latex input: mmd-article-header Title: Java Network Programming Notes Author: Ethan C. Petuchowski Base Header Level: 1 latex mode: memoir Keywords: Java, programming language, syntax, fundamentals CSS: http://fletcherpenney.net/css/document.css xhtml header: copyright: 2014 Ethan Petuchowski latex input: mmd-natbib-plain latex input: mmd-article-begin-doc latex footer: mmd-memoir-footer

Overview

- The *only* transport-layer protocols Java supports are TCP & UDP; for anything else, you must link to native code via the Java Native Interface
- TCP uses *stream* sockets, through which one generally just writes to an `OutputStream` and reads from an `InputStream` of bytes that remain in-order and uncorrupted and are (practically) guaranteed delivery by the implementation of the protocol by the operating system.
    - Unless you're using NIO; see below for more on that
- UDP uses *datagram* sockets, through which you `send` and `receive` objects called `DatagramPackets`, which are just a length, a destination, and data
- Unless you're using NIO, everything *blocks*: e.g. connecting to servers, listening for clients, reads, writes, and disconnecting (for TCP)
    - By default most of these actions may block *indefinitely*
    - For reading and connecting, you can configure a timeout, after which you will receive an `InterruptedIOException`
    - For *writing* to a TCP stream, you *cannot* configure a timeout
- If your server must handle multiple clients, there are a few basic options
    - Deal with one at a time

```
void mainLoop() {
    while (true) {
        Socket s = serverSocket.accept();
        handle(s);
    }
}
void handle(Socket s) {
    InputStream in = s.getInputStream();
    // process request, etc.
    s.close();
}
```

    - Create a new thread to handle each incoming client

```
void mainLoop() {
    while (true) {
        Socket s = serverSocket.accept();
        new Thread() {
            @Override public void run() {
                InputStream in = s.getInputStream();
                // process request, etc.
                s.close();
            }
        }.start()
    }
}
```

- Use a thread pool to handle requests

```
// There are actually a multitude of executors to choos
// from. This one will execute each task on an existing
// thread if one is idle, and will create a thread othe
// Threads idle for too long are terminated.
ExecutorService executor = Executors.newCachedThreadPoo

void mainLoop() {
    while (true) {
        Socket s = serverSocket.accept();
        executor.execute(new TheHandler(s));
    }
}
static class TheHandler implements Runnable {
    Socket s;
    public TheHandler(Socket s) { this.s = s; }
    @Override public void run() {
        InputStream in = s.getInputStream();
        // process request, etc.
        s.close();
    }
}
```

- Use NIO (rather complicated)
- Use a framework like Netty, Akka, etc. that wraps this stuff up in a ribbon and a tie

10K feet above NIO

- If you're using NIO, you create `Channels` of bytes into and out of sockets (or file handles)
- You register a `Selector` to be notified when the `Channel` is ready to be read from or written to

- You query the `Selector` to tell you which `Channel` are ready, and may then take action on those that are
- You get data in and out by passing a `Buffer` to the `Channel`

Here's an example based on TCP/IP Sockets in Java, a highly recommended book about this stuff by Calvert and Donahoo.

```java
Selector slctr = Selector.open(); // factory
ServerSocketChannel chnl = ServerSocketChannel.open(); // factory
chnl.socket().bind(inetAddr); // set address to listen on

// For some reason Channels block by default. If we want to
// register with the Selector for notifications, we must turn
// that off.
chnl.configureBlocking(false);

// Notify Selector whenever this Channel has a new connection
// ready to be "accepted". Such a notification still does
// *not* guarantee it will work immediately.
chnl.register(slctr, SelectionKey.OP_ACCEPT);

while (true) {
    // wait configurable period of time to be notified
    // by any registered channel
    int numNotifications = slctr.select(timeoutMS);
    if (numNotifications == 0) {
        // We timed out without any notification.
        // We could do whatever we want here because we're
        // no longer blocked.
    } else {
        // numNotifications different channels have notified us of
        // being available for Connect, Read, Accept, or Write.
        // It is OK to use these keys in concurrent threads.
        for (SelectionKey key : slctr.selectedKeys()) {
            // We're not sure which channel this key belonged to.
            // Also, notification was just a "hint" and we need to
            // check again whether the Channel is available.
            if (key.isAcceptable()) {
                // here's the actual call to accept()
                SocketChannel clientChnl =
                    ((ServerSocketChannel) key.channel()).accept();

                // similar to the ServerSocketChannel
                clientChnl.configureBlocking(false);
                // Except that here we register to notify Selector
                // about being "readable", and
                clientChnl.register(
                    key.selector(),
                    SelectionKey.OP_READ,
                    // We must associate an "attachment" with this
                    // channel. This is the buffer that will be
                    // filled with the incoming bytes rcvd via TCP.
                    ByteBuffer.allocate(NUM_BYTES) // eg 256?
                );
            }
            if (key.isReadable()) {
                // retrieve the readable client socket's channel
                SocketChannel client =
                    (SocketChannel) key.channel();
                // retrieve the ByteBuffer we associated with
                // that channel
                ByteBuffer buf = (ByteBuffer) key.attachment();

                // Attempt to read `buf.remaining()` bytes _from_
                // the Channel _into_ the ByteBuffer.
                int bytesRead = client.read(buf);

                // -1 from read() means end-of-stream, which in
                // this case means the client closed their output
                // side of the TCP connection. We may still be
                // able to send data if that side of the connection
                // has not been closed yet.
                if (bytesRead == -1) client.close();

                else if (bytesRead > 0) {
                    // if our application has data to write back
                    // to the client, we must tell the selector
                    // that we've now become interested in writing
                    key.interestOps(SelectionKey.OP_READ
                                    | SelectionKey.OP_WRITE);
                }
            }
            // socket not closed, and is writable
            if (key.isValid() && key.isWritable()) {
                // beyond the scope of this.
            }
        }
    }
}
```

Tips for Traps

- Don't write to the network through a `PrintStream`
    - It chooses end-of-line chars based on your platform, not the protocol (HTTP uses `\r\n`)
    - It uses the default char encoding of your platform (likely UTF-8), not whatever the server expects (likely UTF-8)
    - It eats all exceptions into this `boolean checkError()` method, when you're better off just using the normal exception hubbub

Connecting to Addresses

class InetAddress

- `java.net.InetAddress` --- Java's representation of an IP address (v4 or v6)

    - DNS lookups are provided by this class
- Acquire one via a static factory

  ```
  InetAddress address = InetAddress.getByName("www.urls4all.co
  ```

  This will look in your cache, and if it's not there connect to your DNS to get the IP address

class URL

- Simplest way to locate and retrieve data from the network
- `final class java.net.URL (extends Object)` uses *strategy design pattern* instead of inheritance to configure instances for different kinds of URLs
    - E.g. protocol handlers are strategies (note these are *application* layer protocols, e.g. HTTP)
- Think about it has having fields like
    - Protocol, hostname, port, path, query string, fragment identifier
- Immutable (makes it thread safe)
- Some constructors (all `throw MalformedURLException`)

  ```
  URL(String url)
  URL(String protocol, String hostame, String file)
  URL(String protocol, String host, int port, String file)
  URL(URL base, String relative)
  ```

- To get data from it you have

  ```
  InputStream   openStream()              // most common
  URLConnection openConnection([Proxy])   // more configurable
  Object        getContent([Class[]])     // don't use this
  ```

- Encode Strings into URLs using

  ```
  String encoded = URLEncoder.encode("MyCrazy@*&^ STring", "UT
  ```

    - There is a similar `decode(String s, String encoding)` method

Web Scraping

Jsoup

This is a 3rd party library for downloading and traversing Web content which allows jQuery-style selecting.

```
Document doc = Jsoup.connect("http://en.wikipedia.org/").get();
Elements newsHeadlines = doc.select("#mp-itn b a");
```

Utilities

Bind server to first available port among given choices

From Stackoverflow

```
public ServerSocket create(int[] ports) throws IOException {
    for (int port : ports) {
        try { return new ServerSocket(port); }
        catch (IOException ex) { continue; } /* try next port */
    }
    throw new IOException("no free port found");
}
```

Now use it like so:

```
try {
    ServerSocket s = create(new int[] { 3843, 4584, 4843 });
    System.out.println("listening on port: " + s.getLocalPort());
}
catch (IOException ex) { System.err.println("no available ports");
```

Advanced protocol development

Netty

- Library for implementing fast & scalable network protocols over TCP/UDP
    - e.g. when a plain-jane HTTP server is not going to cut it for serving your huge files.
- It uses Java's NIO framework, but is easier to use
- It is an *asynchronous event-driven network application framework* along with tooling for rapid development of maintainable, high-performance, high-scalability protocol servers and clients.
- It facilitates TCP/UDP socket server development for custom protocols using Java's NIO framework

Sources

- Netty User Guide