Mobile & Distributed Systems: Mandatory Exercises

13. May

2008

This document contains discussions about the four mandatory exercises for the Mobile and Distributed Systems course at ITU.

Covering the code

Group #1:
Jeppe Larsen
Christopher Pedersen
Andreas Nauta Pedersen

Contents

Introduction

All source files including the API to the implementations can be found online at:

http://www.bluenight.dk

The report (which you are reading), can likewise be found on the website.

Project 1: The Loopers Game

Implementation Discussion (covering 1-3)

As a general overview of the loopers game, it is simply implemented with a number of loopers situated within a grid that is of any possible size. These loopers, whose movement is controlled within themselves in separate threads, can move across the grid one step at a time and occupy spaces up to its own length. When a looper tries to move onto a new field in a grid, it tries to acquire a semaphore that is situated in that space. If that semaphore has been counted down (when another looper is passing over it), it waits until that looper has finished moving through the space. When a looper has finished moving over a space it, it releases the semaphore and thus other loopers can then acquire that space as its own.

A looper in our system started out occupying its length+1 at maximum when it was moving, however this has changed in the game as a looper in our game never occupies more than its space. This was due to the fact that it made logically more sense that a looper would coil up before protruding its head. However if it is critical that a looper stretches before pulling its behind with it, it is simply a matter of acquiring the semaphore in front before releasing the hind, instead of the other way around.

Deadlock, Livelock and Starvation (covering 4)

Deadlock

To analyze the deadlock situation of the loopers game, we can use the three conditions for deadlocks.

1. Mutual Exclusion
2. Hold and Wait
3. No Preemption

All of the above conditions are present in the normal loopers game, where the grid consists of semaphores that only one looper can access at a time (Mutual Exclusion) and the fact that loopers can only themselves release the semaphores (No Preemption). A looper also holds it resources until it moves away from them, however that is only the case because they have a length longer than one (in this looper game). In the normal looper game, where a looper extends itself before pulling, it would have to wait for another looper before releasing its tail semaphore. In the case of a coiling looper, it releases its tail before trying to access the front space (however a one spaced looper is hardly a looper). If in fact a one spaced coiling looper would be present, hold and wait would not be a problem, however since a looper continuously holds the middle body elements while waiting for passersby, hold and wait is still present.

Starvation

The implementation has a possibility for starvation if a looper starts it wait for another one, and just as it has passed another looper tries to acquire the same semaphore again. If this continually happens, then the first looper will be starved since it can never acquire. However a fairness rating should ensure that threads that have been longer in line will get time to access the semaphore resource.

Livelock

The standard loopers game would not have problems with livelock, however implementing a deadlock solution where loopers would move to another place if the square they initially wanted is occupied, might produce a livelock as two opposing loopers might choose the same alternate path and crash again, doing it continuously.

Resolving Deadlocks (covering 5)

The implementation of a deadlock free solution is tricky. Diverting a waiting looper would seem to end a lot of deadlock trouble that happens when more loopers want the same space, however if there are enough loopers in the vicinity, it might not be possible to avert to another location. However it would seem that the outermost loopers seeing to pass through will eventually choose another path and thus a mass deadlock would still be avoided.

Alternatively a kind of monitor that waits until a thread reports that it has been waiting too long, could kill loopers that hog too much space or loopers that have been standing still for a long time.

The looper game has been implemented with the first solution.

Project 2: TorusMan

Overview

The general idea of our implementation was to use simple TCP connections to hand out packages whenever they come in.

A TCP server can be started that waits for incoming connections. Whenever an instance of the game is created, the client can enter the ip address of the server (it is possible to leave the address space blank and localhost will be chosen as default), and then play a game that sends packages to the server which handles them accordingly.

The server hands the packages out to all the clients that are attached to it, and keeps doing so until it is closed. When it receives a package from a client it simply hands this package out to every client connected. This meant that whenever a new client is added, it receives packages whenever another client sends packages to the server.

The information that is sent between the clients and the server, uses the serialize functions that are already present on the relevant objects. However since a receiving end, client or server, does not exactly know what kind of information is passed to it, a byte is sent first before the information that signifies what kind of information the receiver can expect next. In this implementation there is a signal for ManData and for the points that are picked up when the Torus runs around in the game.

Files

- **SInitGame**
  Used to set up the game with our classes instead of the standard.
- **DistTorus**
  Used to determine what happens when a player utilizes certain functions on a client. Is used to start the sending of packages.
- **Network**
  Static class that holds information on signals and ports.
- **Connection**
  Holds information on a connection between a client and a server.
- **TCPServer**
  The server that start listening for clients and handles them accordingly.
- **ServerSender**
  Handles all outgoing packages from the server, to all connected clients.
- **ServerListener**
  Handles all incoming packages from clients to the server.
- **ClientSender**
  Handles all outgoing packages of the client.
- **ClientListener**
  Handles all incoming packages to the client.

Design Decisions

As of now the solution is a client server system that has package listeners and senders on both sides. The listeners and senders contain the relevant data stream to send or receive information. When servers and clients are started, a thread is created for all the incoming packages to them; these are the only two threads that are in our implementation.

Threads are only created for listeners because we felt it would flood to many packages on the connection if we created a new thread for all the sending information all the time as well. The implementation waits for the player to do anything before it sends packages, which means that whenever a player moves his Torus, a package is sent with the moved information. There is also sent a score package whenever a player picks up points.

The listener, that runs in its own thread for the clients, in turn take whatever packages are available on the connection and feeds them to the relevant functions, such as updating points or moving other Torus'.

The implementation could have been made with both sender and listener classes being threads, having senders loop and wait for commands to come from the player and sending information as soon as something has been fed to it that is send able, however this was deemed to be unnecessarily bothersome.

It is also worth noting that we decided to make the server just pass out information immediately instead of holding a backup of the information and sending that out regularly.

Conclusion

The implementation is not flawless, since that when new players arrive after someone has already moved on the board, but then stands still, no information will be sent to the new player that has just arrived. This means that only new data is transmitted to players that enter, which makes it practically impossible to enter a game in the midst of it.

There is also a problem with the positioning of the Torus', whenever someone leaves a board and goes onto another one. We are unsure if this problem is due to the fact that a board has to be sent at some point and received by other clients or if the before mentioned problem has caused this as well.

Overall the implementation receives the correct data and in the right order, however new connections are not kept up to date unless all players connect at the same time.

Keeping a history of the data on the server might help solve some of the score issues whenever someone connects in the middle of a game, since the overall score could be kept for each team and clients would send their scores that would be added to the total score, which, at an interval, could be sent to all clients and then be updated for them.

Project 3: Time Synchronization

Overview

As the time synchronization builds on the previous assignment, a few features were added to the code. Generally the server keeps track of its own time, counting it up accordingly. Whenever a client requests a time it sends a request to the server (every 5 seconds), and then continues to play the game normally for now.

When the server gets a time from a client it sends its own time back to the client. When the client receives the time from the server, it logs its own time and then computes the time skew according to Cristians algorithm.

The server handles its own time in a separate time thread.

Files

- **ServerTime**
  Handles the time of the server and continually counts up.
- **ServerListener**
  Has been changed so that it sends it's time to the client when it receives a time package.
- **ClientTimeSender**
  Sends a request for the server time every so often.
- **ClientListener**
  Has been changed to also fit in the third assignment. In the end computes the time skew when the time package comes in.

Design Decisions

Out design solution is pretty simple, elaborating on the above described the client starts it all by sending a request to the server that it wants its time. When the server receives the signal, it sends a time signal back to the client with the server time.

When the client receives the server time, it logs the time it received it. The round time is computed by subtracting the end time by the start time.

The time skew is then computed by halving the round robin time and adding the server's time, lastly subtracting the man's time.

$$timeSkew = \left(serverTime + \frac{roundTime}{2}\right) - manTime$$

Conclusion

The round robin time seems to be zero always. We think that this is due to the relative fast execution and sending time on localhost. The more positive side of the implementation is that it seems to show a realistic skew between the client and the server, which is the time difference between starting the server and the client connecting to it.

Project 4: Mobile TorusMan

Overview

The project builds upon the structure created in Project 2. We have made some changes however. We have tried to create project 4 as much in line with the philosophy we had in mind for project 2/3, but especially the client/server architecture had to be rethought.

Since no one wants to have a dedicated server on a mobile phone, we had to make all "clients" be both client and server in one. This basically meant that when the previous projects clients sent they're information and received it from one place (server), the clients now have to send and receive to and from all other clients in the system.

Files

- **BTListener**
  Receives information from all other players.
- **BTSender**
  Sends all the information to all other players.
- **DistTorus**
  The distributed torus that handles what happens when a player takes action.
- **Network**
  Contains the signals used for sending packages between devices.
- **Service**
  Contains an URL to a service that has been discovered.
- **SInitGame**
  Sets up the game to use our classes.

Design Decisions

Each game (player) now has both a server and a client and each game send information to all others. We have chosen to not implement this version as a proxy server. The program now runs with a thread for both the client and the server, the reason for having the client, that sends information, in a thread is basically because we need the client to sleep for 5 second before actually playing, in order to give each game a chance to start its server.

We chose not to make players able to join the game after it has started, but it would be possible to let players join during a game by letting each game re-listen for other devices. But we see the 5 sec as a good chance to do some pregame taunting of your opponents.

During the start up, each client registers the other servers and saves them in a Vector list. When a client needs to send something, we go through the vector one at a time and send information to all the other servers. The server side keeps listening for incoming connections, and when such an event occurs, it receives the packages and handles them accordingly.

Conclusion

The game, to some extent, suffers from the same problem as project 2 where players sometimes won't see each other, since storage of data over time is not done. However since we have made a case out of letting players join only under pre-game startup, the problem should not arrive too often when playing.

As said, an improvement to the system would be to hold information longer and by timed intervals send information. However the system works and players are able to play against each other on mobile devices.