Worlds: A distributed MMO *(DRAFT)*

Ryan Walker
ryan.cjw@gmail.com

Abstract. A protocol defining how anyone can contribute to an unbounded universe could allow the flexibility to organically grow an MMO faster than any proprietary system. This paper overviews a protocol that enables developers to bolt their game into in common universe.

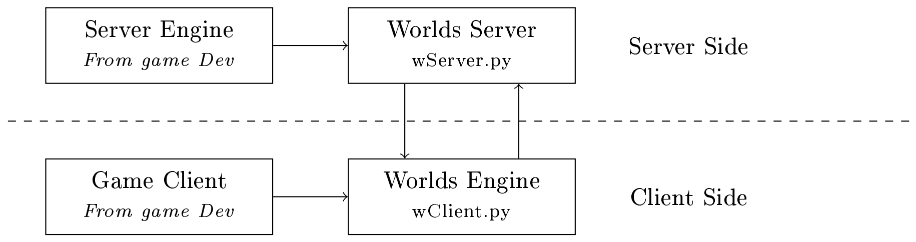
1 Worlds

1.1 Introduction

For an open software ecosystem to grow organically there should be outlets for contribution. In the context of a massively multiplayer online game the outlets become more complicated then a simple code repository. Fair game mechanics are built on a fragile ecosystem that have negative consequences if managed improperly. The fairness and security of a game are dependent on it's network forming consensus. This is trivially done with conventional methods. A server maintains a secure connection with a player who pipes actions to the server. The server then provides ground truth for the network. Distributed systems require a very different approach. Blockchains present clear deficiencies, they are too slow and the chain would get too large. An alternate method will now be presented.

1.2 Overview

A world, defined as w_k, is a node that designs a user experience and hosts players. Worlds provide the game client which the players interact with. In addition they host servers that form consensus among their world. A world can have up to four adjacent worlds determined by itself. This forms a network, which allows players to explore and share common items(5), experience(4) and currency(6) between worlds.



The worlds engine is an application that manages the elements that are common throughout all worlds. Examples being: player private keys, item ledgers(5.1) and action ledgers(**??**). The world designer has full creative freedom over the game client and server engine.

2 Actions

Actions are piped into the worlds engine from the game client, following this the engine hashes, signs and submits the actions to the worlds server. If these actions are to have a result, the worlds server engine hashes, signs and submits the actions back to the player.

2.1 Action Listing

An action listing is a data construct containing possible player actions. It's possible for a world to make their own action listing, defining an unlimited amount of possible player actions. World reactions are also reside on this list and start at address code: *0x7F*.

Fig. 1. Section of an Action Listing

```
# toolkit/ActionListing.yaml

# Actions
## Movement commands
Move:
 Code: 0x00
 N: 0x00 # Move North
 E: 0x01 # Move East
 S: 0x02 # Move South
 W: 0x03 # Move West
 Up: 0x04 # Move Up
 Down: 0x05 # Move Down
```

2.2 Action Listing Translation

Worlds might not share the exact same action listing. An action listing translation is a data construct used to translate actions from one worlds actions listing to another worlds action listing.

3 Transport

Players, defined as P_k, can enter a world one of two ways. The first being a **player genesis**(3.3) and the second being a **world transfer** (3.2). It is possible for players to transfer to whichever worlds they want. But it's may not be possible to keep their items and stats as typically only adjacent worlds have agreed on exchangeability of items. The world transfer is done using the mechanics discussed below.

3.1 Transport Hash

A transport hash is a hash of an action ledger, $h_s(AL(P_k, w_k))$, these are secured by the worlds and are used to prove a player is presenting an honest action ledger. The transport hash is formed when a player leaves the world. A **Forward Transport Hash** is a transport hash kept in a special location. This is explained more in the world transfer section, it is defined as $h_{sf}(AL(P_k, w_k))$.

3.2 World Transfer

A detailed state machine outlines how players can move about neighboring worlds. Honest worlds must follow this procedure, if they misbehave they could be risking a **World Disconnect** (7.2) from their neighboring worlds.

Eg: A player, defined as P_1, wants to move from w_1 to w_2 and then to w_3 (Figure 2), The network values in this scenario are defined as...

Fig. 2. P_1 moving from w_1 to w_2

$$\begin{tikzcd} w_1 \arrow[r, "P_1"] & w_2 \arrow[r, dash] & w_3 \end{tikzcd}$$

	w_1	w_2	w_3
$h_s(AL(P_1, w_k))$	$NULL$	$NULL$	$NULL$
$h_{sf}(AL(P_1, w_k))$	$NULL$	$NULL$	$NULL$
Neighbor	w_2	w_1 & w_3	w_2

1. P_1 sends a signed world entry packet to w_2
2. w_2 insures that w_1 is adjasent to itself
3. w_2 must verify that P_1 currently resides in w_1, this is done by ensuring $h_{sf}(AL(P_1, w_k)) = NULL$. This is found by sending a signed data request to w_1
4. P_1 presents $AL(P_1, w_1)$ to w_2
5. w_1 calculates $h_s(AL(P_1, w_1))$ using the AL on the serverside
6. w_2 calculates $h_s(AL(P_1, w_1))$ using the AL provided by P_1, the hashes must match
7. (Optional) An **Action Ledger Traceback** (7.1) can be complete
8. P_1 is now granted access to w_2 and can submit actions
9. w_1 must store $h_s(AL(P_1, w_1))$, $AL(P_1, w_1)$ can be deleted

Fig. 3. P_1 in w_2

$$\begin{tikzcd} w_1 \arrow[r, no head] & w_2,\ P_1 \arrow[r, no head] & w_3 \end{tikzcd}$$

	w_1	w_2	w_3
$h_s(AL(P_1, w_k))$	$h_s(AL(P_1, w_1))$	$NULL$	$NULL$
$h_{sf}(AL(P_1, w_k))$	$NULL$	$NULL$	$NULL$
Neighbor	w_2	w_1 & w_3	w_2

Fig. 4. P_1 in w_3

$$\begin{tikzcd} w_1 \arrow[r, no head] & w_2 \arrow[r, no head] & w_3,\ P_1 \end{tikzcd}$$

	w_1	w_2	w_3
$h_s(AL(P_1, w_k))$	$h_s(AL(P_1, w_1))$	$h_s(AL(P_1, w_2))$	$NULL$
$h_{sf}(AL(P_1, w_k))$	$h_s(AL(P_1, w_2))$	$NULL$	$NULL$
Neighbor	w_2	w_1 & w_3	w_2

3.3 Player Genesis

Player Genesis is the creation of a new player, for this to occur a player must digitally sign a genesis package with unix time. The world must ensure there are no existing values entered for that player. The player is then instated into the world with all the initial player values set to zero.

3.4 Refuge Package

A refugee package is an action ledger that dates to the time that the package is sent. This allows worlds to reinstate players without the need of a world transfer. This is useful for player that have been orphaned into a world. This typically happens during either a **world disconnect** (7.2) or **world outage** (9.3). This has the effect of splitting the player's history into two forks, which opens the possibility of an **action ledger conflicts** (9.4).

4 Experience

Experience is distributed to players using experience packages. This is simply a package that is of the form shown by figure 4. When a player is to received experiance this package is hashed, signed and sent to a player from the world. Players are expected to keep all these packages. This same technique can be used to distribute anything to players that doesn't required transferability or fungability.

Fig. 5. Experience Package

```
ExpPkg{
    PlayerPublicKey
    WorldPublicKey
    ExpAmount
    ExpType
    UnixTime
}
```

5 Items

Items can be introduced by any node on the network. To craft an item the item package, defined as i_p, must be signed by the node and submitted the worlds smart contract to prove ownership and lock the funds associated with the item. It is the worlds responsibility to judge which items are credible, this can be simply done by examining the public key in the item package. to bring an item into a world is must be signed by the *Worlds Smart Contract*, which remains

Fig. 6. Item Package i - Wood

```
ItemName = 'Wood'
ItemClass = 'Material'
ItemHash = hash(ItemName, ItemClass);
```

Fig. 7. Item Package i

```
ItemName = 'Crossbow'
ItemClass = 'TwoHandedDistance'
ItemHash = hash(ItemName, ItemClass);
```

5.1 Item Ledger

An item ledger contains the history of an item. Using an item ledger it is possible to trace the item back to it's origin. It is possible to use this to understand how many items worlds are producing, which can be used to prove worlds are not obeying economic caps. In order to own an item, a player must hold the item ledger and for the most recent hash of the item ledger to reside on the servers of the current world the player is it.

Fig. 8. Item Ledger *il*

```
ItemLgr{
    ItemPkg
    Tx
}
```

6 Currency

A common currency between worlds is required. For the game theoretical elements to work this currency must be zero sum, meaning worlds cannot generate it. Ideally it is tracked through an existing blockchain. Implementing it should be as simple as building it into the Worlds engine. Worlds can introduce mechanics for generating revenue, like entrance fees, subscriptions or exchangeability for items. No native currency has been selected at this time.

6.1 Tokens

Worlds may require a currency of their own, this offers the ability to distribute and print tokens. These tokens have exchangeability for the currency described above. Worlds are incentivised to not abuse the system to have decent exchange rates. These tokes should also be tracked on a Blockchain. It is possible that some worlds might acknowledge other worlds currency and offer rewards in game for them.

7 Engine Mechanics

The mechanics below are simply suggestions. As the engine is completely open, worlds are free to impose whatever mechanics they wish. Worlds with drastically different game mechanics will probably not be bordering, this limits gameplay but maintains fairness. Players are able to play in whatever worlds they wish - but they must start from scratch in non-adjacent clusters.

7.1 Action Ledger traceback

The system is not entirely trustless, the neighboring worlds need to trust each other. It is possible for neighboring worlds to have disagreements and still function. For instance, w_1 might introduce an item that w_2 considers too powerful. In a case like this w_1 world can just neglect it. Actions presented using that item would be considered illegal and not entered into the action ledger of the w_1. Worlds that are not entirely trusted can be audited, consider Figure 9
It is possible that w_1 might have conflicts with the rules of w_3, however w_1 did not choose be adjacent to w_2. During the worlds transfer, w_1 would not get the

Fig. 9. Three adjacent worlds

$$\begin{tikzcd} w_1 \arrow[r, dash] & w_2 \arrow[r, dash] & w_3 \end{tikzcd}$$

action ledger from w_3. Completely illegal action could have been committed in w_3 (eg. free money). In this case w_1 can request an **Action Ledger Traceback**(7.1). If this is requested, the player must provide a list of action ledgers that date back to either the player genesis or to the last entry of the world committing the audit. The world then needs to obtain $h_s AL(P_k, w_{k:n})$ from w_k to w_n. If the hashes match then the players history has been confirmed.

Below are some mechanics that can be employed to combat illegal actions.

7.2 World Disconnects

It is possible a world may issue a disconnect of an adjacent world, this means that players may no longer travel between these worlds. This is considered to be the more drastic and can leave players orphaned in the disconnected world, see **Orphaned Players** (**??**).

7.3 One Way Gates

Worlds can only control their adjacent neighbors, they have no way of controlling the neighbors of their neighbors. Consider Figure 9 If w_1 issues a one-way-gate against w_3 player are no longer allowed to travel back into w_1 if they have traveled through w_3.

7.4 World Economic Caps

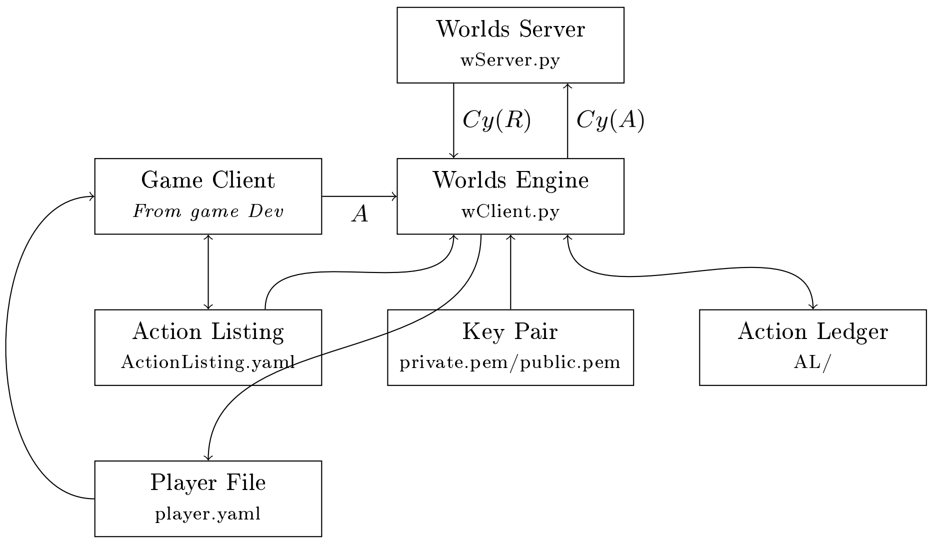
It's possible that worlds may require neighboring worlds to employ a *world economic cap.* This sets a hard cap to the amount of resources that can be distributed to players from the environment in a given time period. This can cap experience points, coins, items, etc...

7.5 Item Ledger

Worlds with greater security requirements might require an item ledger upon world transfers. An item ledger is a data construct the holds an item's history. The first entry should be the items genesis, which is signed by the world that is issuing the item. To own an item, a player must own the most recent item ledger and for the hash to be verified by the world the player current resides. For more details see section 5.1.

8 System Architecture

A: Action
$Cy(A)$: Signed Action
R: Response
$Cy(R)$: Signed Response



9 Outstanding Issues

9.1 Action Ledger Fragmentation

Action ledger fragmentation is when a significant portion of a players action ledger has become 'neglected' due to worlds with action disagreements. This should be scarce, as worlds with disagreements should consider a disconnect or changing rules.

9.2 Malicious Worlds

Traditionally players would have no recourse against malicious worlds. But it possible that after a world disconnect adjacent worlds might allow players to rejoin, in addition they have the option to neglect the malicious activity.

9.3 World Outages

In the event of a server outages players may become stranded. The first line of defense for this should be server mirrors, these do no have to be as capable as the main servers but should allow players to exit. In the even of a permanent world outage, adjacent worlds should accept signed **refugee packages**(3.4).

9.4 Action Ledger Conflicts

Figure 10 depicts a possible scenario in which w_2 disconnects from w_3 and w_5. If a player resides in $w_{3:5}$ there is now no way to return to $w_{1:2}$. $w_{1:2}$ may accept a refugee package and reinstate the player. The player now has the option to play on either forks. In the event of a future path creation between $w_{3:5}$ and $w_{1:2}$ it is possible that that player forks might collide.

Fig. 10. Action Ledger Conflict

$$\begin{tikzcd}
 & w_5 \arrow[r, dash] \arrow[d, dash, "/" marking] & w_4 \arrow[d, dash] \\
w_1 \arrow[r, dash] & w_2 \arrow[r, dash, "/" marking] & w_3
\end{tikzcd}$$

This is called an action ledger conflict. Even with an action ledger traceback being done it is possible for items or funds to be duplicated. To mitigate situations like this is should be common practice to avoid joining worlds that have player history that collides. Another possible combatant is **item ledgers** (5.1) and **exchanges rates** (**??**).