Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

КОМПЕНСАЦІЯ ЗАТРИМОК ТА ВТРАТИ ПАКЕТІВ В ДИНАМІЧНИХ ОНЛАЙН ІГРАХ

Текстова частина до курсової роботи
за спеціальністю „Інженерія програмного забезпечення” 121

Керівник курсової роботи
к.ф.-м.н_____
(прізвище та ініціали)

_____
(підпис)

“____” _____ 2020 р.

Виконав студент

_____
(прізвище та ініціали)

“____” _____ 2020 р.

Київ 2020

Оглавление

Анотація

Об’єктом моєї курсової роботи є дослідження та власна реалізація алгоритмів, які використовують в сучасних онлайн іграх з топологією “Клієнт-Сервер” для компенсації затримок та втрати пакетів під час їх передачі в мережі. Метою проекту є створення онлайн гри, в якій наглядно продемонстровано вплив затримок на сприйняття гри клієнтами та вирішення цієї проблеми шляхом їх компенсації.

Під час виконання роботи було досліджено декілька шляхів компенсації затримок в мережі, результати їх застосування в онлайн іграх та обгрунтовано як, де і коли варто використовувати той чи інший підхід.

В результаті було створено онлайн гру (топології Клієнт - Сервер), в якій шляхом використання алгоритмів задля компенсації затримок вдалося мінімізувати час передачі / підтвердження отримання пакетів.

.

Перелік прийнятих скорочень

RTT (англ. Round Trip Time) - час передачі/підтвердження; час, за який виконується передача пакету від одного вузла до іншого та для зворотної передачі пакету відповіді.

Затримка – інтервал часу між ідентифікованою причиною та спостерігаємим ефектом.

RPC (англ. Remote Procedure Call) - клас технологій, що дозволяють комп'ютерним програмам викликати функції або процедури в іншому адресному просторі (на віддалених комп'ютерах або в незалежній сторонньої системі).

MTU (Maximum Transmission Unit) – максимальний розмір пакету Ethernet.

FCS (Frame Check Sequence) – контрольна послідовність кадру, яка зберігає значення контрольної суми, згенерованої на основі двох полів адрес source та destination, поля розміру пакету (довжини), фактичних даних та доповнення.

Вступ

Ще пару десятків років тому швидкість передачі даних в мережі вимірювалась в кілобайтах за секунду, і вже тоді розробники онлайн ігор мали певні проблеми з втратою пакетів та затримками при передачі. Зараз швидкість в сотні разів більша, а проблема компенсації затримок така ж актуальна. Мережеві онлайн ігри займають дуже впливове місце в сучасній ігровій індустрії, кількість гравців, які можуть одночасно брати участь в одній грі може перевищувати десятки тисяч гравців. Навіть сучасні технології не дають можливості повністю уникнути затримок, а користувачі завжди будуть хотіти, щоб при якійсь дії іншого користувача вони одразу ж отримували актуальні дані про це.

Проблема затримок є дуже актуальною темою, адже при затримці пакетів на 16-100 мс (не для всіх видів ігор, для шахів онлайн затримка навіть у 500 мс може бути не критичною) гравець починає відчувати погану чуйність гри, і це відштовхує його. Також проблема посилюється тим, що у клієнтів є немережеві фактори, які також впливають на загальну затримку, і вони зазвичай непідвласні розробникам ігор. Через затримку конвеєра зображень, циклу вводу затримка даних може складати 5-10 мс, а то і більше, в залежності від потужності CPU та GPU. Через це, бажання мінімізувати затримку в мережі є ще більш важливою, адже навіть при мінімальних значеннях загальна затримка від кліку миші до відображення дії клієнта у інших клієнтів може наближатись до того порогу, при якому сприйняття гри погіршується.

Тож за мету курсової роботи було поставлено створити проект, який би в показав вплив мережевих затримок на онлайн гру та способи їх компенсування. В якості мови програмування було обрано c++, адже брати якусь високорівневу мову програмування для демонстрації цієї проблеми не дуже практично, а в якості движку було обрано UE4, як один з движків, що підтримує c++.

Робота складається з трьох розділів.

Перший розділ присвячено теорії щодо передачі пакетів в мережі, протоколам передачі даних та причинам виникнення затримок.

Другий розділ присвячено питанню втрати пакетів та способу його вирішення.

В третьому розділі буде описано як можна вирішити ці проблеми. Ми торкнемося теми інтерполяції та прогнозування на стороні клієнта, реплікації об’єктів та на прикладі онлайн шутеру спробуємо мінімізувати онлайн затримки.

Постановка задачі:

1. Розібрати передачу даних в мережі, відштовхуючись від моделі TCP/IP та причини виникнення затримок на кожному рівні моделі:
    - Фізичний рівень;
    - Канальний рівень;
    - Мережевий рівень;
    - Транспортний рівень;
    - Прикладний рівень;
2. Розібрати основні методи компенсації втрати пакетів:
    - Визначити як реплікувати об`єкти;
    - Вирішити проблему втрати пакетів;
3. Розібрати основні методи компенсації затримок:
    - Розглянути інтерполяцію на стороні клієнта;
    - Розглянути прогнозування на стороні клієнта.

Розділ 1: Передача даних в мережі

1.1 Рівень TCP/IP

Сімейство TCP / IP одночасно вражає своєю красою і лякає. Його краса обумовлена тим, що в теорії це сімейство складається з декількох незалежних абстрактних рівнів, кожен з яких підтримується безліччю протоколів які можна взаємозамінювати між собою, кожен з яких в свою чергу вирішує відповідні завдання та передає дані.

Кожен рівень вирішує певні завдання і задовольняє потреби рівня, розташованого безпосередньо над ним. У ці завдання входять:

- прийом блоку даних для передачі від вище лежачого рівня;
- упаковка даних з додаванням заголовку (Header) і, іноді, колонтитула (Footer) даного рівня;
- передача даних нижче лежачому рівню для подальшої обробки;
- прийом переданих даних від нижчого рівня;
- розпакування даних та видалення заголовку;
- передача даних вищому рівневі для подальшої обробки;

На Рисунку 1.1 зображено модель TCP/IP з точки зору розробника ігор

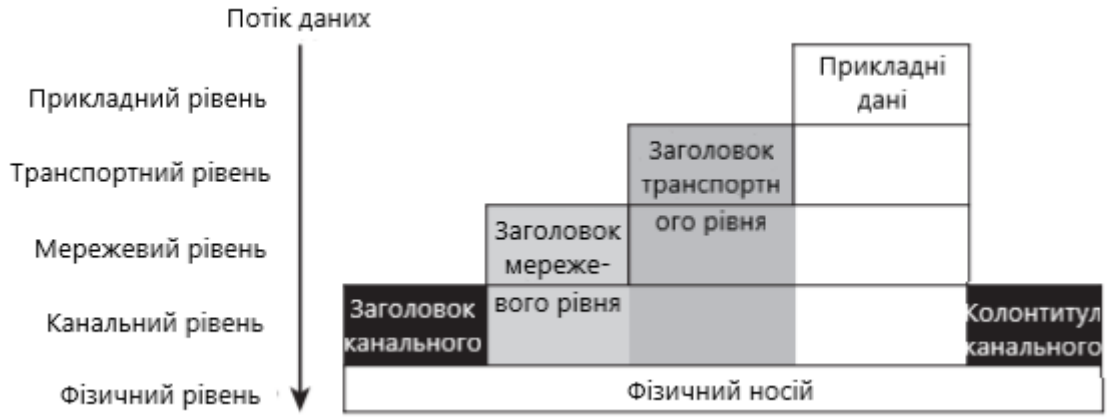

Рисунок 1.1 – Багаторівнева модель TCP/IP

Однак модель не описує, як кожен рівень буде вирішувати свої завдання. В реальності існує безліч протоколів, які можна використовувати на кожному рівні. Знайомі з об'єктно-орієнтованим програмуванням можуть розглядати кожен рівень як інтерфейс, а кожен набір протоколів - як реалізацію цього інтерфейсу

1.2 Рівень TCP/IP: Фізичний рівень

У самому низу моделі TCP/IP знаходиться найбільш простий підтримуючий рівень – фізичний. Його задача – забезпечити фізичний зв`язок між вузлами в мережі. І тут вже виникає перше обмеження на швидкість передачі даних – незалежно від типу носія, інформація не може поширюватись швидше, ніж швидкість світла. Через це виникає перша причина затримок – *затримка розповсюдження*. Вона складає приблизно 3.301 нс за кожен метр, який проходить пакет в мережі. Це означає, що для того, щоб пакет подолав відстань від Києва до Вашингтону необхідно 25,8 мс, і це при найкращих умовах. З цього можна зробити висновок, що розгортати сервер бажано там, звідки відстань до користувачів не буде перевищувати 5000 кілометрів, тоді затримка розповсюдження не буде перевищувати 15 мс. Якщо ж гравці знаходяться по всьому світу, тоді бажано розгорнути декілька серверів для кожного регіону. Поки що немає можливості якось по-інакшому вплинути на цю затримку, тому обирати локацію серверу потрібно обачливо.

1.3 Рівень TCP/IP: Канальний рівень

Канальний рівень - це те місце, звідки бере початок логічна частина моделі. Його завдання - реалізувати спосіб взаємодії між фізично зв'язаними вузлами. Тобто канальний рівень повинен надавати метод, за допомогою якого один вузол може упакувати інформацію і передати її на фізичному рівні так, щоб інший вузол

міг, прийнявши пакет, відновити з нього передану інформацію. Одиниця передачі даних на цьому рівні називається кадром, а в задачі канального рівня входять:

- визначення способу ідентифікації вузлів, щоб кадр міг досягти адресата;
- визначення формату фрейму, що включає адресу одержувача і source дані;
- визначення максимального розміру кадру щоб верх лежачі рівні знали, який обсяг даних можна відправити в одному пакеті;
- визначення способу фізичного перетворення кадру в електричні сигнали;

Також, доставка кадру вузлу призначення лише можлива, але на даному рівні вона не гарантована. Існує безліч факторів, що впливають на цілісність цього електричного сигналу. Пошкодження фізичного носія, електромагнітні перешкоди та відмова обладнання легко можуть призвести до втрати кадру. Канальний рівень *не передбачає* застосування будь-яких додаткових зусиль, щоб визначити факт доставки кадру адресату або повторну його передачу, якщо доставка не була успішною. Тому взаємодії на канальному рівні називають ненадійними. Будь-який високорівневий протокол, який гарантує надійну доставку даних, повинен сам реалізувати такі гарантії.

Для кожного різновиду фізичного носія, обраного для реалізації фізичного рівня, є відповідний протокол або набір протоколів, що забезпечують всі необхідні функції на канальному рівні. Я не буду перераховувати усі, а торкнусь лише одного сімейства протоколів – Ethernet. Для розробника тут найважливішим буде те, що стандарт Ethernet визначає максимальний об`єм фактичних даних (MTU), який дорівнює 1500 байт. Також дуже важлива особливість полягає в тому, що не дивлячись на те, що Ethernet не гарантує доставку, він гарантує запобігання доставки пошкоджених даних.

Для цього рівня також є затримка, яку називають *затримкою на передачу*. Після того, як маршрутизатор отримав пакет, для того, щоб передати його далі маршрутизатор має інтерфейс канального рівня. Протокол канального рівня визначає з якою швидкістю біти будуть записуватись в носій. І тут і виникає наступне обмеження в швидкості – час, необхідний на запис бітів в фізичний

носій. 1-мегабітне Ethernet з`єднання дозволяє передавати в кабель приблизно один мільйон бітів за секунду. Отже, щоб записати повний пакет з 1500 байтів, при такому з`єднанні необхідно 12 мс.

Для того, щоб зменшити цю затримку дуже важливо використовувати високошвидкісне інтернет з`єднання. Також один з способів зменшити загальну затримку для усіх пакетів – передавати пакети якомога більшого розміру, завдяки чому зменшується вплив заголовків на затримку на передачу.

1.4 Рівень TCP/IP: Мережевий рівень

Завдання мережевого рівня - забезпечити інфраструктуру для логічної адресації, щоб можна було легко міняти апаратні компоненти вузла, групувати вузли в окремі підмережі і дозволяти вузлам у віддалених підмережах, що використовують різні фізичні носії та протоколи канального рівня, передавати повідомлення один одному. В даний час для реалізації необхідних функцій мережевого рівня найбільш широко використовується інтернет-протокол версії 4 (IPv4) та інтернет-протокол версії 6 (IPv6).

Для розробника на цьому рівні важливо те, що максимальний розмір пакету IPv4 дорівнює 65535 байтам, коли MTU дорівнює 1500 байтам. А як ми знаємо, пакети мережевого рівня будуть загортатись в кадри на канальному рівні, і через те, що розмір IPv4 пакету може значно перевищувати значення MTU, існує механізм фрагментації, який при отриманні пакету з розміром більшим за MTU, розіб`є його на менші, розмір яких буде дорівнювати MTU. Такий механізм передбачає наявність додаткових полів в Header`і пакету, а саме поле ідентифікатору фрагменту, поле зміщення та поле флагів. Сам Header займає цілих 20 байт, через що для фактичних даних залишається 1480, що дуже важливо пам`ятати. Якщо ж використовується IPv6 протокол, то там фрагментація відсутня, пакети що перевищують значення MTU будуть просто скинуті, тобто

треба піклуватись, щоб значення розміру пакету завжди не перевищувало 1500 байт.

Також варто пам`ятати, що пакети можуть надходити не в правильному порядку та навіть декілька разів через те, що маршрутизатор відправляв їх по різним маршрутам.

Але на цьому рівні можна виокремити такі дві причини виникнення затримок: *затримка на обробку* та *затримка в черзі.* Перша спричинена тим, що маршрутизатор читає пакети з мережевого інтерфейсу, перевіряє IP-адресу отримувача, визначає наступний хост куди передати пакет та виводить пакет в відповідний інтерфейс. І час, необхідний на перевірку та дослідження адреси отримувача та визначення подальшого маршруту і є причиною *затримки на обробку*. Також іноді необхідно виконати додаткові дії, такі як перетворення IP адрес та шифрування даних, на що також необхідний час. Друга затримка спричинена тим, що маршрутизатор має обмежену пропускну здатність. Якщо пакети занадто швидко поступають до мережевого інтерфейсу, вони розміщуються в приймальну чергу, звідки будуть оброблені. І навпаки, якщо ж занадто багато пакетів необхідно відправити, маршрутизатор розміщує їх в вихідну чергу, адже мережевий інтерфейс може відправляти пакети лише по одному.

Затримку на обробку можна вважати не в компетенції розробника, але насправді в сучасних маршрутизаторах все дуже оптимізовано, тому ця затримка в більшості випадків не перевищує 1 мс.

Затримку в черзі можна зменшити мінімізувавши затрати на обробку та передачу. Варто пам`ятати, що ця затримка однакова як для пакету розміром 1500 байт, так і для пакету розміром 100 байт. Через це замість 15 пакетів розміром 100 байт краще відправляти один пакет розміром 1500 байт, тим самим уникаючи накопичення пакетів в черзі.

1.5 Рівень TCP/IP: Транспортний рівень

Задача транспортного рівня - забезпечити можливість взаємодії окремих процесів, що виконуються на цих вузлах. Так як на одному вузлі може виконуватися безліч процесів, не завжди достатньо знати, що вузол А надіслав IP-пакет вузлу В: коли вузол В прийме IP-пакет, він повинен знати, якому процесу передати його вміст для подальшої обробки.

Існує безліч протоколів транспортного рівня, але розробнику найбільш цікаві TCP (Протокол керування передачею) та UDP (Протокол датаграм користувача). Перший з них забезпечує надійну передачу даних та гарантує отримання пакетів в вихідному порядку, інший – ні. Також TCP має власну реалізацію для керування потоком даних, тим самим зменшуючи затори в мережі та завдяки чому пакети втрачаються з меншою вірогідністю.

І коли виникає питання коли який протокол використовувати, майже очевидним є рішення використовувати TCP, адже він гарантує доставку. Але така логіка цілком неправильна. Логіка протоколу TCP, яка відповідає за вищенаведені функції, несе за собою ряд недоліків.

Перший – те, що протокол забов`язаний гарантувати надійність та правильний порядок доставки усіх відправлених даних. Через це, якщо якісь дані з низьким пріоритетом будуть втрачені в мережі, а за ними будуть йти важливі для гравця дані (наприклад перші дані про те, що хтось з гравців використав візуальний ефект, а другі дані – що в гравця зараз стріляють), то протокол TCP забезпечить правильний порядок доставки, тобто дані про постріл не будуть оброблені доти, доки після повторної передачі даних з візуальним ефектом, вони врешті решт не будуть оброблені. А вони можуть втратитись вдруге, втретє. За цей час інший гравець вже встигне тричі вистрілити, коли ви ще не отримаєте дані про перший постріл, і будете як ні в чому не бувало стояти там де стояли.

Другий – те, що дані, які є неактуальними, можуть бути передані повторно. Наприклад, якщо дані про позицію гравця будуть втрачені в мережі, то така

інформація буде відправлена повторно, через що на клієнті може відображатись неправильна інформація.

Щоб уникнути вищезгаданого, варто завжди використовувати протокол UDP. Всі функції, які він не виконує сам, можна реалізувати самому, забезпечуючи високу гнучкість у використанні цього протоколу.

Розділ 2: Методи компенсації затримок

2.1 Втрата пакетів

Причин втрати пакетів може бути багато, найбільш вірогідні – ненадійність канального рівня, мережевого рівня та самого фізичного носія. Проблему з мережевим рівнем можна вирішити шляхом впровадження правильної архітектури та зменшенням кількості пакетів шляхом збільшення їх розмірів.

Але все одно, варто піклуватись про перевірку, чи був пакет доставлений. Використовуючи UDP, варто реалізувати такий механізм вручну. Для цього можна створити модуль, який би сповіщав про доставку. Цей модуль не буде самостійно здійснювати повторну доставку, чи здійснювати її взагалі буде вирішувати інший модуль, або навіть інші модулі, адже для деяких даних повторна доставка бажана, для інших же нею можна знехтувати. Щоб реалізувати таку логіку, модуль має унікально ідентифікувати кожен пакет, на стороні отримувача він повинен надсилати підтвердження для кожного пакету, який оброблюється, а на стороні відправника має оброблювати ці підтвердження та давати інформацію про надходження іншим модулям. Завдяки такій механіці можна реалізувати отримання пакетів в потрібній послідовності, адже пакет з id, який вже неактуальний (вже були отримані пакети з id більшим за id цього пакету) можна просто проігнорувати.

Надання послідовних унікальних id досягається за допомогою збільшення змінної номеру наступного пакету при його підготовці:

```
Packet* DeliveryNotificationMng::WriteWithNewSequenceNumber
(OutputMemoryBitStream& stream)
{
    SequenceNumber id = ++_sequenceNumber;
    stream.Write( id );

    ++_packetCount;
```

```
        if( _shouldProcAcks )
        {
                _packets.emplace_back(sequenceNumber);

                return &_packets.back();
        }
        else
        {
                return nullptr;
        }
}
```

Підтвердження на стороні отримувача та імітація скидування неактуальних пакетів:

```
void DeliveryNotificationManager :: addPendingAck (PacketSequenceNumber id )
{
        if( _pendingAcks.size() == 0 || !_pendingAcks.back().Extend( _sequenceNumber
) )
        {
                _pendingAcks.emplace_back( id );
        }
}
bool DeliveryNotificationMng::ProcessSequenceNumber( InputMemoryBitStream&
stream )
{
        SequenceNumber  sequenceNumber;
        stream.read(sequenceNumber);

        if( sequenceNumber == _sequenceNumber )
        {
```

```
            _sequenceNumber = sequenceNumber + 1;
            if( _shouldSendAcks )
            {
                addPendingAck( sequenceNumber );
            }
            return true;
        }
        else if( sequenceNumber < _sequenceNumber )
        {
            return false;
        }
        else if( sequenceNumber > mNextExpectedSequenceNumber )
        {
            _sequenceNumber = sequenceNumber + 1;
            if( _shouldSendAcks )
            {
                addPendingAck( sequenceNumber );
            }
            return true;
        }
        return false;
}
```

І коли вузол-отримувач буде готовий відправити пакет відповіді, він запише усі підтвердження в вихідний пакет:

```
void DeliveryNotificationManager::WritePendingAcks (OutputMemoryBitStream&
pck)
{
 bool hasAcks = (_pendingAcks.size() > 1);
 pck.Write(hasAcks);
 if(hasAcks)
```

```
 {
 _pendingAcks.front().Write(pck);
 _pendingAcks.pop_front();
 }
}
```

Клієнт-відправник в свою чергу має оброблювати отримані підтвердження:

```
void DeliveryNotificationMng::ProcessAcks( InputMemoryBitStream& stream)
{
        bool hasAcks;
        stream.Read( hasAcks );
        if( hasAcks )
        {
                AckRange ackRange;
                ackRange.Read( stream);
                SequenceNumber nextAckd= ackRange.GetStart();
                uint32_t onePastAckd = nextAckd + ackRange.GetCount();
                while( nextAckd < onePastAckd && !mInFlightPackets.empty() )
                {
                        const auto& nextPacket = mInFlightPackets.front();
                        SequenceNumber nextSequenceNumber =
nextPacket.GetSequenceNumber();
                        if( nextSequenceNumber < nextAckd)
                        {
                                Packet* copyOfInFlightPacket = nextPacket;
                                _packets.pop_front();
                                handlePacketDeliveryFailure( copyOfInFlightPacket );
                        }
                        else if( nextSequenceNumber == nextAckd)
                        {
                                HandlePacketDeliverySuccess( nextPacket );
```

```
                    _packets.pop_front();

                    ++ nextAckd;
                }
                else if( nextSequenceNumber > nextAckd)
                {
                    ++ nextAckd;
                }
            }
        }
}
```

2.2 Реплікація об`єктів

Модуль, що лежить вище за DeliveryNotificationMng має відповідати за реплікацію об`єктів. В моєму випадку це ReplicationManager, який має такий інтерфейс:

```
class ReplicationManagerForClient
{
public:
    void read( InputMemoryBitStream& stream);
private:
    void readAndCreateAction( InputMemoryBitStream& stream, int id);
    void readAndUpdateAction( InputMemoryBitStream& stream, int id);
    void readAndDestroyAction( InputMemoryBitStream& stream, int id);
};
```

Вище лежачі модулі створюють потоки виведення, готують пакети та викликають метод read, який в свою чергу в залежності від поля action буде викликати readAndCreateAction(), readAndUpdateAction() або readAndDestroyAction(), щоб створити, змінити або видалити об'єкт з яким працюють.

Розділ 3: Основні методи компенсації затримок

3.1 Інтерполяція на стороні клієнта

Пригальмовування, викликане низькою частотою оновлення стану сервером, може змусити гравців думати, що гра відбувається повільніше, ніж насправді. Один із способів вирішити ситуацію - використовувати інтерполяцію на стороні клієнта. При використанні інтерполяції ігрові персонажі не переміщаються так званими ривками з отриманням нових даних з сервера. Замість цього кожний раз, коли клієнт отримує новий стан об'єкта, він плавно інтерполює цей стан протягом деякого інтервалу часу. Це називається локальним фільтром сприйняття (local perception filter).

Нехай IPeriod (Interpolation Period) - це період інтерполяції в мс, тобто інтервал часу, протягом якого клієнт буде інтерполювати старий стан в новий, а PPeriod (Packet Period) - період пакета в мс, тобто інтервал часу, протягом якого сервер чекає на можливість послати новий пакет. Клієнт завершує інтерполяцію через IPeriod мс після прибуття пакета. Відповідно, якщо IPeriod менше PPeriod, клієнт закінчить інтерполяцію раніше, ніж прибуде новий пакет, і гравець все ще буде спостерігати зміну картинки ривками. Щоб забезпечити плавну зміну стану гри в кожному кадрі, величина IPeriod повинна бути не менша за PPeriod. У цьому випадку, коли клієнт завершить інтерполяцію до заданого стану, він вже отримає наступний стан і зможе повторити цей процес спочатку.

Величина IPeriod повинна бути якомога меншою, адже це буде впливати на час, на який відстає клієнт від серверу. Через такі обмеження необхідно, щоб IPeriod строго дорівнював PPeriod. З цією ціллю клієнт має знати, з якою частотою сервер надсилає пакети.

Інтерполяція на стороні клієнта вважається консервативним алгоритмом: незважаючи на те що іноді цей прийом може допомогти представити стан, який сервер ніколи не передавав явно, він відображає всього лише проміжні стани між

двома точками, які сервер дійсно моделював. Клієнт згладжує перехід з одного стану в інший, але він ніколи не намагається вгадати, що робить сервер, і тому ніколи не виявиться в невірному стані.

3.2 Прогнозування на стороні клієнта

Інтерполяція на стороні клієнта може згладити відчуття плинності гри, але вона ніяк не наблизить гравця до того, що в дійсності відбувається на сервері. Навіть з дуже коротким періодом інтерполяції гравець все одно буде спостерігати стан, який відстає від стану на сервері не менше ніж на 1/2 RTT. Щоб показати стан, максимально наближений до актуального, гра повинна переключитися з інтерполяції на екстраполяцію. За допомогою екстраполяції клієнт може на основі прийнятого старого стану передбачити більш свіжий стан і відобразити його на екрані. Прийоми, які здійснюють подібну екстраполяцію, часто називають *прогнозуванням на стороні клієнта*.

Щоб екстраполювати поточний стан, клієнт повинен мати можливість виконати ту ж модель, яка виконується на сервері. Отримавши новий стан, клієнт знає, що він відстає від істинного приблизно на 1/2 RTT мс. Щоб наблизити стан до істинного, клієнт виконує моделювання на додатковий інтервал в 1/2 RTT мс. В результаті на екрані буде відображено досить точну екстраполяцію стану ігрової моделі на сервері. Для підтримки актуальності цього стану клієнт продовжує виконувати моделювання в кожному кадрі і відображати результати на екрані. У якийсь момент клієнт прийме від сервера пакет з новим станом і змоделює його за 1/2 RTT мс, в результаті чого воно ідеально співпаде зі статком, який клієнт вже розрахував, спираючись на попередньо отриманий стан.

Щоб виконати екстраполяцію на 1/2 RTT, потрібно спочатку визначити величину RTT. Так як час на сервері і клієнті можуть бути розсінхронізовані, найпростішим буде прийом, коли сервер додасть в пакет позначку часу, а клієнт

перевіре її. Але такий підхід не спрацює. Замість цього клієнт повинен визначити повний період RTT і розділити його навпіл.

```
void NetworkMngServer::HandleInputPacket (ClientProxy clientProxy, InputMemoryBitStream& stream)
{
    uint32_t mCount = 0;
    Move move;
    stream.read(mCount, 2);
    for( ; mCount > 0; mCount--)
    {
        if( move.Read( stream) )
        {
            if( clientProxy-> getUnprocessedMovementList(). AddMoveIfNew(move))
            {
                clientProxy->setIsLastMovementTimestampDirty (true);
            }
        }
    }
}
```

Також, розрахунок траєкторії руху не може приховати затримку від локального гравця. Уявіть, що гравець А на клієнті А починає рух вперед. Алгоритм розрахунку траєкторії використовує стан, надісланий сервером, тому з

моменту, коли він натисне на клавішу «up», і до моменту, коли інформація про ввід потрапить на сервер, пройде 1/2 RTT, і тільки після цього сервер скоректує швидкість його персонажа. Потім знадобиться ще 1/2 RTT, щоб змінилося значення швидкості повернулося до клієнта А, і тільки після цього гра зможе задіяти алгоритм розрахунку траєкторії. В результаті виникає затримка довжиною в період RTT між моментом, коли гравець натисне кнопку, і моментом, коли він побачить результат. Існує найкраща альтернатива. Гравець А вводить всі свої команди безпосередньо на клієнті А, тому гра на клієнті А може безпосередньо використовувати введення для моделювання. Як тільки гравець А натисне клавішу «up», клієнт може почати моделювати рух свого персонажа. Коли пакет з інформацією досягне сервера, він також почне моделювання, змінюючи стан персонажа А. В ідеалі переміщення повинні виглядати для локального гравця так, як якщо б він грав в немережеву гру.

Коли клієнт отримає пакет зі станом, він викличе метод для обробки кроків з відмітками часу, які повернув сервер. Виявивши такий крок, він відніме позначку часу з значення поточного часу, щоб визначити величину RTT, яка стане в нагоді для розрахунку траєкторії руху. Потім викличе RemovedProcessedMoves, щоб видалити кроки з відмітками часу, меншими даної позначки або які збігаються з нею. Це означає, що по завершенні методу для обробки кроків локальний список кроків на клієнті буде містити тільки ті з них, які ще не досягли сервера і тому повинні бути застосовані до будь-якого стану, отриманого від сервера.

Перегровку кроків можно зробити так:

```
bool isLocalPlayer = (GetPlayerId()==NetworkMngClient::sInstance->GetPlayerId());

 if(isLocalPlayer)

 {

        clientPredictionAfterReplication(readState);
```

```
}
else
{
        clientPredictionAfterReplicationForRemotePlayer(readState);
}
if(!IsCreatePacket(readState))
{
        InterpolateClientSidePrediction(oldRotation, oldLocation, oldVelocity,
!isLocalPlayer);
}
```

clientPredictionAfterReplication спочатку перевіряє, чи є нові координати. Якщо ні, значить, додаткове моделювання не потрібно. В іншому випадку метод виконує обхід всіх кроків в списку і застосовує їх до локального персонажу. Тим самим моделюються всі дії гравця, поки не враховані сервером. Якщо на сервері не сталося нічого несподіваного, ця функція повинна залишити гравця в тому ж стані, в якому він був раніше, до виклику методу read.

Якщо отримана інформація про remote гравця, clientPredictionAfterReplicationForRemotePlayer змоделює нове місце розташування, використовуючи останніq відомbq стан. Для цього викликається simulateMovement з відповідним інтервалом часу, без обробки введення викликом processInput. І знову, якщо на сервері не відбулося нічого несподіваного, віддалений персонаж повинен залишитися в тому ж стані, в якому перебував до виклику read. Однак на відміну від локального персонажа ймовірність несподіваної зміни стану віддаленого персонажа набагато вище: віддалені гравці часто віддають різні команди, змінюючи напрямок руху і подібні команди.

Після прогнозування на стороні клієнта метод read() під кінець викликає InterpolateClientSidePrediction (), щоб обробити будь-які зміни стану. Отримавши старbq стан, метод інтерполяції зможе визначити, наскільки великі відхилення, якщо вони взагалі є, і згладить перехід зі старого стану в новий.

Також можна використати приховування затримки із застосуванням хитрощів і оптимістичного алгоритму. Майже всі дії в відеоіграх мають деякі візуальні ознаки, що повідомляють, що щось сталося. Спалах повідомляє про постріл, а чарівники махають руками і вимовляють слова, перш ніж виплеснутисвою магію. Зазвичай тривалість відтворення цих графічних ефектів ніяк не менше часу передачі пакету на сервер і отримання відповіді. Це означає, що на клієнті може дати локальному гравцеві миттєвий зворотний зв'язок у відповідь на будь-яке введення та відтворити відповідний анімаційний ефект, чекаючи, поки моделювання буде виконано на сервері.

Висновки

Розглянута тема є дуже актуальною на даний момент, наведені алгоритми та прийоми можна застосовувати при проектуванні та створенні безпосередньо онлайн шутерів, стратегій тощо.

В результат виконання курсової роботи було створено проект, в якому було наглядно продемонстровано вплив затримок на ігровий інтерфейс та сприйняття гри клієнтами. Було створено власну надбудову над протоколом UDP, завдяки чому вдалось реалізувати наведені алгоритми. Врешті решт вдалося мінімізувати загальну затримку на передачу пакетів в мережі (вона складає від 24 до 120 мс для різних клієнтів), завдяки чому гра на клієнті виглядає так, як нібито гравець грає в Single Player моді. В своїй роботі я не надав великої уваги безпеці передачі даних, але попри це маніпулювання даними, отриманими гравцем з серверу задля нечесної гри майже неможливе.

Список використаної літератури

Електронні ресурси	1. Усі наведені практики та алгоритми: Dzhoshua Gleizer, Sandzhai_Madhav “Ігри розраховані на багато користувачів: Розробка онлайн застосунків”. 2. Детально про прогнозування на стороні клієнта: https://moluch.ru/archive/216/52143/ 3. Ще про прогнозування на стороні клієнта: https://genapilot.ru/netcode-explained 4. Про інтерполяцію на стороні клієнта: https://habr.com/ru/post/302834/ 5. Про реалізацію алгоритмів компенсації затримок в UnrealEngine4: https://habr.com/ru/company/mailru/blog/352634/