В целом существует два основных типа сетевых архитектур: peer-to-peer и клиент-серверная. В архитектуре peer-to-peer (p2p) данные передаются между любыми парами подключенных игроков, а в клиент-серверной архитектуре данные передаются только между игроками и сервером.  
  
Хотя архитектура peer-to-peer по-прежнему используется в некоторых играх, стандартом является клиент-серверная: она проще в реализации, требует канал меньшей ширины и облегчает защиту от читерства. Поэтому в этом руководстве мы сосредоточимся на клиент-серверной архитектуре.  
  
В частности, нас больше всего интересуют авторитарные серверы: в таких системах сервер всегда прав. Например, если игрок думает, что находится в координатах (10, 5), а сервер говорит ему, что он в (5, 3), то клиент должен заменить свою позицию той, которую передаёт сервер, а не наоборот. Использование авторитарных серверов упрощает распознавание читеров.  
  
В игровых сетевых системах есть три основных компонента:

* Транспортный протокол: как передаются данные между клиентами и сервером.
* Протокол приложения: что передаётся от клиентов серверу и от сервера клиентам и в каком формате.
* Логика приложения: как передаваемые данные используются для обновления состояния клиентов и сервера.

Очень важно понять роль каждой части и связанные с ними трудности.

Транспортный протокол

Первый шаг заключается в выборе протокола для транспортировки данных между сервером и клиентами. Для этого существует два Интернет-протокола: [TCP](https://en.wikipedia.org/wiki/Transmission_Control_Protocol) и [UDP](https://en.wikipedia.org/wiki/User_Datagram_Protocol). Но вы можете создать и собственный транспортный протокол на основе одного из них или применить библиотеку, в которой они используются.

Сравнение TCP и UDP

И TCP, и UDP основаны на [IP](https://en.wikipedia.org/wiki/Internet_Protocol). IP позволяет передавать пакет от источника получателю, но не даёт гарантий, что отправленный пакет рано или поздно попадёт к получателю, что он доберётся до него хотя бы раз и что последовательность пакетов придёт в правильном порядке. Более того, пакет может содержать только ограниченный размер данных, задаваемый величиной [MTU](https://en.wikipedia.org/wiki/Maximum_transmission_unit).  
  
UDP является всего лишь тонким слоем поверх IP. Следовательно, он имеет те же ограничения. В отличие от него, TCP обладает множеством особенностей. Он обеспечивает надёжное упорядоченное соединение между двумя узлами с проверкой на ошибки. Следовательно, TCP очень удобен и используется во множестве других протоколов, например, в [HTTP](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol), [FTP](https://en.wikipedia.org/wiki/File_Transfer_Protocol) и [SMTP](https://en.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol). Но все эти функции имеют свою цену: [*задержку*](https://en.wikipedia.org/wiki/Latency_(engineering)).  
  
Чтобы понять, почему эти функции могут вызывать задержку, надо разобраться, как работает TCP. Когда узел-отправитель передаёт пакет узлу-получателю, он ожидает получить подтверждение (ACK). Если спустя определённое время он не получает его (потому что пакет или подтверждение было утеряно, или по каким-то другим причинам), то отправляет пакет повторно. Более того, TCP гарантирует получение пакетов в правильном порядке, поэтому пока утерянный пакет не получен, все остальные пакеты не могут быть обработаны, даже если они уже получены узлом-получателем.  
  
Но как вы наверно понимаете, задержка в многопользовательских играх очень важна, особенно в таких активных жанрах, как FPS. Именно поэтому многие игры используют UDP с собственным протоколом.  
  
Собственный протокол на основе UDP может быть эффективнее TCP по различным причинам. Например, он может помечать некоторые пакеты как надёжные, а другие — как ненадёжные. Поэтому его не волнует, добрался ли ненадёжный пакет до получателя. Или он может обрабатывать несколько потоков данных, чтобы потерянный в одном потоке пакет не замедлял остальные потоки. Например, может существовать поток для ввода игрока и ещё один поток для сообщений чата. Если сообщение чата, которое не является срочными данными, потеряно, то оно не замедлит срабатывание ввода, который является неотложным. Или же собственный протокол может реализовать надёжность иначе, чем в TCP, чтобы быть более эффективным в условиях видеоигр.  
  
Итак, если TCP такой отстойный, то мы будем создавать свой транспортный протокол на основе UDP?  
  
Всё немного сложнее. Даже хотя TCP почти субоптимален для игровых сетевых систем, он может вполне хорошо работать конкретно в вашей игре и сэкономить ваше драгоценное время. Например, задержка может и не быть проблемой для пошаговой игры или игры, в которую можно играть только в сетях LAN, где задержки и утеря пакетов намного меньше, чем в Интернете.  
  
Во многих успешных играх, в том числе World of Warcraft, Minecraft и Terraria, используется TCP. Однако в большинстве FPS применяются собственные протоколы на основе UDP.

## Сетевые библиотеки

Если вам нужно что-то более эффективное, чем TCP, но вы не хотите заморачиваться реализацией собственного протокола и вдаваться во множество подробностей, то можете воспользоваться сетевой библиотекой. Их очень много:

* [yojimbo](https://github.com/networkprotocol/yojimbo) Гленна Фидлера
* [RakNet](https://github.com/facebookarchive/RakNet), которая больше не поддерживается, но её форк [SLikeNet](https://github.com/SLikeSoft/SLikeNet) похоже ещё активен.
* [ENet](http://enet.bespin.org/) — это библиотека, созданная для многопользовательского FPS [Cube](http://cubeengine.com/)
* [GameNetworkingSockets](https://github.com/ValveSoftware/GameNetworkingSockets) компании Valve

Транспортный протокол: заключение

Подведём итог: существует два основных транспортных протокола: TCP и UDP. TCP обладает множеством полезных особенностей: надёжность, сохранение порядка пакетов, обнаружение ошибок. У UDP всего этого нет, зато TCP по своей природе обладает повышенными задержками, недопустимыми для некоторых игр. То есть для обеспечения низких задержек можно создать собственный протокол на основе UDP или использовать библиотеку, реализующую транспортный протокол на UDP и адаптированную для многопользовательских видеоигр.  
  
Выбор между TCP, UDP и библиотекой зависит от нескольких факторов. Во-первых, от потребностей игры: нужны ли ей низкие задержки? Во-вторых, от требований протокола приложения: нужен ли ему надёжный протокол? Как мы увидим из следующей части, можно создать протокол приложения, для которого вполне подойдёт ненадёжный протокол. Наконец, нужно ещё учитывать опытность разработчика сетевого движка.  
  
У меня есть два совета:

* Максимально абстрагируйте транспортный протокол от остальной части приложения, чтобы его можно было легко заменить, не переписывая весь код.
* Не занимайтесь преждевременной оптимизацией. Если вы не специалист по сетям и не уверены, нужен ли вам собственный транспортный протокол на основе UDP, то можете начать с TCP или библиотеки, обеспечивающих надёжность, а затем протестировать и измерить производительность. Если возникают проблемы и вы уверены, что причина заключается в транспортном протоколе, то возможно настало время создавать собственный транспортный протокол.

Протокол приложения

Теперь, когда мы можем обмениваться данными между клиентами и сервером, нужно решить, какие именно данные передавать и в каком формате.  
  
Классическая схема заключается в том, что клиенты отправляют серверу ввод или действия, а сервер отправляет клиентам текущее игровое состояние.  
  
Сервер отправляет не полное, а отфильтрованное состояние с сущностями, которые находятся рядом с игроком. Он делает это по трём причинам. Во-первых, полное состояние может быть слишком большим для передачи с высокой частотой. Во-вторых, клиентов в основном интересуют визуальные и аудиоданные, потому что бОльшая часть игровой логики симулируется на сервере игры. В-третьих, в некоторых играх игрок не должен знать определённых данных, например, позицию противника на другом конце карты, ведь в противном случае он может сниффить пакеты и точно знать, куда двигаться, чтобы его убить.

Сериализация

Первым шагом будет преобразование данных, которые мы хотим отправить (ввод или игровое состояние), в подходящий для передачи формат. Этот процесс называется *[сериализацией](https://en.wikipedia.org/wiki/Serialization)*.  
  
В голову сразу приходит мысль использовать человекочитаемый формат, например JSON или XML. Но это будет совершенно неэффективно и впустую займёт большую часть канала.  
  
Вместо этого рекомендуется использовать двоичный формат, который намного более компактен. То есть пакеты будут содержать только несколько байтов. Здесь нужно учитывать проблему [*порядка байтов*](https://en.wikipedia.org/wiki/Endianness), который на разных компьютерах может отличаться.  
  
Для сериализации данных можно использовать библиотеку, например:

* [FlatBuffers](https://google.github.io/flatbuffers/) компании Google
* [Cap’n Proto](https://capnproto.org/) компании Sandstorm
* [cereal](http://uscilab.github.io/cereal/serialization_archives.html) Шейна Гранта и Рэндольфа Вурхиса

Только убедитесь, что библиотека создаёт портируемые архивы и заботится о порядке байтов.  
  
Альтернативным решением может быть самостоятельная реализация, она не особо сложна, особенно если в коде вы используете ориентированный на данные подход. Кроме того, она позволит вам выполнять оптимизации, которые не всегда возможны при использовании библиотеки.

## Сжатие

Количество данных, передаваемых между клиентами и сервером, ограничено пропускной способностью канала. Сжатие данных позволит передавать в каждом снэпшоте больше данных, увеличить частоту обновления или просто снизить требования к каналу.

### **Битовая упаковка**

Первая техника — это битовая упаковка. Она заключается в использовании ровно того количества битов, которое необходимо для описания нужной величины. Например, если у вас есть перечисление, которое может иметь 16 различных значений, то вместо целого байта (8 бит) можно использовать всего 4 бита.

Битовая упаковка особенно хорошо работает с дискретизацией, которая будет темой следующего раздела.

### **Дискретизация**

[Дискретизация](https://en.wikipedia.org/wiki/Quantization_(signal_processing)) — это техника сжатия с потерями, которая заключается в использовании для кодирования величины только подмножества возможных значений. Проще всего реализовать дискретизацию округлением чисел с плавающей запятой.

### **Алгоритмы сжатия**

Следующей техникой будут алгоритмы сжатия без потерь.  
  
Вот, на мой взгляд, три самых интересных алгоритма, которые нужно знать:

* [Кодирование Хаффмана](https://en.wikipedia.org/wiki/Huffman_coding) с заранее вычисленным кодом, которое чрезвычайно быстро и может давать хорошие результаты. Оно использовалось для сжатия пакетов в сетевом движке Quake3.
* [zlib](http://www.zlib.net/) — алгоритм сжатия общего назначения, который никогда не увеличивает объём данных. Как можно увидеть [здесь](https://en.wikipedia.org/wiki/Zlib), он применялся во множестве областей применения. Для обновления состояний он может оказаться избыточным. Но он может и пригодиться, если вам нужно отправлять клиентам с сервера ассеты, длинные тексты или рельеф.
* [Копирование длин серий](https://en.wikipedia.org/wiki/Run-length_encoding) — это, наверно, простейший алгоритм сжатия, но он очень эффективен для определённых типов данных, и может использоваться как этап предварительной обработки перед zlib. Он особенно подходит для сжатия рельефа, состоящего из тайлов или вокселей, в которых множество соседних элементов повторяется.

### **Дельта-сжатие**

Последняя методика сжатия — это дельта-сжатие. Она заключается в том, что передаются только различия между текущим игровым состоянием и последним состоянием, полученным клиентом.  
  
Впервые она была применена в сетевом движке Quake3.

Шифрование

Кроме того вам может понадобиться шифровать передачу информации между клиентами и сервером. На это есть несколько причин:

* приватность/конфиденциальность: сообщения могут быть прочитаны только получателем, и ни одному другому лицу, выполняющему сниффинг сети, не удастся их прочесть.
* аутентификация: человек, желающий исполнять роль игрока, должен знать его ключ.
* предотвращение читерства: злонамеренным игрокам будет намного сложнее создавать собственные пакеты для читерства, им придётся воспроизводить схему шифрования и находить ключ (который меняется при каждом соединении).

## Протокол приложения: заключение

На этом мы закончим с протоколом приложения. Я считаю, что сжатие совершенно необязательно и решение о его использовании зависит только от игры и требуемой пропускной способности канала. Шифрование, на мой взгляд, обязательно, но в первом прототипе можно обойтись без него.

# Логика приложения

Теперь мы способны обновлять состояние в клиенте, но можем столкнуться с проблемами задержек. Игроку, выполнив ввод, нужно ждать обновления состояния игры от сервера, чтобы увидеть, какое воздействие он оказал на мир.  
  
Более того, между двумя обновлениями состояния мир совершенно статичен. Если частота обновления состояний низка, то движения будут очень дёрганными.  
  
Существует несколько техник, позволяющих снизить влияние этой проблемы, и в следующем разделе я о них расскажу.

## Техники сглаживания задержек

Первая техника заключается в том, чтобы применять результат ввода напрямую, не ожидая ответа от сервера. Это называется прогнозированием на стороне клиента. Однако когда клиент получает обновление от сервера, он должен убедиться, что его прогноз был верным. Если это не так, то ему нужно просто изменить своё состояние согласно полученному от сервера, потому что сервер авторитарен. Эта техника впервые была использована в Quake.   
  
Второй набор техник используется для сглаживания движения других сущностей между двумя обновлениями состояния. Существует два способа решения этой задачи: интерполяция и экстраполяция. В случае интерполяции берутся два последних состояния и показывается переход из одного в другое. Её недостаток в том, что она вызывает небольшую долю задержки, потому что клиент всегда видит то, что происходило в прошлом. Экстраполяция заключается в прогнозировании того, где сейчас должны находиться сущности на основании последнего состояния, полученного клиентом. Её недостаток в том, что если сущность полностью меняет направление движения, то возникнет большая погрешность между прогнозом и реальной позицией.  
  
Последняя, самая продвинутая техника, полезная только в FPS — это компенсация лага. При использовании компенсации лага сервер учитывает задержки клиента, когда он стреляет в цель. Например, если игрок выполнил хедшот на своём экране, но в реальности его цель из-за задержки находилась в другом месте, то было бы нечестно отказывать игроку в праве на убийство из-за задержки. Поэтому сервер выполняет перемотку времени назад, на тот момент, когда игрок выстрелил, чтобы симулировать, что видел игрок на своём экране, и проверить коллизию между его выстрелом и целью.

## Предотвращение читерства

Существует две основные техники предотвращения читерства.  
  
Первая: усложнение отправки читерами вредоносных пакетов. Как сказано выше, хорошим способом её реализации является шифрование.  
  
Вторая: авторитарный сервер должен получать только команды/ввод/действия. Клиент не должен иметь возможности изменять состояние на сервере, кроме как отправкой ввода. Тогда сервер каждый раз при получении ввода должен перед его применением проверять его на допустимость.

## Логика приложения: заключение

Рекомендую вам реализовать способ симуляции больших задержек и низких частот обновления, чтобы иметь возможность протестировать поведение своей игры в плохих условиях, даже когда клиент и сервер запущены на одном компьютере. Это сильно упростит реализацию методик сглаживания задержек.