F4
Mera om TCP
Serverarkitektur

EDA095 Nätverksprogrammering

Roger Henriksson
Datavetenskap
Lunds Universitet

Java New I/O – java.nio

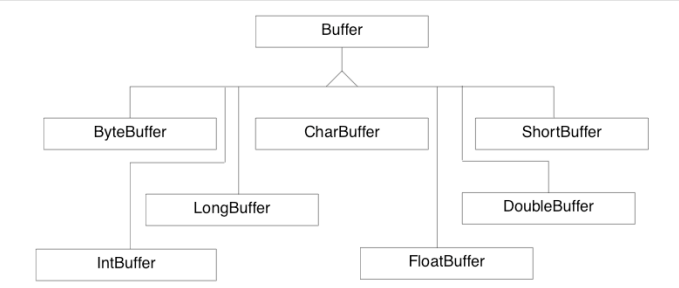
Ny modell för sekvensiell I/O från Java 1.4.
Stöder icke-blockerande I/O.

Översikt

- *SocketChannel* – ersätter Socket samt InputStream och OutputStream.
- *ServerSocketChannel* – ersätter ServerSocket.
- *Buffer* – read/write arbetar mot buffertar.
- *Selector/SelectorKey* – stöd för icke-blockerande I/O.

Buffer

All I/O går via buffertar som kan lagra olika primitiva typer.



En buffert har en längd och en intern markör som talar om var i bufferten läsning/skrivning ska ske härnäst.

ByteBuffer

Skapa en ByteBuffer

```
public static ByteBuffer allocate(int capacity);
```

Exempel: `ByteBuffer buffer = ByteBuffer.allocate(20);`

Läsa/skriva

```
public byte get();
public byte get(int index);
public ByteBuffer put(byte b);
public ByteBuffer put(int index, byte b);
```

Motsvarande i andra buffertklasser

I ByteBuffer även metoder för att koda om int/long/float etc till Bytes.
Ex: `getChar(),putChar(char c),getLong(),putLong(long l);`

Manipulera bufferten

Tömma bufferten

```
public Buffer clear();
```

Markören

```
public int position();
public Buffer position(int newPosition);
public Buffer rewind();
public Buffer flip();
```

Storlek

```
public int capacity();
public int limit();
public Buffer limit(newLimit);
public int remaining();
public boolean hasRemaining();
```


SocketChannel

Skapa en uppkoppling

```
public static SocketChannel open(SocketAddress remote)
                                 throws IOException;
```

…

Exempel:

```
SocketAddress address = new InetSocketAddress("www.cs.lth.se",80);
SocketChannel channel = SocketChannel.open(address);
```

Läsa

```
public int read(Buffer dst) throws IOException;
```

Garantera att en buffert fylls helt

```
While(buf.hasRemaining() && chnl.read(buf)!=-1);
```


SocketChannel, fortsättning

Skriva

```
public int write(Buffer src) throws IOException;
```

Garantera att hela bufferten skrivs ut

```
while (buf.hasRemaining() && chnl.write(buf)!=-1);
```

Stänga en channel

```
public int close() throws IOException;
public boolean isOpen();
```

DEMO – NIOReader.java

ServerSocketChannel

Avsedd för att ta emot uppkopplingar

Metoder

```
public static ServerSocketChannel open()
                                  throws IOException;
public ServerSocket socket();
public SocketChannel accept() throws IOException;
```

Måste knytas till ett portnummer via ett ServerSocket-objekt!

```
ServerSocketChannel server = ServerSocketChannel.open();
ServerSocket socket = server.socket();
SocketAddress address = new InetSocketAddress(80);
Socket.bind(address);
```

DEMO – NIOServer1.java

Icke-blockerande I/O

En channel kan konfigureras så att anrop av accept() och read() aldrig blockerar.

```
public SelectableChannel configureBlocking(boolean block)
                                    throws IOException;
```

Kan vi använda detta för att bygga en enkeltrådad server som betjänar flera klienter samtidigt?

Idé
Skriv en loop som skriver till de anslutna klienterna och en gång per varv gör ett icke-blockerande anrop av accept().

DEMO – NIOServer2.java

Busy-wait!
Fyllda nätverksbuffertar ger problem.

Selector

Mekanism för att kunna blockera i väntan på att något ska hända på en av flera channels:

- accept() möjligt
- read() möjligt
- write() möjligt

Skapa en Selector

```
public static Selector open() throws IOException;
```

Channels registreras i en Selector tillsammans med vilken typ av händelser man vill vänta på. I Channel-klasserna:

```
public SelectionKey register(Selector sel, int ops)
                            throws ClosedChannelException;
public SelectionKey register(Selector sel, int ops,
             Object att) throws ClosedChannelException;
```


Vänta på en händelse

Blockera i väntan på en händelse

```
public int select() throws IOException;
public int select(long timeout) throws IOException;
public int selectNow() throws IOException;
```

Vad var det som hände?

```
public Set selectedKeys();
```

Resultatet av selectedKeys() är ett Set innehållande ett antal object av typen SelectedKey.

SelectionKey

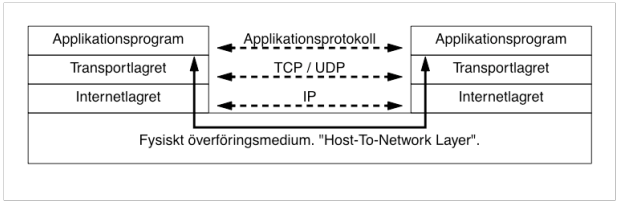
SelectionKey representerar en channel/händelse registrerad i en Selector.

```
public boolean isAcceptable();
public boolean isReadable();
public boolean isWriteable();
public SelectableChannel channel();
public Object attachment();
```

DEMO – NIOServer3.java

Design av applikationsprotokoll

Hittills stort fokus på gränssnittet mellan applikationsprogram och transportlagret.



Applikationsprotokoll och TCP
- Avgränsning av fält/meddelanden
- Kodning av datatyper som en sekvens av bytes
- Teckenkodning – radslut
- "Big-endian/little-endian"

Protokolldesign – exempel

I kursen Realtidsprogrammering behöver man överföra en ström av JPEG-bilder tillsammans med tidsstämplar. För varje bild:
- Tidsstämpel (heltal, long)
- JPEG-bild (ett variabelt antal sekvensiella bytes)

Problem
- Hur representerar man en tidsstämpel? Binärt? ASCII? Ordning?
- Hur vet man när alla bytes i själva bilden sänts?
- När börjar nästa meddelande (bild)?

Exempel, fortsättning

Protokollförslag:

1. Tidsstämpel – en long i binär form, 8 bytes, big-endian
2. Antal bytes i den efterföljande JPEG-bilden – long, 8 bytes, big-endian
3. JPEG-bilden – den sekvens av bytes som utgör bilden

HTTP

Protokoll för kommunikation mellan webbläsare och webbserver.

- RFC1945 (HTTP 1.0) och RFC2616 (HTTP 1.1).
- Textbaserat
- Radbaserat, radslut: CR + LF
- HTTP 1.0:
  - Anslutning via TCP
  - Request
  - Response
  - Nedkoppling

HTTP, fortsättning

Request (avslutas med dubbla radslut, CR+LF+CR+LF):

```
GET /index.html HTTP/1.1
Accept: text/html, text/plain, image/gif, image/jpeg
User-Agent: Mozilla/4.0
Host: ygg.cs.lth.se
```

Response:

```
HTTP/1.1 200 OK
Date: Wed, 29 Mar 2006 08:20:32 GMT
Server: Apache/2.0.40 (Red Hat Linux)
Last-Modified: Mon, 21 Mar 2006 18:17:07 GMT
Content-type: text/html; charset=ISO-8859-1
Content-length: 107
<html>
<head>
...
```


Design: enkel HTTP-server

```
while (true) {
   socket = serversocket.accept();
   parseAndResponse(socket);
   socket.close();
}
```

Nackdelar
- Kan endast hantera en uppkoppling åt gången.
- Utnyttjar tillgänglig bandbredd dåligt.

Design: enkel multitrådad server

Main

```
while (true) {
   socket = serversocket.accept();
   new RequestHandler(socket).start();
}
```

RequestHandler

```
parseAndResponse(socket);
socket.close();
```

Fördelar
- enkel design
- klarar flera uppkopplingar parallellt

Nackdel
- Skalar inte upp. Starta nya trådar dyrt.

Design: enkeltrådad server med NIO

Enkeltrådad server enligt principerna beskrivna i början av föreläsningen. Icke-blockerande I/O.

Fördelar
- Bättre bandbreddsutnyttjande – vi lägger hela tiden CPU-kraft på de anslutningar där det finns något att göra.
- Ingen overhead för trådhantering

Nackdel
- Snabbt väldigt komplex kod.

Design: server med trådpool

En pool med n stycken RequestHandlertrådar.

Main

```
while (true) {
   socket = serversocket.accept();
   thread = pool.getFree();
   thread.assign(socket);
}
```

RequestHandler

```
while (true) {
   socket = acceptJob();
   parseAndResponse(socket);
   socket.close();
}
```

Fördel: Undviker att starta nya trådar hela tiden.
Nackdel: Begränsat antal samtidiga anslutningar.

Design: trådpool och jobblista

En kö med ankomna anslutningar och en pool av servertrådar.

Main

```
while (true) {
   socket = serversocket.accept();
   jobs.insert(socket);
}
```

RequestHandler

```
while (true) {
   socket = jobs.getJob();
   parseAndResponse(socket);
   socket.close();
}
```

Variant av föregående lösning.
Exempel: Se JHTTP, kursboken kapitel 10.