Lockstep

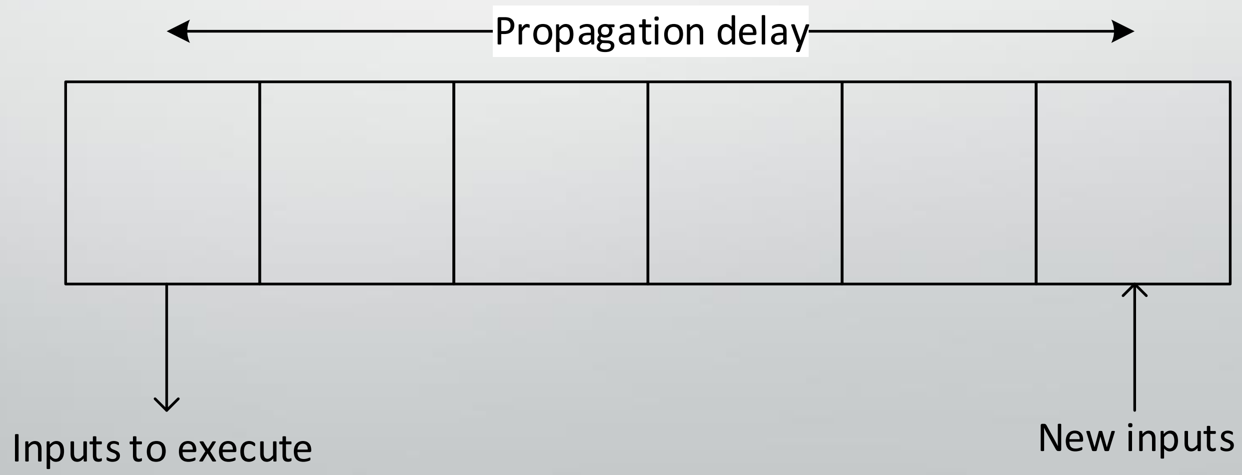
Study and implementation of a network library for distributed deterministic simulations

Raffaele Zippo, Enrico Meloni

- Inputs for the simulation are generated by clients
- Each client computes the new state of the simulation by applying inputs
- All the clients need a coherent view of the state
- Case study: Multiplayer RTS

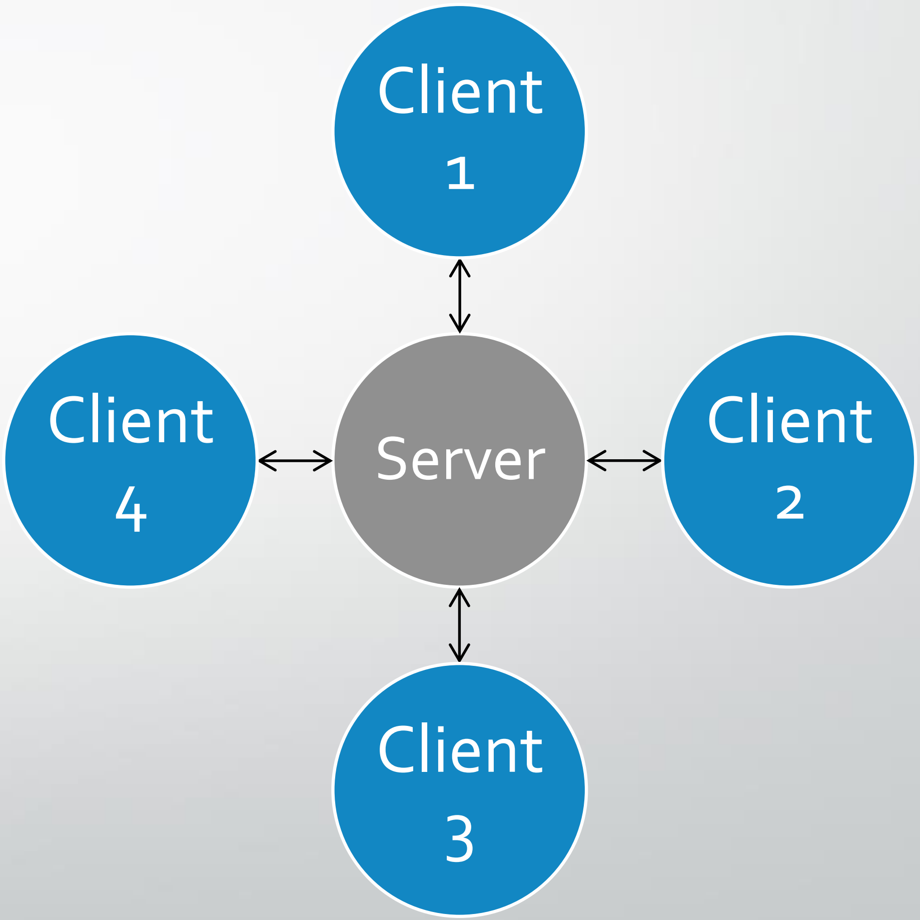
The FrameQueue

- Time is divided in chunks called *frames*. Each input is assigned to a frame
- All clients execute the same input during the same frame
- By scheduling new inputs for a later frame, we can introduce enough delay to propagate it to all clients and keep executing without interruptions
- The delay introduced should be dimensioned for the application requirements and network characteristics



Network architecture

- Client-Server architecture
  - Each clients sends its inputs to the server
  - The server forwards to each client others' inputs
- Better performance and scalability
  - Assuming better client-server connection
  - P2P would easily saturate client's upload and increase delays

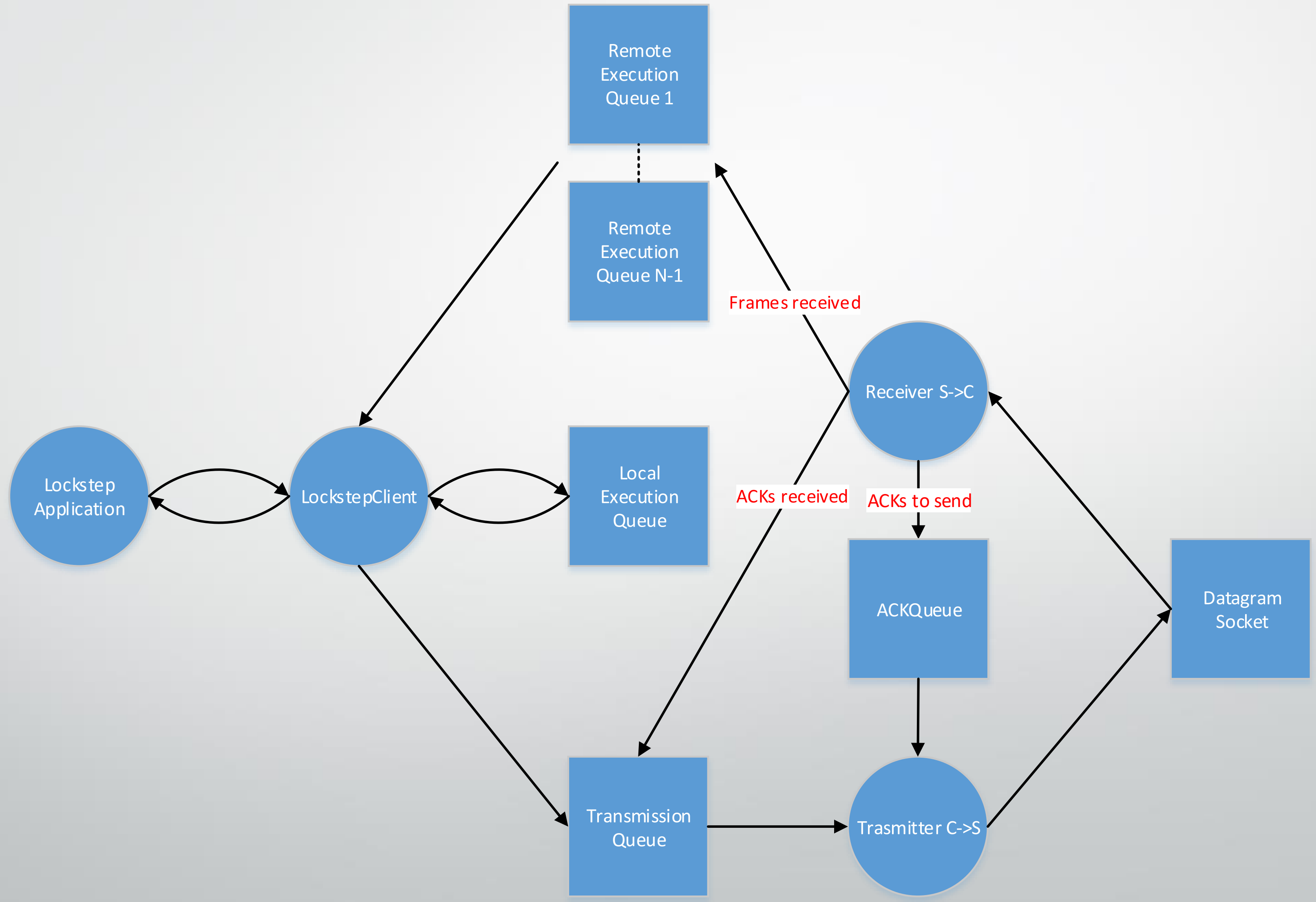


Network protocol

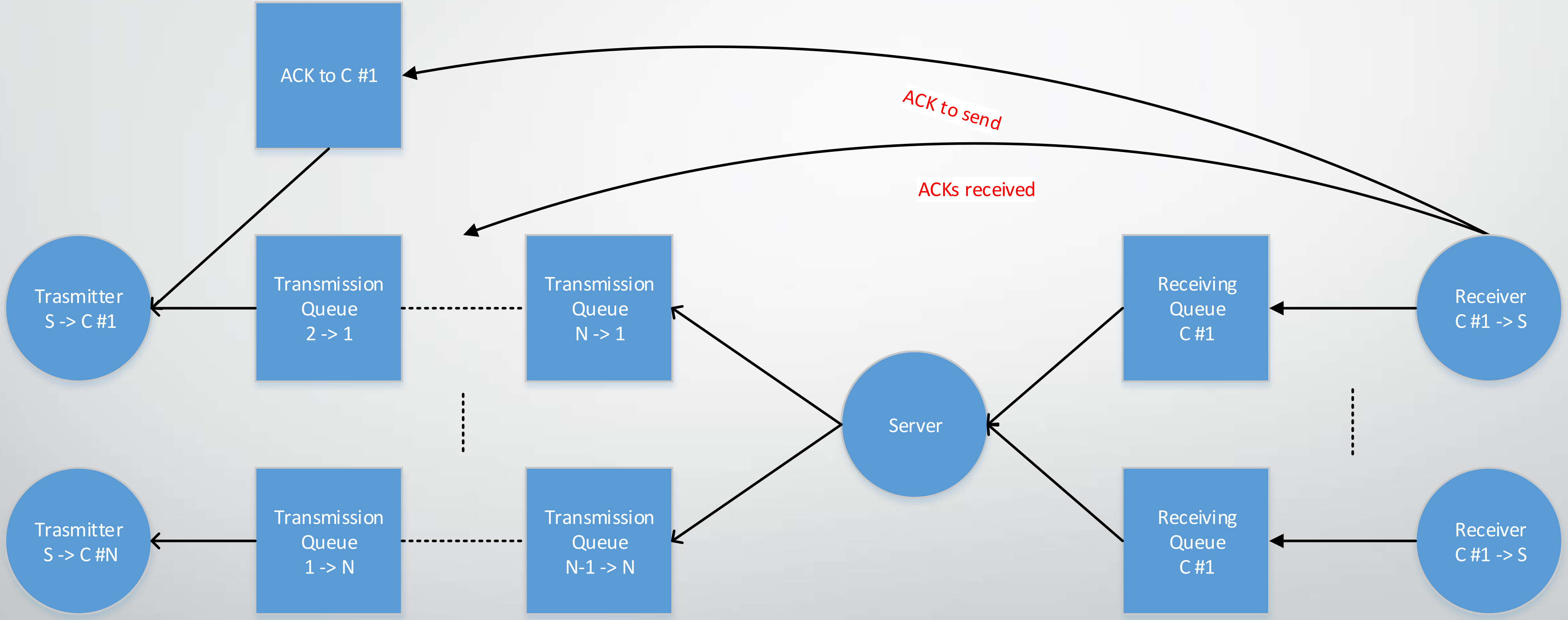
- Custom protocol based on UDP
  - Reliability
  - Minimum delay
- LockstepCommand: interface implementing Serializable
  - Implemented by the application command
  - Using Java serialization leads to high bandwidth usage
  - Minor solution by adding GZIP compression of outgoing packets

Software demonstration

Client Structure



Server structure



Client-side Synchronization (1)

- N clients C_i, each one with N execution queues E_j
- E_j stores every received input generated by C_j
- C_i and E_j keep track of the next frame number, let's say k
- C_i needs command k from every E_j to execute frame k
- If some command is missing in its queue, client stops and wait

Client-side Synchronization (2)

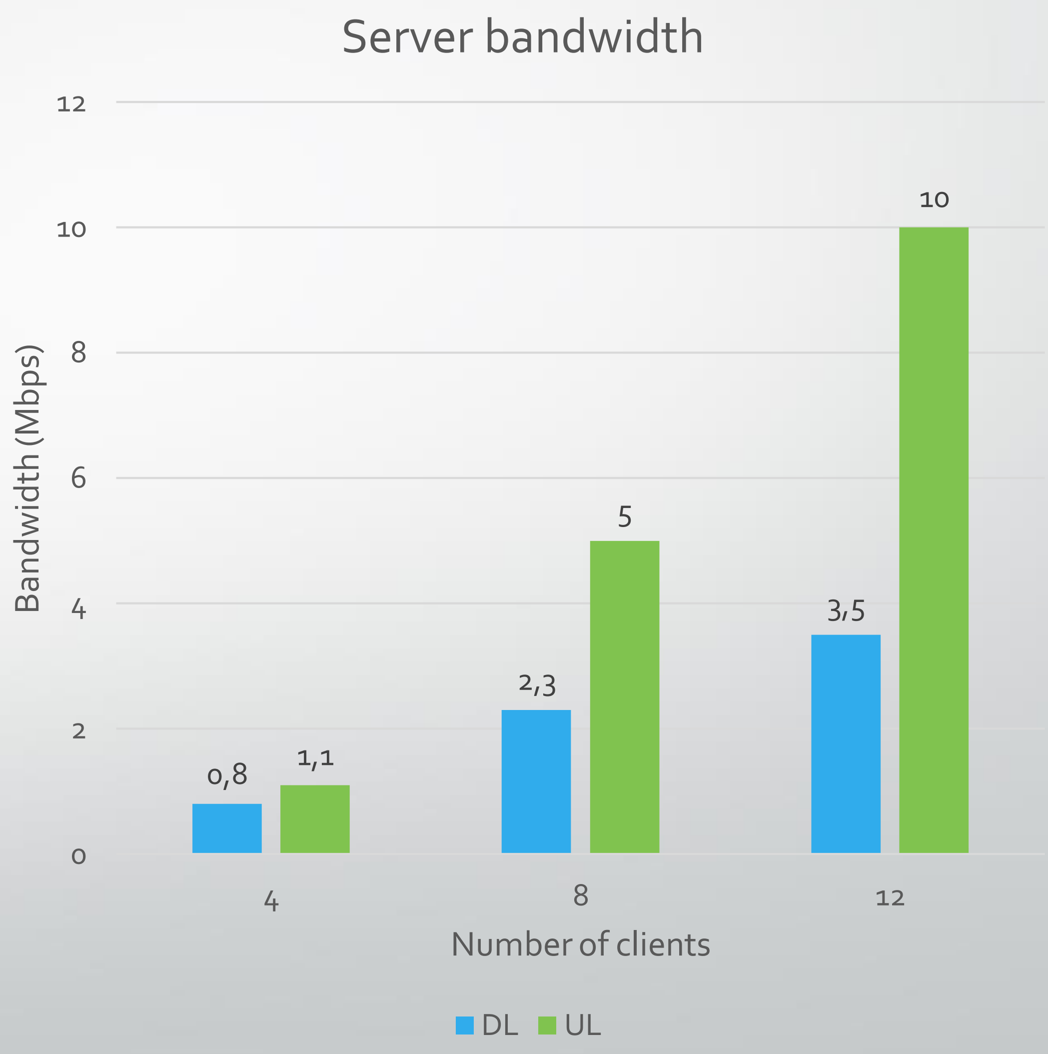
- Synchronization is achieved with a Semaphore
- E_j releases a permit when it has the next command k
- C_i tries to acquire N permits from semaphore
  - Failure to acquire: signal the application to suspend simulation, wait for all permits
  - Permits available: continue execution, resume simulation signal if suspended

Server-side Synchronization

- N execution queues E_j for storing input received by C_j
- Semaphore is used
- E_j releases a permit as soon as it receives a command
- Server waits on semaphore
- When input is available, server forwards it to transmitters
- Clients waits for in-order command, server waits for any command

Scalability

- The protocol has low bandwidth needs for clients
  - Receives $O(N)$ messages
  - Sends $O(1)$ messages
- Server instead has high bandwidth needs
  - Receives $O(N)$ messages
  - Sends $O(N^2)$ messages



Future developments

- Receivers and transmitters are mostly idle
- Use idle time to improve serialization
  - Key objective: reducing bandwidth
- Optional P2P architecture
  - For high-bandwidth clients (LAN)
- Adaptive input-to-execution delay
- Security
  - Authentication
  - Integrity
  - State validation (anti-cheat)

Main-server-thread	173.235 ms (100%)	495 ms
lockstep.LockstepServer.**run** ()	173.235 ms (100%)	495 ms
lockstep.LockstepServer.**handshakePhase** ()	92.285 ms (53,3%)	191 ms
java.util.concurrent.Semaphore.**acquire** ()	80.850 ms (46,7%)	203 ms
Self time	99,5 ms (0,1%)	99,5 ms
Receiver-from-23664	85.276 ms (100%)	85.276 ms
lockstep.LockstepReceiver.**run** ()	85.276 ms (100%)	85.276 ms
java.net.DatagramSocket.**receive** ()	84.440 ms (99%)	84.440 ms
java.io.ObjectInputStream.**readObject** ()	727 ms (0,9%)	727 ms
Self time	96,9 ms (0,1%)	96,9 ms
java.io.ObjectInputStream.**<init>** ()	10,8 ms (0%)	10,8 ms
Receiver-from-44134	85.070 ms (100%)	85.070 ms
lockstep.LockstepReceiver.**run** ()	85.070 ms (100%)	85.070 ms
java.net.DatagramSocket.**receive** ()	83.677 ms (98,4%)	83.677 ms
java.io.ObjectInputStream.**readObject** ()	1.070 ms (1,3%)	1.070 ms
lockstep.LockstepReceiver.**messageSwitch** ()	125 ms (0,1%)	125 ms
java.io.ObjectInputStream.**<init>** ()	102 ms (0,1%)	102 ms
Self time	93,9 ms (0,1%)	93,9 ms
Receiver-from-45749	84.088 ms (100%)	84.088 ms
lockstep.LockstepReceiver.**run** ()	84.088 ms (100%)	84.088 ms
java.net.DatagramSocket.**receive** ()	82.973 ms (98,7%)	82.973 ms
java.io.ObjectInputStream.**readObject** ()	763 ms (0,9%)	763 ms
java.io.ObjectInputStream.**close** ()	194 ms (0,2%)	194 ms
java.io.ObjectInputStream.**<init>** ()	156 ms (0,2%)	156 ms
Self time	0,000 ms (0%)	0,000 ms
Receiver-from-79335	83.192 ms (100%)	83.192 ms
Receiver-from-46387	82.199 ms (100%)	82.199 ms
Receiver-from-109846	81.068 ms (100%)	81.068 ms
Transmitter-to-109846	81.068 ms (100%)	5.956 ms
lockstep.LockstepTransmitter.**run** ()	81.068 ms (100%)	5.956 ms
java.lang.Thread.**sleep[native]** ()	75.111 ms (92,7%)	0,000 ms
lockstep.LockstepTransmitter.**processCommands** ()	5.956 ms (7,3%)	5.956 ms
Self time	0,000 ms (0%)	0,000 ms
Transmitter-to-44134	81.068 ms (100%)	6.667 ms