CS4344 Assignment 2

Networked Multiplayer Pong Game

Semester 2, 2014/15

A0073063M - Tay Yang Shun
tay.yang.shun@gmail.com

A0088498A - Nguyen Trung Hieu
ngtrhieu0011@gmail.com

1. Local Lag

Chosen local lag value: **240ms**

From our knowledge of Human-Computer Interaction, a human's perceptual processor cycle is 100ms. Hence local lag values below 100ms will go totally unnoticed by players.

Both cycle times for the human cognitive processor and motor processors are 70ms. In total, one perceive-recognize-act cycle is 100 + 70 + 70 = 240ms. Hence we have decided to use 240ms as the local lag value, which is a value that is long enough for the user to predict the path and future location of the ball, and adjust the paddle position and have the paddle at where it wants it to be. The chosen value is small enough and unlikely to compromise the player's skill.

With audio added, we noticed that the local lag can be extended larger than 240ms. The amount of extension depends on the audio itself.

New implementation of local lag:

`Pong.js`

```
var Pong = {
    HEIGHT : 400,                   // height of Pong game window
    WIDTH : 400,                    // width of Pong game window
    PORT : 4344,                    // port of Pong game
    FRAME_RATE : 25,                // frame rate of Pong game
    SERVER_NAME : "localhost",      // server name of Pong game

    // New value
    LOCAL_LAG : 240                 // the client local lag in ms
}
```

`PongClient.js`

```
var onMouseMove = function (e) {
    ...

    // Short circuiting the paddle movement, local lag applied
    var estimatedLag = Math.min(Pong.LOCAL_LAG, delay);
    setTimeout(function() {
        myPaddle.x = newMouseX;
    }, estimatedLag);

    ...
}
```

A new property, `LOCAL_LAG` has been added to the `Pong` object and it has a value of **240**. Since the purpose of using `LOCAL_LAG` is to hide some of the network delay, we only use `LOCAL_LAG` when the network delay is larger than `LOCAL_LAG`. Otherwise, the network delay value itself will be used.

2. Hold and Wait

NOTE: *Since a large portion of the logic in PongClient and PongBot are similar, we will only describe the changes made in PongClient.*

In order to implement Hold and Wait, the machine but be able to control whether the ball should be moving. In order to do that, we modify the `Ball.updatePosition()` method to accept an explicit parameter to control the movement.

`Ball.js`

```
this.updatePosition = function (enabled) {
    var now = getTimestamp(); // get the current time in millisecond resolution

    // Update position only if ball update is enabled
    if (enabled && lastUpdate > 0) {
        that.x += that.vx * (now - lastUpdate) * Pong.FRAME_RATE/1000;
        that.y += that.vy * (now - lastUpdate) * Pong.FRAME_RATE/1000;
    }

    lastUpdate = now;
}
```

In this code, the ball will be moved to the next frame's location if `enabled = true`. Otherwise, the ball will just be frozen in place.

Since the `PongServer` should always simulate the ball movement, `ball.updatePosition()` always receives `true` as the parameter:

`PongServer.js`

```
var gameLoop = function () {
    ...

    // Move ball
    ball.updatePosition(true);
    ball.checkForBounce(p1.paddle, p2.paddle);

    ...
}
```

However, in both `PongClient` and `PongBot`, the simulation should be controlled. Here we use the already implemented `Ball.moving` property to determine the ball movement.

`PongClient.js`

```
var gameLoop = function () {
    ball.updatePosition(ball.moving);
    ...
}
```

The `PongClient` should also be able to switch the ball movement on/off. In order to achieve this, we modify the `Ball.checkForBounce()` method to return a boolean value stating whether the ball need to wait for server before moving.

Below is the pseudo code of the modified method:

`Ball.js`

```
this.checkForBounce = function(topPaddle, bottomPaddle) {
    var waitForServer = false;

    // Check for bouncing
    if (<BOUNCE_OFF_HORIZONTALLY>) {
        ...                                 // Do nothing !
    } else if (<GOES_OUT_OF_BOUNDS>) {
        ...
        waitForServer = true;               // Wait for server to declare dead
    } else if (<CHECK_BOUNCE_PADDLE>) {
        ...
        waitForServer = updateVelocity(paddle.x); // Depend on whether the ball
                                                  // actually hits any of the paddles
    }
```

```
    return waitForServer;
}
```

Here `updateVelocity()` is modified to return `true` if the ball hits the paddle, `false` otherwise.

Now, `PongClient` just needs to toggle the `Ball.moving` field, as followed:

`PongClient.js`

```
var gameLoop = function () {
    ball.updatePosition(ball.moving);

    var waitForServer = false;
    if (myPaddle.y < Paddle.HEIGHT) {
        // my paddle is on top
        waitForServer = ball.checkForBounce(myPaddle, opponentPaddle);
    } else {
        // my paddle is at the bottom
        waitForServer = ball.checkForBounce(opponentPaddle, myPaddle);
    }

    // Stop the ball and wait for server
    if (ball.moving && waitForServer) {
        AudioManager.playBounceSound();
        ball.moving = false;
    }

    ...
}

...

var initNetwork = function() {

    case "updateVelocity":
        ...
        // Periodically resync ball position to prevent error
        // in calculation to propagate.
        ball.x = message.ballX;
        ball.y = message.ballY;
        ball.moving = true;
        ...
}
```

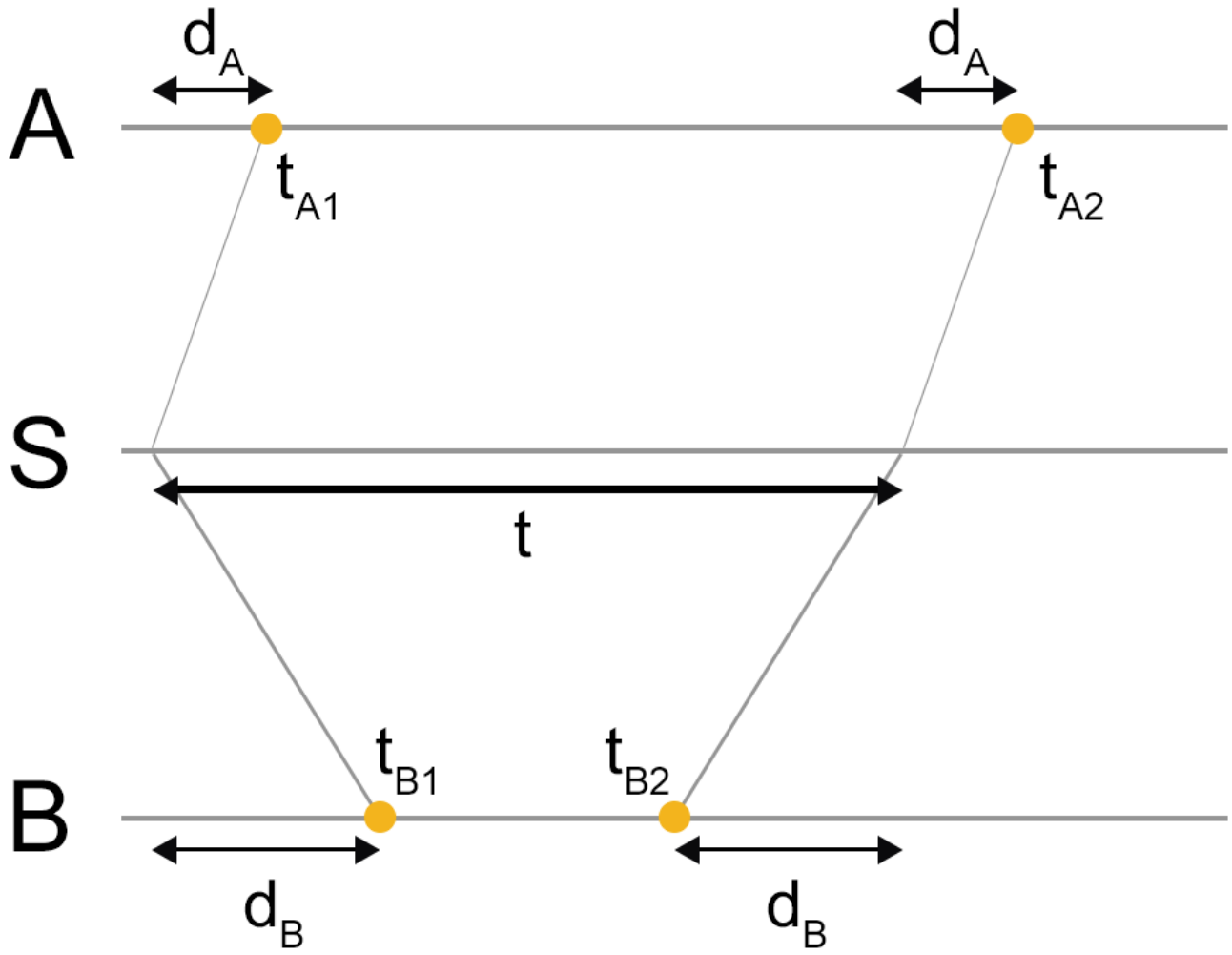
Here, `PongClient` will attempt to stop the `Ball` whenever it detects `Ball` bounces of `Paddle` or goes out of bounce. Then, the `Ball` movement will be resumed at the next `updateVelocity` (or the whole game state will be restarted upon receiving `outOfBound`).

Bonus: Play a sound when the ball hits the paddle

Yes, the playing of a beep sound when the ball sticks to the paddle makes the period less noticeable. The player's cognitive attention is distracted by the sound which makes it harder to notice that the ball sticks to the paddle.

3. Local Perception Filter

In Local Perception Filter (LPF), entities near the player are rendered in real-time while entities near the remote player must be rendered closer to that remote player's time (delayed time). Here in this game, as a rough approximation, we do not modify the acceleration of the ball; we only modify the ball's velocity. Hence when the ball is travelling away from the player's paddle, it is travelling at a slower speed; whereas when the ball is travelling towards the player's paddle, it is travelling at a faster speed.



The diagram above shows what happens when the ball leaves Paddle A for Paddle B. d_A and d_B indicates the client-server delay for A and B respectively. t is the duration that a ball travelling at a normal speed will need to travel from one paddle to the other.

At t_{A1} and t_{B1}, the ball has just bounced off paddle A. In order to be fair for player B, the final position of paddle B has to be relayed to the server before the ball reaches the B side at the server. Hence, on player B side, the duration that the ball is given to travel from one end to the other is $t - 2t_{B1}$. At t_{A2}, player A will see that the ball has just bounced off paddle B. On player A side, the duration that the ball is given to travel from one end to the other remains the same, t.

Hence the new velocity magnitude when ball is travelling towards the player can be calculated with the new (shorter) duration in mind.

In order to implement LPF, we modified the `updateVelocity` case on `PongClient` as follows:

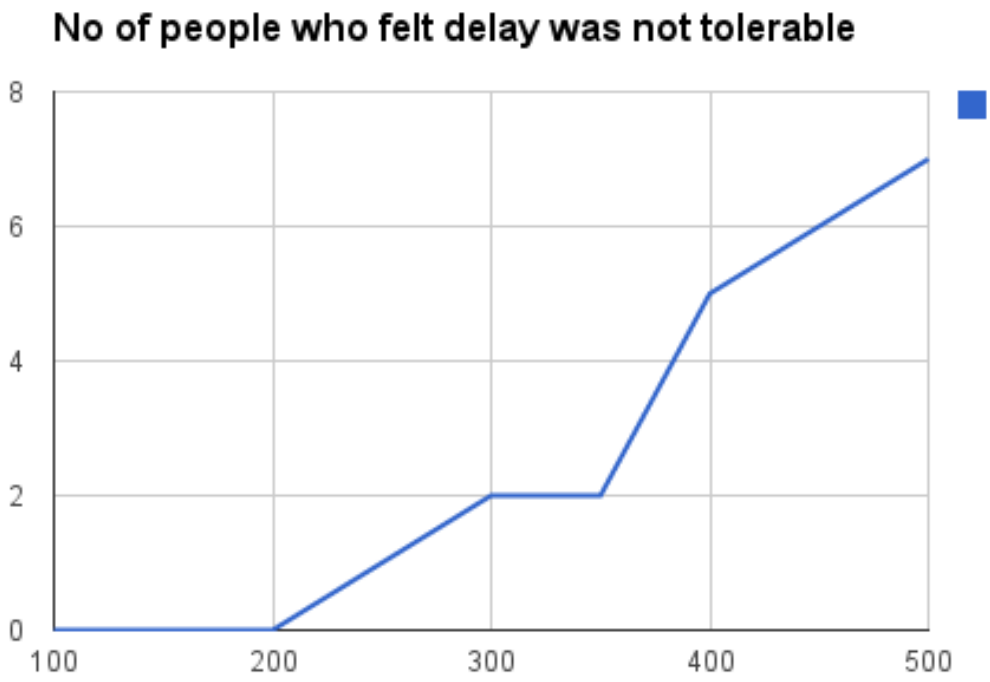
```
case "updateVelocity":
    ...
    ball.moving = true;

    // Local Perception Filter
    var isBallComing =  (myPaddle.y < Paddle.HEIGHT && ball.vy < 0) // my paddle is up and         the ball moving upward
                      || (myPaddle.y > Paddle.HEIGHT && ball.vy > 0); // my paddle is at bottom and the ball moving downward
    // Adjust ball velocity based on estimated delay
    var travellingDuration = Math.abs((Pong.HEIGHT - 2*Paddle.HEIGHT) / ball.vy);
    var adaptedTime = isBallComing ? travellingDuration - 2*delay*Pong.FRAME_RATE/1000 : travellingDuration;
    var adaptedVY = ball.vy / Math.abs(ball.vy) * Pong.HEIGHT / adaptedTime;
    ball.vy = adaptedVY;            // Overwrite original vy value
    ball.vx = message.ballVX * adaptedVY / message.ballVY;
    ...
}
```

Here, we try to use the data inside the message to predict the duration the server's ball will take to travel across the playing field (`travellingDuration`). Using the duration predicted and the client-server delay, we will estimate the duration the client's ball should be moving across the playing field (`adaptedTime`). With the estimated duration and the known size of the playing field, estimated velocity can be calculated (`adaptedVY`).

Notice here we only use vertical velocity (`BallVY`) to estimate the travelling duration, because only the vertical component actually affects `travellingDuration`. However, both `ball.vy` and `ball.vx` should be updated to ensure the velocity direction is maintained.

4. Tolerable Network Delay

We did a playtesting session with 8 different people. Increasing their delay in increments of 50ms while playing with a bot. At anytime, when they felt that the delay was not tolerable, they would let us know.



The biggest increase in the number of people who felt that delay was not tolerable was from 350ms to 400ms. Hence, with all 3 techniques implemented, the delay can go up as high as **350ms** and still be tolerated by most players. While the game is technically playable above this threshold, players will start to notice and feel uneasy with the ball sticking to the paddle. Above **400ms**, player will start to notice the ball's difference in speed due to LPF and find it is significantly more challenging to play.

Therefore, we can safely conclude the game's tolerable network delay to be **350ms**.