# 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 DrawableEntity – 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 (a fejlesztőkön kívül álló okokból). 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.  
<Ide érdemes rakni egy packages UML diagramot>

# 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]](#footnote-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.

<Valami deployment-es diagram>

# A játék logika

## Játék szabályok

## Entitás menedzsment

## Ütközés detektálás

## Player input kezelése

# A játék motor

## Sprite generálás

## Renderelés

## Render-only entitások

# Szerver-kliens kommunikáció megvalósítása

## Múltbéli próbálkozások

## Byte alapú üzenet formátum

## Kimeneti MessageQueue

## Bemeneti MessageQueue és a listener

## Üzenet csomagolás

# 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.

# 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-ünknek (latency) 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 numbert.

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.

1. Tudom, hogy JSON-ban nincsen comment. [↑](#footnote-ref-1)