

POOR MAN'S EBAY

Summary

We present two programs written in Java, a client and a server. These programs together implement a client-server protocol which allows people to sell and buy items.

The Protocol

The protocol uses TCP for the main client-server session and UDP for asynchronous message delivery from server to client. It is network-layer agnostic. It has been developed and tested on IPv4.

Main Session

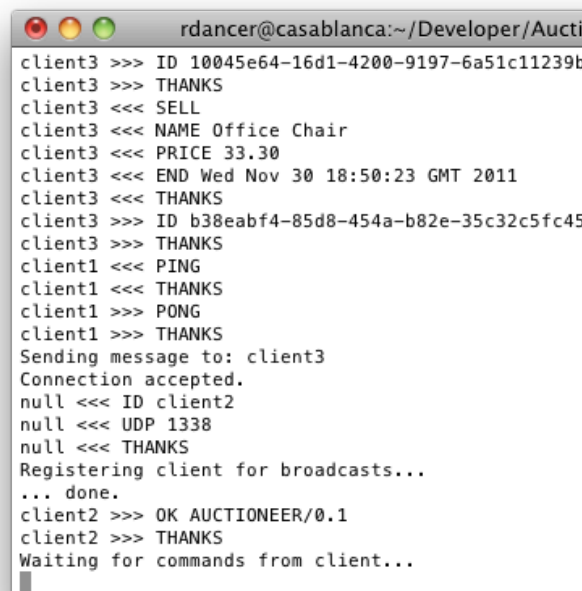
The main part of the protocol comprises of a sequence of client requests and associated server responses. There are. Each request type starts with a special keyword. Each message comprises of two or more lines. A line starts with a keyword, followed by zero or more arguments. The first line of a message specifies the request type. The eight client request types are: *HELLO*, *PING*, *BROWSE*, *SELL*, *BID*, *CANCEL*, *CONFIRM*, and *BYE*. The last line starts with the keyword *THANKS*. The client must start the session with a *HELLO* request. The session ends when the server receives a *BYE* request. The session will also end when the TCP connection is reset or times out.

Asynchronous Messages

Some of the information the server shares with the client does not fit within the paradigm of a sequential request-response model. Some other protocols show many approaches are possible: The POP3 protocol uses polling, the IMAP protocol uses unsolicited server responses. We have chosen to use a separate communication channel and send unsolicited information via UDP. The message format is identical to the request format described above. There is only one message type: *ALERT*.

Course Requirements and Assessment Criteria

- The number of connections the server can handle is virtually unlimited, constrained by the hardware performance and the Java threading architecture.
- The server keeps in memory the list of auctions, both active and expired. An improved version of the server would use persistent storage such as an SQL database to store this data.



```
rdancer@casablanca:~/Developer/Aucti
client3 >>> ID 10045e64-16d1-4200-9197-6a51c11239b
client3 >>> THANKS
client3 <<< SELL
client3 <<< NAME Office Chair
client3 <<< PRICE 33.30
client3 <<< END Wed Nov 30 18:50:23 GMT 2011
client3 <<< THANKS
client3 >>> ID b38eabf4-85d8-454a-b82e-35c32c5fc45
client3 >>> THANKS
client1 <<< PING
client1 <<< THANKS
client1 >>> PONG
client1 >>> THANKS
Sending message to: client3
Connection accepted.
null <<< ID client2
null <<< UDP 1338
null <<< THANKS
Registering client for broadcasts...
... done.
client2 >>> OK AUCTIONEER/0.1
client2 >>> THANKS
Waiting for commands from client...
```

Fig. 1: The server running in a terminal

- Client places an item for sale using the *SELL* request. The server replies with the auction identifier or with an *ERROR* if listing the item was unsuccessful. This request corresponds to the **sell** command in the client program.
- When the client sends the *BROWSE* request, the server replies with a listing of all the items it tracks. This request corresponds to the **browse** command in the client program.
- The client can remove an item by sending a *CANCEL* message. The server sends back a random 128-bit token. When the server receives a *CONFIRM* message containing this token, the cancelation is performed. The server checks that the *CANCEL* message comes from the item's seller. However, the *CONFIRM* message does not need to be checked: Provided that the token is truly random, it cannot be forged. This feature of the protocol can be leveraged to extend the protocol to do other asynchronous commands in a similar manner. These requests is implemented by the **cancel** command in the client program.
- The server runs a separate thread which checks periodically for expired items and sends an asynchronous *ALERT* message to both the seller and the winning bidder (if any). The client program will display all alerts it receives from the server.

Limitations

While this protocol would support client authentication, we have not implemented it because we feel that the complex issues of privacy and security are beyond the scope of this project. An encryption layer such as SSL can be readily layered under this protocol to provide such features.

Setup and Use

Along with the source code, two compiled Java classes are provided: one each for the client and the server. Both need to be run with command-line arguments specifying the basic networking and protocol settings. Full IP connectivity between server and clients is necessary — in particular, the client must be able to establish a TCP connection to the server; and the server must be able to send UDP packets to the client. The programs will not attempt to traverse NAT or test for firewalls, as these complex issues are beyond the scope of this assignment.

Starting the Server

Run the server from the command line. You can specify the TCP port number the server should listen on:

```
java Server [PORT_NUMBER]
```

For example, to have the server listen on TCP port 1337 (the default), run the server like this:

```
java Server 1337
```

The server provides copious console output, printing all the exchanges it shares with clients, as well as other useful information.

Starting the Client

```
java Client <HOST> <PORT> <CLIENT_ID>
```

For example, to connect to TCP port 1337 on the local host, with an identifier “bidder2”, run as:

```
java Client localhost 1337 bidder2
```

Using the Client

The command-line interface provides integrated help facility the user can access by typing “**help**”. The verbosity level is turned down by default. To appreciate the client-server communication, the verbosity level can be increased by typing “**verbose**”.

Improvements

The protocol would benefit from authentication (so that server would establish that the client is who it says it is). This can be readily implemented by requiring the client to supply a shared secret or pass phrase during the session establishment phase, alongside its ID. The server would then check its database of identifier-secret pairs, and only allow the session to proceed if the secrets match. The question of key management arises: how do the client and the server agree on the shared secret in the first place? One of the options would be to use systems where a trusted notary (or a *certification authority*) brokers this transaction. We could use SSL as a security as well as authentication level.

Currently, the server keeps all the information about items and clients in memory. This is sub-optimal, as a server failure or simply a restart means all this information is lost. Persistent storage, for example an SQL database could be used to persist data across invocations.

The UDP messages sent from server to client will not be delivered if the client is behind a NAT. Further development would ensure that NAT traversal is implemented — *Universal Plug and Play* could be used.

Public Domain Source Code

No third-party source code used.

References and Bibliography

Postel, J: “*Internet Protocol*,” RFC 760, USC/Information Sciences Institute (1980)

Postel, J: “*Transmission Control Protocol*,” RFC 761, USC/Information Sciences Institute (1980)

Postel, J: “*User Datagram Protocol*,” RFC 768, USC/Information Sciences Institute (1980)

Myers, J & Rose, M: “*Post Office Protocol - Version 3*,” RFC 1939 (1996)

Bradner, S: “*Key words for use in RFCs to Indicate Requirement Levels*,” RFC 2119 (1997)

Gosling, J & Joy, B & Steele, G & Bracha, G: *The Java Language Specification 3/e*, Addison-Wesley (2005)

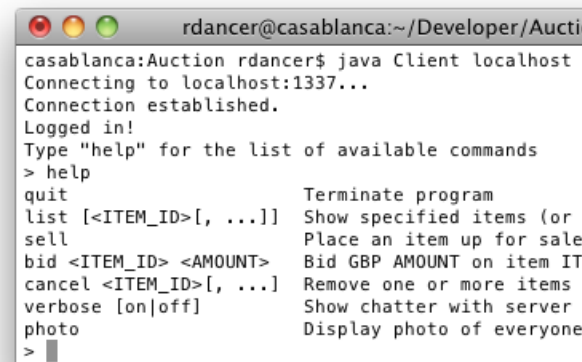


Fig. 2: The client waiting for input