Cluster-Optimized MMO Server Protocol

Andrew Zhai
andrewz@stanford.edu

Dmitry Kislyuk
dkislyuk@stanford.edu

1. Introduction

Overwatch, League of Legends, and World of Warcraft are some of the most popular games in the world. With the growing adoption of E-Sports, development of online multiplayer games seems primed with opportunties. There however is a significant barrier to entry for independent developers as deploying these online multiplayer games usually involve the cost and maintainence of expensive centralized game servers.

In this work we explore a semi-decentralized game server protocol (COMMO) to reduce the cost of such a centralized server. This saving is done by hosting the game in a peer to peer manner. In this framework, the centralized game server reduces to a shard assignment and client service discovery system, thus requiring minimal resources. We note that a purely decentralized design introduces considerable concerns regarding coordination, and thus propose a shared-responsibility design where certain aspects of the protocol can be centralized, while pushing the majority of the communication burden to the clients.

For a game to start, the centralized game server will wait for a minimum number of players to join (which is a common "lobby" mechanic found in many games). After enough clients have joined, the centralized server assigns each client to one (or more) shards. A **shard** is a sub-region of the game world (Figure 3) and the clients managing that shard are responsible for the state of players in that region of the game world. As such COMMO relies on clients to take on dual responsibilities in both being a player that can move around the world and optionally a game server that manages event in a particular shard of the game world. There is a one to one mapping between shard and a RAFT [8] cluster of clients to ensure there is a converged game state with replication. Ideally we want each RAFT cluster to be formed with multiple clients for tolerance to protect against failure cases (e.g. a client loses connection or logs out of the game), as well as load balancing purposes, which we describe in detail in subsequent sections.

The sharding of the game world into sub-regions allow us to take advantage of proximity of players to other clients to limit communication overhead. Specifically, actions are only broadcast to agents who could be affected by the action, allowing clients to only need to communicate with the shard it's currently residing in and a few neighboring shards that are within its proximity.

In this paper, we will first describe the overall design of COMMO, followed by a description of a simple game we developed to test as an application on top of the protocol. While trivial by design, we hope the our application will illustrate how many core mechanics found in most games, such as moving and interacting with other clients, would work with the COMMO protocol. We conclude that while COMMO will add inevitable delay compared to a single-server design, making it ill-suited for games with extreme latency dependence, it will be able to scale to far more players and large game worlds, while keeping central server costs to a minimum, making it ideal for MMORPG designs. Code implementation is available at: https://github.com/dkislyuk/commo.

2. Protocol Design

In this section we describe the key components of the COMMO protocol, including our decentralized design (which asks a client to also be a lightweight game server), and exploiting the causality effects of game actions to reduce communication requirements among nodes. A high level view of our design can be seen in Figure 1.

2.1. Client roles

COMMO clients can take on one or two out of three possible roles (player, shard manager, and shard leader), communicated via the `get_shard_assignments` call (the master game server is responsible for these assignments). All clients are assumed to take the role of a *player client*, meaning they are active players in the game. Next, the game server can assign the role of a *shard manager*, which means that client will be responsible for coordinating actions within that particular shard. A shard in COMMO represents a subspace of the game world. Clients who wish to initiate actions inside of the boundaries of that subspace must coordinate with shard directly. Note that shard management assignment does not depend on the initial client location, under the assumption that over the course of the game, the client will be moving around the game space and

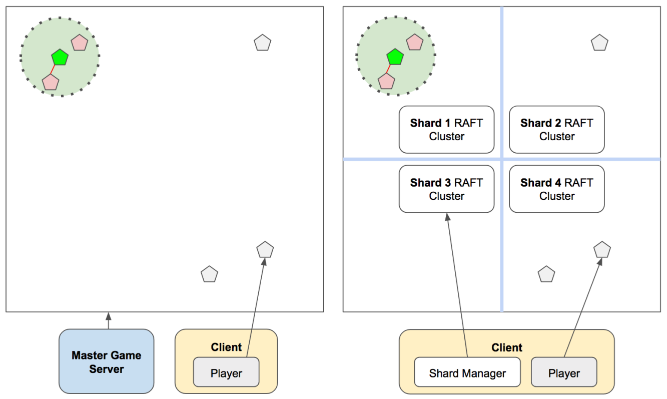

Figure 1. A simple game where the active player (green) can target other online players within its proximity. Players that can be targeted are in red and the proximity area is shown around the active player. **Left**: A single server design of a multiplayer online game where a master game server manages game mechanics and clients assume the role of players. **Right**: High level design of COMMO where the game world is divided into shards (delimited by the light blue lines) managed by clients. A client will always be a player but can optionally be assigned as a shard manager/leader who is responsible for game mechanics within the shard region of the game world. Every shard can have multiple clients managing it with consensus and replication done via RAFT. Image best viewed in color.

thus the initial shard it will join (as a player) will not be permanent. Typically the game server will assign 3 or 5 shard managers per game shard, which implies that the minimum client requirement to start the game is the respective multiple of the number of shards, as we don't want to assign a single client multiple shard manager duties. Having multiple clients managing a shard enables failure tolerance as a single client can disconnect without affecting gameplay within the shard it manages. Although in the current implementation, shard managers are assigned randomly, future work can explore assigning shard managers using a geographic proximity heuristic, since we want the consensus protocol to be run with minimal network round-trip time. Additionally, the master server should relieve of slower clients of serving responsibilities altogether, since this negatively impacts all players within that shard.

Upon learning of the shard management duties, a client (if chosen to be a manager) attempts to establish contact with other fellow managers within the assigned shard, and initiate a RAFT protocol on a dedicated port. Once a majority is established, one of the clients is elected to be a *shard leader* (the second possible role a client may be assigned). The shard leader then reports back to the master game server with the `confirm_shard_leader` call. The client then starts a local instance of the game server for the shard it represents, and assumes dual responsibilities as **both** a player and a decentralized server node. The shard game server is described in the next section.

The game is allowed to begin when the master game server has a confirmed leader for all of the shards, which implies that at least a quorum of clients have acknowledged their server responsibilities for each shard. A finalized *shard mapping* is broadcast to all clients, which contains all the information a client needs for shard discovery. Based on the initial game location of the client (depending on the game design, this may be chosen by the player or the server), an initial shard is chosen to host that client, and a connection to one of the shard managers is established. For load balancing purposes, the shard manager is chosen randomly within the shard cluster, but the connection is sticky as to not re-establish TCP connections on every action. At this point, the game will run without the master game server; clients will communicate with shards managed by other clients directly for game mechanics in a decentralized manner.

Depending on the game design, COMMO can support clients joining after the game has started, although they will only be assigned the player-client role. This can be done by having the new client learn of the shard assignments through the master game server and having the new client broadcast its existence to the shard it wishes to join. Future implementations may ask new clients to replace disconnected shard managers by using the dynamic membership properties of RAFT. There would need to be a new mechanism for updating the shard mapping each client locally stores, although this can be solved with more mature implementations of service discovery such as Zookeeper [7].

2.2. Shard cluster

The role of the COMMO shard cluster is to synchronize game state among all players whose actions currently affect the subspace controlled by the shard. Because we assume that events must be applied in a consistent order to ensure game correctness, a consensus protocol is used (provided by RAFT). This also allows us to provide a greater level of tolerance for a game shard as we can withstand client failures without affecting the availability of the game shard. A shard-game server needs to be implemented in such a way that the game state is replicated via RAFT. The replicated game state encapsulates all necessary details to get a complete view of all events occurring in the shard, so that any member of the shard cluster can process a client-player action. To monitor the health of the shard, the shard leader and the master game server periodically exchange health checks which include the total number of players connected to that shard. In a future iteration of the COMMO protocol, dynamic re-sizing of shards based on player load will be supported, as it is unrealistic to assume that player distribution will remain random throughout the game, as players converge to compete for common objectives.

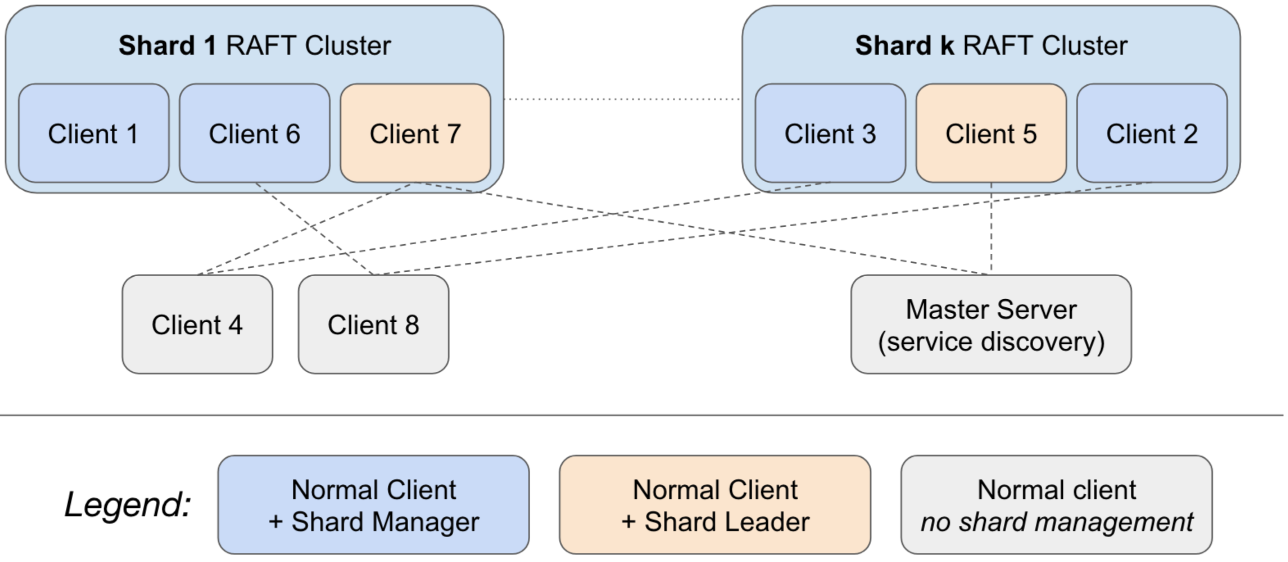
Shard 1 RAFT Cluster
Client 1
Client 6
Client 7
Shard k RAFT Cluster
Client 3
Client 5
Client 2
Client 4
Client 8
Master Server (service discovery)
Legend:
Normal Client + Shard Manager
Normal Client + Shard Leader
Normal client *no shard management*

Figure 2. Illustration of COMMO client roles. A client can be assigned shard manager responsibilities and additionally can be elected to be a shard leader. All clients are expected to also be player-clients (normal players of the game). In COMMO, the shard clusters replicate and synchronize local version of the game state, while the master game server (of which all clients know of) acts as a discovery service, and coordinating players joining and shard re-shuffling. In this example, we have multiple shards (only shard 1 and shard k are shown), each responsible for a small portion of the game world. Depending on the proximity zone of an action, a client may broadcast its actions to multiple shards (for example, client 4 and client 8 may be interacting with each other near a shard boundary), although a client belongs to only a single shard at a time, as determined by its location in the game world. Image best viewed in color.

A player-client joins a shard by contacting a member of the shard cluster via the `join_shard` call, which includes a `player_state` argument to allow the shard server to insert the player into the local game representation, and replicate this state across the shard cluster. The join request is approved only if the clients current location is within the boundaries of the shard (information known to the shard server at creation time). Note that a player-client may communicate with a shard without joining it by broadcasting an action whose proximity zone overlaps with the shard boundaries, a mechanism explained in more detail in the next section and seen in Figure 3. If a player performs a move action which places them within the boundaries of a new shard, the client will first perform a `leave_shard` call to notify the shard cluster that it is no longer reporting to it. In practice, players may experience a small delay when crossing shard boundaries (which may or may not be visible to the player, depending on the game design), due to the additional leave and join network calls, which block the client until both of the shards acknowledge the new state. This is an unfortunate drawback of using a sharded game world, but the effects of this can be mitigated through shard design which takes advantage of natural barriers within the game, which could minimize shard boundary crossings.

2.3. Proximity-based client optimization

The COMMO protocol exploits a basic idea in games featuring a large game-space: a player typically is most concerned with actions occurring nearby, and can afford to have delayed updates (or even no updates) from players far away. Games with features such as large maps (with long distances between groups of players), fog-of-war, complex non-transparent terrain, and few globally-visible events are thus well suited for COMMO.

A game utilizing the COMMO protocol should have a well-defined *proximity function*, which, given a user and an action, establishes what is the maximum distance within the game space where that action would have an effect. This causality property allows us to minimize communication overhead, which is often a concern in decentralized system design, by only notifying clients within the proximity zone of the action. Specifically, upon taking an action, a client first uses the proximity function to determine the zone within the game space where the effect could be seen, and broadcast that action to all shards that the proximity zone overlaps with. Depending on the game design, certain actions may have a null proximity zone (for example, if a player modifies some non-visible aspect of their character), in which case only the shard containing that player needs to be notified. On the other extreme, certain actions may have global consequences in the game-space,

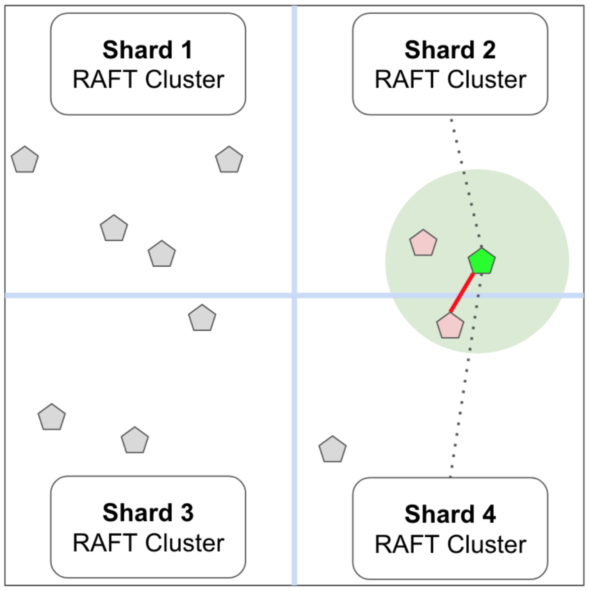

Figure 3. Simple example of in-game proximity: the player client (green) takes an action (red line) which can only affect players (red) within the proximity zone (green circle). Many real game mechanics have such a property, and this can be exploited by prioritizing communication to the clients which may be affected by the action. In this four-shard example, the player client is part of Shard 2, but because its proximity zone includes Shard 4 as well, the action is broadcasted to both shards (note that the blue grid lines indicate shard boundaries). An important component of utilizing the COMMO protocol effectively is minimizing multi-shard broadcasts during the course of the game. A more realistic game implementation would ideally have a smaller proximity zone compared to this example. Image best viewed in color.

meaning that all shards must be notified in the broadcast. COMMO is ill-suited for such requests, and instead we assume that the game design limits the players actions to mostly local events, with a small proximity zone relative to the global game space size. Additionally, an important design choice in configuring COMMO is choosing shard sizes large enough such that in most cases the effect of a user's action only need to be broadcast to a single shard. Assuming non-overlapping shards, there will be cases during normal game play where this assumption will break (for example, when actions are taken close to a shard boundary, and the player's actions must be broadcast to multiple shards simultaneously), but our goal is to minimize such scenarios.

During normal operation, as it moves and engages in the game, a client repeatedly broadcasts the `take_action` call based on the proximity function of that action, using its local shard mapping to contact look up the address information of all relevant shard clusters. In the current design of COMMO, the game is sharded in a deterministic and fixed manner, meaning that the clients can always reliably know which shard cluster to contact, although we expect this to change with the introduction of dynamic re-sharding in the future.

2.4. Failures

As a decentralized protocol, COMMO needs to have properties allowing it to tolerate client failures as part of normal operation, especially because clients are expected to be far less reliable compared to dedicated game servers. Although most multiplayer games are designed to continue even with players disconnecting, COMMO has the added complexity of handing over shard management responsibilities of the departed player. In this implementation of the COMMO protocol, we only assume fail-stop failures, however in production the protocol likely will need to support Byzantine failures. This is a requirement for future work on this protocol. It should be noted that many games have separate systems, such as mandatory client-side exploit detectors (e.g. Valve Anti-Cheat software), for dealing with bad actors that may be used on top of the existing COMMO protottol to provide better safety. Due to the random assignment of shard-management responsibilities, cryptographic signing of messages would be able to add a substantial amount of security against rogue bad actors in the system, but this line of work requires more investigation.

In the event that a client is disconnected which had shard leader responsibilities, the rest of the RAFT quorum (i.e. the remaining shard managers for the respective shard) would begin a new leader election process. A newly elected leader then notifies the master game server using the `new_shard_leader` call. The master game server will then update the global shard mapping, so any new clients joining the game will know which shard managers are currently available. As existing connections to the failed shard manager begin to time out, clients playing in the current shard will randomly pick a new shard manager to redistribute their load. The protocol will fail if the majority of clients disconnect in the same shard cluster (since event replication will no longer achieve consensus), but we expect this to be a rare event, mitigated partially by the fact that shard managements assignments are random.

Further, COMMO can tolerate a disconnection of the master game server. Since this server acts as our service discovery mechanism, in this state no new clients can join the game, but the game can still continue with existing players in a fully decentralized manner.

3. Implementation

In this section, we describe some technical implementation details along with a simple game we developed as a test application on top of the COMMO protocol.

We use Thrift [6] to implement the communication serialization and service design, and make available a Python implementation of COMMO, along with a simple game implemented in PyGame [4] used to test the validity of the protocol. We make use of the `PySyncObj` library [5] to implement the RAFT protocol, which implements three primary functions for COMMO shard clusters:

1. **Leader election**. Although the master game server could assign the leader and handle re-elections (if the assigned leader fails), we want our design to have resilience and operational stability during a fully decentralized mode if the master fails. Therefore we rely on RAFT's timeout-based leader election to keep the shard cluster organized.

2. **Action log replication**. `PySyncObj`[5] provides a useful abstraction of a synchronized event log which can be used to order and replicate all client events within the shard to all the shard managers. This is a critical component for ensuring game correctness, especially since two clients interacting with each other in the game might be doing so through two separate shard managers.

3. **Dynamic membership changes**. The original RAFT protocol provides a mechanism for modifying the members forming the shard cluster. Although the current version of COMMO does not support adding new shard managers after a game has started, this is a future direction of this work.

3.1. Simple Game

We describe our simple game (Gotta Catch 'em All) in Figure 4. Within the game, we have implemented three mechanics that are the core of a majority of online multiplayer games:

1. **Move**. A player should be able to move around the game world, including across shard boundaries.

2. **Attack**. A player should be able to attack a player within their proximity, which allows us to test our proximity function implementation and verify that our actions are broadcast to a limited number of shards (ideally just one).

3. **Heal**. A player should be able to heal a player within their proximity, by the same mechanic.

We have implemented this game through both a baseline centralized single server design and through our COMMO semi-decentralized design. We describe both implementations below.

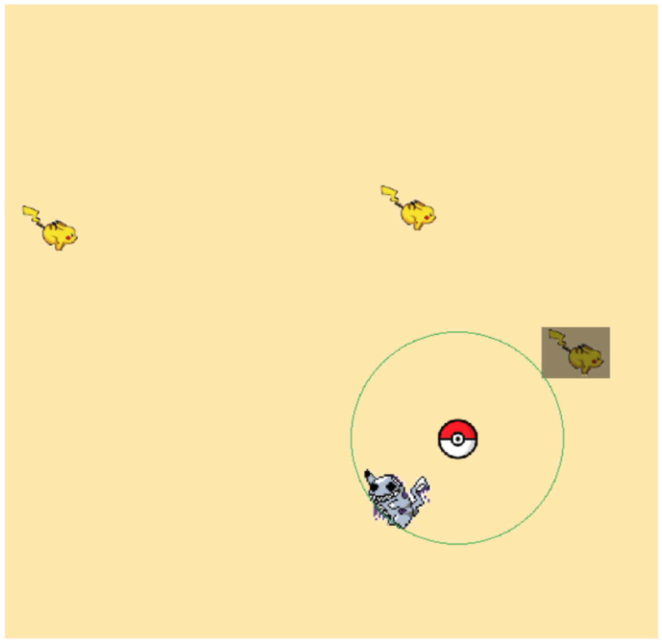

Figure 4. **Gotta Catch 'em All (Pikachu version)**: Players control a move-able Pokeball [3] in this game. The goal is to find all the Pikachus [2] in the game world and attack (catch) them. Attack is done by clicking on a Pikachu within the proximity delimited by the green circle. A Pikachu's health is shown through the darkness of the sprite and a caught Pikachu is shown through a dead Pikachu sprite [1]. Players can also heal a Pikachu by right clicking on the target. This work is done for research purposes only. A Demo Video of this game can be seen `https://www.youtube.com/watch?v=AwKvfg5hXfg&t=373s`

3.2. Centralized Single Server Implementation

We developed this single server implementation to serve as a baseline for our decentralized design, allowing us to decouple the game mechanics implementation and the decentralized protocol implementation.

A game starts out with clients asking the centralized server to `join_game(PlayerType type)`, returning the player id corresponding to the client. For our demo, we have developed two `PlayerTypes`. A *Random* agent represented by a Pikachu who randomly move to destinations in the game world and a *Player* agent represented by a Pokeball who can be controlled through a keyboard to move, attack, or heal.

After receiving the `player id`, the clients will continuously ping the server with a `start_game()` message to ask for when it is valid to start the game. The centralized server will eventually return a game start status when enough players have joined the game along with the initial `PlayerState(location, health, type)` of the player.

During normal game play, clients will send `take_action(player_id, action)` requests to the master server where an action can be moving, attacking, and healing. The server will respond with a success or failed status for whether or not the action was committed. One reason for a failed status is if an attack or heal action failed the proximity verification done on the server to ensure that the attack and heal targets are valid. In this implementation, the master server is becomes the bottleneck for all game logic.

3.3. COMMO Implementation

The COMMO implementation of our game implements all the core roles described in Section 2, along with the proximity zone functionality for efficient action broadcasts. We tested the game with up to 8 shards, with shard cluster size of 3, meaning that in addition to 1 master game server, there were 24 shard servers, which would be capable of supporting hundreds of clients. The COMMO implementation made available can successfully disconnect the master game server and still operate in a decentralized manner. We refer the reader to our code implementation for more details. We selected a RAFT timeout range (for random timeout selection) of 0.2 to 0.6 seconds, with a leader timeout period of 10 seconds. During replication, our RAFT implementation uses 65kb send and receive buffers, although we this can be optimized based on the maximum size of action updates for the specific game implementation.

4. Conclusion

COMMO is a semi-decentralized game server protocol which relies on a game-world sharding scheme to assign fixed areas of responsibilities to clients, which in turn assume dual roles of being both a player and a shard server. To limit the load on the clients, the protocol relies on limited inter-shard communication by exploiting the proximity zones of actions in the game world. We believe this sort of decentralized design can help certain game designs scale to hundreds or thousands of clients, without requiring large server resources. By limiting the role of the master server to service discovery and player registration, we push the majority of the game logic to local shard clusters. Future work can focus on more dynamic clustering (such as live re-sharding with no downtime) and security concerns, particularly bad actors who exploit their given shard management responsibilities.

References

[1] Pikachu (faint) sprite. https://mibevan.deviantart.com/. 5

[2] Pikachu sprite. https://scratch.mit.edu/projects/11545946/. 5

[3] Pokeball sprite. https://www.redbubble.com/shop/pokeball+sprite+stickers. 5

[4] pygame. https://github.com/pygame/pygame. 5

[5] PySyncObj. https://github.com/bakwc/PySyncObj. 5

[6] A. Agarwal, M. Slee, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook, 4 2007. 5

[7] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. 2

[8] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. 1

A. COMMO Base Protocol Specification

Note that the same `CommoServer` interface is used by both the master game server and the shard servers.

```
service CommoServer {
  i32 join_game(1: PlayerType type)
  ShardLeaderAssignmentResponse
     get_shard_assignments()
  void confirm_shard_leader(
     1: i32 player_id,
     2: i32 shard_id)
  StartGameResponse start_game()
  JoinShardResponse join_shard(
     1: i32 player_id,
     2: i32 shard_id,
     3: PlayerState player_state)
  ActionResponse take_action(
     1: i32 player_id,
     2: Action action)
  void new_shard_leader(
     1: i32 player_id,
     2: i32 shard_id)
  LeaveShardResponse leave_shard(
     1: i32 player_id,
     2: i32 shard_id),
}


struct StartGameResponse {
    1: GameStatus status,
     2: map<i32, list<ServerPort>> shard_mapping
}
```