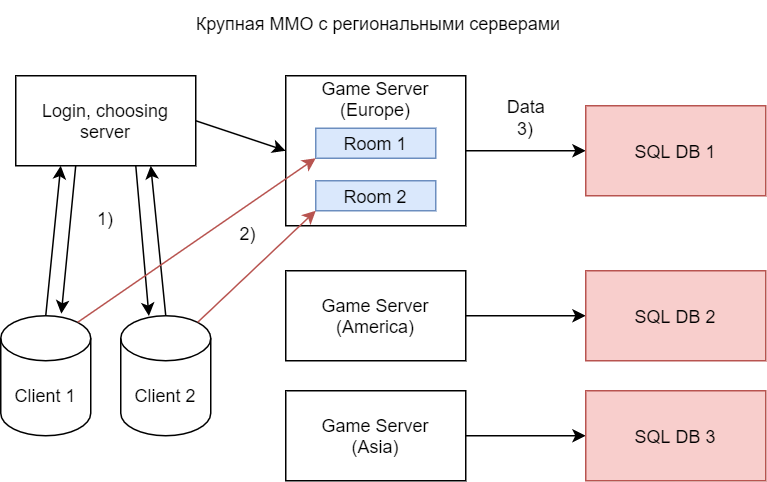
В качестве задачи возьмем устройство абстрактной MMO игры. Клиент-агент может передвигаться по виртуальному миру без телепортации (или только в редких случаях) и взаимодействовать с другими агентами, изменяя их внутреннее состояние. Виртуальный мир может быть разделен на регионы, между которыми появляется экран загрузки. Мы сразу исключаем из рассмотрения игры со структурой матчей двух команд с ограниченным числом игроков, потому что их архитектура не является сложной. В этом случае часто используется комбинация из трех систем: сервис матчмейкинга, сервис одного матча, сервис публикации результатов игры. Сначала игроки, ищущие матч, объединяются в группу, затем для этой группы поднимается экземпляр игрового сервера хоста, затем в конце игры этот сервер публикует данные об окончании игры и уничтожается. В качестве хоста может выступать и один из игроков-клиентов.



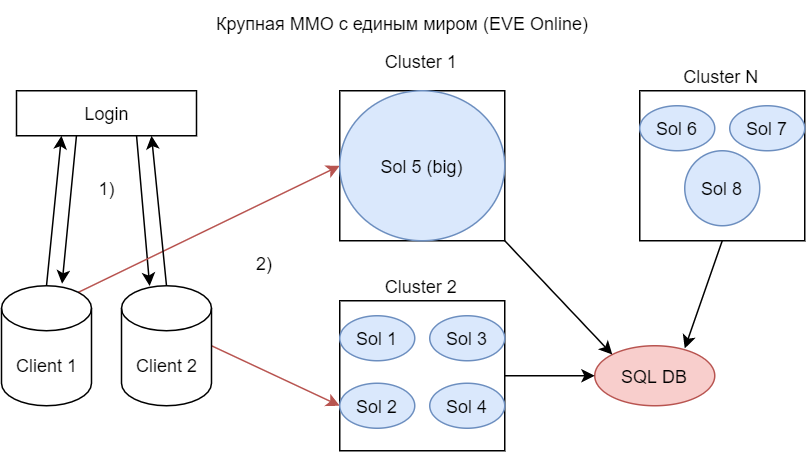
Серверы работают в stateful режиме, потому что вся информация о внутренних состояниях агентов и текущие взаимодействия образуют общее состояние системы. Это состояние, которое всегда берется за истинное, необходимо доставить до всех клиентов, чтобы они могли видеть других игроков, а также видеть взаимодействия с собой и между другими агентами. Но, как клиенты синхронизируют свое состояние с сервером, так и сервер меняет свое состояние в зависимости от действий клиентов. Об этом будет далее, но для начала зададим важное правило, что существует определенный радиус видимости вокруг агента, вне которого он не видит других агентов и не может взаимодействовать с ними. Это правило позволяет уменьшить нагрузку на сервер и на передачу данных по сети. Сервер определяет видимость агентов по отношению друг к другу и рассылает только необходимые данные клиентам.

Определим, каким образом происходит обмен данными в системе. Все клиенты производят обмен данными напрямую только с сервером, не друг с другом. Это происходит, чтобы обеспечить стабильное общее состояние, и потому что мы по умолчанию не можем доверять данным с клиента. Сервер должен проверять валидность действий, выполняемых агентом, потому что на стороне клиента любое локальное состояние может быть подделано. Если сервер полностью сам управляет локальными состояниями агентов, а от них принимает только запросы на изменения состояния, и клиент не может изменить свое состояние сам без согласия сервера, то сервер считается полностью авторитарным. Этот подход удобен для организации честной игры, но дает ощутимую задержку в изменении клиентом своего же состояния (и чужого). Если использовать такой подход для управления передвижением агента, то управление будет производиться с задержкой в 2 \* ping + server\_frame\_time. Ping в обычном режиме может быть от 1-2 мс до 300 мс и больше. Поэтому часто делают так: система дает клиенту изменять свое локальное состояние, в том числе изменять свое положение в пространстве и даже локальное внутреннее состояние для других агентов, но все равно проверяет допустимость таких изменений у себя, и если оно засекло невозможное расхождение в текущем состоянии на сервере и предлагаемом состоянии от клиента, то оно возвращает состояние к последнему валидному. Так, если агент слишком быстро передвигается в мире, то его может отбрасывать назад раз в секунду. Допустим, что, внедряя такую логику, мы получили на своем клиенте актуальное положение самого себя в пространстве. Но для положения других агентов все равно будет задержка в T=(ping\_me + ping\_other + server\_frame\_time). Для получения самых актуальных данных о положении других агентов можно использовать предсказания на стороне клиента, основываясь на переданных данных о положении и скорости агента, используя различные алгоритмы для «сшивания» траектории предсказанной и полученной.

Для обмена данных используют протоколы UDP и TCP. Протокол http не используется, потому что он требует ответа на каждый запрос и создает полностью новый запрос каждый раз. Протоколы UDP и TCP оба устанавливают канал, по которому будут отправляться данные в течение продолжительного времени, и инициатором сообщения является отправитель, а не получатель, то есть получатель автоматически принимает все сообщения, пришедшие по каналу связи. Разница в протоколах UDP и TCP в том, что в TCP гарантируется доставка всех сообщений, повторно отправляя сообщения, если они не дошли. Во многих ситуациях к тому моменту, как старое утерянное сообщение дойдет, оно будет неактуальным. Так, информацию о передвижении объекта лучше всего посылать через UDP, потому что важна только самая последняя информация, а часть из многочисленных N сообщений в секунду точно дойдет до получателя. Из-за отсутствия гарантии получения сообщений протокол UDP быстрее, чем TCP. Но для некоторых действий (пр. единичной атаки) предпочтительнее, когда есть гарантия получения сообщения. Если клиент – страница в браузере, то можно использовать технологию websocket, работающую поверх TCP.



Перед системой стоит задача минимизации латентности, поэтому клиенты отправляют данные напрямую на компьютер-сервер, обрабатывающий мир, без посредников Gateway’ев. Рассмотрим несколько вариантов масштабирования MMO. Первый вариант, самый частый, заключается в том, что масштабирование системы происходит за счет добавления слабо связных экземпляров серверов. Часто они даже являются статически заданными. Обычной бывает ситуация, когда игроков разделяют географически на американский, европейский и азиатский сервер. Часто на одном сервере (а скорее кластере серверов) делают несколько миров-комнат (по одному на физический сервер), то есть игроки на Европейском сервере разделены на N параллельных независимых симуляций (эти симуляции уже могут быть динамическими). В этих независимых процессах не синхронизируют самих игроков, но в случае, если в MMO предусмотрена экономика и профиль персонажа, то можно объединить эти миры через общую биржу и экономику через отдельный сервис и разрешить игрокам менять миры, сохраняя их профиль в БД по выходу из игры или с некоторой периодичностью. Большие проблемы с нагрузкой могут возникнуть, если никак не синхронизировать общее мета-состояние мира (экономики). С этим столкнулись разработчики из Amazon при запуске своей MMO New World. При входе в игру в первый раз игроки могли сами выбирать, в какой комнате играть. Часто они выбирали первые из списка, образуя неравномерную нагрузку. Также в игре была независимая экономика на каждом из миров-комнат, вследствие чего разработчики запретили играть за одного персонажа на разных мирах, требуя создавать нового в каждом мире. Этот факт удерживал игроков на выбранных ими впервые серверах, усугубляя ситуацию неравномерности нагрузки. Доходило до того, что большая часть игроков находилась в ожидании, в то время как другие полупустые сервера простаивали. Фактически это пример проблемы, которая может возникнуть при неправильной реализации sticky sessions.



Второй вариант, это когда виртуальное пространство разделено по серверам, то есть различные регионы игрового мира находятся на разных серверах. Это позволяет иметь большее количество агентов в одном регионе (и меньше параллельных миров-комнат), но регионы имеют четкие границы, взаимодействие агентов через которые невозможно. Интересным примером реализации такой схемы является MMO EVE Online. Все игроки реального мира находятся в одном виртуальном, без ограничений на количество игроков в одном регионе. Сам виртуальный мир – это процедурно генерируемое космическое пространство, а регионы – отдельные солнечные системы (около 8000 шт.). Переход между регионами доступен только через определенные порталы. Благодаря такой обособленности регионов, каждому региону выделяется одна виртуальная машина, а переходы между регионами достигаются за счет отправки с одного сервера на другой информации о намерении игрока подключиться и переподключении самого клиента. Количество игроков в разных регионах постоянно меняется, и в некоторые моменты возникают пиковые нагрузки, во много раз превышающие обычные. Рекордная нагрузка – битва на 7,500 игроков в течение 21 часа. Во время повышения количества игроков в регионе, виртуальная машина мигрирует на более мощный сервер, а если количество игроков повышается далее, то игровое время замедляется. Разработчики писали (в 2010 году), что проблем с оперативной памятью сервера у них не возникает, хватает 4Гб. Так как некоторые запросы выходят за пределы своего сервера, то разработчики продумали собственное решения для удаленных RPC запросов. В итоге, из-за простоты скриптинга и поддержки асинхронных микропотоков, они выбрали Stackless Python. Вследствие отказа от традиционного стэка вызовов как в C, он дает дополнительные возможности для создания легковесных микропотоков, сопрограмм (ко-рутин), коммуникационных каналов и сериализации задач. Похожим функционалом также обладает язык Go. Плюс языка Stackless Python в том, что в нем можно задавать места в функциях, в который может происходить смена контекста, упрощая многопоточное программирование. В качестве БД разработчики используют единую реляционную SQL базу данных. Для некоторых данных они используют "read uncommitted", то есть не ставят Lock при чтении данных. Они используют только SSD для хранения информации. Другая проблема, с которой столкнулись разработчики, состояла в балансировке нагрузки. Они предположили, что размещение виртуальных машин для соседних солнечных систем должно быть локализовано, и кластеризации зависела только от положения солнечных систем в игре, но это привело к плохому балансу нагрузки. Полностью убрав зависимость от расстояний и оставив только параметр загруженности, баланс улучшился не сильно. Наилучший баланс нагрузки был получен при учете обоих факторов – нагрузки и связности локаций.