Laborationsrapport

D0036D Nätverksprogrammering

Laboration: 4

Pontus Stenlund

Ponste-5@student.ltu.se

2017-05-28

Innehåll

Problemspecifikation 1
Deluppgift A 1
Deluppgift B 1
Deluppgift C 1
Användarhandledning 2
Systembeskrivning 2
Client 2
Player 2
Client 2
Controller 3
Model 3
View 3
Messages 3
Server 3
ThreadHelper 4
Design och genomförande 4
Meddelanden 4
Klienten 4
Server 5
Problematik 5
Lösningens begränsningar 5
Klienten 5
Servern 6

Problemspecifikation

Vår uppgift var att implementera ett nätverksbaserat spel (både klient och server) som stöder att flera klienter kan vara uppkopplade mot servern samtidigt via TCP.

Servern skulle implementeras i C eller C++ och köras på vår Linux användare. Klienten fick implementeras i valfritt språk.

Vi hade som krav att kommunikationen i spelet skulle följa specifikationen i SpelProtokoll.pdf.

Laborationen var uppdelad i tre mindre delar.

Deluppgift A
Vi skulle implementera en enkel klient som kunde ansluta sig till en färdig server.

Deluppgift B
I denna deluppgift skulle vi fortsätta med att ansluta oss mot den färdiga servern men implementera ett grafiskt gränssnitt i vår klient. Klienten rita ut spelets deltagare i 2D samtidigt som den kommunicerade med servern. Man skulle skicka ett önskemål till servern om att få förflytta sin spelare till en viss position som sedan bestämmer ifall det är tillåtet för spelaren att förflytta sig till den positionen.

Deluppgift C
I denna uppgift skulle vi implementera vår egen server och använda vår klient för att testa den.

Servern skulle ta emot ett meddelande från klienterna, processa innehållet och sedan skicka uppdateringar till de ansluta klienterna. Servern skulle hantera spelarkollision och så att spelarna inte får gå utanför spelplanens gränser.

Användarhandledning

Servern startas via Putty. När servern startas så letar den efter en ledig port som den sedan skriver ut på konsolen.

När klienten startas så ombeds man att skriva in vilken IP-adress och port nummer som man vill ansluta till. Om korrekt IP och port har angetts så anslutes man till servern och kan spela.

Annars kommer det upp en text på konsolen som säger att man ska kolla IP/port nummer och sedan ange rätt nästa gång.

Kontrollerna för att kontrollera spelaren är:

- W – För att förflytta spelaren uppåt.
- A – För att förflytta spelaren åt vänster.
- S – För att förflytta spelaren nedåt.
- D – För att förflytta spelaren åt höger.
- Escape- För att stänga ner spelet.

Systembeskrivning

Jag har strukturerat upp min klient på följande sätt. För att få ett grafiskt gränssnitt i C++ så har jag använt mig av ett plugin som heter SFML.

Client

Här finns alla meddelanden som klienten kan skicka. Client.h innehåller även structen Player och klassen Client.

Player

Player har dessa funktioner.

Player(), kontruktor för den egna spelaren. Ritar ut en cirkel och färgar den grön. Sätter x och y positionen till 0.

Player(int enemyID), kontruktor för motståndare. Ritar ut en röd cirkel, sätter x och y positionen till 0 och sätter id till enemyID.

SetPos(int xPos, int yPos), sätter x och y till de angivna positionerna och ritar sedan ut spelaren på den angivna positionen +100. Detta är för att det ska stämma överens med hur fönstrets koordinater.

Client

ConnectToServer(string ip, string port), ansluter klienten till angiven IP och port och skapar en tråd ifall anslutningen lyckas.

Listen(), lyssnar på klientens socket efter meddelanden från servern och beroende på vilket meddelande som den tar emot så utförs olika saker. T.ex. om den tar emot ett NewPlayer meddelande så ritar den ut en ny röd cirkel på spelplanen.

Send(int xPos, int yPos), skickar ett meddelande till server om den önskade positionen som man vill förflytta sig till.

Quit(), skickar ett LeaveMsg till servern att man ska lämna spelet och stänger sedan socketen.

ThreadListener(), är en hjälpfunktion för att starta tråden. Anropar Listen() varje frame.

Controller
Är klassen som har referenser till Model och View. Controller har dessa funktioner.

InitView(), anropar View.InitView().

Update(), registrerar ifall någon knapp på tangentbordet har blivit nedtryckt. Om WASD har blivit nedtryckt så anropar den Client.Send(xPos, yPos). Om Escape har blivit nedtryckt så anropar den Client.Quit() och View.CloseWindow(). Oavsett så anropar den Draw().

GetModel(), returnerar model.

GetClient(), returnerar klienten som finns i model.

GetPlayer(), returnerar player som finns i client.

GetList(), returnerar listan med spelare som model har.

Draw(), rensar fönstret från alla spelare som är utritade och ritar sedan ut alla spelare igen.

Model
Model har följande funktioner.

GetClient(), returernar client objektet.

GetPlayer(), returnerar den egna spelaren.

GetList(), returnerar listan med alla spelare som finns i client.

SetPos(int playerID, int xPos, int yPos), sätter positionen på spelaren med givet id.

Connect(string ipAdress, string port), anropar Client.ConnectToServer(string ip, string port) som ansluter till servern.

UpdatePos(int xPos, int yPos), sätter positionen på den egna spelaren som finns i client.

View
Denna klass sköter det grafiska i programmet. View har följande funktioner.

InitView(), skapar ett fönster med storleken 800x600.

Draw(sf::CircleShape player), ritar ut den angivna spelaren på skärmen.

CloseWindow(), stänger fönstret.

GetWindow(), returnerar fönstret.

Servern är uppdelat på följande sätt.

Messages
Innehåller alla meddelanden som servern kan ta emot. Dessa finns angivna i SpelProtokoll.pdf.

Server
Server har följande funktioner.

StartServer(), binder serverSocket på IP-adressen och letar efter en ledig port som den sedan skriver ut. Sedan ligger den och väntar på anslutningar. När den sedan får en anslutning så skapas en tråd

som sedan hanterar mottagandet av data från klienten. Sedan lägger den sig och väntar på anslutningar igen.

SendUpdate(char buf[], int size), skickar data till alla anslutna klienter.

GetThreads(), returnerar listan med ThreadHelpers.

RemovePlayer(int id), tar bort spelaren med det angivna id och joinar tråden till main tråden.

ShutDown(), stänger av servern.

ThreadHelper

Denna klass anses som spelare. Den innehåller en server pekare, x och y position och clientsocket.

ThreadHelper har följande funktioner.

ThreadHelper(int socket, Server* serv, int ID), kontruktor som tilldelar socket, en serverpekare och ID till objektet.

Recieve(), det är här som tråden ligger och lyssnar efter data från klienten. När den får data från klienten så beroende på vad det är för meddelande så anropar den olika funktioner som finns i Server. T.ex. om tråden tar emot ett Leave message från sin klient så skapar den ett ChangeMsg med typen PlayerLeave och anropar sedan server.SendUpdate() och sedan server.RemovePlayer(). Jag använder mig också av mutex för att förhindra att jag får deadlock när jag utför vissa operationer.

Design och genomförande

Jag implementerade klienten i C++, och använde VirtualBox och CodeBlocks för att skriva början av servern i C++ för Linux. Sedan när jag kommit en bit och det bara var små ändringar som behövde göras så använde jag mig av Notepad++ för att ändra och sedan kompilerade jag via putty.

Meddelanden

Både servern och klienten har en headerfil där alla meddelandetyper och deras egenskaper finns lagrade. De meddelanden som jag använder mig av är:

JoinMsg – Klienten skickar det så fort den har fått kontakt med servern

NewPlayerMsg – Skickas från servern varje gång en ny klient har anslutit sig till servern och servern har fått ett JoinMsg. Används för att meddela alla andra klienter att de ska rita ut en ny spelare.

NewPlayerPositionMsg – Skickas från servern till alla spelare varje gång som en spelare har begärt om att få förflytta sig till en godkänd position.

LeaveMsg – Skickas från klienten till servern när den vill lämna spelet.

PlayerLeaveMsg – Skickas från servern till alla klienter när servern har fått ett LeaveMsg från någon spelare.

MoveEvent – Skickas från klienten till servern när den vill förflytta sig.

Klienten

Jag byggde upp klienten så att den skapar en tråd som tar emot data från servern medans huvud tråden skickar data till servern, hanterar input och utritning på skärmen.

Tråden som tar emot data går igenom buffern som tas emot, kollar sedan på vilket MsgHead som meddelandet har och utför olika arbete beroende på vilken typ som meddelandet har.

När klienten vill skicka ett meddelande så skapar den det meddelande som behövs, lägger det i buffern som sedan skickas till servern.

Server

Servern lyssnar konstant efter nya klienter som ansluter. Sedan skapas en tråd för varje klient vars uppgift är att ta emot data från klienten. Sedan beroende på vilket meddelande som tas emot från klienten så skapar tråden ett nytt meddelande, lägger meddelandet i en buffer som sedan skickas till alla klienter.

Problematik

Det största problemet för mig i denna laboration var att skriva kod till Linux. Eftersom Visual Studio för Windows inte har de bibliotek som krävs för att kompilera koden till en körbar fil i Linux så laddade jag hem VirtualBox och en Linux distribution för att kunna få tag på de bibliotek som jag behövde. Sedan var det ett problem i sig att skriva kod på VirtualBox eftersom det var ett litet uppehåll från det att man tryckte på tangenten tills texten kom fram på skärmen.

Sedan när jag började så pass mycket kod att jag kunde kompilera till en körbar fil så var det ett litet problem att behöva maila filen till mig själv för att sedan kunna lägga den på min Linux användare.

Det blev betydligt lättare när jag sedan bara behövde göra små ändringar i koden och jag började använda mig av Notepad++ för att skriva och sedan putty för att kompilera koden. Det underlättade arbetet och jag ångrar att jag inte började laborationen så.

Bland de problem som har uppstått genom koden så tror jag att det största problemet har varit att få mutex att fungera som jag viljat. Gjorde inte nog med efterforskningar på hur man skulle göra vilket gjorde att varje gång jag ville låsa tråden så fick jag ett fel som gjorde att programmet kraschade. Sedan läste jag på lite mer noggrant och insåg att jag aldrig initierade mutex.

Ett annat problem som jag har haft är att jag har varit klantig med att hålla reda på vilka filer jag har jobbat med. Satt och ändrade i den rätta koden trodde jag men jag fick alltid samma fel. Felet var att jag skickade position -100,-100 till den nya klienten som anslöt men av någon anledning så i klienten så tolkade jag det som 36,0. Sedan insåg jag att jag satt och förde över backupfiler som var föråldrade.

Lösningens begränsningar

Klienten

En begränsning som jag kommer att tänka på är att jag bara skapar en tråd som hanterar mottagandet av data. Sedan när jag skickar ett meddelande till servern så hanterar huvud tråden det. Det är lite mindre effektivt eftersom huvud tråden då ska hantera input från användaren, rita ut på skärmen och skicka meddelande till servern. Det skulle ha kunnat bli mer effektivt ifall jag skapade en tråd vars uppgift var att skicka meddelande till servern. Det är inte en stor minskning av effektivitet men det skulle ha varit effektivare med en dedikerad tråd som skötte skickandet av meddelande.

Servern

En begränsning med servern är att jag bara kan ha tio spelare anslutna samtidigt på servern. Om det är tio stycken anslutna och en elfte vill ansluta så tar den emot anslutningen men den skapar ingen tråd för den och sedan ignorerar den all data som den klienten skickar.

En annan begränsning jag har är att jag använder mig av nästlade loopar för att kolla ifall den önskade positionen är okej att flytta sig till. Det sköter de separata trådarna om och att som mest kan det vara tio anslutna klienter, vilket inte innebär en allt för stor minskning av effektivitet men som kan vara värt att ha i åtanke ifall jag någon gång ska implementera en server som ska kunna hantera ett större antal anslutna klienter.

Sedan skulle jag ha bestämt en port som är dedikerad för servern. Då skulle det vara lätt att hålla reda på. Nu så letar den efter första bästa lediga port som den kan sätta sig och lyssna på.