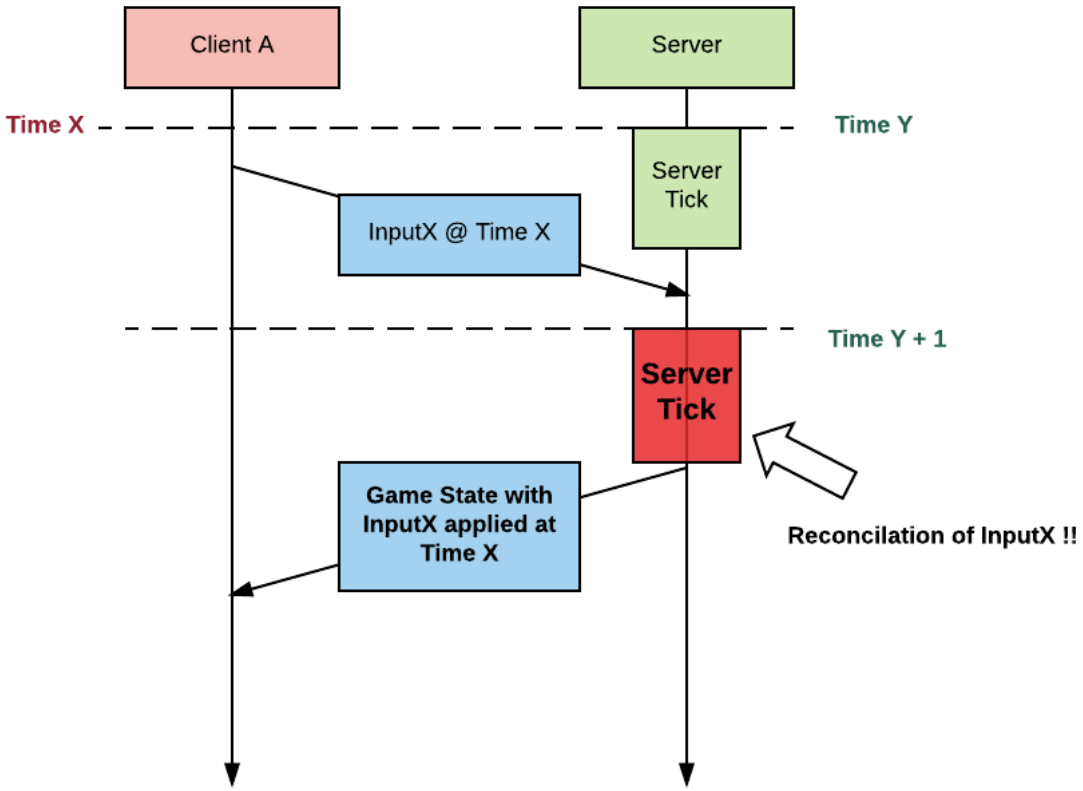
@PatientZero
Переводчик-фрилансер

вчера в 21:53

Синхронизация состояний в многопользовательских играх

перевод

Разработка игр*, Алгоритмы*



Проблема многопользовательских игр

Одна из самых сложных задач многопользовательских игр заключается в синхронизации состояний всех игроков с состоянием сервера. В Интернете есть хорошие статьи по этой теме. Однако в них не достаёт кое-каких подробностей, что может сбивать с толку новичков в программировании игр. Надеюсь, что у меня получится объяснить всё в этой статье.

Я обозначу несколько техник, обычно используемых для решения таких задач. Прежде чем переходить к проблеме, давайте вкратце рассмот принцип работы многопользовательских игр.

Обычно программа игры должна симулировать следующее:

изменения в окружении с учётом времени и вводимых игроками данных.

Игра — это программа, хранящая состояние, поэтому она зависит от времени (реального или логического). Например, PACMAN симулирует окружение, в котором постоянно перемещаются призраки.

Многопользовательская игра не является исключением, однако из-за взаимодействия игроков её сложность намного выше.

Возьмём, например, классическую игру «Змейка»:

Предположим, что мы используем клиент-серверную архитектуру. Игровая логика работает следующим образом:

1. Считывание вводимых игроком данных, изменяющих направление движения змейки. Они могут иметь одно из значений: [←, ↑, →, ↓].
2. Применение вводимых данных в случае их наличия. Это изменяет направление движения змейки.
3. Перемещение змейки на одну единицу измерения пространства.
4. Проверка наличия столкновения каждой из змеек с врагом/стеной/своим телом, затем удаление их из игры.
5. Повтор цикла.

Эта логика должна выполняться на сервере с постоянным интервалом. Как показано ниже, каждый цикл называется *кадром (frame)* или *такт*

(tick).

```
class Server {

  def main(): Unit = {
     while (true) {
        /**
        * 1. Считывание вводимых пользователем данных, имеющих одно из значений: [←, ↑, →, ↓].
        * 2. Применение данных при их наличии, при этом изменяется направление змейки.
        * 3. Перемещение змейки на одну единицу измерения пространства.
        * 4. Проверка столкновения с врагом/стеной/своим телом для каждой змейки, удаление их из игры.
        * 5. Передача нового состояния игры всем клиентам.
        */
        Thread.sleep(100)
     }
  }
}
```

Простейший клиент считывает обновления сервера и рендерит каждый полученный кадр для игрока.

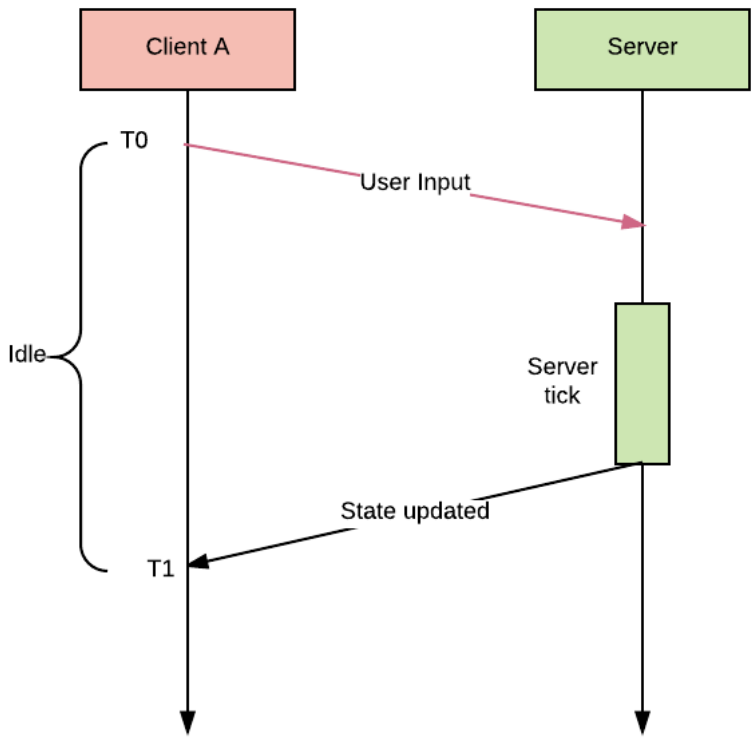
```
class Client {
   def onServerUpdate(state: GameState) = {
      renderGameState(state)
   }
}
```

Обновление состояния с фиксированным шагом

Концепция

Для обеспечения синхронизации всех клиентов проще всего сделать так, чтобы клиент отправлял серверу обновления с фиксированным интервалом. Для примера возьмём интервал в 30 миллисекунд. Обновление содержит введённые пользователем данные, которые могут такж содержать значение *нет вводимых пользователем данных*.

Получив вводимые данные от **всех пользователей**, сервер может перейти к следующему такту с учётом этих данных.



На рисунке выше показано взаимодействие одного клиента с сервером. Надеюсь, проблема для вас настолько же очевидна, как и для меня: может простаивать на интервале от **T0** до **T1**, ожидая для продолжения обновления с сервера. В зависимости от качества сети задержка мож меняться в пределах от 50 до 500 мс, а современные игроки замечают задержки более 100 мс. Поэтому торможение интерфейса пользовател

200 мс будет для некоторых игр *огромной проблемой*.

Это не единственная сложность подхода с фиксированным интервалом.

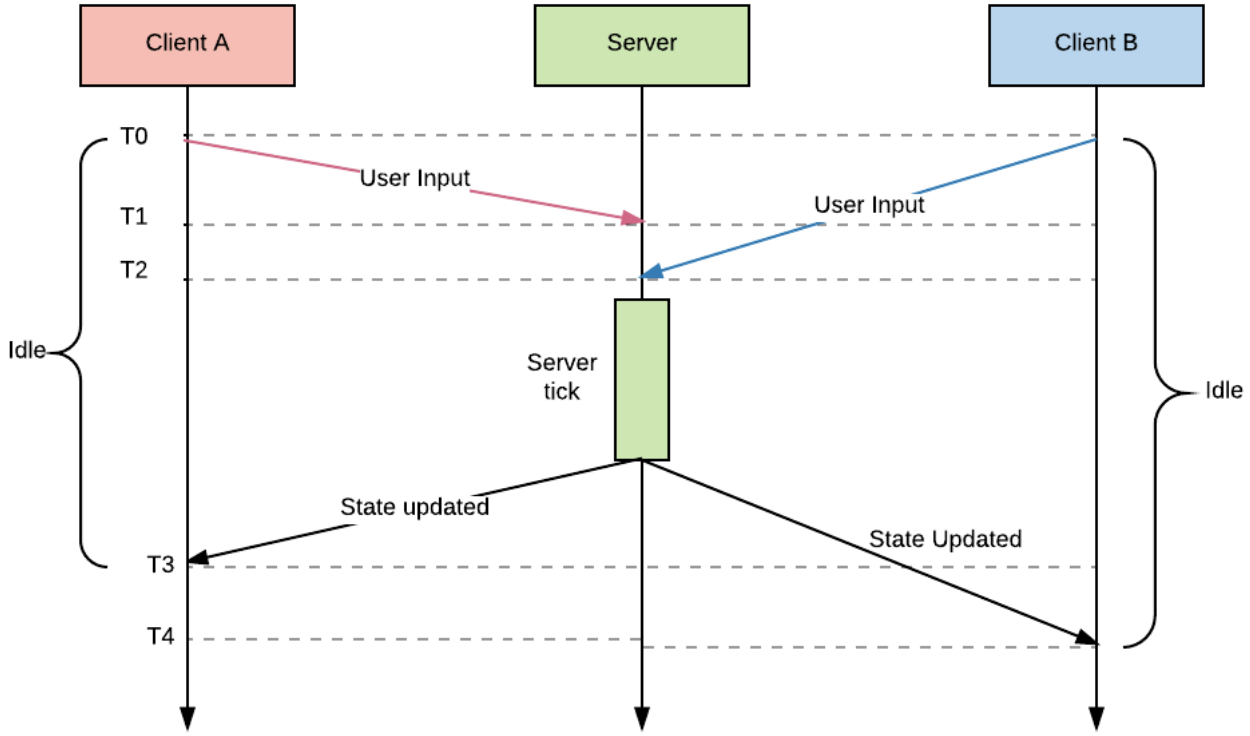


Рисунок выше немного более сложен, он демонстрирует взаимодействие с сервером нескольких клиентов. Видно, что у клиента B более медл
сетевое подключение, поэтому хотя A и B отправляют на сервер вводимые данные в **T0**, обновление от B достигает сервера в **T2**, а не в **T1**.
Поэтому сервер продолжает расчёт только тогда, когда получит все обновления, то есть в **T2**.

Что это значит?
Задержка игры теперь равна задержке самого «лагающего» игрока.
Получается, что мы наказываем всех игроков потому, что у одного из них медленное соединение. Поэтому рано или поздно все игроки уйдут
вашей игры…

Не говоря уже о том, что есть вероятность отсоединения клиента B, которая заблокирует действия сервера до истечения таймаута соединени

Обсуждение

Кроме двух вышеупомянутых проблем, есть ещё несколько:

1. Клиент не будет отвечать, пока не получит обновление состояния от сервера (что ужасно с точки зрения пользователя).
2. Отзывчивость игры зависит от самых «лагающих» игроков. Играете с другом по DSL? Удачи!
3. Соединение будет очень «болтливым»: клиентам нужно регулярно отправлять бесполезные данные, чтобы сервер мог подтвердить, что
   есть вся необходимая для продолжения информация, и это неэффективно.

Во-первых, игры определённого типа имеют иммунитет от этих проблем, например, в большинстве пошаговых игр используются разновиднос
этой модели, потому что клиент должен ждать.

Для медленных игр небольшая задержка тоже приемлема. Хорошим примером может служить **Farm Ville**.

Ещё один хороший пример — **шахматы**, в которых два игрока ходят по очереди и каждый ход длится около 10 секунд.

1. Пользователи должны ждать друг друга по 10 секунд. И они ждут.
2. Два игрока делают ходы по очереди, поэтому задержка одного не влияет на другого.
3. Каждый ход в среднем занимает 5 с (достаточно одного запроса каждые 5 секунд).

Но как насчёт быстрых игр? Например для всех FPS из-за таких проблем решение с фиксированными интервалами не подходит. В оставшейс
части статьи мы узнаем, как решить эти проблемы.

Прогнозирование клиента

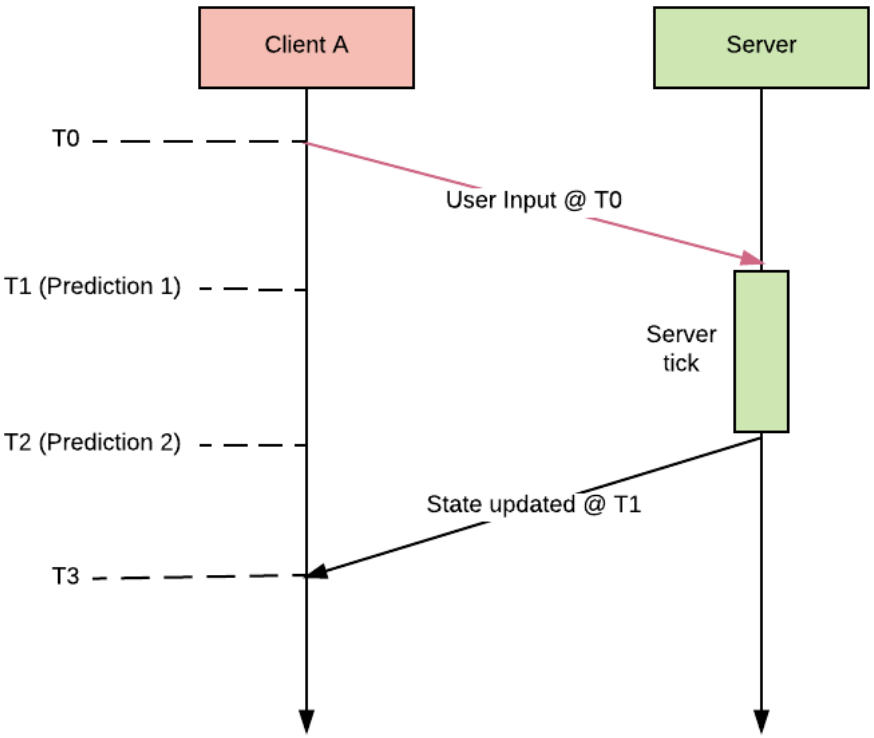
Давайте сначала решим проблему отклика игрока. Игра реагирует через 500 мс после того, как игрок нажал на кнопку, из-за чего игровой п рушится.

Как решить эту проблему?

Концепция

Кое-кто из читателей уже знает ответ: вместо ожидания обновления сервера клиент на самом деле эмулирует игру, выполняя игровую логик локально (т.е. на машине клиента).

Предположим, для расчёта состояния игры в `Tn` нам нужно знать состояние в `Tn−1` и введённые пользователем в `Tn−1` данные.



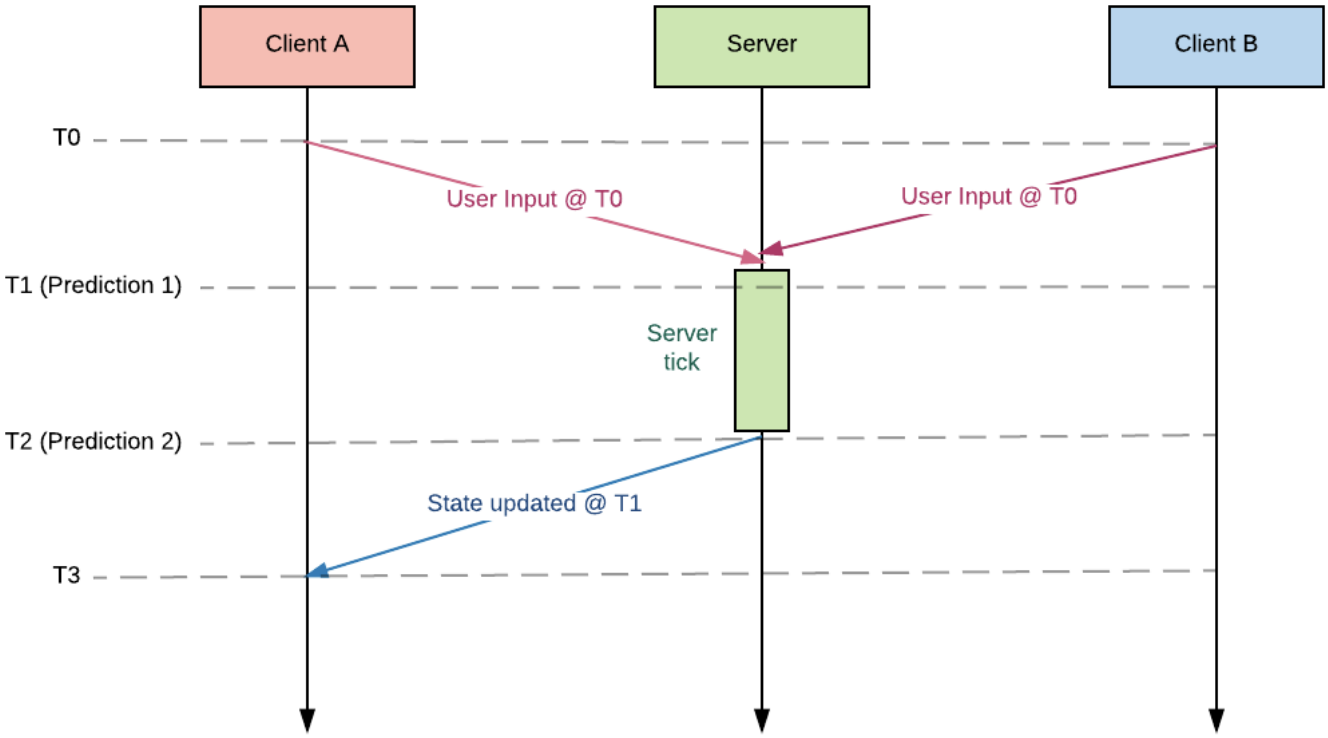
Идея проста: давайте сделаем фиксированную скорость обновления, которая в нашем примере равна *одной единице времени*.

Клиент отправляет вводимые данные на сервер в **T0** для эмуляции состояния игры в **T1**, поэтому клиент затем может рендерить игру, не ожи обновления состояния от сервера, которое будет получено только в **T3**.

Такой подход работает только в следующих условиях:

1. Обновления состояния игры детерминированы, т.е. нет никакой случайности или она прозрачна, и сервер с клиентом могут воспроизво из одинаковых вводимых данных одинаковые игровые состояния.
2. Клиент имеет всю информацию, необходимую для выполнения игровой логики.
3. Примечание: 1 *это не всегда верно,* но мы можем попробовать сделать их как можно более похожими и игнорировать небольшие разли например, вычисления с плавающей запятой на разных платформах, и использовать то же начальное число (seed) для псевдослучайно алгоритма.

Пункт 2 тоже не всегда верен. Я объясню:



На рисунке выше клиент A всё ещё пытается эмулировать состояние игры в **T1** с помощью информации, полученной в **T0**, но клиент B в **T0** у отправил вводимые данные, о которых не знает клиент A.

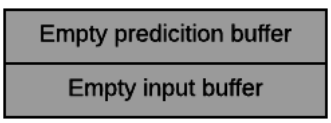
Это значит, что прогноз клиента A о **T1** будет ошибочным. К счастью, поскольку клиент A по-прежнему получает состояние **T1** от сервера, он возможность исправить свою ошибку в **T3**.

Стороне клиента необходимо выяснить, была ли верной предыдущая эмуляция, и как можно разрешить конфликты.

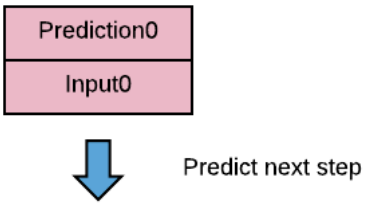
Разрешение конфликтов обычно называется **согласованием (Reconcilation)**.

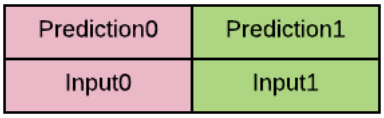
Реализация **согласования** зависит от конкретных условий использования. Я покажу простейший пример, в котором мы просто откажемся от прогнозирования и заменим его точным состоянием, получаемым от сервера.

1. Клиенту нужно хранить два буфера: один для прогнозов, другой для вводимых данных. Его в дальнейшем можно использовать для вычисления прогнозов. Не забывайте, что **состояние Tn** вычисляется исходя из **состояния Tn-1** и **вводимых данных Tn-1**, которые сначала будут пустыми.

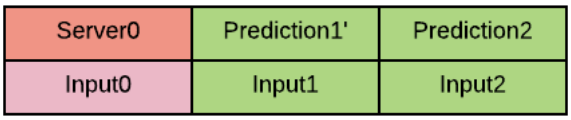


2. Когда игрок нажимает клавишу со стрелкой, вводимые данные сохраняются в InputBuffer, а клиент выполняет прогнозирование, которо затем используется для визуализации. Прогноз сохраняется в PredictionBuffer.

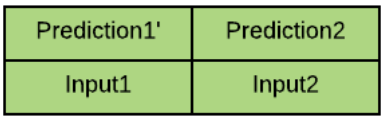




3. В момент получения состояния **State0** от сервера оно не совпадает с прогнозом **Prediction0** клиента, поэтому мы можем заменить **Prediction0** на **State0** и пересчитать **Prediction1**с учётом **Input0** и **State0**.



4. После согласования мы можем безопасно удалить **State0** и **Input0** из буфера. Только после этого мы можем подтвердить, что всё прави



Примечание: согласование имеет недостаток. Если состояние сервера и прогноз клиента отличаются слишком сильно, то при рендеринге мог возникать визуальные ошибки. Например, если мы прогнозируем, что в **T0** враг движется на юг, но в **T3** мы понимаем, что он двинулся на се то согласовываем данные простым использованием состояния с сервера. Враг скачком изменит своё направление, чтобы отобразить правиль положение.

Есть способы справиться с этой проблемой, но они не будут рассмотрены в этой статье.

Обсуждение

Техники прогнозирования на стороне клиента имеет огромное преимущество: клиент работате с собственной частотой обновления (независи частоты обновления сервера), поэтому когда сервер «тормозит», то это не влияет на частоту кадров на стороне клиента.

Но это неизбежно связано с определённой сложностью:

1. Нам нужно обрабатывать больше состояний и логики на стороне клиента (буфер прогнозирования, буфер состояний, логика прогнозирования).
2. Нам нужно определиться, как разрешать конфликты между прогнозом и реальным состоянием на сервере.

И у нас по-прежнему остаются старые проблемы!

1. Ошибки визуализации из-за неверных прогнозов.
2. Частый обмен бесполезными данными.

Заключение

В этой части мы рассмотрели всего два способа реализации сетевого соединения в многопользовательских играх:

1. Обновление состояния с фиксированным шагом
2. Прогнозирование на стороне клиента

Каждый из них имеет свой набор компромиссов, и мы всё ещё не рассмотрели подробнее то, что происходит на стороне сервера.

Интересные статьи по теме

- Серия статей **Gabriel** Fast paced multiplayer: http://www.gabrielgambetta.com/fpm1.html
- Основы сетевой игры **gafferongames**: http://gafferongames.com/networking-for-game-programmers/what-every-programmer-needs-to-know-about-game-networking/
- Статья компании Valve: https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking

Какова роль сервера?

Давайте начнём с определения действий сервера. Типичные задачи сервера:

а) Соединительная точка для игроков
В многопользовательской игре игрокам нужна общая конечная точка для связи друг с другом. Это одна из ролей серверной программы. Даже
модели связи P2P присутствует соединительная точка для обмена сетевой информацией для установки соединения P2P.
б) Обработка информации
Во многих случаях сервер выполняет код симуляции игры, обрабатывает все вводимые игроками данные и обновляет состояние игры. Стоит
учесть, что так бывает не всегда: некоторые современные игры перекладывают большую часть обработки на сторону клиента. В этой статье
считать, что именно сервер несёт ответственность за обработку игры, т.е., например, за создание тактов игры.
в) Единый источник истинного состояния игры
Во многих многопользовательских играх серверная программа также имеет власть над состоянием игры. Основная причина этого — защита о
читерства. Кроме того, гораздо легче ориентироваться, когда есть единственная точка для получения правильного состояния игры.

Наивная реализация сервера

Давайте начнём реализацию сервера самым прямолинейным способом, а затем усовершенствуем его.

Ядром игрового сервера является цикл, выполняющий обновление GameState на основании вводимых пользователями данных. Этот цикл об
называется TICK (такт) и обозначается следующим образом:

$(STATE_n , INPUT_n) => STATE_{n+1}$

Упрощённый сниппет кода сервера может выглядеть так:

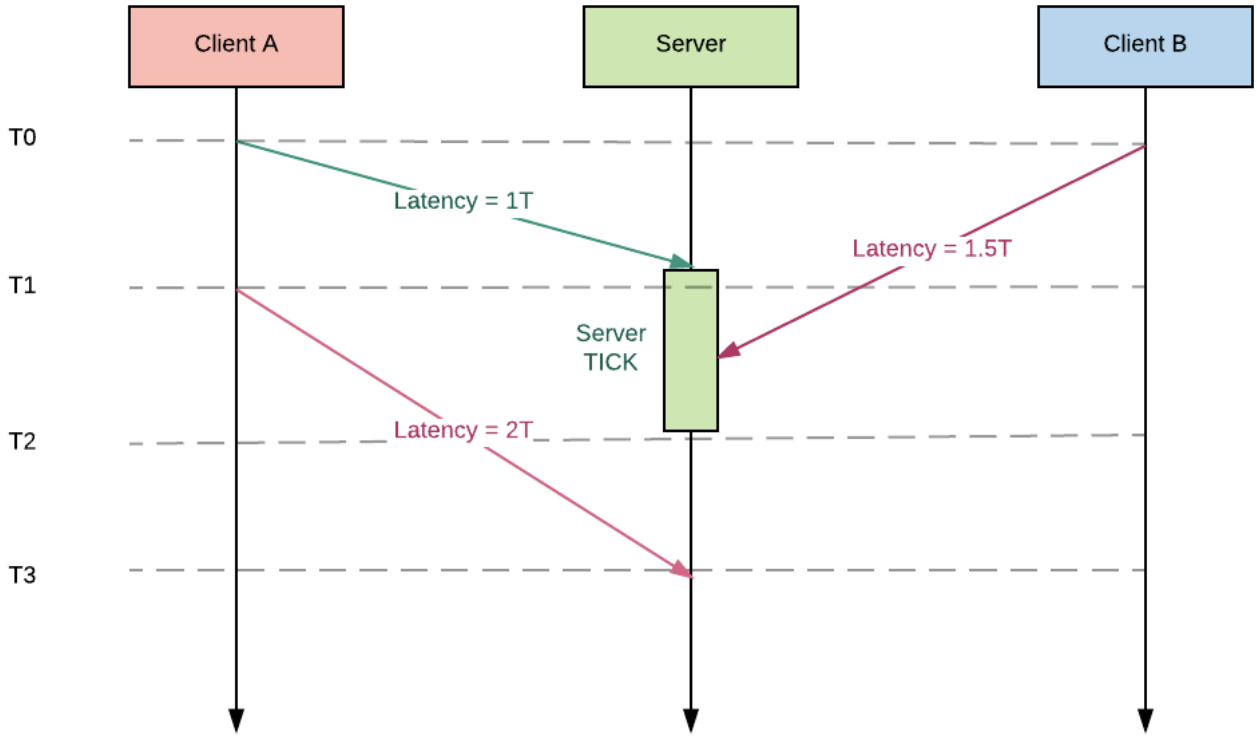
```
def onReceivedInput(i: UserInput) = {
  storeInputToBuffer(i)
}

while(!gameEnded) {
  val allUserInputs = readInputFromBuffer()
  currentState      = step(currentState, allUserInputs)  // т.е. (STATEn , INPUTn) => STATEn+1
  sendStateToAllPlayers(currentState)
}
```

Обсуждение

Надеюсь, сниппет кода выглядит для вас интуитивно понятным и прямолинейным: сервер просто принимает вводимые данных из буфера и
применяет их в следующей функции `TICK` для получения нового состояния GameState. Давайте назовём этот подход *жадным игровым циклом*
потому что он пытается обработать данные как можно быстрее. Это нормально, если не задумываться о нашей несовершенной Вселенной, в
которой солнечный свет достигает Земли за восемь минут.

Здесь снова становится важной задержка.

Тот факт, что сервер обрабатывает вводимые данные из буфера каждый `TICK` означает, что GameState зависит от задержки сети. На схеме н
показано, почему это становится проблемой.



На схеме показаны два клиента, отправляющие вводимые данные серверу. Мы видим два интересных факта.

1. Запросы занимают разное время между разными клиентами и сервером: **1** единицу времени от клиента A до сервера, **1,5** единицы врем клиента B до сервера.
2. Запросы занимают разное время для одного клиента: первый запрос занял **1** единицу времени, второй — **2** единицы времени.

Если говорить вкратце, то задержка непостоянна, даже для одного и того же соединения.

Непостоянная задержка в сочетании с *жадным игровым циклом* приводят к нескольким проблемам. Мы рассмотрим их ниже.

Не работает прогнозировани е на стороне клиента	Если мы не можем спрогнозировать время получения сервером вводимых данных (из-за задержки), мы не можем делать прогнозов с высокой точностью.
Игроки с низкой задержкой получают преимущество	Если вводимые данные быстрее попадают на сервер, то они будут обработаны быстрее, что создаёт нечестное преимущ для игроков с быстрыми сетями. Например, если два игрока одновременно выстрелят друг в друга, они должны будут уб друг друга тоже одновременно, но игрок B имеет меньшую задержку, поэтому убивает игрока A ещё до того, как команд игрока A будет обработана.

Для сглаживания непостоянной задержки существует простое решение — рассмотренное выше обновление с фиксированным шагом. Сервер продолжает вычисления, пока не получит вводимые данные от всех игроков. У такого подхода есть два преимущества:

1. Не требуется прогнозирование на стороне клиента
2. У всех игроков будет та же задержка, что и у самого медленного игрока, что устраняет вышеупомянутое преимущество.

Однако этот подход не работает в быстрых активных играх из-за низкой отзывчивости.

В следующем разделе мы поговорим о том, как заставить сторону сервера работать в быстрых играх.

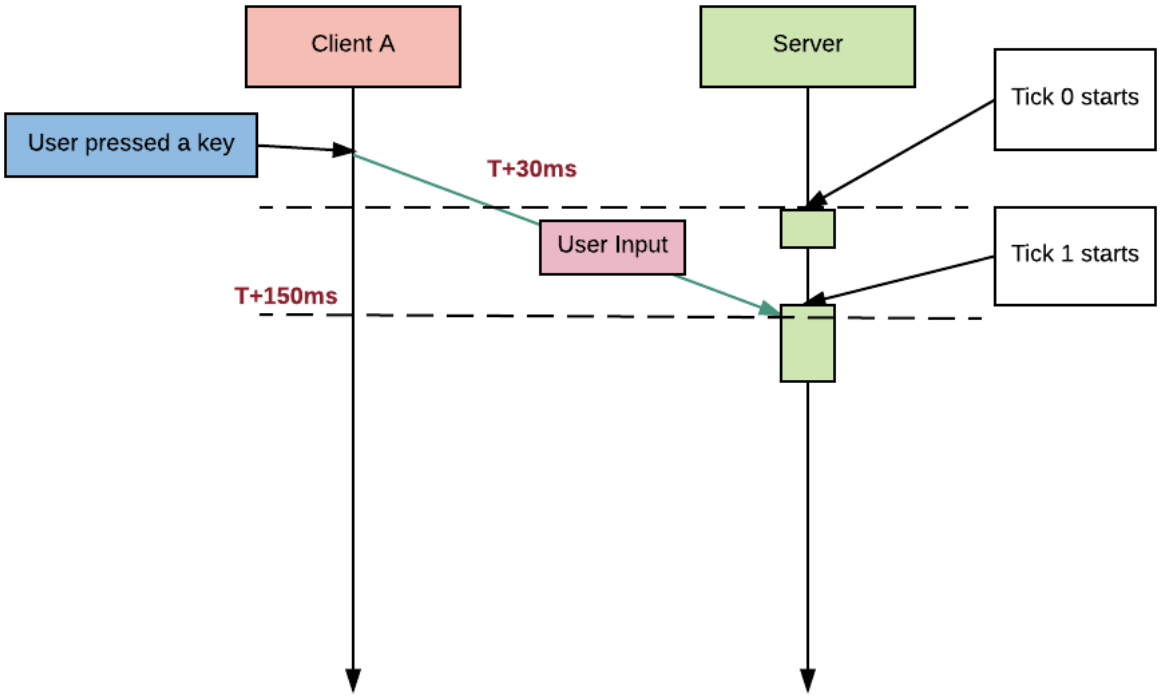
Согласование на сервере

Для решения проблемы неточного прогнозирования на стороне клиента нам нужно сделать клиент-серверное взаимодействие более предсказуемым с точки зрения клиента. Когда игрок нажимает клавишу на стороне клиента, то клиентская программа должна знать, когда эт вводимые данные будут обработаны на стороне сервера.

Один возможный способ реализации этого — **позволить клиенту предлагать, когда необходимо применить вводимые данные**. Таким образом, сторона клиента сможет точно предсказывать время их применения. Термин «предлагать» использован, потому что сервер может отклонить это предложение, если оно неверно, например, игрок пытается произнести заклинание, хотя у него закончилась мана.

Вводимые данные должны применяться почти сразу после ввода данных игроком, например, $\mathbf{T_{input} + X}$, где X — задержка. Точное значение зависит от игры, для отзывчивости обычно необходима задержка менее 100 мс. Заметьте, что X может быть и нулём. В таком случае данные применяются сразу же после ввода пользователем.

Давайте примем X = 30 мс, что примерно равняется одному кадру при 30 кадрах в секунду. Для передачи на сервер данным требуется 150 м тогда существует большая вероятность того, что когда вводимые данные достигнут сервера, кадр для ввода уже будет пропущен.



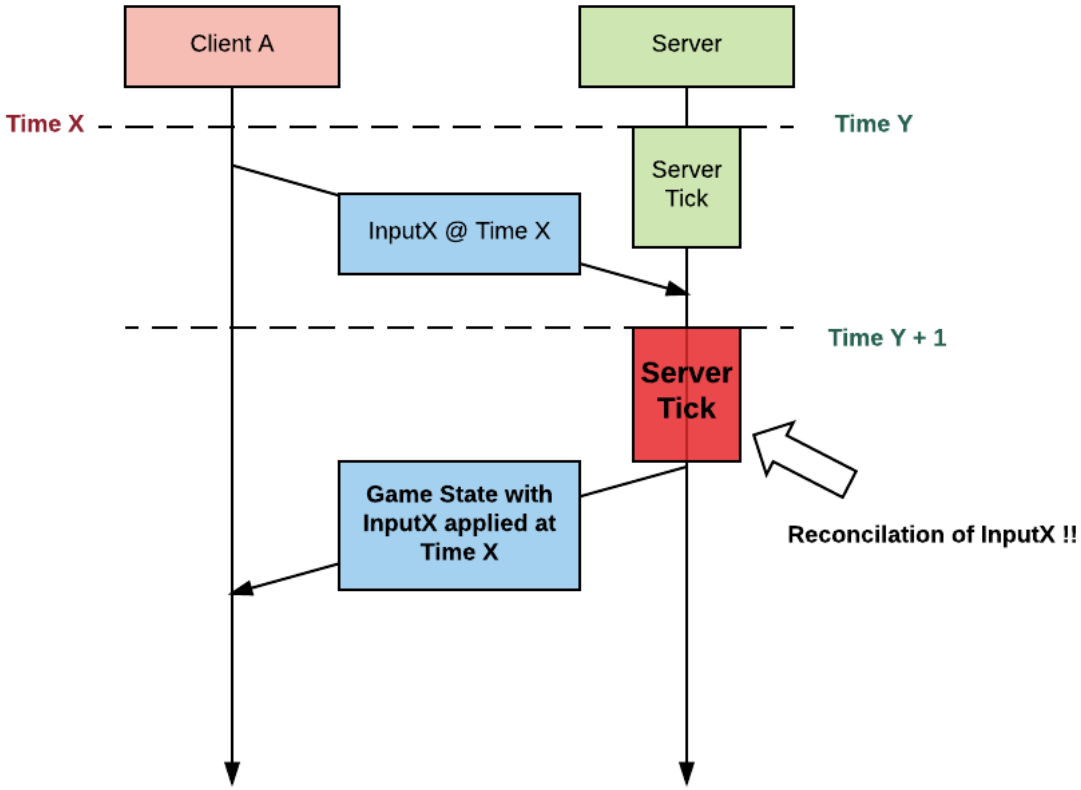
Посмотрите на схему: пользователь A нажал клавишу в **T**. Эти данные должны обработаться в **T + 30 мс**, но вводимые данные из-за задерж получены сервером в **T + 150 мс**, что уже находится за пределами **T + 30 мс**. Решением этой проблемы мы займёмся в данном разделе.

Как сервер применяет вводимые данные, которые должны были случиться в прошлом?

Концепция

Вы наверно помните, что прогнозирование на стороне клиента имело ту же проблему с неточными прогнозами из-за недостатка информации противниках. Неверные прогнозы позже корректировались обновлениями состояния с сервера с помощью согласования. Ту же технику можн использовать здесь. Единственная разница в том, что мы исправляем GameState на сервере на основе данных, вводимых клиентами.

Все вводимые пользователями данные должны иметь метки времени. Эти метки используются для того, чтобы сообщить серверу, когда их ну обрабатывать.



*Примечание: на первой пунктирной линии **Time X** на стороне клиента, но **Time Y** на стороне сервера. Это интересная особенность многопользовательских игр (и многих других распределённых систем): поскольку клиент и сервер работают независимо, время на клиенте и сервере обычно отличается. Наш алгоритм позволяет справиться с этой разницей.*

На схеме выше показано взаимодействие между одним клиентом и сервером.

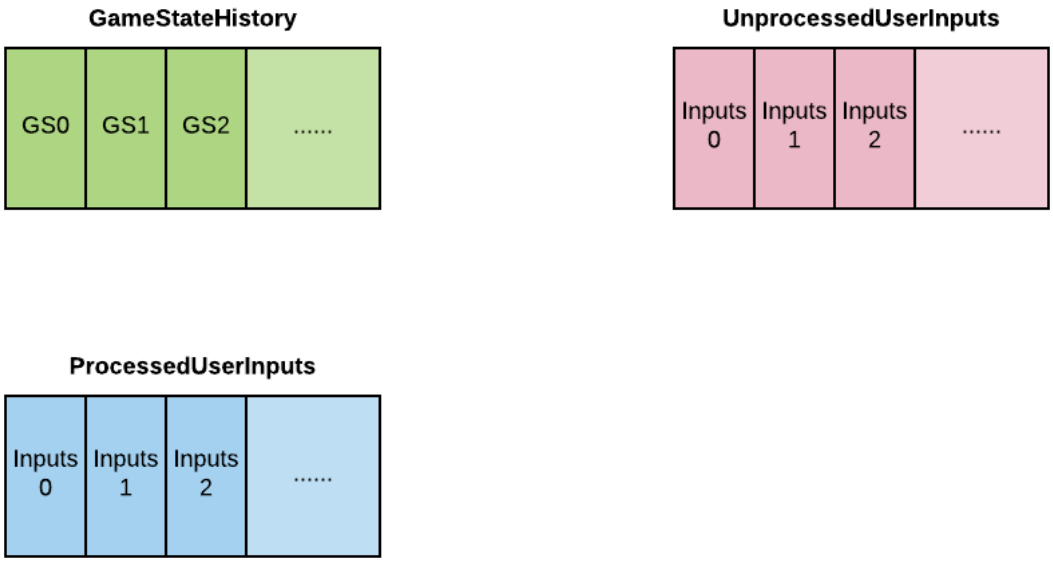
1. Клиент отправляет вводимые данные с меткой времени, сообщая серверу, что эти данные клиента A должны произойти в Time X.
2. Сервер получает запрос в Time Y. Положим, что Time X раньше, чем Time Y. При разработке алгоритма надо принять, что Time Y больше меньше Time X, это обеспечит нам большую гибкость.
3. **Красное поле** — это момент выполнения согласования. Сервер должен применить Input X к последнему состоянию игры, чтобы казало ввод Input X произошёл в Time X.
4. Передаваемое сервером GameState тоже содержит метку времени, которая необходима для согласования и стороны сервера, и стороны клиента.

Подробности согласования (**красное поле**)

1. Сервер должен хранить

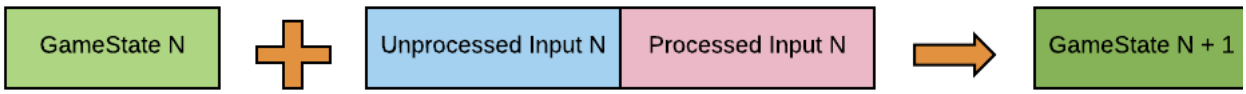
    - **GameStateHistory** — историю состояний GameState в течение кадра времени **P**, например, все GameState за последнюю секунду
    - **ProcessedUserInput** — историю вводимых данных UserInput, обработанных за кадр времени **P**, например, то же значение, что и времени GameStateHistory

- **UnprocessedUserInput** — полученные, но ещё не обработанные UserInput, тоже в кадре времени **P**



2. Когда сервер получает от пользователя вводимые данные, они должны вставляться в **UnprocessedUserInput**.
3. Затем, в следующем кадре сервера

    1. Выполняется проверка на наличие вводимых данных в **UnprocessedUserInput**, которые старше текущего кадра
    2. Если их нет, то всё в порядке, просто выполняется игровая логика с последним GameState и соответствующими вводимыми данны (при их наличии), и трансляция клиентам.
    3. Если они есть, то это значит, что часть ранее сгенерированных игровых состояний ошибочна из-за отсутствующей информации, и нужно это исправить.
    4. Сначала нам надо найти самые старые необработанные вводимые данные, допустим во время Time N, (подсказка: эта операция выполняется быстро, если **UnprocessedUserInput** отсортирован).
    5. Затем нам нужно получить соответствующее состояние GameState в Time N из **GameStateHistory** и обработанные вводимые данн Time N из **ProcessedUserInput**
    6. С помощью этих трёх фрагментов данных мы можем создать новое, более точное GameState.

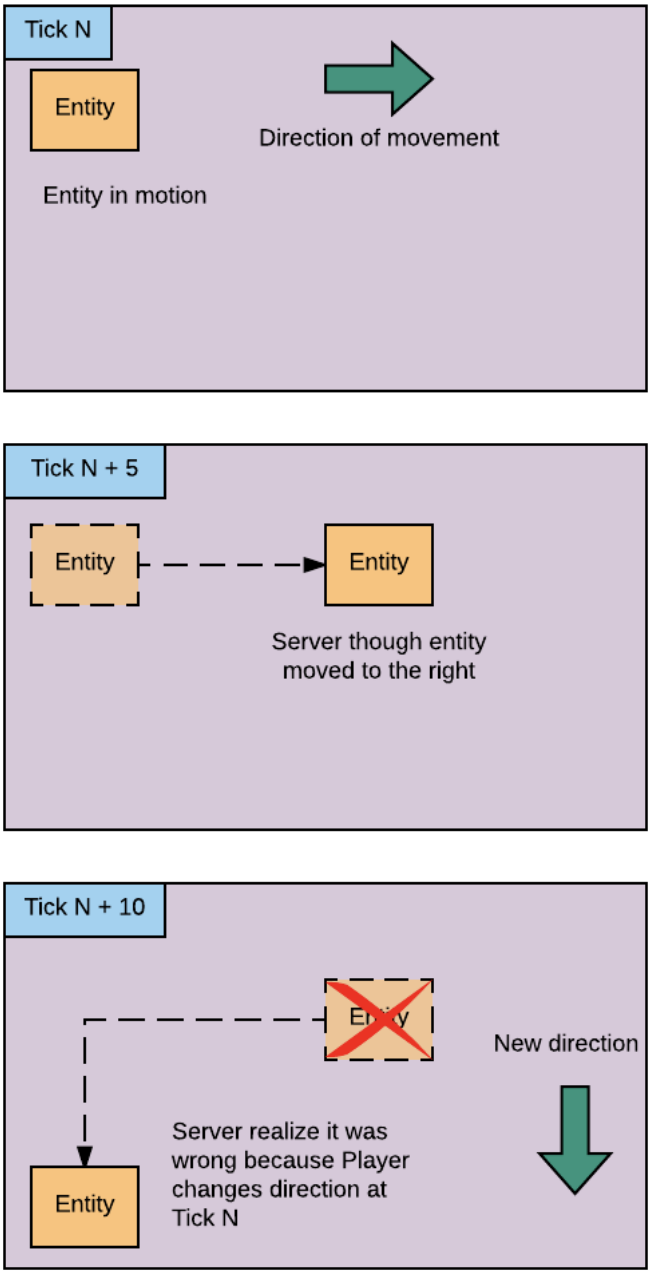


    7. Затем перемещаем необработанные вводимые данные Unprocessed Input N в ProcessedUserInput, чтобы можно было использовать будущем для согласования.
    8. Обновляем GameState N в **GameStateHistory**
    9. Повторяем шаги с 4 по 7 для N+1, N+2 ..., пока не получим последнее GameState.
    10. Сервер отправляет свой последний кадр всем игрокам.

Обсуждение

Согласование на стороне сервера страдает от тех же проблем, что и согласование в клиенте. Когда нужно согласование, значит, мы сделали не так, и мы исправляем ошибку, меняя историю. Это значит, что мы не можем применить необратимые последствия, например, убийство игр Такие необратимые последствия можно применять только когда они поступают из **GameStateHistory**, т.е. когда их больше нельзя перезапис

Кроме того, неверные GameState иногда приводят к ужасным скачкам UI. На схеме ниже показано, как это происходит.

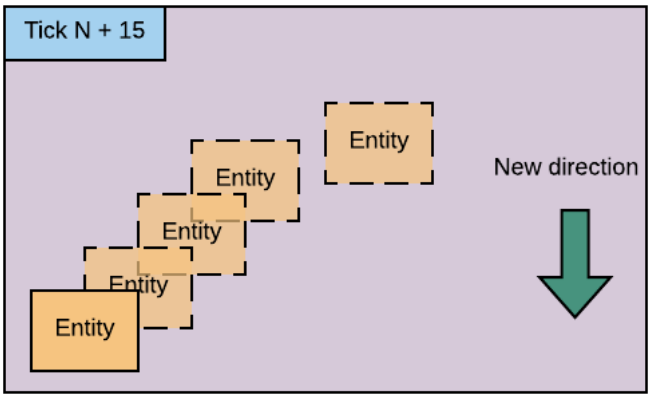


Объект сначала находится в левом верхнем углу и двигается вправо. Спустя пять таков он смещается вправо, но затем сервер получает введ пользователем данные, сообщающие, что объект изменил направление в Tick N, поэтому сервер согласует состояние игры. При этом объект внезапно **перескакивает** в левый нижний угол экрана.

Возможно, я преувеличиваю это влияние, иногда объект двигается не так далеко и скачок менее заметен, но во многих случаях он всё равно очевиден. Мы можем контролировать скачки, меняя размер **GameStateHistory**, **UnprocessedUserInput** и **ProcessedUserInput**. Чем меньш размер буфера, тем меньше будут скачки, потому что мы будем менее терпимы к сильно запаздывающим вводимым данным. Например, если вводимые данные запаздывают более чем на 100 мс, то они игнорируются, а игрок с пингом > 200 мс не сможет играть в игру.

*Мы можем пожертвовать **толерантностью к задержкам сети** для более **точного обновления состояния игры**, или наоборот.*

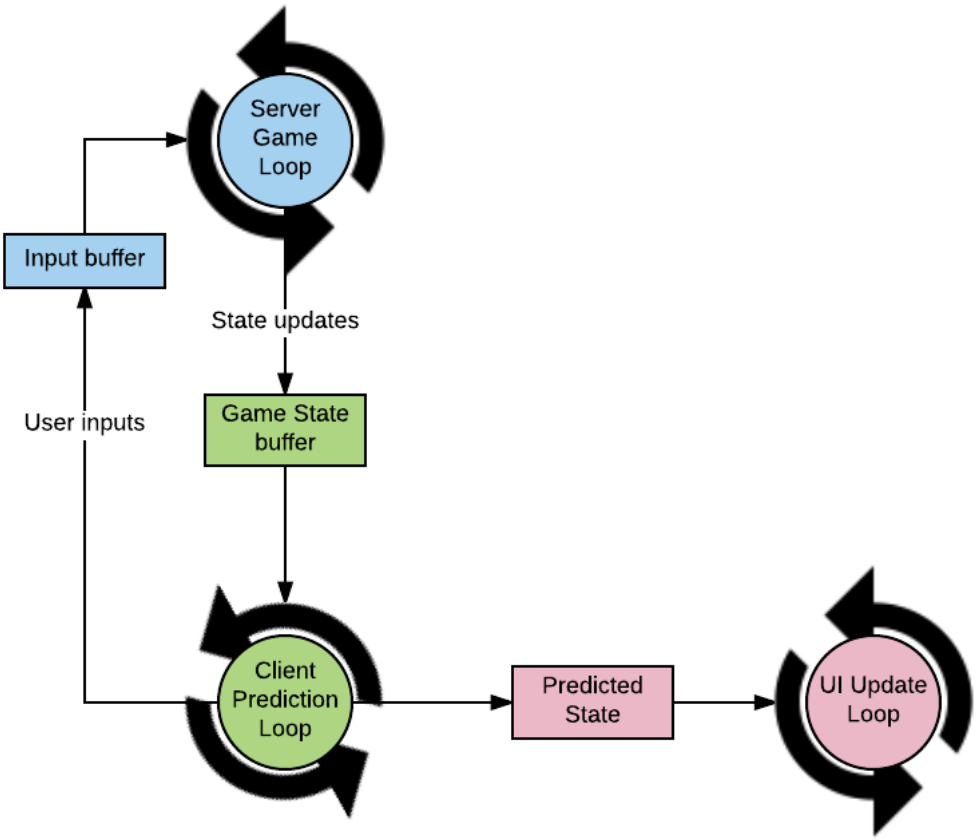
Есть одна популярная техника для борьбы с проблемой неточных Game State — это **интерполяция объектов** (**Entity Interpolation**). Идея заключается в сглаживании скачков растягиванием их на короткие промежутки времени.



В этой статье я не буду описывать подробности реализации **интерполяции объектов**, однако приведу полезные ссылки в конце.

Подводим итог

Мы обсудили способы работы клиентов и сервера в многопользовательских играх.



В целом, многопользовательская игра содержит три свободно соединяемых цикла: **серверный игровой цикл**, **клиентский цикл прогнозирования** и **клиентский цикл рендеринга UI**. Создав между ними буфер, можно разделить процесс их выполнения, что обеспечи нам гибкость в создании более качественного игрового процесса.

Заключение

На этом заканчивается моя статья о многопользовательских играх. Многое по теме я узнал от специалистов в этой области, также мне очень пример простой многопользовательской игры. Я показал только один способ реализации многопользовательского сервера, есть и другие. Вы подходящего зависит от типа создаваемой вами игры. Рекомендую вам изучить некоторые подходы, создав простую игру.

Спасибо за чтение, удачного хакинга!

Ссылки и дополнительное чтение

- [Интерполяция объектов] — http://www.gabrielgambetta.com/fpm3.html
- [Интерполяция объектов] — http://gafferongames.com/networked-physics/snapshots-and-interpolation/
- [Компенсация лагов] — https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking#Lag_compensation

многопользовательские игры, multiplayer, синхронизация

4,9k 153

Перевод: Qing Wei

@PatientZero карма 239,5 рейтинг 512,1

Переводчик-фрилансер

ПОХОЖИЕ ПУБЛИКАЦИИ

25 июля 2013 в 20:20

Создание многопользовательской realtime игры на node.js

+54 42,9k 425 47

9 апреля 2013 в 17:14

Мысли о будущем компьютерных игр

+5 22,8k 59 40

29 сентября 2012 в 11:04

Игра MMO на карте комикса Click and Drag

+44 18,3k 69 18

САМОЕ ЧИТАЕМОЕ

Разраб

Сейчас | Сутки | Неделя | Месяц

Kotlin vs. Java: скорость компиляции

+35 6,7k 20 11

Kotlin для Android: Теперь официально

+41 7,5k 29 41

Все образовательные проекты Mail.Ru Group

+32 6,6k 131 15

Scheduling: мифы и реальность. Опыт Яндекса

+76 11,1k 78 7

Синхронизация состояний в многопользовательских играх

+34 4,9k 152 6

Комментарии (6)

ertaquo 18 мая 2017 в 22:39 #

В игре Teeworlds именно так все и сделано. Есть некое «ядро» игры, в котором ведутся все основные расчеты физики мира. Это ядро используется одновреме
сервере и на клиенте (для предсказаний). Вдобавок файлы с его кодом используются при расчете специального хеша. При подключении клиента к серверу эти
сверяются, и если они не совпадают, значит на клиенте и на сервере используются разные, скорее всего несовместимые версии игры.
Правда, последний момент не слишком удобен для игр без автообновления. В некоторых версиях разработчики сами костыль ставили, чтобы передавался ста
хеш.

alexoron 18 мая 2017 в 23:14

Лучше у Варгейминга статью на эту тему напишите.
Насколько помню серваков у них только для одних танков 10 штук.

molnij 19 мая 2017 в 11:41

Танки это скушно :)
Во-первых скорость по меркам FPS/MMORPG низкая, во-вторых количество игроков в сессии жестко лимитировано.

vanxant 19 мая 2017 в 03:13

Ну есть ещё чисто «игромеханические» трюки, позволяющие сглаживать скачки. В первую очередь анимации действий, тех же скиллов, выстрелов, резких повс
т.д. Ну, вы же не можете в реальности бежать влево и внезапно, тут же, побежать вправо. У вас будет вполне себе задержка на торможение, разворот и разгон.
время можно и нужно закладывать в анимации. У игроков с быстрым пингом анимации отработают как надо, у лагающих — будут дёргаться, но сами позиции о
не будут слишком сильно скакать.

xandr 19 мая 2017 в 12:25

В качестве классического примера решения задачи предсказания действий игрока интересующимся можно посоветовать взглянуть на механику
многопользовательской игры первого Quake. Вот обзор его исходного кода, смотреть часть «Прогнозирование». Насколько мне известно, это было одно из пер
решений такого рода в игровой индустрии.

gresolio 19 мая 2017 в 13:34

Перечитывал этот абзац много раз, что-то трудно воспринимается:

> Это значит, что мы не можем применить необратимые последствия, например, убийство игроков. Такие необратимые последствия можно применять только
> они поступают из GameStateHistory, т.е. когда их больше нельзя перезаписывать.

> This means we cannot apply irreversible outcomes, i.e, killing a players, such irreversible outcomes will only be applied when it goes out of the GameStateHistory, ie. v
> cannot be rewriten anymore.

Кто в теме, объясните пожалуйста подробней, а то чувствую от меня ускользает полное понимание этой идеи.

Логично выглядит так, что если в серверном буфере GameStateHistory например в GameState N записано что определённого игрока уже замочили, а затем про
согласование на сервере (появились данные в UnprocessedUserInput которые старше текущего кадра) и будут обновлены GameState N+1, N+2,… — то этот игр
никак не может «ожить», это необратимое последствие, перезаписывать в этой ситуации нельзя.

А если наоборот, например в GameState N записано что определённый игрок жив, но затем после согласования (появились данные в UnprocessedUserInput кот
старше текущего кадра, скажем от клиента с медленным соединением) оказывается что в следующем GameState N+1 его замочили, то перезаписывается, нел
ему оставаться бессмертным :) Это будет выглядеть «как смерть из-за угла», у Valve доках это следствие применения Lag Compensation: For instance, if a highly
player shoots at a less lagged player and scores a hit, it can appear to the less lagged player that the lagged player has somehow «shot around a corner».

Правильно ли, что описанная тут методика согласования, является попыткой обобщить Lag Compensation, который специфичен именно для шутеров?

1,9k 27 6