Networking in Multiplayer Games



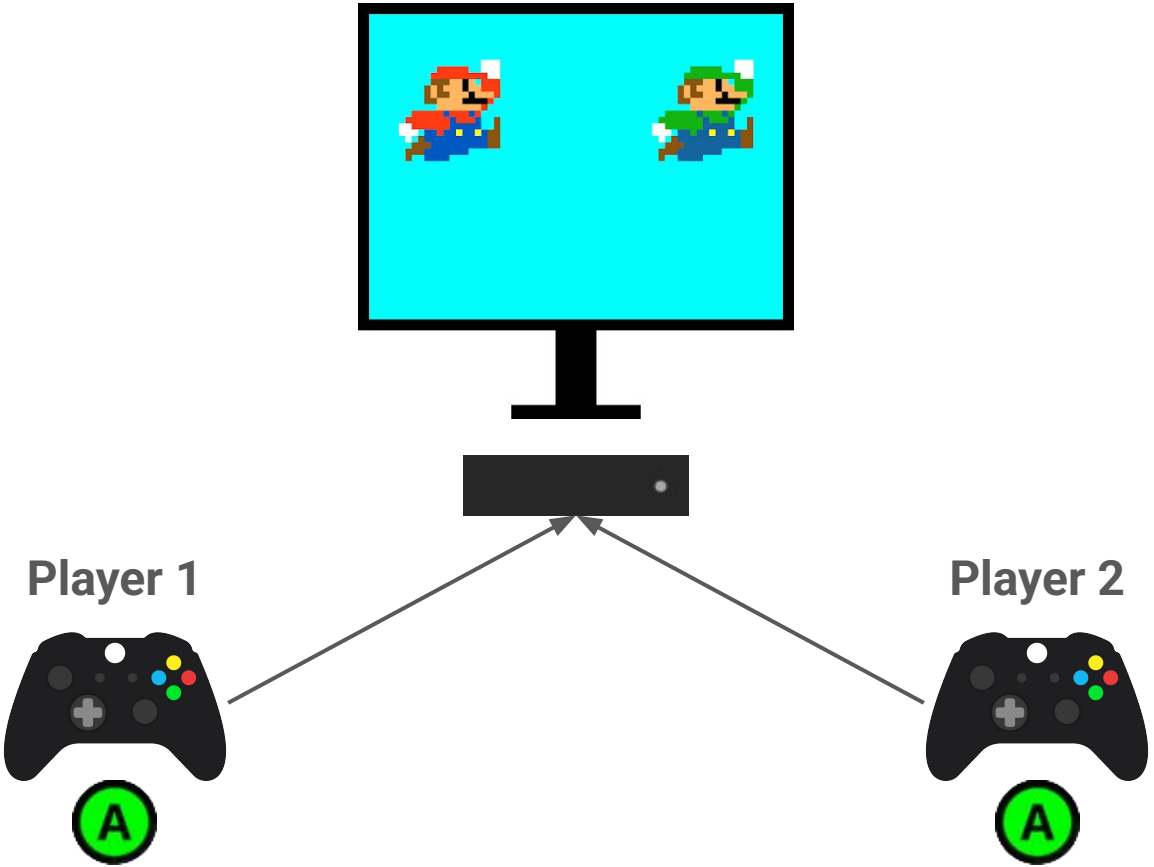


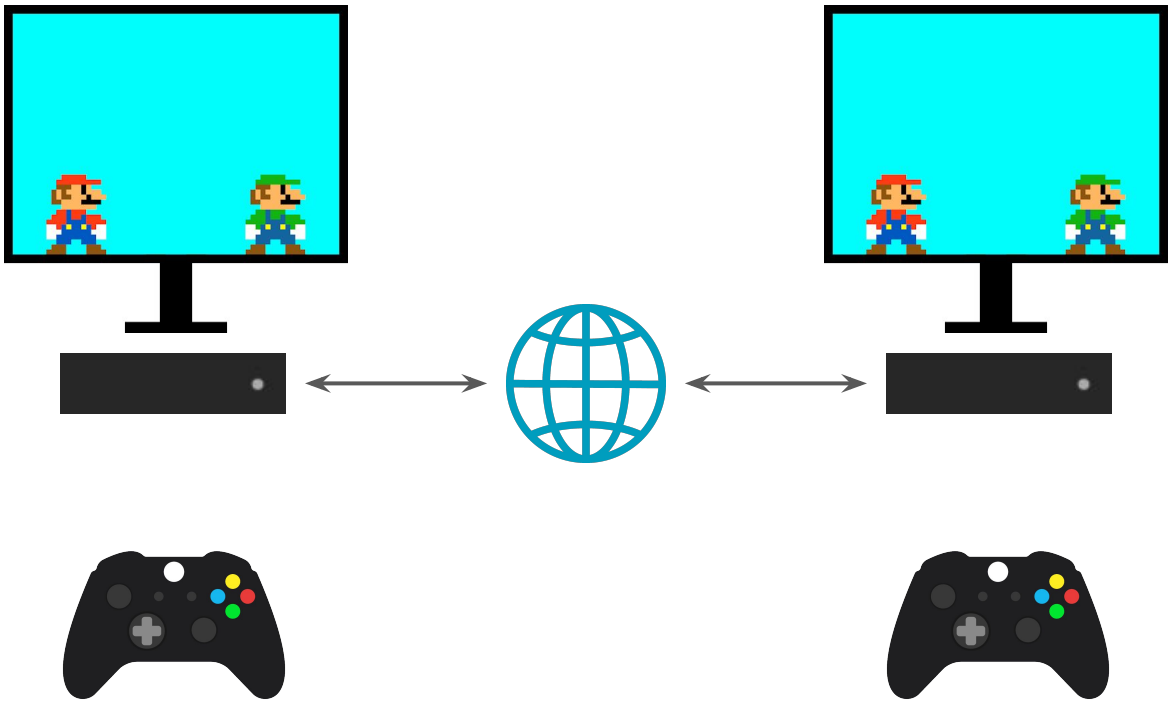
The three parts to this presentation:

1. Why adding multiplayer to games is easy
2. A quick demo of a multiplayer game
3. Why I was lying to you in part 1

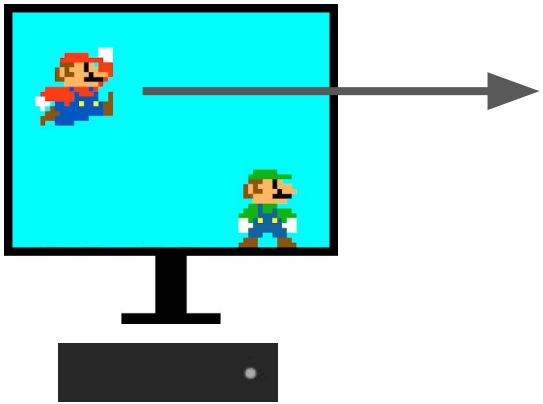
The three parts to this presentation:

1. **Why adding multiplayer to games is easy**
2. A quick demo of a multiplayer game
3. Why I was lying to you in part 1


Player 1
Player 2
A
A



Event-based system

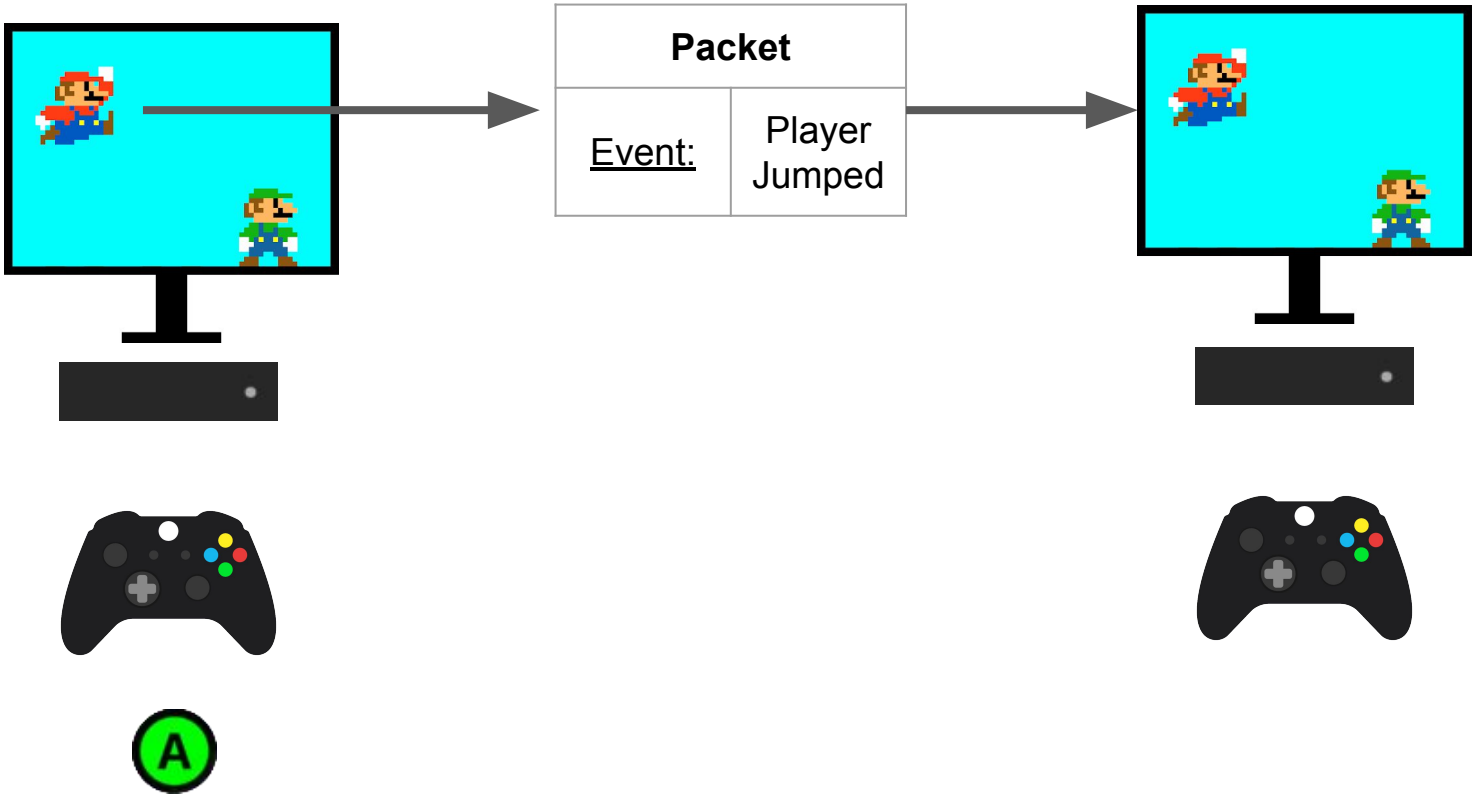




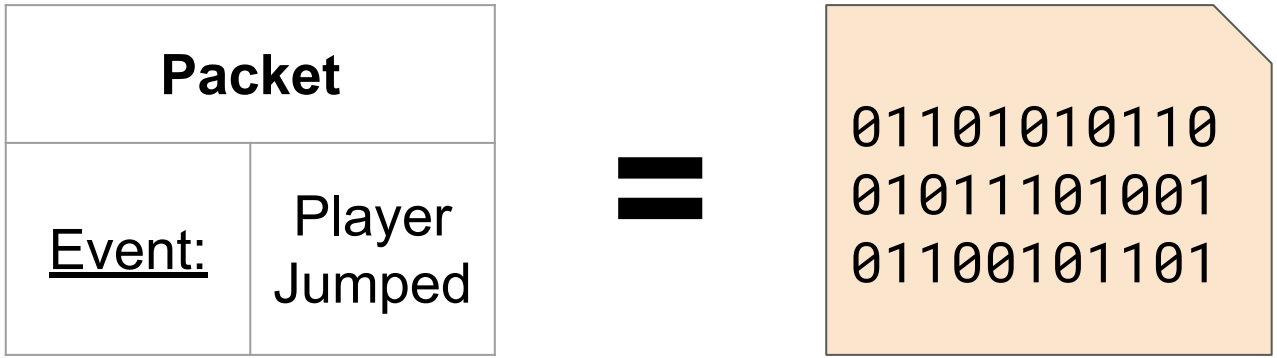


Packet	
Event:	Player Jumped

Event-based system



How should the clients send data to each other?



12.34.56.78

31.41.59.26

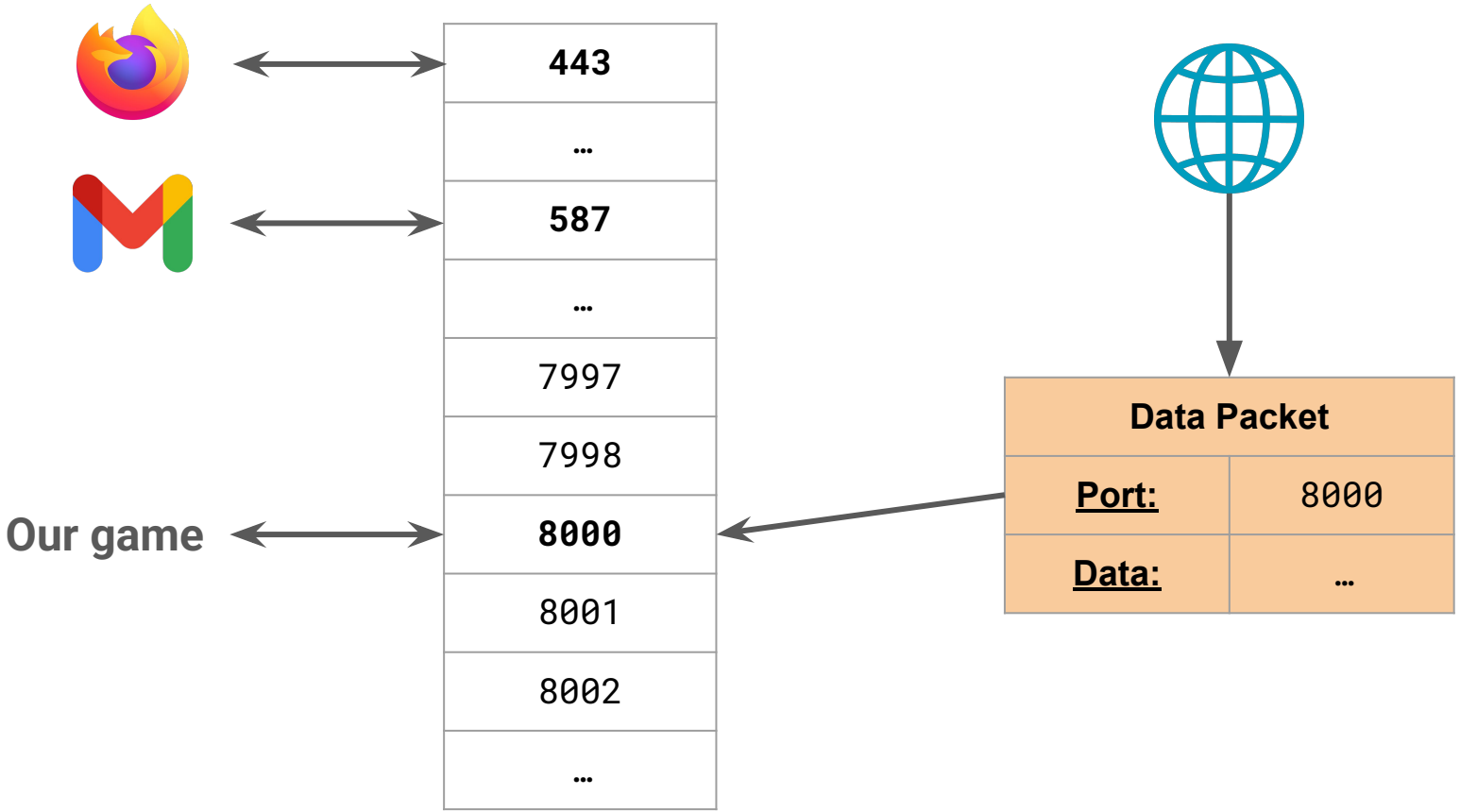


12.34.56.78



A program sends and receives data on specific **ports**.

Ports: (0-65535)					
	…	**2388**	**2389**	**2390**	**…**


443
…
587
…
7997
7998
8000
8001
8002
…
Our game
Data Packet
Port:
8000
Data:
…

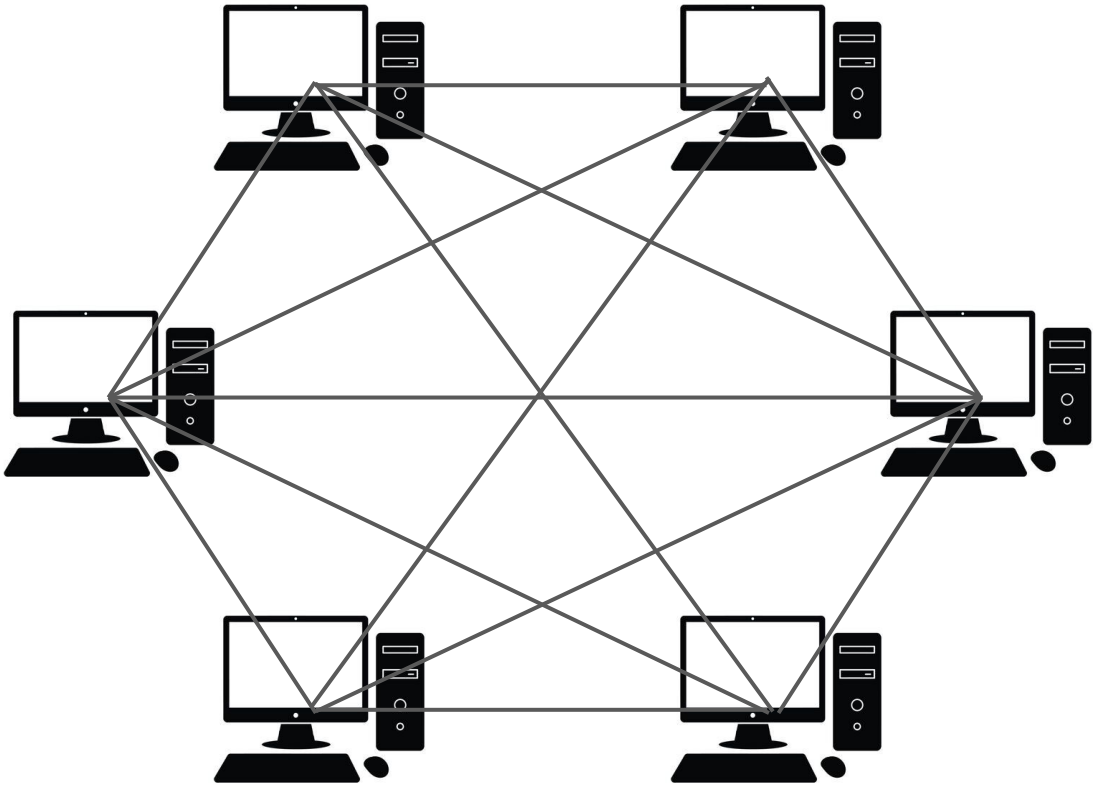
Peer-to-Peer (P2P)

IP: 12.34.56.78
Port: 3000

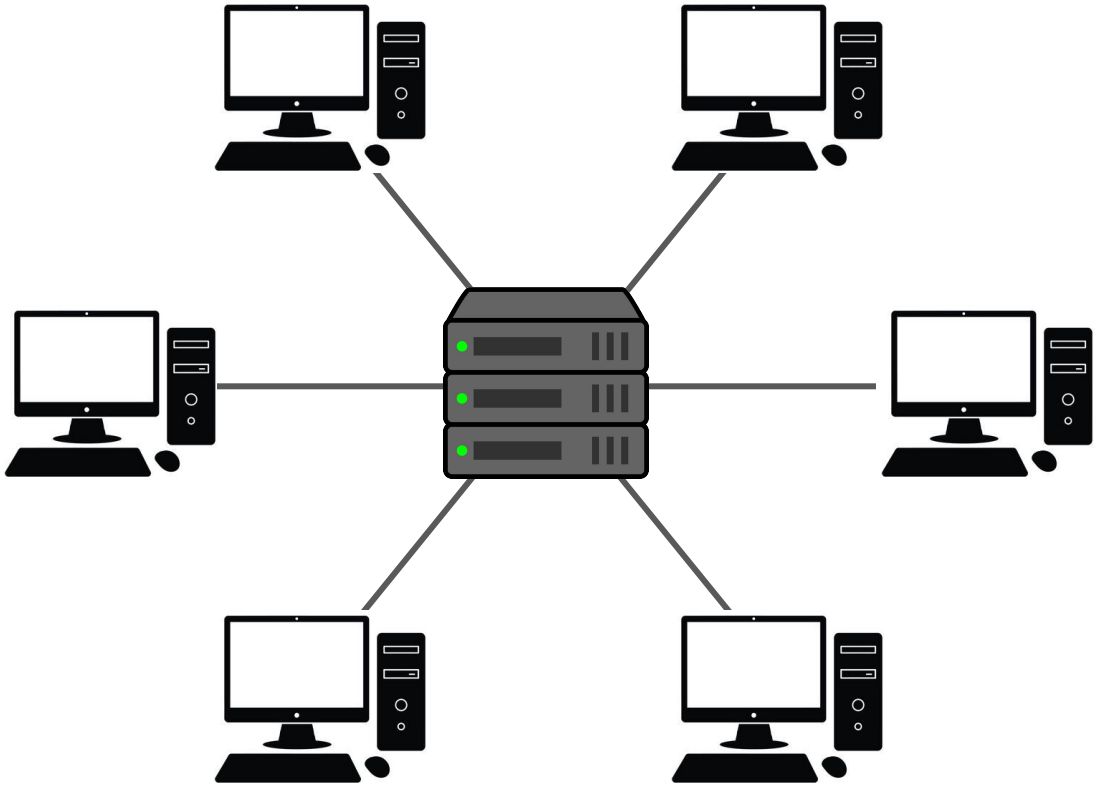
IP: 31.41.59.26
Port: 3000



Peer-to-Peer (P2P)



Client-Server



Transport Protocols

TCP vs UDP



TCP

Packet 1

```
0110110
1001011
0100101
```

Packet 2

```
1011010
0110110
1001011
```

Packet 3

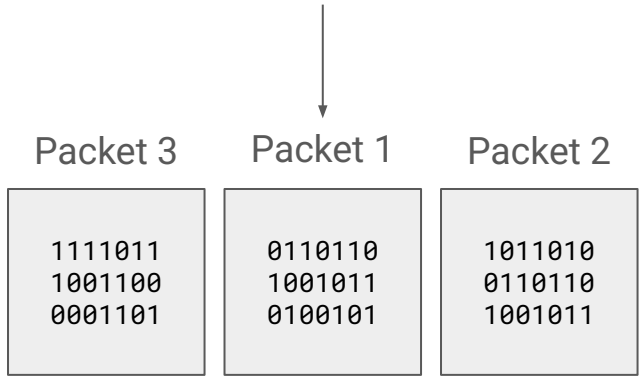
```
1111011
1001100
0001101
```





TCP

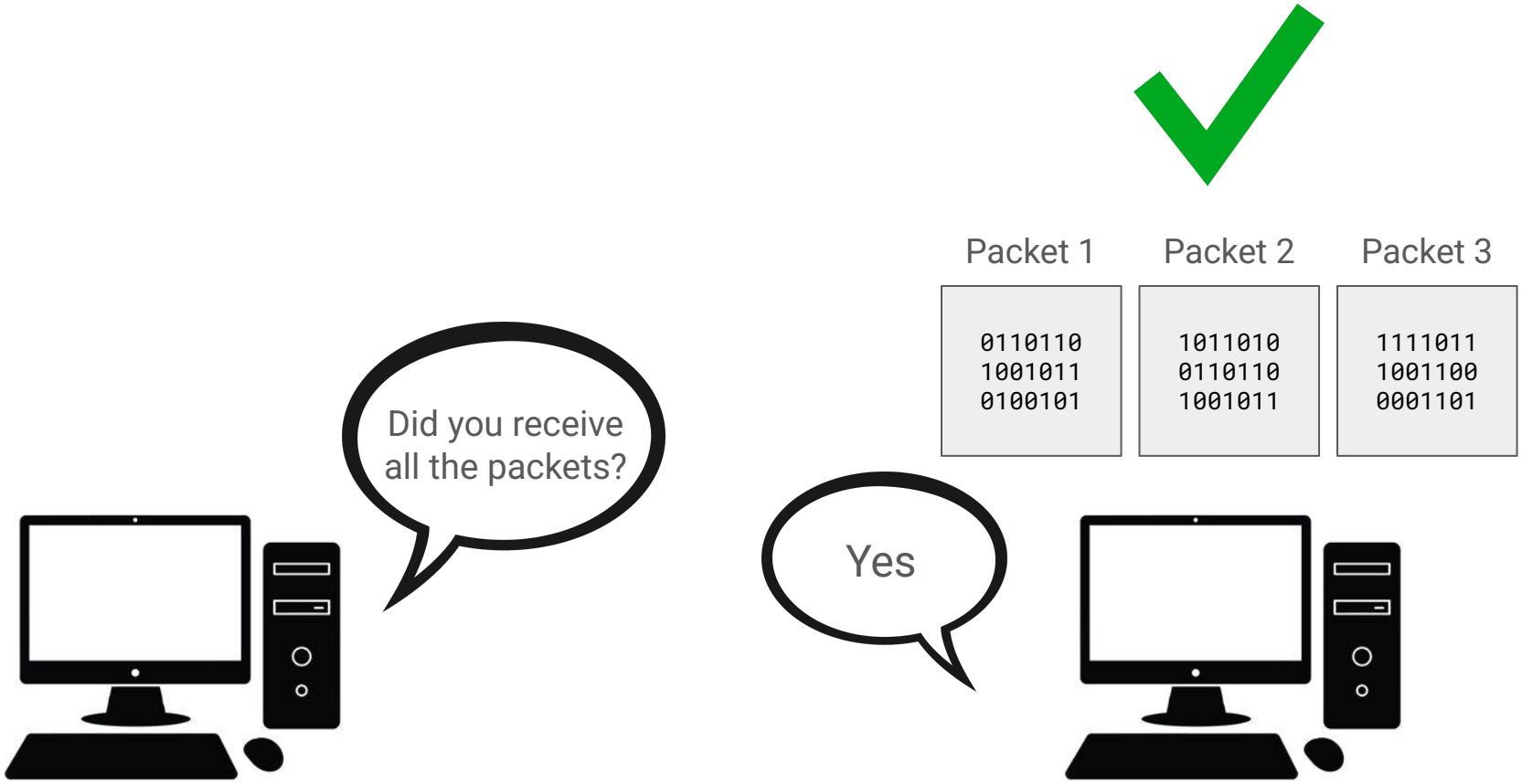
Wrong order!



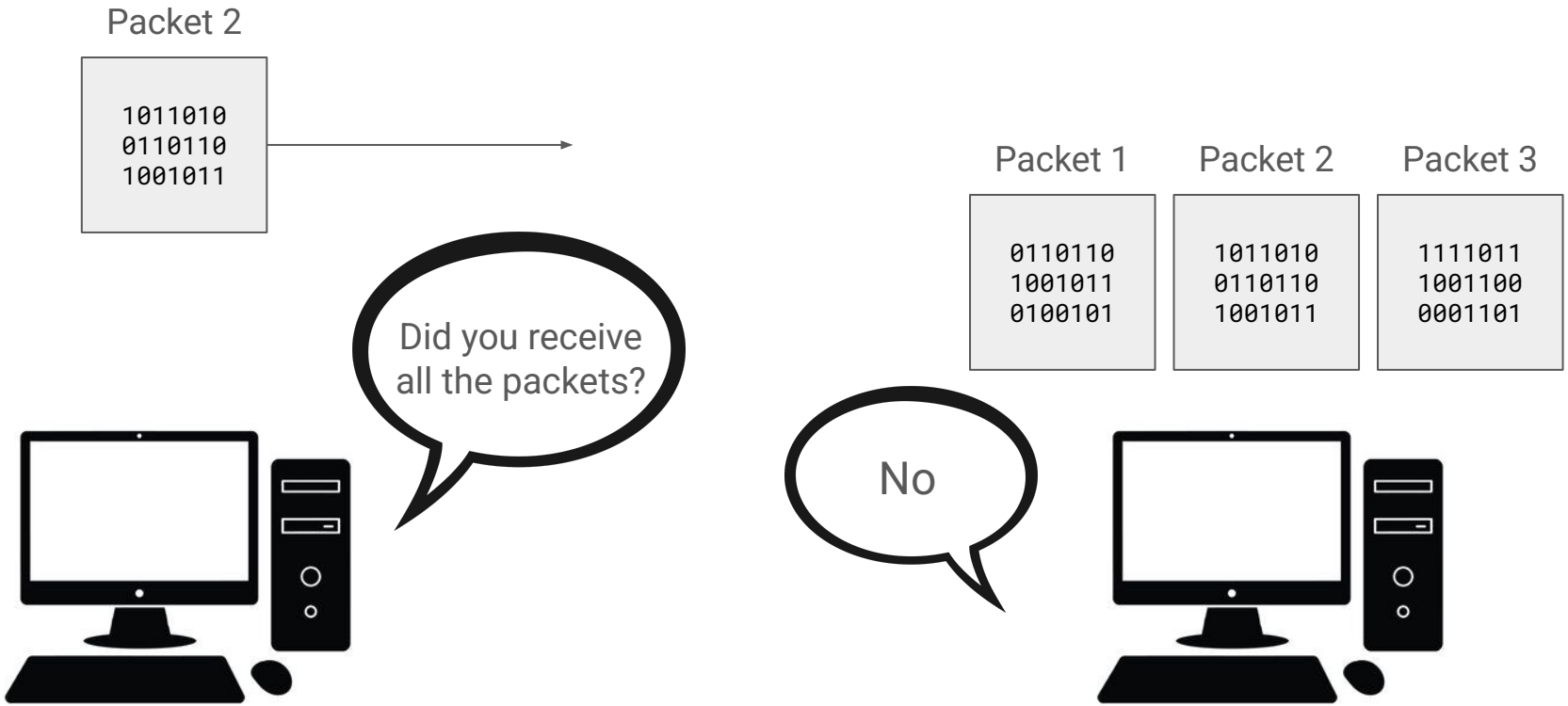




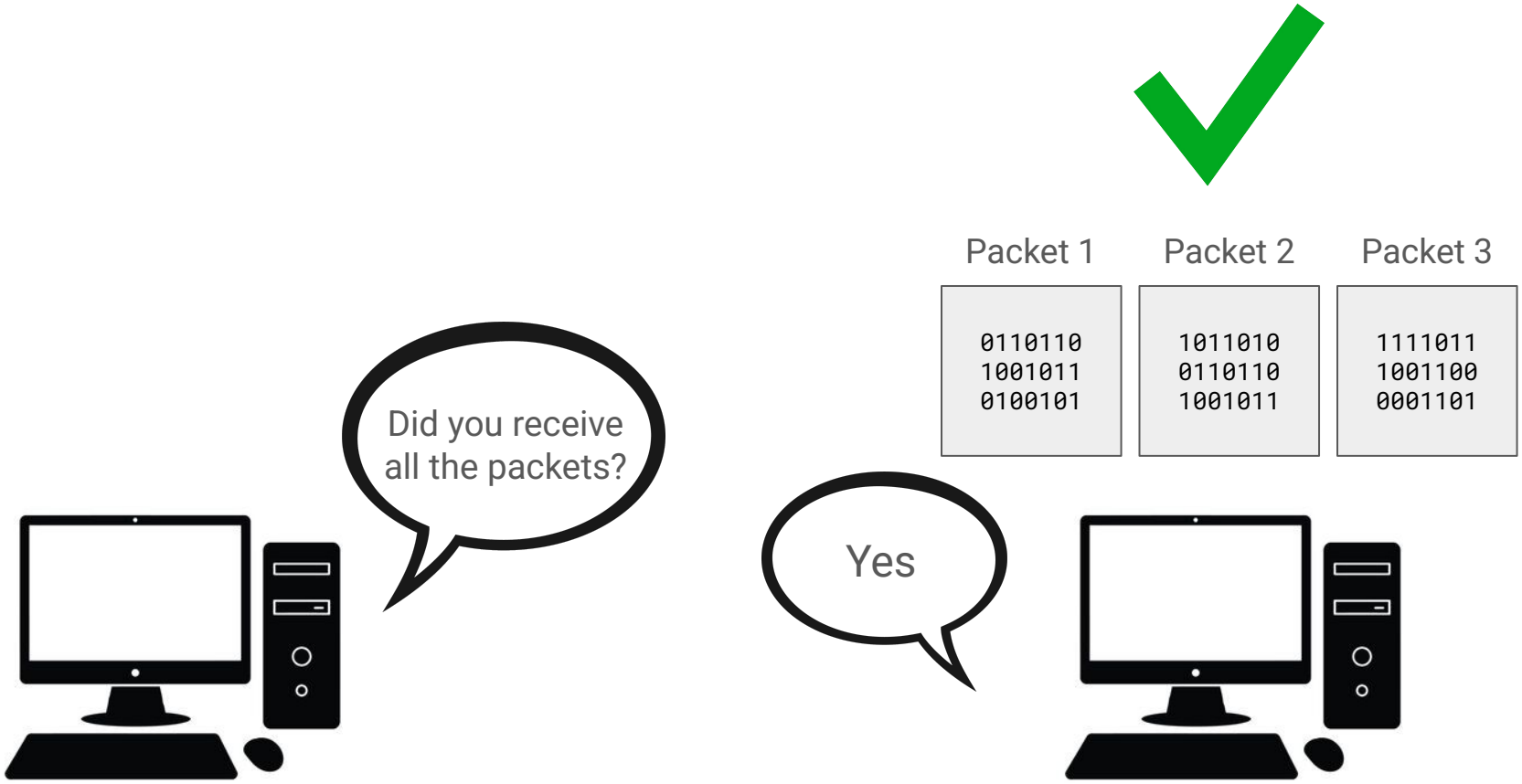
TCP


Packet 1
Packet 2
Packet 3
0110110
1001011
0100101
1011010
0110110
1001011
1111011
1001100
0001101
Did you receive all the packets?
Yes

TCP

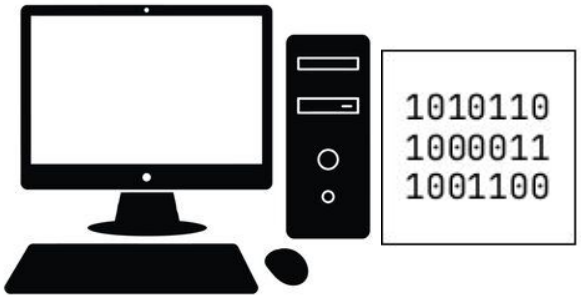

Packet 2
1011010
0110110
1001011
Packet 1
0110110
1001011
0100101
Packet 2
1011010
0110110
1001011
Packet 3
1111011
1001100
0001101
Did you receive all the packets?
No

TCP


Packet 1
Packet 2
Packet 3
0110110
1001011
0100101
1011010
0110110
1001011
1111011
1001100
0001101
Did you receive all the packets?
Yes

UDP

- No regard for order
- No regard for whether the packets arrive or not
- If packets get corrupted, they just get discarded







TCP vs UDP

This can take a long time!

Packet 1

```
0110110
1001011
0100101
```

Packet 2

```
1011010
0110110
1001011
```

Packet 3

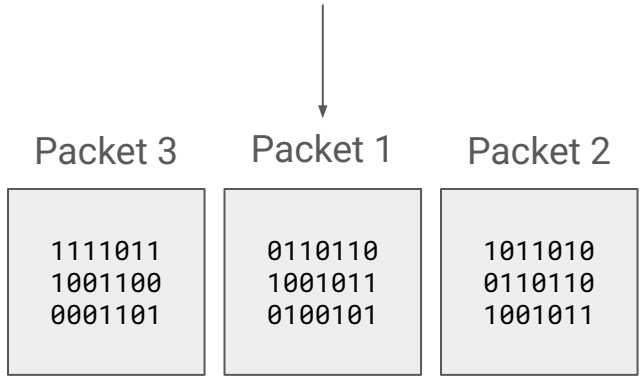
```
1111011
1001100
0001101
```





This can take a long time!

Wrong order!



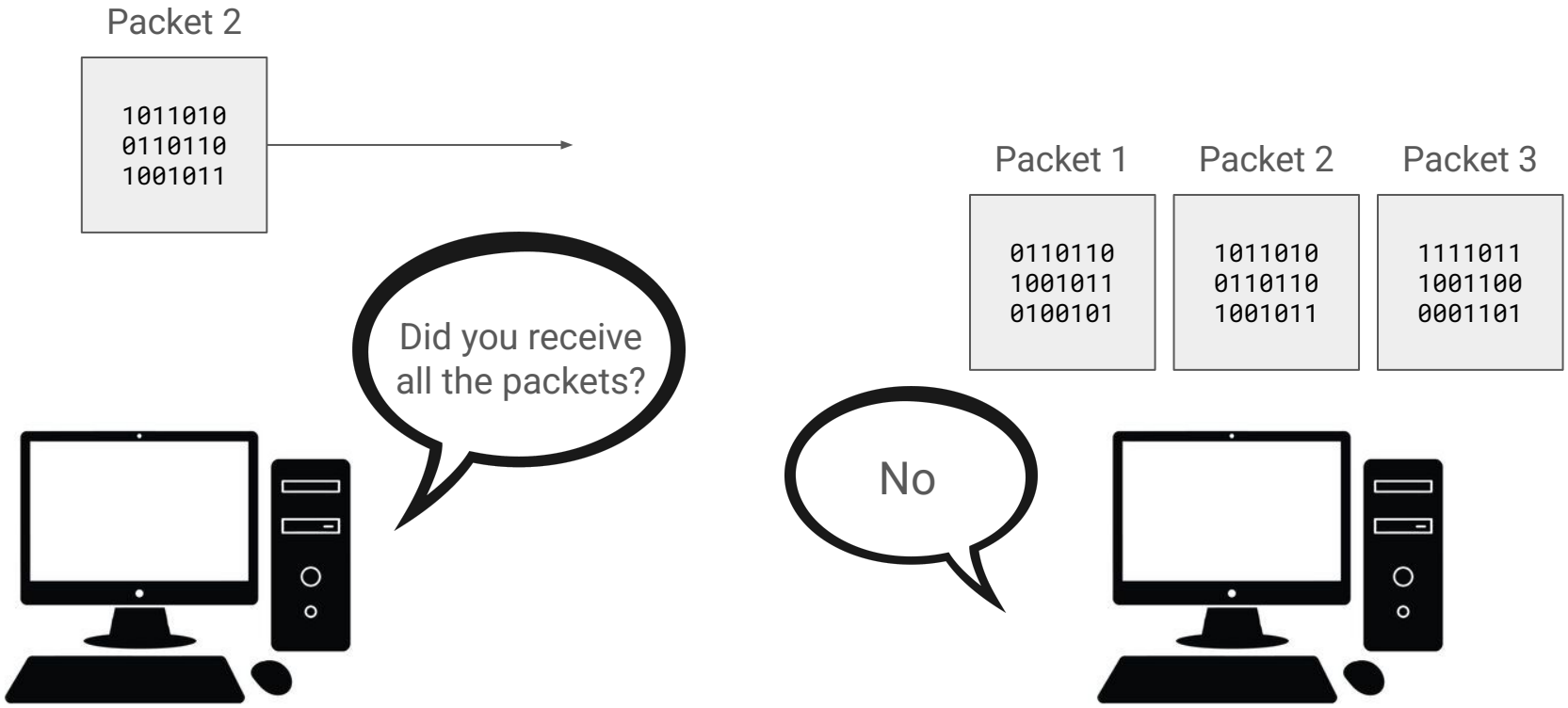




This can take a long time!



This can take a long time!



This can take a long time!

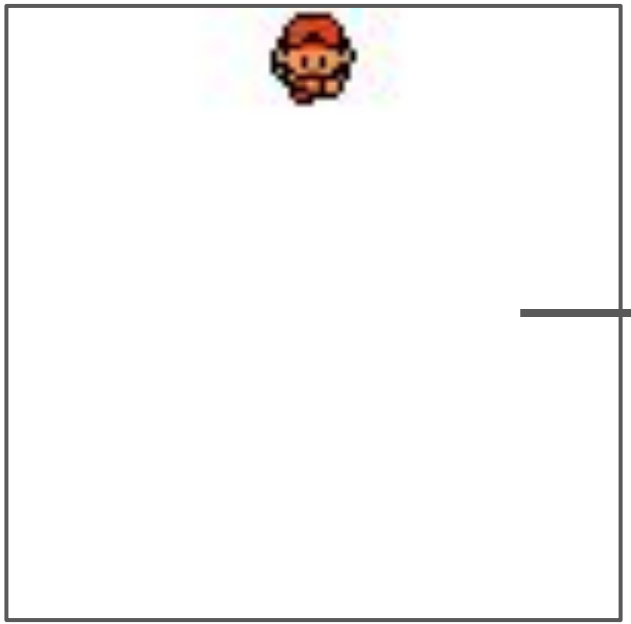




TCP

All these could be waiting to be processed

Packet 4	Packet 5	Packet 6	Packet 7	Packet 8
0110110 1001011 0100101	1011010 0110110 1001011	1111011 1001100 0001101	0110110 1001011 0100101	1011010 0110110 1001011



Packet	
Event:	Player Moved
Data:	X, Y

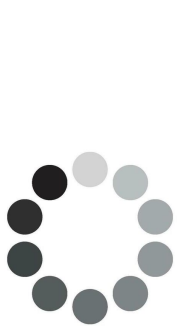
60 FPS

This is a lot of packets!

Benefits of TCP:

- All packets are guaranteed to arrive.
- All packets will be received in order.

We don't care about this one anymore!



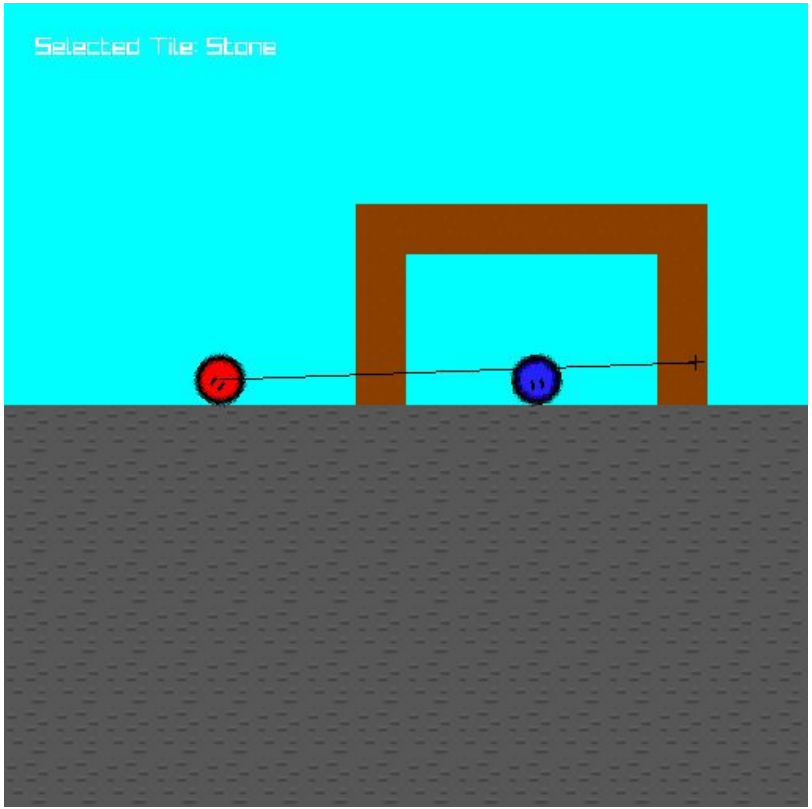
Packet 1	
Event:	Player Moved
Data:	X, Y

Packet 2	
Event:	Player Moved
Data:	X, Y

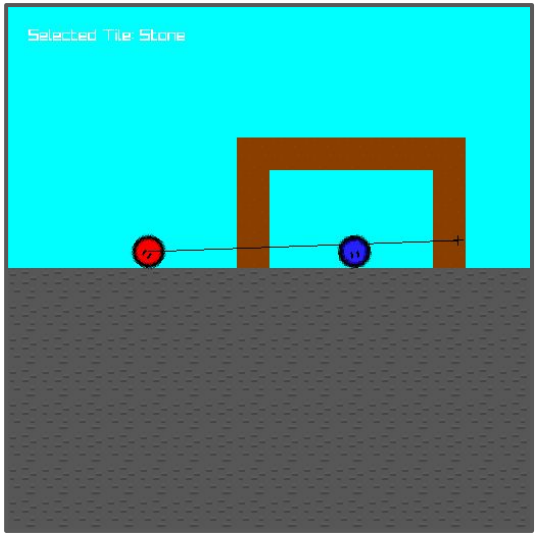
Packet 3	
Event:	Player Moved
Data:	X, Y

Packet 4	
Event:	Player Moved
Data:	X, Y

A simple P2P multiplayer demo



Player 1



Actions
• Move around the map • Place/remove blocks

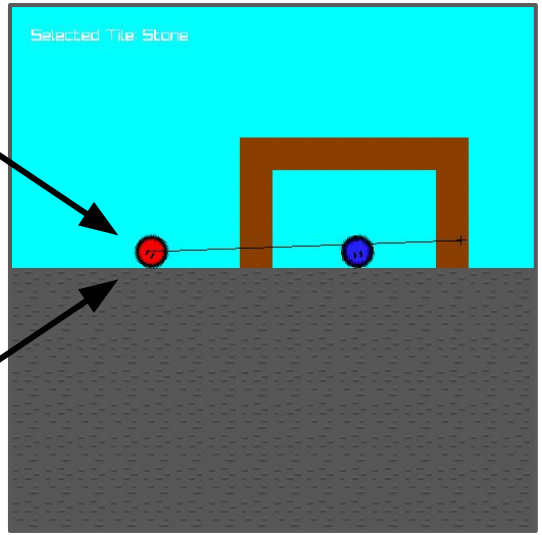
Packet	
Event:	Player Moved
Data:	**PlayerID**, X, Y

Packet	
Event:	Block Changed
Data:	**PlayerID**, X, Y, Tile Type

Packet	
Event:	Player Moved
Data:	**PlayerID**, X, Y

Packet	
Event:	Block Changed
Data:	**PlayerID**, X, Y, Tile Type

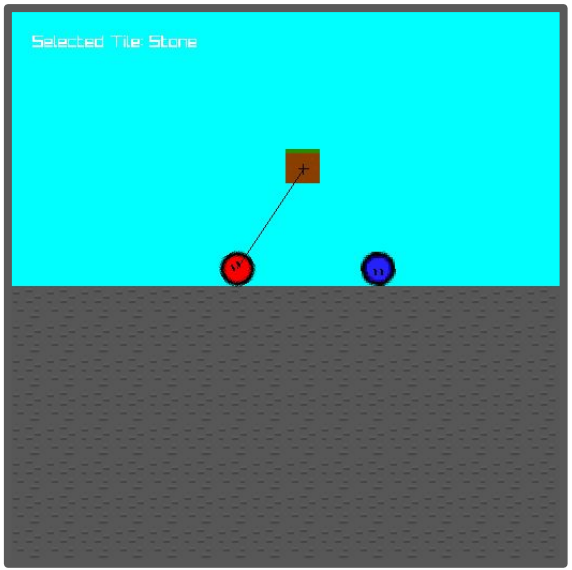
Player 2



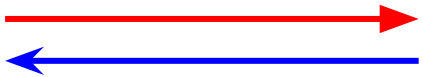
Are there any problems with this system?

What if two players try to place different blocks at the same place at the same time?

Player 1

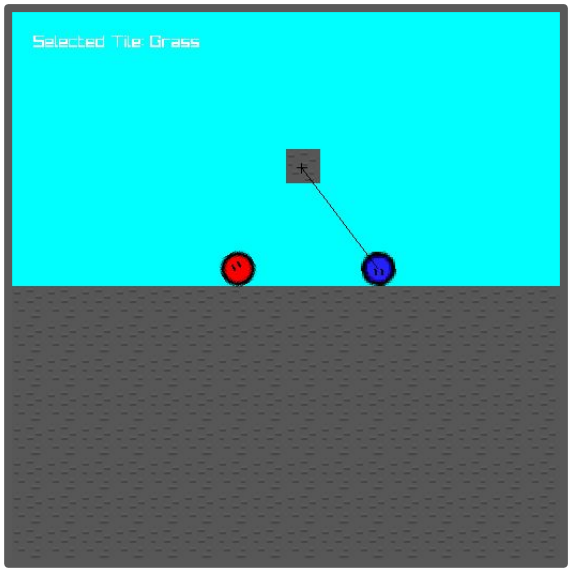


Packet	
Event:	Block Changed
Data:	PlayerID = 0 X = 8 , Y = 4 Tile Type = 1



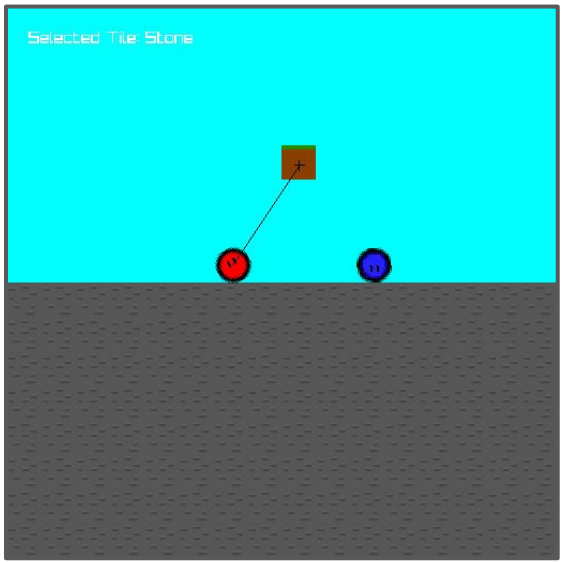
Packet	
Event:	Block Changed
Data:	PlayerID = 1 X = 8 , Y = 4 Tile Type = 3

Player 2

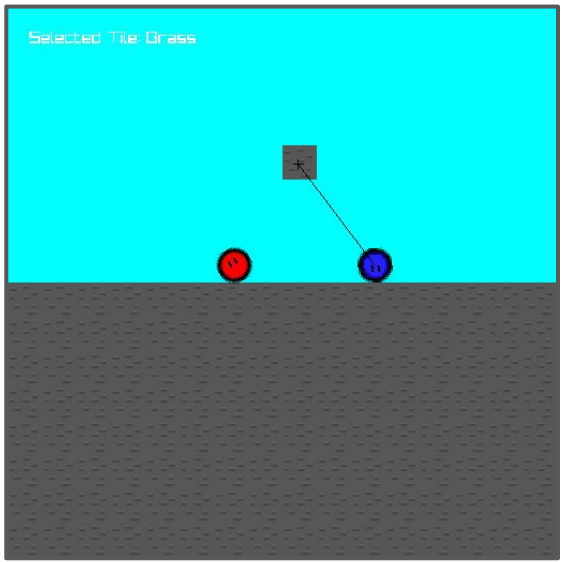


The players have gone out of sync!

Player 1



Player 2

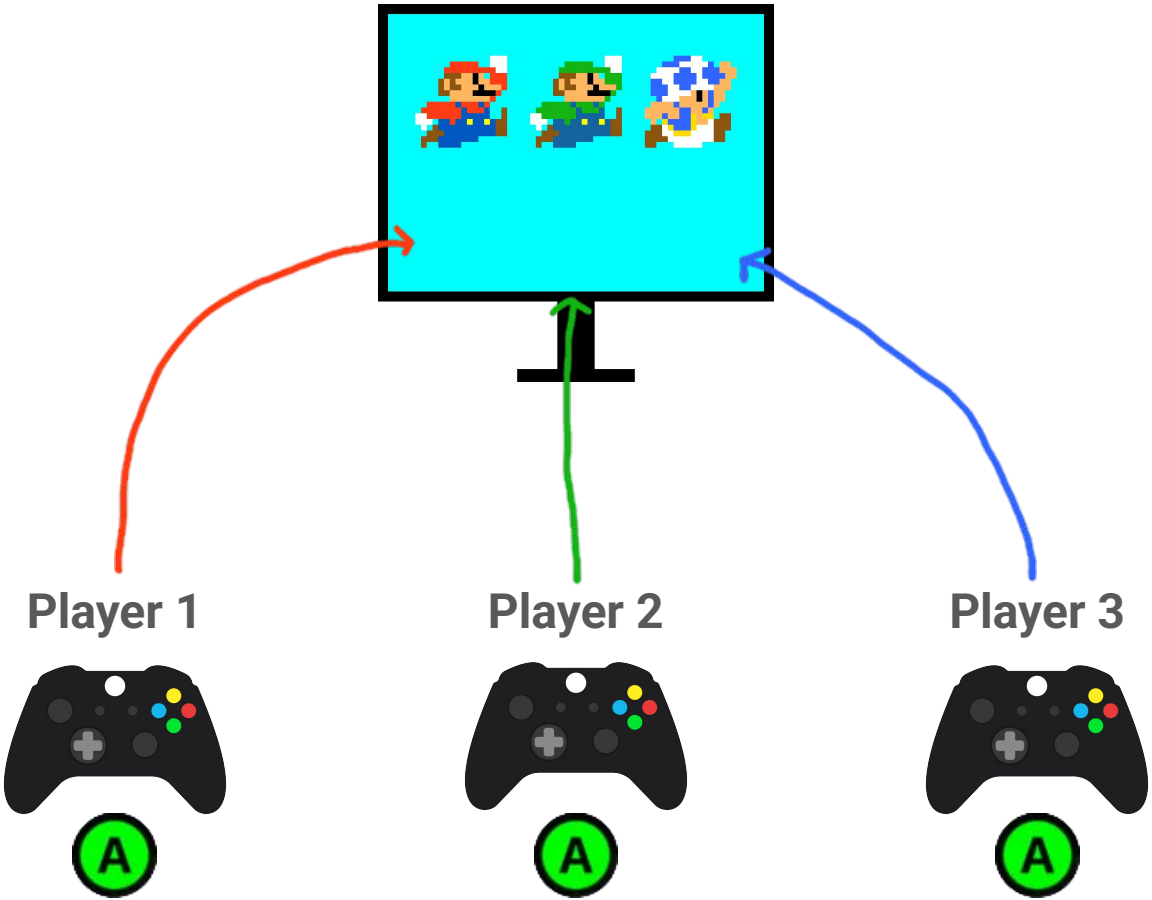


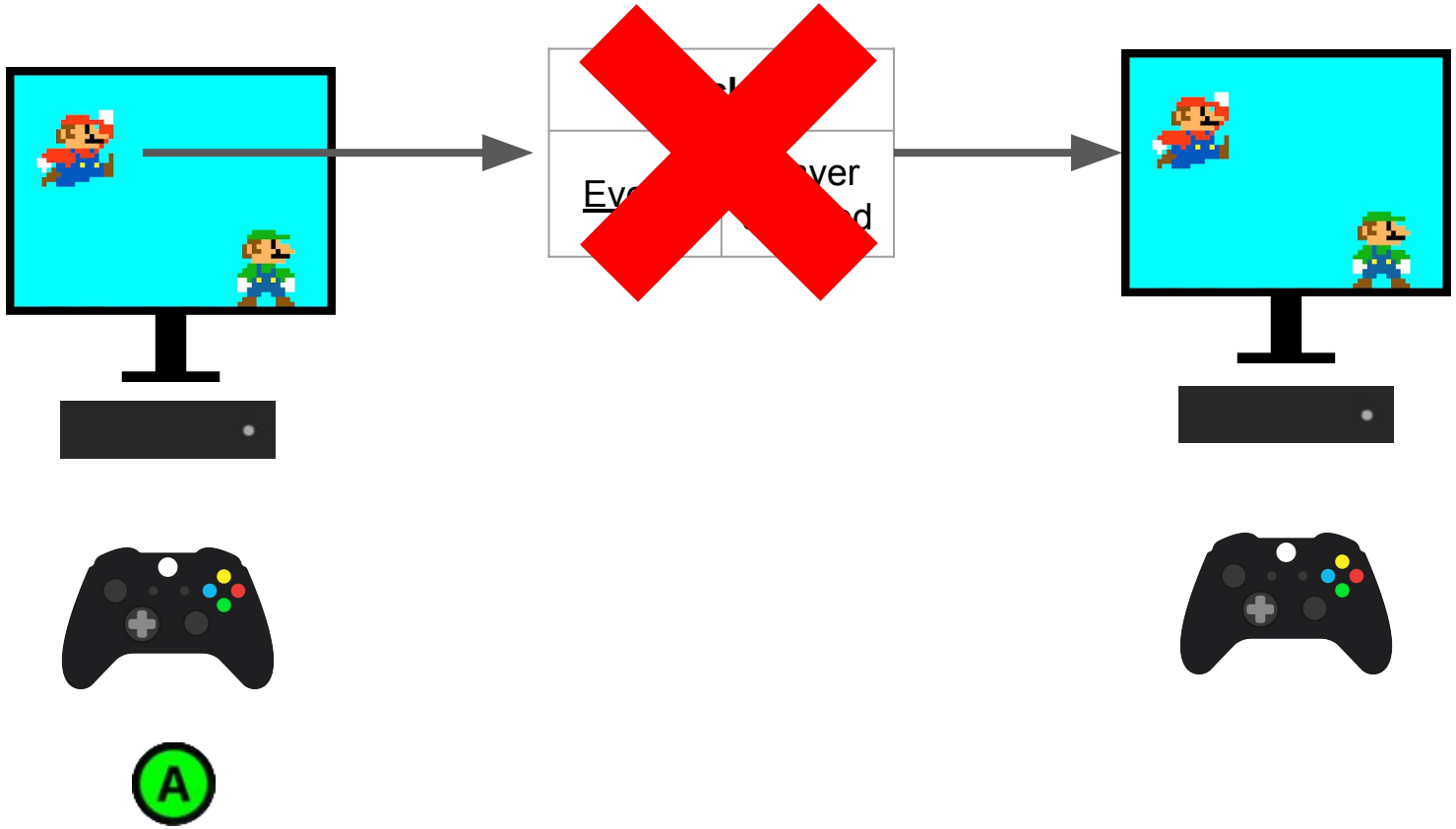
Netcode Architectures

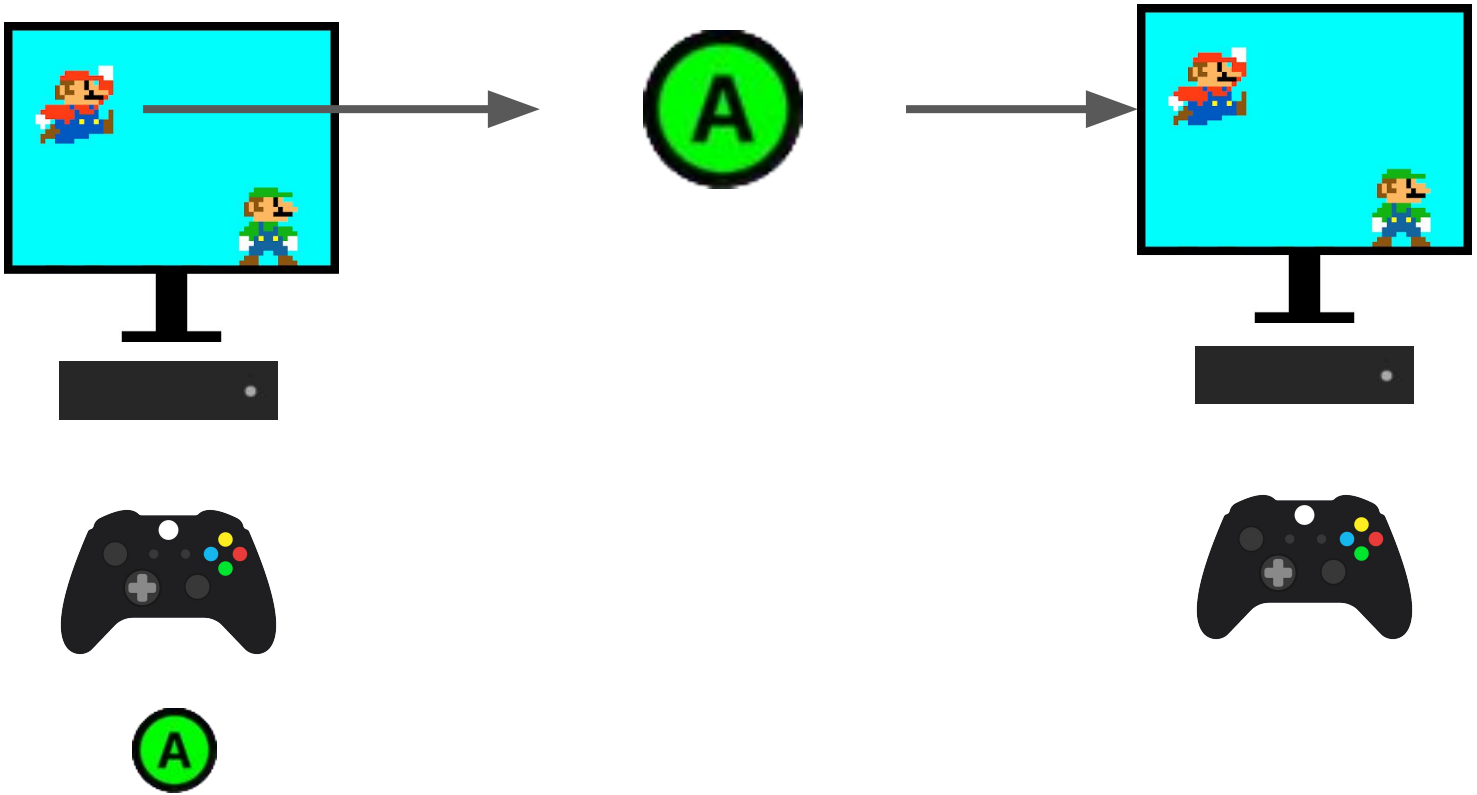
- Lockstep
- Rollback
- Snapshot Interpolation

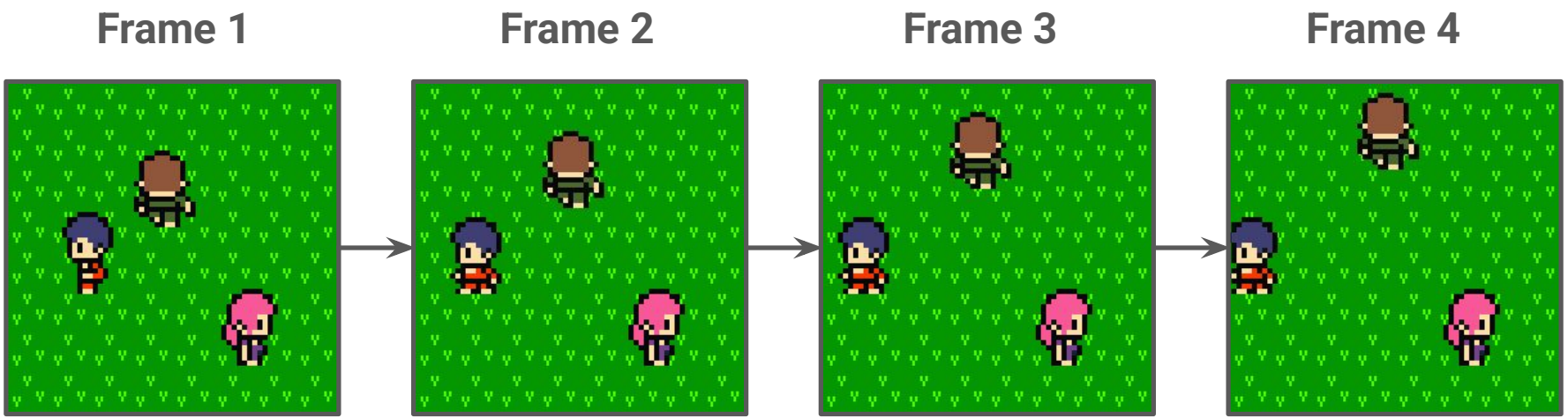
Netcode Architectures

- **Lockstep**
- Rollback
- Snapshot Interpolation


Player 1
Player 2
Player 3
A
A
A


A

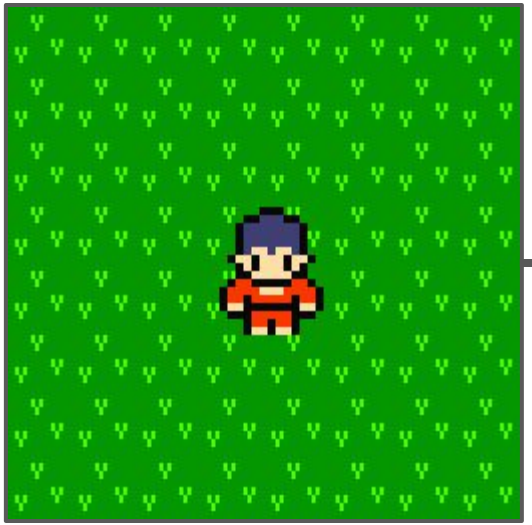

A
A



60 FPS

Always deterministic!

Frame 1

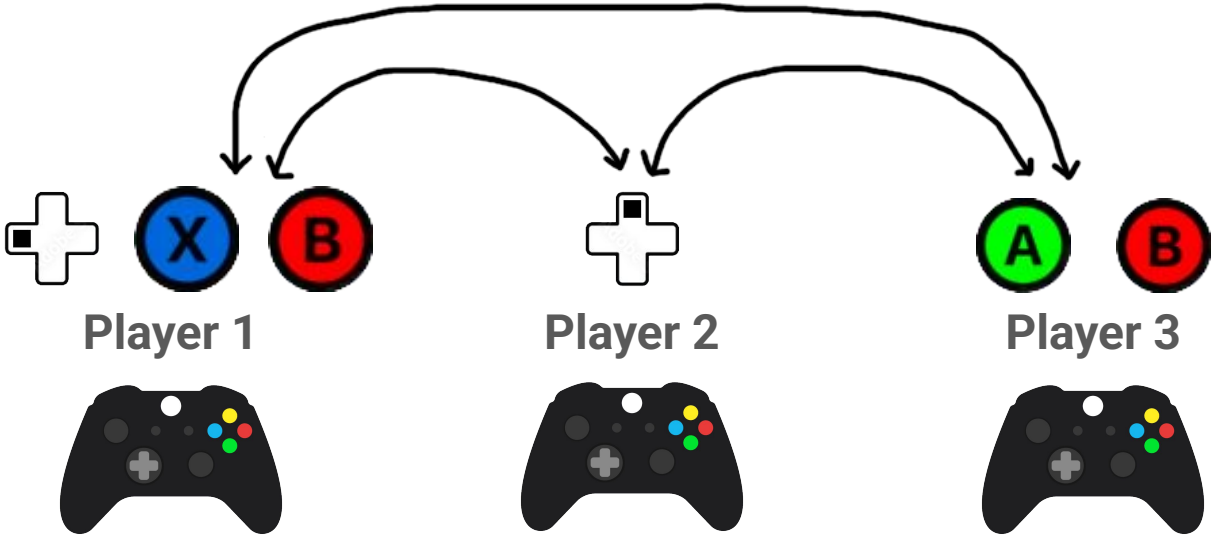




Frame 2



Lockstep Netcode



P1	X	B	
P2			
P3	A	B	

P1	X	B	
P2			
P3	A	B	

P1	X	B	
P2			
P3	A	B	

Player 1



P1	X B
P2	
P3	A B



Player 2



P1	X B
P2	
P3	A B



Player 3



P1	X B
P2	
P3	A B



Lockstep Netcode

Pros

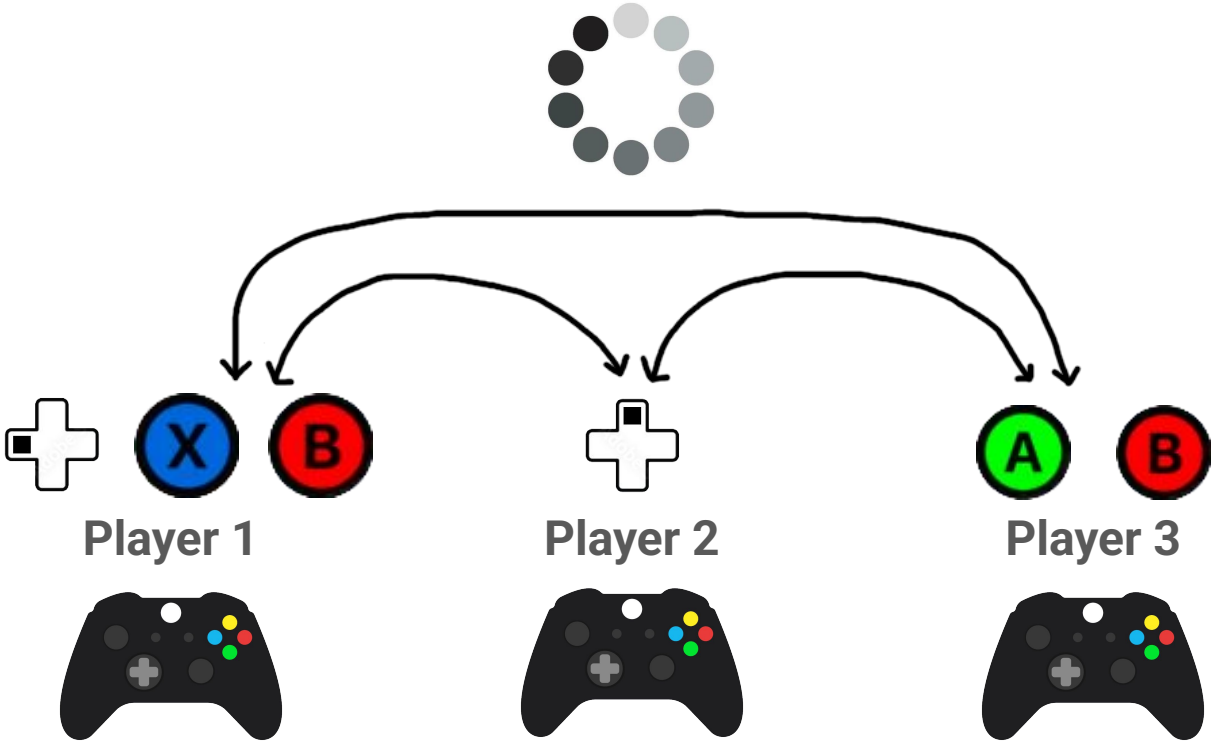
- Doesn’t really require any networking in the game code
  - Can be added to a pre-existing code pretty easily
- Anti-cheat is practically built in by design
- Doesn’t use much bandwidth

Cons

What if just one person has a bad connection?

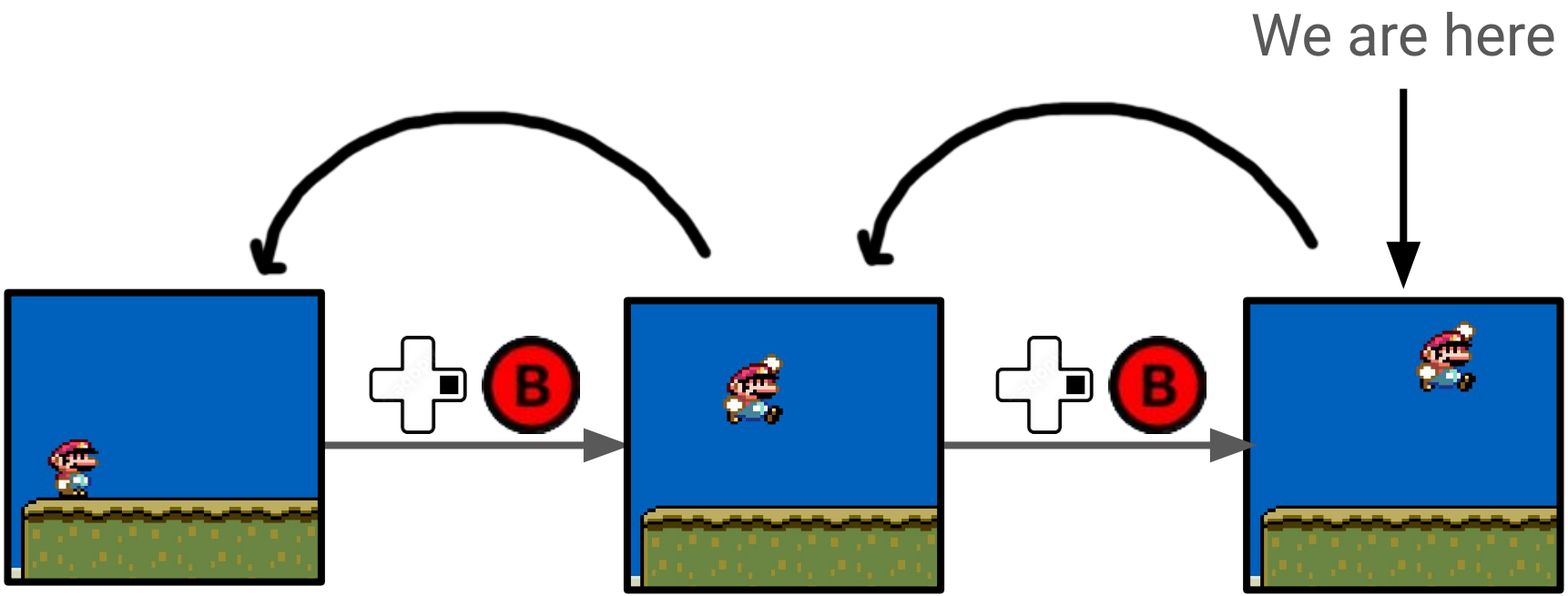
5 FPS

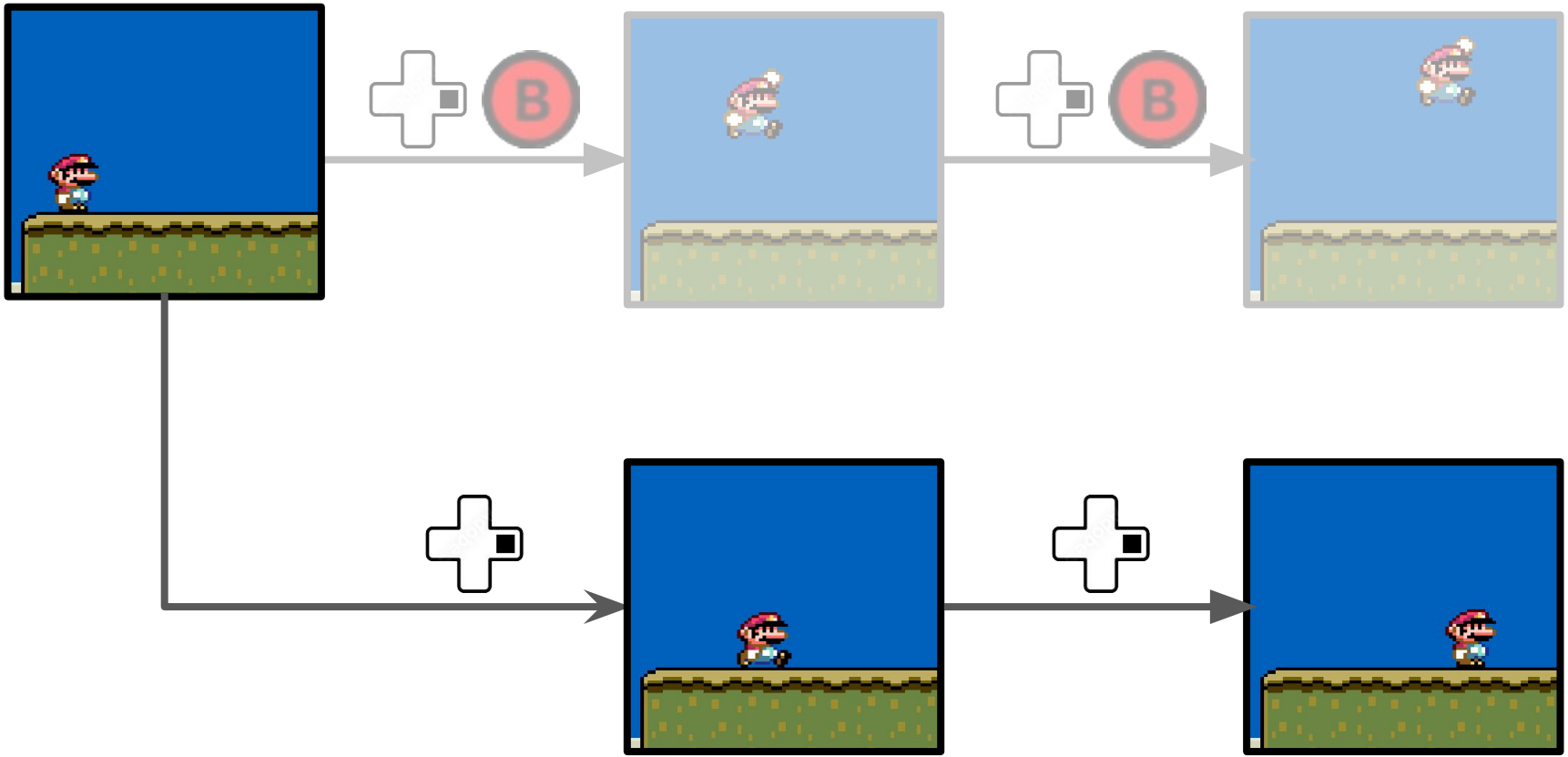
Lockstep Netcode

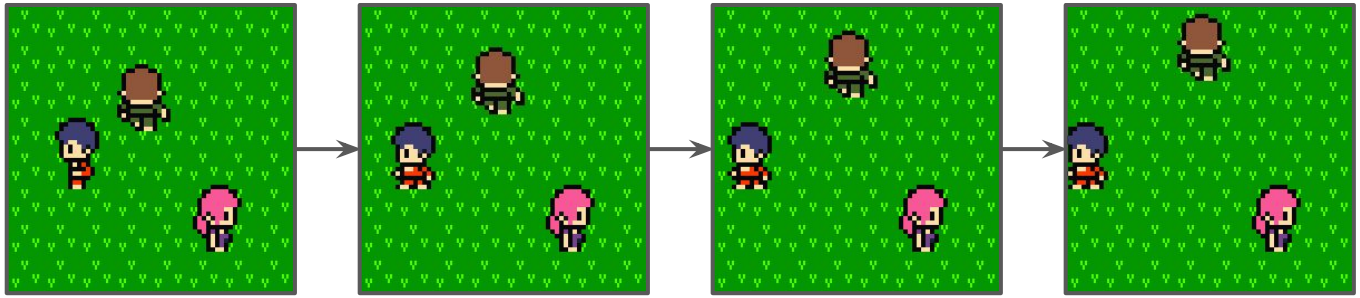


Some examples of netcode architectures:

- Lockstep
- **Rollback**
- Snapshot Interpolation

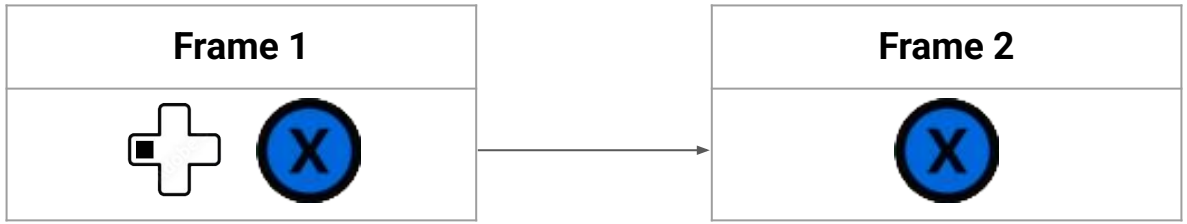

We are here
B
B


B
B



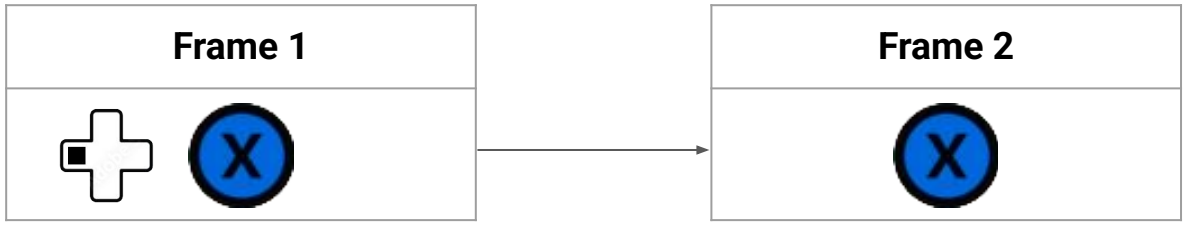
	Frame 1	Frame 2	Frame 3	Frame 4
P1				
P2	X	X	X	X
P3	A B	A B	A B	A B

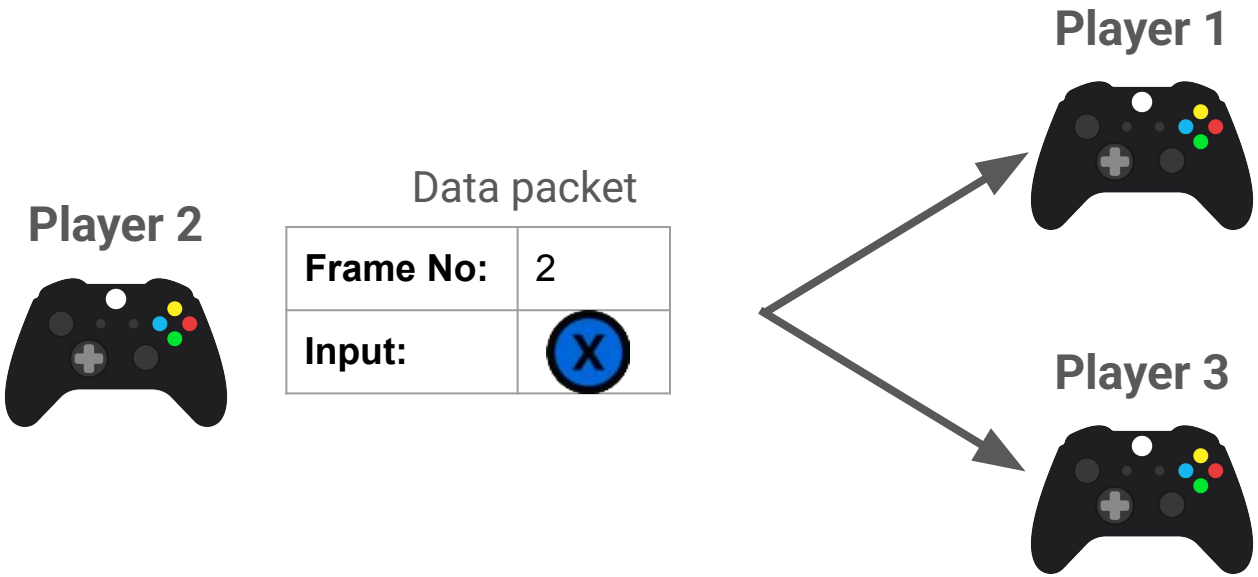
What actually happened…

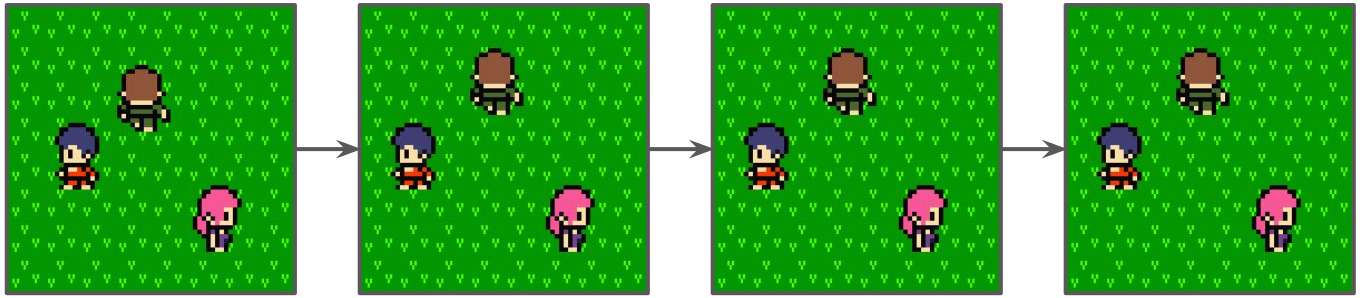


Player 2




Frame 1
Frame 2

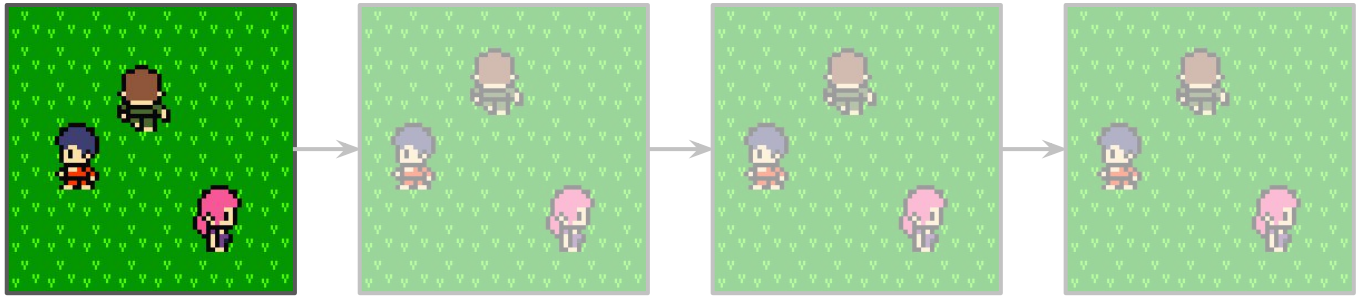

Player 1
Player 2
Data packet
Frame No:
2
Input:
Player 3



	Frame 1	Frame 2	Frame 3	Frame 4
P1				
P2	X	X	X	X
P3	A B	A B	A B	A B

P2:

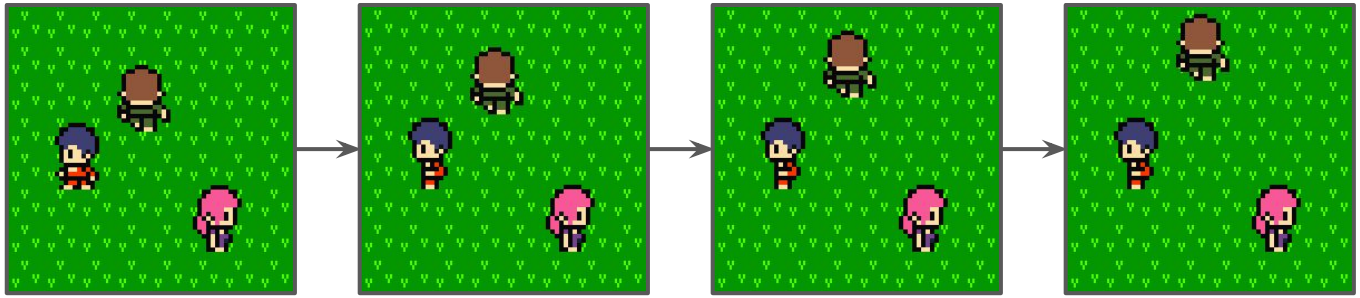
Frame No:	2
Input:	X



	Frame 1	Frame 2	Frame 3	Frame 4
P1				
P2	X	X		
P3	A B	A B		

P2:

Frame No:	2
Input:	X



	Frame 1	Frame 2	Frame 3	Frame 4
P1				
P2	X	X	X	X
P3	A B	A B	A B	A B

P2:

Frame No:	2
Input:	X

How to make Rollback Netcode work:

- Just like in Lockstep, we need a deterministic function to advance a game state from a set of player inputs.

```
struct GameState {
    ...
}
```

```
GameState advance(GameState state, PlayerInputSet player_inputs) {
    ...
}
```

How to make Rollback Netcode work:

- Just like in Lockstep, we need a deterministic function to advance a game state from a set of player inputs.

- **We also need a way to rollback a certain amount of frames.**

```
struct GameState {
    ...
}
```

```
GameState previous_game_states[20];
```

Rollback Netcode

Pros

- Similar pros to Lockstep
- Is far more seamless as clients don't have to constantly wait for the others.
- Can use even less bandwidth as clients only need to send input data if their inputs have changed.

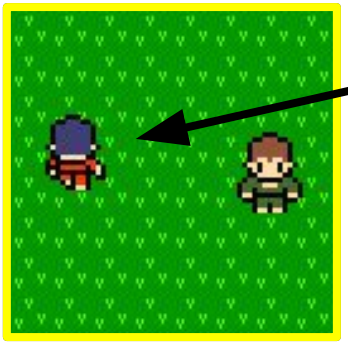
Cons

- Rollback is not always feasible for every kind of game (especially if there's a lot going on in the game state)
- Can be CPU demanding for the clients if frames have to be recalculated a lot

Some examples of netcode architectures:

- Lockstep
- Rollback
- **Snapshot Interpolation**

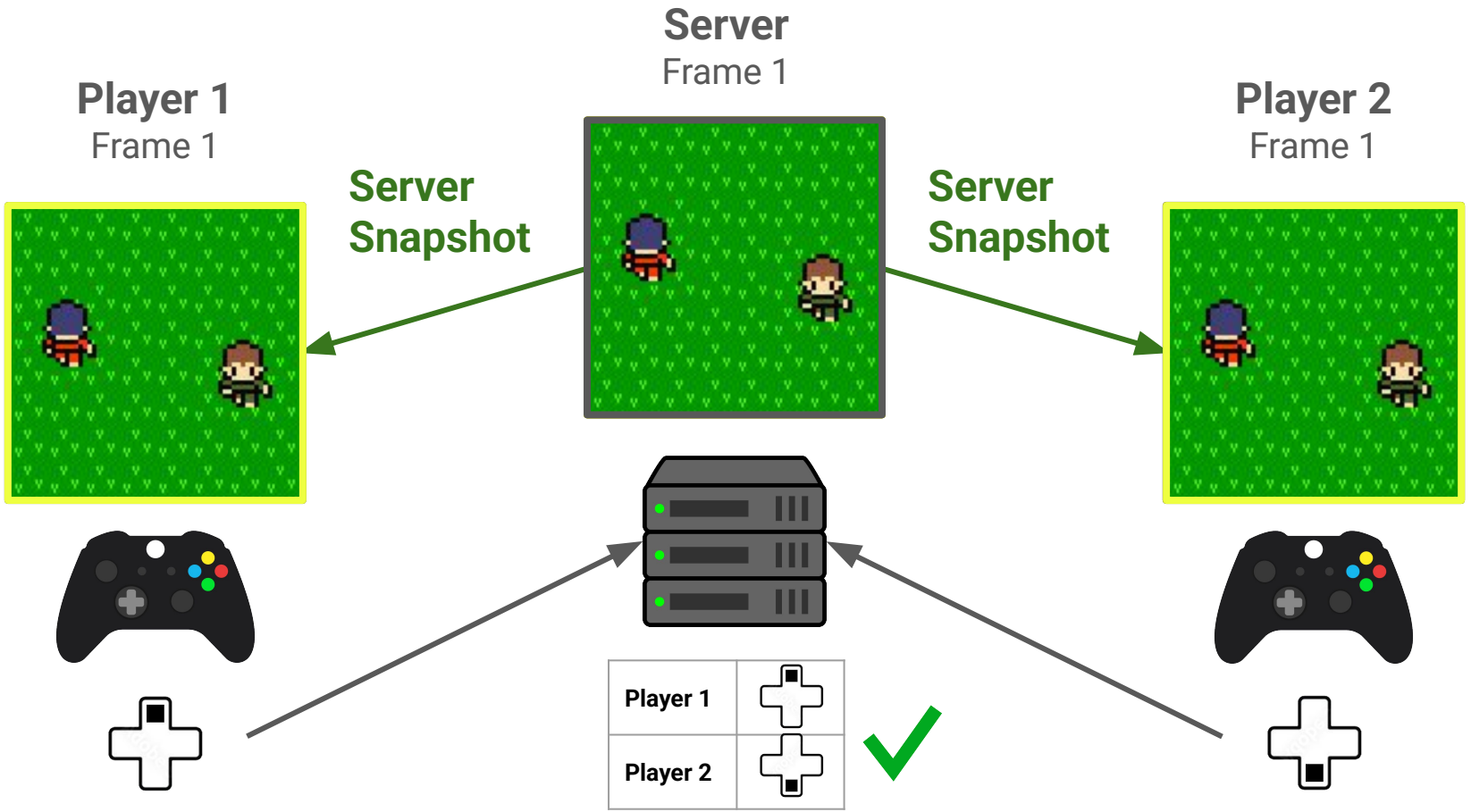
Player 1
Frame 1







We **predict** the next frame on the client side.


Server
Frame 1
Player 1
Frame 1
Player 2
Frame 1
Server
Snapshot
Server
Snapshot
Player 1
Player 2

Player 1
Frame **3**

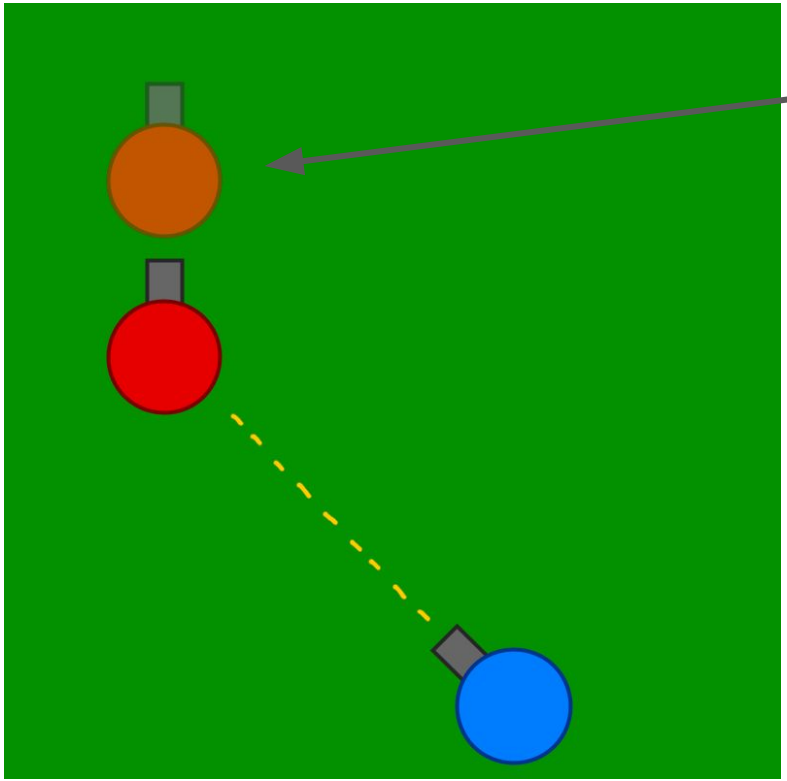
Server
Frame 1





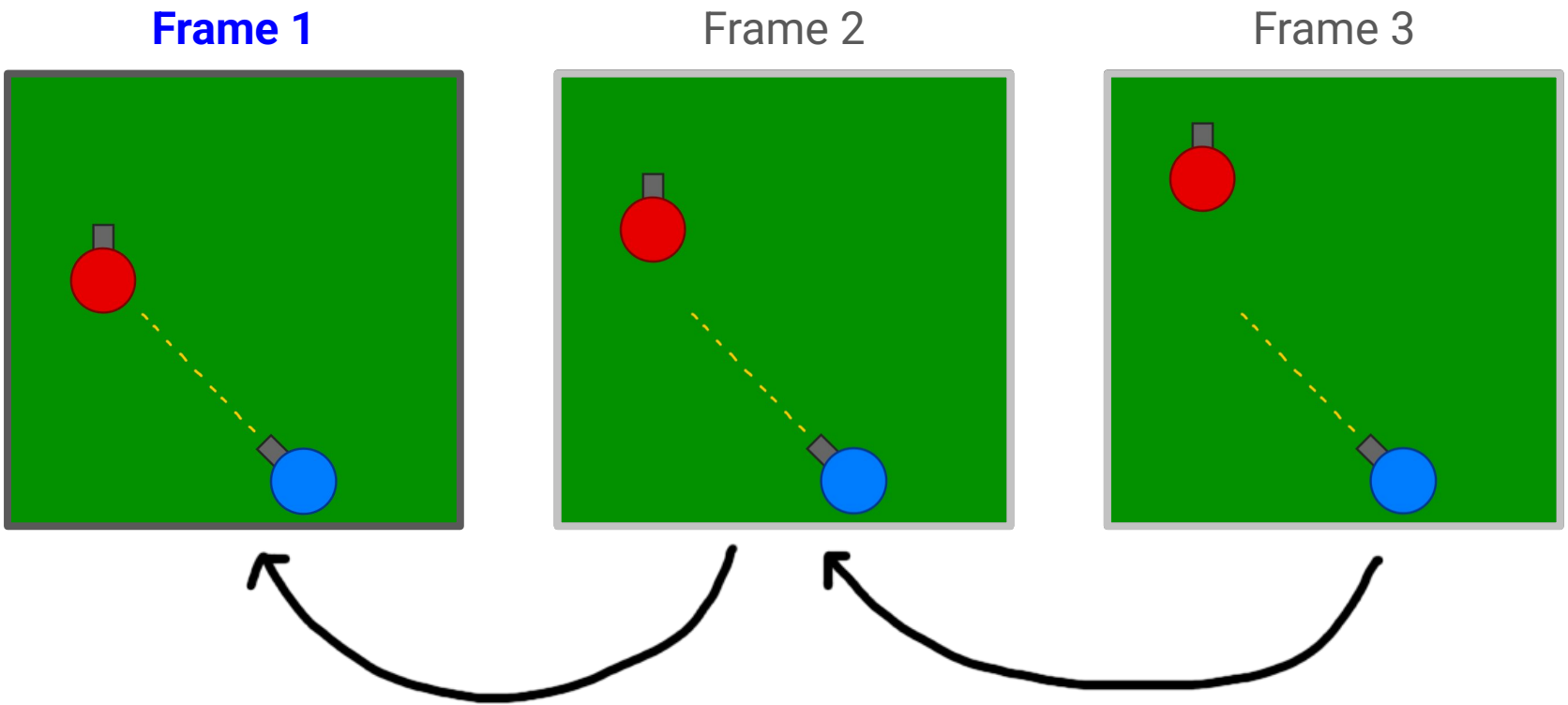
Frame 1	Frame 2	Frame 3
		

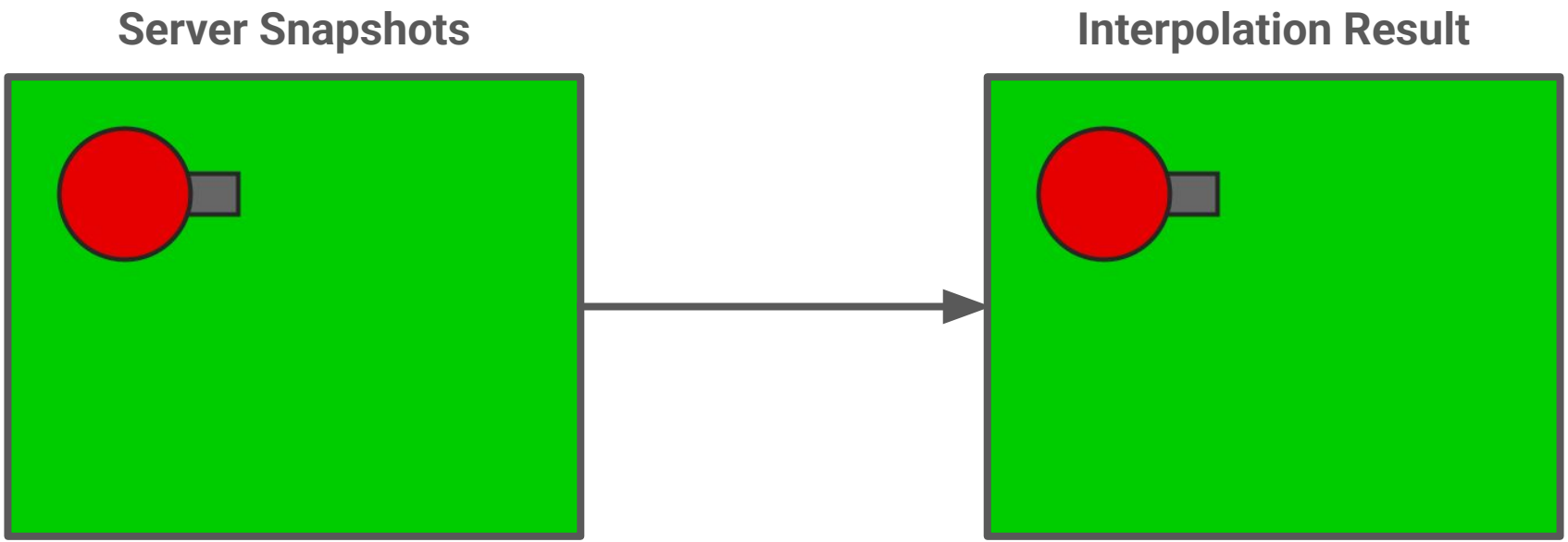
These inputs get reapplied to the snapshot that just came in!



Red’s actual position on the server

Backwards Reconciliation




Server Snapshots
Interpolation Result

Snapshot Interpolation Netcode

Pros

- Way less demanding on the client side.
- Doesn’t require game logic to be exactly deterministic.

Cons

- Uses far more bandwidth than lockstep/rollback since the server is sending entire game states across the network.

Sockets in Python

Recap

- We will be using UDP as our transportation layer protocol
- We will send packets between two sockets
- **There is no guarantee that data will arrive**
- **The connection is stateless**

IP *refers to a specific device on a network that the application is on*

Port Number *is where on that device the socket is listening for data for a specific application*



Sockets in Python

```python
import socket
```

```python
SOCK = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
SOCK.bind((IP, PORT))
SOCK.setblocking(False)
```

Receiving Data

```python
while True:
    try:
        data, addr = SOCK.recvfrom(1024)
    except BlockingIOError:
        break
```

Sending Data

```python
SOCK.sendto(packet_data, (IP, port))
```



Bytes in Python

All of these functions use the python type 'bytes', a sequence of binary

```python
def pack_data(message: str) -> bytes:
    return str.encode(message)
    # OR
    return message.encode("utf-8")


def unpack_data(data: bytes) -> str:
    return bytes.decode(data)
```

NOT RECOMMENDED!!!

```python
def pack_data(var1, var2, var3) -> bytes:
    return struct.pack("!HHH", var1, var2, var3)


def unpack_data(data: bytes) -> tuple[int, int, int]:
    return struct.unpack("!HHH", data)
```



Bytes in Python

Char	Data Type	Size (bytes)	Char	Data Type	Size (bytes)
b	Signed char	1	B	Unsigned char	1
h	Signed short	2	H	Unsigned short	2
i	Signed int	4	I	Unsigned int	4
f	Float	4	d	Double	8

```
6 def pack_data(var1, var2, var3) -> bytes:
5     return struct.pack("!HHH", var1, var2, var3)
4
3
2 def unpack_data(data: bytes) -> tuple[int, int, int]:
1     return struct.unpack("!HHH", data)
```



How sending works in the activity

```python
# Sends a bytestream to all of the specified destination ports
def send_data(data: bytes) -> None:
    sock = SOCK
    if not hasattr(send_data, "buffer"):
        send_data.buffer = []
    global DESTINATION_PORTS
    current_time = time.monotonic()
    send_data.buffer.append((data, current_time))

    remaining = []

    for packet_data, timestamp in send_data.buffer:
        if current_time - timestamp >= DELAY / 1000:
            for port in DESTINATION_PORTS:
                sock.sendto(packet_data, (IP, port))
                print("sent message")
        else:
            remaining.append((packet_data, timestamp))
    send_data.buffer = remaining
```



How receiving works in the activity

```
22 # Returns a list of bytes, may be empty, each item is its own transmission
21 def receive_data() -> list:
20     sock = SOCK
19     return_data = []
18     while True:
17         try:
16             data, addr = sock.recvfrom(1024)
15             return_data.append(data)
14         except BlockingIOError:
13             break
12     return return_data
```



The game

```python
class Player:
    def __init__(self, colour: int):
        self.player_pos = rl.Vector2(0, 0)
        self.player_vel = rl.Vector2(0, 0)
        self.player_grounded = False
        self.colour = colour


class GameState:
    def __init__(self):
        self.spritesheet = None
        self.grid = [0 for _ in range(TILES_X * TILES_Y)]
        self.player = Player(COLOUR)
        self.other_players = []
        self.other_players.append(Player(1))
        self.updated_tiles: dict[tuple[int, int], int] = {}
        self.selected_tile = 1

```



Useful Functions

```gdscript
def get_tile(state: GameState, x, y: int) -> int:
    return state.grid[y * TILES_X + x]


def set_tile(state: GameState, x, y: int, tile: int):
    state.grid[y * TILES_X + x] = tile

```



Functions to complete!!!

```
22 def send_state(state):
21     for pos, block in state.updated_tiles.items():
20         x = pos[0]
19         y = pos[1]
18         pass
17
16
15 def receive_state(state):
14     for data in receive_data():
13        pass
12
11 # Turn a variables into a single bytestream
10 def pack_data() -> bytes:
 9     return struct.pack()
 8
 7
 6 # Turn bytes into a list of variables
 5 def unpack_data(data: bytes) -> tuple:
 4     return struct.unpack()
```



Running the Program

If you are on the lab computers, you will need to set up a Python Virtual Environment (venv) to install the raylib module.

Command Line Args

--connect: can be used multiple times, ports that you will send data to
--port: the port your program runs on
--colour: the colour of your sprite

FIRST PROGRAM

>> python block_game.py --port 5005 --connect 5006 --colour 0

SECOND PROGRAM

>> python block_game.py --port 5006 --connect 5005 --colour 1

Colour	#
Red	0
Blue	1
Yellow	2
Green	3



Exercises

If you are on the lab computers, you will need to set up a Python Virtual Environment (venv) to install the raylib module.

Task 1: Share block placements over the network

You will need to complete the following functions

- *send_state()*
- *receive_state()*
- *pack_data()*
- *unpack_data()*

You will need to update the variable *state*. **You do not need to return this**

Task 2: Share the positions of both players over the networks

You will need to update the **binary format** and send additional packets for **player position**. You should use the same **binary format** for both transmission types

Again, you will need to update the variable *state*. **You do not need to return this**

Task 3: Edit the program to allow an unlimited number of users simultaneously

You are welcome to change GameState and player for this task

