*Rise of Kings* Networking Architecture

*Rise of Kings* operates on a peer-to-peer architecture using lockstep to emulate synchronized, identical simulations across all clients. It is not a “true” peer-to-peer style, because one of the clients acts as a “host,” broadcasting data from one client to the rest, including the original sender. The following diagram illustrates this concept:

CLIENT (HOST)

message

CLIENT

CLIENT

CLIENT

This style allows for minimization of bandwidth necessities, allowing peers (all clients excluding the host client) to only communicate with the host, rather than sending all messages to all peers (which would create an O(n2) bandwidth requirement). Only the host bears this responsibility, hence the host should have the best bandwidth out of the clients.

# Lockstep

As stated, *Rise of Kings*, relies on the [lockstep networking protocol](https://en.wikipedia.org/wiki/Lockstep_protocol) to minimize bandwidth requirements and allow for massive game states across clients.

This is done via a “turn-based” system. Each turn consists of several steps: [command processing](#_Command_Processing), sending out “done” messages, command execution, and network latency calculation. Let’s analyze these steps in detail.

## Command Processing

This step of the inter-client communication actually doesn’t occur every turn, but rather occurs continuously as commands are issued. Commands are various user-input actions such as “user right-clicked the mouse at location (x, y).” Basically, the entire event simulation is handled through the network, rather than locally. The only local simulation is drawing of selection boxes and handling mouse movement events, because that would involve too much bandwidth and is unnecessary information for client-side simulation, since only the one executing the selection needs to see the boxes.

Commands are JavaScript objects, serialized into JSON objects by Peer.js and sent across the wire. They are not immediately executed, but are rather marked for future execution on turn *x+n*, where *n* can vary based on current game latency (usually 2-5). Hence, command execution structure is as follows:

|  |  |  |
| --- | --- | --- |
| **Turn 1000** | Command Queue  **CLIENT**  *Nothing is done during this turn. In between this turn and the next, though, a command occurs.* | Command Queue  **CLIENT (HOST)**  *Between this turn and the next, we’ve received a command from the client and immediately echoed it back to him.* |
| **Turn 1001** | Command Queue   * **Turn 1002**: Right click   **CLIENT** | Command Queue   * **Turn 1002**: Right click   **CLIENT (HOST)** |
| **Turn 1002** | Command Queue   * **Turn 1002**: Right click   **CLIENT**  *The command is executed.* | Command Queue   * **Turn 1002**: Right click   **CLIENT (HOST)**  *The command is executed.* |

## “Done” Messages

At the end of every turn, clients will send out “done” messages. These indicate that the turn has ended on the local machine and the lockstep process is ready to proceed. The host immediately echoes these messages to the other clients.

### Problems

Assume latency is 250ms, meaning any sent data will take 125ms to reach the destination, and an echo will take an additional 125ms. Turn length is set to a static 200ms. Hence the following interaction between clients occurs:

**Turn 1** (first iteration) – 0ms:

* Peer sends command for execution in **turn 3**.
* Peer sends “Turn Complete” message.
* Host sends “Turn Complete” message.

**Turn 1** (second iteration) – 200ms**:**

* Neither client has received enough “Turn Complete” messages, so we have to wait another iteration.
* Peer sends an additional “Turn Complete” message.
* Host sends an additional “Turn Complete” message.

**Turn 2** (first iteration) – 400ms:

* Prior to the turn processing iteration, both host and peer got the other’s respective “complete” packet for **turn 1**, increasing the tick.
* Peer sends “Turn Complete” message.
* Host sends “Turn Complete” message.

**Turn 2** (second iteration) – 600ms:

* Neither client has received enough “Turn Complete” messages, so we have to wait another iteration.
* Peer sends an additional “Turn Complete” message.
* Host sends an additional “Turn Complete” message.

**Turn 3** – 800ms:

* Prior to the turn processing iteration, both host and peer got the other’s respective “complete” packet for **turn 2**, increasing the tick.
* Both clients execute the command from **turn 1**.
* Peer sends “Turn Complete” message.
* Host sends “Turn Complete” message.

### Proposed Solution

Assume that latency is the same as in the situation above, but now we have a dynamic speed control system that regulates turn length based on calculated latency between clients. As a result, the [dynamic turn speed controller](#_Speed_Control) sets turn speed to 300ms (250ms latency + 50ms safety delay), to allow for all commands to be received by all clients before the turn ends.

**Turn 1** – 0ms:

* Peer sends command for execution in **turn 3**.
* Peer sends “Turn Complete” message.
* Host sends “Turn Complete” message.

**Turn 2** – 300ms**:**

* Prior to the turn processing iteration, both host and peer got the other’s respective “complete” packet, increasing the tick.
* Peer sends “Turn Complete” message.
* Host sends “Turn Complete” message.
* Peer to host latency during the last turn was calculated to be 150ms, so turn length is adjusted to 200ms.

**Turn 3** – 500ms:

* Prior to the turn processing iteration, both host and peer got the other’s respective “complete” packet, increasing the tick.
* Peer sends “Turn Complete” message.
* Host sends “Turn Complete” message.
* Both clients execute the command from **turn 1**.

As you can see, this prevents duplicate iterations of turns when waiting for completion messages. Additionally, it allows for the command to be executed within 500ms of initiation (and this is assuming a pretty bad initial latency of 300ms), rather than within 800ms as outlined in the initial system.

**Side Note:***An important transition from both of these models from the original model is that tick increases are performed immediately when enough “Turn Complete” messages are received, rather than waiting until the next tick for iterative processing.*

## Command Execution

JavaScript allows *Rise of Kings* to easily decouple the rendering, game loop, and network loop. Rendering occurs at ~60 fps, varying with window behavior, using requestAnimationFrame. The game loop occurs at a precise 60 fps, using setTimeout. The network loop occurs at a variable interval based on latency (described in a [later section](#_Speed_Control)) using setInterval. The network loop handles iteration of the current “turn,” whereas actual sending and receiving is done immediately when events fire.

This decoupled system means that there may not be any commands in any given processing frame. Command processing occurs in the game loop. All commands for the current execution turn are processed, then stored in an internal archive (for replays, and syncing additional players if need be), and deleted from the command queue.

send(data)

onRecv(data)

**Game Loop**

Command Processing

Process message and increase turn count if need be

Event Handling

**Network Loop**

Calculate latency and adjust turn tick rate

## Speed Control

Pings are calculated via round-trip timestamp differentiation. A peer will send commands to the host in a JSON-serialized message. This object contains a timestamp field that contains the peer’s current time with 1/10th of a millisecond accuracy (on most systems [using window.performance.now()]. When the host receives the message it will immediately broadcast to all clients. Once the sender receives the confirmation of their message, they compare the timestamp to current time, hence calculating round-trip time.

This round-trip time is attached to every subsequent message and recalculated on every message. The host receives these times from all peers. It calculates the maximum ping in the current game session, and sends a message indicating

max ping + threshold = new turn tick length

should be used by clients henceforth to increase the turn and process commands.

# Differentiation Between Host and Peer

There are only a few special actions that the host performs. The host is in charge of relaying / echoing received messages to all clients, in order to minimize bandwidth usage and network load for the other clients. Additionally, the host periodically calculates maximum client latency and sends updates to clients in order to dynamically update the turn tick delay. Finally, the host will also request periodic game state hashes (every couple seconds) in order to validate synchronization between clients.