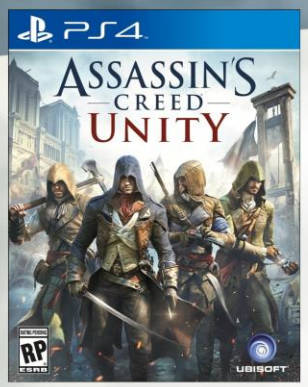
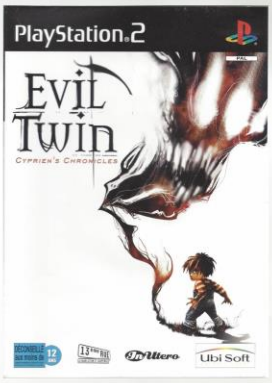
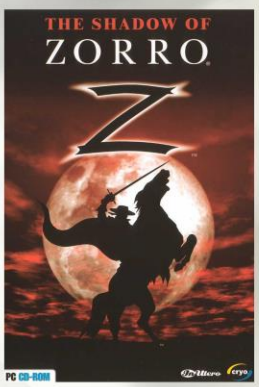
Gameplay and AI networking in Assassin's Creed Unity

Charles Lefebvre
AI programmer, Ubisoft Montreal
charles.lefebvre@ubisoft.com, @FrozenInMtl



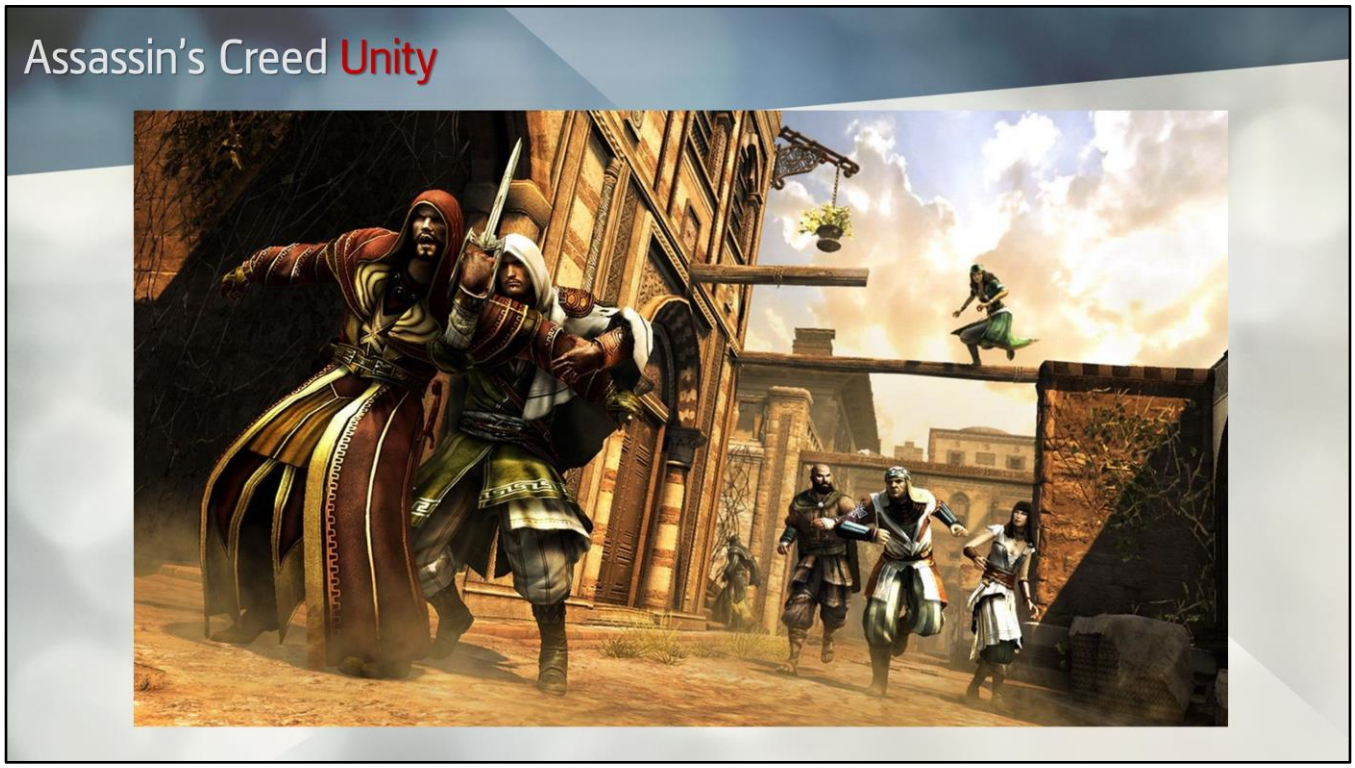




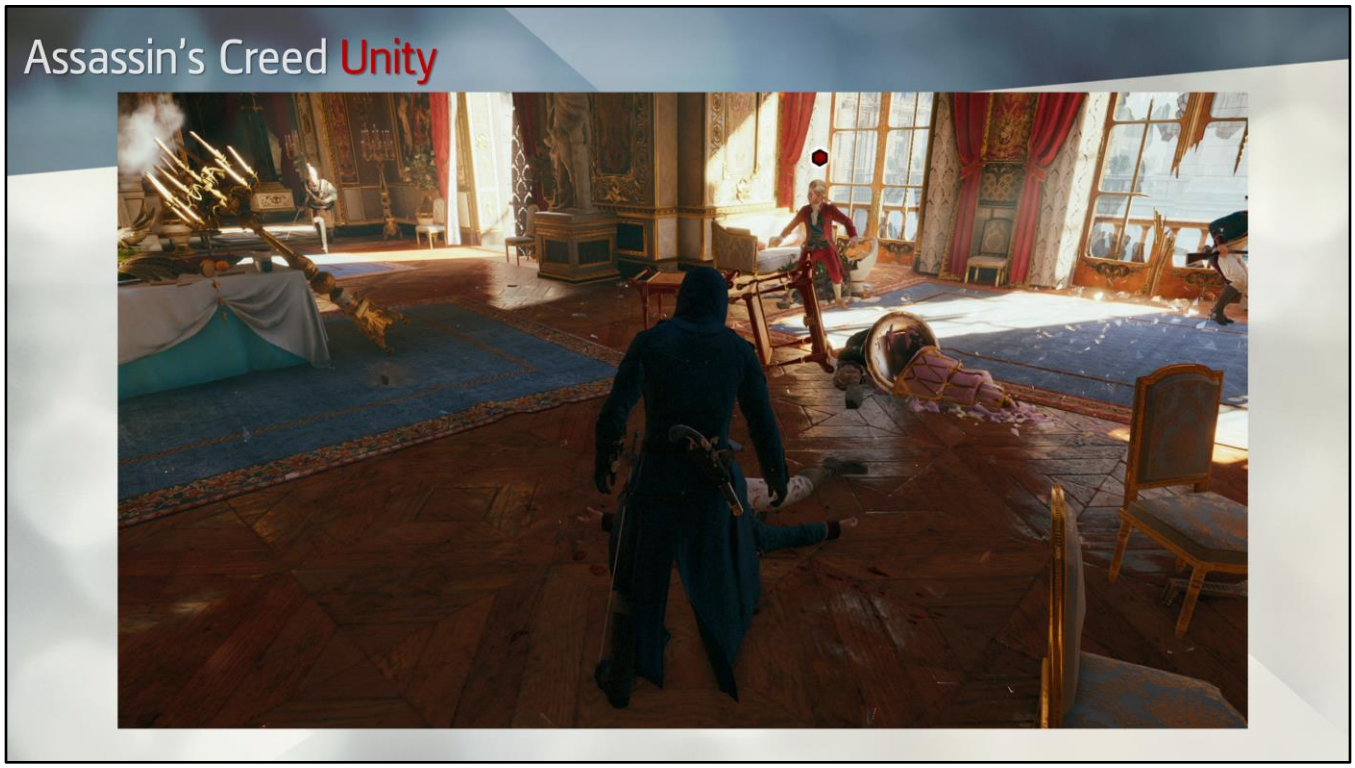
Charles Lefebvre
AI programmer, Ubisoft Montreal
charles.lefebvre@ubisoft.com, @FrozenInMtl

Assassin's creed is a next-gen game, in Paris during the French Revolution, with huge crowds…

And coop



In the previous Assassin's Creed, the multiplayer game was a different executable. It was running in a different map, not in an open world. Also it was using a server.



Let's see it !

Open world game

Coop peer to peer, 1 to 4 players

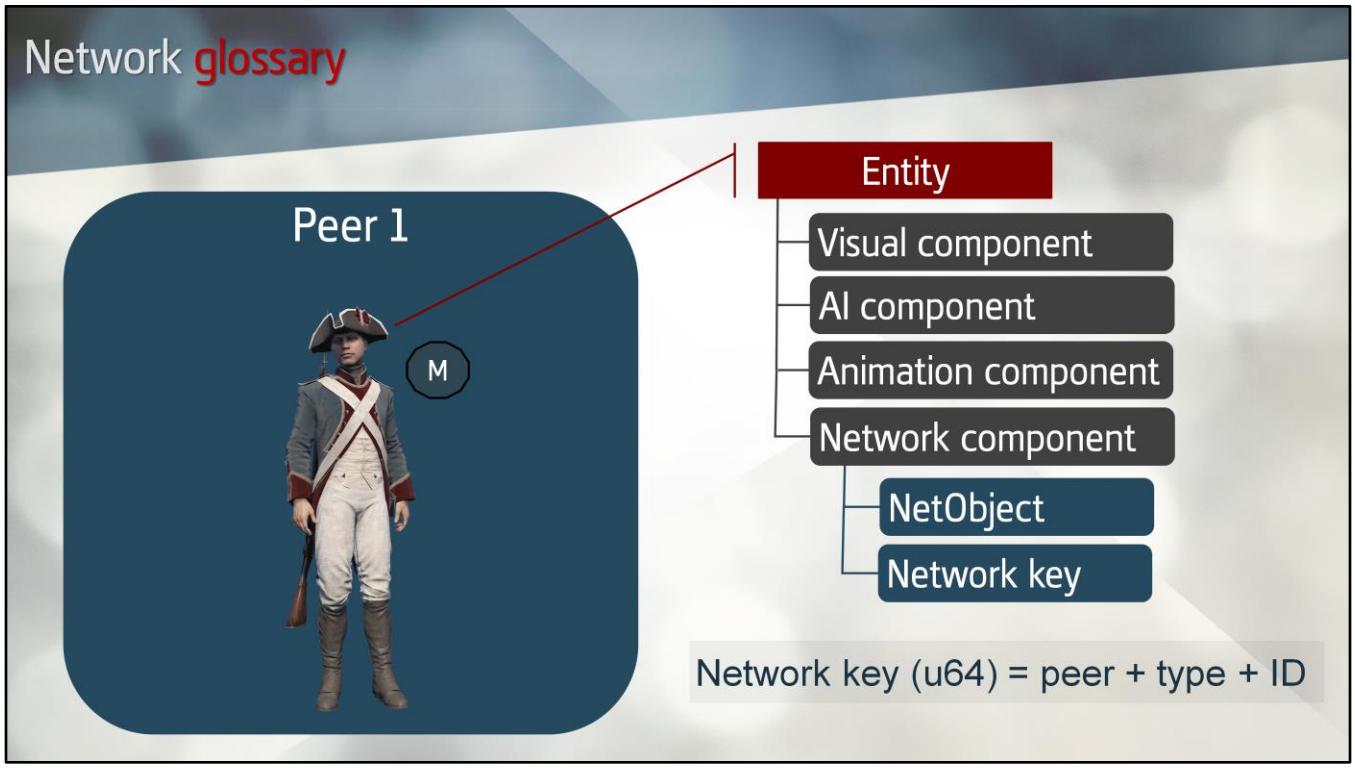
Huge existing code base

Simulation bubble around each player: 80 meters

Loading grid: blocks of 32 meters (around 40 blocks loaded)


NETWORK
TOOLS

Lost of technical worlds….



Peer = machine of the gamer

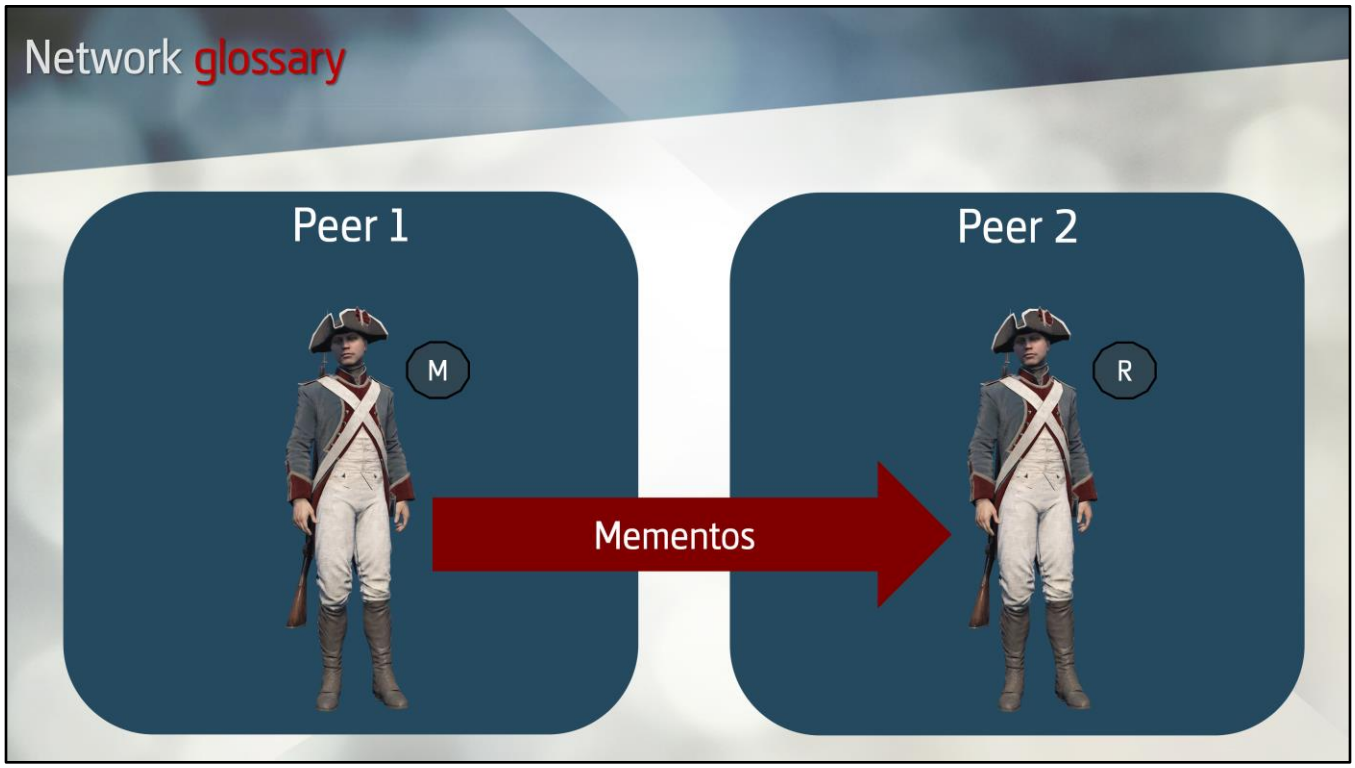
NPC is an entity

Component architecture

Entity with network component = replicated entity

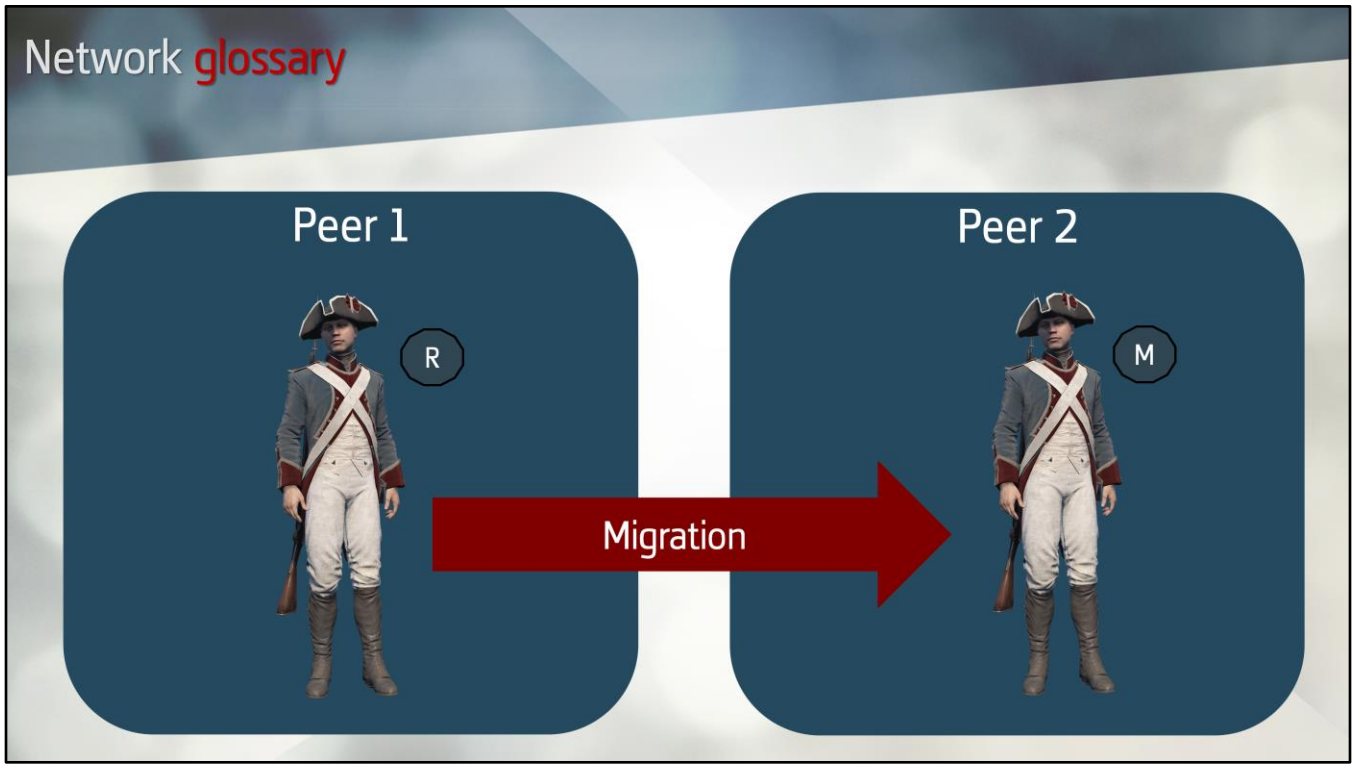
NetObject: used for network discovery, and communication

1st entity spawned = master



Replica created when master is discovered

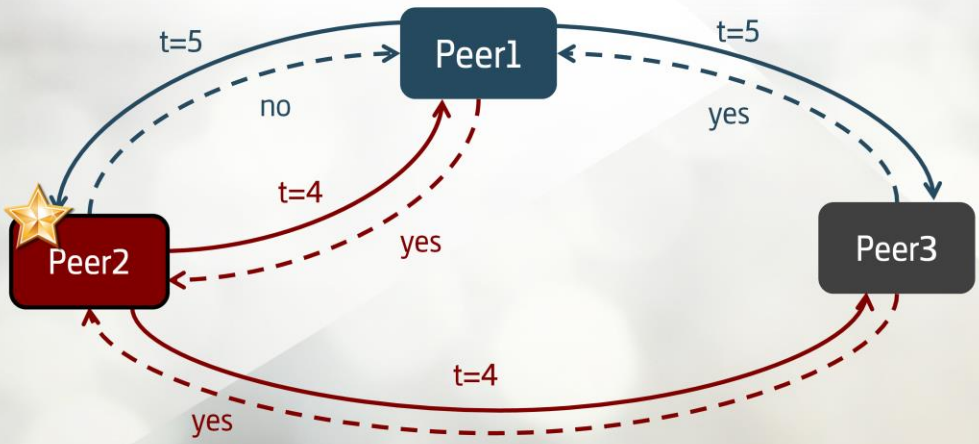
Mementos: auto replication of variable (master to replica)



Migration: switch from replica to master

Network tools: NetTokens

NetTokens: Layer for peer to peer conflict resolution



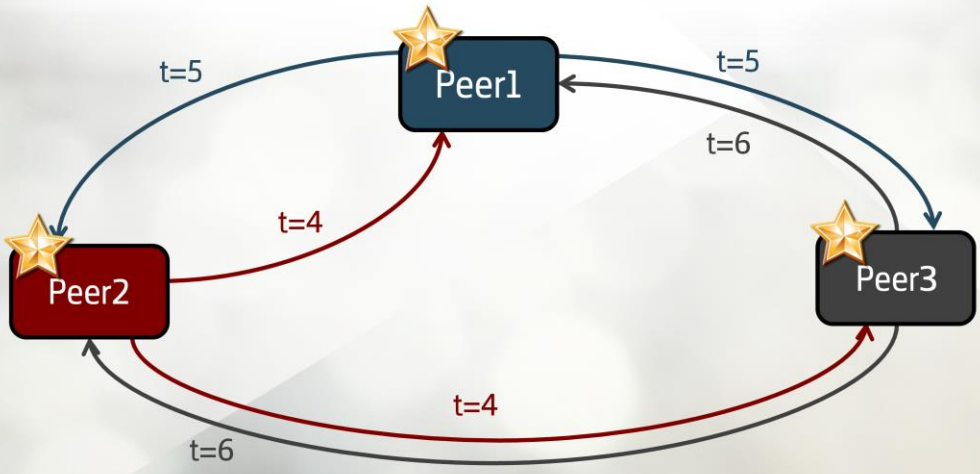
NetTokens are time based: the first one that has requested the token is the winner.

The time used is the network time. The time is in each message, so each peer get the same comparison result.

In case of equality the peer with the lower peer ID wins.

Network tools: NetTokens

Can also be used to wait for all players



Network tools: remote procedure call

Remote procedure call (RPC): execute a function on a peer, this function will be executed on other peers

```
network void PlayAnimation(const AnimationParams & params);
```

- Can be persisted for Join In Progress
- Can be executed later, or re-executed
- Can be ordered or not

A RPC is converted into a reliable message.

A RPC can be persisted for Join In Progress: these functions returns a cookie than you can persist. As long as it is persisted, this function will be executed on all peers that create a new replica of this object.

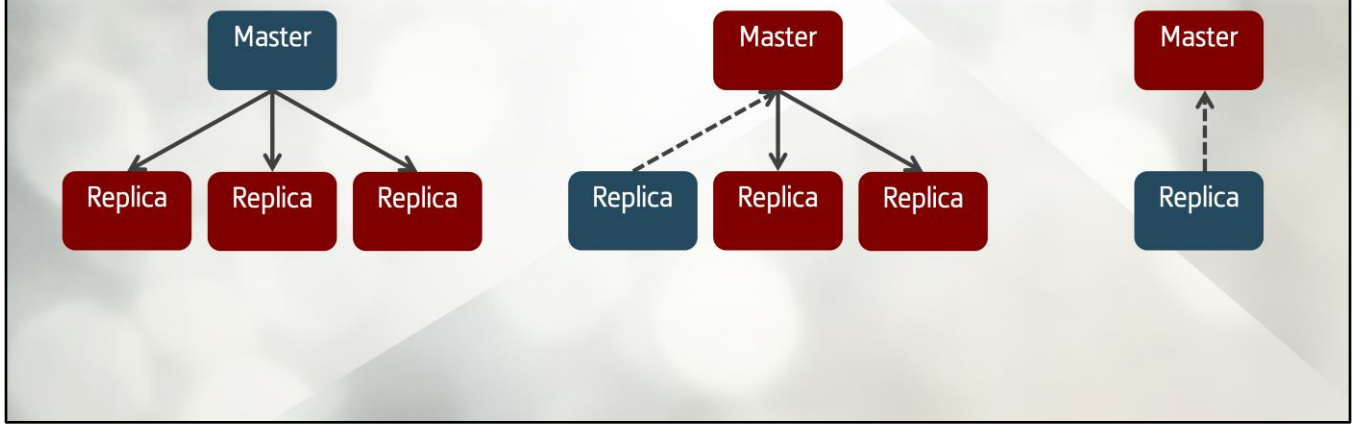
You can execute or reexecute a RPC through the cookie.

If you receive a RPC *PlayAnimation(Animation * anim)*, but your entity is not ready to play this animation, you can keep the cookie, and execute the RPC when your entity is ready. You can even call *PlayAnimation* each frame until it is successful.

If RPC1 is ordered, RPC2 is not, and RPC1 transport packet is lost, then RPC2 can be executed.

Network tools: remote procedure call

Flow examples:



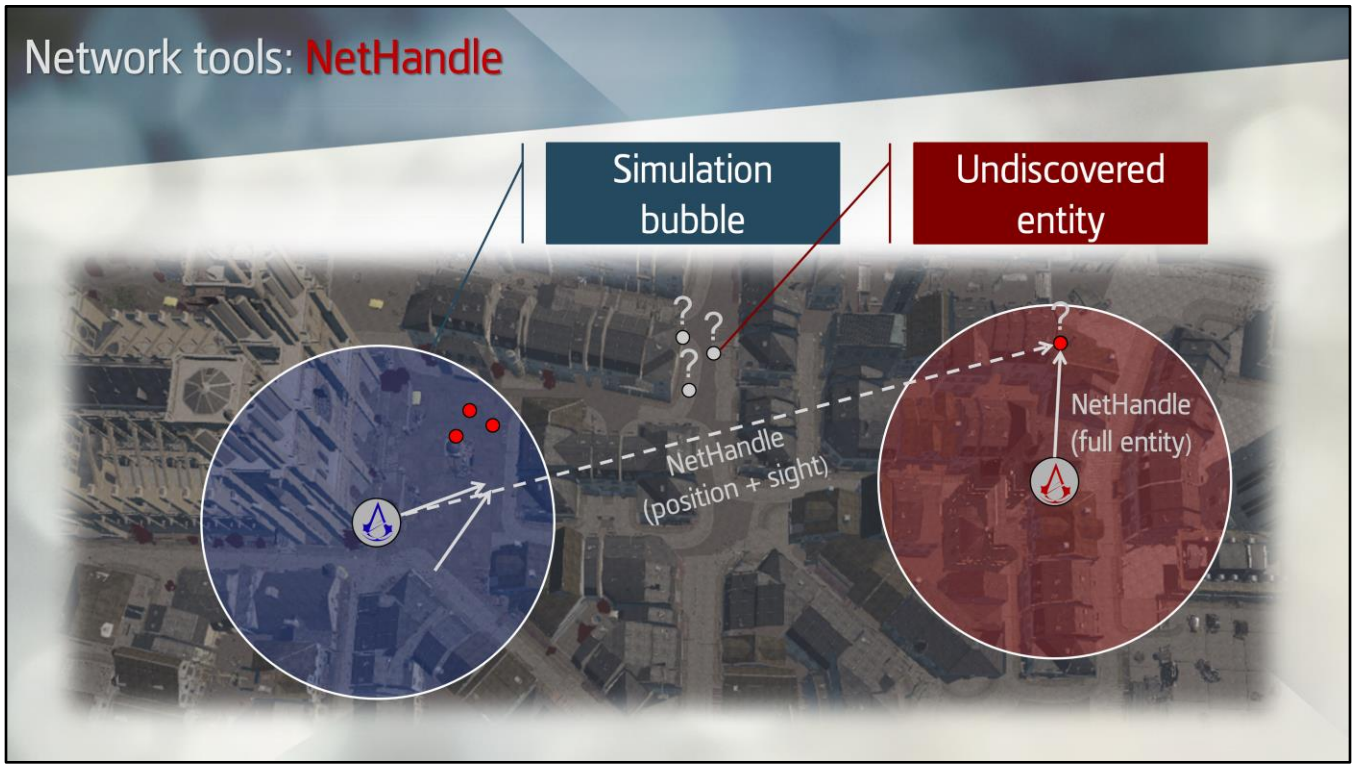
Example 1: the master executes a RPC, it is then broadcast and executed on all replicas.

Example 2: a replica executes a RPC, it is then unicast to the master that executes it, and the master broadcast it to all the replicas except the initial sender.

Network tools: NetHandle

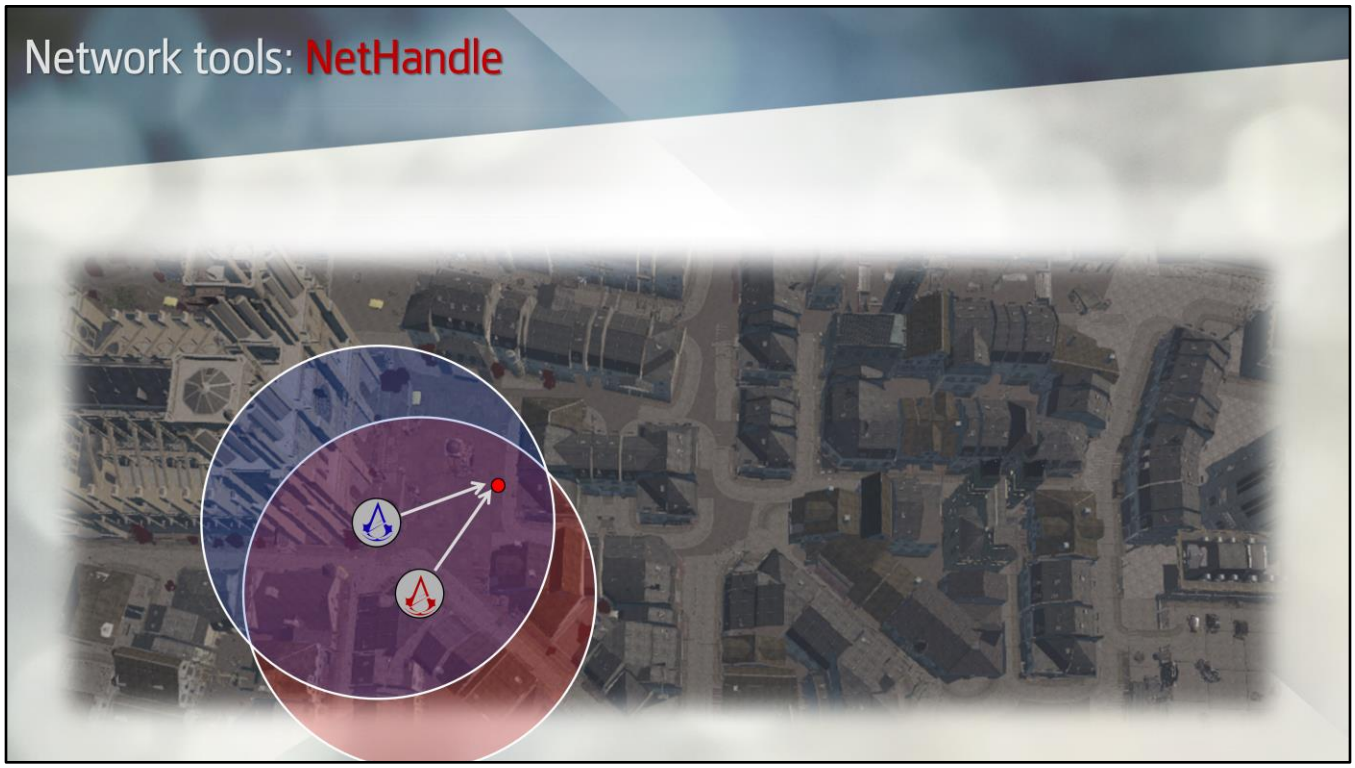
```
struct NetHandle
{
   Handle<Entity> m_Entity;
   NetKey m_NetKey;

   bool IsValid();
   Entity * GetEntity();
   bool GetPosition(vec4& position);
   bool GetSight(vec4& position);
}
```



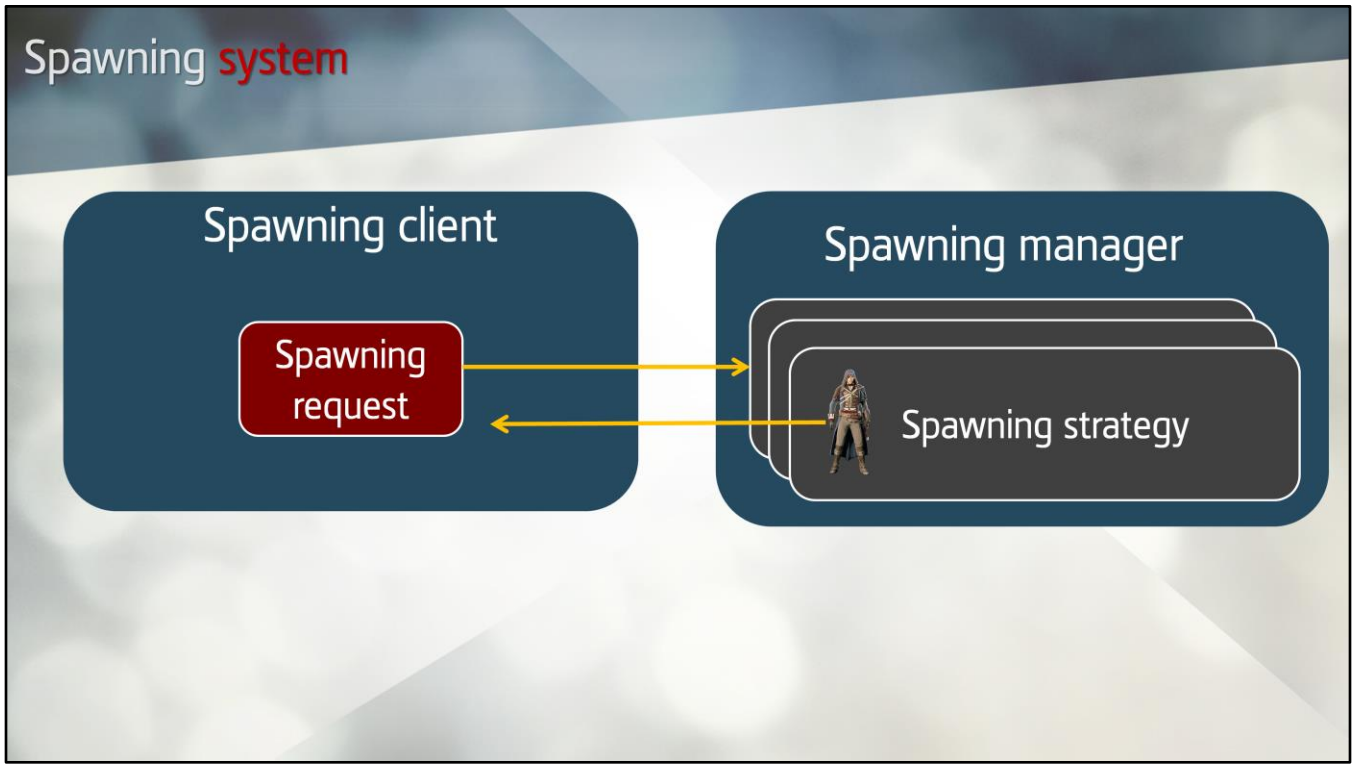
If player 2 has created a transient replicated entity, and it sends to player 1 a NetHandle to this entity, then player 1 can get the position and the sight of this entity. It can also find out if this entity still exists for player 2.

Network tools: NetHandle



If the entity and the red player move toward the blue player, the entity is discovered and the NetHandle is resolved.


SPAWNING
SYSTEM



Asynchronous process: pool of spawning requests.

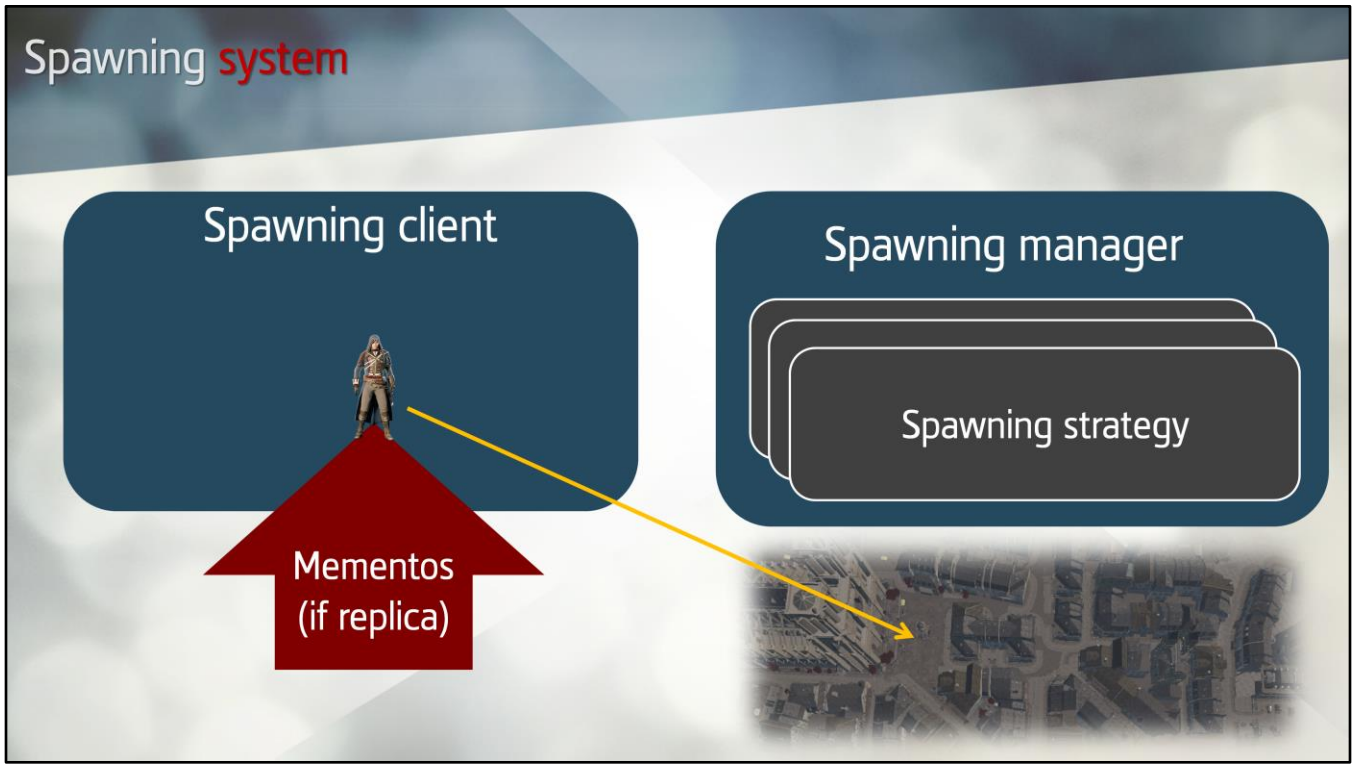
The spawning manager execute each frame the most urgent spawning requests.

Different strategies:

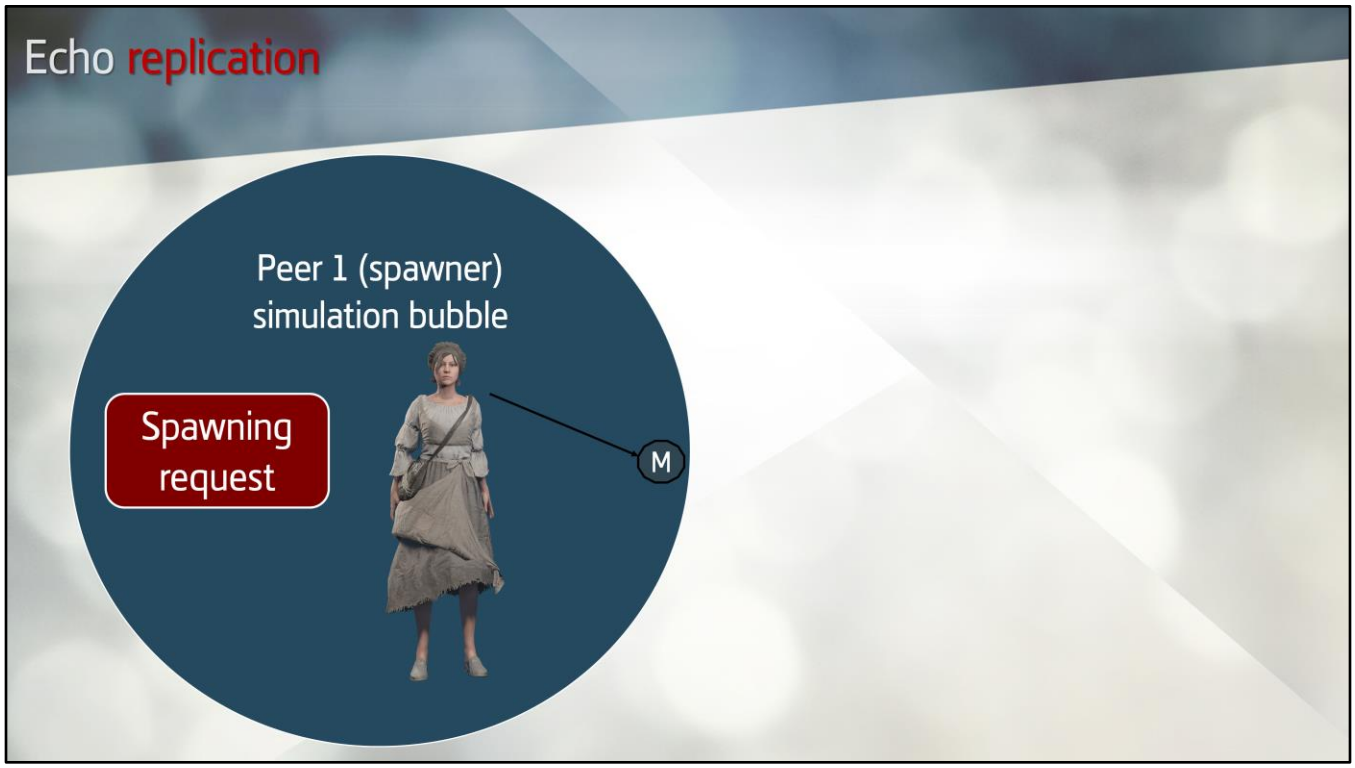
- Spawning: generate a new entity.
- Acquisition: reuse an entity released by another spawning client.

Around 500 spawned NPCs, but less than 60 are individually replicated.

The entities not individually replicated are managed by the massive crowd system that handles replication at a higher level.



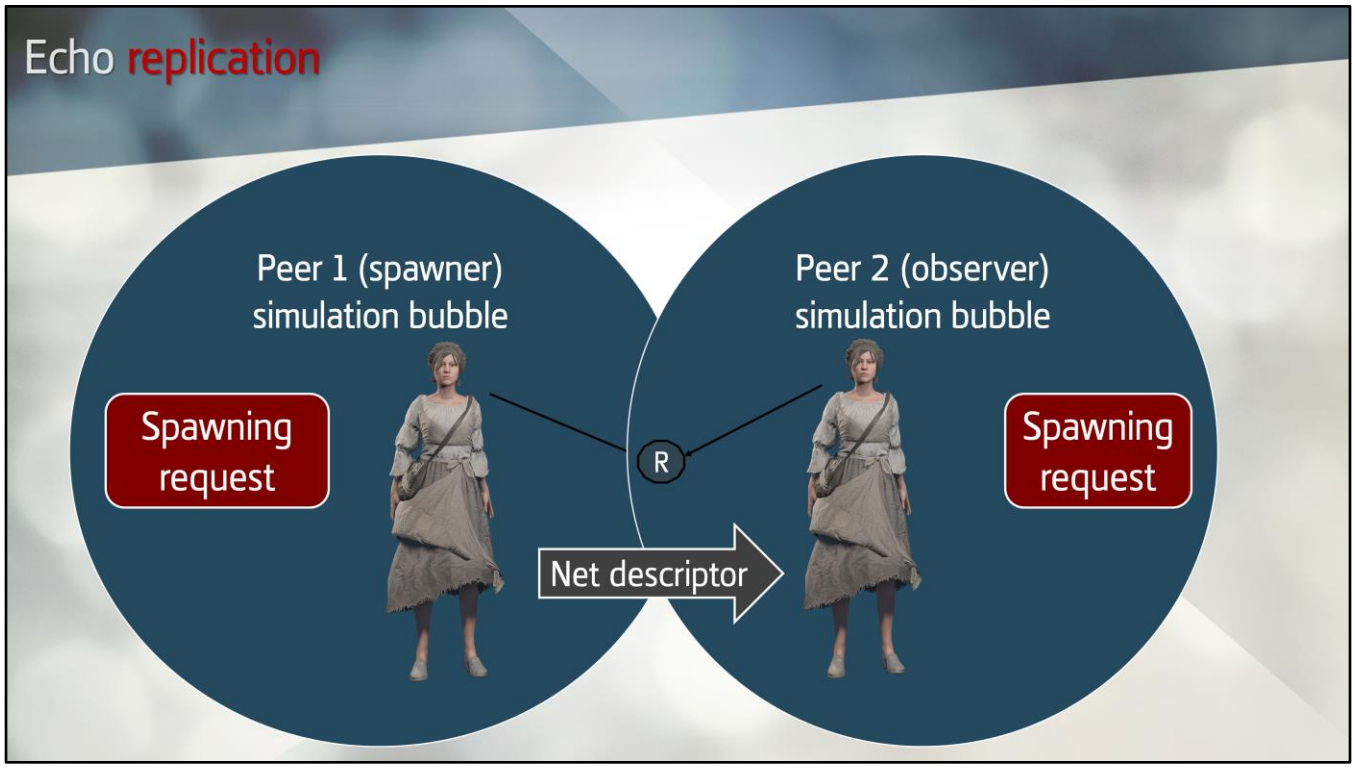
If the entity is a replica, mementos are applied before adding the entity to the world, so it is in sync.

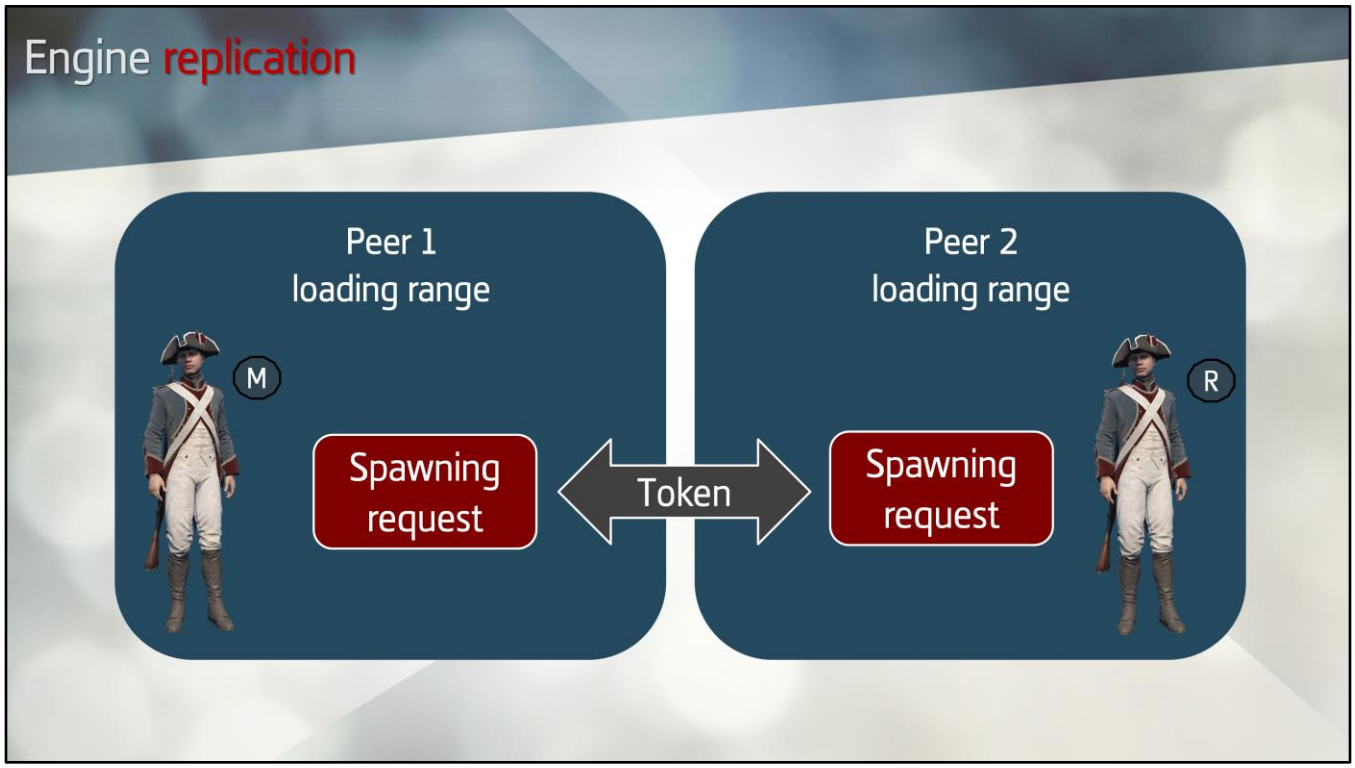


In Echo, master is spawned, replicas are automatically created

This is for systems with unpredictable spawning request, ex: crowd members, players

Only the NPC type and a seed are broadcast.


Echo replication
Peer 1 (spawner) simulation bubble
Peer 2 (observer) simulation bubble
Spawning request
R
Net descriptor
Spawning request



In engine, a gameplay system decides to spawn an entity.

This gameplay coordinator can be active on multiple peers, and we don’t want to have as many entities than peers.

A spawning request is added on all peers. It asks for a token, and the spawning request that manage to acquire the token will spawn the master. The token is also used to set the netkey of the entity.

The other peers will spawn a replica.

The generation seed is generated with the ID of the spawning client, so nothing is broadcast as the NPC type is known by all peers

Systemic gameplay

- Replicate the crowd event generation in echo
- Replicate the victim/thief in engine

Spawning system: conclusions

(+) Almost no change in the spawning clients code

(+) Cheap in bandwidth

(+) Data is the same in single and multiplayer

(-) No network balancing

(-) A lot of edge cases to handle when a spawning request is cancelled

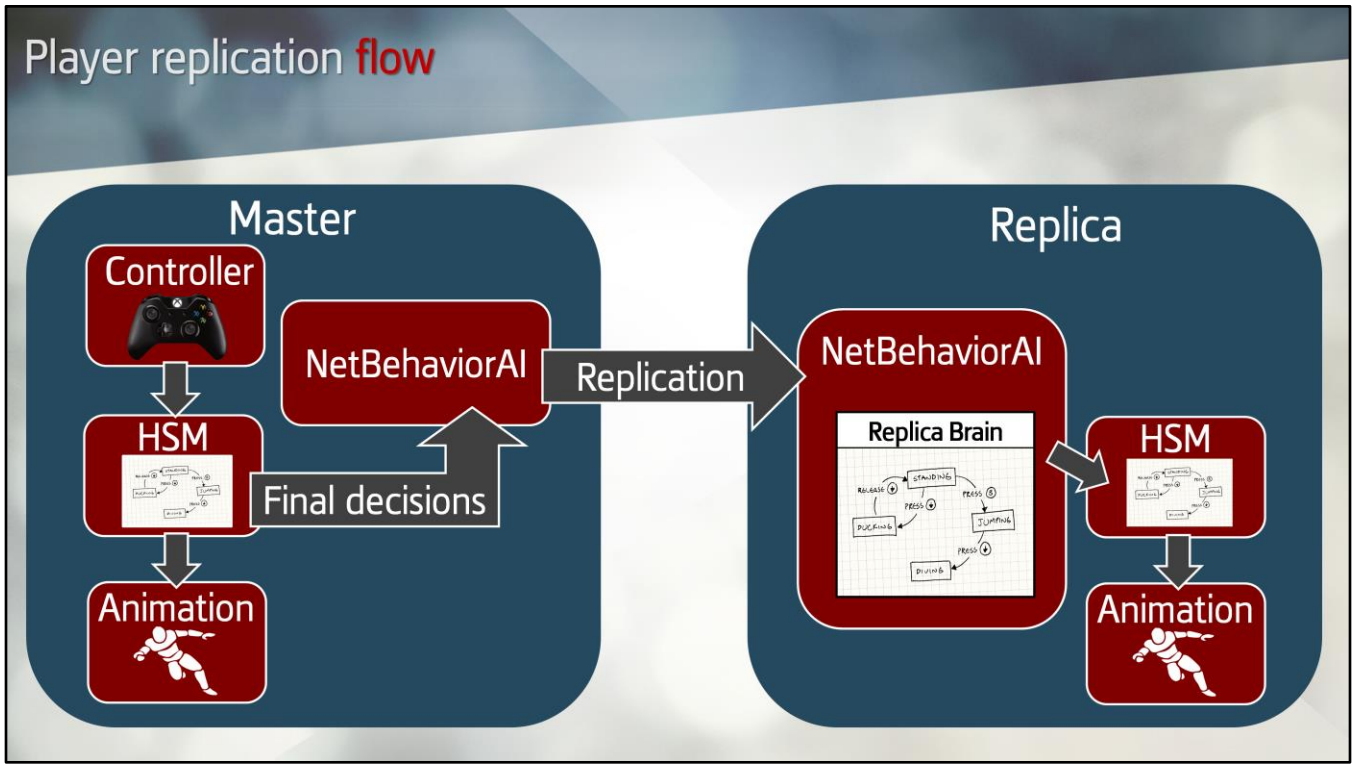
(-) Systemic gameplay is not easy to replicate


REPLICATION OF
PLAYER BEHAVIOR

Player replication: objectives

- Need a way to replicate the player's movements with good accuracy
- Animation quality of replicas must be the same as masters
- Must support join in progress
- Shouldn't take too much bandwidth

This is a tool to debug player replication: a player replica is created locally, and receives all messages with a fixed latency.



Controller = hardware, HSM (Human State Machine) = code, Animation = data

We could replicate the controller input, but the game is not deterministic.

We could replicate the animation, but that would be very expensive.

We are replicated the animation logic.

Replication state machine allows to cope with network realities such as loss packets, latency variations.

Not too intrusive on HSM code.

Virtually no environment checks on replicas, all done by the master = low CPU cost

Player final decisions

What is a final decision ? Land on beam, enter window.

Player replication: conclusions

(+) Good replication quality

(+) Can recover nicely from divergences

(-) Expensive in bandwidth

(-) 3 state machines (HSM / NetBehaviorAI / animation graph) to maintained

3 state machines:

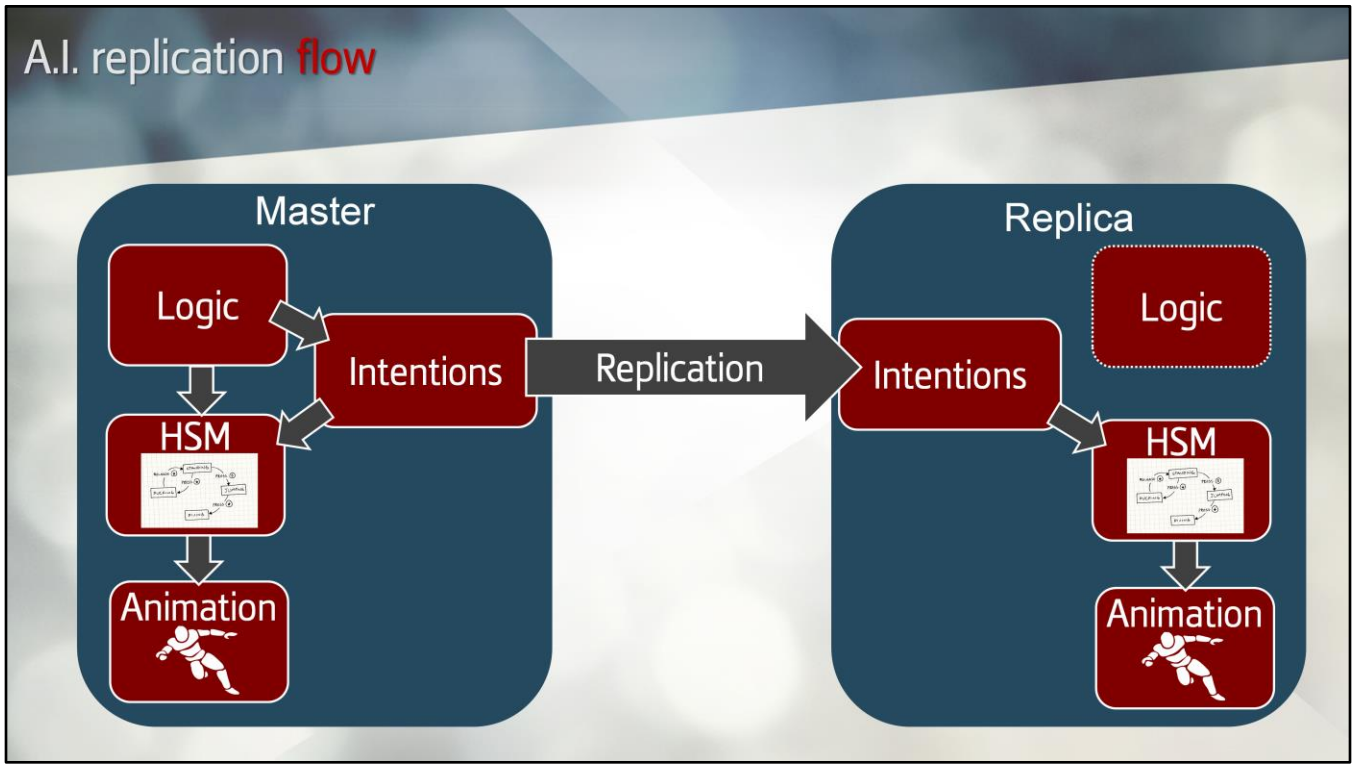
HSM -> enter in all states + send final decisions

Animation graph: extra transitions for the replica


A.I. REPLICATION

A.I. replication: objectives

- Replicate NPC behavior at a very low cost
- Not a perfect match: not a first person game
- Quality of realization: same for masters and replicas
- Existing behaviors easily updated to support multiplayer



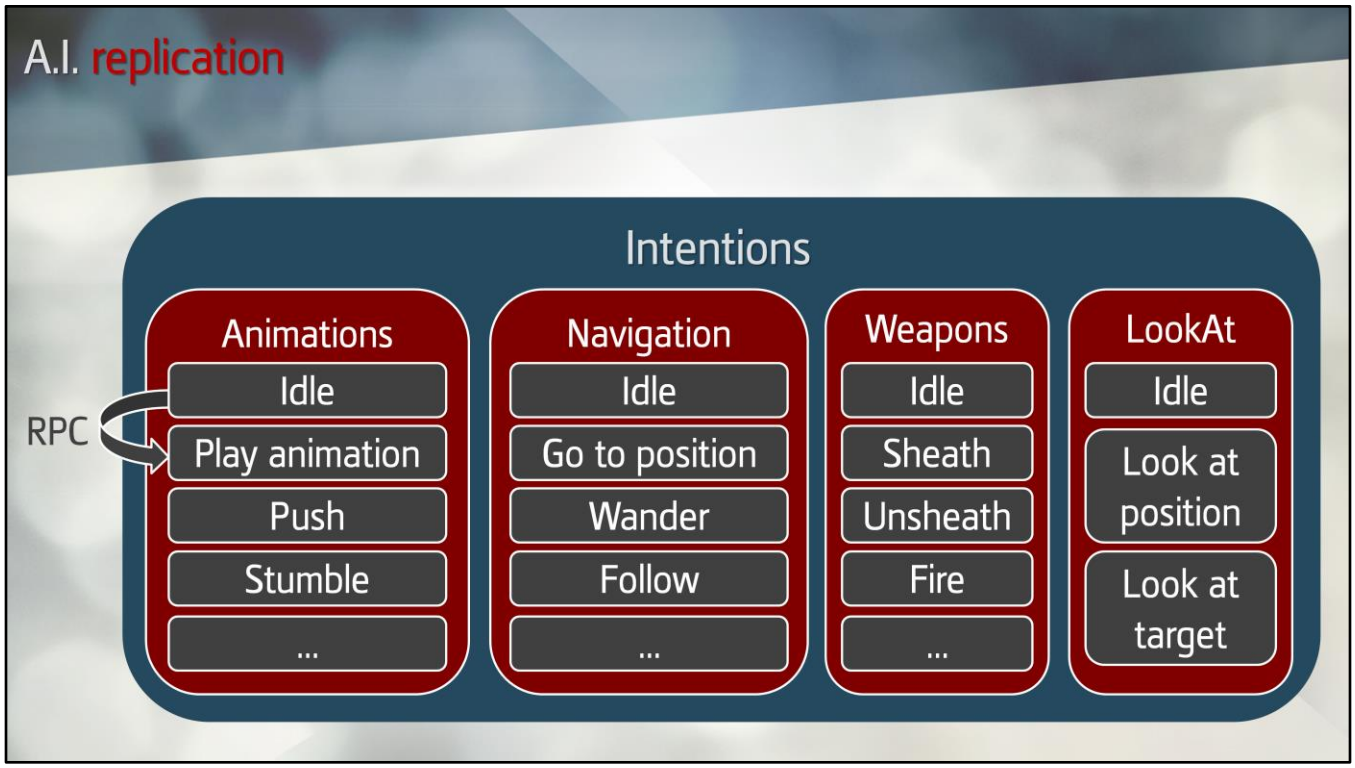
Cannot reuse the player behavior replication:

- Too expensive in bandwidth.
- Doesn't handle migration: the local player is always a master

Ideally NPCs should be replicated at the character logic.

Since there is a huge existing code base, we have actually added a new layer of replication between the character logic and the animation logic

Intentions: layer of replication between the character logic, and the animation logic



The intention framework is a state machine. There are 4 concurrent states: Animations, navigation, weapons and lookat.

All these states can be active at the same time, as we can have a NPC walking, while playing an upper body animation, with his sword unsheathed, looking at his target.

Under all these concurrent states, we have some exclusive states.

First, there is always an Idle state, when the concurrent state is not active.

Then we have some states for all the possible actions

Transitions to an active state are triggered by RPCs, and are a persisted for join in progress.

Once an action is completed, there is a transition to the Idle state; the RPC is unpersisted.

NPC replication : conclusions

(+) Cheap in bandwidth

(+) Seamless migrations

(-) Average replication quality

(-) Services (navigation, animation…) must also be synchronized to prevent behavior branching

GAMEPLAY COORDINATORS

Gameplay coordinators

- Spawn NPCs
- UI
- Interactions

A gameplay coordinator manages the behavior of a list of NPCs, and handle their interactions with the player.

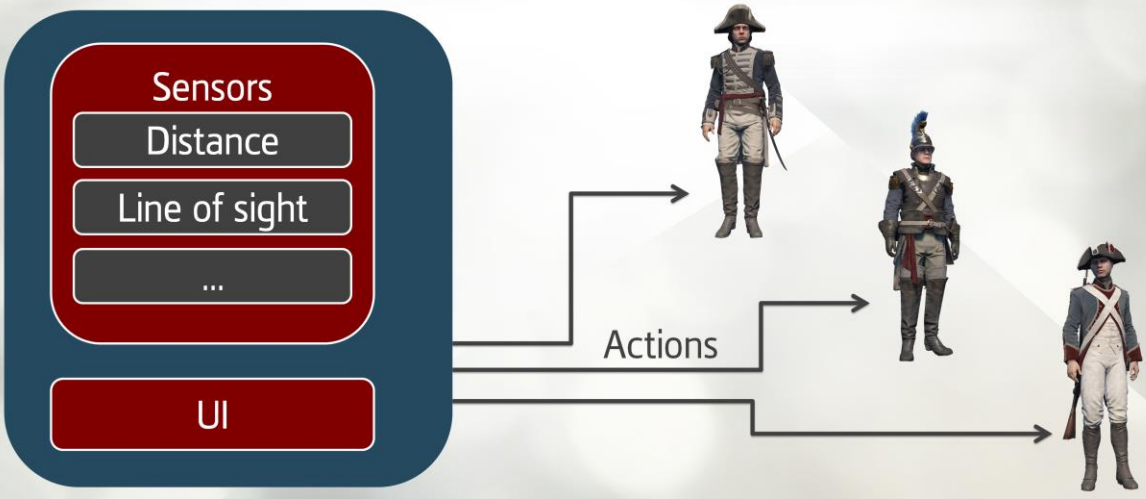
Once the NPC is spawned, the gameplay coordinator pushes a character logic on its AI component.

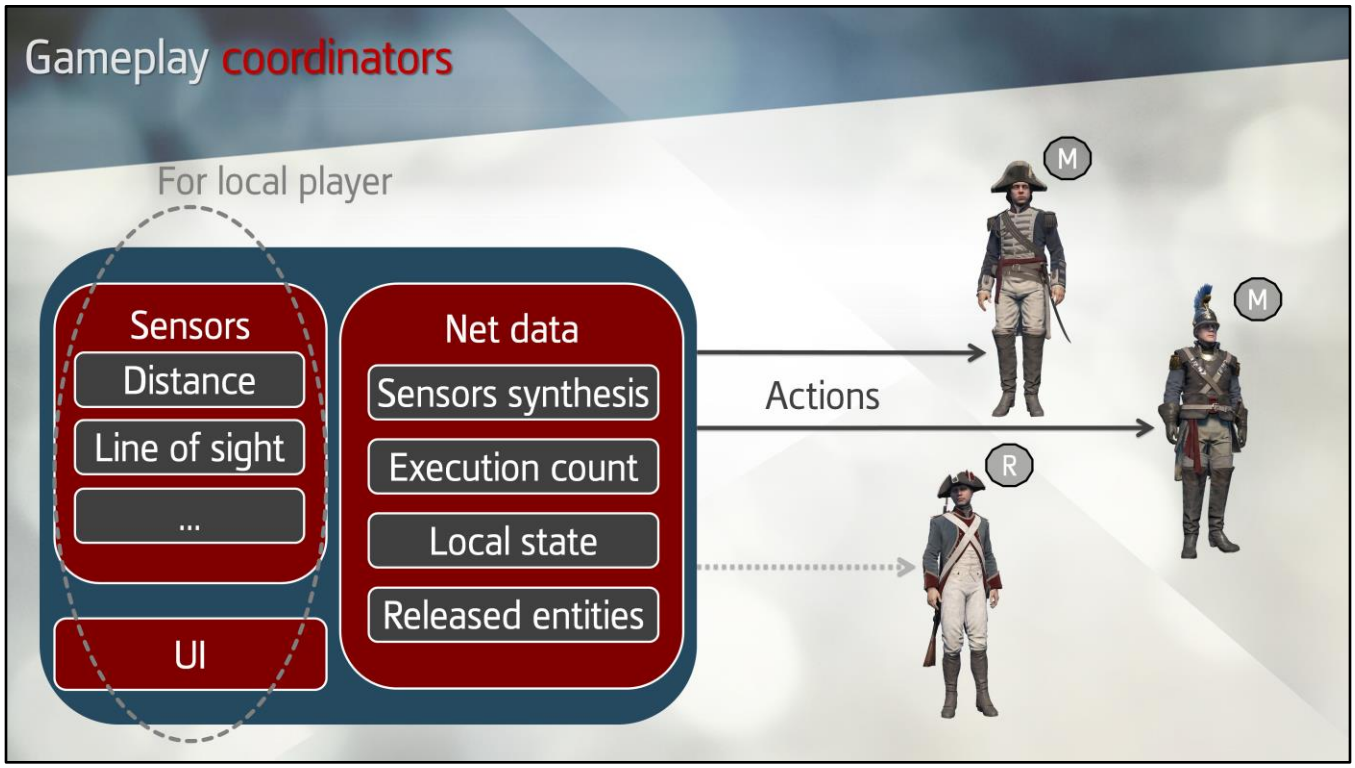
Examples of gameplay coordinators: tail, defend, patrol guards…
Around 40 different coordinators in ACU.

Examples of character logic: navigate, fight, search, play animation…

Gameplay coordinators

Coordinators in previous Assassin's Creed games





- Actions are not applied on replicas
- Sensors and UI is for local player
- Net data is shared between all instances
- Actions are applied on masters

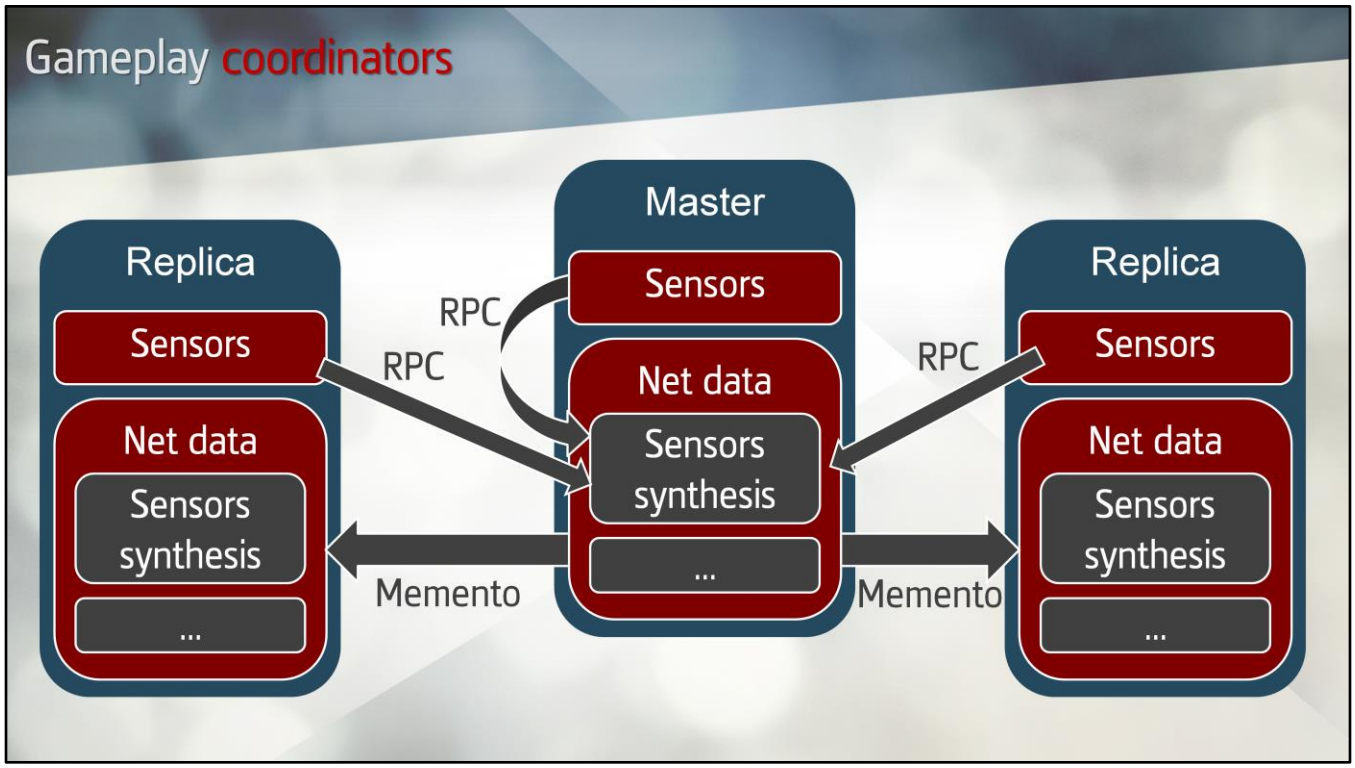
A replica coordinator sends data to the master through RPCs. The master does the synthesis of these information, and broadcast it to the replicas through a memento.

Since masters and replicas share the same data, they take the same decisions.

In case of migration, or disconnection, everything is already shared.

Example for shared data: for an interaction coordinator, each peer will notify the master whereas the local player is ready to trigger the interaction.

All players will be aware of the state of the other players, and will allow the interaction accordingly.


Gameplay coordinators
Master
Sensors
Net data
Sensors synthesis
...
Replica
Sensors
Net data
Sensors synthesis
...
Replica
Sensors
Net data
Sensors synthesis
...
RPC
RPC
RPC
Memento
Memento

Gameplay coordinators: conclusions

(+) Good replication quality

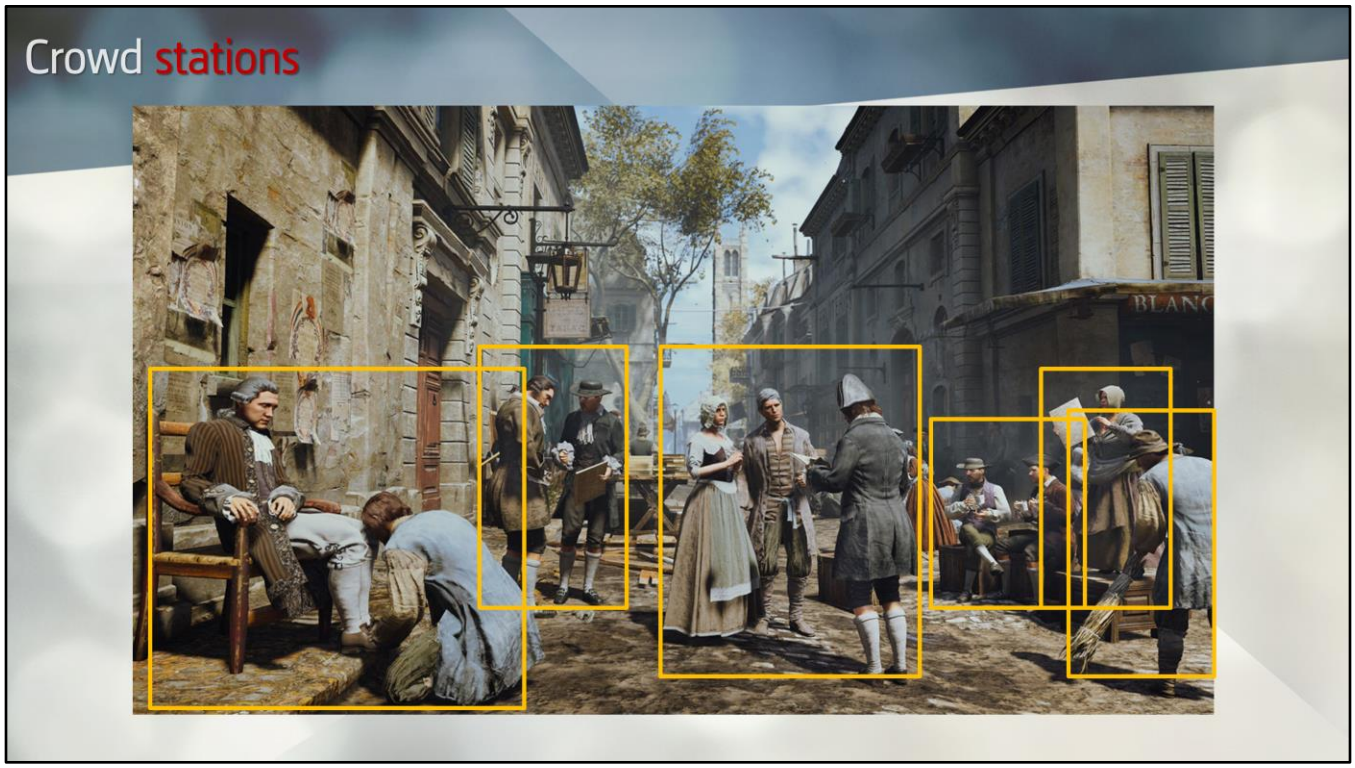
(+) Cheap in bandwidth

(+) Code and data are the same in single and multiplayer

(-) Join in progress can be tricky

(-) Reaction times can become noticeable


CROWD
STATIONS

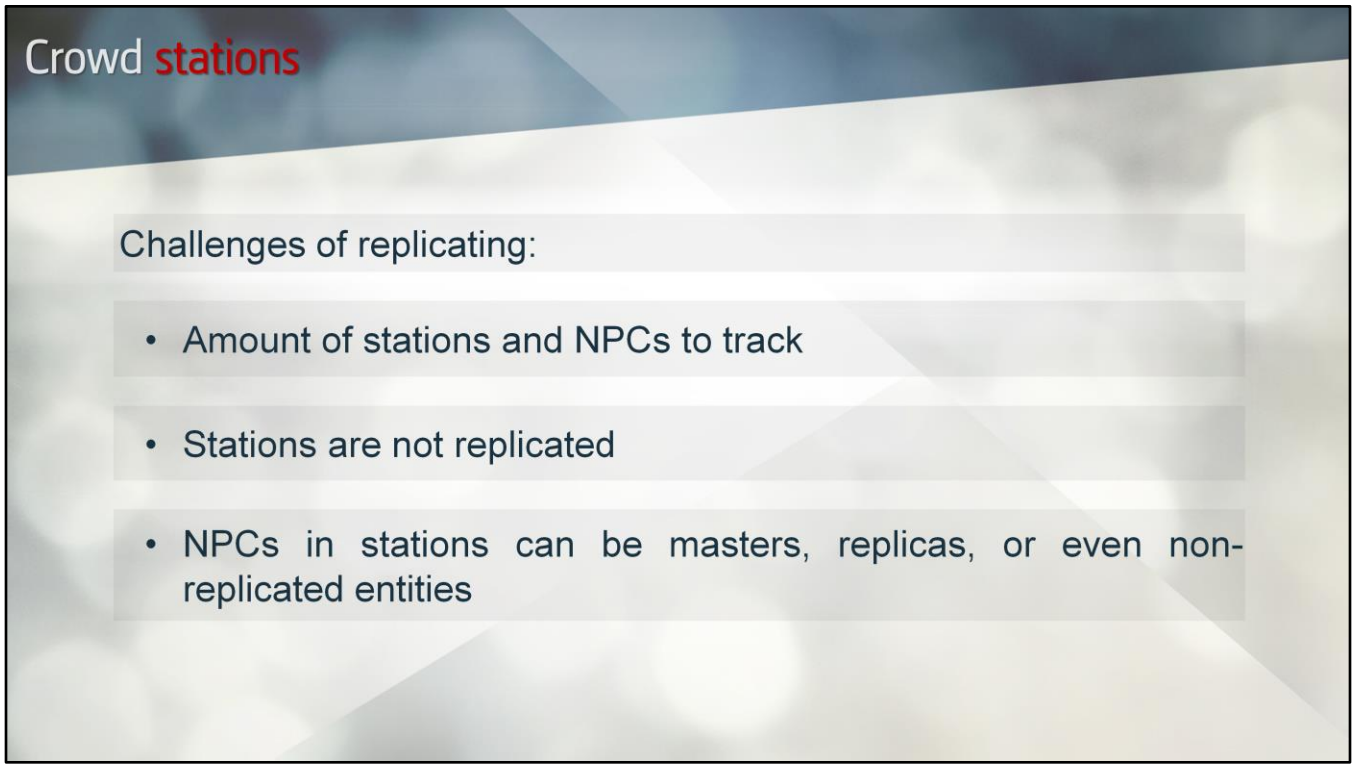


Crowd stations: NPCs that are not walkers e.g. merchant stands, artisans, groups of people talking, etc. They are not essential for gameplay.

Crowd stations

How does it work in single player :

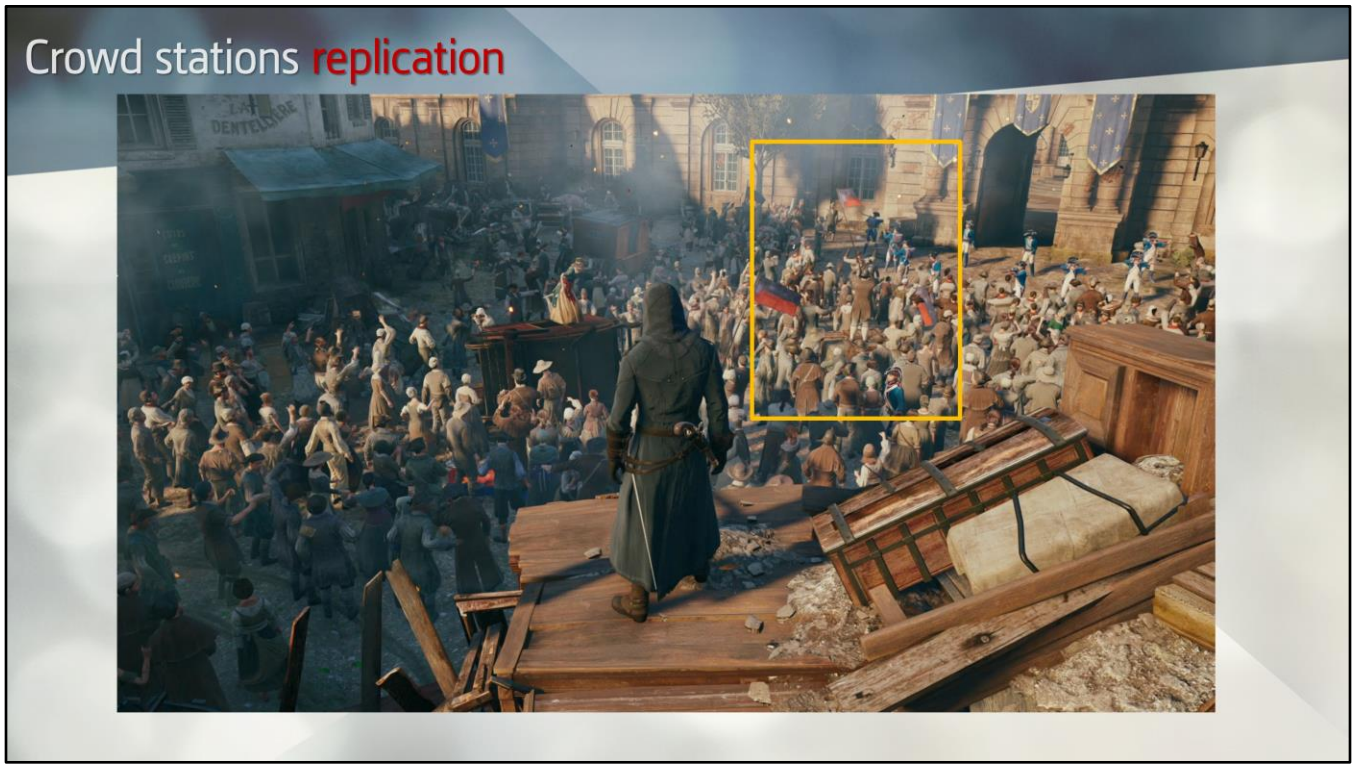
- Crowd Life Manager updates all stations
- Stations = data containers (not systemic)
- Stations loaded with cells
- NPCs spawned/released by stations



Around 900 stations loaded, and up to 600 spawned NPCs

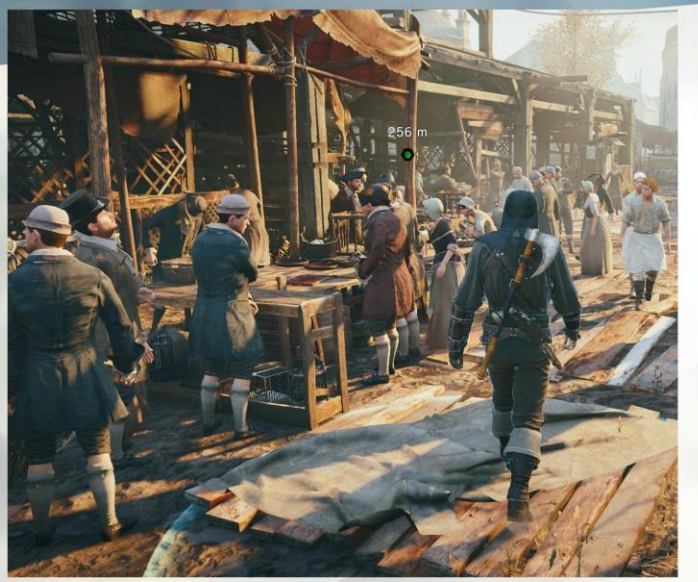
non-replicated entities: bulk crowd members.

Crowd stations replication



After a reaction, a station is deactivated, NPCs don’t belong to this station anymore.

Crowd stations replication



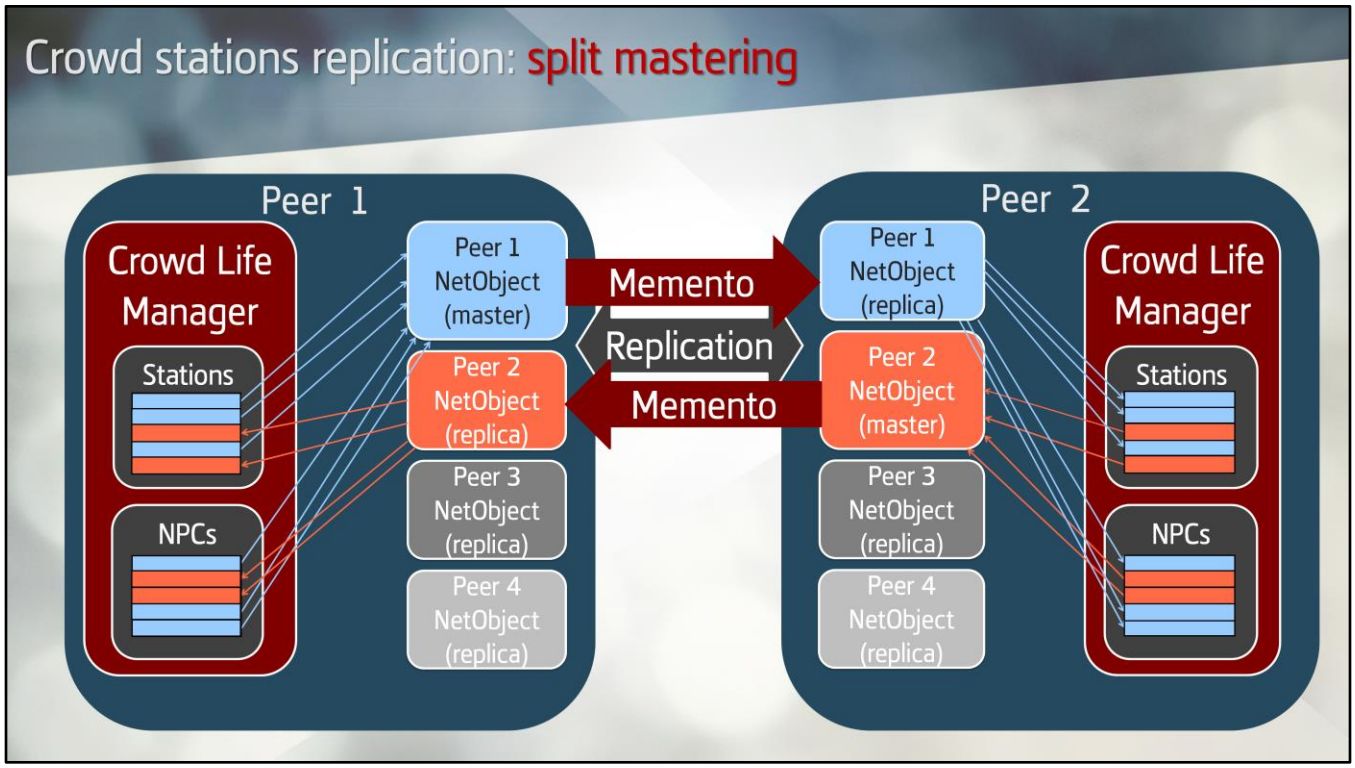
What to replicate ?

- NPCs
- Station state

Master = token winner

State: active, started, aborted, completed, number NPC spawned, released, lost

Master = token winner (token requested by the cell)



How do we replicate: Split Mastering

Each manager creates it's own NetObject

NetObjects discovered are linked to the crowd life manager

Stations:

Check who will manage each station using NetTokens

Winner of token is the Master of that station.

NPCs:

Check if the entity is a master or a replica.

For all “Local Masters”, write the state changes in the master NetObject

For all “Remote Replicas”, listen for the memento callback of remote peers and change state to follow master.

This split mastering pattern is also used for the proxies, and for the players data (inventory, customization)

Crowd stations: conclusions

(+) Only 1 net object per peer = big gains on bandwidth/CPU

(+) Each peer can run their own logics

(+) Data is already available on discovery

(-) Data is broadcast on all peers even if they don't need it

(-) Execution logic tends to be complicated


CONCLUSION

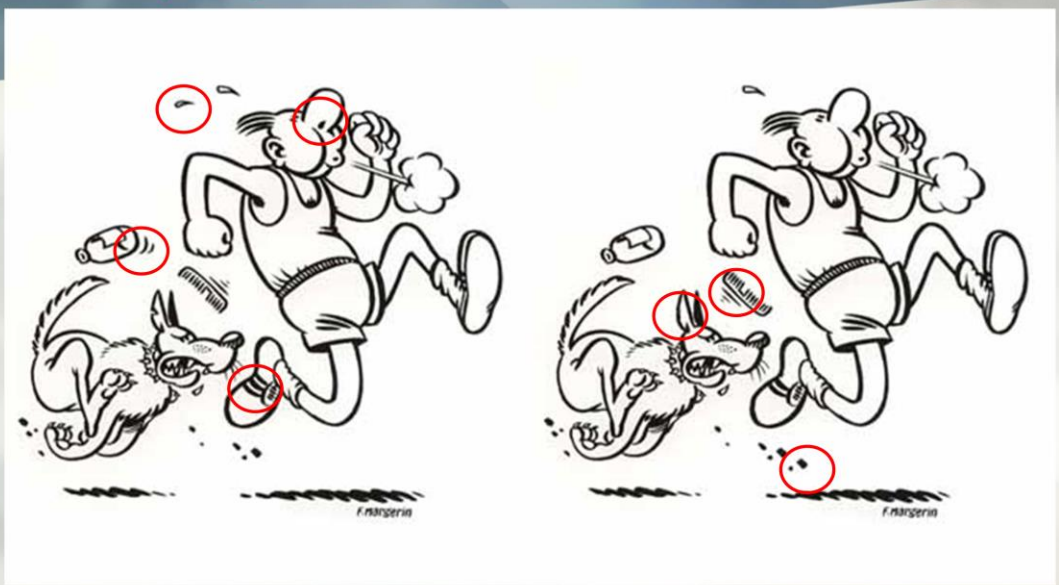
Network stats

Network stats

- Below TRC of 92 kbps per peer (around 30 kbps outside fight)
- Most expensive systems:
  - Proxies: list of vectors
  - Player, but high replication quality
  - Crowd life: lots of stations, actually cheap per station

Proxies are a saving for other systems

Replication quality



Hard to evaluate, similar to playing this game.

Gameplay replication: conclusion

(+) Cheap in bandwidth, no additional CPU cost

(+) Good replication quality

(+) Multiplayer team = gameplay team

(-) Limited to a low number of players

(-) Hard to identify sources of divergences

Need a lot of logs and a good logging tool to debug

Gameplay replication: conclusion

Existing code kept and converted to multiplayer


THANKS !
QUESTIONS ?
Special thanks: Maxime Mercier, Félix Duchesneau, Eric Bibeau, Karl Dubois, Nicolas Bonnelly-Belanger, Sebastien Scieux, Rémi Toupin-Gaudet, and all the Assassin's Creed Unity team