

Hemuppgift 2: Non-Blocking Sockets

Nätverksprogrammering, ID1212

Evan Saboo
saboo@kth.se

2017-11-23

<https://github.com/ZpeedX/Network-Programming/tree/master/Homework%202>

1 Introduktion

Målet med andra hemuppgiften är att kunna utveckla en applikation som kan kommunicera med en server över icke-blockerande TCP eller UDP sockets, medans servern ska kunna hantera flera kommunikationer samtidigt.

Applikationen ska också ha en strukturerad arkitektur för att kunna dela upp logik och användargränssnitt.

Till sist ska applikationen kunna vara användarvänlig, dvs. den ska använda flera trådar för att göra användargränssnittet responsiv och för att dölja kommunikationsfördröjningar.

Uppgiften bygger på första hemuppgiften men nu ska man använda icke-blockerande sockets och hantera trådar på ett annat sätt för att förbättra skalbarhet och prestanda.

2 Litteraturstudie

Uppgiften utfördes med hjälp av dessa källor:

* Jag fick mest hjälp från icke-blockerande sockets föreläsningarna. Koden jag har skrivit har fått mest inspiration från chatt exemplet i föreläsningen.

* Jag använde mig av StackOverflow hemsidan för att få hjälp med problem som uppstod och för att få kort och enkel information om några java syntaxer.¹

* Java dokumentationshemsidan var också hjälpsam för att få Java syntaxförklaringar.²

¹ <https://stackoverflow.com>

² <https://docs.oracle.com/javase/7/docs/api/>

3 Metod

Det var inte mycket att gå igenom i föreläsningarna eftersom bara kommunikationen mellan klienten och servern och multitråd hanteringen skulle ändras i applikationen. Men icke-blockerande sockets är inte så lätt att implementera som blockerande socket. Man behövde lära sig om buffers, selectors, selection keys och socket channels för att kunna hantera flera klienter samtidigt.

Föreläsningarna var mycket hjälpsamma eftersom chatt exemplet var ganska lika applikationen jag skulle förbättra. Jag behövde bara förstå chatt applikationens implementation av icke-blockerande sockets för att kunna implementera det i min applikation.

Det första problemet applikationen i första hemuppgiften var att den använde blockerande sockets. Användning av blockerande socket innebär att endast en socket kan vara aktiv när som helst i någon tråd (eftersom den blockerar i väntan på aktivitet).

Med icke-blockerande sockets kan du hantera en mycket större mängd klienter: det kan gå upp till hundratusentals i en enda process - men hanteringen blir lite mer komplicerad.

För att implementera icke-blockerande sockets med multi trådar behövde jag tänka om hur klienten kommunicerade servern. Jag behövde använda socket channels vilket gjorde att man kunde konfigurera dem till icke blockerande.

Ett annat problem med applikationen var att om flera klienter var kopplade till servern skulle servern behöva en tråd per klient. Detta skulle resultera i flera trådar som körs samtidigt och då skulle operativsystemet behöva göra flera "context-switching" vilket är sega ner varje tråds snabbhet, och dessutom tar varje tråd upp minne.

Detta löstes med hjälp av selectors och selection keys. En selector ger en mekanism för att övervaka en eller flera socket kanaler och kunna känna igen när en eller flera blir tillgängliga för dataöverföring. Då behöver man bara använda en tråd som hanterar flera kommunikationer.

För att hantera logik i servern användes en liten kollektion av trådar som fick uppgifter av kommunikationstråden när det behövdes.

4 Resultat

I första figuren visas en demonstration när applikationen kör. Klienten kopplas automatiskt till servern när klientsidan startas och när spelaren får välkomstmeddelandet betyder det att klienten lyckades koppla till servern eftersom servern skickar välkomstmeddelandet direkt till varje ny klient.

MVC arkitekturen i serverkoden har inte ändrats men klientsidan har ingen controller längre (Figur 2). Eftersom klienten hanterar trådar på ett annat sätt så behövs inte controller, då den användes förut för att ge kommunikationsuppgifter till en kollektion begränsat antal trådar.

Jag kommer ta upp två viktiga punkter som den förbättrade applikationen följer och förklara hur koden följer punkterna:

- **Bara icke-blockerande sockets används:**

För att göra socket channels till icke-blockerande behövde jag kalla på en metod `configureBlocking(boolean)` som tar emot en boolean parameter, om man sätter false kommer den bli icke-blockerande. I figur 3 initieras severns socket channel genom att göra den till icke-blockerande, samma sak görs för varje ny klient som kopplas till servern. I klientsidan används också sockets channel för att skicka och ta emot meddelanden från servern. Klientsidan konfigurerar sitt socket channel till icke-blockerande på samma sätt som servern.

För att en tråd i servern ska kunna hantera flera socket channels samtidigt (dvs flera klient kommunikationer) behöver man använda selectors.

En selector väljer en registrerad kanal som är klar att läsa / skriva / acceptera / ansluta. Bara en av dem kan hända i taget, men det händer snabbt, så man märker inte ens väntetiden. Det finns 4 operationer som kan utföras:

- * Läsa
- * Skriva
- *Acceptera
- *Ansluta

Dessa operationer är `SelectionKeys` (`SelectionKey.class`). De berättar för selector vilken operation är redo att utföra.

Server socket channel i servern använder operationerna läsa, skriva och

acceptera nya klienter. I klienten används operationerna läsa, skriva och ansluta till servern.

Eftersom servern hanterar flera klient channels så går vi genom alla SelectionKeys och utföra deras operationer, annars sover servertråden medan det inte någon operation att utföra. I figur 4 kan man se hur server tråden går igenom Selectionkeys och utför operationerna.

- **Både servern och klienten ska vara multitrådad.**

I klienten används en tråd för att hantera spelarens inmatningar och en tråd för att hantera kommunikationen med servern. Alla meddelanden som kommer från servern skickas vidare till en ny tråd (skapad av ForkJoinPool API:n) för att skriva ut till spelaren, dvs den nya tråden hanterar alla utskrifter till spelaren.

I servern används en tråd för att hantera alla kommunikationer, dvs. acceptera nya klienter, skriva till klienter och ta emot meddelanden från klienter.

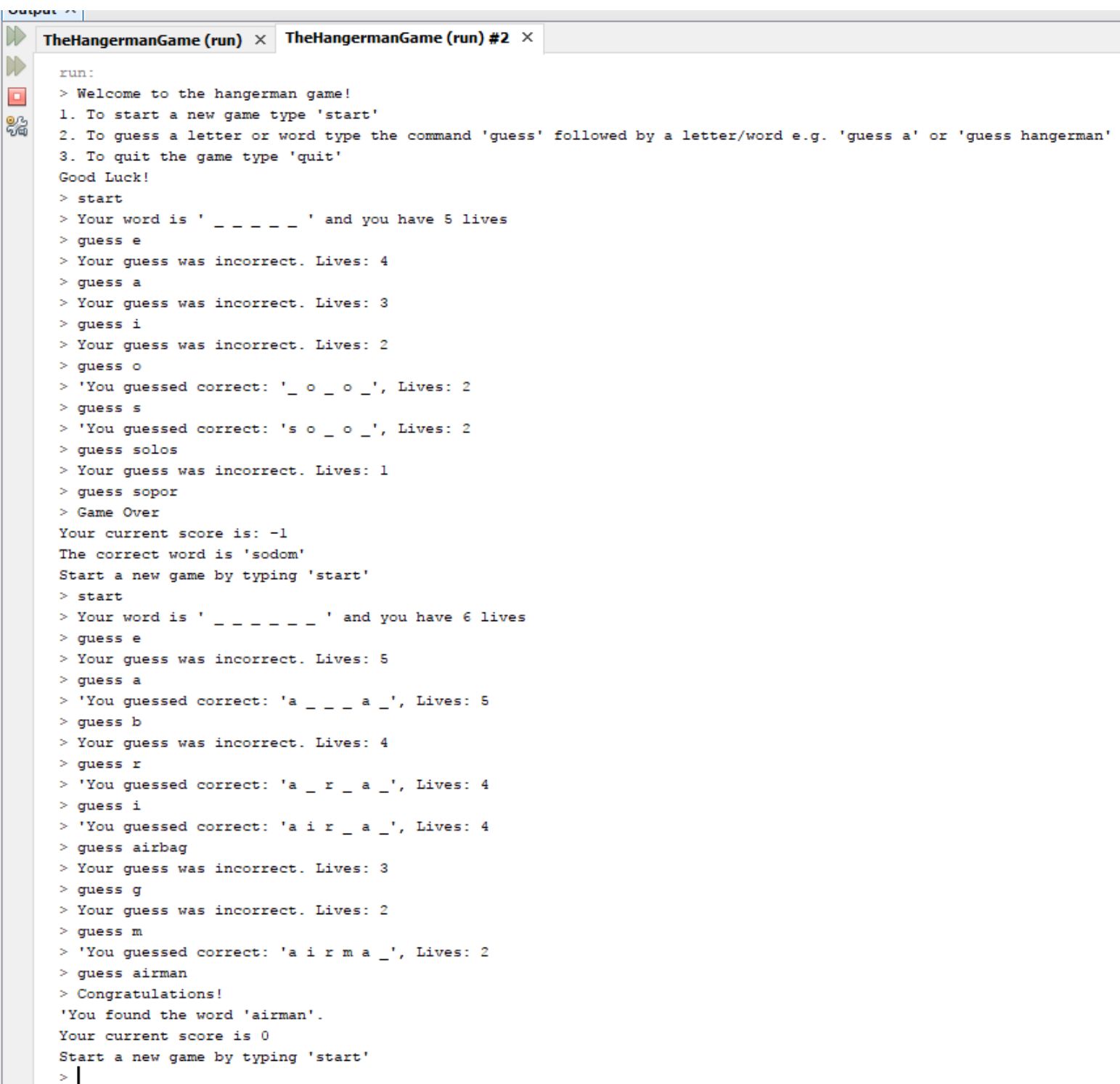
För att hantera logik och uträkningar ger kommunikationstråden uppgifterna till ForkJoinPool API:n. När uträkningarna är klara läggs svaret i en bytebuffer och kommunikationstråden väcks upp för att skicka svaret till klienten.

En annan viktig del i koden som behöver tas upp är NIO ByteBuffers. För att kunna skicka och ta emot meddelandet via socket channels behöver man använda ByteBuffers. En buffer är i huvudsak ett block av minne där du kan skriva data, som du sedan kan läsa igen. Det här minnesblocket är inslaget i ett NIO Buffer-objekt, vilket ger en uppsättning av metoder som gör det lättare att arbeta med minnesblocket.

För att använda en buffer för att läsa och skriva data följde jag dessa 4 steg:

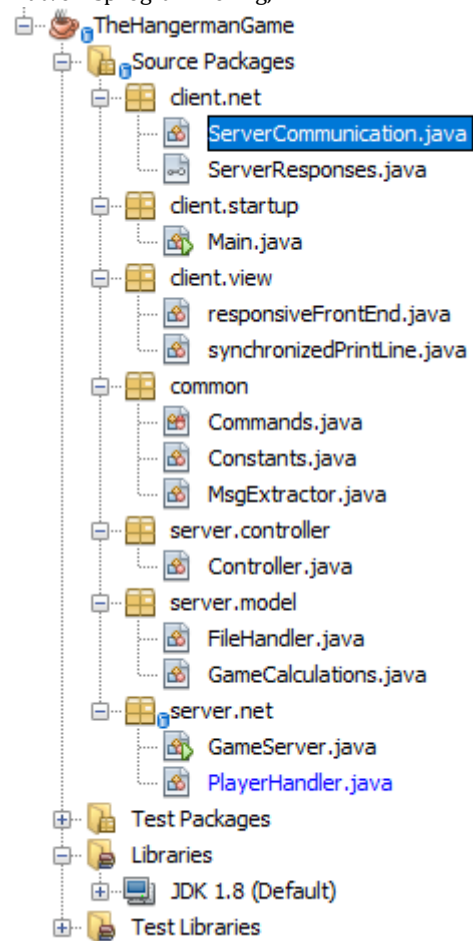
1. Skriv data till buffern med `buffer.put(byte[] source)`.
2. Avända `buffer.flip()` för kunna läsa från buffern.
3. Läs data från buffern med `buffer.get(byte[] destination)`.
4. `buffer.clear()` för att rensa buffern och göra den till skrivbar igen.

Innan man kan använda en buffer måste man först allokera antal bytes i buffern med hjälp av `allocate()` eller `allocateDirect()`.



```
run:
> Welcome to the hangerman game!
1. To start a new game type 'start'
2. To guess a letter or word type the command 'guess' followed by a letter/word e.g. 'guess a' or 'guess hangerman'
3. To quit the game type 'quit'
Good Luck!
> start
> Your word is ' _ _ _ _ _ ' and you have 5 lives
> guess e
> Your guess was incorrect. Lives: 4
> guess a
> Your guess was incorrect. Lives: 3
> guess i
> Your guess was incorrect. Lives: 2
> guess o
> 'You guessed correct: '_ o _ o _', Lives: 2
> guess s
> 'You guessed correct: 's o _ o _', Lives: 2
> guess solos
> Your guess was incorrect. Lives: 1
> guess sopor
> Game Over
Your current score is: -1
The correct word is 'sodom'
Start a new game by typing 'start'
> start
> Your word is ' _ _ _ _ _ ' and you have 6 lives
> guess e
> Your guess was incorrect. Lives: 5
> guess a
> 'You guessed correct: 'a _ _ _ a _', Lives: 5
> guess b
> Your guess was incorrect. Lives: 4
> guess r
> 'You guessed correct: 'a _ r _ a _', Lives: 4
> guess i
> 'You guessed correct: 'a i r _ a _', Lives: 4
> guess airbag
> Your guess was incorrect. Lives: 3
> guess g
> Your guess was incorrect. Lives: 2
> guess m
> 'You guessed correct: 'a i r m a _', Lives: 2
> guess airman
> Congratulations!
'You found the word 'airman'.
Your current score is 0
Start a new game by typing 'start'
> |
```

Figur 1: Användargränssnittet för hangerman spelet.



Figur 2: Applikationens struktur i MVC mönster

```
/**
 * Accepts a new connection by creating a new socket channel with a new
 * playerhandler for the connected client
 * Also configure socket channel to non-blocking
 * @param key Clients selection key
 * @throws IOException if failed to create new socket channel
 */
private void acceptNewPlayer(SelectionKey key) throws IOException {
    ServerSocketChannel serverSocketChannel = (ServerSocketChannel) key.channel();
    SocketChannel playerChannel = serverSocketChannel.accept();
    playerChannel.configureBlocking(false);
    PlayerHandler handler = new PlayerHandler(this, playerChannel, contr);
    playerChannel.register(selector, SelectionKey.OP_WRITE, handler);
    playerChannel.setOption(StandardSocketOptions.SO_LINGER, LINGER_TIME);
}

/**
 * Initilize the listening server socket channel and make it non-blocking.
 * Also initilize the selector and register it to the server channel.
 */
private void initSocketAndSelector() throws IOException {
    listeningServerChannel = ServerSocketChannel.open();
    listeningServerChannel.configureBlocking(false);
    listeningServerChannel.bind(new InetSocketAddress(Constants.NETWORK_PORT));

    selector = Selector.open();
    listeningServerChannel.register(selector, SelectionKey.OP_ACCEPT);
}
```

Figur 3: Två metoder från GameServer klassen. `initSocketAndSelector()` initierar servern socket channel och `acceptNewPlayer()` skapar ny socket channel för varje ny klient.


```
private void startServer() {  
    try {  
        initSocketAndSelector();  
        while (true) {  
            selector.select();  
            Iterator<SelectionKey> keys = selector.selectedKeys().iterator();  
            while (keys.hasNext()) {  
                SelectionKey key = keys.next();  
                keys.remove();  
                if (!key.isValid()) {  
                    continue;  
                }  
                if (key.isAcceptable()) {  
                    acceptNewPlayer(key);  
                } else if (key.isReadable()) {  
                    getPlayerMsg(key);  
                } else if (key.isWritable()) {  
                    sendToPlayer(key);  
                }  
            }  
        }  
    } catch (IOException e) {  
        System.err.println("Connection failed.");  
    }  
}
```

Figur 4: Kod i GameServer klassen för att hantera flera klienter genom att använda selector.

5 Diskussion

Kraven nedan uppfylldes:

- *Programmet följer MVC arkitekturen för att ha bättre struktur på logiken och användargränssnittet och kunna kontrollera alla operationer som används mellan dem.

- *Jag använde icke-blockerande sockets med selectors för att låta en tråd hantera flera klienter samtidigt.

- * För att spelaren ska ha en reponsiv användargränssnitt behövde jag använda flera trådar som tog hand om användarinput och server kommunikationen.

Jag skulle inte säga att jag stötte på stora problem under utvecklingen. Det mesta av tiden gick åt att förstå hur icke-blockerande sockets fungerade och hur man ska använda dem.

Jag märkte inga stora buggar i programmet eftersom det flesta var redan löst i hemuppgift 1.

6 Kommentar om kursen

Jag har spenderat 4–5 timmar åt att gå igenom föreläsningarna innan jag började med hemuppgiften. Tiden det tog att utföra hemuppgiften tog 10–12 timmar. Rapporten tog ungefär 5 timmar att skriva.