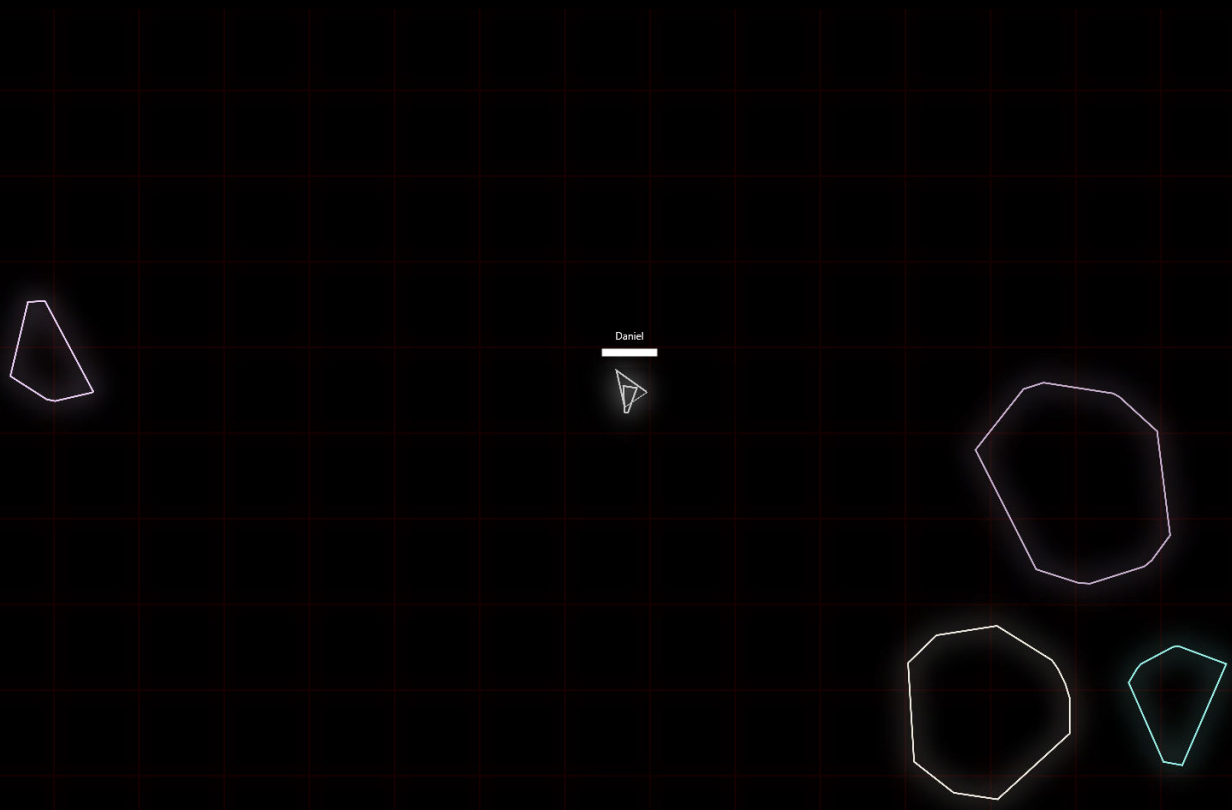
NeonTDS

Multiplayer Top Down Shooter



Készítették:
Fehérvári Attila
Kovács Milán
Zsolnai Dániel

2019. 12. 14.

Tartalom

A projekt struktúrája

Mivel egy hálózati játék fejlesztésénél elengedhetetlen, hogy legyenek megosztott és nem megosztott kód részletek, valamivel bonyolultabb projekt struktúrára van szükség. A következő szekció ezt hivatott bemutatni.

GameLogic

Ez a projekt tartalmazza az osztott játék logikai kódot. Mind a `NeonTDS` (ami igazából `GameClient` is lehetne) mind a `GameServer` referenciaként tartalmazza.

A projekt felelősségei többek között:

- Entitás menedzsment
    - Létrehozás
    - Frissítés
    - Törlés
- Ütközés detektálás
- Power up logika
- Player input lekezelése

Platform kompatibilitási megfontolások

A projekt egy .NET standard 2.0-s projekt mivel a `GameServer` linuxon is futtatható, ezért .NET core kompatibilisnek kellett lennie a logikának.

Pontosan ebből kifolyólag nem is találunk a grafikai motort támogató könyvtárra semmiféle dependenciát (Win2D). Ezért kellett kissé bonyolultan elszeparálni az entitást és a kirajzolt entitást.

NeonTDS

Ennek a projektnek a felelőssége a játék motor (Game Engine) megvalósítása. Itt valósul meg az imént említett `Drawable` – `Entity` szeparáció is.

Tartalmazza ezen kívül még a kliens specifikus hálózati kódot.

A project egy UWP alkalmazás, hogy tudjuk használni a Win2D könyvtárat a grafika megvalósítására. Emiatt .NET framework-ot és nem core-t vagy standardet használ, mint a többi projekt.

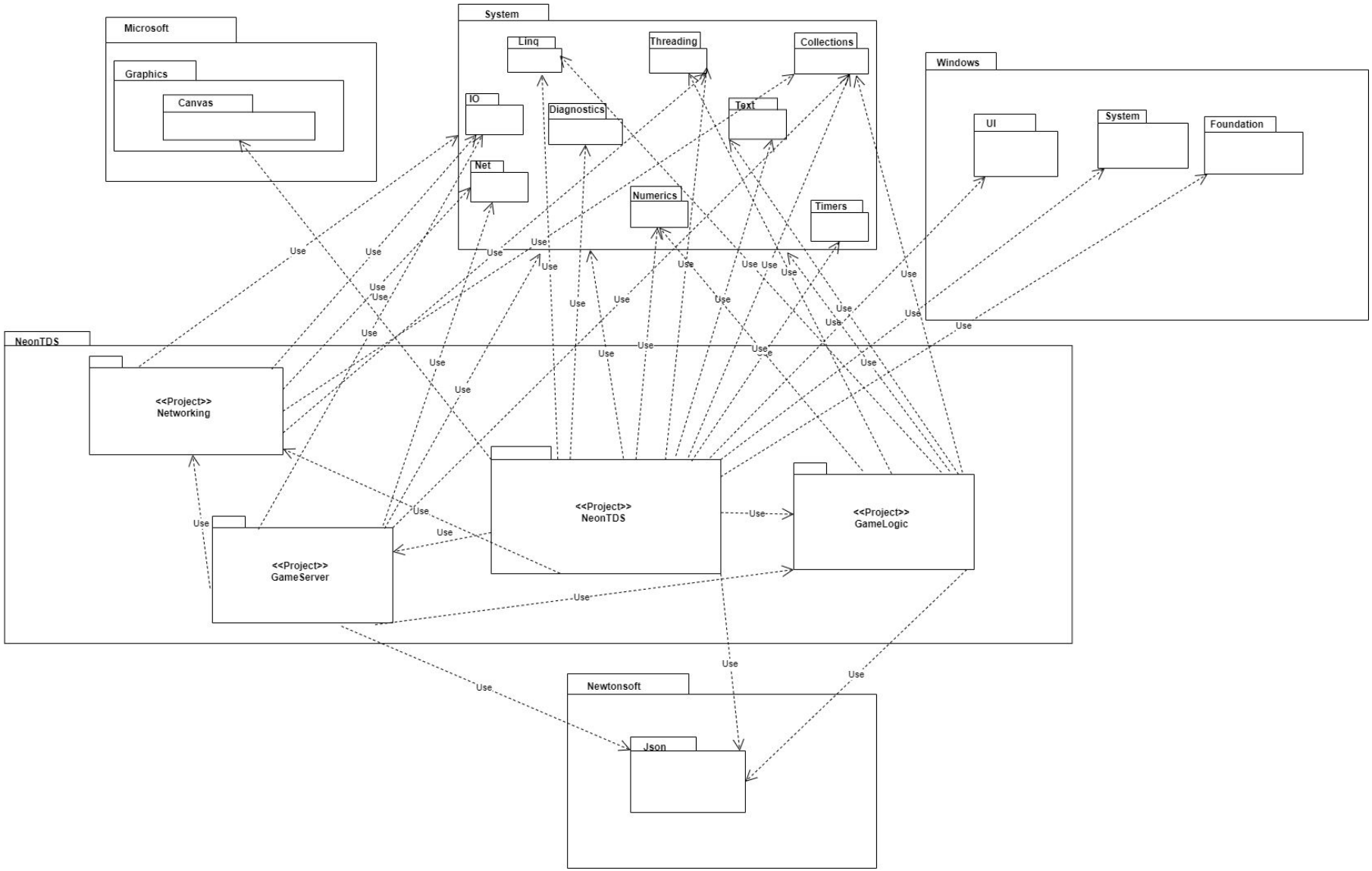
GameServer

A GameServer egy .NET core console application, hogy tudjon futni linux alatt (mivel egy Debian VPS ált a rendelkezésünkre a teszteléshez).

A `GameServer` használja a `GameLogic` és Networking Shared Library-ket, de azon kívül igen letisztult. Csak a `GameClient`-el egészíti ki a Networking-et. Illetve megvalósítja a szerver oldali Game loop-ot, amiben összeszedi és csomagolja az üzeneteket, amiket a kliensnek küld.

Networking

A Networking szintén egy .NET standard Shared Library, amit a kliens és a szerveroldali kód is használ. Ez valósítja meg a hálózati architektúrát, amiről bővebben a *Szerver-kliens kommunikáció megvalósítása* című fejezetben lesz szó. Nagy vonalakban egy egyszerű és egy multiplexelt UDP kliens valamit az üzenet típusok és a szerializációjuk tartozik ide.


Microsoft
Graphics
Canvas
System
Linq
Threading
Collections
IO
Diagnostics
Text
Net
Numerics
Timers
Windows
UI
System
Foundation
NeonTDS
<<Project>>
Networking
<<Project>>
NeonTDS
<<Project>>
GameLogic
<<Project>>
GameServer
Use
Newtonsoft
Json

Éles teszteléshez használt környezet

Mivel ez a játék nehezen tesztelhető lokálisan (két UWP appot nem lehet egyszerre debuggolni vagy csak nagyon körülményesen), ezért érdemes volt felállítani egy remote szervert.

A remote szerver egy Debian VPS, de legjobb tudomásom szerint bármilyen linux disztribúción működnie kéne (ahol a .Net core supportált).

Ahhoz, hogy a játék is tudja, hol a szerver, egy `config.json`-t használ, amely így néz ki:

```
{
	"ServerIP": "134.209.232.177", // Ezek a defaultok is[1]
	"ServerPort": 32132
}
```

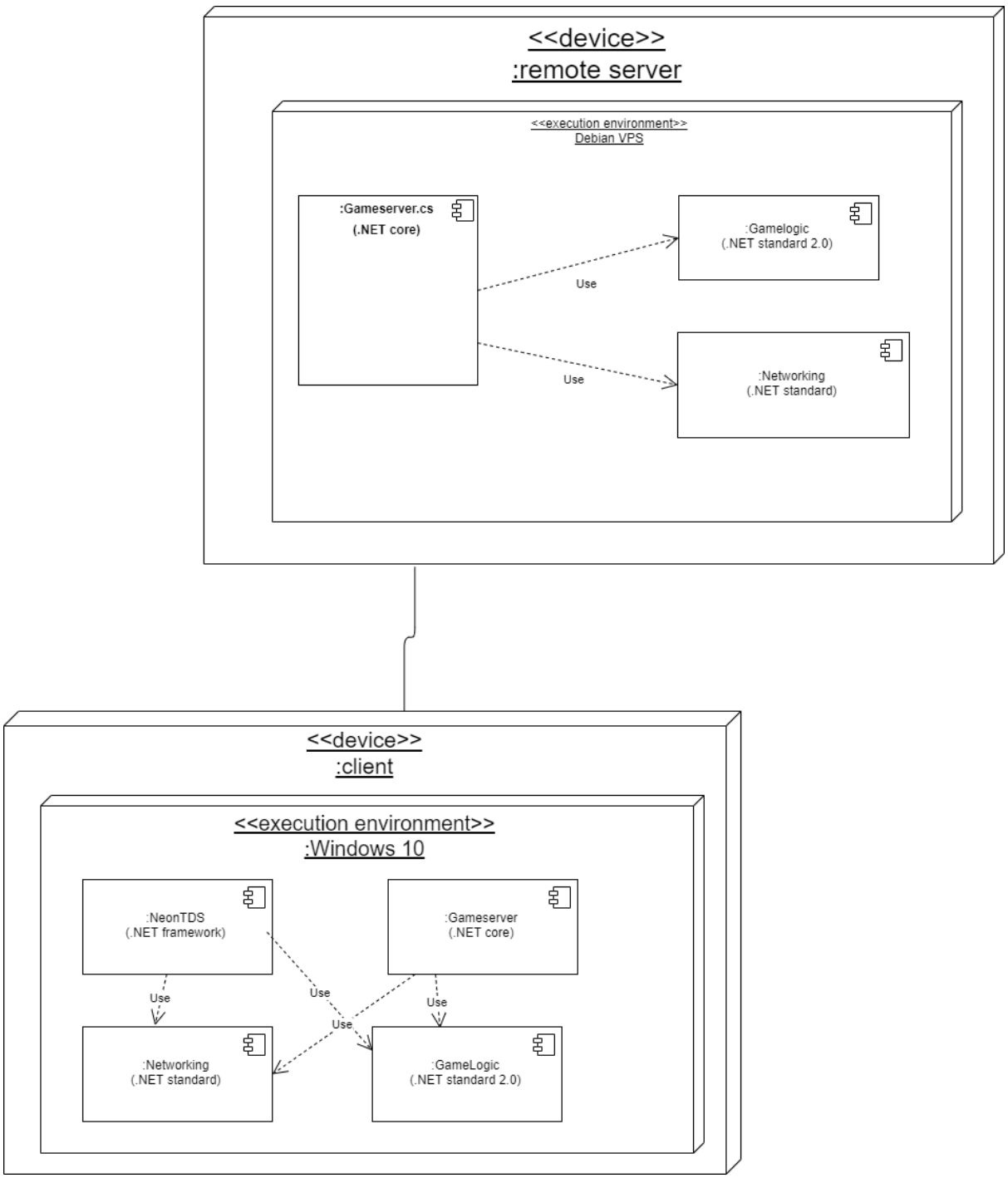
A szerver szintén tartalmaz egy szinte teljesen ugyanilyen JSON fájlt, csak az IP nélkül.

Build szerver

A kényelmes fejlesztés jegyében a VPS-en fut egy Build szerver, amely kiválasztott branchek-hez futó szerver instance-okat rendel. Emelett github-os webhook-okkal képes detektálni a pusholást és azonnal lepullolja és újrabuildeli az adott instancet. Egyszerre continous integration és process daemon.

A build server Node.js (express) technológiával készült, mert itt nem volt megkötve a kezünk és a C# túlbonyolult lenne ehhez. Magát a szevert egy PM2 nevű node-os process menedzser futtatja a VPS-en belül.

[1] Tudom, hogy JSON-ban nincsen comment.


<<device>>
:remote server
<<execution environment>>
Debian VPS
:Gameserver.cs
(.NET core)
:Gamelogic
(.NET standard 2.0)
Use
Use
:Networking
(.NET standard)
<<device>>
:client
<<execution environment>>
:Windows 10
:NeonTDS
(.NET framework)
:Gameserver
(.NET core)
Use
Use
Use
Use
:Networking
(.NET standard)
:GameLogic
(.NET standard 2.0)

A játék logika

Játékszabályok

A játék alapvetően egyszerű. A játékosok egy limitált (de egész nagy) pályán mozoghatnak. A játékosok, ha egymásnak mennek mind a ketten meghalnak.

A játékosok lelőhetik egymást ezzel sebezve a másikat. Minden játékosnak van életereje, ami kiegészülhet egy pajzserővel is ha pajzs power up-ot vesznek fel.

A pályán folyamatosan spawnolnak power up-ok, amiket fel véve valami extrára lehet szert tenni.

Power up-ok:

- Rapid fire (sárga színű) – gyorsabb lövés pár másodpercig
- Shield (sötétkék színű) – pajzserő az életerőhöz (gyakorlatilag második független életerő, ami prioritást élvez)
- Sniper (világoskék színű) – azonnal (shielden keresztül is) ölő és mindenen áthaladó lövedék, amiből csak egy darabot kap a felvevő játékos

Ha valamely játkos (igazából bármely entitás) a játék széléhez ér megöli a lézer azonnal.

Kliens és szerveroldali logika elszeparálása

Annak jegyében, hogy minél egyszerűbben megosztható legyen a `GameLogic` kódja és a kliens, illetve szerver specifikus részeket egy helyen lehessen implementálni az `EntityManager` tartalmaz egy `IsServerSide` flag-et. Ez alapvetően nem a legjobb megoldás, hiszen így mind a két oldal tartalmaz halott kódot, amit sosem használ. Jobban belegondolva rájöhetünk, hogy egy definiált compile time változó lett volna a jó ötlet, de ez csak későn jutott eszünkbe.

A játékszabályok implementálása

A legtöbb játékszabályt a `Player` entitás implementálja, innen tudjuk meg mi történik, ha a `Player` más entitásokkal ütközik. Ez az osztály implementálja a respawn logikát is.

A kliens nem sebezheti és respawn-olhatja a `Player`-t így ezek a logikák `IsServerSide=true` esetén érvényesülnek csak.

Entitás menedzsment

Az entitások menedzseléséért az `EntityManager` felelős, amely a létrehozást, törlést és frissítést végzi.

Az `EntityManager` egy .Net-es `Dictionary` (`HashMap`)-ben tárolja az entitásokat az azonosítójuk alapján.

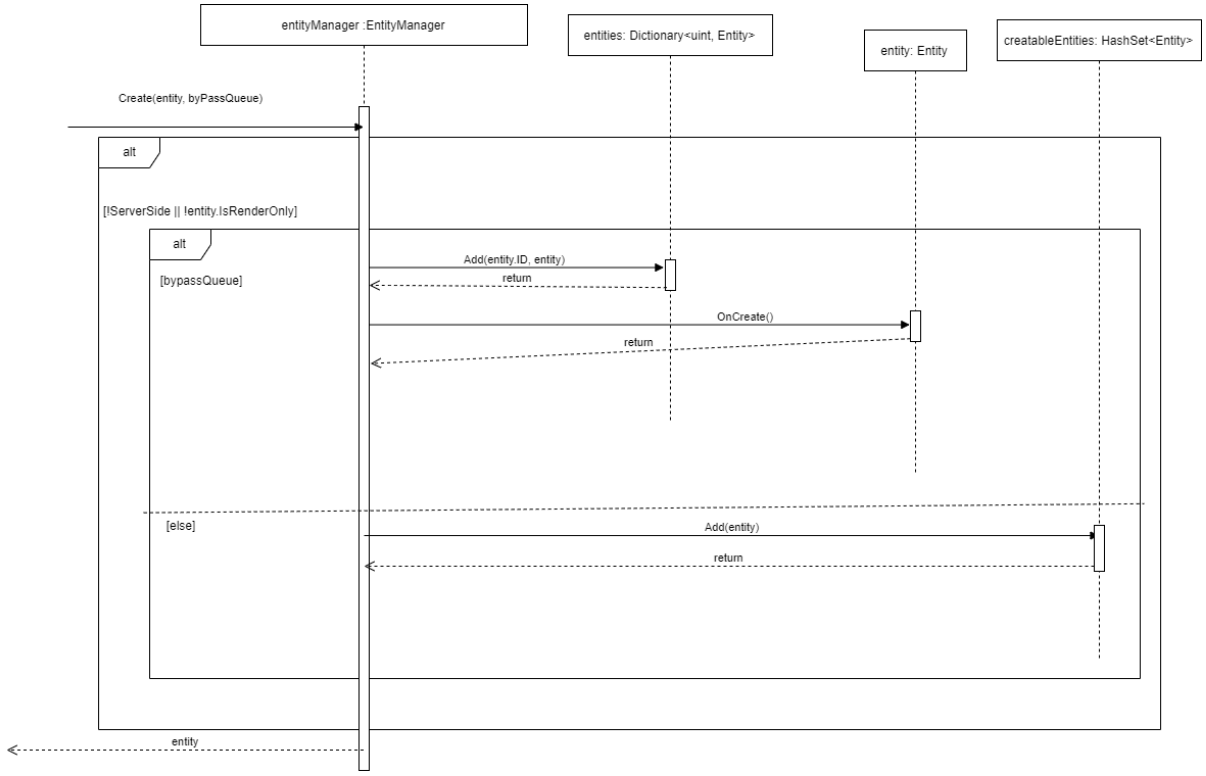
Entitás azonosító

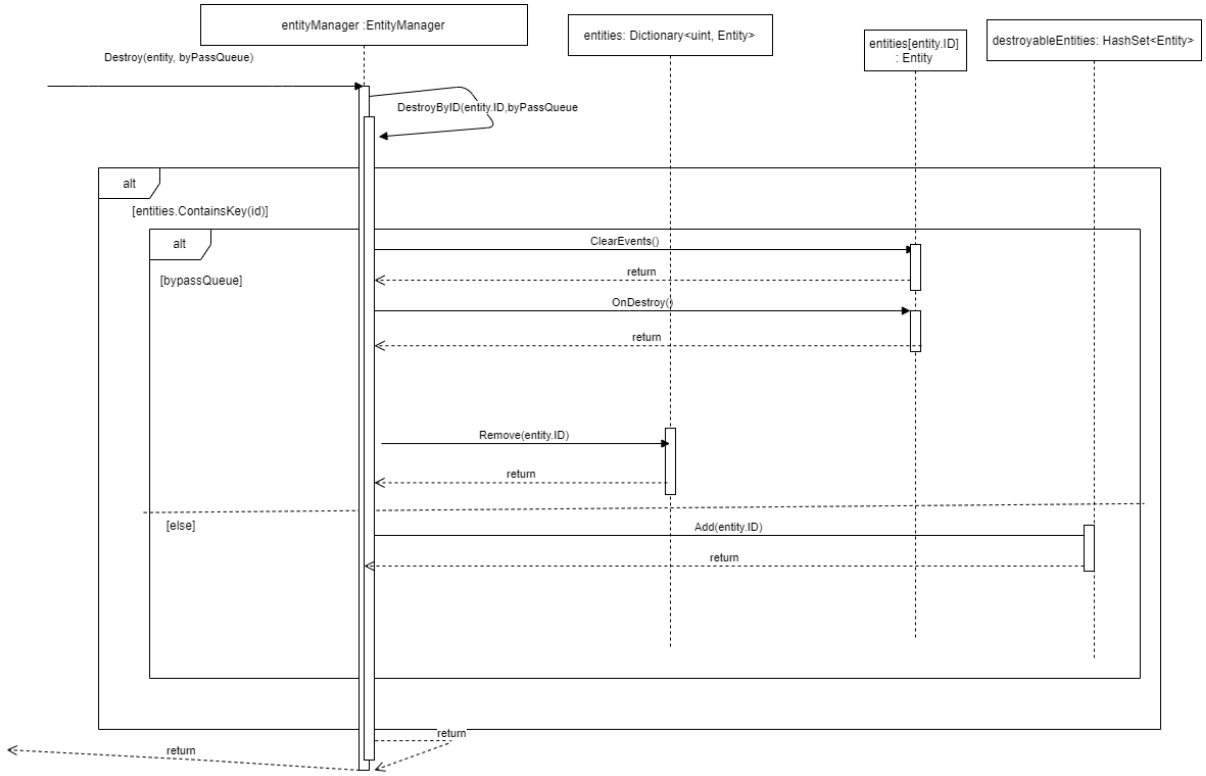
Az entitások azonosítója egy egyszerű 32 bites unsigned integer, amit a hálózati architektúra is használ az azonosításhoz. Ehhez pluszba egy egyedi rendszert alkalmaztunk a csak kliens oldali entitások ID-ja a legfelső bitet 1-re állítja míg minden nem csak kliensoldali entitás legfelső bitje 0. Erre végső soron nem volt feltétlen szükség.

Entitás létrehozás és törlés

Mivel egy `foreach` loop-on belül nem lehet törölni, erre bevett szokás, hogy egy ideiglenes listába pakoljuk a létrehozandó és törlendő entitásokat, amit a végén alkalmazunk a tényleges entitás listára.

Ez a két ideiglenes lista egy-egy `HashSet`-ként lett implementálva, mivel a sorrend mindegy és ezzel könnyen kiküszöbölhető ugyanannak az entitásnak a többszöri hozzáadása.


entityManager :EntityManager
entities: Dictionary<uint, Entity>
entity: Entity
creatableEntities: HashSet<Entity>
Create(entity, byPassQueue)
alt
[!ServerSide || !entity.IsRenderOnly]
alt
[bypassQueue]
Add(entity.ID, entity)
return
OnCreate()
return
[else]
Add(entity)
return
entity


entityManager :EntityManager
entities: Dictionary<uint, Entity>
entities[entity.ID] : Entity
destroyableEntities: HashSet<Entity>
Destroy(entity, byPassQueue)
DestroyByID(entity.ID,byPassQueue
alt
[entities.ContainsKey(id)]
alt
[bypassQueue]
ClearEvents()
return
OnDestroy()
return
Remove(entity.ID)
return
[else]
Add(entity.ID)
return
return
return

Update függvény

A már említett entitás lista kezelésen túl, ebben a függvényben frissülnek maguk az entitások (meghívjuk az `Update` függvényüket).

Ez a függvény kezeli le az „entitás gyilkos” pályahatárokat, amelyre két okból van szükség. Egyrészt nem akartunk végtelen pályát, szóval egyfajta gameplay oka is van a dolognak. Másrészt fontos, hogy a lövedékeink soha az életben nem tűnnek el hacsak nem mondjuk nekik, ezért felelős még a pályahatár, hogy ha érintkeznek vele az entitások, azonnal törlődjenek.

Ezen kívül ide került még a Power up spawn-olással kapcsolatos logika is, aminek igazából nincsen jó oka.

Entitás események

Az entitások az `EntityManager`-től kapnak információt arról, amikor létrejönnek vagy törlődnek ezért felelős az `OnCreate` és az `OnDestroy` virtuális függvénypár, amit kedvünkre felüldefiniálhatunk az entitásokban. az entitásokban. Ezen kívül az entitások tartalmaznak egy `CollidesWith` függvényt, amelyet az érintett ütközés logika hív meg (performancia megfontolások miatt nem csinálunk automatikus ütközés detektálást mindig minden entitásra).

EntityManager események

Az `EntityManager` események feladata kettős. Egyrészt a kliens feltud iratkozni az egyes eseményekre ilyen olyan effektek létrehozása céljából. Másrészt a szerver oldal feltud iratkozni, hogy üzeneteket tudjon belőle csinálni. Gyakorlatilag ez a híd a szerver oldali üzenet gyártás és a `GameLogic` kód között.

Az események rendes C# event-ek, néhányukhoz tartozik egy `Invoke<Event>` függvény is amit maguk az entitások hívnak meg.

Használható események:

- `EntityCreated`
- `EntityDestroyed`
- `PlayerRespawned`
- `PlayerActivePowerUpChanged`

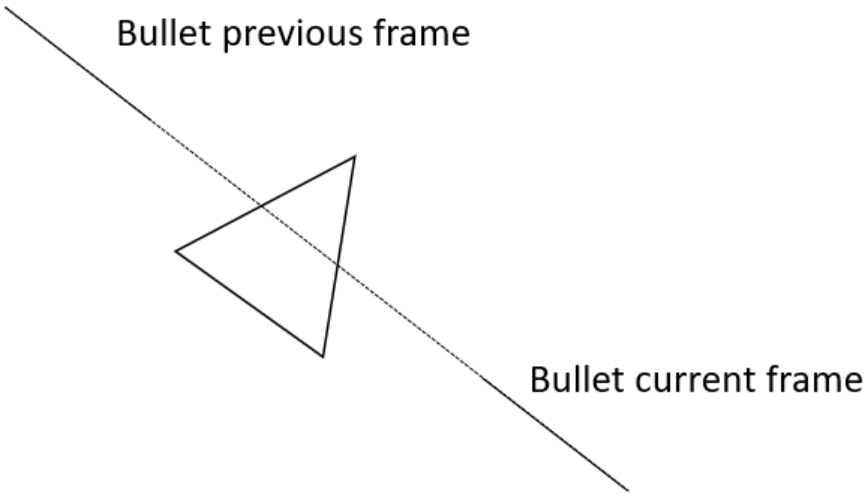
- `PlayerHealthChanged`

Ütközés detektálás

Az algoritmus szakaszok metszésén alapul. `Player` (vagy jövőben más zárt körvonalú entitások) tesztelésére a `TestClosedShapes` algoritmus használható, amely minden szakaszt

minden szakasszal megpróbál ütköztetni. Ez bár nem a legeffektívebb megoldás, a gyakorlatban egész szépen működik.

A `Bullet` – `Player` ütközést máshogyan kell detektálni hiszen a `Bullet` igen nagy sebességű lehet, tehát simán átugorhatja a `Player`-t.



A lövedék elkéne, hogy találja a játékost

Ezért egyfajta ray tracelésre van szükség, erre való a `TestBulletHit` függvény.

Játékos bemenetének kezelése

A játékos (felhasználó) alapvetően a következő féleképpen képes hatni a játékra:

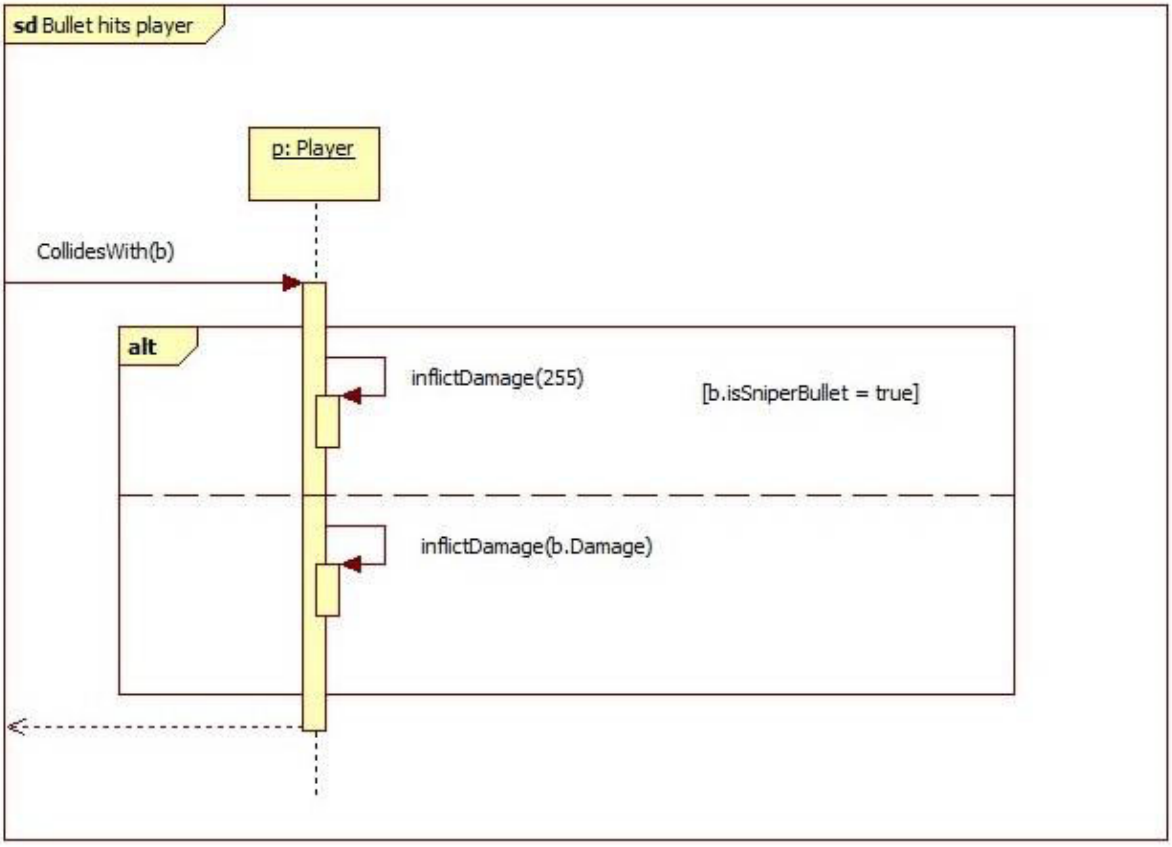
- W – Sebesség növelése (bizonyos maximumig)
- S – Sebesség csökkentése (bizonyos minimumig)
- A – Fordulás balra
- D – Fordulás jobbra
- Egér pozíciója felé néz az ágyú
- Egér balgomb lenyomva tartása – tüzelési állapot igaz (elengedés esetén hamis)

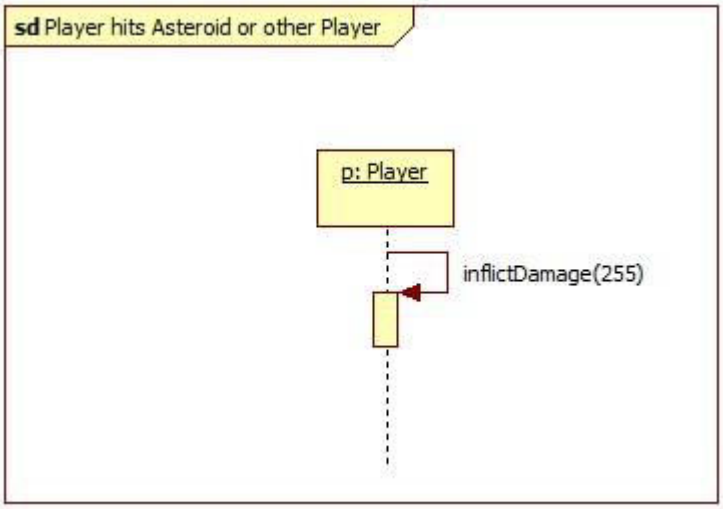
Ezeket az inputokat alapvetően a szerver kezeli le, de lokálisan is megtörténik (a gyakorlatban ez nem igazán látszik).

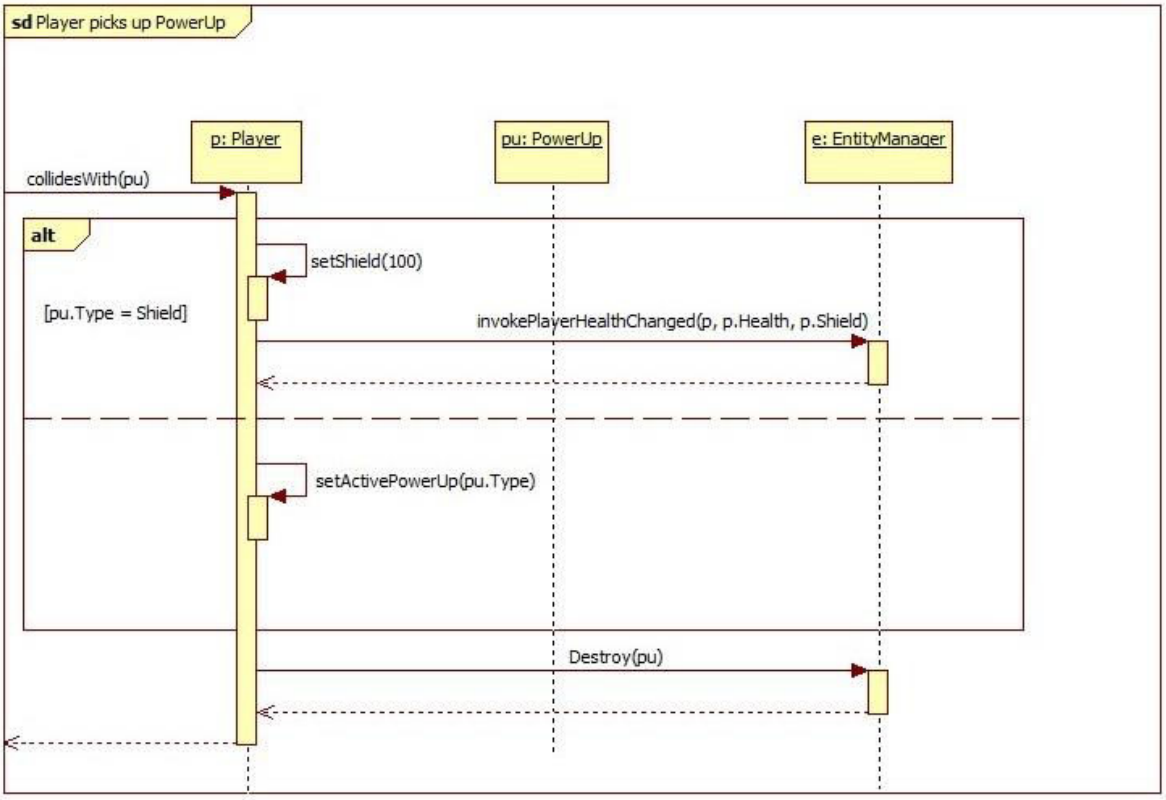
A `Player` entitást indirekten tudjuk szabályozni. A mozgást az adott irányba vett egyenes vonalú egyenletes mozgás valósítja meg, ami minden Entitás esetében az `Update` függvény alapértelmezett logikája (mivel a legtöbb entitás kihasználja).

A lövedékek az ágyú irányába fognak kilőni, ezt a lövedék spawn-olást a szerver utasítására végezheti csak a kliens, vagyis alapvetően ez is `IsServerSide=true` logika.

Játék logikai esetek






sd Player picks up PowerUp
p: Player
pu: PowerUp
e: EntityManager
collidesWith(pu)
alt
setShield(100)
[pu.Type = Shield]
invokePlayerHealthChanged(p, p.Health, p.Shield)
setActivePowerUp(pu.Type)
Destroy(pu)

A játék motor

A játék kliens egy UWP alkalmazás, ezért a játék motorja egy Win2D nevű library-n alapul. Alapvetően a Win2D egy `CanvasAnimatedControl` nevű vezérlőt bocsát a rendelkezésünkre. Ez egy vektor grafikát is támogató Canvas, aminek van egy `CreateResources`, `Update` és egy `Draw` eseménye. A Canvas események a `Game` class-ban vannak megvalósítva, hogy teljesen el tudjuk szeparálni a kódot az UWP alkalmazás Page-étől.

Sprite generálás

A játékunk alapvetően vektor grafikus, de mivel a vektor grafika renderelése alapvetően lassabb és kevesebb objektum számnál is már leesik az FPS, ezért sprite batch-inget alkalmazunk. Erre a Win2D `DrawingContext`-jének van beépített megoldása. Ehhez azonban Bitmap-ek kellenek és nem vektor grafikus leírás. Erre a konverzióra szolgál a `SpriteBuilder` class, amely egy `Shape` reprezentáció és néhány paraméter után `CanvasRenderTarget` segítségével off screen kirendereli a `Shape`-ket. Ezt gyakorlatilag minden Game indításnál megteszi, ami bár nem a legjobb megoldás ennyi Sprite-nál (3 statikus spriteunk van) nem számottevő.

Input kezelés

Alapvetően a beviteli eszközök, mint események (üzenetek) jellenek meg Windowson. Egy játék esetében sokkal inkább szeretjük framenként fel dolgozni a bemenet változásokat. Ezért alapvetően tárolnunk kell az event based input eredményét. Erre szolgál az `InputManager`, amely alapvetően 3 állapotot vezet:

- `actualState` – az input jelenlegi állása az event-ek alapján
- `CurrentState` – az input jelenlegi frame-ben elmentett állapota (csak frame feldolgozás során értelmezhető aktuálisnak)
- `PreviousState` – az input az előző frame-ben elmentett állapota (az előző `CurrentState`)

A current és previous állapotokkal megvalósítható Input variációk:

- Éppen lenyomva (`PressState.Pressed`) – `CurrentState = true`, `PreviousState = false`
- Éppen felengedve (`PressState.Released`) – `CurrentState = false`, `PreviousState = true`

- Most le van nyomva (`PressState.Down`) – `CurrentState = true`, `PreviousState = mindegy`
- Most fel van engedve (`PressState.Up`) – `CurrentState = false`, `PreviousState = mindegy`

Ezek kiolvasására használható függvények az `IsKey(VirtualKey key, PressState pressState)` és az `IsMouse(MouseButton button, PressState pressState)`.

Ezen kívül az `InputManager` tárolja még a mostani és az előző frame-ben elmentett kurzor pozíciót.

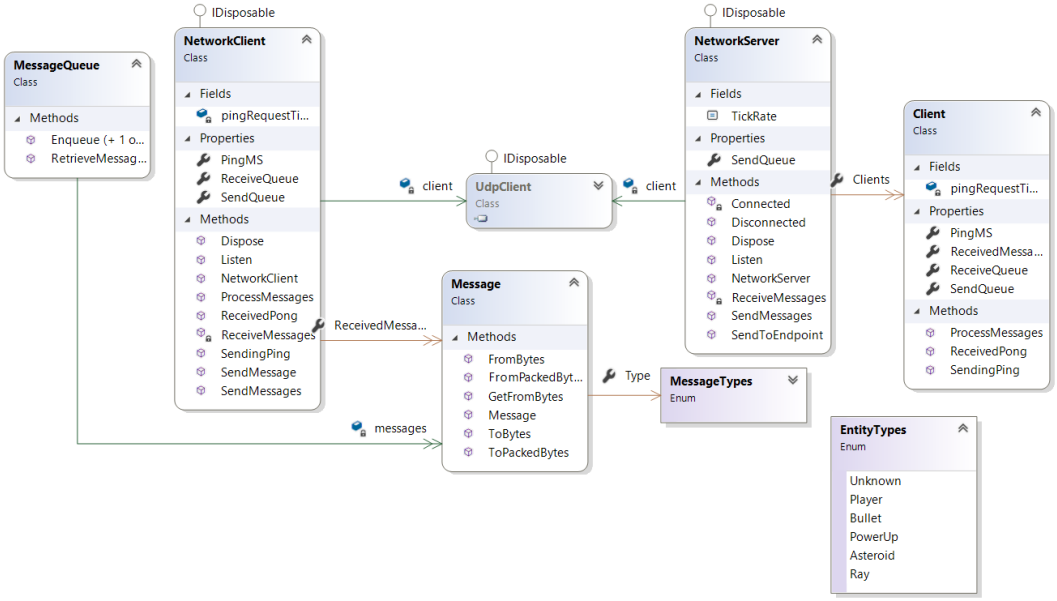
Renderelés

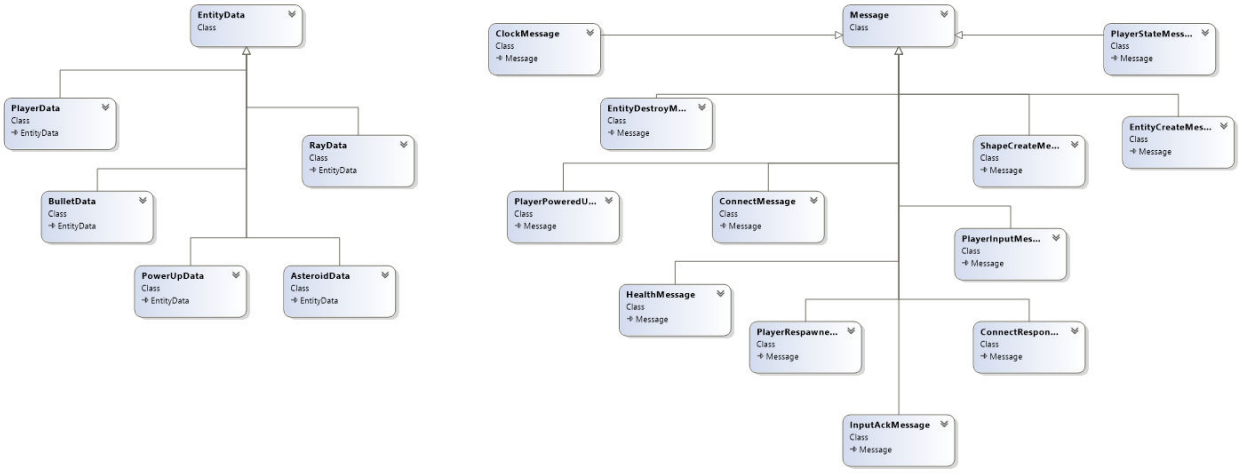
A renderelés játékokban alapvetően egyszerű feladat. Fogjuk az entitást rádobunk egy `Draw(SpriteBatch sb, …)` függvényt és ezt implementáljuk. Itt nem ilyen egyszerű a helyzet, hiszen a `GameLogic` osztálykönyvtár tartalmazza az entitásokat és mivel ezt a potenciálisan nem UWP-t támogató szervernek is tudnia kell használni, ezért nem függhetünk a Win2D-től benne. Kell tehát valami `Entity – Drawable` összerendelés.

Ezért az összerendelésért felelős az `EntityRenderer`, amely fenntartja a listáját a kirajzolandó entitásoknak. A `CreateDrawable` és a `DestroyDrawable` függvényét az `EntityManager` `EntityCreated` és `EntityDestroyed` event-jeire kell rákötni. Ezek után az `EntityRenderer` `Draw` függvényét meghívva az összes entitás kirajzolódik.

Az `EntityRenderer` felel még a `Shape` – `CanvasBitmap` összerendelésért, amit a statikus `Sprite`-ok esetén a `SetupSprites`, a dinamikus sprite-ok esetén (aszteroidák) pedig az `AddDynamicSprite` függvény kezel.

Szerver-kliens kommunikáció megvalósítása





Múltbéli próbálkozások

A csapat alapötlete az volt, hogy ne bonyolítsuk túl a dolgot és használjunk JSON alapú üzeneteket. Egy másik egyszerűsítés a 2 üzenetes rendszer:

- Kliens – Input üzenetet küld, amely tartalmazza az input `CurrentState`-jét
- Szerver – `GameState` üzenetet küld, amely tartalmazza az összes entitás minden állapotát

Ez azonban több szempontból iszonyatos megoldás. A kliens tárol információt az entitásokról és nem mindig kéne mindent leküldeni. Pazarolja tehát a sávszélességet. A másik sávszélesség pazarlás a JSON. Hamar rá kellett jönnünk, hogy nem véletlen nem szokás Real time multiplayer játékoknál szöveges üzeneteket használni. Ez a megoldás sok problémát okozott már kisebb hálózati problémáknál is.

Byte alapú üzenet formátum

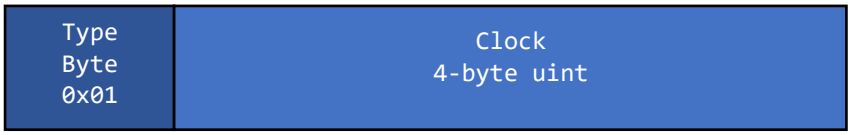
A byte alapú üzenetek alapvetően sokkal jobban teljesítettek. Már érzésre is javult a hálózati teljesítmény. Őszintén szólva nem számítottunk ekkora javulásra, de elég nyilvánvaló, hogy sokkal kevesebb sávszélességet használ a megoldás.

Amellett, hogy byte alapú az adatátvitel, az üzenetek típusait is átdolgoztuk és így egy sokkal optimalizáltabb üzenet rendszert kaptunk.

Üzenet típusok

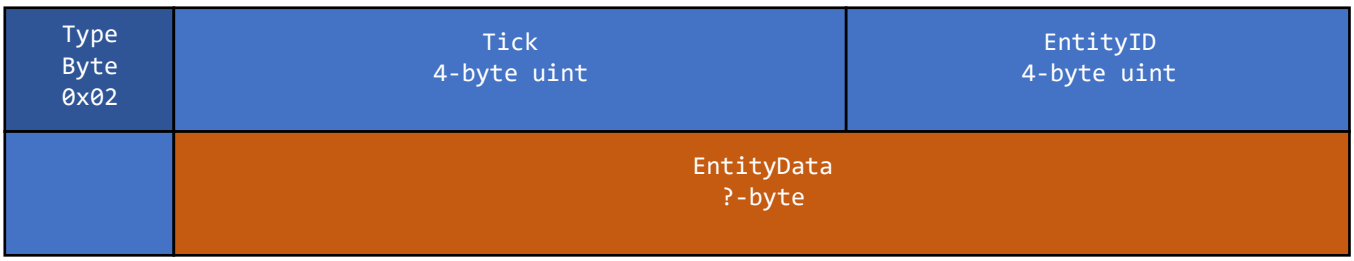
Clock

Ez az üzenet leküldi a kliensnek a szerver aktuális tick-jét. Végül nem használtuk fel semmire. Időzítésre lehetne.



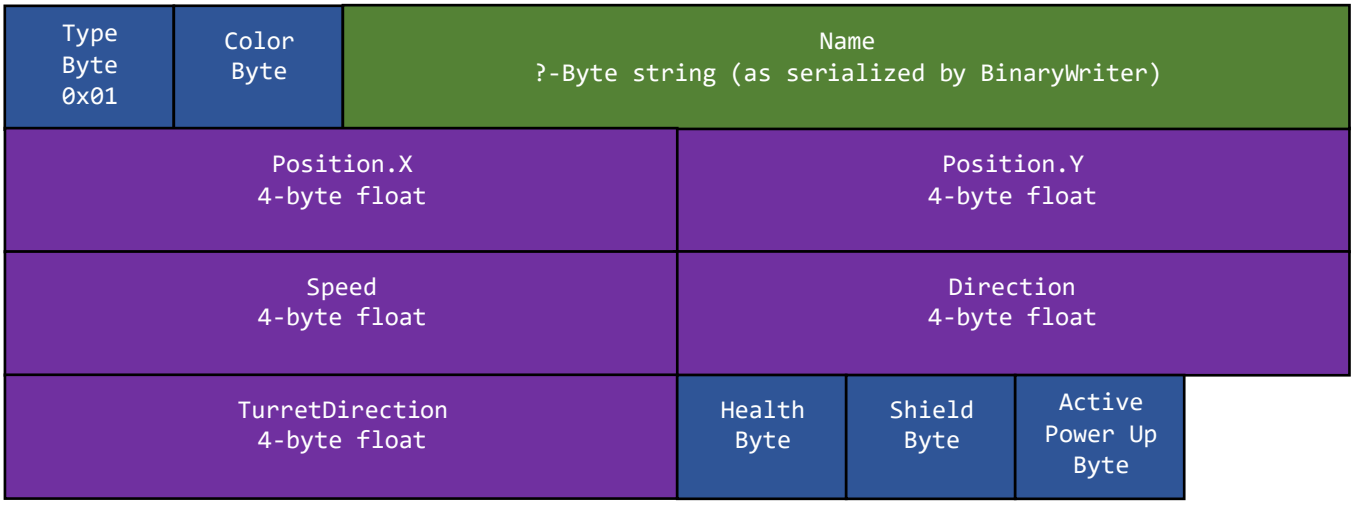
EntityCreate

A szerver oldalon entitás jött létre, vagy a kliens éppen csatlakozott és még nem tud a már

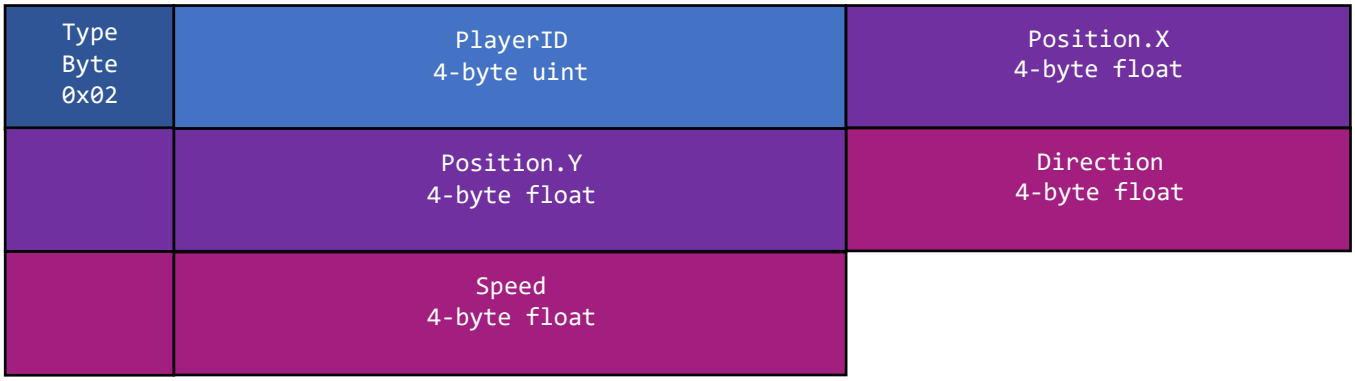


létező entitásokról.

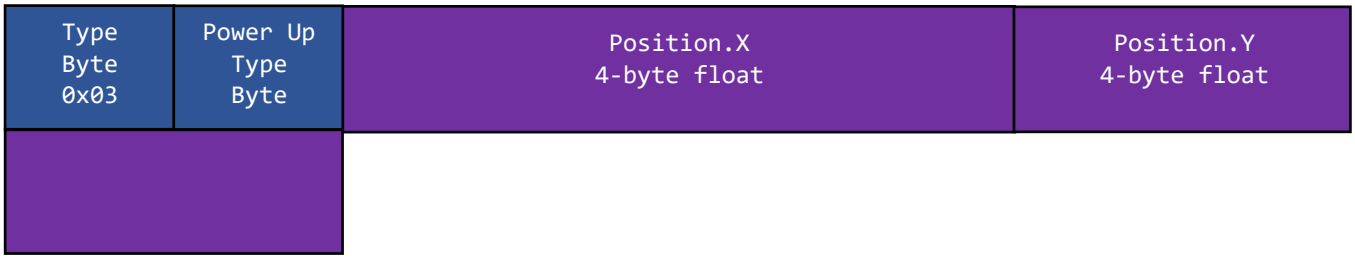
PlayerData: EntityData



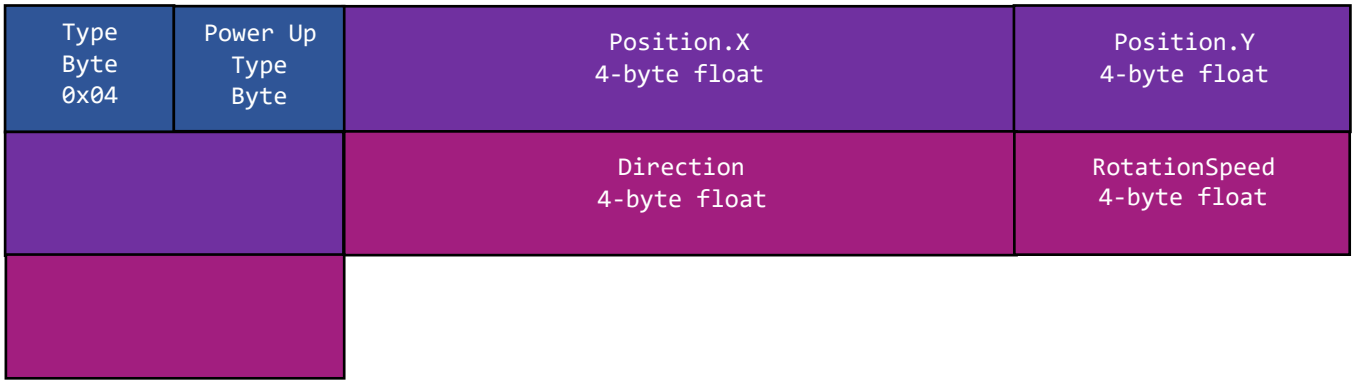
BulletData: EntityData



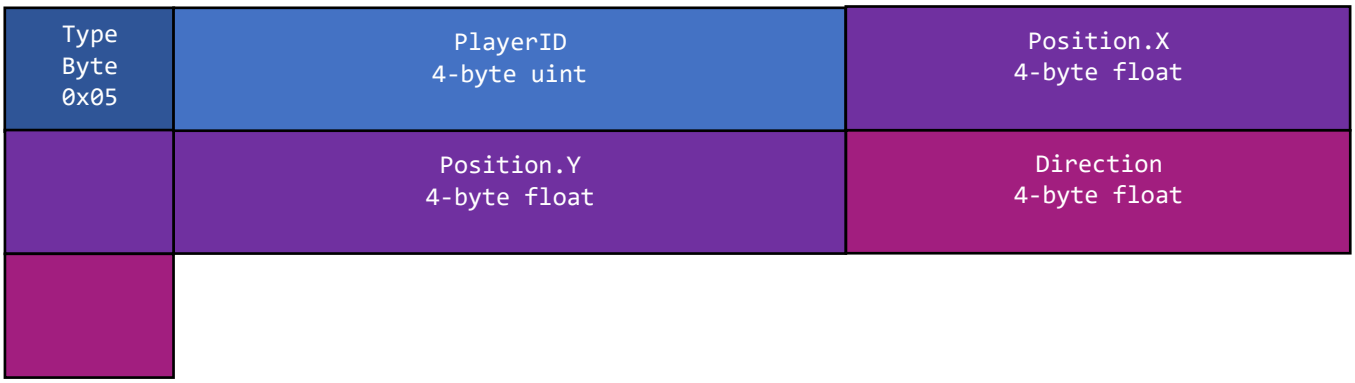
PowerUpData: EntityData



AsteroidData: EntityData

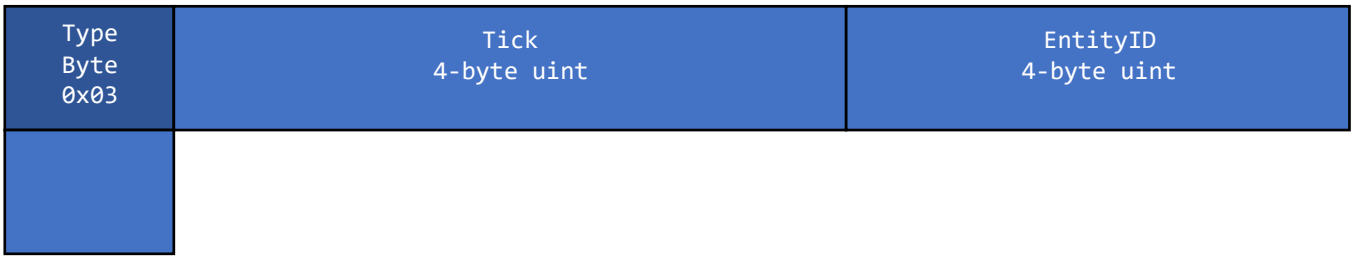


RayData: EntityData



EntityDestroy

Amikor a szerver oldalon egy entitás megszűnik ez az üzenet küldődik a kliensnek.



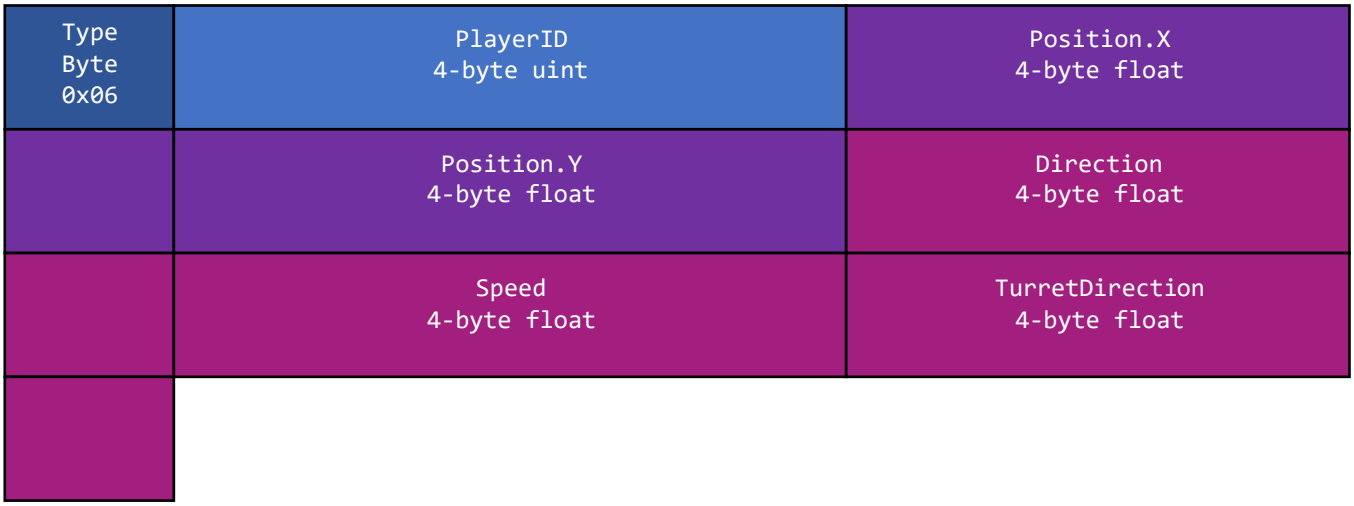
PlayerState

A `Player`-ek státusz változása (státusz = pozíció, irány, sebesség, ágyú irány)

Health

Egy `Player` élet vagy pajzsereje megváltozott

PlayerRespawned

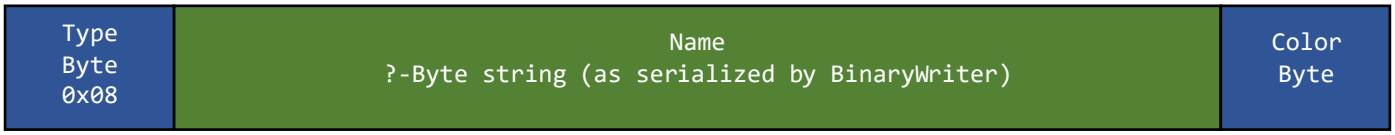


Egy `Player` respawnolt.

PlayerPoweredUp

Egy `Player` felszedett egy power up-ot. A típus megmondja milyet. A shield nem tartozik ide, arról `Health` üzenet jön.

Connect



Valaki csatlakozni szeretne a szerverhez.

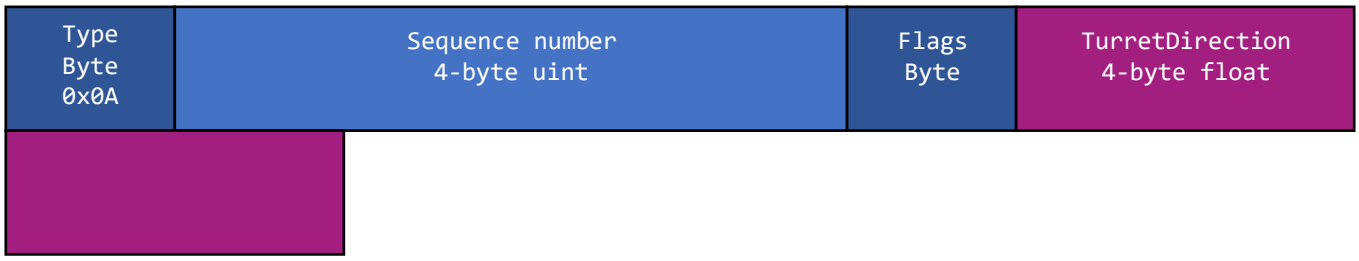
ConnectResponse

A csatlakozás elfogadása vagy elutasítása és a saját `Player` entitás ID-jának elküldésére

szolgáló üzenet.

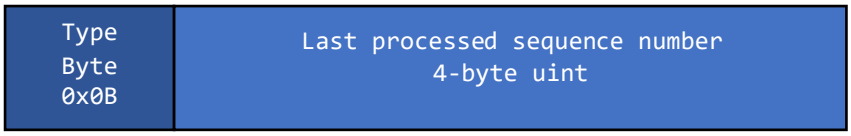
PlayerInput

A játékos bemenetét küldi le a szervernek.



A `Flags` byte a következőképpen áll össze (egy téglalap 1 bit, most significant bit first):

PlayerInputAck



A szerver elküldi minden tick után a legutolsó feldolgozott Input szekvencia számát.

Disconnect

1 byte-os üzenet, amellyek a kliens jelezheti explicit, hogy bontani fogja a kapcsolatot. A szever ezen felül timeout-olja a klienseket ha ez az üzenet nem jönne meg, de nem küld a kliens életjelet.

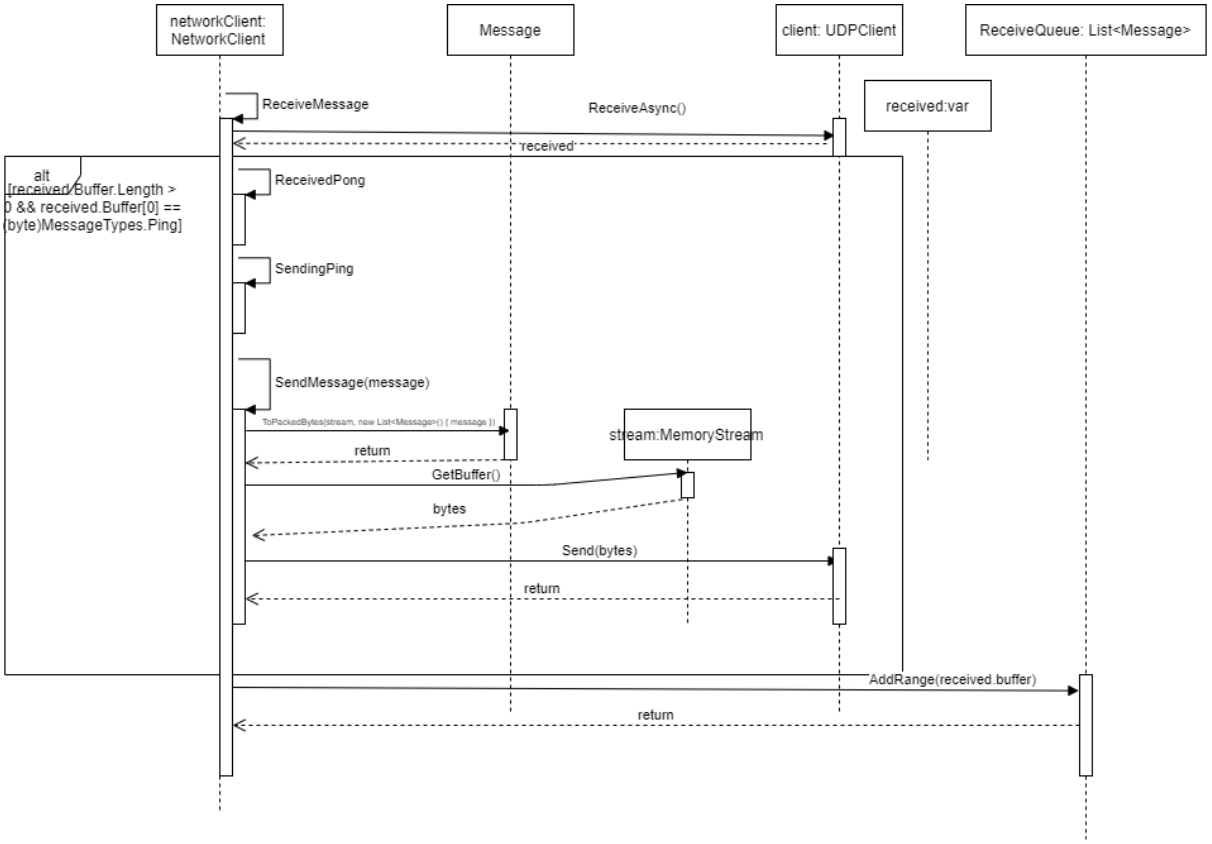
Ping

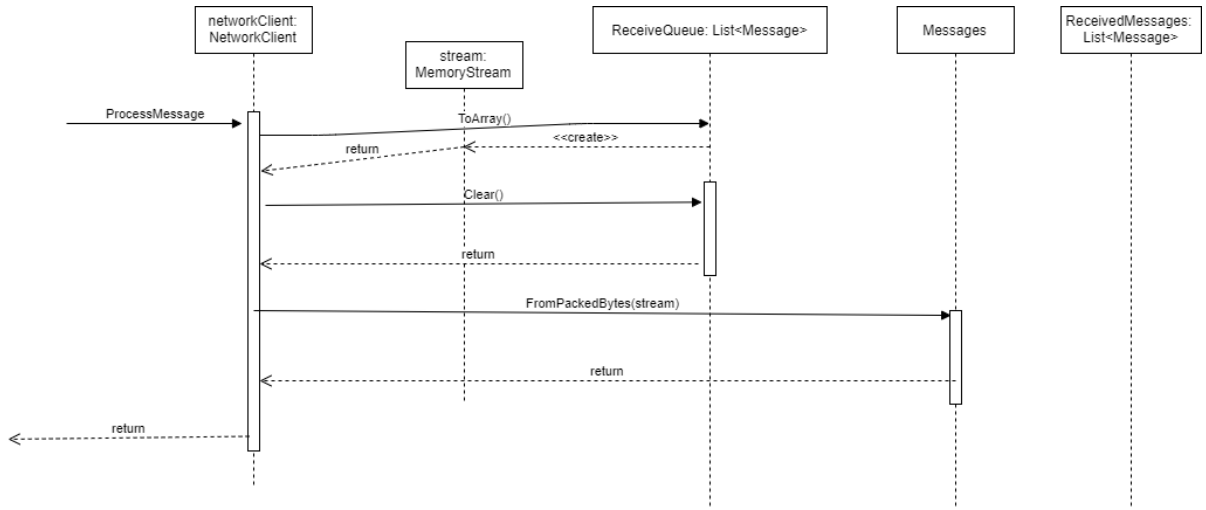
1 byte-os üzenet, amit pingeléskor küld a kliens és a szerver is. A ping-elés lényegében folyamatosan maximális sebességgel történik, így semmi időbélyeget nem kell küldeni az

```
Type
Byte
0x0D
```

üzenetekben, a két oldal ki tudja a round trip time-ot számolni.

Üzenet fogadás





Az üzenet fogadásról a `NetworkClient Listen` függvénye gondoskodik, amiben aszinktron fogad üzenetet az UDP klienstől. Ekkor egyrészt le ellenőrzi, hogy ez egy ping-e amire azonnal visszaküldi a válasz pinget, másrészt hozzáadja egy Byte bufferhez a fogadott üzenetet.

Felmerülhet a kérdés miért nem dekódoljuk azonnal? A dekódolás viszonylag erőforrásigényes és nem akarjuk egyáltalán blokkolni a listener thread-et, inkább a szerver tick-jében fogjuk dekódolni az üzeneteket rögtön a feldolgozás előtt, erre szolgál a `NetworkClient ProcessMessages` függvénye. Ezután egyszerűen üzenetek listájaként tudunk foglalkozni a beérkezett és dekódolt üzenetekkel.

Üzenet küldés

<Ide kéne egy UML szekvencia diagram a üzenet küldésről és kódolásról/csomagolásról>

Az üzenetek küldése több lépésben történik: Kódolás => Csomagolás => Küldés UDP client-en keresztül.

A kimenő üzenetek esetében is alkalmazunk üzenet sort, csak itt nem byte alapon, hanem üzenet lista alapon. Minden elküldendő üzenetet berakunk a `MessageQueue`-ba, amiből minden `NetworkClient`-nek van egy darab, szerver oldalon pedig van egy globális és egy kliensenkénti `MessageQueue` is. Ennek a feladata csak a tárolás, majd ebből fog dolgozni a csomagolás.

A mi megoldásunk szerint minden egy adott tick-en keletkezett üzenet egyetlen UDP datagramként fog kiküldődni. Ez nem feltétlen a legjobb megoldás és sokkal több üzenet esetén ésszerű lenne több kisebb csomagot csinálni az üzenetekből.

A küldés ebben az esetben nem aszinkron, hanem szinkron történik. Bár az egyes kliensek esetén lehetséges, hogy nyerhetnénk az aszinkron üzenet küldésen, nem érezhető úgy, hogy ez lenne szűk keresztmetszet.

Különböző kliensek kezelése szerver oldalon

A kliens esetében az UDP kliens egy hasznos absztrakció, hiszen egy végberendezéssel kommunikálunk. A szerver esetében nem ilyen egyszerű a helyzet, a szerver UDP kliense gyakorlatilag a szerver hálózati interfészét reprezentálja és nem az egyes klienseket. El kell tehát tárolnunk a klienseink IP cím – port párosát, hogy később is megtudjuk őket címezni. Ennek kezelésére szolgál a `NetworkClient`-el majdnem teljesen azonos implementációjú NetworkServer. Az egyes klienseket pedig a `Client` reprezentálja. A használt adatstruktúra egy `IPEndPoint` – `Client` `Dictionary`, hogy gyorsan megtaláljuk a klienseinket IP cím alapján. Különbség még a már említett per kliens `SendQueue` és `RecieveQueue`, ami szintén a `Client` class-ban található.

Mivel a `Client` alapvetően csak hálózati absztrakció magának a `GameServer`-nek is kellett egy adatstruktúra, ez a `GameClient`, amit szintén egy `IPEndPoint` – `GameClient` `Dictionary`-ben tárolunk a `GameServer/Program` osztályban. Ez felel a `KeepAliveTick` számláló és a hozzá tartozó `Player` entitás tárolásáért.

Dead reckoning használata az effektíven statikus entitásoknál

A játék alapvetően 3 fajta entitásból áll Networking szempontból:

- `Player`
- `Bullet`
- `Power up`

A 3 entitás különböző jellegű üzenetekkel frissül. Egy `Bullet` esetén semmi értelme nincsen újra és újra lekérni a szerver-t a pozícióról. Csak a létrehozás és törlési üzenetek fontosak, ezért a szerver nem is küld a pozícióról üzenetet. A kliens tudja úgynevezetten prediktálni (jósolni), hol lesz a lövedék a következő frameben. Amikor azt feltételezzük, hogy egy lövedék nem mozog sehova csak amerre fele kilőtték az ún. dead reckoning elvet alkalmazzuk. Ez az elv alkalmazott gyakorlatilag a Power up-ok szempontjából is, hiszen azok pont a másik véglet, nem mozognak.

A `Player` esetében nem ilyen egyszerű a helyzet így nem is tudunk igazából semmi jól működöt csinálni bonyolultabb algoritmusok nélkül.

Egy speciális eset, hogy a `Player` entitás ágyújának az irányát mindig „elhihetjük lokálisan”, hiszen az csak a vizualitást fogja befolyásolni és úgymond real time frissül, a szerver nem befolyásolja.

Lag kompenzációról

A játék hálózati architektúrájánál minden erőfeszítés ellenére nem sikerült értelmes lag kompenzációt összehozni. Így a végső megoldás alapvetően a szerver oldal válasza után lépteti csak az inputot érvénybe. Így bár a kliens oldalon létezik kliens oldali predikcióra lehetőség az nincs kihasználva a `Player`-ek esetében.

Ennek a gyakorlati következménye, hogy a ping-nek (round trip time) megfelelő input lag-ot tapasztal a játékos. Ez egy limitációja a kurrens implementációnak.

Hogyan próbáltuk ezt a problémát megoldani?

Mivel ez egy igen nagy probléma és megtöri a játékélményt akár már 50-60 ping esetén is, ezért más megoldásra lenne szükség. Egy lehetséges megoldás, hogy a lokális `Player`-t a jelenben tartjuk, vagyis „elfogadjuk” az inputját feltételezve, hogy a szerver is ugyanúgy

fogja feldolgozni, ami elvben igaz is kis eltéréssel (csomag vesztés, csomag átrendeződés vagy egyszerűen időzítés béli okokból).

A szerver által leküldött lokális `Player`-t tehát frissíteni kellene, ha egy bizonyos küszöböt átlép a hiba pozícióban, forgásban vagy sebességben. Ehhez tudnunk kellene mi volt a változás amióta a szever feldolgozta az inputot. Ehhez az inputokra egy `Sequence number`-t (sorszámot) rakunk, amivel azonosítani tudjuk az inputjainkat. A kliens elmenti a lokális `Player` státuszát a bizonyos `sequence number`-ekhez és ezt veti össze a szerver által visszaküldött `Player` státusszal, amely visszaküldi a legutóbb feldolgozott `sequence number`-t.

Idáig el is jutottunk, ekkor az ötlet az lenne, hogy kivonjuk a kliens mentett pozíciót és hozzáadjuk a szerver által visszaküldöttet. Elvben ekkor a delta ugyanúgy maradna és a játékos egy kisebb „ugrást” venne észre, de semmi több.

Azonban ez a megoldás nem sikerült. Ez a fajta korrekciós algoritmus divergens és túlkorrigál. Valószínű oka ennek az, hogy a szerver és a kliens még sincsenek annyira közel időzítésben. Ennek a problémának megoldásához azonban a fejlesztő csapat kevésnek bizonyult.