*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.

There is an issue with turn execution due to latency between clients. The following diagram will demonstrate this issue:

Assume latency is 200ms, meaning any sent data will take 100ms to reach the destination, and an echo will take an additional 100ms. As a result, the dynamic turn speed controller sets turn speed to 250ms, to allow for all commands to be received before the turn ends.

**Turn 1** (first iteration) – 0ms:

* Peer sends command for execution in turn 1003.
* Peer sends “Turn Complete” message.
* Host sends “Turn Complete” message.

**Turn 2** (second iteration) – 250ms**:**

* 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.

**Turn 3** – 500ms:

Host got peer’s “complete,” now ready to tick.

* Peer got host’s “complete,” now ready to tick.