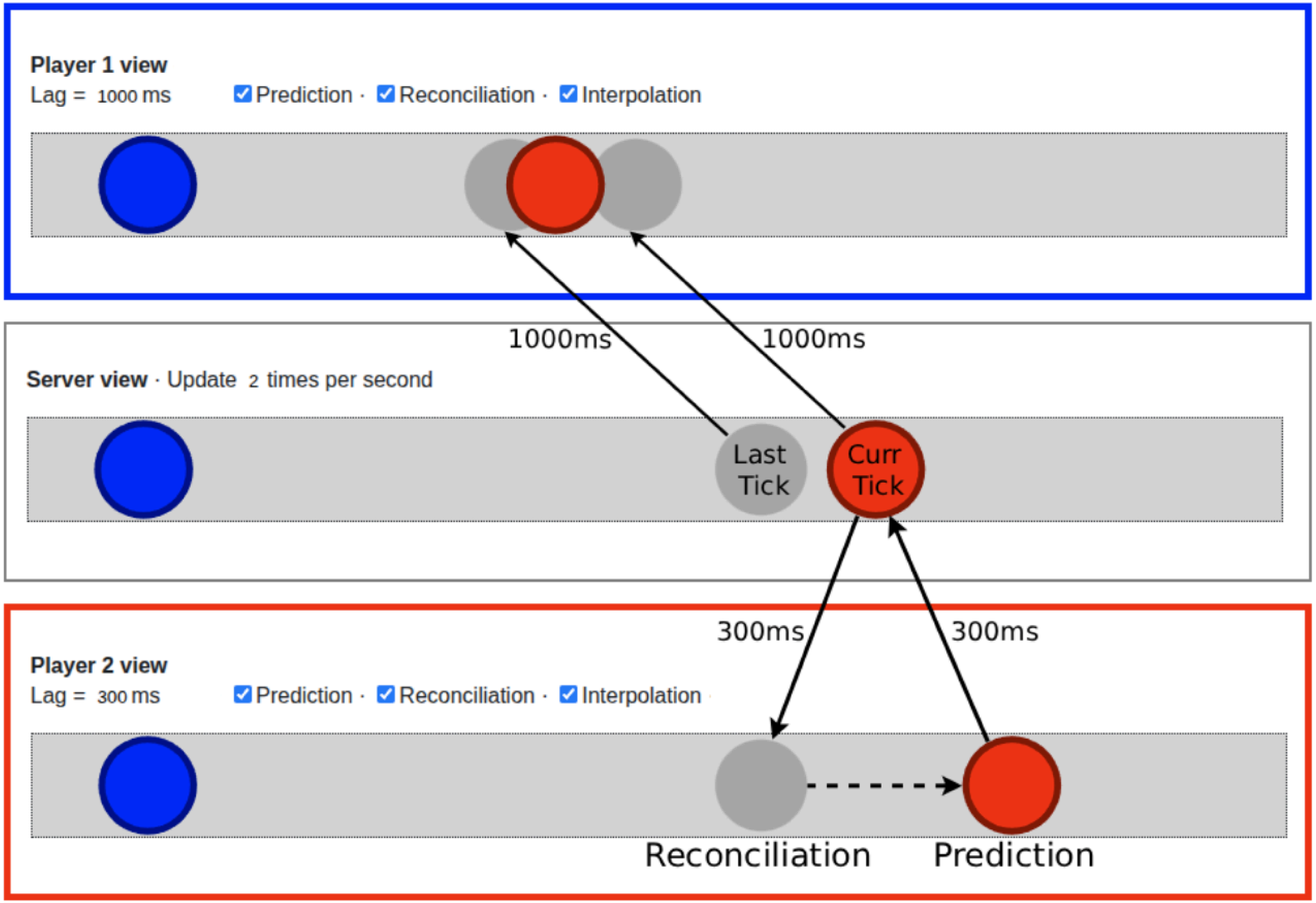
KRAFTON
GAME DEVELOPER PPT



Player 1 view

Lag = 1000 ms ☑ Prediction · ☑ Reconciliation · ☑ Interpolation

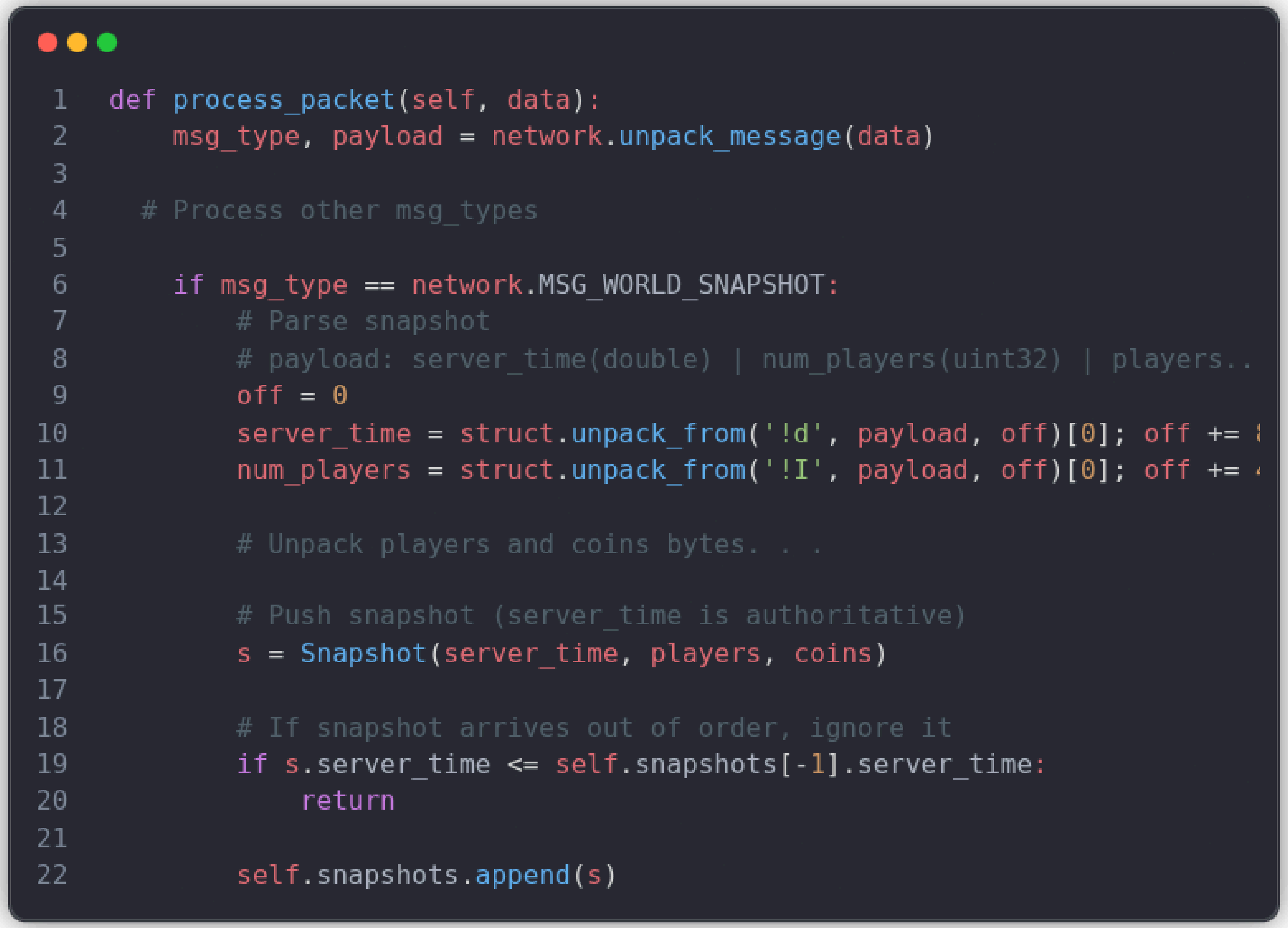
Server view · Update 2 times per second

Player 2 view

Lag = 300 ms ☑ Prediction · ☑ Reconciliation · ☑ Interpolation

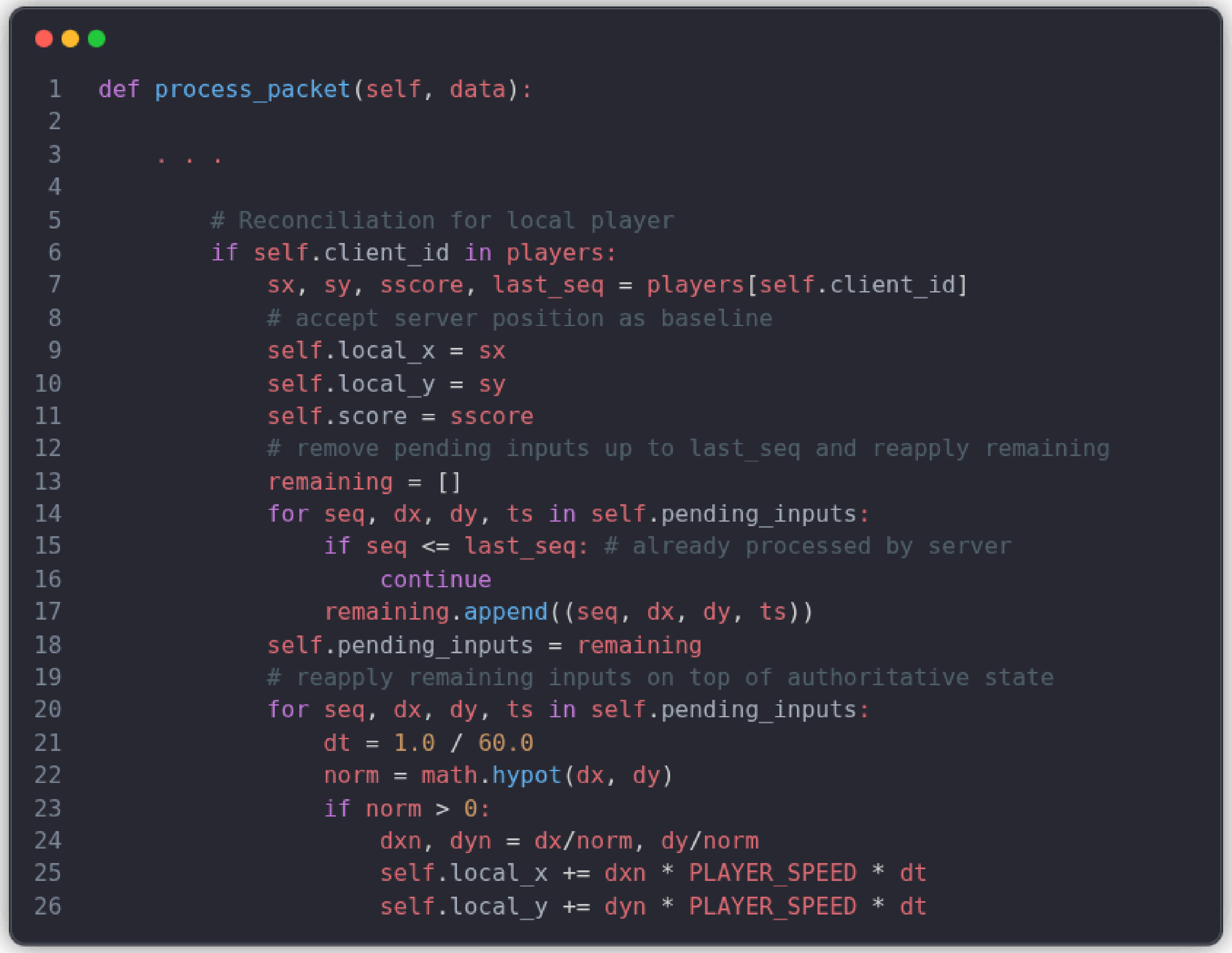
https://www.gabrielgambetta.com/client-side-prediction-live-demo.html

Client Packet Processing

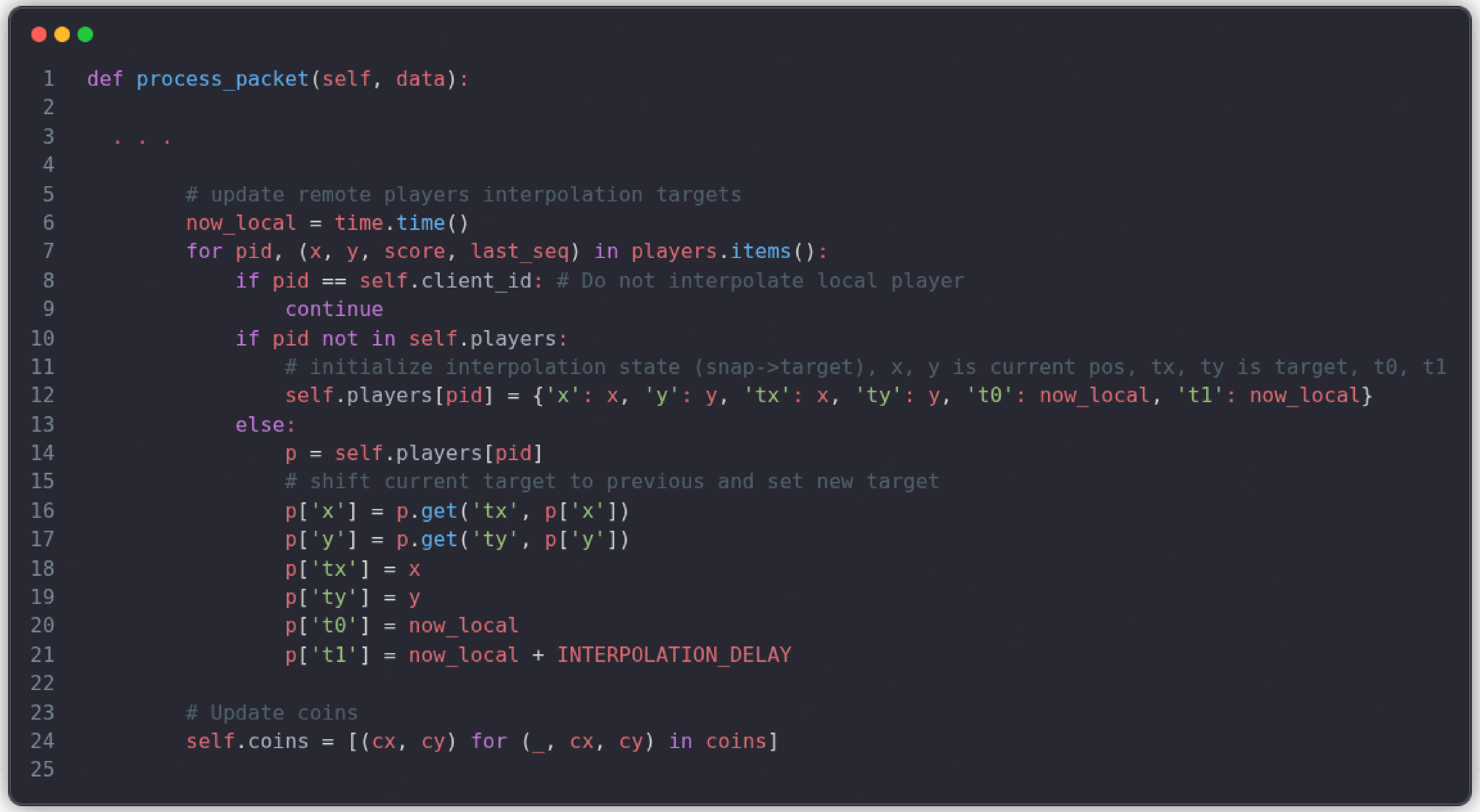


```
def process_packet(self, data):
    msg_type, payload = network.unpack_message(data)

  # Process other msg_types

    if msg_type == network.MSG_WORLD_SNAPSHOT:
        # Parse snapshot
        # payload: server_time(double) | num_players(uint32) | players..
        off = 0
        server_time = struct.unpack_from('!d', payload, off)[0]; off +=
        num_players = struct.unpack_from('!I', payload, off)[0]; off +=

        # Unpack players and coins bytes. . .

        # Push snapshot (server_time is authoritative)
        s = Snapshot(server_time, players, coins)

        # If snapshot arrives out of order, ignore it
        if s.server_time <= self.snapshots[-1].server_time:
            return

        self.snapshots.append(s)
```

Client Reconciliation and Entity Targets



```python
def process_packet(self, data):

    . . .

        # Reconciliation for local player
        if self.client_id in players:
            sx, sy, sscore, last_seq = players[self.client_id]
            # accept server position as baseline
            self.local_x = sx
            self.local_y = sy
            self.score = sscore
            # remove pending inputs up to last_seq and reapply remaining
            remaining = []
            for seq, dx, dy, ts in self.pending_inputs:
                if seq <= last_seq: # already processed by server
                    continue
                remaining.append((seq, dx, dy, ts))
            self.pending_inputs = remaining
            # reapply remaining inputs on top of authoritative state
            for seq, dx, dy, ts in self.pending_inputs:
                dt = 1.0 / 60.0
                norm = math.hypot(dx, dy)
                if norm > 0:
                    dxn, dyn = dx/norm, dy/norm
                    self.local_x += dxn * PLAYER_SPEED * dt
                    self.local_y += dyn * PLAYER_SPEED * dt
```



```python
def process_packet(self, data):

  . . .

        # update remote players interpolation targets
        now_local = time.time()
        for pid, (x, y, score, last_seq) in players.items():
            if pid == self.client_id: # Do not interpolate local player
                continue
            if pid not in self.players:
                # initialize interpolation state (snap->target), x, y is current pos, tx, ty is target, t0, t1
                self.players[pid] = {'x': x, 'y': y, 'tx': x, 'ty': y, 't0': now_local, 't1': now_local}
            else:
                p = self.players[pid]
                # shift current target to previous and set new target
                p['x'] = p.get('tx', p['x'])
                p['y'] = p.get('ty', p['y'])
                p['tx'] = x
                p['ty'] = y
                p['t0'] = now_local
                p['t1'] = now_local + INTERPOLATION_DELAY

        # Update coins
        self.coins = [(cx, cy) for (_, cx, cy) in coins]

```

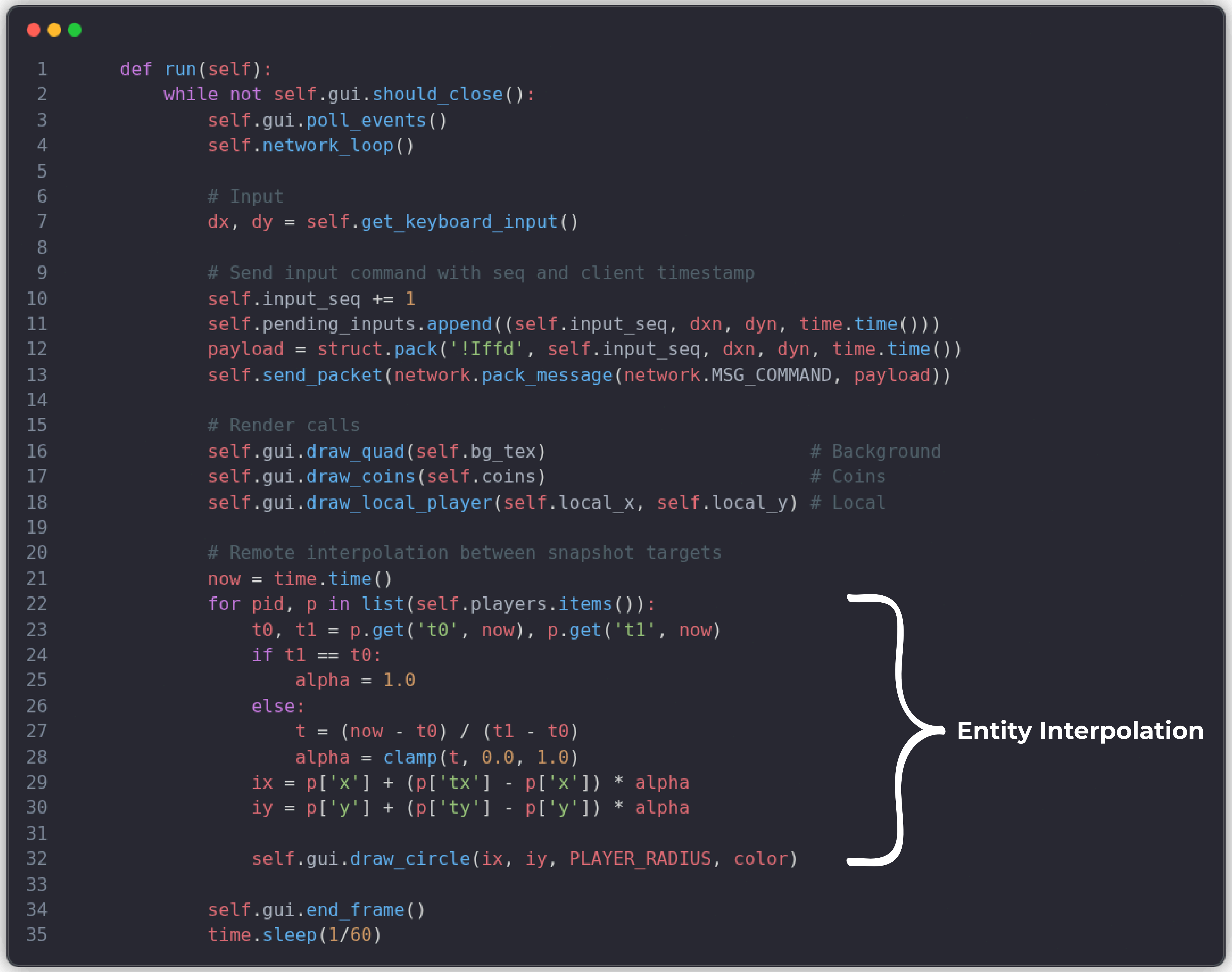
Reconciliation:

- Assume the server is the ground truth
- Predict using the pending inputs that are not yet processed by the server

Entity interpolation data:

- Player (x, y) ← Previous player position
- Player (target x, target y) ← New player positions **(from server)**
- t0 ← local_time **(start interpolating from previous frame)**
- t1 ← time + tick_delay **(Interpolate for 1 server tick)**

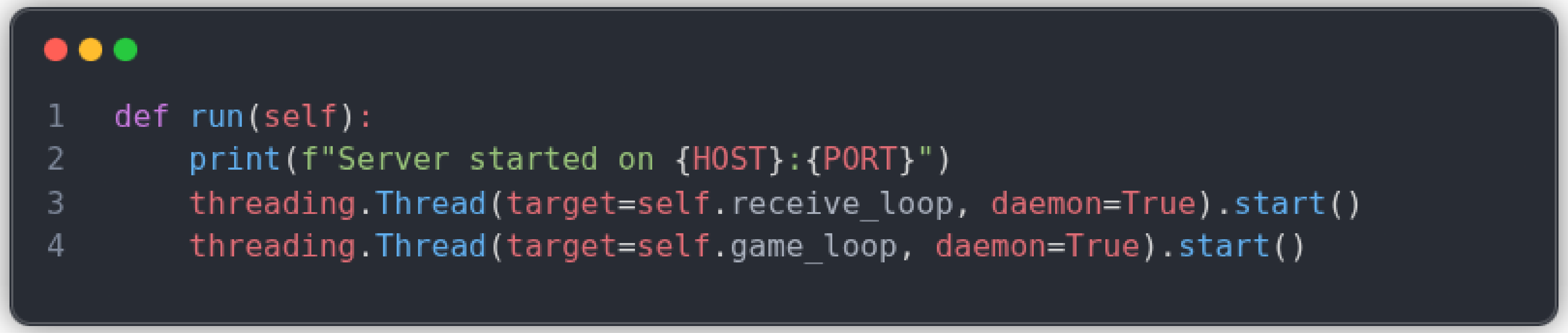
Client Entity Interpolation



```python
def run(self):
    while not self.gui.should_close():
        self.gui.poll_events()
        self.network_loop()

        # Input
        dx, dy = self.get_keyboard_input()

        # Send input command with seq and client timestamp
        self.input_seq += 1
        self.pending_inputs.append((self.input_seq, dxn, dyn, time.time()))
        payload = struct.pack('!Iffd', self.input_seq, dxn, dyn, time.time())
        self.send_packet(network.pack_message(network.MSG_COMMAND, payload))

        # Render calls
        self.gui.draw_quad(self.bg_tex)                        # Background
        self.gui.draw_coins(self.coins)                        # Coins
        self.gui.draw_local_player(self.local_x, self.local_y) # Local

        # Remote interpolation between snapshot targets
        now = time.time()
        for pid, p in list(self.players.items()):
            t0, t1 = p.get('t0', now), p.get('t1', now)
            if t1 == t0:
                alpha = 1.0
            else:
                t = (now - t0) / (t1 - t0)
                alpha = clamp(t, 0.0, 1.0)
            ix = p['x'] + (p['tx'] - p['x']) * alpha
            iy = p['y'] + (p['ty'] - p['y']) * alpha

            self.gui.draw_circle(ix, iy, PLAYER_RADIUS, color)

        self.gui.end_frame()
        time.sleep(1/60)
```

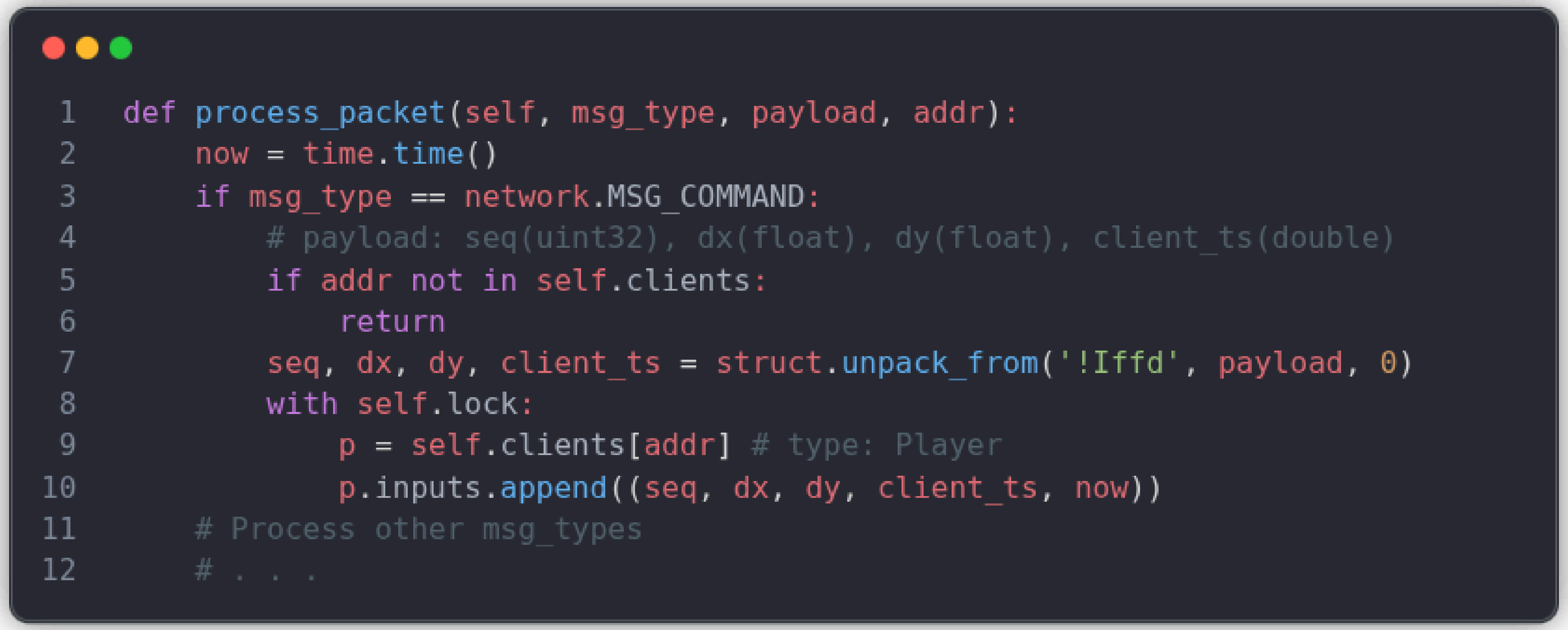
Entity Interpolation

Server Recevier Thread



```python
def run(self):
    print(f"Server started on {HOST}:{PORT}")
    threading.Thread(target=self.receive_loop, daemon=True).start()
    threading.Thread(target=self.game_loop, daemon=True).start()
```

Receiver loop gets data from socket and process packets

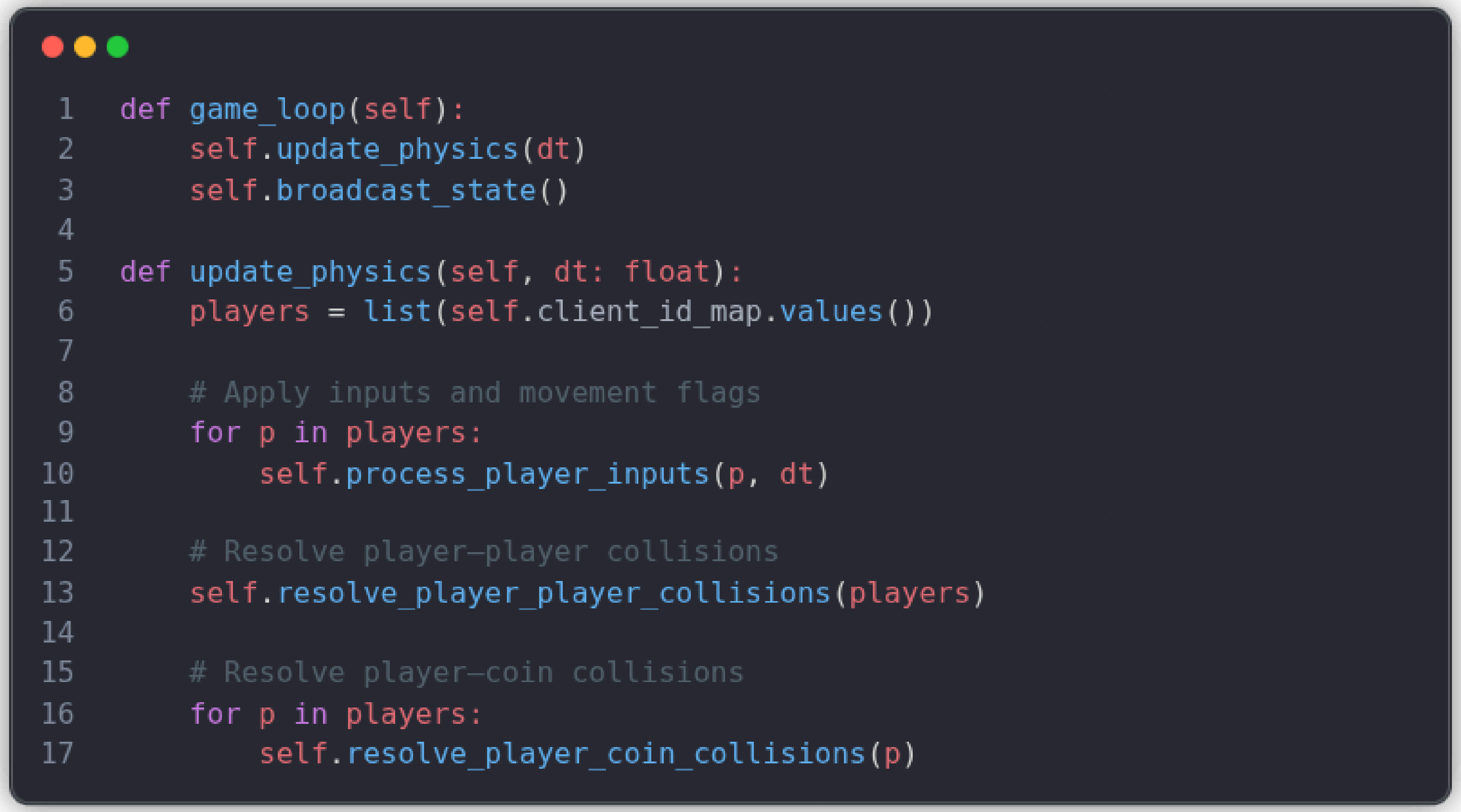


```python
def process_packet(self, msg_type, payload, addr):
    now = time.time()
    if msg_type == network.MSG_COMMAND:
        # payload: seq(uint32), dx(float), dy(float), client_ts(double)
        if addr not in self.clients:
            return
        seq, dx, dy, client_ts = struct.unpack_from('!Iffd', payload, 0)
        with self.lock:
            p = self.clients[addr] # type: Player
            p.inputs.append((seq, dx, dy, client_ts, now))
    # Process other msg_types
    # . . .
```

Append client inputs to a queue to be processed by the server tick thread

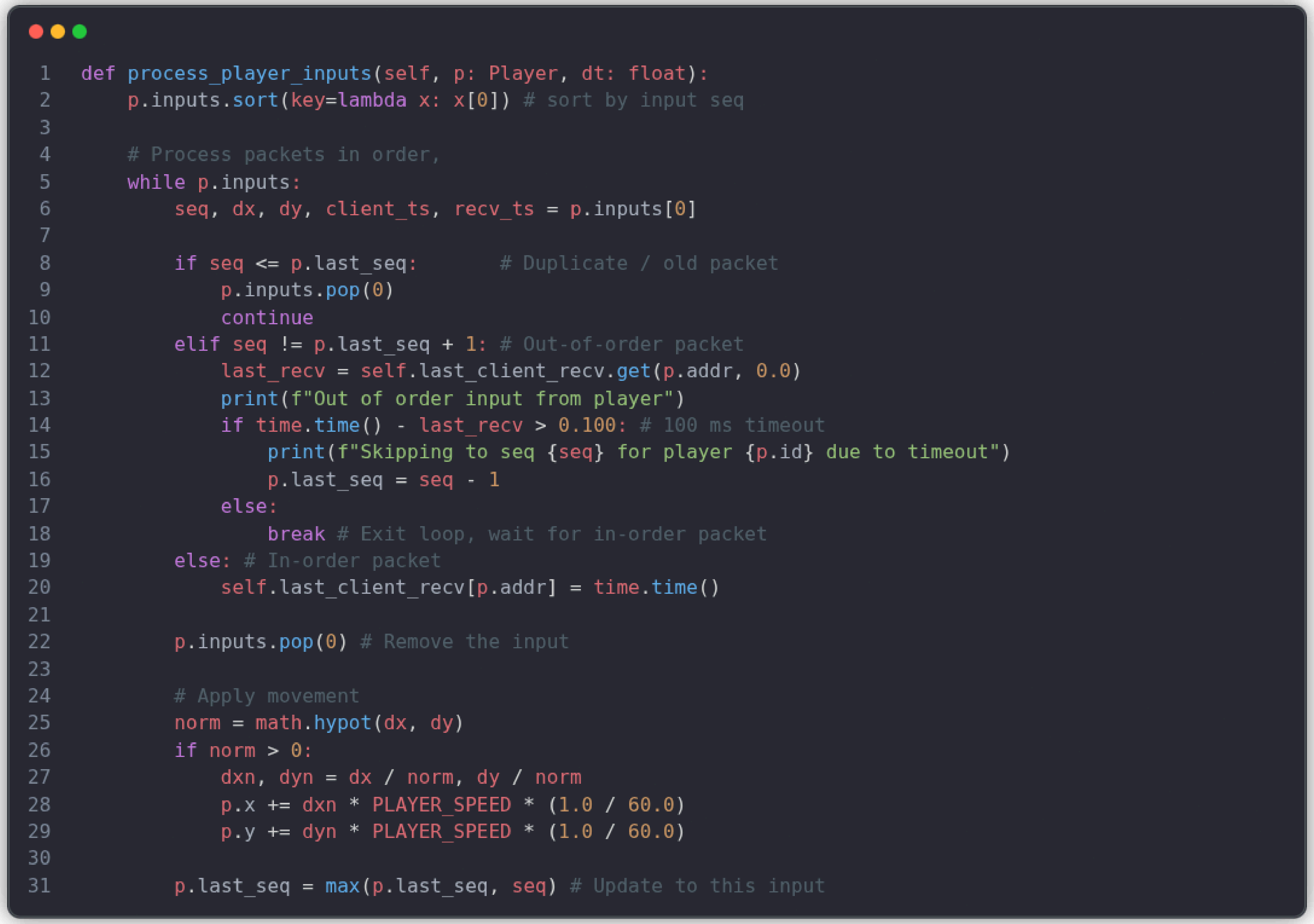
Server Game Thread

Game loop runs at 60Hz and handles player input + collision + coin pickups



```python
def game_loop(self):
    self.update_physics(dt)
    self.broadcast_state()

def update_physics(self, dt: float):
    players = list(self.client_id_map.values())

    # Apply inputs and movement flags
    for p in players:
        self.process_player_inputs(p, dt)

    # Resolve player–player collisions
    self.resolve_player_player_collisions(players)

    # Resolve player–coin collisions
    for p in players:
        self.resolve_player_coin_collisions(p)
```

Server - Client Input Processing



```python
def process_player_inputs(self, p: Player, dt: float):
    p.inputs.sort(key=lambda x: x[0]) # sort by input seq

    # Process packets in order,
    while p.inputs:
        seq, dx, dy, client_ts, recv_ts = p.inputs[0]

        if seq <= p.last_seq:       # Duplicate / old packet
            p.inputs.pop(0)
            continue
        elif seq != p.last_seq + 1: # Out-of-order packet
            last_recv = self.last_client_recv.get(p.addr, 0.0)
            print(f"Out of order input from player")
            if time.time() - last_recv > 0.100: # 100 ms timeout
                print(f"Skipping to seq {seq} for player {p.id} due to timeout")
                p.last_seq = seq - 1
            else:
                break # Exit loop, wait for in-order packet
        else: # In-order packet
            self.last_client_recv[p.addr] = time.time()

        p.inputs.pop(0) # Remove the input

        # Apply movement
        norm = math.hypot(dx, dy)
        if norm > 0:
            dxn, dyn = dx / norm, dy / norm
            p.x += dxn * PLAYER_SPEED * (1.0 / 60.0)
            p.y += dyn * PLAYER_SPEED * (1.0 / 60.0)

        p.last_seq = max(p.last_seq, seq) # Update to this input
```

- **Latency variation can cause out-of-order packets**

- **Simple Solution:**
    - Wait for packets to arrive
    - If timeout(100ms), then move on
    - Discard packets later than 100ms later the latest packet

THANK YOU