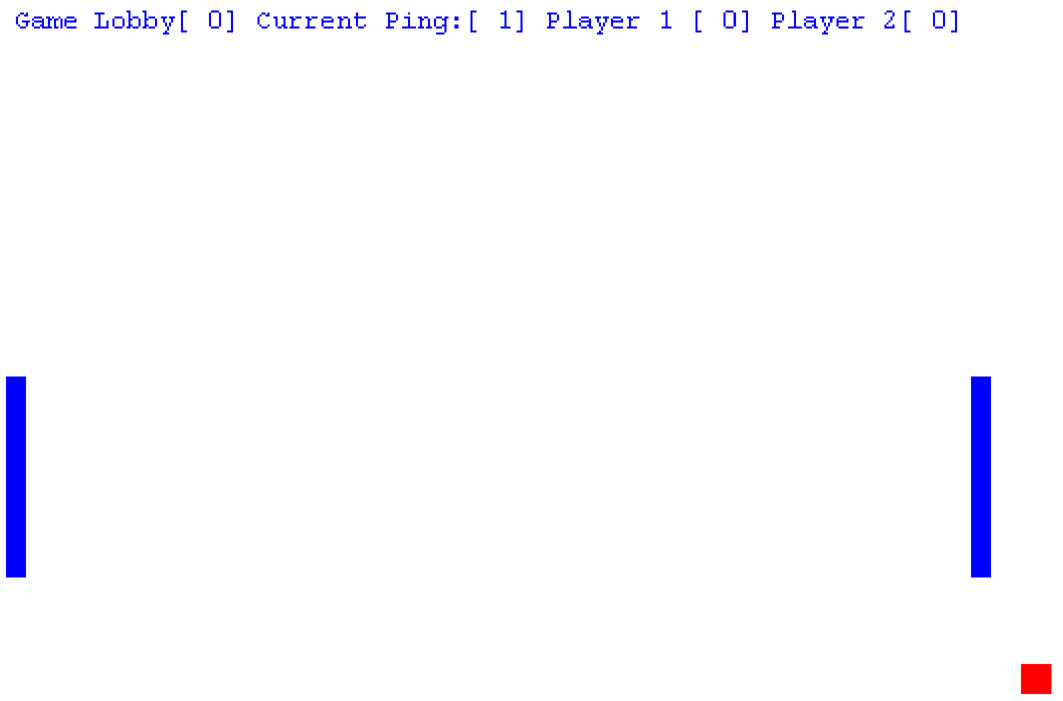
CI346 - Programming Languages, concurrency and client server computing

Simon Date - 11818785
Computer Science



Network Pong - Documentation

Contents

Outline of documentation 3

How issues are handled 3-4

Overview Critique 5

Version 1 Critique 6

Version 2 Critique 7

Version 3 Critique 8

Lock & lock free data structure comparison 9-14

Overview and Version

For this assignment I was required to create a pong that uses network functionality to enable multiple players to play the game together, over Multicast and TCP/IP technologies. I have handed in ***Version 3*** which includes all the features of version 1 and 2.

Program Listing

Available to view in the src directory in this folder.

Javadoc

Available to view in the doc directory in this folder.

Running the game

The game can be run using the Jar directory in this folder. Instructions on how to run are avialble within the README.

Issues addressed

To create this system I had to confront several issues that can often be very problematic when creating a network based application. Most noteable the way that the system deals with concurrency, having multiple users access entering data simultaneously; Real time systems, how the game deals with sending data constantly; And latency fairness, how the game deals with a player who has slower connection to the server then the other.

Concurrency

Concurrency in the system allows for multiple threads. Multiple threads are needed for a multiplayer game that works over a network. So that users can send data, while simultaneously being able to receive data. My implementation involved having lots of different objects that do not have knowledge of the operations of the others. For example the S_PongView class creates two NetObjectWriters at construction that allow for the server to send data simultaneously from the server to the clients.. The server can receive messages from these players, who both can only move their own bat, to prevent data being accessed by two people at the same time and update the game incorrectly. The system is set up so that no two threads can try and compete for the same resources. For the multicast implementation only the Server writes data. The Observers all instantiation their own listener and wait listen for the game messages. Concurrency for the game lobbies is created by attaching the Game Lobby number before the message so that the observer can decide if the message is regarding their game.

Real time issues

Another issue that I had to plan around is how to ensure that the system will work in real time. A real time is system that will automatically change after a certain period of time, regardless of what input the users have entered into the system. Real time data structures can not always process operations as fast as new data is added. In a usual system a buffer is needed so that this data can be processed later, however this approach is not correct for a game like pong. Because a game movement is sent late, or if the game state is updated too slowly. It will lead to an unplayable experience for the player. It is up to the programmer to limit the amount of data that is being sent over the network. To ensure that this buffer is never reached and the game is always playable. If it is reached the game will become laggier, the longer the game runs for.

Latency fairness

Latency fairness is the issue of ensuring that both users who are playing the game remotely from the server have a fair experience playing the game. As data is sent to and from over a network latency is added, due to the time it takes for a message to be sent and received, being much greater than processing data locally on the same machine. I have tried to control this issue by measuring the amount of latency that is being added to every message sent, to each of the users playing. Storing this as integer on the server. Then comparing the two to see which user has the greater latency and add then create an artificial delay for the opposing player so that they are both receiving messages simultaneously. Ensuring a fair game.

Critique of implementation

Overall critique

I used the framework that was provided to create my system. The game runs on a Server-Client system. Where the player of the game is running on the Client and the game itself is hosted on the server. The client acts as a thin client, merely accepting instructions and rendering them. The running of the game itself is done on the S_ActiveModel class which moves the ball, and detects if the ball has hit either a wall or a player. If the ball hits a wall on the left or right it updates the game to register the other player scoring. This is done on the server side so that the client acts as a thin client that does no logic itself.

This framework used the Model View Controller (MVC) on both the client side and server side. The Model, stores the state of the game. The View, on the client side, handles the rendering of the game using Java swing, a Java GUI. On the server side the View sends the games updates to the client. The controller is used on the client side to receive keyboard input from the user to the server. On the Server this controller can be used to manage the game, however I have not implemented feature.

I have not had an opportunity to test the game running over a non localhost network. However during the labs for the module we sent messages over a network to other computers running the same program. So all it would require is for the clients to change the location that they connect to in the Global.Java source file by editing the HOST and PORT variables.

Version 1

The first version of the game requires the running of the application using networking. I implemented this using TCP/IP, with the game using localhost.The network is setup by the use of Socket on the Client side and ServerSocket on the Server side. The ServerSocket has the port that game is running on and the client has the address of the server as well as the port it connects on. The client then uses this socket and passes it to a Player class which has a NetObjectReader and NetObjectWriter class which uses the socket to send and receive messages to and from the server. These two classes were created by Mike as subclasses of ObjectInputStream and ObjectOutputStream which have methods which are easier than using their superclasses implementation.

The loop of the game involves a player seeing the current game state on their monitor. They will press a button to move their bat to where they want it to go. This message will be sent to the server side. Which receives the input from both players. As well as updating the model according to player input the model also updates the game which moves the ball. Every time that the ball moves the new game state needs to be sent back to the clients to re render.

For example, game movements contain a String with the movement of either up or down according to which key was pressed. The server has each of the players of the game running in their own thread. So they can determine which player sent the movement and update the model with their movement. The server needs to send the game coordinates back to the players. Including the ball's X and Y position and the bats Y position, as well as the games score. These are stored as doubles and ints on the game server so need to be converted into String format using a method then when back on the client side need to be converted back before being updated the model on the client side.

Classes with code implemented

- **Client-** Connects to server
- **Player-** Running loop that gets game state
- **C_PongModel-** Stores state of game on client side
- **C_PongController-** Sends player movement
- **Utils-** Methods for converting string and double arrays
- **Server-** Enables clients to connect
- **S_PongView-** Sends messages to server
- **S_PongModel-** Saves server state.

Version 2

The player second version required two new features to be added. Including the ability to have many games to be running on the single server. This can be easily implemented by adding a infinitely running while loop that allows for any amount of Clients to be opened and paired together for a game. New instances of all the classes need to run such as Server view and model are created for each one, so that they all run independently of each other.

The second feature was the ability to display time it took for a message to be sent to the server and for it to be received back to the client. I implemented this as a check that is performed at a certain interval by sending a message from the client that contains the current time, the current time is got by calling System.currentTimeMills(). The client knows that this is a latency check because it reads it of being type Long rather than a String for a message. It sends the time back. The client will then receive a Long message on their side and then will get the current time and minus the time from the message. The difference between the two is the latency of the system. The latency is updated in the model and shown to the user.

Using the knowledge of the latency I can make the game fairer for the player that has worse latency by adding artificial latency to the other player. The server model stores the latency of both players. For the one that has less it creates a wait before it sends it to the other. that is equal to that of the other player minus the lower latency player. This ensures a fair game for both players.

In my second version I also added the passing of the score to the player. This is calculated by detecting in the ball hits the side of the game screen. If it does it adds a point for the player on the opposite side. This score is passed with the game data in the double. Although If I had more time I would have liked to find another way to implement this because it seems wasteful of resources to constantly send this when it only changes quite infrequently.

Classes with code implemented

- **Client -** Enables multiple games
- **Player -** Sends request for game ping
- **C_PongModel -** stores the game lobby number, goals
- **C_PongView-** Updated to show ping and goals.
- **Server-** Loop allows for many games
- **S_PongView-** sends game score, calculates delay for players before sending.
- **S_PongModel-** Methods for saving and retrieving goals
- **S_ActiveModel-** Detects a goal

Version 3

The third version enables users to be able to be able to spectate games using multicast. Multicast is a different way to network that writes to a ‘group’ rather than a single other socket that will receive the message. This allows for many users to listen in this group and all receive the same message simultaneously. Unfortunately I did not manage to implement the option for players to receive game updates over multicast, they can only receive over TCP/IP

The server S_PongView sends a multicast message as well as the usual TCP message to the players. This message includes the number of the lobby at the start of the message. As well as the actual message itself.

A Observer class has been added that has it’s own Main class so that it can be run. To run an argument must be included which is the game that the observer wants to watch. So 1 will observe game lobby 1. The observer is setup similarly to the player by creating a model and view. However a controller is not created and instead of the TCP being created by connecting the socket to the server socket the multicast is set up Meaning that the observer can only watch the game and not interact with it. Once everything is set up a loop is run listening to the the multicast group. Every message is checked for the game lobby number, which is found as the first character. If it is the message number is the same as the game lobby then the message is read and converted into doubles which update the observer’s model.

Classes with code implemented

- **Observer-** New class that creates an observer for a game.
- **S_PongView-** sends messages over multicast to observers.

Lock and Lock Free Data structures

Introduction

While a programmer is creating a complex computer system that involves several simultaneously running threads that are competing for a fixed resource, he must be extremely wary of the issues that can arise when multiple users try to access a data structure at the same time and try to write over the data that currently exists. If a change happens then it means that the information that one user may have read will be obsolete, if they use this obsolete information to try and make a change to the system then the system is likely to reach an incorrect state. For example if two users both try and iterate on a integer inside a data structure at the same time. They will both read the current value together. The user who submits their write change first will iterate the value. When the second user submits their iteration it will result in them only returning their change. This will result in the variable stored in the data structure only being iterated once, when it should have happened twice. This is obviously an important issue that needs to be addressed by a programmer. He can try and eliminate the risk of incorrect data by the use of either Lock or Lock free data structures to control how the system deals with concurrency.

Locked data structures

Since this started becoming an issue that programmer have had to deal with. The most common approach to solving the problem was the use of locked data structures to synchronize access to these shared resources.

Locked data structures work by using locks to allow only a single user to access a data structure. This works by allocating the lock to the user who wishes to access it. Once a user has access to it, he is the only user who may be allowed to perform operations on the data structure, a form of mutual exclusion to that part of the code in the system. If other users try to access the data structure while it is still being used by the first user, they will be denied access. Once the initial user has finished with the data structure, the thread relinquishes control of the lock and the second user is given the lock so that he or she may use the data structure. There are several different types of locks that a programmer may use. These vary from the simpler ones that are the easiest to implement to the more complex ones with more features that make them more useful to the programmer.

Semaphores

The easiest lock to implement and the first solution to the problem to be widely used is a semaphore lock, invented by Dutch Computer Scientist, Edsger Dijkstra**1** . A semaphore operates by allocating locks for every single data structure that is synchronised in the system. It allows multiple users to access them once. If a user is accessing the resource they are given the lock to the resource and becomes the user able to access the data structure until they signal that she is done. There is an integer value that controls the number of instances of the resource that are made available to be given to threads. Every time a resource is allocated the integer is decremented. If a resource is returned to the pool then it's number is incremented again. If the resource is demanded so much that the number has become a negative then there are no more instances of the resource available to be allocated. The system will place users waiting in a queue or use round robin scheduling until a resource is made available.

Semaphore can be used in Java by importing the java class Semaphore by calling java.util.concurrent.semaphore;
This class contains methods that allow the user to attempt to acquire(aquire()) a lock if they wish to access the data structure or when are finished using one release it (release()) so that another user can access the data.

Benefits and drawbacks

The benefits and drawbacks of the semaphore implementation lie in it's simplicity. Due to it being the easiest form of data locking it can usually be implemented into a program with very few lines of code. Which is obviously of great benefit to the programmer, saving a lot of development time. However there are some very critical downsides to the system. Firstly that the semaphore lock makes no distinction between a user trying to access it in read only mode, a user who is not making changes to the system. Or a read and write only mode, that allows the user to update the values inside the data structure. Computer systems do not need to worry about users who are only intending to read the data to be synchronised because there will be no change in data. As such the fact that the Semaphore can not distinguish between these two circumstances means that the system is potentially running a lot less efficiently than it could have been.

This type of data structure may be appropriate in a system where the majority of the accessing being done by the users are both read and write operations. So that the downfall of not being able to distinguish between read or read and write operations is reduced.

Spin lock

Another type of lock based data structure that can be used by a programmer is the Spinlock. If the data structure is currently being used by another thread the Spinlock places the other thread trying to access the data structure inside of a loop. The loop checks at a certain interval that is decided by the programmer if the lock has become available. Once the other thread has relinquished control of the data structure, either voluntarily or forcibly by the system the next time the other thread loop checks for availability it will be granted it assuming that it was at the front of the queue for it's availability. Due to the fact that spinlocks are constantly using resources to 'busy wait' or loop continuously. It means that the computer has to dedicate processor time to ensure that this loop is running until fulfilled either by the lock being given to the thread or by it being manually given up on. Spinlocks are often seen to be more useful in systems where the interval between waiting and accessing the data structure is short. This means that the system will quickly be able to inform the user when they will be able to access the data.

Lock Benefits

Locked data structures in general can be stated to several benefits. They are usually quite easy for the programmer to implement. Usually with a few lines of code that can be accessed through a library that exists. In Java objects have methods built into them for features such waiting and notify. That allows for them to be controlled by threads to allow or disallow access by a simple method call. A programmer can also take advantage of thread priority to try and ensure that important threads will have higher priority to a data structure. In systems with limited cores. A lock based data structure is often more optimal than a lock free based one due the resources it requires to maintain a lock free system.

Lock drawbacks

However they also have several drawbacks that make them a less compelling choice for a programmer to choose as they're way to deal with concurrency in data structures.

Deadlocking

Firstly they are highly prone to deadlocking. Deadlocking is situation that can occur in software where multiple threads in a system are waiting for an event that can only be fulfilled by another thread in the system. That in turn can not be completed until the first thread has been fulfilled. The two, or more, threads reliance on each other results in **2**. To give an analogy imagine two trains approaching each other on a single line of rail. They both stop before they collide. However they have no way to continue, since they both require complete control of the rail to either move on and let the other pass. As such they are both stuck forever unable to move. Deadlocking can arise when four conditions **3**, the Coffman conditions, are met simultaneously in a system: Mutual Exclusion, or at least one resource being held in a non-shareable mode. Hold and Wait, a process is currently holding at least one resource and is requesting additional resources. No Preemption, the system must not deallocate resources once they have been allocated, they can only release resources voluntarily. Circular wait, a process must be waiting for a resource which is being held by another process, which is waiting for the first process to release the resource. If anyone on of these are not met then they system will not enter a deadlock state. However since a lock system will often rely on at least several of these states happening. It is quite often that a situation may occur where they are all happening.

Convoying

Another problem with using locks is in convoying. This problem occurs when several threads of equal priority are contending for the same lock. Each time a thread tries and fails to acquire the lock it leads to a context switch. This will lead to a loop where this happens many times. Although the system will eventually let get the lock first. It leads to a severe performance hit.

Lock Free data structures

The alternative to a lock based system for preventing multiple simultaneous editing of a data structure is the use of lock free data structures. A data structure can be classified as lock free if it's operations do not require mutual exclusion to allow a system to push and pop items from the stack.**4** Lock free data structures avoid many of the common problems associated with conventional lock techniques.

Transactional memory

Transactional memory is a way to prevent overriding of data without using locks. It can either be implemented at a software or hardware level. At both levels it requires the creation of transactions. Which are a series of instructions to perform on the data structure. Once these instructions are gathered it is compared to what other cores in the system have in their transactional memory. If no conflicts are found then the changes are committed to ram. If a conflict is found then the transaction is discarded and started again.

Hardware

Implementation overview and RISC processor

Transactional memory can be implemented at a hardware level, to allow for lock free, atomic read and write operations to memory locations.Through the use of the Load-link and store-conditional (LL/SC) pair of instructions. The LL/SC works by returning the current value of a memory location when it is called. The store condition can only be applied to the memory location if no updates have occurred to that location since the load link. The RISC processors support the LL/SC instruction set.**5** They have also several design features that have made them able to deal with being able to deal with multiple threads accessing data. Including pipelining that allows for simultaneous execution of parts and a large number of registers that prevent in large amounts of interactions with memory.

Haswell Processor

In more mainstream computers, the newest intel chipset haswell, released summer 2013 is the x86 processor to have implemented hardware support for transactional memory**6**. Through Intel's Transactional Synchronization Extension (TSX)**7**. The extension provides two interfaces for a programmer to take advantage of. The Hardware Lock Elision (HLE) is used to be backwards compatiable with non TSX supported processors. It does this by adding two instruction prefixes, XACQUIRE and XRELEASE. Which override existing instructions to provide a lock free system. The other interface, Restricted Transactional memory (RTM), is an alternative that gives greater flexibility to the programmer compared to HLE.

Software

Transactional memory can also be implemented at a software level in the form of a non-blocking algorithm. It can be called analogous to database transactions which are used to ensure that databases can not be wrongly updated by users accessing them simultaneously. Similarly to the way that hardware transactional memory is implemented. A ‘transaction’ is created which contains a series of reads and writes which are to be committed atomically, assuming that the reader can confirm that the data has not changed since the transaction was created. In which case the transaction is discarded.

Software Transactional Memory runs quite optimistically in term of conflicts not occurring, that is it assumes that it will often not have to compete for resources. The amount of time to fix a transaction that can not be committed is much greater than that of a lock based data structure. The risk of a conflict is reduced by having more cores in a computer system processor.

Software lock free data structures has native support in a few programming languages such as Clojure **8.** The majority of languages has at least some support. Java has a number of extensions that allow for a programmer to create lock free data structures for his system. Washington University has created an extension for java called AtomJava that allows for atomic statements to be created. **9**

Benefits and drawbacks

Transactional memory as a concept has a number of a benefits that it can bring to the table as an alternative to lock based data structures. Including the avoidance of the risk of deadlock and livelock, which can happen by the holding of locks by multiple threads. Transactional memory avoids these problems because each transaction is viewed in isolation as a single thread computation. Which means that a transaction manager can prevent deadlock .

This implementation at the hardware level is beneficial to a programmer as it means that he can write can take advantage of hardware that is purpose made to allow for transactions to be sent. This means that messages will be sent a lot faster then they would be at software emulation level. However this means that the system that is being developed for have to run on a processor that is built to support transactional memory. Although with Haswell supporting Transactional Memory. It seems that this will be less of an issue in the future.

Software transactional memory brings a lot of benefits to a programmer as he doesn’t need to worry about the hardware. He can just implement in the language he is using, either natively or through an extension, and know the the system will be lock free.

Lock free data structures are less compelling on a system that has a small amount of cores. This is due to the overhead associated with logging and committing transactions into the system.

Comparison & Conclusion

Lock based data structures can provide to be very volatile in a system due to the high likelihood to begin deadlocking, live locking or convoying which will all seriously harm the running of the system. Lock free data structures avoid the risk of by performing operations on the data structures atomically. Meaning that the series of transactions that are being committed must be sent together ensuring zero chance of erroneous data being stored whilst ensuring that locks are not used .I believe in the future, as hardware becomes stronger for both desktop and mobile, and more hardware has support for lock free data structures. I believe that lock free will become more and more of a superior choice.

However if A Lock-Free data structure can not be implemented due to time restraints on the programmer or resource restraints of not having powerful hardware that is capable of transactional memory. A lock based data structure is perfectly serviceable. Semaphore could be used, and is very easy for the programmer to implement in a system with multiple instances of data structures that can be accessed in the system. In a system where only there is extremely high risk of a conflict a spin lock may be more ideal. However it is critical t

References

1 EW.Dijktra Archives, accessed 22/03/2014
http://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html
2 Abraham Silberschatz, 2006, Operating System Principles, 7th edition, Wiley India Pvt
3 Avoid Java deadlock, accessed 22/03/2014
http://www.javaworld.com/article/2075692/java-concurrency/avoid-synchronization-deadlocks.html
4 Transactional memory: Architectural support for lock-free data structures, accessed 22/03/2014
http://cs.brown.edu/~mph/HerlihyM93/herlihy93transactional.pdf
5 What is RISC?, accessed 22/03/2014
http://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/whatis/index.html
6 Transcational memory going mainstream, accesed 22/03/2014
http://arstechnica.com/business/2012/02/transactional-memory-going-mainstream-with-intel-haswell/
7 Analysis of Haswell transactional memory, accessed 22/03/2014
http://www.realworldtech.com/haswell-tm/
8 Clojure home page, accessed 22/03/2014 www.clojure.org
9 WASP atom java page, accessed 22/03/2014 http://wasp.cs.washington.edu/wasp_atomjava.html