Distribuerede Systemer
Aflevering 1

Mikkel Milo, 201505317
Sophie Chau 201404403

April 2017

1 Introduktion

Vi har lavet en distribuerede teksteditor, der fungerer med én server og én klient. Vi har fået givet nogle klasser på forhånd og UI for en Java applikation. Derudover har vi genbrugt noget fra `Server` - og `Client` klassen fra opgave 3 samt lavet to nye klasser, der håndterer forbindelsen.

2 Specifikation

Systemet understøtter følgende handlinger:

- At lytte på forbindelser eller selv at forbinde til en anden peer.
- At lukke en forbindelse (enten server-side eller client-side).
- At genskabe forbindelse med tidligere server.
- At forbinde til ny server efter man har lukket forbindelsen til den gamle.
- At modtage ny forbindelse fra ny klient, hvis den gamle er lukket.
- At skabe forbindelse mellem to forskellige, fysiske enheder på samme netværk.

Systemet understøtter *ikke* følgende handlinger:

- At forbinde mere end én klient til en server.
- At samtidig have to forbindelser hvor man er server i den ene og klient i den anden. Man kan kun være forbundet til én peer ad gangen.

Når en editor opdager, at en forbindelse er lukket, sker der desuden følgende: begge tekstfelter tømmes (denne funktionalitet ændres sandsynligvis i senere versioner), og det nederste farves rødt for at indikere, at der ikke er forbindelse i øjeblikket. Hvis en forbindelse genetableres (enten med samme eller ny modpart) farves tekstfeltet hvidt igen.

3 Systemdesign

Vi vil i denne sektion beskrive den overordnede struktur af systemet. Først ud fra et klassediagram, hvor vi beskriver de enkelte klassers ansvar. Derefter vil vi beskrive event-flowet i systemet. Til sidst forklarer vi relevante kodestumper.
Vi ser nedenfor et forsimplet klassediagram af systemet. Forholdene mellem klasserne `DistributedTextEditor`, `EventReplayer` og `DocumentEventCapturer` er uændret ift. den udleverede kode, selvom vi har lavet ændringer i dem. Vi har tilføjet fire centrale klasser: `Server`, `Client`, `ConnectionHandler` og `ConnectionListener`. `Server` og `Client` starter hhv. en server og en klient, og udsteder et `ConnectionHandler` objekt, som man kan bruge til at sende og modtage data fra den givne server/klient. `ConnectionHandler` er et subjekt i et observer pattern, hvor objektet er en `ClosedConnectionListener` som implementeres af `ConnectionListener`. Denne bruger vi til at udføre eventuelle clean-ups når en forbindelse bliver lukket. `ConnectionHandler` har dermed ansvar for at sende og modtage data for en given forbindelse (socket) - uafhængig om den forbindelse er oprettet af en klient eller en server.

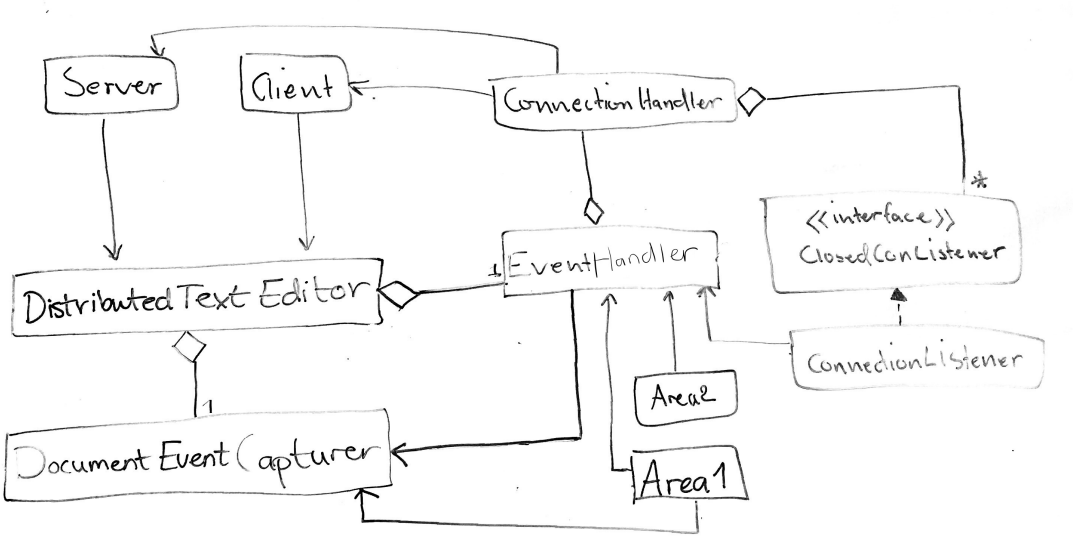

Figure 1: Klassediagram

Det overordnede flow i systemet er som følger: Ved opstart vælger man enten at lytte på indkommende forbindelser, eller at selv forbinde til en given ip-addresse. I det første tilfælde oprettes en server, og i andet tilfælde oprettes en klient. Uafhængigt af udfaldet oprettes der et `ConnectionHandler` objekt når der er blevet etableret en forbindelse med en peer. Vi skelner altså ikke mellem klient og server i `ConnectionHandler`en efter initialisering. Når `ConnectionHandler`-objektet er oprettet (enten i `Client` eller `Server`) gives den videre til `EventHandler`en, som foretager primært to ting: at tage `MyTextEvent`s ud af en kø og sende dem til sin peer, og at lytte på indkommende `MyTextEvent`s fra sin peer. Bemærk at håndteringen af forbindelsen er ens om man er server eller klient.

3.1 Kodestumper

Vi starter med at gennemgå vores implementationer af `Connect`, `Listen` og `Disconnect` i `DistributedTextEditor`, hvor vi også vil se nærmere på `Server` og `Client`. Derefter ser vi på `ConnectionHandler`, og til sidst `EventHandler`.

De næste tre kodestumper er fra `DistributedTextEditor` klassen. Hvad de har tilfælles er at de alle kalder `saveOld()`, som tjekker om der er blevet lavet om i `JTextArea`s og ellers 'cleaner' vi den øverste `JTextArea`.

`Listen` starter en ny tråd, der opretter et nyt `Server` objekt. Vi giver `Server` objektet et `EventHandler` objekt (*er*) og en integer (port) med som argument. Dvs. `Listen` starter en ny server, som en klient kan forbinde til.

```
1   Action Listen = new AbstractAction("Listen") {
      private static final long serialVersionUID = 1L;
3
      public void actionPerformed(ActionEvent e) {
5       saveOld();
        area1.setText("");
7       try {
          server = new Server(er, Integer.parseInt(portNumber.getText()));
9       } catch (IOException ex) {
          ex.printStackTrace();
11      }
        new Thread(server).start();
13      setTitle("I'm listening on " + ipaddress.getText() + " : " + portNumber.getText());
        changed = false;
15      Save.setEnabled(false);
        SaveAs.setEnabled(false);
17    }
  };
```

src/DistributedTextEditor.java

`Connect` ligner `Listen` ret meget, men opretter i stedet et `Client` objekt og giver så dette objekt en ip-adresse, `EventHandler` objekt og en port med som argument.

```
  Action Connect = new AbstractAction("Connect") {
2   private static final long serialVersionUID = 1L;

4   public void actionPerformed(ActionEvent e) {
      saveOld();
6     area1.setText("");
      client = new Client(ipaddress.getText(), er, Integer.parseInt(portNumber.getText()),
    DistributedTextEditor.this);
8     new Thread(client).start();
      setTitle("Trying to connect to " + ipaddress.getText() + ":" + portNumber.getText());
10
      changed = false;
12    Save.setEnabled(false);
      SaveAs.setEnabled(false);
14
    }
16 };
```

src/DistributedTextEditor.java

Hvis serveren har forbindelse til en klient, vil `Disconnect` afbryde forbindelsen og lukke serveren ned.

```
Action Disconnect = new AbstractAction("Disconnect") {
  private static final long serialVersionUID = 1L;

  public void actionPerformed(ActionEvent e) {
    setTitle("Disconnected");
    area1.setText("");
    if (client != null) {
      client.disconnect();
    }
    if (server != null) {
      server.shutdown();
    }
  }
};
```

src/DistributedTextEditor.java

Nedenfor ses `run()`-metoden for `Server` klassen. Her venter den på at få forbindelse med en klient. Når der oprettes forbindelse (dvs. linje 6 er blevet udført) laver den in- og output streams til objekter. Ud fra dette laver den en `ConnectionHandler`, som bliver sat i `EventHandleren` i linje 12.

```
@SuppressWarnings("static-access")
public void run() {
  while (Thread.currentThread().isInterrupted() == false) {
    try {
      System.out.println("Waiting for client on "
          + serverSocket.getInetAddress().getLocalHost().getHostAddress() + " : " + port);
      Socket socket = waitForConnectionFromClient();

      System.out.println("Connection from " + socket.getRemoteSocketAddress());
      ObjectOutputStream objOutStream = new ObjectOutputStream(socket.getOutputStream());
      ObjectInputStream objInputStream = new ObjectInputStream(socket.getInputStream());
      handler = new ConnectionHandler(socket, objInputStream, objOutStream);
      eventHandler.setConnectionHandler(handler);
    } catch (SocketTimeoutException s) {
      System.out.println("Socket timed out!");
      break;
    } catch (IOException e) {
      System.err.println(e);
      break;
    }
  }
}
```

src/Server.java

Nedenfor ses klientens `run()`-metode, som stortset er identisk til serverens `run()`-metode bortset fra, at i stedet for at vente på indkommende forbindelser, så prøver vi selv at oprette en forbindelse til en ip-addresse. Ligeledes opretter den også et `ConnectionHandler` objekt, og sætter det i `EventHandler`. Den notificerer `DistributedTextEditor` klassen, når der er forbindelse, så titlen i vinduet viser *Connected* i stedet for *Connecting*.

```
1   public void run() {
      System.out.println("Starting client. Type CTRL-D to shut down.");
3     socket = connectToServer(serverName);
      if (socket != null) {
5       frame.clientConnected();
        System.out.println("Connected to " + socket);
7       try {
          // For sending objects to the server
9         ObjectOutputStream objOutStream = new ObjectOutputStream(socket.getOutputStream());
          ObjectInputStream objInputStream = new ObjectInputStream(socket.getInputStream());
11        objOutStream.flush();
          handler = new ConnectionHandler(socket, objInputStream, objOutStream);
13        eventHandler.setConnectionHandler(handler);
        } catch (IOException e) {
15        System.err.println(e);
          disconnect();
17      }
      }
19  }
```

src/Client.java

Vi ser nedenfor en klassespecifikation af `ConnectionHandler`-klassens feltvariabler og metoder. Vi vælger ikke at beskrive implementationen af klassens metoder, da de er forholdsvis trivielle og selvforklarende ud fra metodenavnene. Klassen har som nævnt en socket og tilhørende in- og output streams. Da den er subjekt i en observer pattern har den også en liste af observers, som den kan notificere med metoden `notifyListeners()`. Klassen har selvfølgelig metoder for at sende og modtage data, og ikke mindst en metode til sikkert at lukke en forbindelse. Hver gang en server eller en klient ønsker at lukke deres forbindelse, kalder de denne metode på deres tilhørende `ConnectionHandler` objekt. Dette indebærer at kalde `close()` på socket, in, og out samt at sætte feltvariablen `isClosed()` til true.

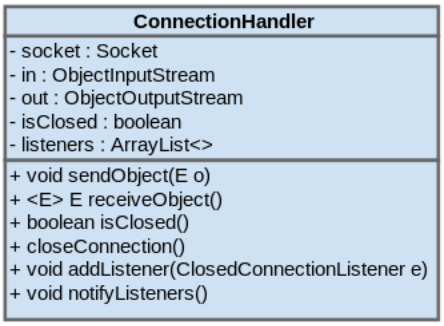

Figure 2: klassespecifikation af `ConnectionHandler`

`listenOnPeerEvent()` tjekker om vi har forbindelse. Hvis det er tilfældet, lytter vi på `MyTextEvents` og udfører dem på det nederste `JTextArea`. Hvis der fanges en exception, antager programmet at den har mistet forbindelsen, og lukker den derfor. Dette gør vi fordi man ikke på nogen måde kan tjekke, om en forbindelse er i live.

```
  public void listenOnPeerEvent() {
    new Thread(new Runnable() {
      public void run() {
        while (true) {
          if (connectionHandler != null && !connectionHandler.isClosed()) {
            MyTextEvent mte = null;
            try {
              // blocks until received object
              mte = (MyTextEvent) connectionHandler.receiveObject();
            } catch (IOException ex) {
              sleep(10);
              System.out.println("closing connection with server.");
              connectionHandler.closeConnection();

            }

            if (mte instanceof TextInsertEvent) {
              final TextInsertEvent tie = (TextInsertEvent) mte;
              EventQueue.invokeLater(new Runnable() {
                public void run() {
                  try {
                    area2.insert(tie.getText(), tie.getOffset());
                  } catch (Exception e) {
                    System.err.println(e);
                  }
                }
              });
            } else if (mte instanceof TextRemoveEvent) {
              final TextRemoveEvent tre = (TextRemoveEvent) mte;
              EventQueue.invokeLater(new Runnable() {
                public void run() {
                  try {
                    area2.replaceRange(null, tre.getOffset(), tre.getOffset() + tre.
  getLength());
                  } catch (Exception e) {
                    System.err.println(e);
                  }
                }
              });
            }
          }
        }
      }
    }).start();
  }
```

src/EventHandler.java

`run()`-metoden kører `listenOnPeerEvent()` i en seperat tråd, og tager events ud af køen fra `DocumentEventCapturer`-klassen. Her blokerer metoden indtil der er et event at tage. Når vi har forbindelse sender vi events afsted. Hvis vi får en exception, antager vi at vi ikke længere har forbindelse og lukker den ellers ned.

```
1  public void run() {
2
3    // runs in own thread
4    listenOnPeerEvent();
5
6    boolean wasInterrupted = false;
7    while (!wasInterrupted) {
8      try {
9        MyTextEvent mte = dec.take();
10       if (connectionHandler != null && !connectionHandler.isClosed()) {
11         connectionHandler.sendObject(mte);
12       }
13     } catch (IOException ex) {
14       connectionHandler.closeConnection();
15
16     } catch (Exception e) {
17       e.printStackTrace();
18       wasInterrupted = true;
19       connectionHandler.closeConnection();
20
21     }
22   }
23   System.out.println("I'm the thread running the EventHandler, now I die!");
24 }
```

src/EventHandler.java

4 Guide til at køre programmet

Efter at have åbnet to terminaler og kompileret alle Java-filerne, køres to `DistributedTextEditor` (DTE) simultant. Herefter skal disse instruktioner følges:

1. I DTE1 køres serveren: File → Listen.
2. I DTE2 køres klienten: File → Connect.
   - Der er sørget for at den rigtige ip-adresse og port nr. står i de nederste små tekstfelter.
   - En besked *"Would you like to save Unitled?"* dukker op. Der vælges *"No"*.

   Der er nu forbindelse mellem klient og server. I øverste tekstfelt i DTE1 kan serveren skrive og i det nederste felt vises hvad klienten har skrevet. Det omvendte gælder for klientens DTE2.

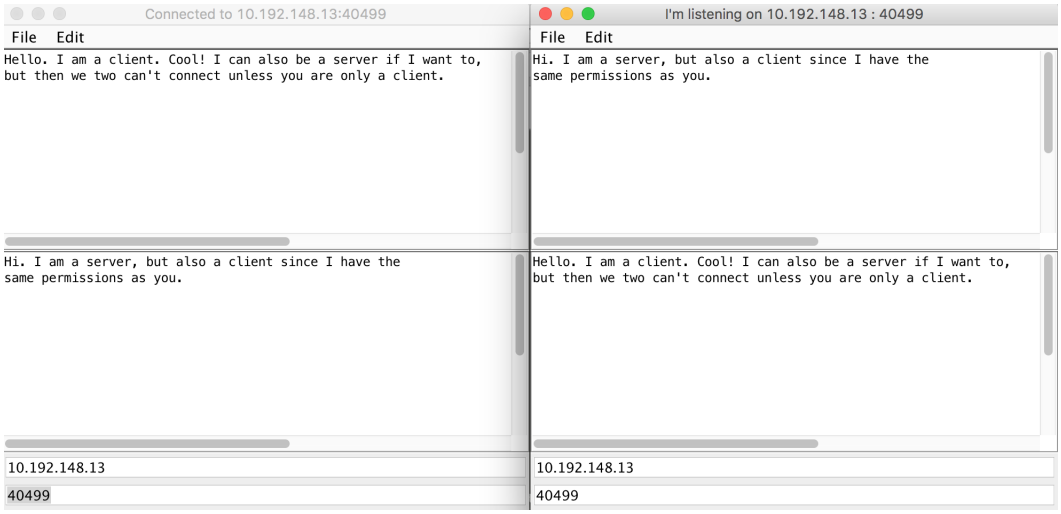

Figure 3: DTE'er kommunikerer som klient og servent.

3. I DTE2 afbryder vi forbindelsen: File → Disconnect.

   Nu bliver de nederste JTextAreas røde, for at indikere at vi har mistet forbindelse.

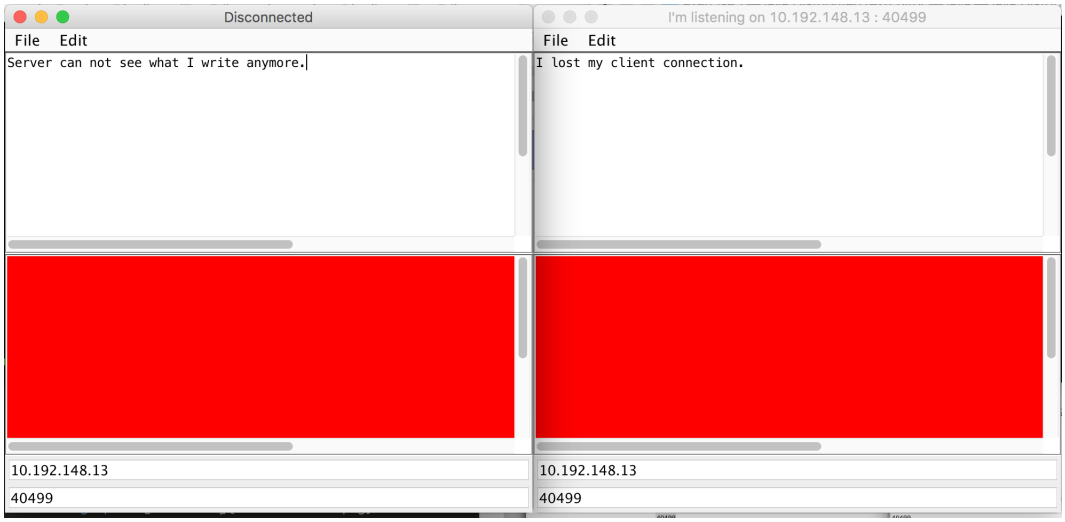

Figure 4: Forbindelsen er lukket.

4. Forbind igen (step 2).
5. Afbryd forbindelsen for begge DTE'er.
6. Lad nu DTE2 være server: File → Listen og DTE1 være klient: File → Connect.

Vi har nu vist at vores DTE'erne både kan agere server og klient, dvs. programmet understøtter én klient og én servent.

5 Konklusion/refleksion

Vores program kan håndtere én klient og én servent på både en og to fysiske maskiner (FM) samt en virtuel maskine (VM). Programmet kan hhv. være klient/server på to forskellige FM'er og/eller VM'er. Nedenstående pile (bortset fra den sidste) indikerer at både venstre- og højresiden kan agere klient/server.

- FM1 $\longleftrightarrow$ FM2
- FM2 $\longleftrightarrow$ VM1
- FM1 $\longleftrightarrow$ VM2
- VM1 $\longleftrightarrow$ VM2
- FM1 (klient) $\longleftrightarrow$ VM1 (server)

Dvs. den eneste forbindelse vi ikke har fået til at virke, er hvis vi på samme maskine starter en server på den fysiske maskine og en klient på den virtuelle maskine. Vi tænker at det har noget at gøre med port forwarding på de virtuelle maskiner, da vi sagtens kan forbinde til en fysisk server på en anden maskine med en virtuel klient. Under alle omstændigheder tror vi ikke dette er et problem med koden, men derimod opsætningen af VM'en/maskinernes netværk. Vores fysiske maskiner kan kun forbindes til hinanden, hvis vi er på et andet WiFi end eduroam. Igen ser vi ikke dette som et problem med vores kode, men som et netværksproblem som vi ikke har magt over.

Desuden understøtter programmet ikke flere klienter og servere på samme tid, men vi har overvejet hvordan vores program relativt let vil kunne understøtte det ved at have en liste af `ConnectionHandler` elementer i `EventHandler`en i stedet for blot ét. Hver gang man enten modtager eller sender events skal man da iterere gennem listen af `ConnectionHandler`s og sende/modtage til/fra hver af dem.