In this article, we propose an asynchronous messaging solution using callbacks. The basic problem we want to solve is to achieve atomic transactions among shards (starting from two shards). We will first illustrate an atomic transaction problem via train-and-hotel problem. Then we will present our solution with a new EVM instruction CALL_REMOTE. Finally, we will summarize our further concerns and comments.

Hotel-And-Train Problem

Here is the definition of the problem from Ethereum wiki “Suppose that a user wants to purchase a train ticket and reserve a hotel, and wants to make sure that the operation is atomic - either both reservations succeed or neither do.”

Before we move further, let us consider the following code to illustrate the problem:

contract Hotel { mapping(uint => Room) rooms;

```
struct Room { // Struct
    uint price;
    address guest;
    // Other fields such as date, room no, etc
}

function Reserve(uint id) payable public {
    Room memory room = rooms[id];
    require(room.price != 0);
    require(room.guest == address(0));
    require(msg.value == room.price);
    rooms[id].guest = msg.sender;
}
```

// Other code ... }

```
contract Train {
    mapping(uint => Ticket) tickets;

    struct Ticket { // Struct
        uint price;
        address buyer;
        // Other fields such as seat no., route no., etc
    }

    function Buy(uint id) payable public {
        Ticket memory ticket = tickets[id];
        require(ticket.price != 0);
        require(ticket.buyer == address(0));
        require(msg.value == ticket.price);
        tickets[id].buyer = msg.sender;
    }

    // Other code ...
}
```

where both contracts have a list of tickets/rooms and a user could call Hotel.Reserve() or Train.Buy() methods to reserve a room or buy a train ticket, respectively.

If both contracts are on the same shard, the following simple contract (pseudo-code) could address the problem easily:

contract TravelAgent { Hotel hotel; Train train;

```
function PlaceOrder(uint roomId, uint roomPrice, uint ticketId, uint ticketPrice) payable public {
    require(msg.value == ticketPrice + roomPrice);  // safeMath needed here
    require(hotel.call.value(roomPrice).Reserve(roomId));
    require(train.call.value(ticketPrice).Buy(ticketId));
}

}
```

However, achieving the atomicity of such transaction in multiple shards is still a challenging problem in blockchain sharding.

Asynchronous Message with Callback

Now we introduce a new EVM code CALL_REMOTE, which is similar to CALL with additional parameters such as

- remote_shard_id: The id to locate the remote shard
- callback_initial_data: Initial data in callback message data
- callback_max_return_data_size: Maximum number of user data can be transferred from remote shard to local shard. If the return data outputted by the contract is greater than the value, then it will be truncated. The truncated data will be

appended to callback message data

In our implement, after calling the CALL_REMOTE, the local shard will generate a message to the remote shard, and append to the virtual FIFO message queue of the remote shard. By producing a new block, remote shard will process the messages one-by-one from the queue until the cross-shard gas limit of the block is reached and leaves the position info of the first unprocessed message of the queue to the block header. The next block will resume the processing of the rest messages with the position info - this makes sure that all messages from other shards will be processed eventually. to receives the messages in a light-way, the node running the remote shard could be the light-client of the source shard.

With CALL_REMOTE, we could create the following contract to address hotel-and-train problem:

contract TravelAgent { Hotel hotel; Train train;

```
function PlaceOrderStart(uint roomId, uint roomPrice, uint ticketId, uint ticketPrice) payable public {
   require(msg.value == ticketPrice + roomPrice);  // safeMath needed here
   require(hotel.call.value(roomPrice).Reserve(roomId));
   train.call_remote.at(shard_id).value(ticketPrice).callback(PlaceOrderDone(roomId)).Buy(ticketId);
}

function PlaceOrderDone(callback_status cstatus, uint roomId) {
   require(msg.is_callback)
   if (cstatus == failure) {
      // Rollback the transaction
      hotel.cancel(roomId);
   }
}
```

where train.call_remote() will create a message to the remote shard and call Buy(ticketId) of the contract in the remote shard. After the remote shard processes the message, it will automatically generate a callback message back to source shard via virtual message queue. When the callback message returns and is being processed, it calls PlaceOrderDone(roomId) which is encoded in callback_initial_data. In addition, remote contract call status - success or failure can be retrieved in a callback_status variable. Note that if the remote call is failed, to roll back the transaction, hotel contract should support the following Cancel() method:

// Code in contract Hotel function Cancel(uint id) public { Room memory room = rooms[id]; require(room.price != 0); require(room.guest == msg.sender) rooms[id].guest = address(0); msg.sender.transfer(room.price); }

Await/Async Syntax

The callback model may be further simplified by using await/async syntax, which is a syntax sugar that wraps the underlying handling of the callback by compliers:

contract TravelAgent { Hotel hotel; Train train;
async function PlaceOrder(uint roomId, uint roomPrice, uint ticketId, uint ticketPrice) payable public { require(msg.value == ticketPrice + roomPrice); // safeMath needed here hotel.call.Reserve(roomId).value(roomPrice); result = await train.call.Buy(ticketId).value(ticketPrice); // This runs in another message, global variables may change if (result == failure) { // Rollback the transaction hotel.cancel(roomId) } } }

The await command will save all local memory variable to contract storage, and retrieve back and set to the local memory variables when the callback returns (roomId in our example). With the aforementioned code, we could guarantee that once the transaction is finished (in the sense that callback is completed), the hotel and the room are either booked or leave unbooked.

Other Concerns:

- Message Delivery Time.

In our design, we guarantee that all messages (remote call and callback) will be delivered eventually, however, we cannot guarantee when they will be delivered. This may leave a state that a room is booked for a while, while the message of buying the ticket is not yet processed if the remote shard is congested. One way to address the problem is add the expiration when booking the room. If the room is booked, nobody could reserve the book until the reservation is expired. After expiration, the callback will fail to complete the atomicity, but we may allow user canceling the remote call with high gasprice at remote shard. The callback message will still be issued, which treats the remote contract call is reverted.

- Insufficient Gas When Running Callback Message.

Another tricky case is that when running the callback code, the callback message may have insufficient gas, which means that the callback may never be executed. A solution is to allow a user (or any user) to submit a callback resume transaction with extra gas to fuel the callback message. Extra care must be taken such as only callback message out of gas can be fueled.

- Restriction on Calling Callback Functions By Users.

Note that a callback function may be accessible by other users by directly calling the function. A simple fix is to add require(msg.is_callback) to ensure that such function will only be called in a callback context.

Further concerns? Any questions are welcome!