Improving Java Network Programming

Brian Runk, b.runk@morganstanley.com
25 Apr 2006
Topics

- Background
- A simple distributed application
- `java.net` programming model
- `java.nio` programming model
- A better programming model
- Issues
- Q & A
Enterprise Application Infrastructure

• Global team providing software infrastructure and developer tools
  - C++, Java, .Net, Perl, Python
  - Linux, Solaris, Windows

• True software reuse
  - Single place for feature requirements and bug fixes
  - Well known repositories for documentation and best practices
  - All application developers can leverage expertise

• Support for Firmwide initiatives

• Changes to infrastructure affects many applications
Java Toolkit Team

• 15 people globally
• 40+ proprietary and open source libraries
• Developer tools, build infrastructure
• Documentation, best practices, guidelines
• Hundreds of Java developers
• Thousands of Java applications
A Simple Distributed Application

Client -> Server
Definitions

• **Connection**: a two way communication channel between two processes

• **Socket**: one end of a connection

• **Client**: connection initiator

• **Server**: connection acceptor

• **Message**: a meaningful and complete set of bytes
• **Blocking sockets**
  - you wait until work is done, maybe forever
    - no timeout for synchronous operations

• **Simple stream based API**
  
  ```java
  Socket s = new Socket("foo.ms.com", 12345);
  
  - write these bytes
  byte[] data = ...;
  s.getOutputStream().write(data);

  - I’m expecting data, read as much as you can into this byte buffer, tell me how many bytes you got
  byte[] data = new byte[1024];
  int numRead = s.getInputStream().read(buf);
  ```
• Servers must have at least one thread per client

ServerSocket svr = new ServerSocket(12345);

while (Socket s = svr.accept())
{
    new SocketProcessorThread(s).start();
}
class SocketProcessorThread extends Thread {
    SocketProcessorThread(Socket s) { ... } 
    public void run() {
        do {
            Request req = readRequest(s.getInputStream());
            Response res = calculateResponse(req);
            s.getOutputStream().write(res);
        }
        while (the client hasn't disconnected or otherwise indicated that it's done)
    }
}

• Doesn’t scale to thousands of clients
• **Non-blocking sockets**
  - will only do as much as they can without blocking
  - report back to you how much was done

• **Complex Selector based API**

```java
SocketChannel sc = SocketChannel.open();
InetSocketAddress addr = ...;
if (!sc.connect(addr))
{
    Selector selector = Selector.open();
    sc.register(selector, SelectionKey.OP_CONNECT);
    if (selector.select() > 0) { //blocks until ready
        if (!sc.finishConnect()) {
            // not connected
        }
    }
}
```
ServerSocketChannel server = ServerSocketChannel.open();
server.configureBlocking(false);
server.socket().bind(new InetSocketAddress(port));
Selector selector = Selector.open();
server.register(selector, SelectionKey.OP_ACCEPT);
while (true) {
    if (selector.select() > 0) {
        for (SelectionKey key : selector.selectedKeys()) {
            if (key.isAcceptable()) {
                // accept client SocketChannel
                // register it with the selector for READ
            } else if (key.isReadable()) {
                // do something with data read from SocketChannel
                // unless there was none, meaning the client disconnected
                // if you want to send a response, hang on to it
                // and register with selector for WRITE
            } else if (key.isWritable()) {
                // get the message you held on to for this client
                // send as much as you can. if done, unregister
                // with selector for WRITE
            }
        }
    }
}
Event Based Network Programming

• Callback API for network events
  - got connected
  - got disconnected
  - data arrived
  - sent data completed
  - new client connection
  - client connection disconnected

• Single thread for I/O
Event Based Client

IOThread loop = new IOThread();
loop.start();

Client c = new Client(loop, new HostPort("foo.ms.com:12345"), "FooClient");
c.addListener(new ClientListener());
c.startConnect(); // asynchronous
public class ClientListener implements ClientCallbacks {
    public void connectCallback() {
        // connection is established, send login data
        // or first message
    }
    public void disconnectCallback() {
        // the server disconnected. what can we do
        // other than log an error?
    }
    public void readCallback(byte[] data) {
        // data arrived, can do something now.
        // but what if it’s not a whole message?
        // what if it’s three messages?
    }
    public void sendDone() {
        // in case you’re wondering
    }
}
Event Based Server

```java
IOThread loop = new IOThread();
loop.start();
Server s = new Server(loop, 12345, "FooServer");
c.addListener(new ServerListener());
c.startAccepting();
```
public class ServerListener implements ServerCallbacks {
    public void clientConnectCallback(Client c) {
        // a new client has connected
        // initialize any client context
    }

    public void clientDisconnectCallback(Client c) {
        // the client disconnected. cleanup
    }

    public void readCallback(Client c, byte[] data) {
        // client sent some data, can do something now.
        // but what if it’s not a whole message?
        // what if it’s three messages?
    }
}
Issues

• Long running callbacks
  – I/O thread is shared. Long running callbacks slow down other clients

• Message framing
  – What if the bytes I get aren’t quite the bytes I want?

• Resilience
  – What if I didn’t want to get disconnected?
Long Running Callbacks

• Keep callbacks short
  - requires developer compliance
  - not always possible

• Distribute connections in a pool of I/O threads
  - same problem exists but at potentially smaller scale

• Only use I/O threads for I/O
  - use a thread pool for application work
Message Framing

• Message definition framework

```java
public interface MessageDefinition {
    byte[] makeMessage(byte[] readBuffer);
}
```

• Framework buffers read bytes
  – Uses application provided `MessageDefinition` implementation to make messages
  – Only invokes `readCallback` with complete messages.
Resilience

• If connect fails, keep trying.
  − solves startup ordering problem
  − exponential backoff

• If you get disconnected, try to reconnect
  − same as connect
  − list of primary and backup servers
    • try primary servers first
      − start with server you were just connected to
    • any primary ok
    • backups should be failed back

• Only clients reconnect
Resilience

- Lost messages
  - No guarantees over TCP

- Guaranteed message delivery
  - Message persistence + acknowledgement
    - Requires login
    - Potentially complicates failover
  - Messaging middleware
    - Decouples message producer and consumer
    - Potentially adds latency
    - Transactional capability
  - Turns lost message problem into duplicate message problem
Questions?
Thank you.

Brian Runk, b.runk@morganstanley.com