Java Networking and IO

Introduction

Sockets use *TCP/IP* transport protocol and are the last piece of network communication between two hosts.

TCP: It is a reliable data transfer protocol that ensures that the data sent is complete and correct and requires to establish a connection.

You do not usually have to deal with them since there are protocols built on top of them like *HTTP* or *FTP*.

Java offers a blocking and non-blocking alternative to create sockets, and depending on your requirements, you might consider the one or the other.

Java Blocking IO

The Java blocking IO API is included in **JDK** under the package java.net.
This API is based on flows of byte streams and character streams that can be read or written. There is not an index that you can use to move forth and back, like in an array, it is simply a continuous flow of data. It is possible to create pipelines of filters attached to input and output streams (e.g. PrintStream, BufferedOutputStream, etc).

Example

```
FileOutputStream fileStream;
BufferedOutputStream bufferedStream;
fileStream = new FileOutputStream("prova.txt");
bufferedStream = new
BufferedOutputStream(fileStream);
BufferedOutputStream bufferedStream= new
BufferedOutputStream(new FileOutputStream("prova.txt"));

DataInputStream asciiIn;
PrintStream asciiOut;
asciiIn = new DataInputStream(new FileInputStream("in.txt"));
asciiOut = new PrintStream(new FileOutputStream("out.txt"));
PrintStream asciiOut;
asciiOut = new PrintStream(new BufferedOutputStream( new FileOutputStream("out.txt")));
```

Socket in Java

```
//port scanning
for (int i = 1; i< 1024; i++){
   try{
    new Socket(host, i);
    System.out.println("Esiste un servizio sulla porta" + i);
   } catch (UnknownHostException ex){
       System.out.println("Host Sconosciuto");
   } catch (IOException ex) {
      System.out.println("Non esiste un servizio sulla porta"+i);
    }
}}}
```

Filters

```
import java.net.*;
import java.io.*;
import java.util.*;
...
InetAddress ia = InetAddress.getByName("localhost");
int port = 12345;
Socket s = null;
try{
  s = new Socket (ia, port);
  } catch (Exception e){...}

InputStream is = s.getInputStream( );
OutputStream os = s.getOutputStream( );
DataInputStream netIn = new DataInputStream(is);
DataOutputStream netOut = new DataOutputStream(os);
netOut.write(buffer);
....
```

Server Socket

```
import java.net.*;
import java.io.*;
int port= 12345;

ServerSocket s = new ServerSocket(port,2);
Socket sdati = s.accept();
InputStream is = sdati.getInputStream();
OutputStream out = sdati.getOutputStream();
DataInputStream netIn = new DataInputStream(is);
DataOutputStream netOut = new DataOutputStream(out);
...
```

Singlethreaded Server

```
public class SingleThreadedServer implements Runnable{

    protected int          serverPort   = 8080;
    protected ServerSocket serverSocket = null;
    protected boolean      isStopped    = false;
    protected Thread       runningThread= null;

    public SingleThreadedServer(int port){
        this.serverPort = port;
    }

    public void run(){
        synchronized(this){
            this.runningThread = Thread.currentThread();
        }
        try {
            this.serverSocket = new ServerSocket(this.serverPort);
        } catch (IOException e) {
            throw new RuntimeException("Cannot open port 8080", e);
        }

        while(! isStopped()){
            Socket clientSocket = null;
            try {
                clientSocket = this.serverSocket.accept();
            } catch (IOException e) {
                if(isStopped()) {
                    System.out.println("Server Stopped.") ;
                    return;
                }
                throw new RuntimeException(
                    "Error accepting client connection", e);
            }
            try {
                processClientRequest(clientSocket);
            } catch (Exception e) {
                //log exception and go on to next request.
            }
        }

        System.out.println("Server Stopped.");
    }

    private void processClientRequest(Socket clientSocket)
    throws Exception {
        InputStream  input  = clientSocket.getInputStream();
        OutputStream output = clientSocket.getOutputStream();
        long time = System.currentTimeMillis();
        byte[] responseDocument = …
        byte[] responseHeader = …
        output.write(responseHeader);
        output.write(responseDocument);
        output.close();
        input.close();
        System.out.println("Request processed: " + time);
    }
   private synchronized boolean isStopped() {
        return this.isStopped;
    }

   public synchronized void stop(){
        this.isStopped = true;
        try {
            this.serverSocket.close();
        } catch (IOException e) {
            throw new RuntimeException("Error closing server", e);
        }
    }
}
```

In short what the server does is this:

1. Wait for a client request
2. Process client request
3. Repeat from 1.

The single threaded server processes the incoming requests in the same thread that accepts the client connection.

Clients can only connect to the server while the server is inside the serverSocket.accept() method call.

The longer time the listening thread spends outside the serverSocket.accept() call, the higher the probability that the client will be denied access to the server.

This is the reason that multithreaded servers pass the incoming connections on to worker threads, who will process the request. That way the listening thread spends as little time as possible outside the serverSocket.accept() call.

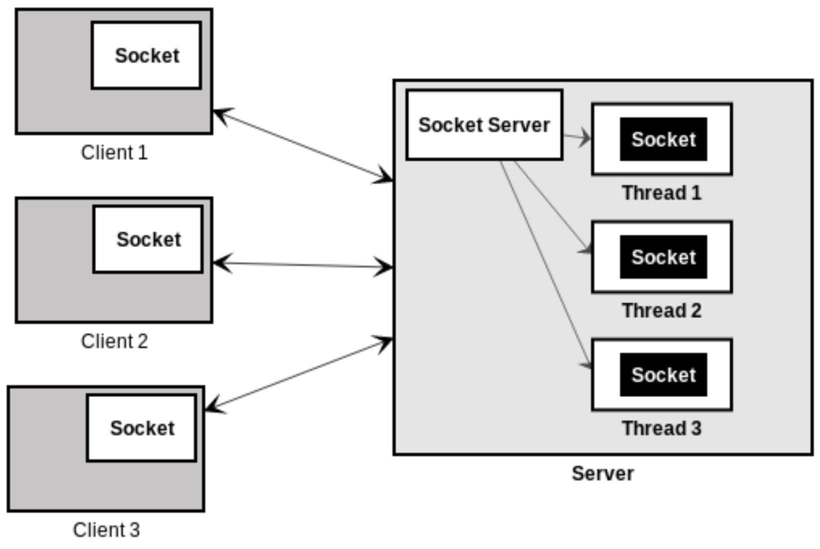
```
SingleThreadedServer server = new SingleThreadedServer(9000);
new Thread(server).start();

try {
    Thread.sleep(10 * 1000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
System.out.println("Stopping Server");
server.stop();
```

http://localhost:9000/

Multithreaded Server in Java

Rather than processing the incoming requests in the same thread that accepts the client connection, the connection is handed off to a worker thread that will process the request.



Here is how the server loop looks in the multithreaded edition:

```
    while(! isStopped()){
        Socket clientSocket = null;
        try {
            clientSocket = this.serverSocket.accept();
        } catch (IOException e) {
            if(isStopped()) {
                System.out.println("Server Stopped.") ;
                return;
            }
            throw new RuntimeException(
                "Error accepting client connection", e);
        }

            new Thread(
            new WorkerRunnable(
            clientSocket, "Multithreaded Server")
            ).start();

    }
```

The only change in the loop from the singlethreaded server to here is the code in bold:

```
new Thread(
    new WorkerRunnable(
        clientSocket, "Multithreaded Server")
).start();
```

The thread listening for incoming requests spends as much time as possible in the serverSocket.accept() call.

That way the risk is minimized for clients being denied access to the server because the listening thread is not inside the accept() call.

WorkerRunnable class passed to the worker thread constructor:

```
package servers;

import java.io.InputStream;
import java.io.OutputStream;
import java.io.IOException;
import java.net.Socket;

public class WorkerRunnable implements Runnable{
    protected Socket clientSocket = null;
    protected String serverText   = null;

    public WorkerRunnable(Socket clientSocket, String serverText) {
        this.clientSocket = clientSocket;
        this.serverText   = serverText;
    }

    public void run() {
        try {
            InputStream input  = clientSocket.getInputStream();
            OutputStream output = clientSocket.getOutputStream();
            output.write("…”).getBytes());
            output.close();
            input.close();
            System.out.println("Request processed: " + time);
        } catch (IOException e) {
            //report exception somewhere.
            e.printStackTrace();
        }
    }
}
```

Thread Pooled Server

Rather than starting a new thread per incoming connection, the connection is wrapped in a Runnable and handed off to a thread pool with a fixed number of threads.

The Runnable's are kept in a **queue** in the thread pool.
When a thread in the thread pool is idle it will take a Runnable from the queue and execute it.

```
    while(! isStopped()){
         Socket clientSocket = null;
         try {
             clientSocket = this.serverSocket.accept();
         } catch (IOException e) {
             if(isStopped()) {
                 System.out.println("Server Stopped.") ;
                 break;
             }
             throw new RuntimeException(
                "Error accepting client connection", e);
         }

            this.threadPool.execute(
            new WorkerRunnable(clientSocket, "Thread Pooled Server"));

     }
```

The only change in the loop from the multithreaded server to here is the code in bold.
Rather than starting a new thread per incoming connection, the WorkerRunnable is passed to the thread pool for execution when a thread in the pool becomes idle.

```
…
public class WorkerRunnable implements Runnable{
    protected Socket clientSocket = null;
    protected String serverText   = null;

    public WorkerRunnable(Socket clientSocket, String serverText) {
        this.clientSocket = clientSocket;
        this.serverText   = serverText;
    }

    public void run() {
        try {
            InputStream input  = clientSocket.getInputStream();
            OutputStream output = clientSocket.getOutputStream();
            output.write(("…").getBytes());
            output.close();
            input.close();
            System.out.println("Request processed: " + time);
        } catch (IOException e) {
            //report exception somewhere.
            e.printStackTrace();
        }
    }
}
```

```
ThreadPooledServer server = new ThreadPooledServer(9000);
new Thread(server).start();

try {
    Thread.sleep(20 * 1000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
System.out.println("Stopping Server");
server.stop();
```

We won’t be able to accept more connections than threads available in our machine. Therefore, if you are expecting to have many connections, you need an alternative.

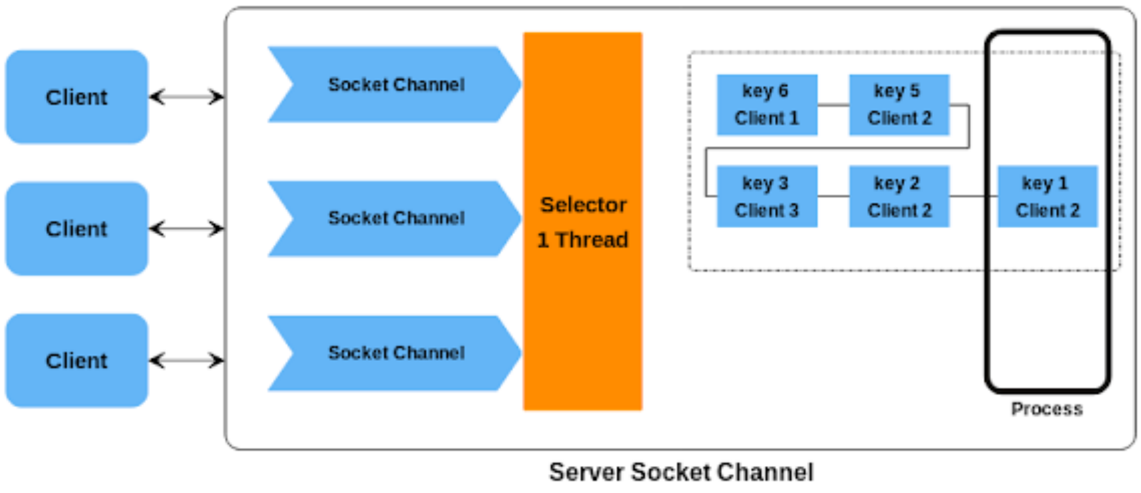
Java NIO (New IO)

Java.nio is a non-blocking API for socket connections, which means you are not tight to the number of threads available.

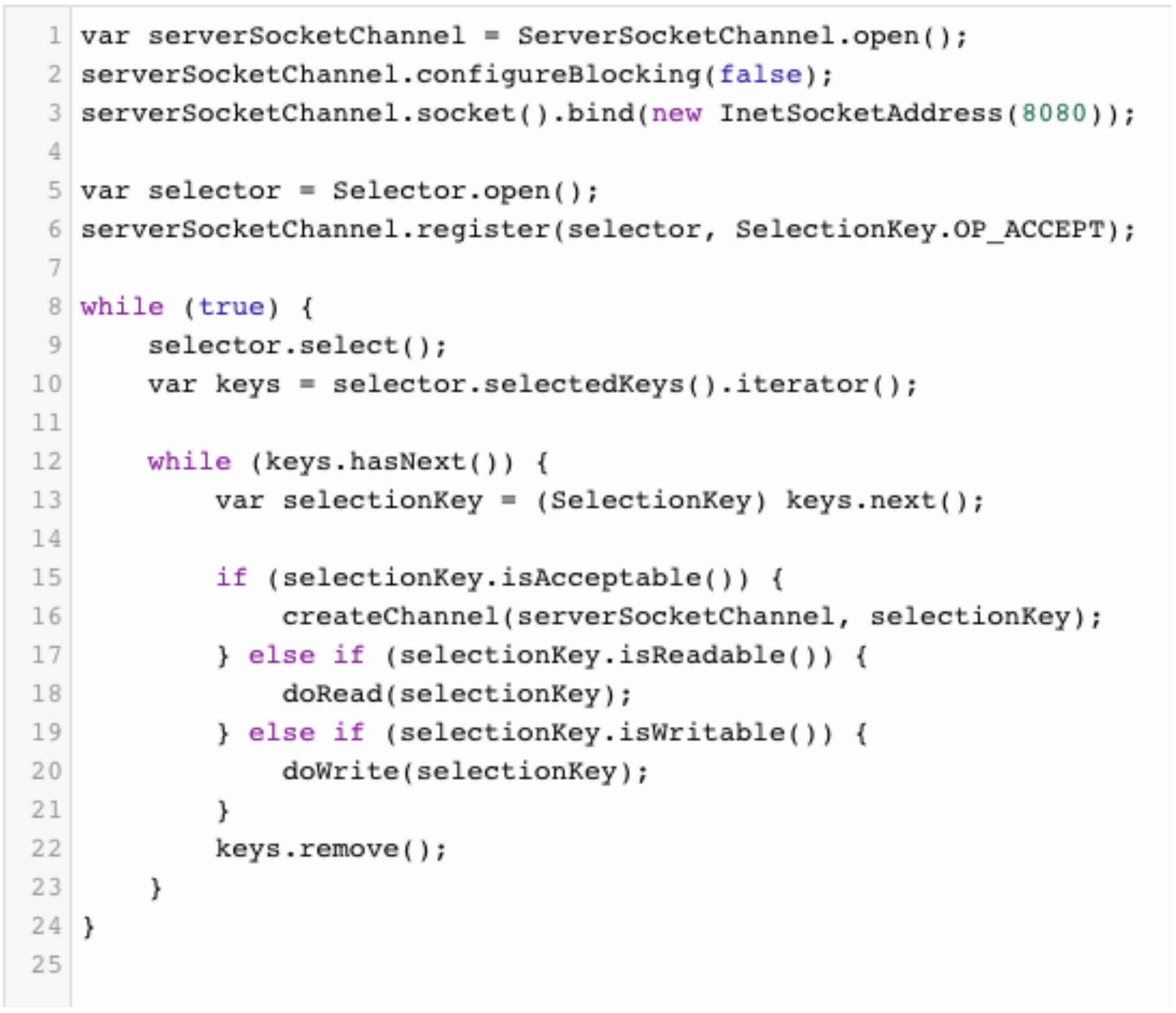
Main elements:

- **Channel**: channels are a combination of input and output streams, so they allow you to read and write, and they use buffers to do these operations.
- **Buffer**: it is a block of memory used to read from a Channel and write into it.
- **Selector**: A Selector can register multiple Channels and will check which ones are ready for accepting new connections. Similar to accept() method of blocking IO, when select() is invoked, it will block the application until a Channel is ready to do an operation. Because a Selector can register many channels, only one thread is required to handle multiple connections.
- **Selection Key**: It contains properties for a particular **Channel**.
  Selection keys are mainly used to know the current interest of the channel (isAcceptable(), isReadable(), isWritable()), get the channel and do operations with that channel.

With this library, one thread can handle multiple connections at once.



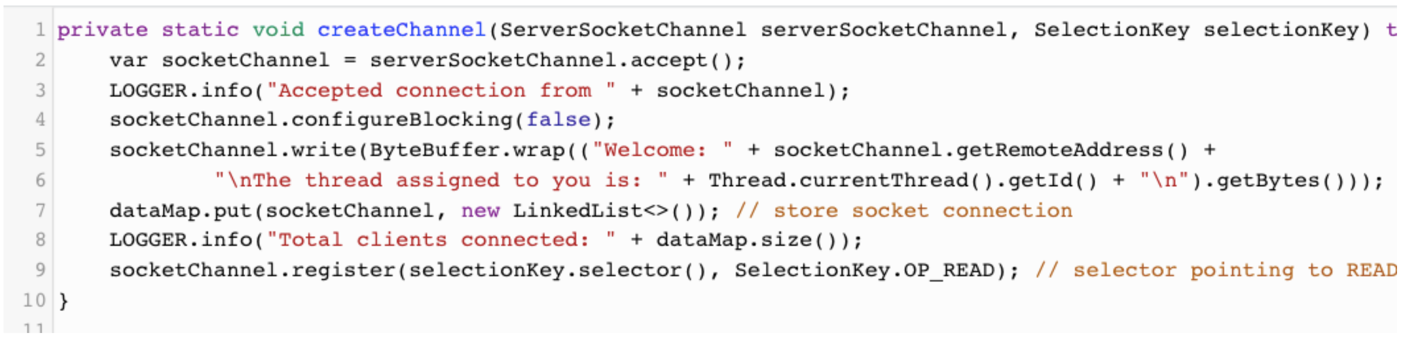
-



```
 1 var serverSocketChannel = ServerSocketChannel.open();
 2 serverSocketChannel.configureBlocking(false);
 3 serverSocketChannel.socket().bind(new InetSocketAddress(8080));
 4
 5 var selector = Selector.open();
 6 serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
 7
 8 while (true) {
 9     selector.select();
10     var keys = selector.selectedKeys().iterator();
11
12     while (keys.hasNext()) {
13         var selectionKey = (SelectionKey) keys.next();
14
15         if (selectionKey.isAcceptable()) {
16             createChannel(serverSocketChannel, selectionKey);
17         } else if (selectionKey.isReadable()) {
18             doRead(selectionKey);
19         } else if (selectionKey.isWritable()) {
20             doWrite(selectionKey);
21         }
22         keys.remove();
23     }
24 }
25
```

1. From lines 1 to 3, a ServerSocketChannel is created, and you have to set it to non-blocking mode explicitly. The socket is also configured to listen on *port 8080*.
2. On lines 5 and 6, a Selector is created and ServerSocketChannel is registered on the Selector with a SelectionKey pointing to ACCEPT operations.
3. To keep the application listening all the time, the blocking method select() is inside an infinite while loop and select() will return when at least one channel is selected wakeup() is invoked or the thread is interrupted.
4. Then, on line 10, a set of keys are returned from the Selector and we will iterate through them in order to execute the ready channels.Every time a new connection is created, isAcceptable() will be true and a new Channel will be registered into the Selector.

Example of channel creation



```java
private static void createChannel(ServerSocketChannel serverSocketChannel, SelectionKey selectionKey) t
    var socketChannel = serverSocketChannel.accept();
    LOGGER.info("Accepted connection from " + socketChannel);
    socketChannel.configureBlocking(false);
    socketChannel.write(ByteBuffer.wrap(("Welcome: " + socketChannel.getRemoteAddress() +
            "\nThe thread assigned to you is: " + Thread.currentThread().getId() + "\n").getBytes()));
    dataMap.put(socketChannel, new LinkedList<>()); // store socket connection
    LOGGER.info("Total clients connected: " + dataMap.size());
    socketChannel.register(selectionKey.selector(), SelectionKey.OP_READ); // selector pointing to READ
}
```

1. To keep track of the data of each channel, it is put in a Map with the socket channel as the key and a list of ByteBuffers.
2. Then, the selector will point to a *READ* operation.

IO vs NIO

Java IO being stream oriented means that you read one or more bytes at a time, from a stream. What you do with the read bytes is up to you. They are not cached anywhere. Furthermore, you cannot move forth and back in the data in a stream. If you need to move forth and back in the data read from a stream, you will need to cache it in a buffer first.

Java NIO's buffer oriented approach is slightly different. Data is read into a buffer from which it is later processed. You can move forth and back in the buffer as you need to. This gives you a bit more flexibility during processing. However, you also need to check if the buffer contains all the data you need in order to fully process it. And, you need to make sure that when reading more data into the buffer, you do not overwrite data in the buffer you have not yet processed.

Blocking vs. Non-blocking IO

Java IO's various streams are blocking. That means, that when a thread invokes a read() or write(), that thread is blocked until there is some data to read, or the data is fully written. The thread can do nothing else in the meantime.

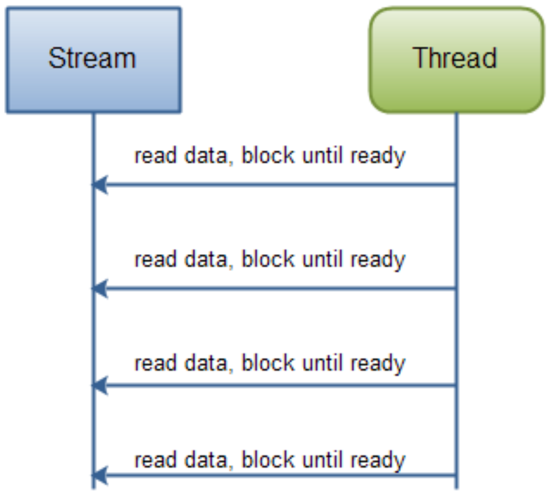
Java NIO's non-blocking mode enables a thread to request reading data from a channel, and only get what is currently available, or nothing at all, if no data is currently available. Rather than remain blocked until data becomes available for reading, the thread can go on with something else.

The same is true for non-blocking writing. A thread can request that some data be written to a channel, but not wait for it to be fully written. The thread can then go on and do something else in the mean time.

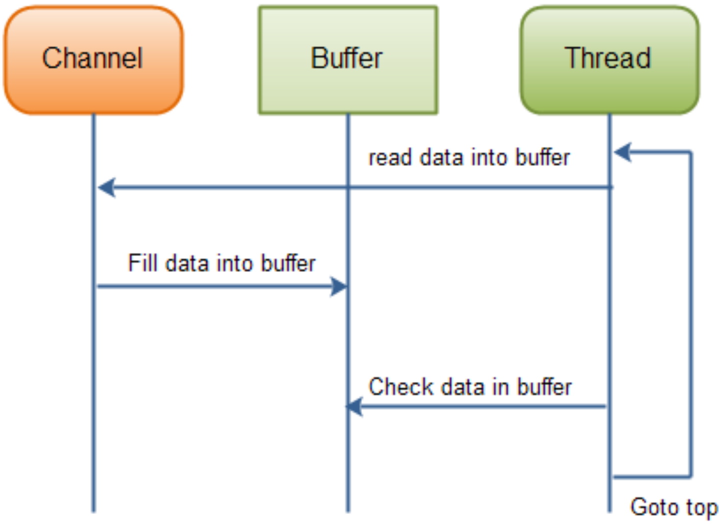
What threads spend their idle time on when not blocked in IO calls, is usually performing IO on other channels in the meantime. That is, a single thread can now manage multiple channels of input and output.

Selectors

Java NIO's selectors allow a single thread to monitor multiple channels of input. You can register multiple channels with a selector, then use a single thread to "select" the channels that have input available for processing, or select the channels that are ready for writing. This selector mechanism makes it easy for a single thread to manage multiple channels.


Stream
Thread
read data, block until ready
read data, block until ready
read data, block until ready
read data, block until ready

Java IO: Reading data from a blocking stream.

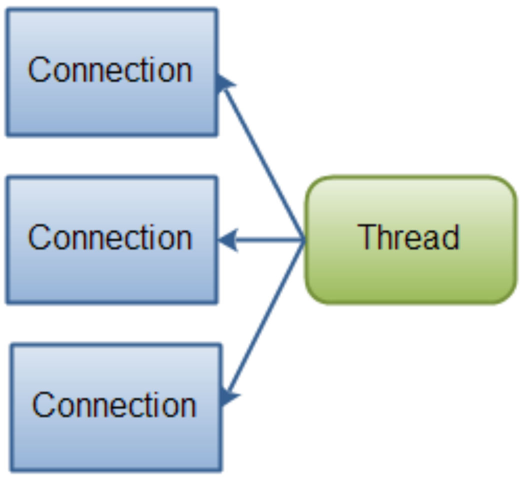


Java NIO: Reading data from a channel until all needed data is in buffer.

Summary

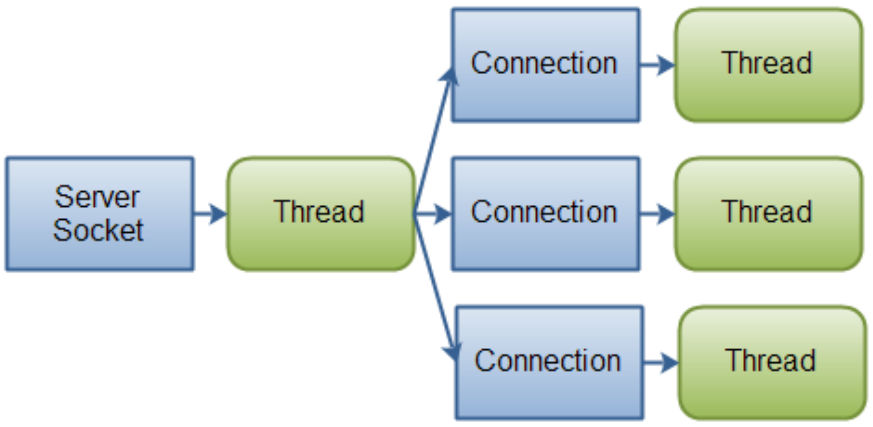
NIO allows you to manage multiple channels (network connections or files) using only a single (or few) threads, but the cost is that parsing the data might be somewhat more complicated than when reading data from a blocking stream.

If you need to manage thousands of open connections simultanously, which each only send a little data, for instance a chat server, implementing the server in NIO is probably an advantage. Similarly, if you need to keep a lot of open connections to other computers, e.g. in a P2P network, using a single thread to manage all of your outbound connections might be an advantage. This one thread, multiple connections design is illustrated in this diagram:



Java NIO: A single thread managing multiple connections.

If you have fewer connections with very high bandwidth, sending a lot of data at a time, perhaps a classic IO server implementation might be the best fit. This diagram illustrates a classic IO server design:


Server Socket
Thread
Connection
Thread
Connection
Thread
Connection
Thread