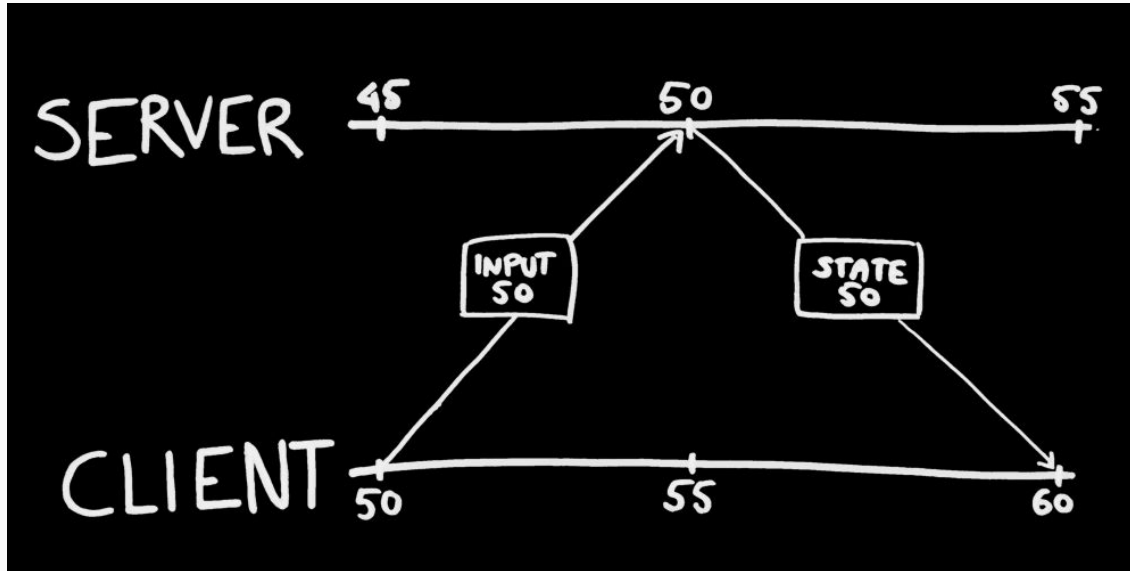


Multiplayer Game in C++ Reliability on top of UDP

Networks and Online Games

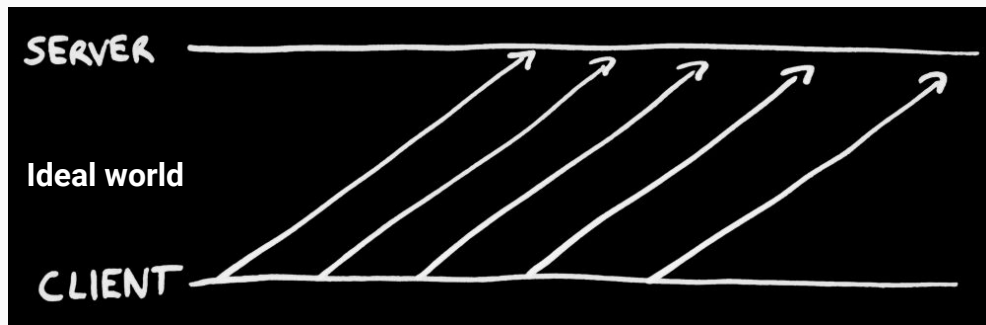
Real world conditions



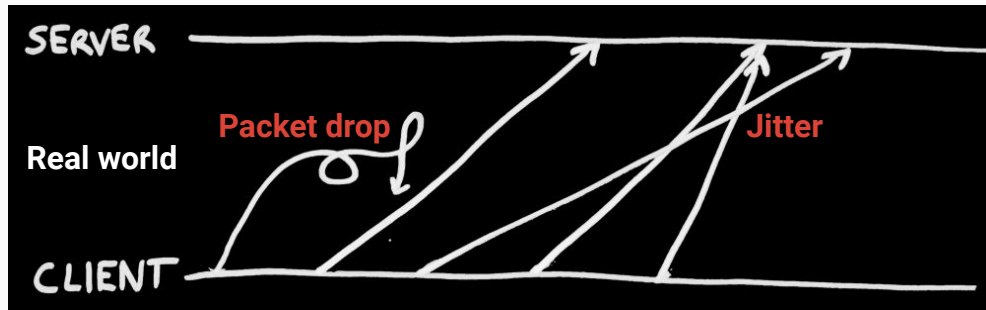
Lag is the delay provoked by the fact that data takes some time to travel and reach its destination.

The time spend for a packet to be sent and receive some response is called **RTT** (round trip time)

Real world conditions

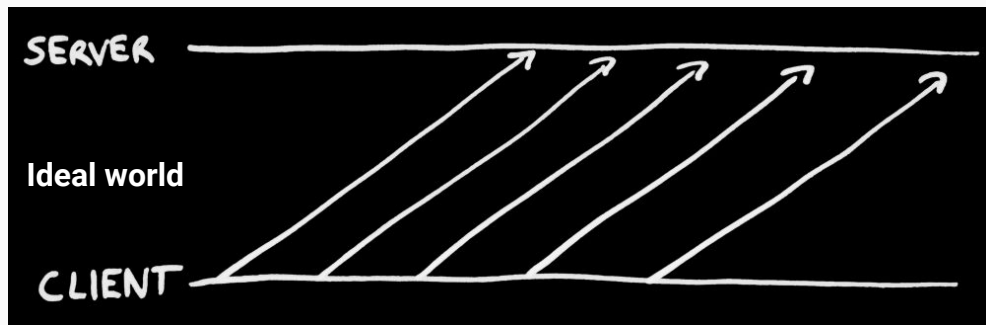


Ideally, packets would always reach its destination, and they would reach it in order (upper image).

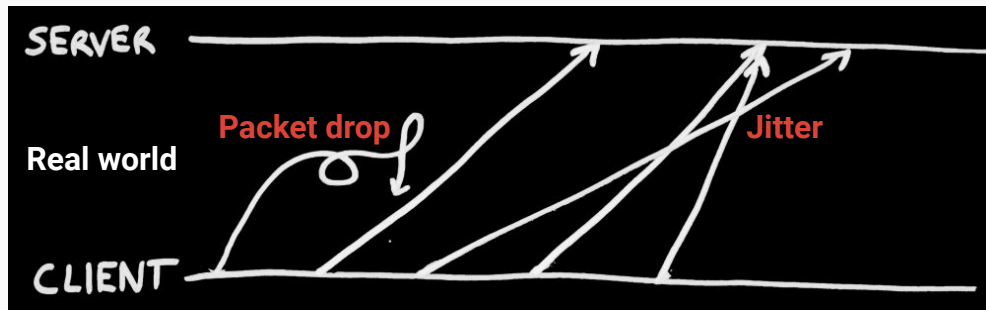


In the real world, however, some packets can be lost and even if they reach its destination, they can arrive out of order (lower image).

Real world conditions

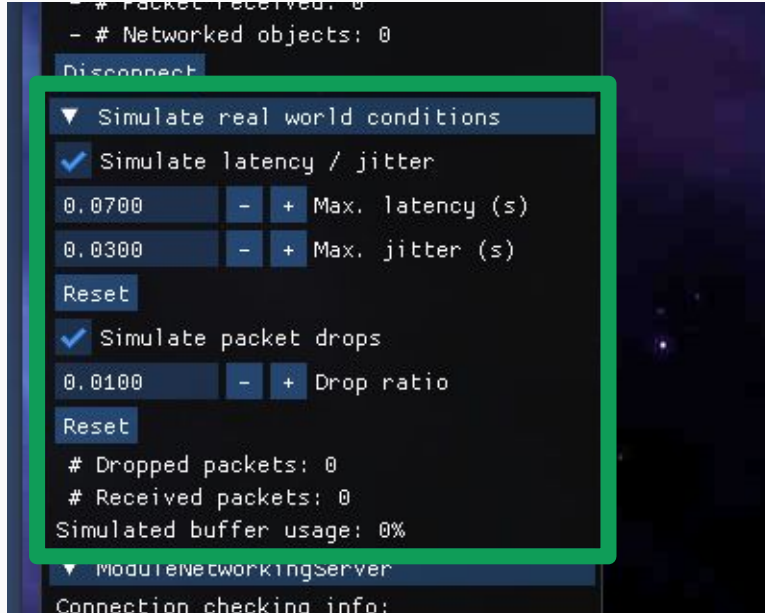


Packet drop or packet loss: is when a packet never reaches its destination.



Jitter: is the variance in time delay between data packets (e.g. because a routing reschedule) and it can provoke packets arriving unordered.

Simulation of real world conditions



UI Options in the provided skeleton

- Both in the client and in the server

Configurable reception parameters

- Latency (in seconds)
- Jitter (in seconds)
- Drop ratio (value from 0 to 1)

What we know about UDP

The good

- Fast
- Datagram / packet based

The bad

- Potential packet loss
- Potential out-of-order deliveries

What we will learn

Method 1

- Overcome packet loss using redundancy
 - We will use it for **input packets**
 - From client, we will send all generated inputs until the server notifies their reception

Method 2

- Packet delivery notification system (DeliveryManager class)
 - We will use it for the world state replication and other stuff

Overcoming packet loss using
redundancy

Main idea

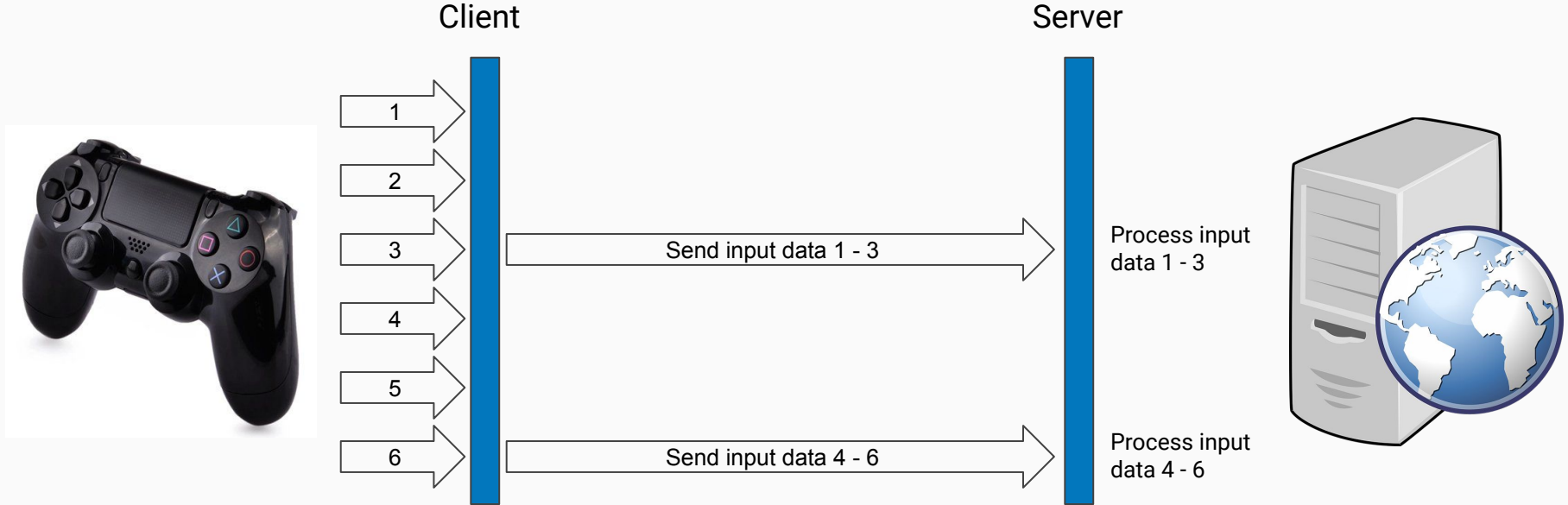
Sending data repeatedly until server notifies their reception

- Each data piece will have a sequence number
- A packet will contain several data pieces in sequence
- We will send all data pieces until their reception is not acknowledged
- **High level of redundancy :-)**

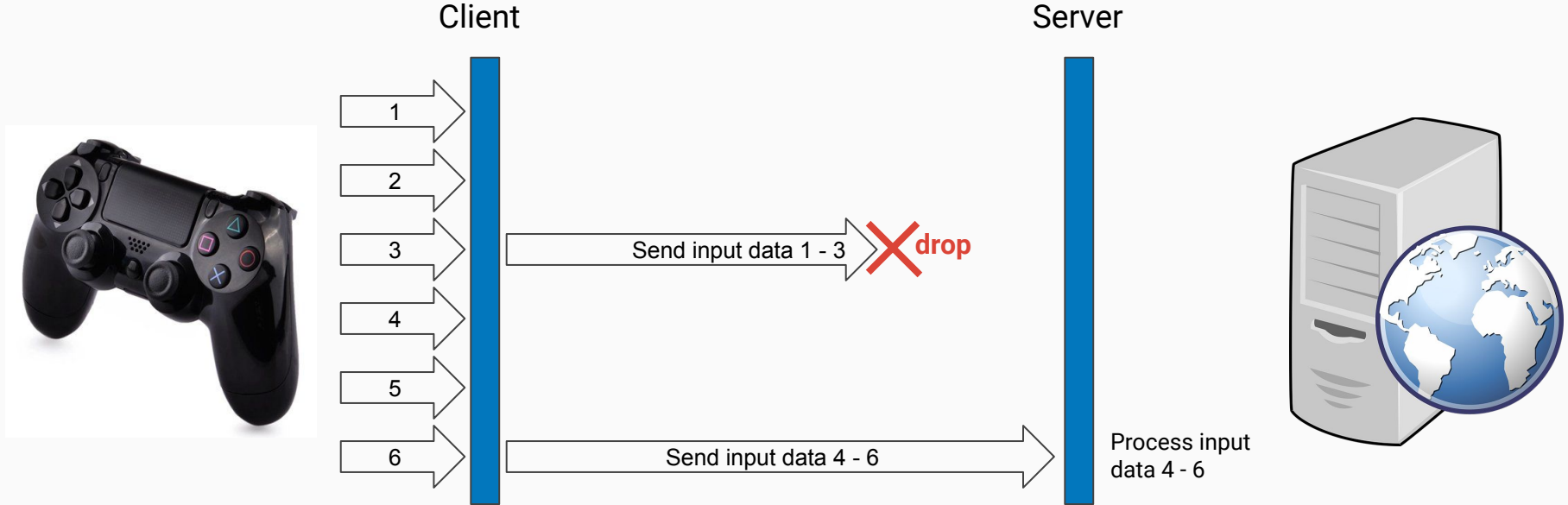
We will use this technique to send input packets

- **Input packets are small, their size will not be large despite redundancy**

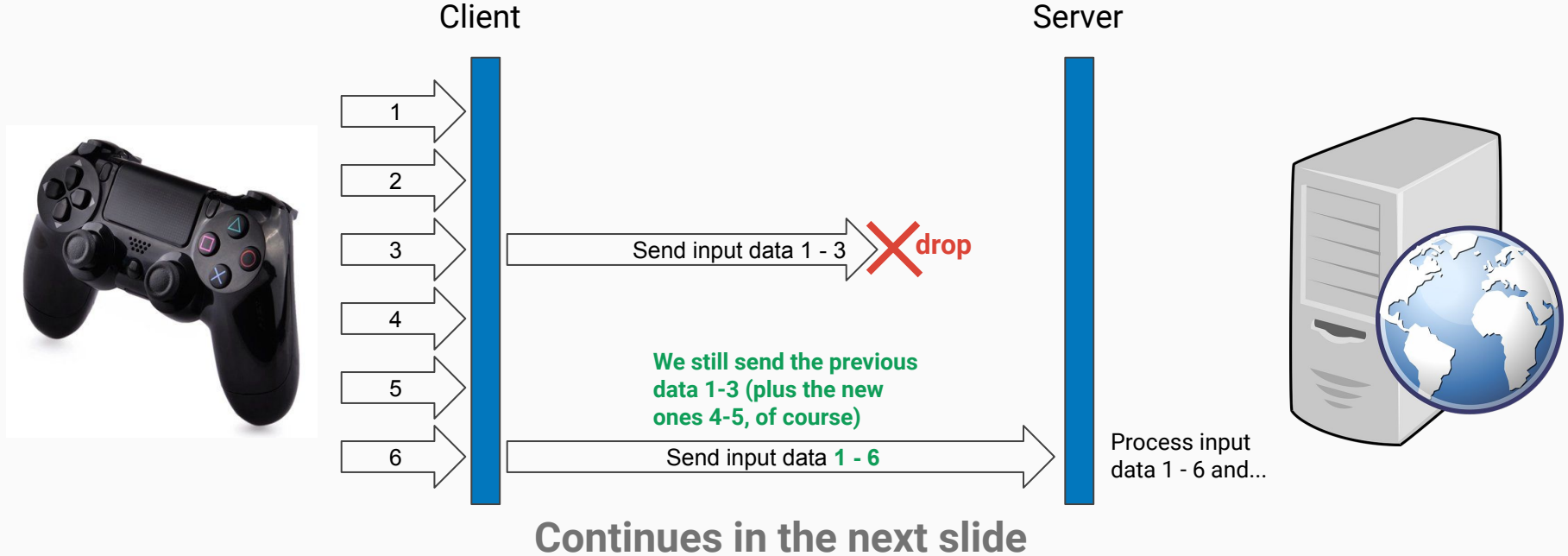
What we are doing so far



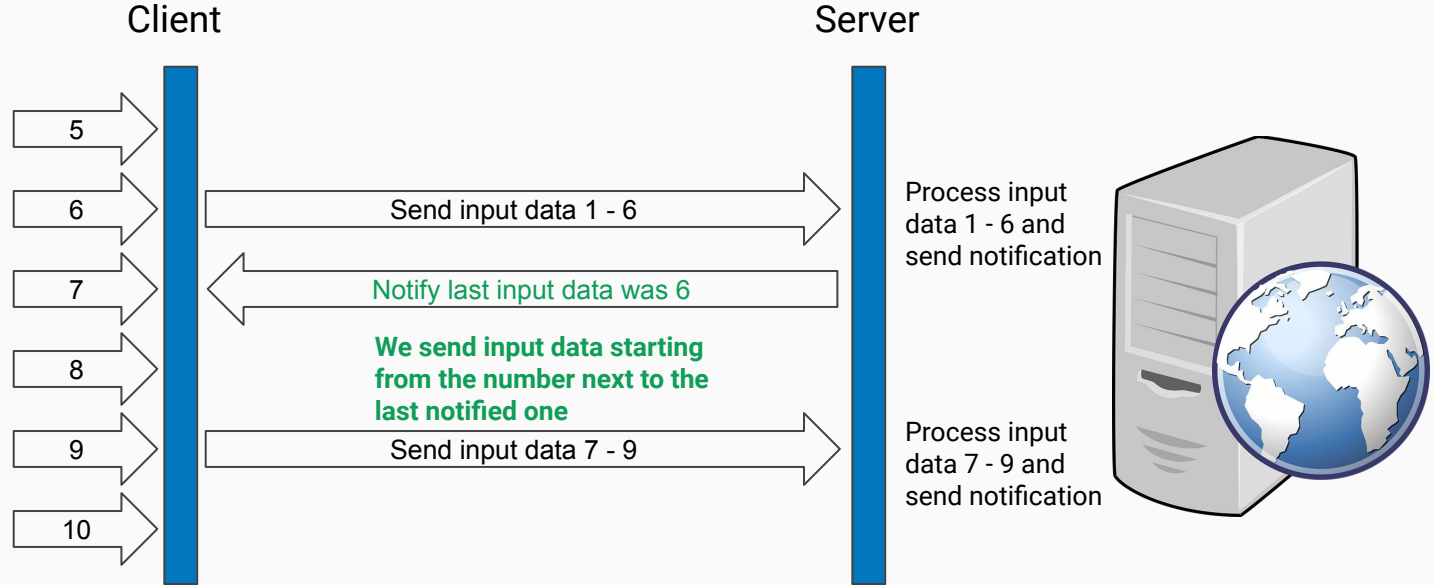
What will actually happen sometimes



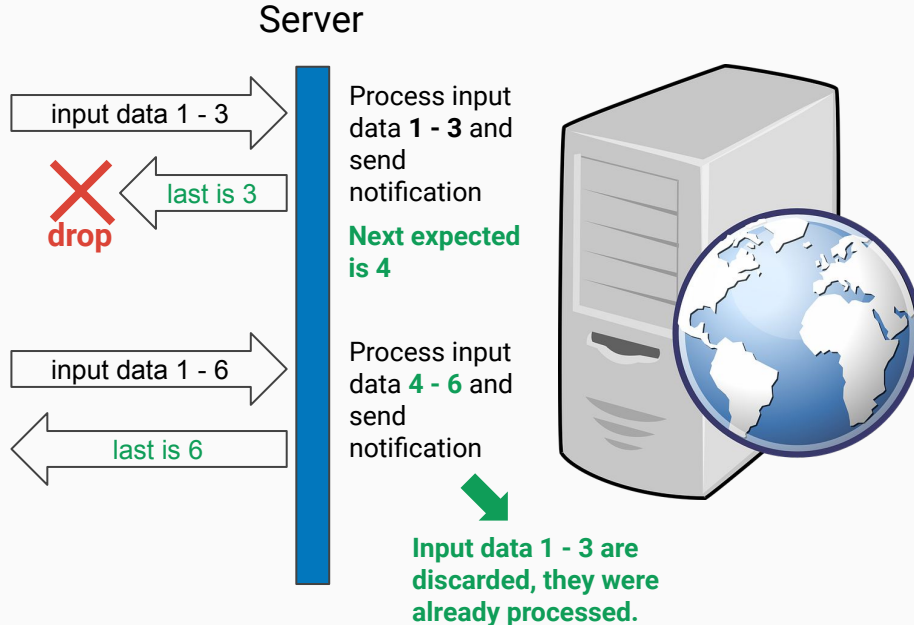
Solving this issue with redundancy



Solving this issue with redundancy



Input packet processing (server side)



Server processes input packets

- Only the next expected input data each time
- Registers the last processed input sequence number
- Sends notification to client (this is deferred into the next replication packet)

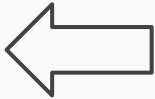
When do we send notifications from server?

A couple of options

- Each time we receive an input data packet
 - Faster response
 - If the client sends input data packets very frequently, we will use a lot of bandwidth
- With the next replication packet
 - Slower response
 - Optimized bandwidth usage

When do we send notifications from server?

A couple of options

- Each time we receive an input data packet
 - Faster response
 - If the client sends input data packets very frequently, we will use a lot of bandwidth
 - With the next replication packet 
 - Slower response
 - Optimized bandwidth usage
- As we will deal with techniques to compensate for lag, and bandwidth usage is a real concern, we will prefer the second one.

Change in ModuleNetworkingClient

```
OutputMemoryStream packet;
packet << ClientMessage::Input;

for (uint32 i = inputDataFront; i < inputDataBack; ++i)
{
    InputPacketData &inputPacketData = inputData[i % ArrayCount(inputData)];
    packet << inputPacketData.sequenceNumber;
    packet << inputPacketData.horizontalAxis;
    packet << inputPacketData.verticalAxis;
    packet << inputPacketData.buttonBits;
}

// Clear the queue
inputDataFront = inputDataBack;

sendPacket(packet, serverAddress);
```

By deleting this, we allow to send repeated packets until receiving the last input packet processed by the server

Clients send input packets

- **inputDataFront** is the index of the first input data to send
- **inputDataBack - 1** is the index of the last input data to send

TODOs

Make the necessary changes to implement this technique

- In **ModuleNetworkingServer**, send notifications about the last processed input data.
- In **ModuleNetworkingClient**, when receiving notifications about input data processing from the server, update inputDataFront to avoid sending already processed input data.

For reference

Read this blog post.

More importantly, read the section entitled “I was wrong”.

<http://www.codersblock.org/blog/multiplayer-fps-part-8>

Packet delivery notification
(DeliveryManager class)

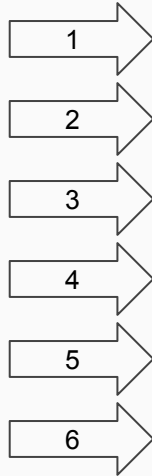
What we are doing so far

Replication commands

- Create
- Update
- Destroy

Server

Client



Written by ReplicationManagerServer
Sent by ModuleNetworkingServer

Replication data 1 - 3

Received by ModuleNetworkingClient
Read by ReplicationManagerClient

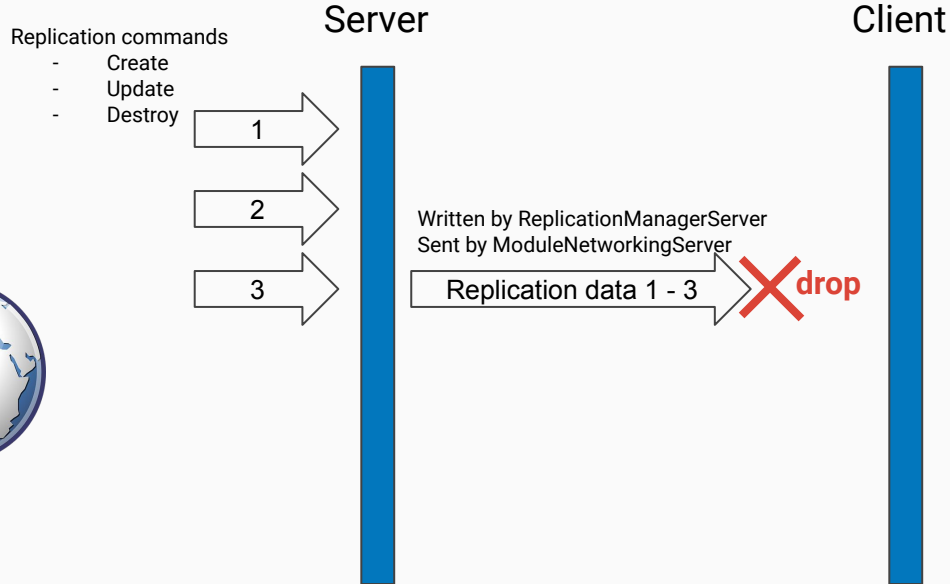
Written by ReplicationManagerServer
Sent by ModuleNetworkingServer

Replication data 4 - 6

Received by ModuleNetworkingClient
Read by ReplicationManagerClient



What will actually happen sometimes



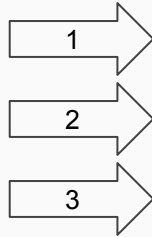
Delivery notification system

Replication commands

- Create
- Update
- Destroy

Server

Client



Replication type
Replication info...

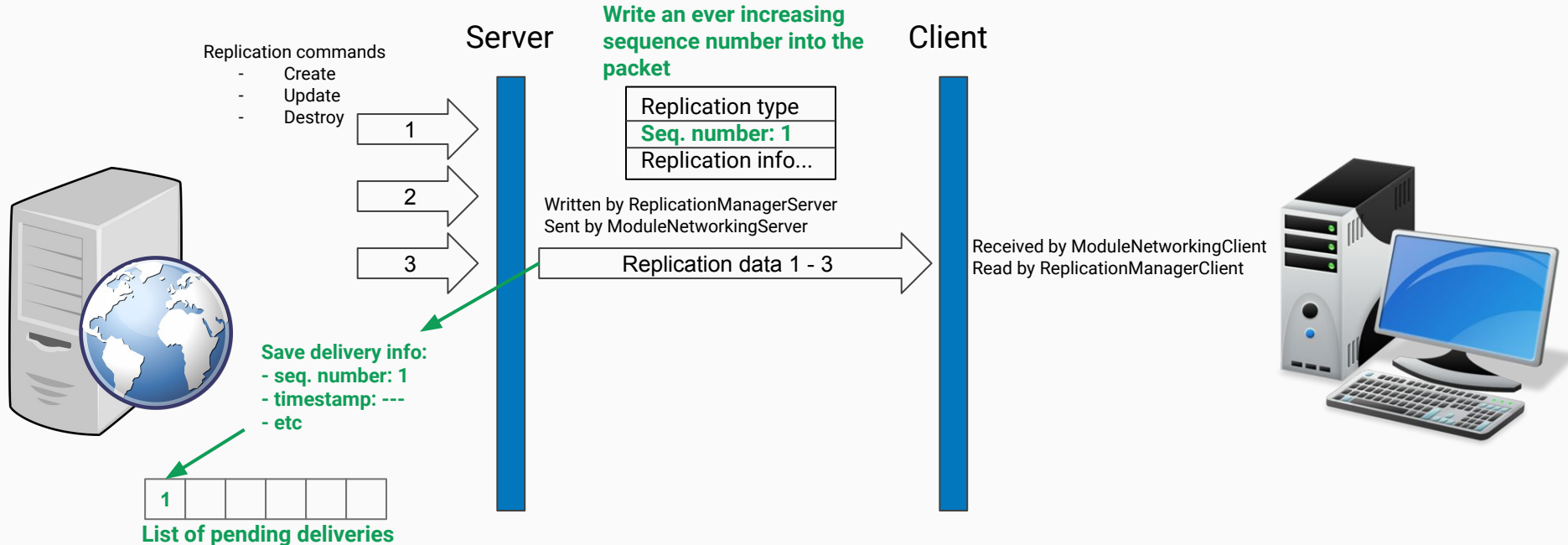
Written by ReplicationManagerServer
Sent by ModuleNetworkingServer

Replication data 1 - 3

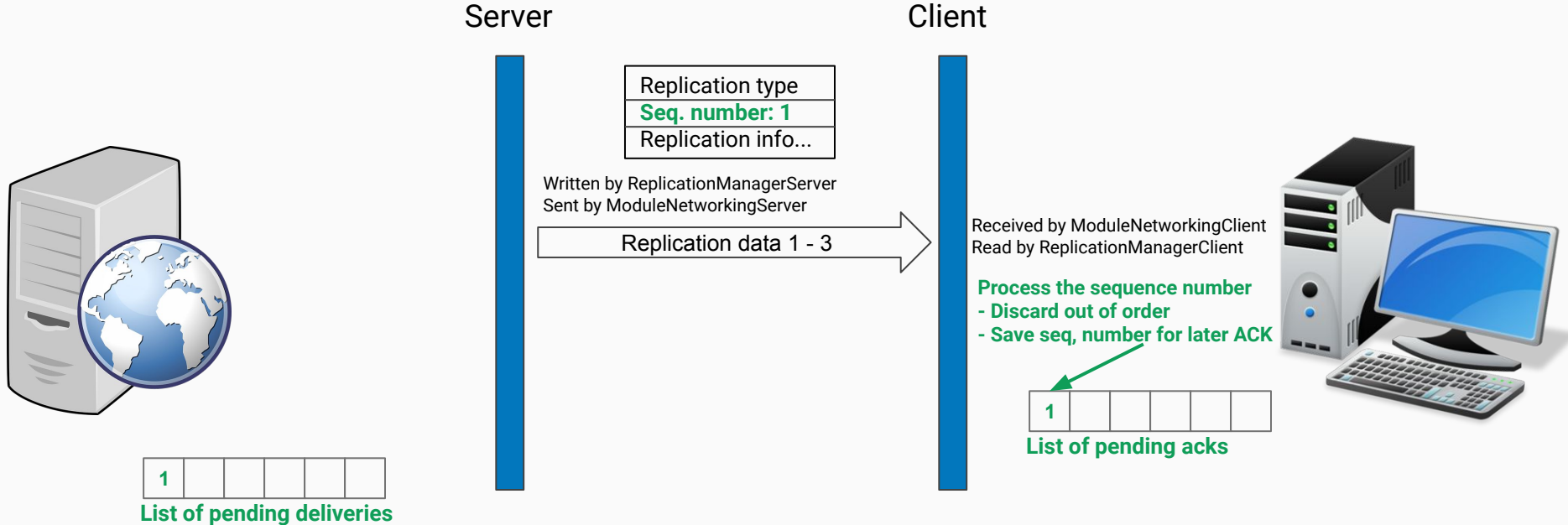
Received by ModuleNetworkingClient
Read by ReplicationManagerClient



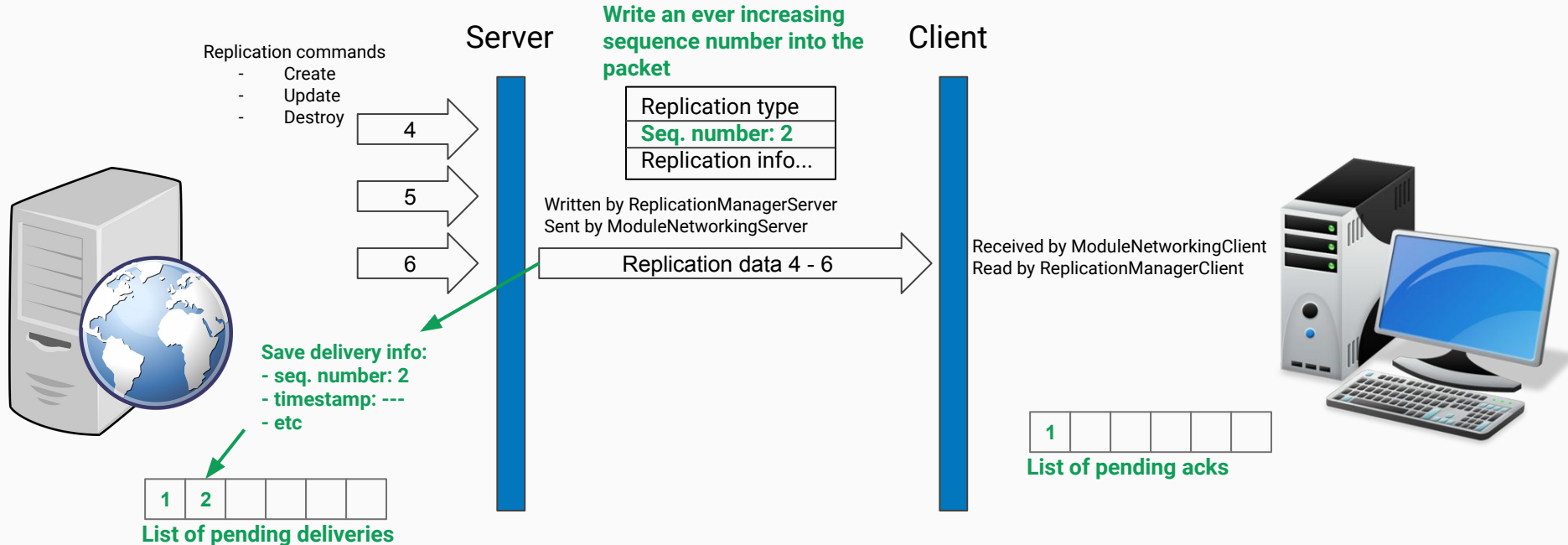
Delivery notification system



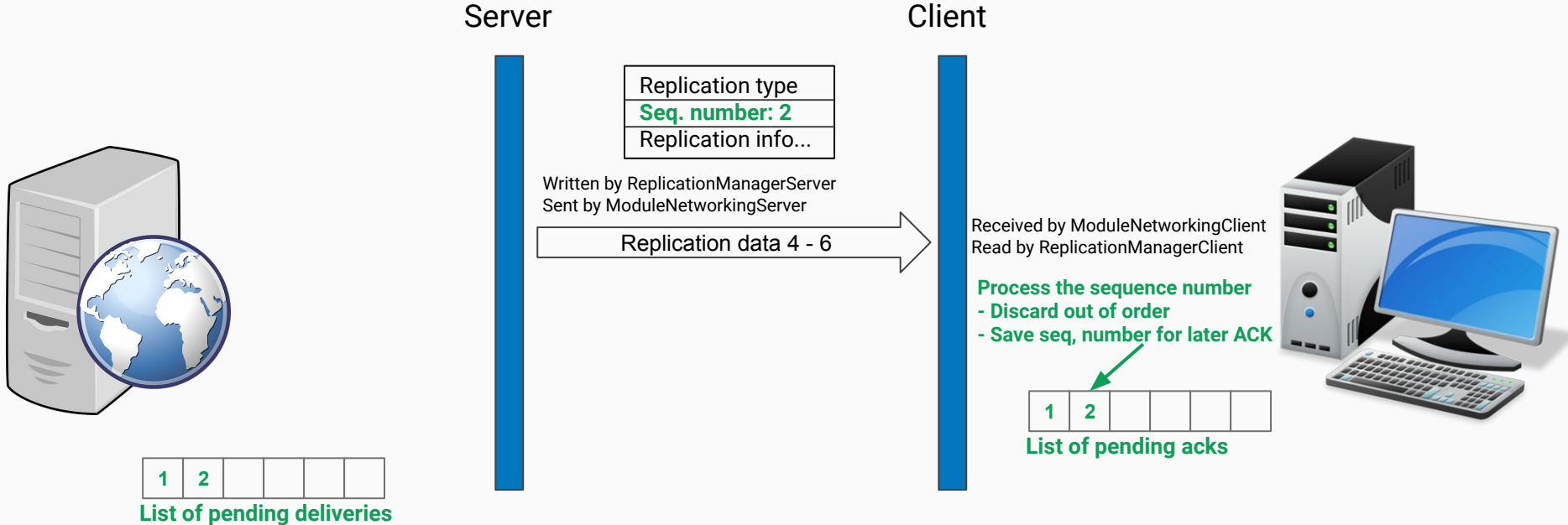
Delivery notification system



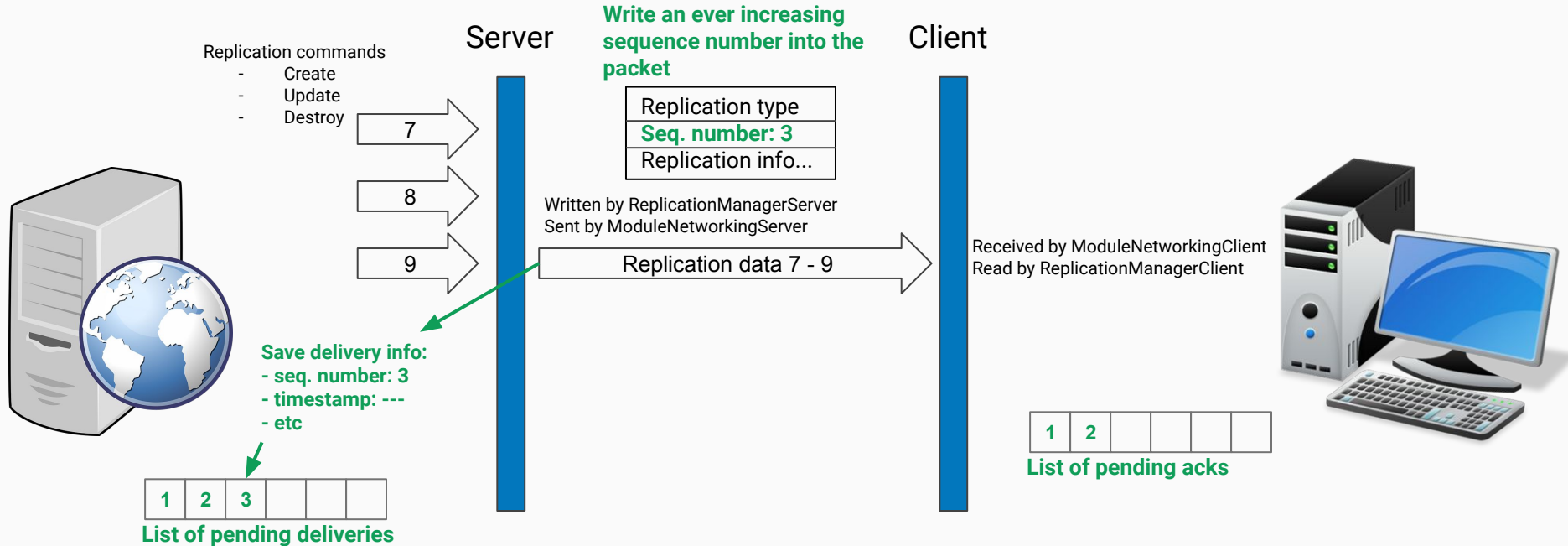
Delivery notification system



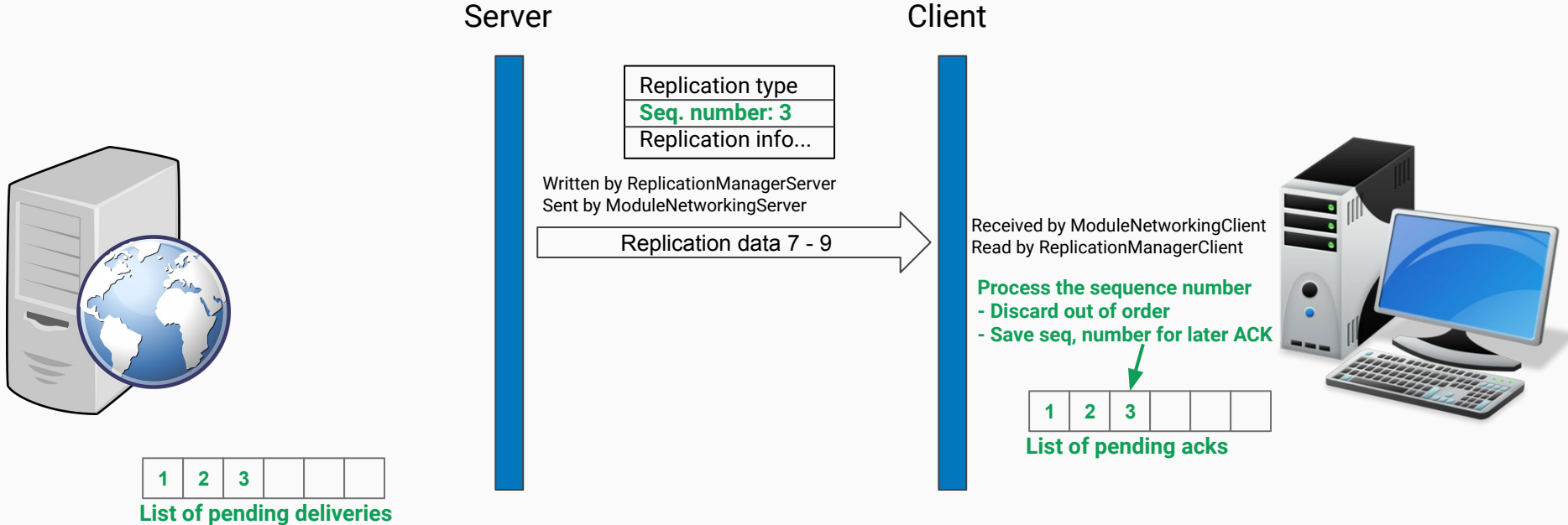
Delivery notification system



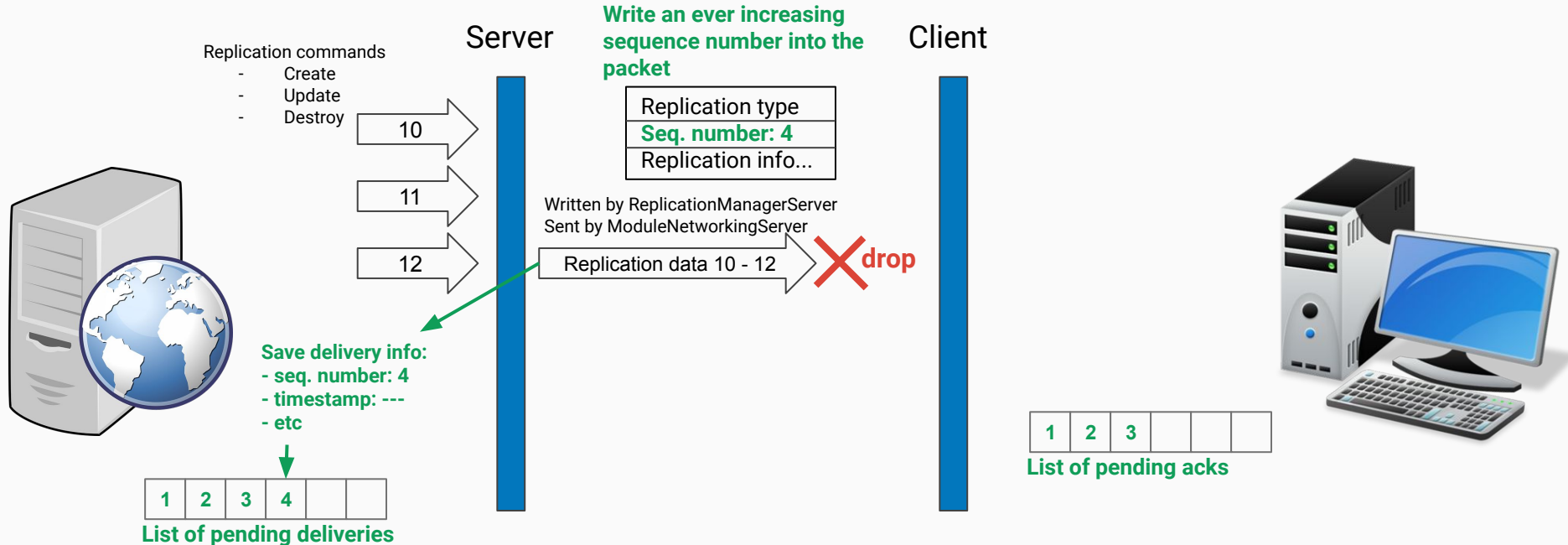
Delivery notification system



Delivery notification system



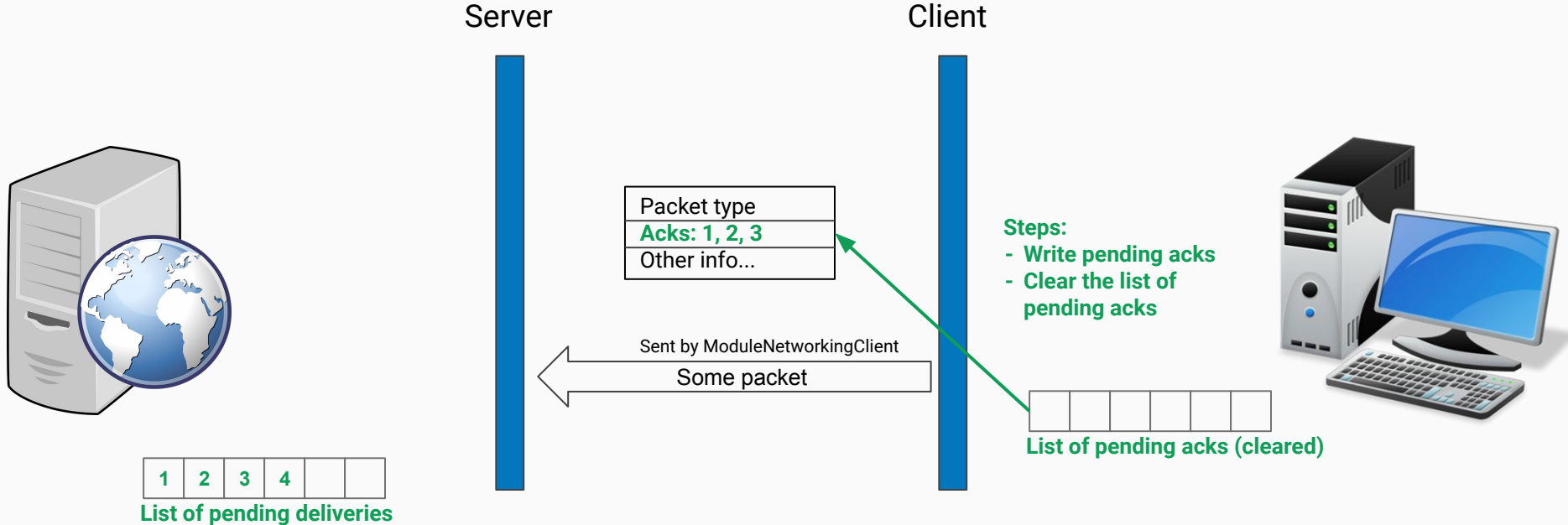
Delivery notification system



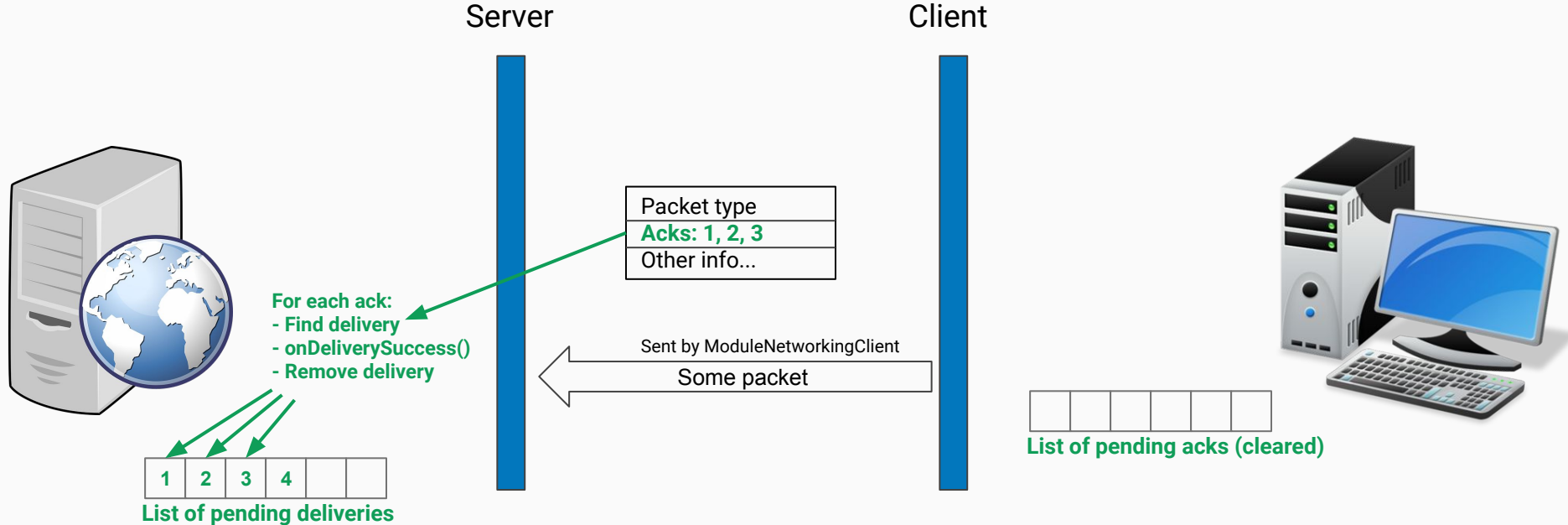
Delivery notification system



Delivery notification system



Delivery notification system



Delivery notification system



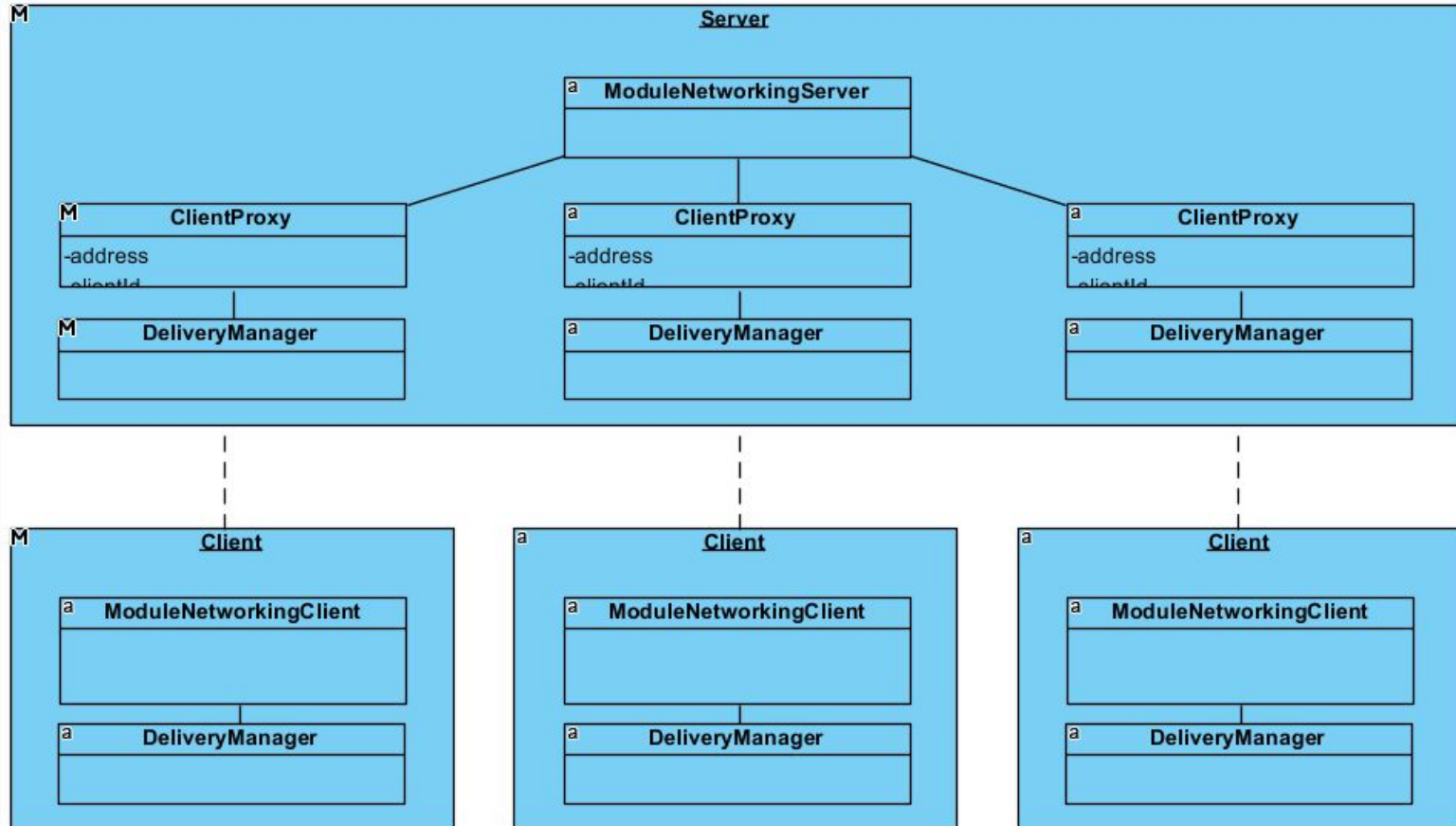
Delivery notification system



Delivery notification system



Schematic overview of the delivery managers



DeliveryManager header

```
class DeliveryManager;

class DeliveryDelegate
{
public:
    virtual void onDeliverySuccess(DeliveryManager *deliveryManager) = 0;
    virtual void onDeliveryFailure(DeliveryManager *deliveryManager) = 0;
};

struct Delivery
{
    uint32 sequenceNumber = 0;
    double dispatchTime = 0.0;
    DeliveryDelegate *delegate = nullptr;
};
```

DeliveryManager header

```
class DeliveryManager
{
public:

    // For senders to write a new seq. numbers into a packet
    Delivery * writeSequenceNumber(OutputMemoryStream &packet);

    // For receivers to process the seq. number from an incoming packet
    bool processSequenceNumber(const InputMemoryStream &packet);

    // For receivers to write ack'ed seq. numbers into a packet
    bool hasSequenceNumbersPendingAck() const;
    void writeSequenceNumbersPendingAck(OutputMemoryStream &packet);

    // For senders to process ack'ed seq. numbers from a packet
    void processAckdSequenceNumbers(const InputMemoryStream &packet);
    void processTimedOutPackets();

    void clear();
```

```
private:

    // Private members (sender side)
    // - The next outgoing sequence number
    // - A list of pending deliveries

    // Private members (receiver side)
    // - The next expected sequence number
    // - A list of sequence numbers pending ack
};
```

Delivery notification system steps I

When sending a new packet (from host A)

- Make a new delivery using *writeSequenceNumber()*
 - Write the sequence number into the packet
 - Store the new delivery into a list into the manager
 - Deliveries contain the sequence number, delivery time, etc
 - Return the delivery
- Register notification callbacks into the Delivery object
 - Whenever we detect the packet was received or dropped, those callbacks will be called
 - `onSuccess()`
 - `onFailure()` - We can choose to send a packet again

Delivery notification system steps II

When receiving the packet (in host B)

- Process the sequence number using *processSequenceNumber()*
 - Check that the sequence number arrives in order
 - If not, discard the whole packet
 - Add the sequence number to a list of “pending acknowledgements”
- If not discarded, deserialize and process the packet normally
- At the end of the frame, or after some time interval
 - Send a packet with all the acknowledged sequence numbers from all received packets
 - For that, use *writeSequenceNumbersPendingAck()*

Delivery notification system steps III

When receiving a packet with acknowledgements (back in host A)

- Process the acknowledged seq. numbers *processAckdSequenceNumbers()*
 - For each sequence number, find the packet delivery information
 - Invoke its callbacks *onSuccess()/onFailure()* as needed
 - Remove the delivery

At each frame we check all pending deliveries for timeout

- Call to *processTimedOutPackets()*
 - For each delivery that timed out, call *onFailure()* and remove the delivery

TODOs

- Implement the **DeliveryManager** (DeliveryManager.h / .cpp)
- In **ModuleNetworkingServer**
 - Write sequence numbers into replication packets
 - Register callbacks (*onSuccess()*, *onFailure()*) into the created delivery
- In **ModuleNetworkingClient**.
 - Process the sequence number from replication packets (discard the packet if out of order)
 - At some point send a packet with sequence number acks
- In **ModuleNetworkingServer** (again)
 - Process sequence number acks coming from some packet (call *onSuccess()* / *onFailure()*)
 - Check for timeouts (call *onFailure()* when needed)

For reference

Multiplayer game programming book

- Read chapter 7: Latency, Jitter, and reliability (you have the pdf)

Useful code in Github:

- [DeliveryNotificationManager header](#)
- [DeliveryNotificationManager source](#)

