Networking lecture 5

Networking Game
Implementation Tips
Hosting(?)

How Do You Create a Networked Game?

1. First create a single player game, and then `add the networking’
2. Create client and server at the same time, and when you’re done, run them and hope it works
3. Make a clear design up front, and then implement it such that every step can be tested along the way

How Do You Create a Networked Game?

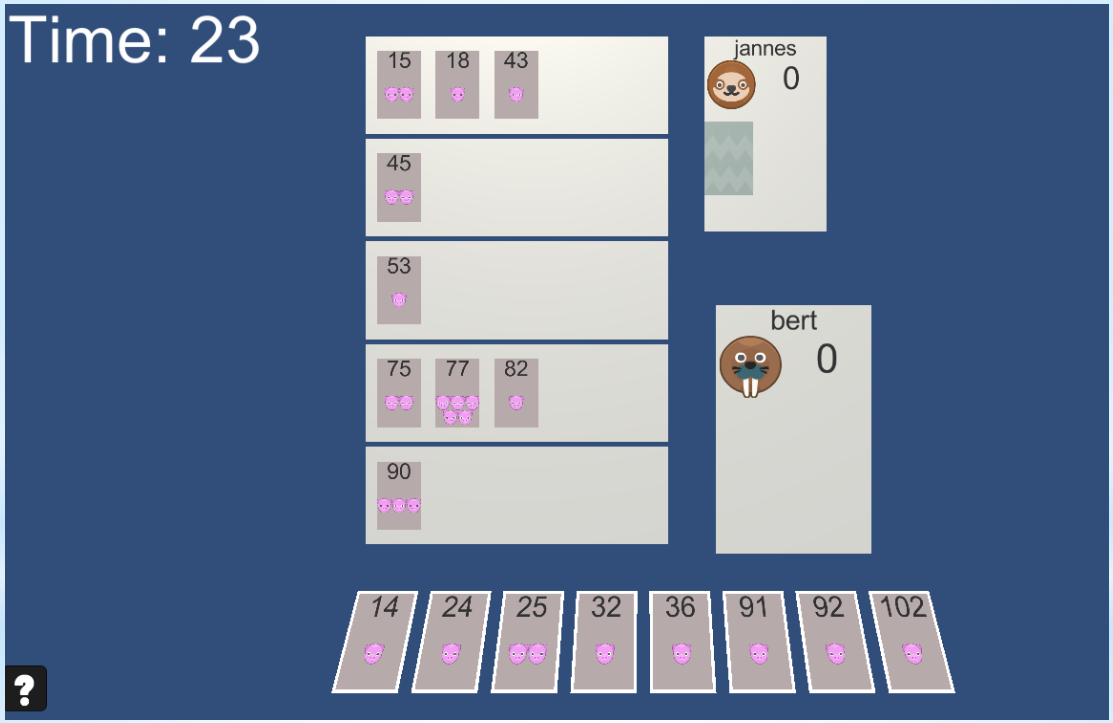
1. ~~First create a single player game, and then `add the networking'~~
2. ~~Create client and server at the same time, and when you're done, run them and hope it works~~
3. Make a clear design up front, and then implement it such that every step can be tested along the way

- *Make a clear design up front, and then implement it such that every step can be tested along the way*

- How can you make a clear design for a networked game?
- How can you split up the implementation into testable steps?

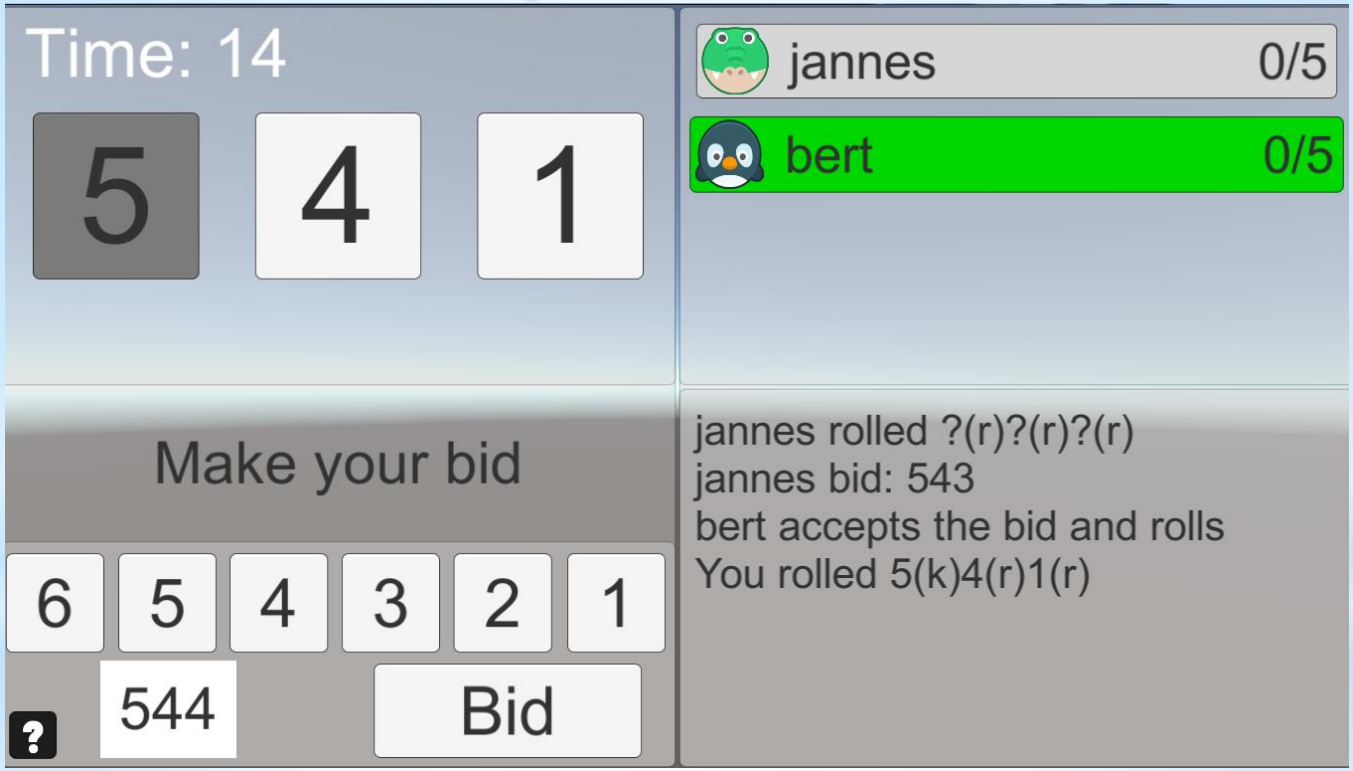
→ Today: some tips, based on example games

Example Game 1



- Take 5: https://boardgamegeek.com/boardgame/153/take-5
- 2-6 players play cards *simultaneously* from their *private* hand, which are then put on the matching rows. They should avoid playing the 6th card in a row, which leads to collecting cards.

Example Game 2



- Bluf Poker: https://nl.wikipedia.org/wiki/Blufpoker (sorry, Dutch only!)
- 2-6 players play *take turns* rolling dice, *hidden* under a cup, and make a corresponding bid. The next player can choose to accept or call the bid. They should avoid making a wrong call, or having their bluf bid called.

Outline

- Design & Implementation – based on examples
  - Game States and Protocol
  - Program Architecture
  - General Tips
- Hosting
  - Professional services
  - WAN vs LAN, and Network Address Translation
  - Port forwarding
  - Peer-to-peer and hole-punching (?)

Design and Implementation 1

Game States and Protocol

Approach

How to start a complex project like a networked game, such that every step can be tested?

(Step 0: Make sure your basic tools work (TcpNetworkConnection, Packet, Server superclass, …?))

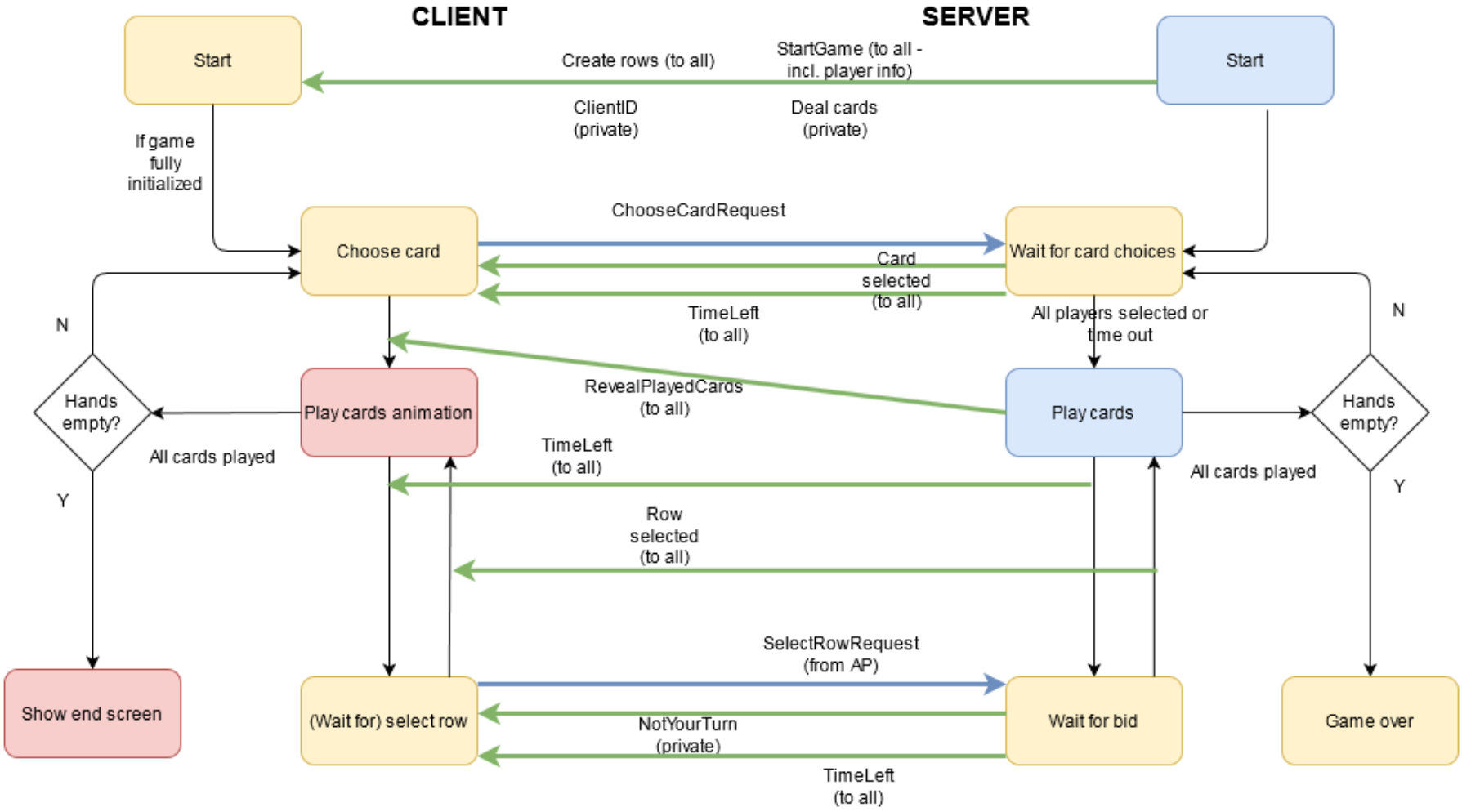
1. Analyze the game on paper; create diagrams
2. Implement the game *Model* (MVC)
3. Create a server that owns a model
4. Create a test client (console? Can send messages in any order)
5. Create a Unity scene with game objects, not enforcing game rules
6. Connect the client to the server
7. Add lobby / game rooms / match making / …

(Again: Test every step!)

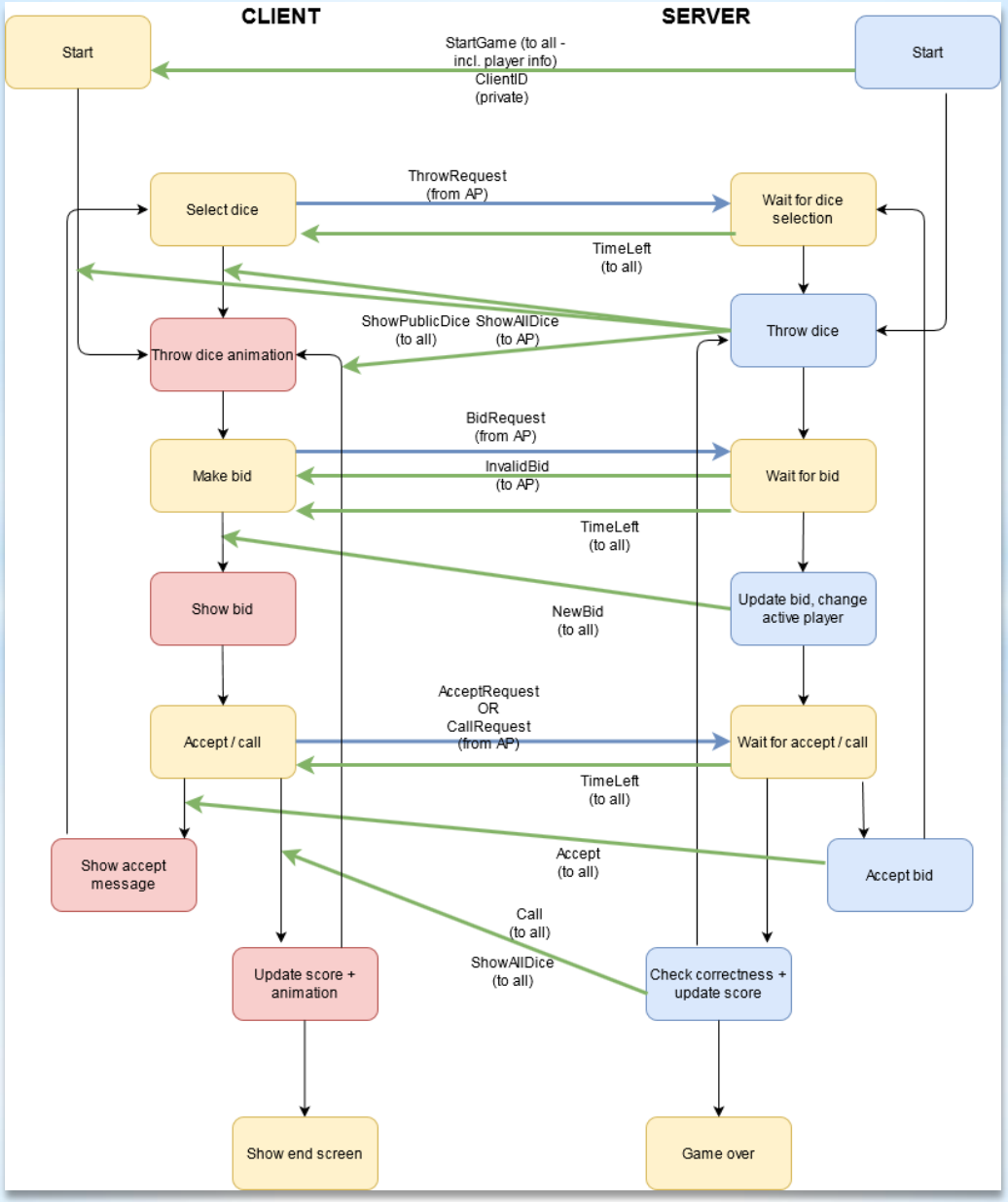
1. Analyze (or design) the game on paper:
    1. Write down the game states and transitions (*FSM*)
    2. Write down *protocol* (client-> server)
        1. Which messages (can) cause state changes?
        2. Which messages are private, which are broadcast?
    3. Create a *UML class diagram* for the Model

Tip: Call client → server messages *…Request,* and server→client messages …*Command* or …*Event.*

FSM and Protocol for Take 5



FSM and Protocol for Bluf Poker



Things to Note

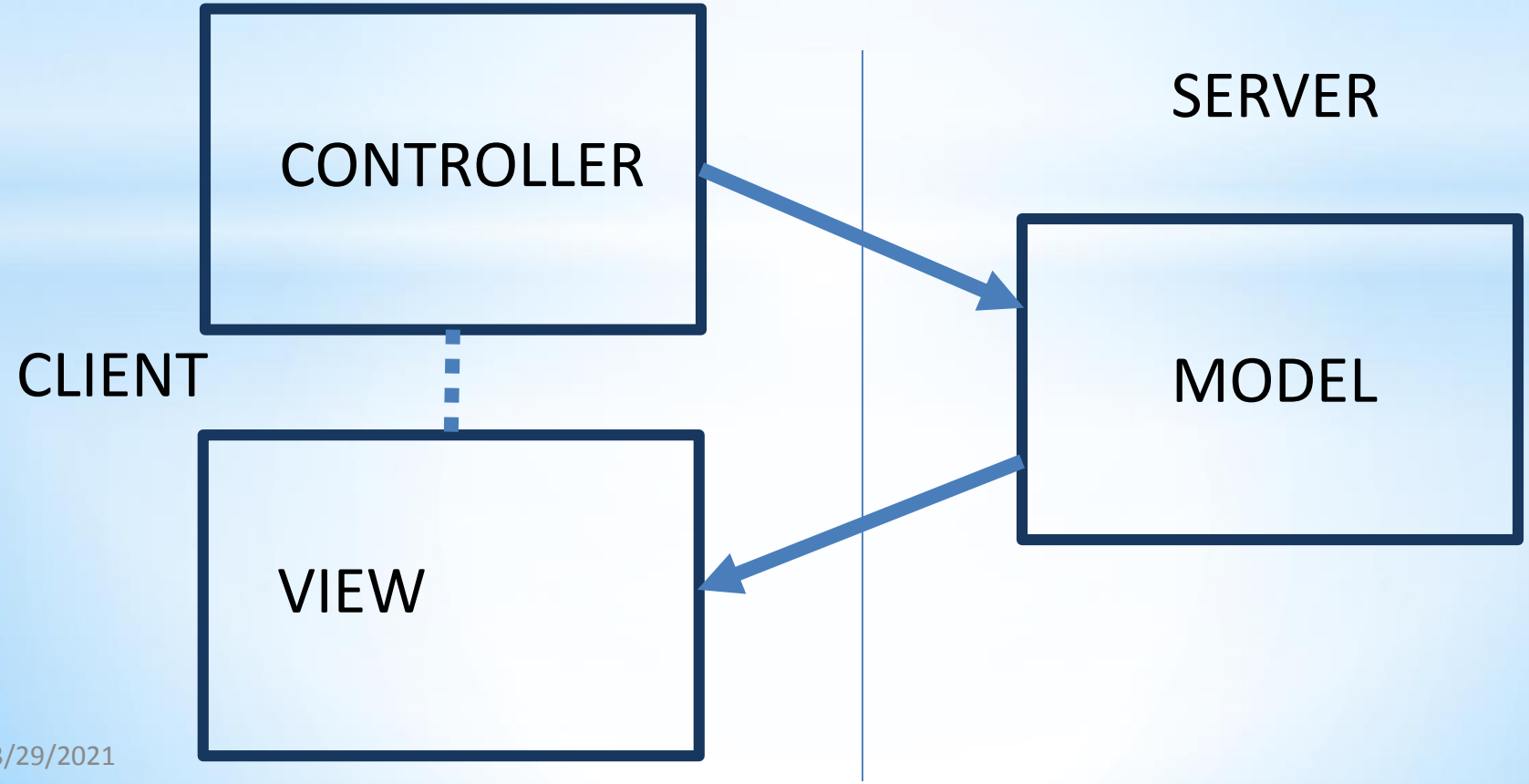
- There are `real states' (yellow) where the server waits for the client(s) to do an action, and `transition states' (blue/red) where the server does some computations, and the client possibly shows some animations in response.
- Keep track of public vs private information!
- If the client knows about the game model, the number of messages sent can be very minimal! (Unlike a heavy-handed Mirror/Photon `sync-transform' approach!)

Design and Implementation 2

Code Architecture

Step 2 & 3

1. Analyze the game on paper; create diagrams
2. Implement the game *Model* (MVC)
3. Create a server that owns a model



Encapsulation?

- In good program design, you don't just want to make every property public: the model owns its data, and the data is modified through its methods.
- Nevertheless, client-side you'd like to be able to change everything directly, based on incoming server commands. Maybe you even need to sync (=overwrite) the game state after a disconnect!

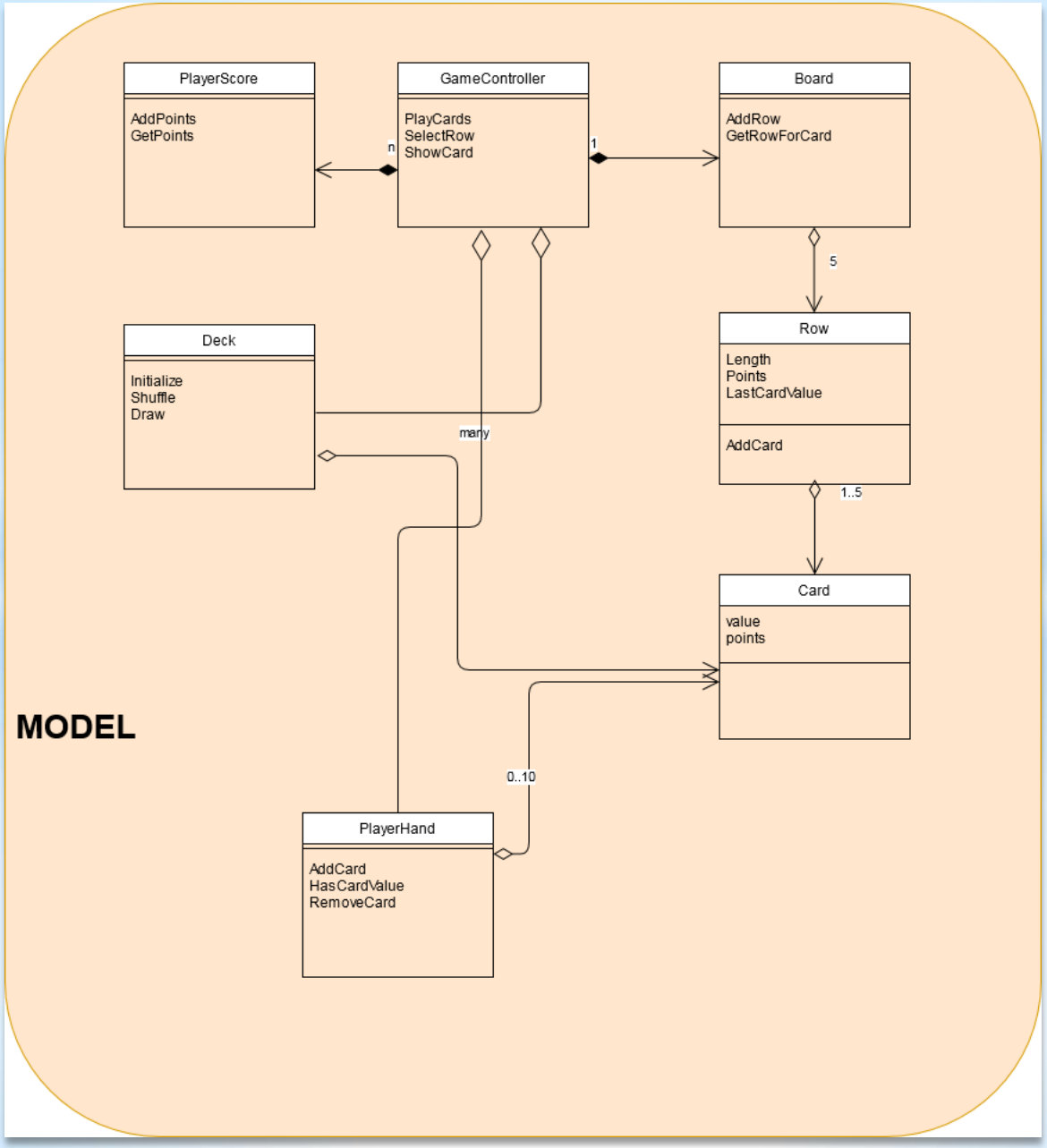
Possible Approach

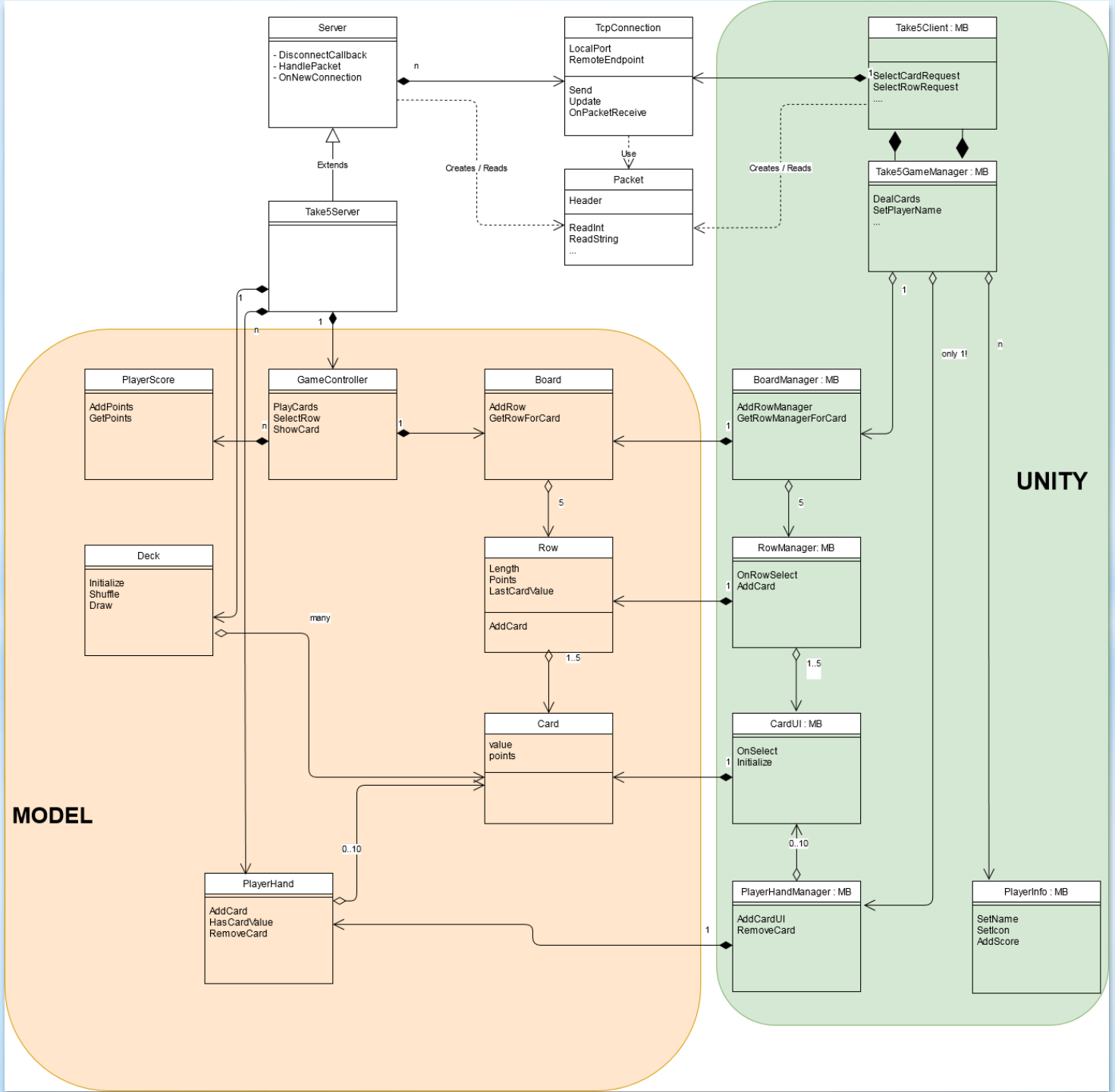
- Super class *MyGameData*
  - Contains all game state data (cards in hand, score, etc.), with basic methods (GetScore, PlayCard). Fields are *protected.*
- Server side: *MyGameModel : MyGameData*
  - Contains game rules. Verifies correct play and updates state accordingly
- Client side: *MyGameUpdater : MyGameData*
  - Adds *public setters* to change properties, based on incoming server commands.

Note:

- The Unity way of doing things (Game Objects or ECS) is not very compatible with a pure MVC pattern
- However, it's good to at least keep the core Model separated from View / Controller (UI) details

- Let's first look at a possible UML for the Take 5 model, and then a full client/server implementation


PlayerScore
AddPoints
GetPoints
GameController
PlayCards
SelectRow
ShowCard
Board
AddRow
GetRowForCard
n
1
5
Deck
Initialize
Shuffle
Draw
Row
Length
Points
LastCardValue
AddCard
many
1..5
Card
value
points
MODEL
0..10
PlayerHand
AddCard
HasCardValue
RemoveCard


Server
- DisconnectCallback
- HandlePacket
- OnNewConnection
TcpConnection
LocalPort
RemoteEndpoint
Send
Update
OnPacketReceive
Take5Client : MB
SelectCardRequest
SelectRowRequest
....
Extends
Creates / Reads
Use
Packet
Header
ReadInt
ReadString
...
Take5Server
Take5GameManager : MB
DealCards
SetPlayerName
...
only 1!
PlayerScore
AddPoints
GetPoints
GameController
PlayCards
SelectRow
ShowCard
Board
AddRow
GetRowForCard
BoardManager : MB
AddRowManager
GetRowManagerForCard
UNITY
Deck
Initialize
Shuffle
Draw
many
Row
Length
Points
LastCardValue
AddCard
RowManager: MB
OnRowSelect
AddCard
Card
value
points
CardUI : MB
OnSelect
Initialize
MODEL
PlayerHand
AddCard
HasCardValue
RemoveCard
PlayerHandManager : MB
AddCardUI
RemoveCard
PlayerInfo : MB
SetName
SetIcon
AddScore

Things to note

- Shared classes between client & server:
  - NetworkingTools (TcpConnection, Packet)
  - The game model
- In Unity, I chose to use MonoBehaviours that wrap their model counterparts (not pure MVC!)
- Separating the networking details from the game implementation:
  - Server vs Take5Server (event based: OnDisconnect, HandlePacket)
  - Take5Client vs Take5GameController (knows nothing about NetworkConnections!)

General Tips

Tips for improving iteration speed

- While working on the game scene: allow starting directly in this scene (without going through login screens) – the game server starts with listener
- Even for a multiplayer game: allow playing with only one player
- Keep your `test client’ (Console project?) at hand

Tips & Pitfalls - Client

- Make no assumptions about what will happen in reaction to user input! (E.g. “I pressed the play button – let’s already load the next scene”)
  - Send requests
  - Do important things based on incoming commands
  - Client may only decide itself on things like playing audio, animations
- When receiving a `change scene’ command: immediately stop reading incoming messages!

Tips & Pitfalls - General

- Cheat protection: don’t send `private information’ (e.g. cards in hand) to all clients – assume everyone can read every packet you send!
  - Less important for casual games, but still, a good habit

Tips & Pitfalls – Server 1

- Avoid complex call stacks + for loops!
  - Example:
    - Foreach client: handle incoming messages (→ Client 1: `StartGameRequest')
    - In reponse: BroadCast `StartGame' command (for loop)
    - While broadcasting: Notice that client 3 is disconnected
    - In response: BroadCast a `Client 3 Disconnected' event
  - In which order do client 2 and 4 receive these commands / events? (!)
  - Solution: *queue delayed actions*

Tips & Pitfalls – Server 2

- Be careful with delayed actions!
  - Example: in the same server update:
    - Client 1 sends a `StartGameRequest' → there are enough players in the room, so queue `StartGameEvent'
    - Client 2 sends a `LeaveRoomRequest' → queue a `RoomChange' event for client 2
  - Now what?
  - To make matters worse, this is a bug that happens very rarely (typically, only at critical times, like when showing your game to a teacher)

Game Suggestions

- Conclusion: for your first game, keep it simple!
- Start with creating diagrams - If they are much bigger than those shown here, reconsider…
- Some suggestions:
  - Skull & Roses
    - http://www.skull-and-roses.com/
  - Can't Stop
    - https://boardgamegeek.com/boardgame/41/cant-stop
  - Zombie Dice
    - https://boardgamegeek.com/boardgame/62871/zombie-dice
  - Martian Dice
    - https://boardgamegeek.com/boardgame/99875/martian-dice
  - Sushi Go
    - https://gamewright.com/product/Sushi-Go

Hosting

Hosting Your Game

- You're all making awesome networking games of course
- …but now you want to share them, and play them with others over the internet (especially in times of isolation…)
  - Casually play with fellow students
  - Demo for portfolio
  - Test performance under real conditions
- Today: how to host your own game server

Professional Hosting

- For running your socket-based executable (as opposed to more common web servers), there are different options
  - Look at `compute' servers
  - Example: Amazon Web Services (AWS), Azure, Google Cloud
- I ran my demo on AWS, EC2 micro. (The first 700 hours are free)
- Info:
  - https://docs.aws.amazon.com/AWSEC2/latest/WindowsGuide/get-set-up-for-amazon-ec2.html
  - https://docs.aws.amazon.com/AWSEC2/latest/WindowsGuide/EC2_GetStarted.html
  - https://docs.aws.amazon.com/quickstarts/latest/vmlaunch/step-2-connect-to-instance.html

Disclaimer

- If you commercially release a networking game or service, always pay for a professional hosting service!
  - High power servers at central positions
  - Guaranteed uptime
  - Scalable solution (maybe your game becomes a hit?)

- …but this always costs money. For testing, or casually playing your own game, you can temporarily host a small server yourself.

First Attempt

- You type *ipconfig* in a terminal, and find that your IP is 192.168.1.2 (or something similar).
- You start your server on port 55555.
- You mail your friend these data, and share your client project with him.
- Your friend tries to log in from his home, but can't…
- …Maybe he even tells you "Hey, my IP is also 192.168.1.2!"
- …What's going on here?

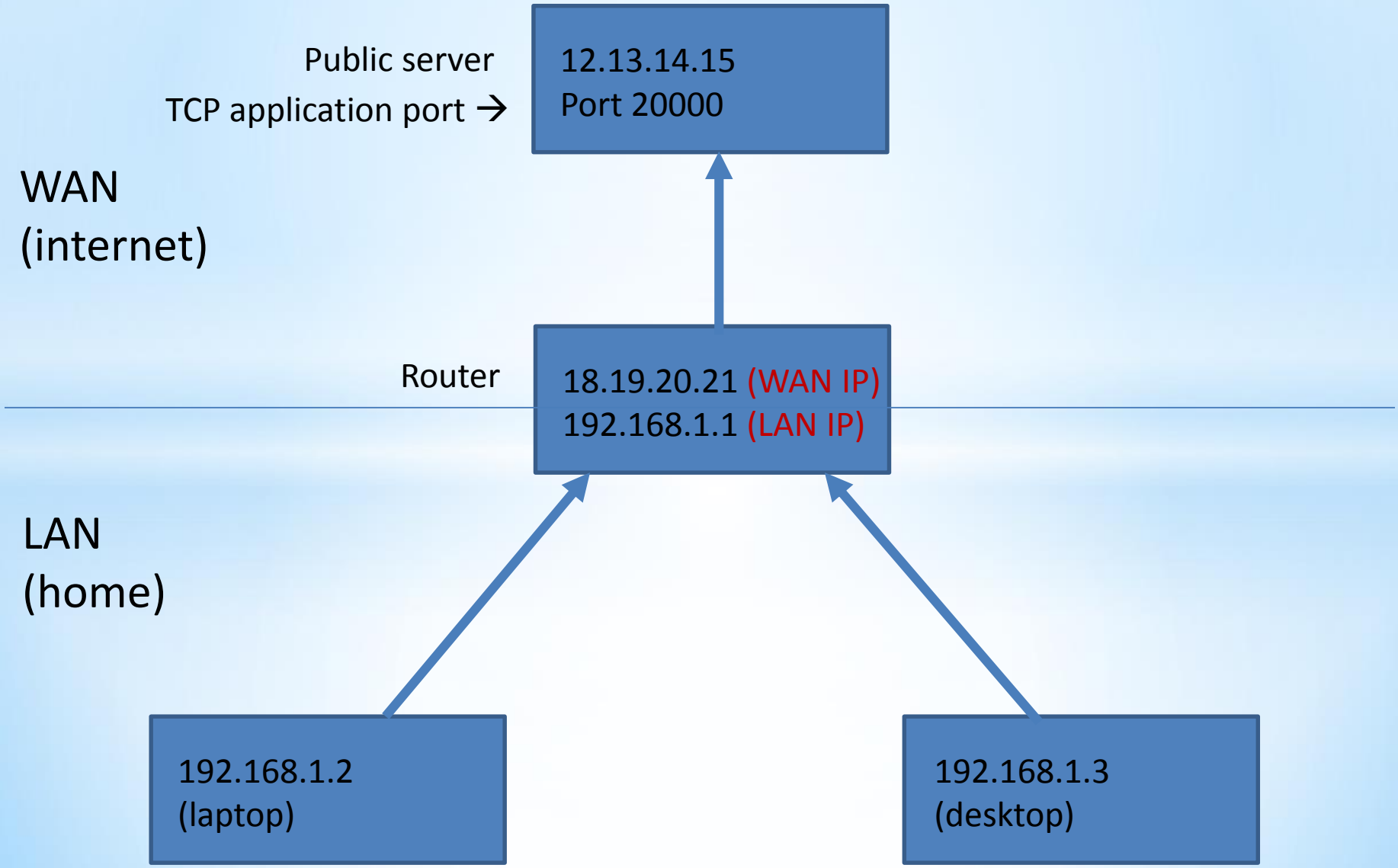
Outline

- WAN vs LAN, and Network Address Translation
- Port forwarding
- Peer-to-peer and hole-punching

LAN, WAN and NAT

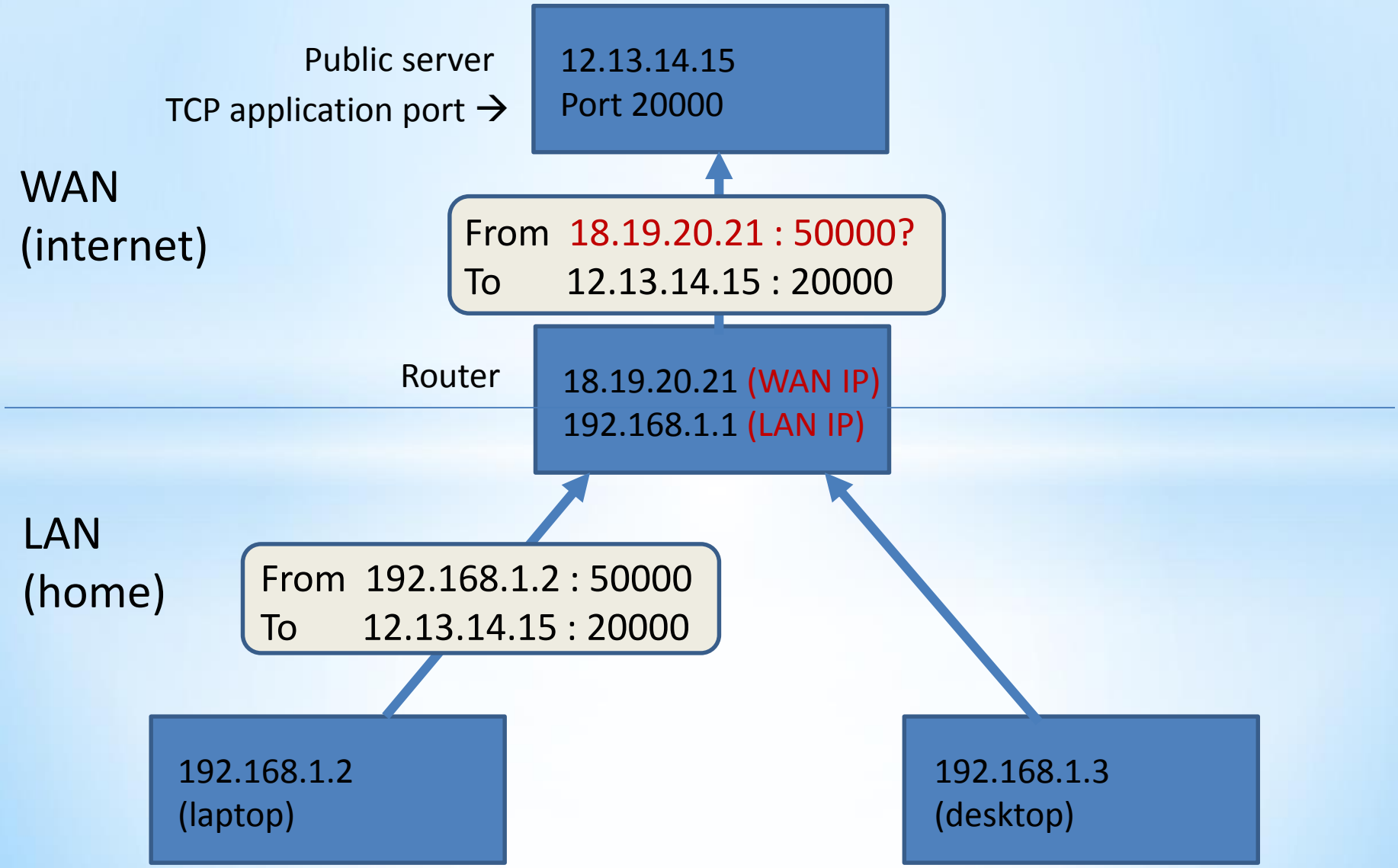
Local Area Network (LAN)

- In your home, you have one *router* that is connected to the internet.
- Multiple devices connect to this router (laptop, desktop, phone, console, more…)
- These devices all have an IP of the form 192.168.X.Y (it might also be e.g. 172.X.Y.Z)
- This is your *Local Area Network (LAN)*
- Your router has a single IP address on the internet ➔ The *Wide Area Network (WAN)*


Public server
TCP application port →
12.13.14.15
Port 20000
WAN
(internet)
Router
18.19.20.21 (WAN IP)
192.168.1.1 (LAN IP)
LAN
(home)
192.168.1.2
(laptop)
192.168.1.3
(desktop)

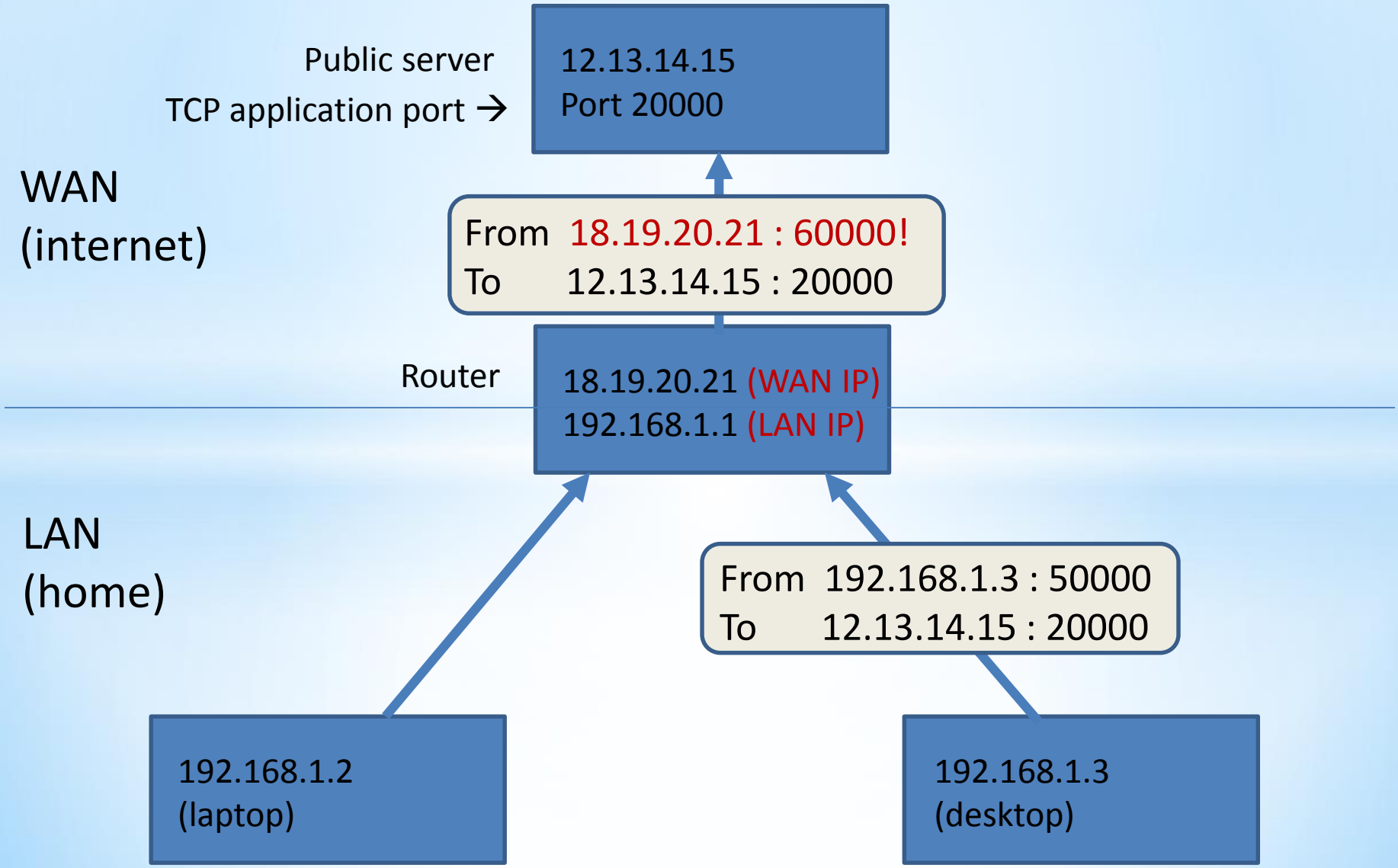
- Whenever you send a packet (e.g. a TCP protocol packet) to a server on the internet, your router overwrites your sender IP with his own WAN IP

- That way, the server knows who to send the reply to

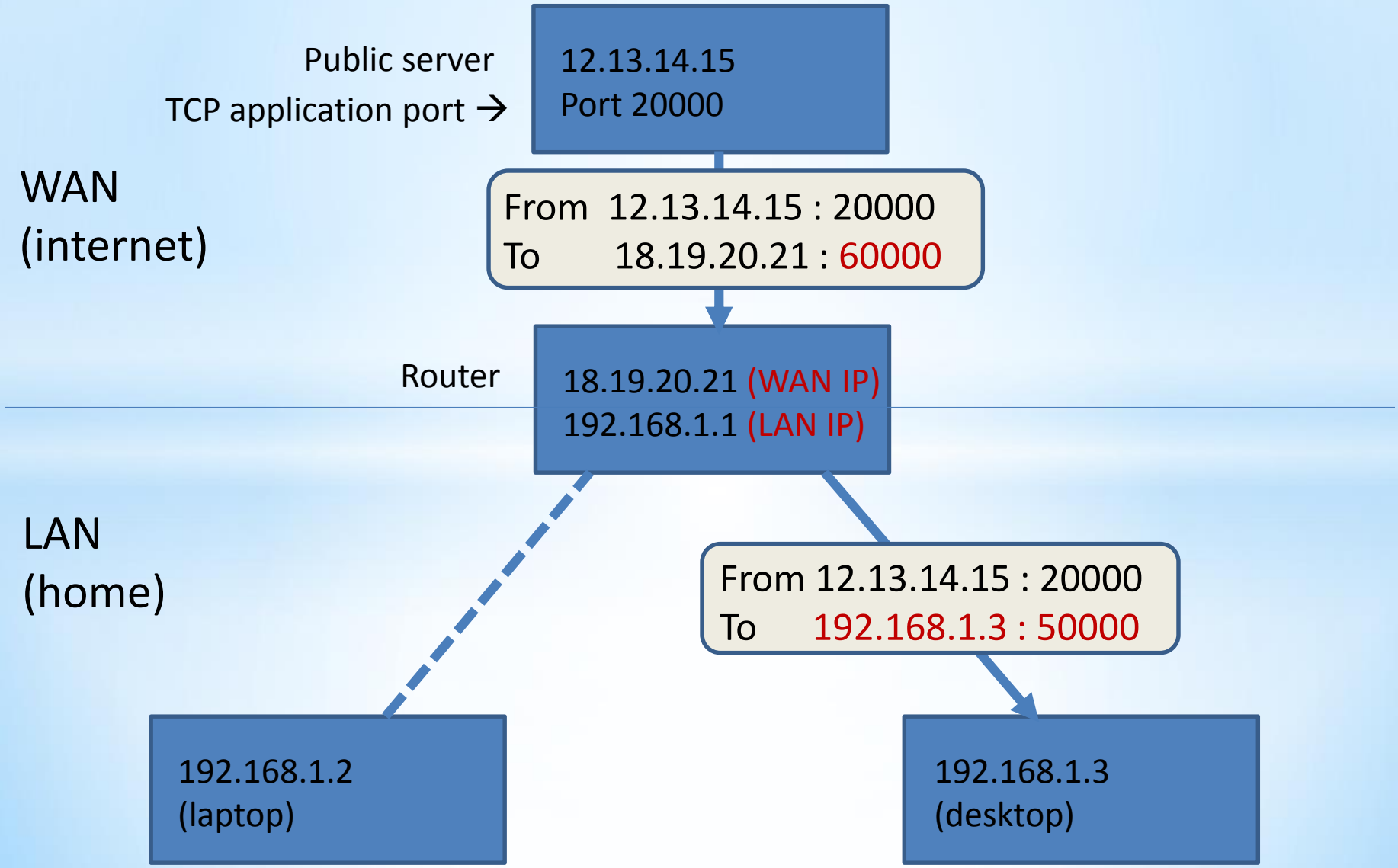

Public server
TCP application port →
12.13.14.15
Port 20000
WAN
(internet)
From 18.19.20.21 : 50000?
To 12.13.14.15 : 20000
Router
18.19.20.21 (WAN IP)
192.168.1.1 (LAN IP)
LAN
(home)
From 192.168.1.2 : 50000
To 12.13.14.15 : 20000
192.168.1.2
(laptop)
192.168.1.3
(desktop)

- The router may also overwrite the *port number* of the packet, to keep track of which LAN device sent it

- (Maybe your house mate is also playing Among Us, and is sending very similar packets!)


Public server
TCP application port →
12.13.14.15
Port 20000
WAN
(internet)
From 18.19.20.21 : 60000!
To 12.13.14.15 : 20000
Router
18.19.20.21 (WAN IP)
192.168.1.1 (LAN IP)
LAN
(home)
From 192.168.1.3 : 50000
To 12.13.14.15 : 20000
192.168.1.2
(laptop)
192.168.1.3
(desktop)

- If the server sends a reply, the router knows where to forward the packet, based on the port number!


Public server
TCP application port →
12.13.14.15
Port 20000
WAN
(internet)
From 12.13.14.15 : 20000
To 18.19.20.21 : 60000
Router
18.19.20.21 (WAN IP)
192.168.1.1 (LAN IP)
LAN
(home)
From 12.13.14.15 : 20000
To 192.168.1.3 : 50000
192.168.1.2
(laptop)
192.168.1.3
(desktop)

Network Address Translation (NAT)

- Your router maintains a (dynamic) *Network Address Translation table* to keep track of this info.

Protocol	LAN IP:	Port:	Assigned port:	Remote IP	Remote port
TCP	192.168.1.2	50000	50000	12.13.14.15	20000
	192.168.1.2	40000	40000	8.8.8.8	80
TCP	192.168.1.3	50000	60000	12.13.14.15	20000
				19.20.21.22	20000

Key things to know

- (In well-behaved routers,) LAN IP + port uniquely determine the *assigned port* (this is a one-to-one match)
- The same LAN IP + port may send packets to different servers (maybe you play the same game on different servers, e.g. for Europe and USA?)
- For security, the router *usually* only accepts incoming packets from registered remote IP + port combinations (=answers to outgoing requests.)
- The router has separate NAT tables for TCP and UDP

- So how do you host your own server on 192.168.1.2 port 55555?
- Maybe: send your WAN IP to your friend
- But even if you figure out your WAN (assigned) port, the router still won't accept packets from strangers on the internet!

- Solution: tell your router to accept these packets, using *port forwarding*

Port Forwarding

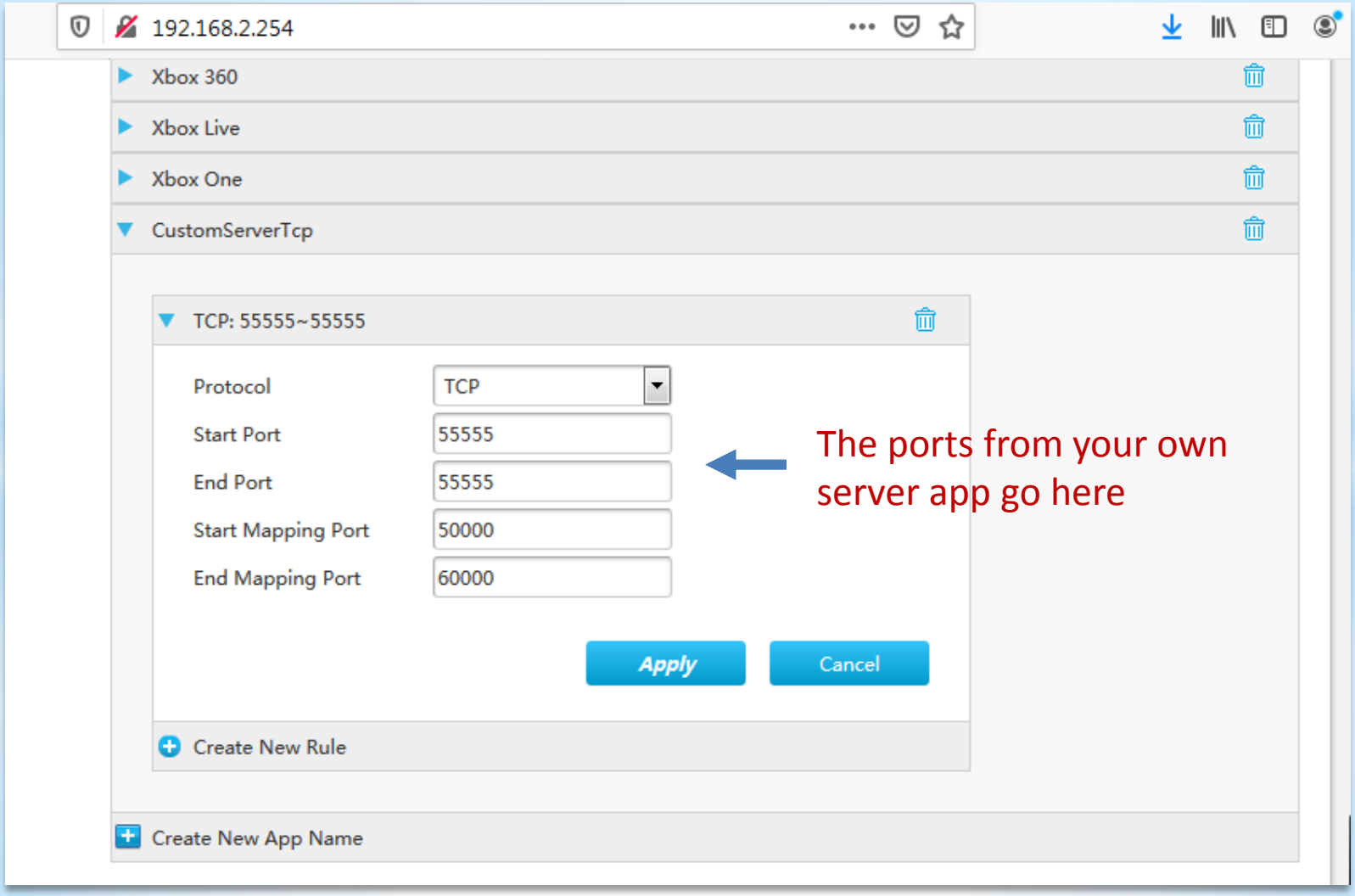
Setting up Port Forwarding

- (Note: this is router specific, but I expect it's similar on your router.)

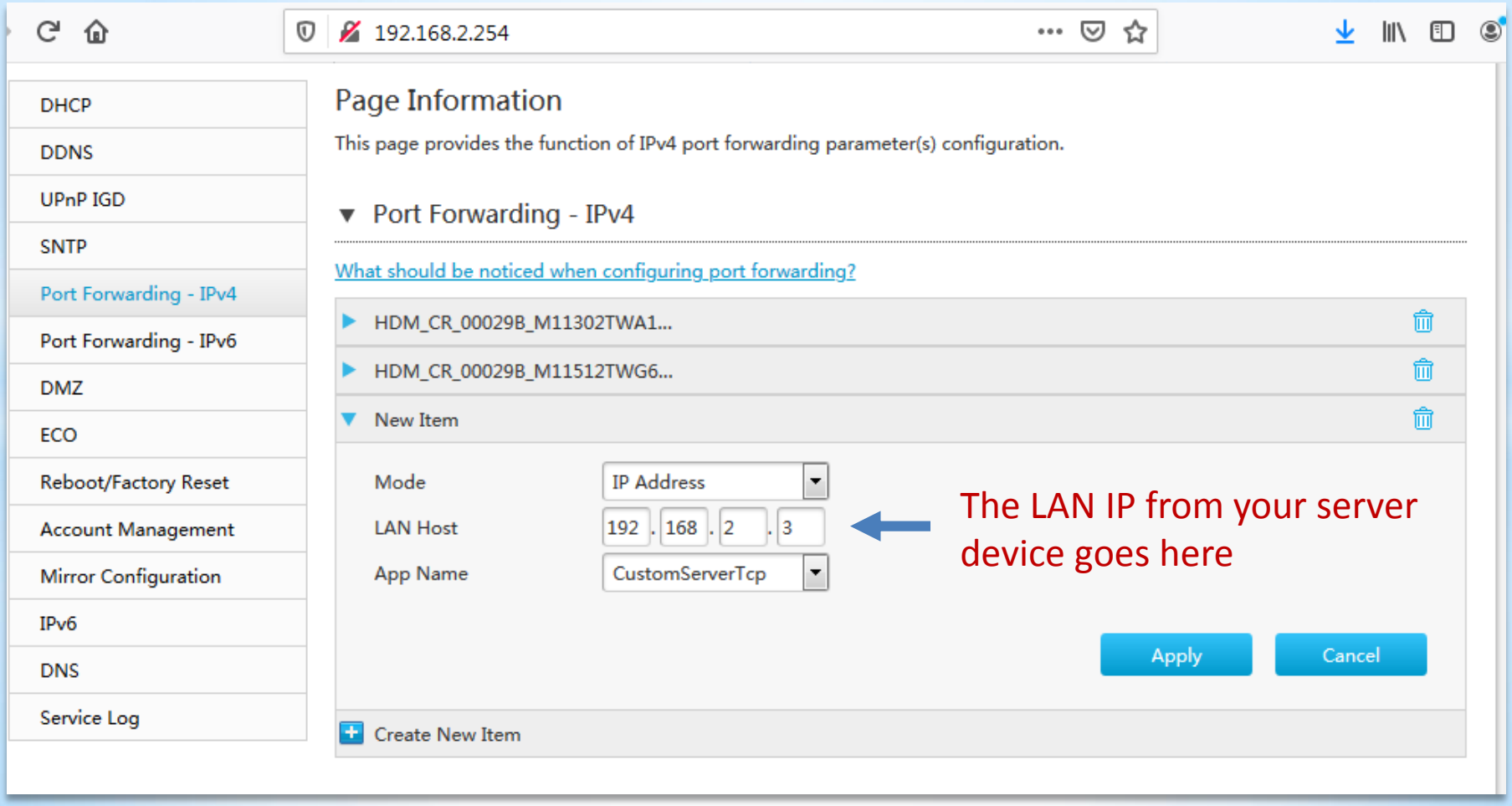
- Log in to your router to change its settings, by typing its LAN address in a browser
  - This router address is the "Default Gateway" that you see from typing *ipconfig* in a terminal

- Log in with your router admin data (if you didn't change it yet: check the password printed on your router?)

Create a Port Forwarding Rule



Create a Port Forwarding Server



- Now you can let your (remote) friend play your game, by
  - Starting your server on port 55555, on the selected device
  - Sending this port number (55555) + your WAN IP to your friend
  - (You can find your WAN IP also somewhere in your router menus, or just google “what is my IP”)

Disclaimer

- Most home routers work the way I described here, but there are also routers that use different NAT conventions
- Especially routers for large corporate or school networks
- …but you don't have their admin passwords anyway ☺ (right?!)

Peer-to-peer and Match Making

Match Making

- Suppose your casual game becomes quite popular among your friends (and their friends)…
- You're happy with this, but you would like to relieve your server from running all of these game instances as an *authoritative server.*
- You just want to host a *match-making server,* such that your friends (and their friends…) can play *peer-to-peer.*

Disclaimer

- Any *competitive game,* with *rankings / leader boards,* or any multiplayer game with *hard earned progress* (e.g. experience points, loot) *must* be hosted on an authoritative server.

- …but casual games that are played just for fun (imagine that!) can be done peer-to-peer.

Match Making

- Players log in to a *match making server* to indicate that they're looking to play a game
- The server matches player pairs or groups (possibly based on geographical location, rankings, player preferences)
- The server forwards the WAN IP + ports from these players to each other.

Problem

- The server forwards the WAN IP + ports from these players to each other:

  "Hey 18.19.20.21 50000 (=A), the player 30.31.32.33 60000 (=B) wants to play with you."

- *Problem:* the router from 30.31.32.33 doesn't accept packets from strangers on the internet. It accepts *replies* from the match making server, but no packets from the stranger 18.19.20.21.
- (…and you don't want to ask casual players to set up port forwarding rules before playing your game.)

Solution

- Solution: *NAT hole punching*

- Idea:
  - Make player B (=30.31.32.33 60000) send a packet first to player A (=18.19.20.21 50000)
  - This packet will drop, because A's router doesn't accept packets from strangers on the internet…
  - …but it does add an entry to the NAT table of Player B's router!!! (→ "hole is punched")
  - Now, if A sends a packet to B, it will arrive!

UDP vs TCP

- This hole punching technique is relatively easy to implement with UDP, because a single UDP client (attached to one local port) can send and receive packets to and from different remote addresses (=the *match making server* and the *other player*).

- …For TCP it's trickier, because TCP is *connection based:* a single socket (local port) belongs to a specific remote IP + port.
- (Recall that UDP & TCP NAT tables are completely separate, so that doesn't help either…)

Sequential TCP Hole Punching

- TCP challenge: a single socket (local port) belongs to a specific remote IP + port.
- Solution:
  - Manually bind a TCP client to a port, connect to match maker
  - After getting an answer, close the TCP client, freeing up the port(!)
  - Then bind a new TCP client to the same port, try to connect to the other player (and know you'll fail).
  - After failing, close the TCP client
  - Finally, start a *TCP listener* on the same port,
  - …get an incoming connection request from the other player,
  - …and start playing!

C# Implementation

- Next: some tips on how to implement this in C#, using TcpClient / TcpListener (instead of low level Sockets)

- General tips:
  - Create a console project for your MatchMaking server
  - Decide on an application protocol (how does the MM server transmit IP + port info?)
  - Extend your Unity chat client from Assignment 2 to add hole punching functionality

Getting EndPoint info (MatchMaker)

```
public class TcpNetworkConnection : NetworkConnection {
    public int LocalPort {
        get {
            return ((IPEndPoint)socket.Client.LocalEndPoint).Port;
        }
    }
    public IPEndPoint RemoteEndPoint {
        get {
            return (IPEndPoint)socket.Client.RemoteEndPoint;
        }
    }

    TcpClient socket;
```

Refresh Socket, Bind to LocalPort

```
void DisposeTcpClient() {
    if (_listener!=null) {
        _listener.Stop();
        _listener=null;
    }
    if (_client!=null) {
        _client.Close();
        _client=null;
    }
}

void RefreshTcpClient() {
    _panelWrapper.AddOutput("Refreshing tcp client");
    DisposeTcpClient();
    _client=new TcpClient(new IPEndPoint(IPAddress.Any, _localPort));
}
```

Basic Hole Punching

```
static TcpListener PunchHole(string remoteIP, int remotePort, int localPort) {
    var remote_endpoint = new IPEndPoint(IPAddress.Parse(remoteIP), remotePort);
    var local_endpoint = new IPEndPoint(IPAddress.Any, localPort);

    Console.WriteLine("Punching hole...");
    // The "using" statement ensures that the underlying socket is disposed properly, such that
    // we can subsequently start a listener on that port.
    using (var outgoing = new TcpClient(local_endpoint))
    {
        try
        {
            outgoing.Connect(remote_endpoint);
            Console.WriteLine("Made outgoing connection. Exiting.");
            // This (server) code sample assumes that this never happens, there's always an exception triggered.
            return null;
        }
        catch (Exception ex)
        {
            Console.WriteLine("Got an exception, as expected:\n{0}\n{1}",ex.GetType(),ex.Message);
        }
    }
    Console.WriteLine("Hole should be punched");
    var listener = new TcpListener(local_endpoint);
    listener.Start();
    return listener;
}
```

Timing

- The TcpClient.Connect call can take a lot of time before it fails! (~5 to 10 seconds)
- Only afterwards, the listener is started
- That's the time that the other client has to wait before making a connection attempt!
- Keep this into account in your application!

Improved Timing

- Using the C# TcpClient, the only way to end a TcpClient connection attempt early is to call *ConnectAsync* instead (which starts another thread), and close the TcpClient before that thread ends (fails).

- Alternatively, use a lower level *Socket*, and set *ExclusiveAddressUse = false* before Binding it.
- Then you can start a listener on the same port at the same time.

→ All of this is pretty advanced! Get the basics working first!

General Implementation Hints

- Include Debug info for every step!
- Always print the Remote and Local EndPoint info

Testing

- Theory and implementation hints are nice, but how can you actually *test this at home*?! (Behind your single router, with devices available to the average student…)
- Option 1: guess or borrow your neighbor's Wifi password ☺
- Option 2: Use Unity to build your client for your phone, and use your 4G data bundle.

Possible Testing Setup

- Run your custom MatchMaking server, and set up port forwarding on your router
- Run one client on your computer, but don’t forget to use your WAN IP (instead of localhost) to connect to the MatchMaker!
- Run your Unity phone (Android) build on your phone, using 4G instead of Wifi.

Other Concerns

- In practice, it might be that the two clients you want to match are behind the same router, in the same LAN.
- In that case, the matchmaker should forward their LAN IP + ports to each other!
- So the clients should also send their LAN IP + port info to the server, with their match request.
- The matchmaker can often recognize this situation if the WAN IPs are the same.
- …in case of “multilayered NATs” this is trickier → out of scope for now

More Information

- An excellent and accessible article:

  Ford, Srisuresh, Kegel, Peer-to-peer communication across network address translators, 2005 USENIX annual technical conference, 2005.

- A good introduction to the (simpler) UDP case:

  Glazer, Madhav, Multiplayer Game Programming, Addision-Wesley, 2016.

  (Chapter 2)