Server for the *wavu* mobile client.

By Piotr Szymański szyman@magres.net.

Table of contents

1. Introduction

The wavu mobile groupware application was developed by Igor Kowalczyk, Piotr Wardaszko, Marcin Więckowski and myself for the Mobile Platform Application Development course at the Technical University of Denmark. The goals of the project were to create a friendly application that would help the user in:

- keeping personal notes and synchronizing them with others,
- exchanging short messages over Bluetooth and Internet connections.

The mobile client was written in Java 2 Micro Edition [1] (J2ME). The server was written in PHP for an easier integration with the web environment.

This project is done in cooperation with Piotr Wardaszko.

2. Project objectives

The objective of this project was to rewrite the server using a more suitable language than PHP. As PHP is primarily a web scripting language, it is not very suitable to build standalone server applications. Its primary drawback is the lack of high-performance socket operation capabilities and the lack of support for threads – only separate processes with shared memory (however, this shared memory cannot be utilized directly to share variables between processes).

As the mobile client is implemented in Java, this language was the most obvious choice for the server platform, as it would make it possible to share some code between the two applications. The server was designed with the following goals in mind:

- Support a large number of simultaneous clients (in the range of thousands), which all maintain a single active TCP connection for the duration of the whole session.
- Have a scalable architecture – support multiple processors.
- Be independent of the persistent storage mechanism – different database types, non-SQL databases, etc.
- Be capable of supporting different communication protocols for possible future uses (possibly not only a server for the wavu application).

The server obviously has to be capable of providing the same services which were also provided by the PHP version of the server. The mentioned services include:

- User authentication
- Contacts (“buddy-list”) and presence information management
- Notes and folders management
- Chat over TCP connection.

As this project is done in cooperation with Piotr Wardaszko, we have divided the work between ourselves in the following manner:

- Conceptual design and UML modelling was performed in strict cooperation.
- My colleague is responsible mainly for the persistence layer (database schemas, data objects and all the associated processing) and stronger encryption support.
- I am responsible for the networking layer (socket IO mechanism, application level protocol, worker thread management).

3. Server design

3.1. Technologies used

In order to fulfill the project design goals, the following technologies were employed:

- Java Platform Standard Edition 6.0 Virtual Machine
- `java.nio` package for high-performance non-blocking IO operations
- `java.util.concurrent` package for thread pool management
- Java Data Objects (JDO) for the persistence mechanism
- Log4j package for logging support.

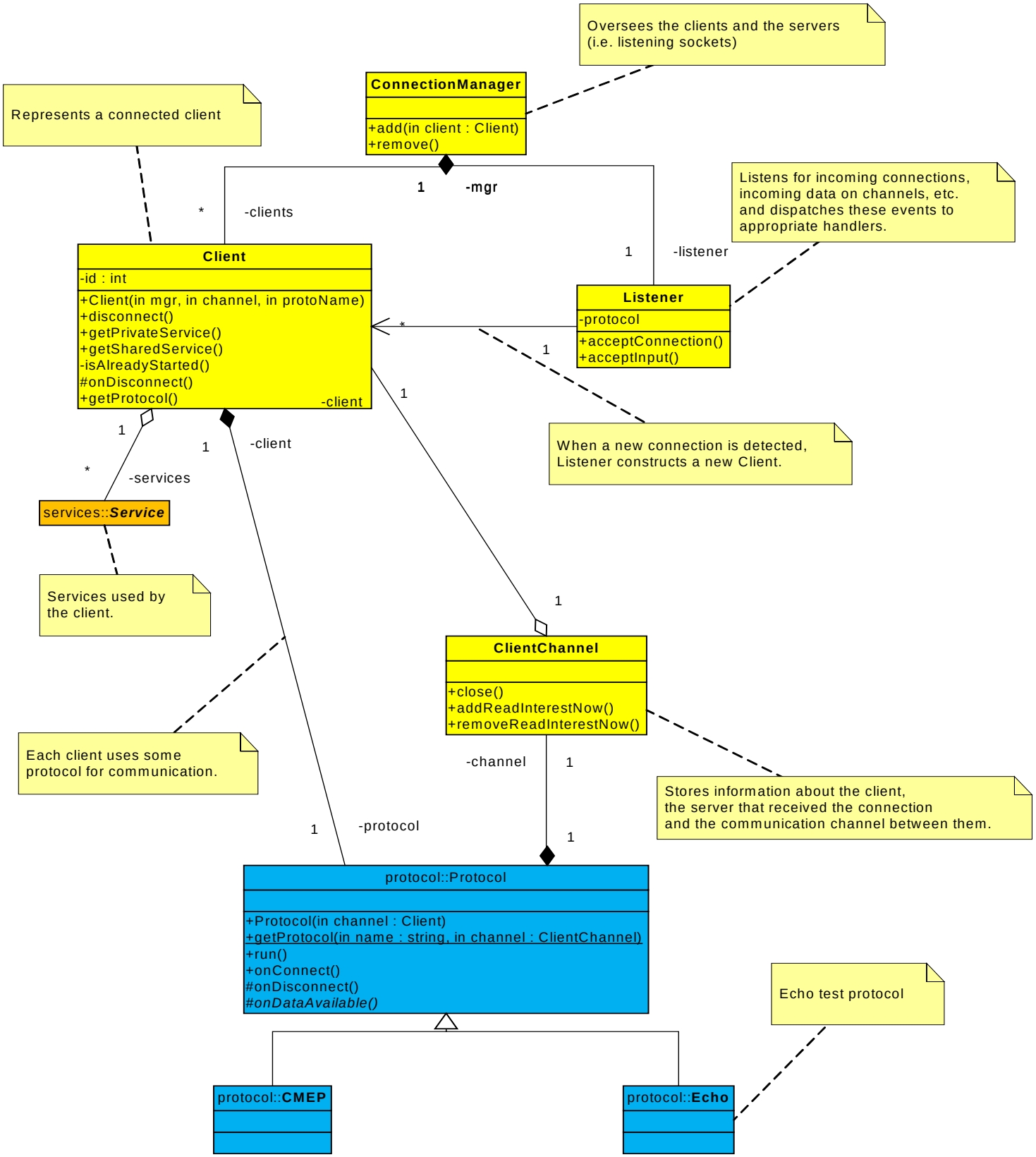
In my description I will concentrate on the parts which were my responsibility.

3.2. Overall server structure

The server architecture is based upon the following primary components:

- Instances of the `Client` class, which represent the connected clients and hold their associated data.
- `Service`s which provide certain functionality that is available to the clients.
- `Protocol`s which provide a means of communication between the connected `Client` and the `Service`s offered by the server.

Due to this design, the server can be used to offer different services using many types of protocols. The diagram below illustrates the mentioned architecture:


Oversees the clients and the servers
(i.e. listening sockets)
ConnectionManager
+add(in client : Client)
+remove()
Represents a connected client
1
-mgr
*
-clients
Listens for incoming connections,
incoming data on channels, etc.
and dispatches these events to
appropriate handlers.
1
-listener
Client
-id : int
+Client(in mgr, in channel, in protoName)
+disconnect()
+getPrivateService()
+getSharedService()
-isAlreadyStarted()
#onDisconnect()
+getProtocol()
-client
Listener
-protocol
+acceptConnection()
+acceptInput()
*
1
1
When a new connection is detected,
Listener constructs a new Client.
1
1
-client
*
-services
services::Service
Services used by
the client.
1
ClientChannel
+close()
+addReadInterestNow()
+removeReadInterestNow()
Each client uses some
protocol for communication.
-channel
1
Stores information about the client,
the server that received the connection
and the communication channel between them.
1
-protocol
1
protocol::Protocol
+Protocol(in channel : Client)
+getProtocol(in name : string, in channel : ClientChannel)
+run()
+onConnect()
#onDisconnect()
#onDataAvailable()
Echo test protocol
protocol::CMEP
protocol::Echo

3.3. Low level IO layer

The solution for the low level IO and request management employed in this server is based upon the **1 dispatcher/N workers** architecture, as outlined by this [2] article. In this architecture, a single dispatcher thread monitors the sockets for IO events, while the actual handling of requests is performed by multiple worker threads.

Other possible architectures include **N dispatchers/no workers** and **M dispatchers/N workers**. Only the latter solution has obvious benefits, but the implementation complexity and the practical efficiency of the solution with a single dispatcher prevented us from choosing that architecture.

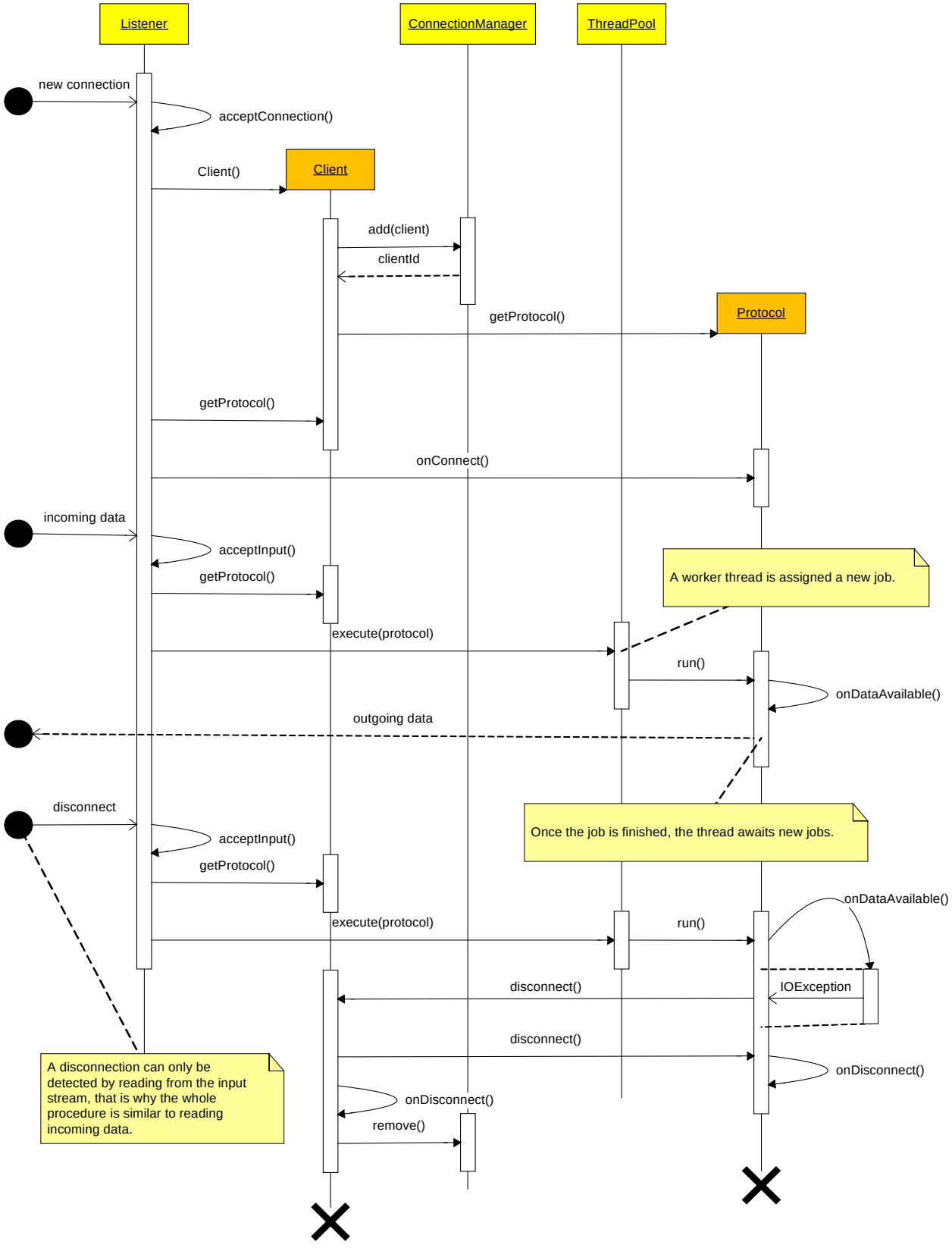
The low level input-output layer consists of a single separate `Listener` thread that monitors all the TCP/IP sockets used by the server. These include two types of sockets:

- **Listening sockets -** the server's sockets bound to certain ports – these accept incoming connections
- **Client sockets -** the sockets representing established connections to the clients.

All sockets used by the server are in the non-blocking state.

While the `Listener` thread is in the monitoring state, it does not consume any processor cycles, as it is placed in a blocking state waiting for some external event to interrupt it. When the monitoring thread detectes activity on any of the sockets (such as an incoming connection, a terminated connection, incoming data or socket ready to accept outgoing data), it dispatches a task to handle that event in a separate worker thread. All such tasks are handled by the proper `Protocol` class which is used to communicate over the given socket.

The handling of socket events is illustrated in the sequence diagram below.



3.3.1. Thread management

All incoming requests are handled by separate worker threads, which allows the `Listener` thread to quickly return to its primary task of monitoring socket activity. The task of managing the threads is given to a standard Java package `java.util.concurrent`, which provides some typical mechanisms for handling multiple threads.

The server keeps a pool of worker threads (with a configurable maximum) and assigns incoming jobs to free worker threads as they come by. If no free thread is currently available,

a new thread is created or, if the pool size limit doesn't allow it, the job is executed in the `Listener` thread.

This approach minimizes the delay associated with creation of new threads for handling jobs, and also minimizes the memory usage of having a separate thread for each connected client.

3.3.2. Data flow management

The server application needs to cope with situations when there is more incoming data than it can handle and when it wants to send more outgoing data than the connected client can accept (or the transport layer can transmit). As the server operates solely on non-blocking sockets, the application must provide its own flow handling routines.

The solution employed in this server involves two methods of the `Protocol` class: `onDataAvailable()` and `onDataWritable()`, and two *states*: read and write interests.

When the `Protocol` is ready to accept more incoming data, it enables its read interest. When new data arrives, the read interest is automatically disabled so that the `Protocol` has to explicitly enable it to receive more data, and then the `onDataAvailable()` method is called.

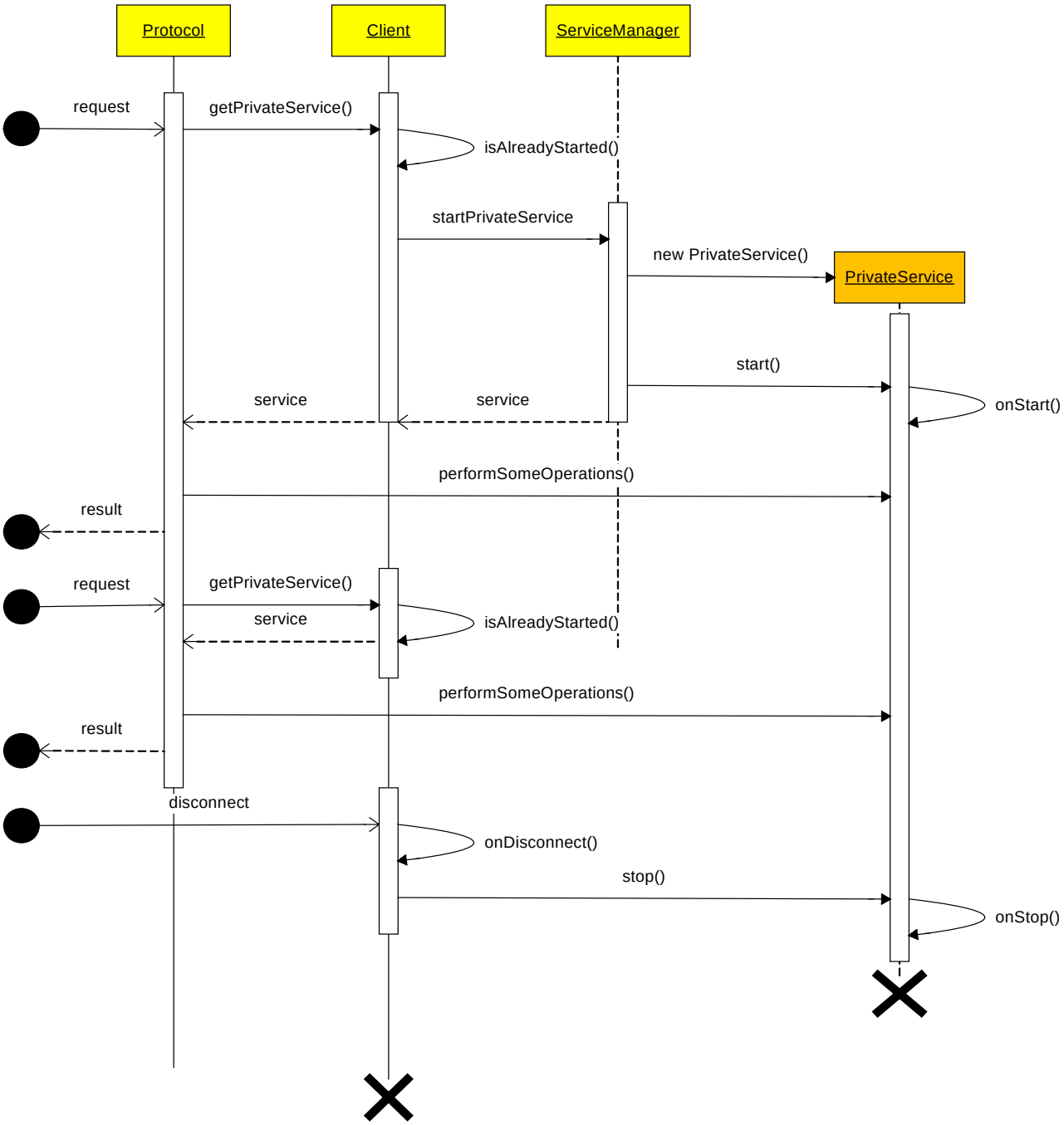
When the `Protocol` has some data to send it should enable its write interest. Once the Listener detects that the socket is ready for sending data, it will first disable the write interest and then call the `onDataWritable()` method.

3.4. Application level protocol

The protocol used by the **wavu** mobile client is called the Compact Message Exchange Protocol (CMEP). It has been designed specifically for the needs of a high-latency, low-bandwidth environment like the GPRS network. A detailed description of this protocol is available in **Appendix A**.

3.5. Service management

The lifecycle of application services is given in the diagram below:



4. Testing methodology and results

The low level communication subsystem of the server has been tested to find out how the server behaves under different load conditions and thread pool sizes.

4.1. Methodology

The tests have been conducted using `ApacheBench, Version 2.0.40-dev`, a tool being part of the Apache httpd2 web server. In order to test the server using the `HTTP`

protocol, a very simple class that understands this protocol was implemented using the `Protocol` base class. The operation of this class is as follows:

1. Accept a new connection.
2. Read and discard all data until an empty line is encountered.
3. Output a simple `HTTP/1.0 200 OK` header and some content whose length is equal to 33712 bytes.
4. Terminate the connection.

The server has been restarted after each test, except for the tests which were marked as conducted on a "warm" server. The server and the testing program were run on separate machines communication over IPv4 TCP protocol.

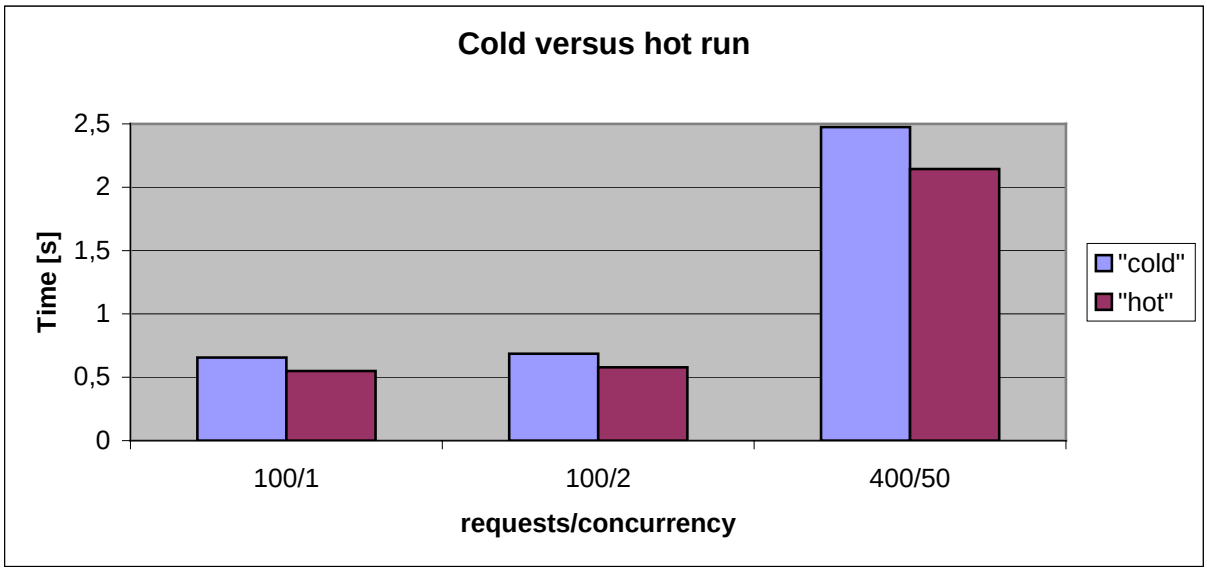
4.2. Results

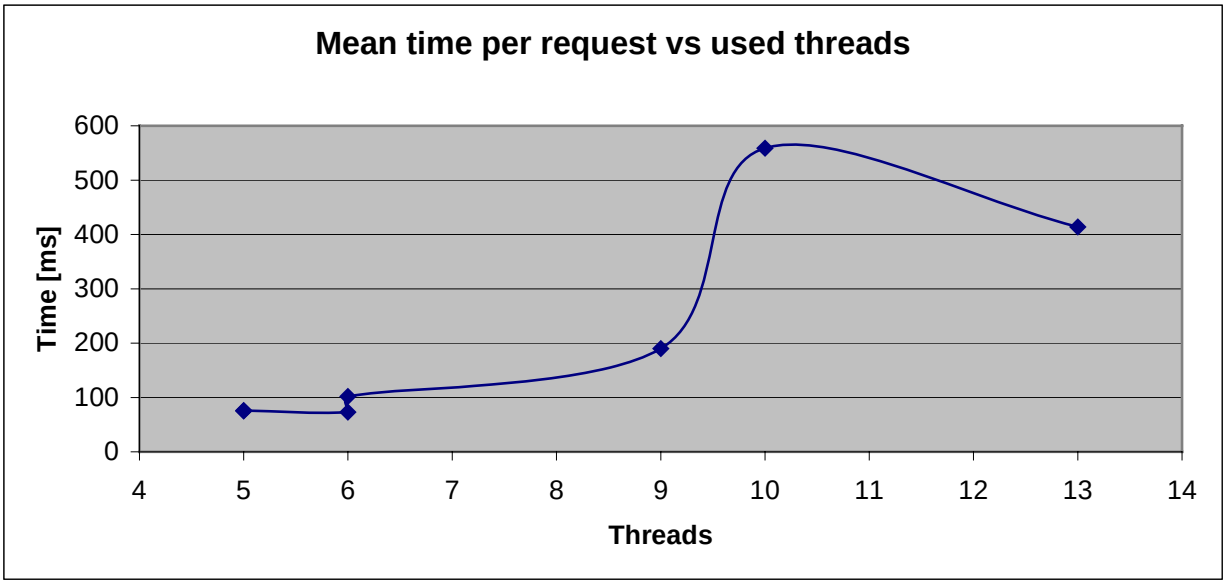
Reqs	Conc.	Pool size	MaxThreads	50%	66%	75%	80%	90%	95%	98%	100%	Total
100	1	1024	5	4	5	5	5	8	16	29	75	656
100	1	1024	5	4	4	4	4	5	10	14	76	550
100	2	1024	6	4	5	5	6	10	16	30	73	686
100	2	1024	6	8	10	13	14	19	21	76	102	579
100	20	1024	9	80	96	118	149	154	156	171	190	529
400	50	1024	10	274	283	288	293	444	492	531	559	2475
400	50	1024	13	260	264	267	270	319	331	399	414	2144
100	1	5	5	4	5	5	6	20	24	40	75	720
100	2	5	5	9	13	16	18	29	62	94	98	805
100	20	5	5	100	120	146	149	171	181	193	202	691
400	50	5	5	257	261	263	264	268	271	276	343	2133

The columns have the following meaning:

1. **Reqs** – total number of requests submitted to the server.
2. **Conc**. – number of multiple requests that were made
3. **Pool size** – maximum allowed number of threads to be created by the server
4. **MaxThreads** – actual maximum number of threads that the server created
5. **50%**…**100% -** indicate the percentage of requests that were completed in a certain time interval given in milliseconds
6. **Total –** the total time it took to complete all the requests

The rows marked in grey represent the data gathered from a test which has been conducted without prior restarting of the server – a so-called "hot" run. The primary difference between a "cold" and a "hot" run is that in the former case, the worker threads need to be started before the server can service the requests, while in the latter the threads are already available.





4.3. Conclusions

Two conclusions can be drawn from the diagrams above.

First of all, as can be seen from the first diagram, starting a new thread is a time-consuming process. That is why a test conducted on an already "warmed-up" server gives better results than on a freshly restarted one.

The second diagram presents the fact that there is a certain number of threads that leads to the fastest processing of incoming requests. This number obviously varies from system to system, as it depends on the system's memory and processor resources.

5. References

1. "The Java ME Platform", http://java.sun.com/javame/index.jsp

2. “Building Highly Scalable Servers with Java NIO”, http://www.onjava.com/pub/a/onjava/2004/09/01/nio.html
3. “Java theory and practice: Thread pools and work queues”, http://www-128.ibm.com/developerworks/library/j-jtp0730.html