CI346 Programming Languages, Concurrency and Client Server Computing

Network Pong

Michael Norris

11831825

BSc Software Engineering

Network Pong

Version Submitted

I am submitting version two.

Program Listing

```
Pong
└── src
    ├── client
    │   ├── Client.java
    │   ├── ClientPongController.java
    │   ├── ClientPongModel.java
    │   ├── ClientPongView.java
    │   └── Player.java
    ├── common
    │   ├── DEBUG.java
    │   ├── GameObject.java
    │   ├── Global.java
    │   ├── NetObjectReader.java
    │   └── NetObjectWriter.java
    └── server
        ├── Game.java
        ├── Player.java
        ├── Server.java
        ├── ServerActiveModel.java
        ├── ServerPongModel.java
        └── ServerPongView.java
```

Class Implementation

NB Most classes have been renamed and their code refactored so that they are more readable and easy to understand.

Client \ Client.java

The client class controls the game for the player. It initialises the client-model, -view and -controller. I attempt to connect to a socket using the predefined constants and if it succeeds then we create a new player and start it.

Client \ ClientPongController.java

Pong controller, handles user interactions. First I check that the player has been assigned a positive integer. This designates that it is initialised and can play. I only allow the player to move their own bat. The movements of which are sent to the server.

Client \ ClientPongModel.java

Model of the game of pong. I added the variable *playerNumber* and the respective getter and setter methods.

Client \ ClientPongView.java

Displays a graphical view of the game of pong. I only made minor changes to this class besides the refactoring.

Client \ Player.java

The player is run as a separate thread so to allow the updates to happen immediately. The player on the client sends updates to the server and then receives updates from the server and updates the model.

Common \ DEBUG.java

The debug class was provided by Mike Smith. I have not edited this class in any way.

Common \ GameObject.java

An Object in the game, represented as a rectangle or ellipse. It holds details of shape, plus possible direction of travel. I haven't edited this class other than to refactor it so that the variable names are more meaningful.

Common \ Global.java

The global class holds all constants that are used throughout the game. They are referenced in many of the other classes. I have edited this class and it differs from the provided class

Common \ NetObjectReader.java

I did not edit this class other than to refactor it.

Common \ NetObjectWriter.java

I added *reset()* to the synchronised method and refactored the code. The reset function was needed otherwise we would keep writing the same object. As I used a string it would be cached.

Server \ Game.java

I have added this class entirely. It runs in its own thread so to allow multiple games to be played at once. The code that fires up all of the ball movement is blocking, so it cannot be run in the server class otherwise it would terminate immediately.

Server \ Player.java

I extracted the player class from the server class file into its own file. The player class on the server receives updates from the client and updates the model with the new data.

Server \ Server.java

The majority of code that was provided to me in this class has been moved to the Game class as creating a new game blocks while it waits for two players. This way I can keep the server open, waiting for players and each time two players are connected a new game is created. I attempt to assign a new server socket. If I assert that this has occurred then it is enters into a *while-true* loop where I instantiate unlimited game objects upon request, constantly accepting new players for new games.

Server \ ServerActiveModel.java

I made only a couple of changes to this class (besides refactoring it). I added a method *processUpdates*

Server \ ServerPongModel.java

I added an ArrayBlockingQueue *updateQueue* to store the bat movements from both players and two methods *queueBatUpdate()* and *processUpdates()* to handle them. The first method *queueBatUpdate* takes a string as an input from the *Player* class on the *Server* and *offers* it to the *updateQueue.* The offer method does not block if it fails. The second method *processUpdates* blocks while it processes the bat movements. It does this by utilising the *drainTo()* method which passes in an ArrayList *updates*. I then check to see that there are three items in the ArrayList (a player number, an *x* position for the bat and a *y* position for the bat). If this *is* the case then it will set the *gameObjects* accordingly.

Server \ ServerPongView.java

I have edited the *update* method in this class. It gets the bat and ball from the server model and concatenates a string consisting of the bats and ball positions along with the player number appended to the front. The string is sent to the *NetObjectWriter.*

Critique

Issues

With my solution, when a game is being played all is well. The server is running and two opponents are playing. When one of the players quits the game the other player is terminated. This is to be expected as they no longer have someone to play against. When multiple games are being plays simultaneously an issue arises when one of the players in *any* game quits, *all* other games are then terminated. This is owing to how I handle exceptions that are thrown when a client cannot be found. I *assert* everywhere which will throw an exception when all is not right. A better way to have done this would be to catch the exception and handle this gracefully by disconnecting the faulty player, the other player and ending the game in question, leaving the other games to continue.

Concurrency & Real Time

I deal with real time and concurrency by making use of a thread safe collection and posting updates to it. This ensures that there are no deadlocking issues. If the collection is being mutated (when draining it in *processUpdates()*) it will block any other access to it until this process has completed ensuring that everything is safe regardless of how the threads interact with one another.

Fairness

One player may be nearer in time to the server than their opponent and this would have an effect on the time it takes to update their bat movements. In this project I have not implemented means to combat unfairness as both players will be run on the local network for demonstration purposes. I would however take it into consideration if I was developing the pong game to be played over a wider network. I would send the round trip time to the server in string update from the client. The server would then compare the times from both players, calculating the difference and send updates back to the client (player) accordingly.

Improvements

I have not completed version three which would have allowed for spectators. I would have implemented this by using multicast. The server would send out game data consisting of the players' bats and the ball position. This would become trickier when multiple games are in progress as I'd also need to keep track of which bats and balls belong to which game. An option could be to allow the spectator to choose which game they would like to watch if there is more than one in progress. Another option would be to send out a string for each game in the format of *gameNumber, bat1PositionX, bat1PositionY, bat2PositionX, bat2PositionY, ballPositionX and ballPositionY,* note the added *gameNumber* at the beginning. This would allow the spectator to view many games at once.

I would have liked to make the game actually work. By this I mean that currently it is a shell of the game pong but scores are not kept and the game does not finish. Points are not won when the ball hits the opponent's edge of the screen.

Collisions on the top and bottom of the bat produce a peculiar behaviour. The ball appears the bounce around inside the bat.

The bats can disappear off the window. This could easily be remedied by checking when the player moves their bat that there is actually space to do so.

Version One

I opened a port on the server to allow incoming connections, connected a client to it and passed messages of type string to it and output them to the console. This proved that version one was successful and I could start sending the bat and ball positions to and from the server.

Version Two

To allow for multiple games to be simultaneously played using the server. I created a game class that took most of the code out of the server class. The server class is used to instantiate new games and game objects without blocking.

Data Structures Report

Sequential processing is when all executions happen one after. This would occur in one thread therefore only one processor is needed to run the algorithm. It doesn't make a difference if the computer has multiple cores as the algorithm would still run in order.

Concurrency in a computer system describes a process as an independent entity with its own address space and state. It can have multiple concurrently executing threads. A thread is a separate execution within a process. An example of this would be a web browser, a user types in a url and the server loads in all of the resources simultaneously.

Advantages

Advantages of concurrency is that you can do multiple things at once. The user perceives this as the program running faster. If you have two threads then the program could potentially take half the time. If there are four threads then the programs could complete in a quarter of the time it would take had it only be ran in a single thread. We can make better use of time as we do not have to wait before starting a new task or returning to an existing task. We may be able to execute several parts of a program simultaneously and reduce the realtime that a program takes.

Issues

One of the issues that can occur when working with multiple threads is that if two or more threads need access the same resource at the same time to update it, which one gets it? If they all try to mutate the resource simultaneously it can become corrupted. Concurrent algorithms can be more complicated than sequential algorithms as it they need to take this into account. The time it takes to execute an algorithm is not always reduced. Communication between the threads is needed.

Another issue could be that events may not occur in the same order upon subsequent executions owing to non related activities out of our control.

Concurrency in software can lead to many new ways of making errors in a program.

Deadlock occurs when the system can't continue as access to a resource is required and it will never be released.

Livelock is a similar to deadlock in which various processes are changing constantly but never get to a point where they can proceed.

Starvation arises when a part of the system is never allocated any resources because other parts are running.

Priority Inversion occurs when a low priority thread prevents a high priority thread from executing.

Solutions

There are different methods to solve the issues brought about by using concurrency. You could implement a control thread which oversees the other thread and passes control of resources between them.

Priorities can be assigned to different threads and they execute in that order. If two or more threads have the same priority, run them in turn.

Transactional Memory

You can think of transactional memory like a transaction in a database. It can be used to produce a lock free system. An algorithm is lock free if the suspension of one or more threads doesn't prevent the progress of any of the other threads. Using conventional methods, writing lock free code is difficult and can quickly lead to issues.

Transactional memory attempts to make changes to shared memory. During this it will either commit those changes of roll-back and discard them. If the changes are discarded for some reason, it will try again until it succeeds.

Small sequences of code can utilise the principles of transactional memory when they would historically require a lock to prevent other threads from accessing the shared data.

Conclusion

Transactional memory definitely comes out on top as no locks occur and the data doesn't get corrupted when resources pass it from one to another.

References

Smith, M. 2014. Client Server / Concurrency. [pdf] Brighton. University of Brighton, School of Computing, Engineering and Mathematics http://www.cem.brighton.ac.uk/staff/mas/pdf/ci346-cs-2013.pdf Last accessed: 2 Apr 2014.