Java: Netværkskommunikation med NIO

Hvad er NIO, og hvorfor er det interessant i forbindelse med netværkskommunikation? Hvorfor er det ikke særlig udbredt, hvis det er interessant? Artiklen forsøger på en så let forståelig måde som muligt bl.a. at besvare disse spørgsmål.

Skrevet den **12. Feb 2009** af **dsj** I kategorien **Programmering / Java** | ★★★☆☆

Hvad er NIO?

NIO står for 'New I/O', og er en teknologi tilføjet af Sun i J2SE fra og med version 1.4 som en ny, alternativ måde at håndtere I/O på. Ikke alternativ fordi det er en anderledes måde i forhold til andre programmeringssprog end Java, og ikke ny fordi man i andre sprog ikke længe har gjort på samme måde. Men alternativ fordi man i Java nu har et andet instrument end den gamle I/O-pakke, og ny fordi det er første gang en så OS-nær teknologi er blevet en del af Java-standarden. NIO dækker over pakken java.nio.*, og er som sagt en for Java-programmører ny måde at foretage disk- og netværksoperationer på, i stedet for den gamle I/O-pakke, java.io.*.

Den gamle I/O-pakke eksisterer som de fleste sikkert er klar over stadig, men den indeholder store ulemper, som i mange år har stillet Java tilbage i forhold til andre sprog som C og C++, når det drejer som om I/O-operationer som f.eks. at læse filer, skrive til og læse fra netværksforbindelser, også kendt som Socket's. Denne artikel vil fokusere på brugen af NIO i forbindelse med netværkskommunikation, selvom NIO som nævnt indeholder flere andre interessante funktionaliteter, såsom mapping af filer til hukommelsen og låsning af filer.

Det handler om tråde

Når vi tænker på netværkskommunikation som Socket's, er de fleste Java-programmører med lidt erfaring indenfor feltet klar over, at for hver Socket-instans kræves én tråd, eller to hvis man ønsker muligheden for at udnytte dagens netværkskort, der understøtter Full Duplex; muligheden for at læse fra og skrive til en netværksforbindelse på samme tid. Dette lyder ikke umiddelbart som et problem. Bevares, et par tråde fra eller til betyder vel ikke det store, når et normalt operativsystem som Windows eller Linux alligevel har en hundrede stykker kørende ved opstart. Det gør det heller ikke nødvendigvis medmindre vi snakker servere.

Når man udvikler en klient-applikation, f.eks. en Internet-browser, til at snakke med en http-server, skal der for at få en effektiv applikation ud af det flere tråde til, og så bekymrer de færreste sig om en eller to tråde til at håndtere kommunikationen med http-serveren. Vender vi sagen 180 grader og ser tingene fra serverens vinkel, er sagen en helt anden. For servere er målet et kunne håndtere, eller servicere, så mange klienter på én gang, som overhovedet muligt. Jo færre klienter en server kan håndtere, jo mere hardware skal der postes penge i, for at opretholde den samme kapacitet.

Lad os som eksempel tage en http-server, udviklet i Java, der anvender java.io.* og Socket's til at kommunikere med sine klienter. En klient opretter en forbindelse til serveren, der modtager denne som en Socket-instans gennem ServerSocket.accept(). Herefter oprettes en BufferedReader og en BufferedWriter på baggrund af Socket-instansens InputStream og OutputStream. For at modtage data fra klienten, må vi sætte en tråd til at kalde readLine() på den nyligt oprettede BufferedReader, der returnerer et String-objekt, når en række data afsluttet med '\n' er modtaget fra klienten. Modtager vi ingen data fra klienten, ja så står tråden blot og venter i BufferedReader.readLine(), til der kommer nogle data, til vi afbryder den,

eller forbindelsen af en eller anden grund afbrydes. Når der skal skrives data tilbage til klienten, skal tråden enten bryde ud af BufferedReader.readLine() for at sende data, eller også skal endnu en tråd oprettes til at skrive data.

Lad os bare sige, at vores http-server nøjes med én tråd, så er regnestykket ikke så svært. Hvis vi ønsker med vores server at håndtere 2000 klienter på samme tid, giver det 2000 tråde, men betyder det noget? Ja, en CPU kan pr. definition kun have én tråd kørende på samme tid, og har vores server 2000 tråde, vil CPU'en blive kraftigt belastet af, meget ofte at skulle skifte rundt mellem de mange tråde; der er nemlig ikke noget der tager tid, som at skifte tråde på CPU'en. Når det er sagt, skal der tages højde for hvilket operativsystem serveren kører på. Windows er langt bedre end Linux til at håndtere en stor mængde tråde, grundet måden hvorpå Thread-instanser i Java mappes til tråde eller processer på, men det vil artiklen ikke gå i dybden med.

To nye Java-fisk: Non-blocking og multiplexing

Problemet ved i servere at anvende Java's gamle I/O-pakke er altså det store antal tråde der kræves. Her er det NIO kommer ind i billedet. Server-udviklere fra andre sprog end Java har længe været bekendte med en teknologi, hvor en eller meget få tråde kan servicere mange hundrede eller tusinde netværksforbindelser på samme tid. Kodeordene er 'non-blocking' Socket's, altså netværksforbindelser der ikke blokere når alle tilgængelige data er læst eller skrevet, og 'multiplexing', altså måden hvorpå vi finder ud af at der skal læses fra eller skrives til en netværksforbindelse, uden at have en tråd blokeret til data modtages i f.eks. BufferedReader.readLine(). Det er to store fisk, som ikke er helt lette at sluge, men ok, her kommer en kort forklaring.

Hvordan virker NIO

NIO består (når vi altså snakker netværksforbindelser) primært af følgende to elementer:

SocketChannel: I stedet for Socket's til at repræsentere en netværksforbindelse, har vi i NIO i stedet en SocketChannel, hvorpå read- og write-operationer kan udføres. SocketChannel's har mulighed for at fungere i en ikke-blokerende tilstand som betyder, at read- og write-operationer, hvis ikke der er data at læse eller skrive, ikke blokere, men blot returnerer øjeblikkeligt. Når der skrives til og læses fra SocketChannel's gøres det ved at bruge ByteBuffer'e, der har den bemærkelsesværdige egenskab, at de kan allokeres direkte i OS' hukommelse med bedre performance til følge.

Selector: Er den komponent der overvåger en række netværksforbindelser og finder ud af, om der skal læses fra eller skrives til nogen af dem. Metoden Selector.select() kaldes og returnerer først, når der skal skrives til eller læses fra en registreret SocketChannel, eller for den sags skyld modtages nye netværksforbindelser gennem en ServerSocketChannel. Selector'en "multiplexer" altså hændelserne READ, WRITE og ACCEPT til de registrerede SelectableChannel-instanser: SocketChannel's eller ServerSocketChannel's.

Samlet giver disse to komponenter mulighed for kun at have én tråd til at servicere mange netværksforbindelser. En enkelt tråd sættes altså til at håndtere en Selector-instans ved at kalde Selector.select() og servicere netværksforbindelserne, når metoden returnerer hvilke det drejer sig om.

Der kan forklares meget mere om hvordan NIO-kode fungerer og nogle eksempler vil ikke være af vejen. Der findes på Internettet flere eksempler herpå, bl.a.:

http://www.owlmountain.com/tutorials/NonBlockingIo.htm
http://www.onjava.com/pub/a/onjava/2002/09/04/nio.html

Hvorfor så interessant da?

Det interessante ved NIO i forbindelse med netværksforbindelser er altså, at ved et kraftigt reduceret

behov for tråde, kan Java-servere yde langt mere end hvad førhen har været muligt; mange flere netværksforbindelser kan håndteres og servicere på samme tid uden et eksplosivt overhead til flere tusinde tråde.

Hvis nu NIO er løsningen til folket, hvorfor er NIO så ikke mere udbredt end det er? Det skyldes flere ting. For det første laver man ikke uden videre en eksisterende server om; NIO har en helt anden struktur end den gamle I/O-pakke, og at gå over til NIO kan betyde meget gennemgribende ændringer i en eksisterende server. For det andet er NIO stadig et forholdsvis nyt begreb, selvom det har været fremme et par år. Alt nyt software indeholder fejl der løbende må rettes. Derfor har frygten for ustabilitet og fejl holdt mange fra at anvende J2SE 1.4 og dermed NIO - fejl har der også været en del af ved den første udgivelse af Sun's J2SE 1.4, men med den seneste version 1.4.2, er mange af disse fejl blevet rettet. Desuden har IBM et også gratis 1.4-miljø på markedet, dog kun til Linux, der siges at være både mere stabilt og effektivt end det fra Sun. Tests jeg personligt har udført i forbindelse med netop omhandlende emne, bekræfter da også denne påstand.

En væsentlig hage ved NIO er også, at det både er sværere at anvende i praksis end den gamle I/O-pakke, og samtidig fra Sun's side, efter min mening, er elendigt dokumenteret. Ja vel er klasser og metoder detaljeret beskrevet i Javadoc'en, men alt for mange ting er man nødt til selv at erfare og finde ud af ved at læse om andres erfaringer, f.eks. i Sun's udviklerforum http://forum.java.sun.com.

Jeg håber at denne i forhold til emnet noget kortfattede artikel, har givet læseren et indblik i, hvad NIO betyder for netværkskommunikation, specielt når det drejer sig om server-applikationer. NIO indeholder som nævnt meget mere end hvad artiklen inddrager, men målet har været at dække nok til at skabe forståelse for de fordele NIO kan give Java-udviklere. Hele artikler kunne skrives om NIO's indre struktur, hvilke fælder NIO indeholder, samt hvordan server-applikationer bygget på NIO bør designes, men det må blive en anden dag!

Kommentar af simonvalter d. 13. Jan 2004 | 1

hehe fik læst inden du satte point på, men den har helt sikkert gjort sig fortjent til dem. Jeg havde slet ikke hørt om nio, men det må jeg kigge nærmere på nu, og du giver en god introduktion... måske kan vi bytte min lærer ud med dig? ;)

Kommentar af minau d. 11. Feb 2004 | 2

Meget velbeskrevet

Kommentar af arne_v d. 13. Jan 2004 | 3

Velfunderet indsigt i NIO's verden.

Kommentar af scarlac d. 14. Jan 2004 | 4

Velformuleret og går lige til sagen, forklarer ulemper og fordele på en forståelig måde.

Kommentar af ttn- d. 30. Jul 2004 | 5

Mønster eksempel på en god artikel. Jeg skal dog lige lære det gamle I/O, før jeg springer ind i det nye :)

Kommentar af aadal_dk d. 19. Apr 2004 | 6

Kommentar af arnejan d. 13. Jan 2004 | 7

Microsoft.NET's pendant til NIO hedder: Asynchronous Sockets.