”Protocol Polymorphism” - Using polymorphism to present different protocols to different clients in a chatserver

Philipp Hancke Carlo von Loesch

August 9, 2003

Abstract

In this article we will show how to build a chat server that is able to serve clients speaking various different protocols. Although talking about chat technology and protocols as an example, the technique proposed itself is not limited to these subjects.

We choose a design relying on polymorphism and dynamic binding. This means that we will overload protocol specific tasks such as parsing and output formatting of the objects that make up the link between client and server core.

We thus eventually arrive at a simple and highly modular design which yet can be amended in many ways.

Contents

1 Introduction and Motivation

1.1 A Short History of Chatting

In the early 1980s people communicated via mail (asynchronously), in Multi User Domains (MUDs), via BITNET, on Bulletin Board Systems (BBSs). talk, a UNIX tool which came along with 4.2 BSD in 1983 enabled a form of host-bound interpersonal communication, but the problem was that you had to talk to people while they were logged in on a specific host.

In 1988 IRC came along. First designed to be part of a BBS, it enabled you to log in from anywhere you wanted and still reach the same people identified by their nickname and the chatrooms they hang around. For a short period IRC was *the* system which allowed anyone to talk to anybody on the internet, but this didn't last very long. In 1990 this state of perfection was first destroyed by political reasons: the conservatives didn't like the "eris" policy of letting anyone on the internet have his own IRC server if he really wanted to, so by brute force divided the net and grabbed as many servers into their half as they could (the great IRC war). The first holigarchic IRC network resulted out of this move, dubbed EFNet as in "eris-free network." They also invented the name "anarchy net" for the remaining original IRC network, making it appear a less safe place to be. EFNet was later shaken by major scalability issues in the technology, which eventually led to further splits.

In 1993 the RFC 1459 [1] was written, which formalized and freezed the protocol. It had already shown IRC did not scale well and there were already people working on enhancements and alternatives, but none of these became accepted. Updated by RFCs 2810 - 2813 [URL1] in 2000 the old RFC 1459 is still more or less authorative.

Another year later the first webchats, browser based environments, appeared. Implemented as server side CGI-programs they used manual or HTTP refresh to get new content.

In 1996 yet another great split shattered the IRC community. Most of the european IRC servers split from the EFNet and formed what is now called IRCNet. Reasons where again more or less political, mainly concerning the power and responsibilities of IRC operators.

A year later a version of psycMUVE gave its debut at stern.de, probably the first webchat that used Netscape's server push technology [URL2], which is clearly superior to client-side refresh and polling.

Shortly after the creation of the Mozilla Project chief delevoper Jamie Zawinski wrote a promising paper, Unity of Interface [2] about modelling of different chat protocols under a common graphical interface. Despite of this, such plan was never implemented. Today Mozilla comes with a plain IRC client called chatzilla.

Having recognized that multiprotocol chat clients are complicated to design and even more difficult to maintain, Jeremie Miller created Jabber [3], an XML-based message routing protocol that aspired to be flexible enough to be the only protocol necessary for this task. Jabberd, the server software for this was released in 1999. The philosophy of Jabber is that complexity should remain at the server side. But curiously every Jabber client has to be able to parse XML. Have you ever felt the need to "talk" XML nodes via telnet? Jabber was also enhanced by two consecutive groupchat protocols, that mainly imitate IRCs groupchat model with one notable exception: Rooms are no longer distributed across a network but remain at one conference server.

1.2 The Current Situation

Today the world of online chatting is split into many incompatible systems, each of which uses their own clients and servers. This retards communication between people and that is what chat is all about. There are three ways how chatting could be made simple again:

- agree on a common protocol This approach is used by the Jabber foundation, that uses Jabber. The main problem is not a technical one but rather a social one, as stated on the ESW wiki [URL3]:

  > The JabberChickenEgg is a problem in technology migration, where lethargy, comfort and 'if it isn't (really) broke don't fix it' keep us trapped using broken-ish, ageing technology: we aspire to use Jabber, but stick to using IRC.

  Or in other words, it is too much to ask from users to throw away their access technologies, refined to their needs, they might have spent years with.

- extending chat clients to support all existent chat protocols This is clearly a bad approach, citing "Programming Jabber" [URL4, Preface page xi]

  > Of course, one obvious solution would be to build a single client that supported all of the IM system protocols, but this approach had two drawbacks:
  >
  > – The proprietary nature of the protocols made it harder to implement the support required and would make the client overly complicated.
  >
  > – Every time the protocol, which wasn't under his control, changed or a new one came along, the client would have to be modified - a task not practical for a large user base

- build servers that are able to speak whatever protocol a client wants them to

If we want to unite chatters all over the world, we must make sure that they will be able to continue using their accustomed clients. Therefore we will favour the last approach. As an example we will build a chatserver that features a single chatroom and provides interfaces for two different clients, namely IRC and telnet-based ones.

2 Polymorphism

The idea of polymorphism as an object oriented design principle is sometimes explained in terms of geometrical figures, e.g. having an abstract object "shape" and concrete subclasses like "square", "circle" or "triangle". The ability of each concrete shape to implement the specific knowledge of how to draw itself is called polymorphism. Deciding at runtime whose objects function is to be called is called dynamic binding.

But polymorphism is by far not limited to this simple example. During this article we will develop a similar model for network protocols, although we will stick with chat protocols. We will use a flexible protocol for modelling the internal communication (this is our "shape") and implement two protocols, namely the IRC protocol ("circle") and a telnet chat protocol ("square") according to their syntax definitions. These interfaces will transform a generic event (draw yourself) into a concrete action.

As it analyses the situation quite well we will quote the "Unity of Interface"-paper again.

It seems clear that chat systems like IRC and ICQ (and the AOL chat rooms, and, to some extent, certain MUDs/MOOs) are similar enough that one could come up with an interface to each of them that makes sense for each: they all incorporate line-at-a-time communication among (potentially) large numbers of people. They each have concepts of ”rooms” and ”private messages” and so forth.

The application of the concept of polymorphism to protocols is what we decided to call ”protocol polymorphism”. It makes a multiprotocol server project surprisingly easy to understand and manage.

2.1 psycMUVE

Despite of the simplicity of the approach, not too many software works use it. Even if they support multiple protocol they often do it in a unelegant non-oo way like writing external proxies. For example, many so-called webchats are based on IRC servers and use standard webservers in conjunction with server-side scripting for proxying. How ugly and inefficient.

The psycMUVE [URL5] is using exactly the approach that we describe in this article (actually this article is part of its technical documentation). It is designed for synchronous conferencing, which you may essentially want to call ”chatting”, but it's more than that, as it generically allows for multi-user messaging for all sorts of applications. No surprise even multiuser flash games exist which use PSYC messaging.

PSYC [4] itself has seen a very long conception span starting around 1989. Its textual representation may look like a classic RFC 822 text protocol turned somewhat inside out, but that's just to improve parsing efficiency. There are much deeper differences. First, there is a concept of method hierarchies as documented in this draft about Keyword naming in PSYC [URL6] Secondly, parameters are named and there is even the possibility that a variable assignment is persistent over a connection. For documentation on this take a look at Variable Modifiers for PSYC [URL7].

psycMUVE serves java applets, IRC, telnet and mud users, jabber clients (under development) and PSYC clients (in theory, as there are not many working psyc clients around). Yes even shockwave flash chat movies using an xml-ified version of PSYC and webchat (commercial version) are possible. This makes it in our eyes an impressive example of this design technique, but far too complex to discuss here.

2.2 Notes on LPC

psycMUVE is written in LPC, an old bytecode language once designed for writing MUDs, that has a notion of inheritance that is different from most other languages.

LPC has a most brilliant syntax for object orientation. No clobbering class wrappers for classes - each class is by definition in its own file. Calling methods is like calling functions in an other file - inheritance feels as easy and lightweight as ”including” an other file. There's no cumbersome necessity to define interfaces as in java, because LPC is capable of multiple inheritance (and i love it). All kinds of non-computer-science people have learned to code OO in LPMUDs within days, because it was all so logical. Wished modern scripting languages had an OO syntax at least half as brilliant as LPC's. Furthermore LPC is the only language which was designed for multiclient internet server applications in the first place, and being so old it is also one of the fastest doing the job.

2.3 Similar Software

twisted.words [5], part of the Twisted framework for writing network applications, is using a very similar design with twisted perspective broker [URL8] as an internal protocol, which is also exposed to the outside world, and also speaking IRC. The specification extracted from the source looks quite similar to what psycMUVE accomplishes, but at the time of this writing the module is effectively unmaintained. Despite of that it looks very promising for rapid development, as the twisted framework already supports parsing of a wide range of protocols. Also, their concept of resources is quite similar to what is done here, as these also provide a "Unity of Interface", but somehow they are limited to the webserver, twisted.web.

3 Building a Python Chatserver

Lets however investigate a simple but powerful example in Python, as this language has a very clean syntax and a powerful framework for asynchronous socket programming that we will need later. We will build a simple chatserver which supports telnet and IRC as an example. A telnet access port, as plain as it sounds, can in fact be very powerful when used with a MUD (multi user dungeon) client like powwow [URL9] or kmud [URL10] that extend the terminal emulation features of telnet with command line editing, history, aliases, text highlighting, automation and so much more. So in some cases a MUD client can be a more effective tool than an IRC client. But let's get back to our little demonstration. For simplicity there will be no real user management and authentification and only a single room. In other words, we are simply relaying messages between all incoming connections.

3.1 Protocols

When talking about chatservers we can not think in terms of the classical client-server request-response model but rather we need some kind of event model where a response can happen without the client requesting them. (Note: this is one of the reasons why http-webchats are so diffucult to implement as HTTP was built with request-response in mind and never designed for streaming and pushing of events) For our example we have a simple event model with only three types of events:

- _message_public: represents some text going to all people in the channel
- _notice_place_enter: is sent when someone enters the room (e.g. connects)
- _notice_place_leave: is sent when someone leaves the room (e.g. quits)

We will also need a unique identifier, commonly known as nick, for each client. For simplicity we choose a **host:port**-scheme Each of the protocols will present them in a different way that we will discuss now.

Telnet Protocol Representation

telnet (short: tn) will represent them as follows:

- _message_public: "[_nick] says: [_text]"
- _notice_place_enter: "[_nick] enters the chat"
- _notice_place_leave: "[_nick] leaves the chat"

Every line that is sent by the client and does not start with a "/" will be interpreted as a message, lines starting with a slash are commands (we will just implement /quit).

IRC Protocol Representation

- _message_public: "[_nick]![_nick]@myhost PRIVMSG #room :[_text]"
- _notice_place_enter: "[_nick]![_nick]@myhost JOIN :#room"
- _notice_place_leave: "[_nick]![_nick]@myhost PART #room :messages embedded in state changes are bad protocol design"

IRC-Clients should behave according to RFC 1459.

3.2 The Simplest Chatroom - a Multiplexer

One of the most important elements in building a chatserver is a multiplexer, a generic "thing" that simply distributes something it gets to all its clients:

```
class Multiplexer:
    def __init__(self):
        self.clients = {}

    def connect(self, client):
        self.clients[client.getsource()] = client

    def disconnect(self, client):
        if self.clients.has_key(client.getsource()):
            del self.clients[client.getsource()]

    def castmsg(self, data):
        for client in self.clients.values():
            client.render(data)
```

It has three functions each of which we will discuss now: **connect(self, client)** which takes as parameter some client (which we will discuss later) and adds it to the internal list of connected clients. **disconnect(self, client)** does the reverse of connect and deletes the client from the internal list. **castmsg(self, data)** will take an arbitrary data structure (we will have to talk about some kind of internal protocol before deciding how this looks like) and call the clients function **client.render** with it.

As you may notice, this multiplexer is a very generic form of a chat room and you may want to make it more intelligent if you extend the server beyond this example.

3.3 Asynchronous Sockets as an Interface to the Net

Now we have to go into network programming a little as we need some way to communicate with the outside world. We will favor asynchronous sockets, e.g. using non-blocking sockets and a select/poll-loop, over multithreaded programming. The reason for this was given by Sam Rushing in a tutorial for programming with pythons asyncore libraries [6] (which is also a good reference if you want to know more about python network programming)

> Why Asynchronous? There are only two ways to have a program on a single processor do 'more than one thing at a time'. Multi-threaded programming is the simplest and most popular way to do it, but there is another very different technique, that lets you have nearly all the advantages of multi-threading, without actually using multiple threads. It's really only practical if your program is I/O bound (I/O is the principle bottleneck). If

your program is CPU bound, then pre-emptive scheduled threads are probably what you really need. Network servers are rarely CPU-bound, however).

Using Java and spawning a thread for each client is extremly popular as you can see in the tutorial Building a Java chat server [7].

We will need an asynchronous socket server that spawns instances of the protocol classes to interpret and render the respective syntax. It will be simple, just creating an instance for each incoming connection. Indeed it is so generic that we can use it for all protocol classes.

The asyncore-classes from the python standard library provide a very convenient interface to asynchronous sockets and even a class for async back-and-forth-conversations, async_chat, which does buffering of both input and output and that we will need later. We could as well have choosen twisted for this, which is even more powerful, but it has to be installed separately.

```
import asyncore, socket
class ProtocolFactory(asyncore.dispatcher):
    def __init__(self, (protocol, port), multiplexer):
        asyncore.dispatcher.__init__(self)

        self.protocol = protocol
        self.multiplexer = multiplexer

        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.bind(("", port))
        self.listen(5)

    def handle_accept(self):
        # this is called by the asyncore library whenever a
        # connection has been established
        conn, addr = self.accept()
        # spawn a client class
        self.protocol(conn, self.multiplexer)
```

The **__init__** function takes two arguments, the first is a tuple of what type of protocol it will spawn and on connection to which tcp port it will do so, the second is a callback to the multiplexer. It binds to a tcp port and starts listening for connections.

Two short notes for people not familiar with python syntax:
the **__init__** *function of a class is called whenever an object is created.*
The **self***-parameter is mandatory as first argument in each class method as it is basically a pointer to the object itself, much like the this-pointer in C++.*

handle_accept(self) *will be called whenever a connection to the port has been made. It will then create an appropiate protocol class and hand over the connection.*

3.4 The Protocol Classes

Now comes the third and most elaborate part of the code: the protocol classes. What will they do?

- protocol conformant parsing of the input
- protocol conformant representation of the output

3.5 Basic Protocol Interface

Let's make a dumb protocol interface that does none of these very well. Here we will discuss the gory details of the implementation and how to use the asyncore framework.

```
import asynchat
class Protocol(asynchat.async_chat):
    def __init__(self, connection, multiplexer):
        asynchat.async_chat.__init__(self, connection)
        # we are doing input buffering
        self.buffer = ""
        # we have a newline-terminated protocol
        self.set_terminator("\n")

        (peerhost, peerport) = connection.getpeername()
        # some kind of identification
        self.source = peerhost + ":" + peerport.__str__()

        self.multiplexer = multiplexer
        self.multiplexer.connect(self)

    def handle_close(self):
        # called if the socket gets closed
        multiplexer.disconnect(self)
        self.close()
    def collect_incoming_data(self, data):
        # called whenever data arrives on the socket
        self.buffer = self.buffer + data
    def getsource(self):
        # little helper
        return self.source
    def found_terminator(self):
        # this will do the protocol parsing
        line = self.buffer
        self.buffer = ""
        self.multiplexer.castmsg(line)
    def render(self, data):
        # this will do output formatting soon
        # but now it simply pushes data over the socket as is
        self.push(data + "\n")
```

The constructor **__init__** was called when the ProtocolFactory created a protocol object on accepting a connection. What it does is basically taking control of that connection, defining what separates logical lines in the protocol by calling **set_terminator**, choosing some kind of identification based on the peer address (we will use this as a substite for the nick) and finally connecting to the multiplexer. **handle_close** will be called when the peer closes the connection unexpectedly, without quitting properly, and we want to disconnect it from our multiplexer. **found_terminator** is called whenever a complete logical line has been received, e.g. the terminator set in the constructor is found. For now it will simply distribute the line received via the multiplexer. **render** is the interface we used earlier in the multiplexer. It simply pushes the line it receives over the socket and appends a newline now.

3.6 Applying Polymorphism

As we have seen, there are two important functions:

- found_terminator(), which gets called by the asyncore libraries event loop whenever the protocol terminator (as defined in __init__ with self.set_terminator) is found. This should initiate the protocol parsing, but at the moment we simply cast a line to the multiplexer.
- render(), which should do the output formatting. As found_terminator just casts a line we will have not much to do now, so we simply push the data over the socket and append a newline character.

Before we can make these functions more elaborate, we need to define some kind of internal protocol. What we need to pass on is:

- the type of the event (someone's joining or leaving, someone's saying something)
- who generated the event (the source)
- some variables like source's nick, what is being said and so on

For our example we will use a simplified form of psycMUVE's PSYC protocol [URL11]. We will pass around tuples of the form *(event type, dictonary of variables)*

As strictly speaking the source is also a variable we will put it into the dictionary. As an example, for a *_message_public* event we would need a tuple that looks like

```
("_message_public", vars = {"_nick" : "somenick",
 "_text" : "arbitrary message string"} )
```

3.7 Refining the Telnet Protocol Interface

So, now it becomes clear what we have to do: parse incoming data according to protocol specification and setting the event type This is the job of found_terminator. For telnet clients this is easy. As we have agreed upon above, everything not starting with a "/" will be interpreted as a message. So the refined protocol class for tn looks like

```
import asynchat

class Protocol(asynchat.async_chat):
    def __init__(self, connection, multiplexer):
        asynchat.async_chat.__init__(self, connection)
        # we are doing input buffering
        self.buffer = ""
        # we have a newline-terminated protocol
        self.set_terminator("\n")

        (peerhost, peerport) = connection.getpeername()
        # some kind of identification
        self.source = peerhost + ":" + peerport.__str__()

        self.multiplexer = multiplexer

        self.multiplexer.connect(self)
        self.on_connect()
        self.multiplexer.castmsg(("_notice_place_enter",
                                  {"_nick" : self.source }))
    def handle_close(self):
        # called if the socket gets closed
        self.multiplexer.disconnect(self)
        self.multiplexer.castmsg(("_notice_place_leave",
                                  {"_nick" : self.source}))
        self.close()
    def collect_incoming_data(self, data):
        # called whenever data arrives on the socket
        self.buffer = self.buffer + data
    def getsource(self):
        # little helper
        return self.source
    def on_connect(self):
        pass
    def found_terminator(self):
        pass
    def render(self, (event, vars)):
        pass


class TelnetProtocol(Protocol):
    def on_connect(self):
        self.push("You␣are␣now␣known␣as␣" + self.source + "\n")
    def found_terminator(self):
        # this will do the protocol parsing
        line = self.buffer
        self.buffer = ""
        if line.startswith("/quit"):
            self.multiplexer.disconnect(self)
            self.multiplexer.castmsg(("_notice_place_leave",
                                      {"_nick" : self.source}))
            self.close_when_done()
            return
        self.multiplexer.castmsg(("_message_public",
                                  {"_nick" : self.source,
                                   "_text" : line }))

    def render(self, (event, vars)):
        line = ""
        if event == "_message_public":
            line += vars["_nick"] + "␣says:␣" + vars["_text"]
        elif event == "_notice_place_enter":
            line += vars["_nick"] + "␣enters␣the␣world"
        elif event == "_notice_place_leave":
            line += vars["_nick"] + "␣leaves␣the␣world"
        if line:
            self.push(line + "\n")
```

As you may see, some things have changed which we will discuss now. The constructor now generates a *_notice_place_enter*-event when the peer connects, also there is a **on_connect** function which became necessary for the IRC-interface. It does nothing important here, just telling the other side what its name is. The same way, handle_close will generate a *_notice_place_leave* event when a peer closes the connection unexpectedly.

> If we would implement message hierarchies we could even generate a *_notice_place_leave_logout* that each protocol implementation could handle different from *_notice_place_leave*. For example we could format it as *connection to [_nick] interrupted*. In PSYC, clients can handle both events in the same way but should display the different formatting. This is one of its major features, separating events and their representation.

Also the **found_terminator** and **render** have become far more complex.

found_terminator now handles the "/quit"-command and takes the appropiate action, in this case casting a *_notice_place_leave*-event to the multiplexer, disconnecting from it and finally closing the connection. As we agreed upon above, every other line will be treated as a public message, and so the interface will generate a *_message_public*-event in that case.

render no longer simply pushes the data over the socket. Now it does the formatting according to the protocol specification.

3.8 The IRC Protocol Interface

Lets look at the IRC Protocol class. They will inherit the base protocol interface and override the found_terminator and render methods.

To say something to the public, IRC clients will send a "PRIVMSG target :message" line. We may for now ignore target, at least if it starts with a "#", the irc-prefix for rooms (PRIVMSG command is used both for private and public messages in IRC, which shows how kludgy IRC is). more precisely if it starts with a "#" and then the name of our room follows (well... we don't have a roomname, so we simply ignore it). We will be ignoring JOIN the noisy way and interpret PART as a QUIT.

```
import Protocol
class IRCProtocol(Protocol.Protocol):
    def on_connect(self):
        # we have to tell the client its nick so it will
        # recognize when we join him into #room
        self.push(":myhost 001 " + self.source + " :Welcome to this strange example\n")
    def found_terminator(self):
        line = self.buffer.strip()
        self.buffer = ""
        lsplit = line.split(None, 1)
        if len(lsplit) > 0:
            cmd = lsplit[0]
            if len(lsplit) > 1:
                target = lsplit[1]
            if cmd == "PRIVMSG":
                if target == '#room':
                    self.multiplexer.castmsg(("_message_public",
                                             {"_nick" : self.source,
                                              "_text" : line.split(":", 1)[1] }))
                else: self.push("ERROR : sorry, just that single room target supported\n")
            elif cmd == "PART" or cmd == "QUIT":
                self.multiplexer.disconnect(self)
                self.multiplexer.castmsg(("_notice_place_leave",
                                          {"_nick" : self.source}))
                self.close()
            else:
                # unsupported command
                self.push("ERROR: unimplemented " + cmd + " \n")
    def render(self, (event, vars)):
        line = ""
        if event == "_message_public" and vars["_nick"] != self.source:
            # echo is not yet common in irc
            line = ":" + vars["_nick"] + "!" + vars["_nick"]
            line +=      "@myhost" + " PRIVMSG #room :" + vars["_text"]
        elif event == "_notice_place_enter":
            line = ":" + vars["_nick"] + "!" + vars["_nick"]
            line +=      "@myhost" + " JOIN #room"
        elif event == "_notice_place_leave":
            line = ":" + vars["_nick"] + "!" + vars["_nick"]
            line +=      "@myhost" + " PART #room :"
            line += "messages embedded in state changes are bad protocol design"
        if line: self.push(line + "\n")
```

As noted earlier, *on_connect* becomes important here. If we would leave it out, IRC-Clients would not recognize that it is them joining #room in there following *_notice_place_enter*-event. Likewise, *found_terminator* and *render* have changed what we will again discuss now.

found_terminator is now a very simple IRC parser. It splits up the first word from the line received, which is to be interpreted as the command. Based on what this command is it will decide which event to generate. **render** does again formatting like the tn-variant, but there is a notable difference. Echoing of the own messages is not common in IRC, rather all IRC-Clients handle displaying what the user himself said directly. So we do not deliver the message in this case as it would presumably be displayed twice.

Having overridden the behaviour of these two functions, we have finished implementing IRC (as far as we agreed upon what we call IRC)

3.9 Putting it all together

At last we have to put all things together and make them run...

```
from Multiplexer import Multiplexer
from ProtocolFactory import ProtocolFactory
import asyncore

from Protocol import TelnetProtocol
from IRCProtocol import IRCProtocol


if __name__ == "__main__":
    print "setting up multiplexer"
    m = Multiplexer()
    print "adding telnet protocol factory"
    basefactory = ProtocolFactory((TelnetProtocol, 2000), m)
    print "adding IRC protocol factory"
    ircfactory = ProtocolFactory((IRCProtocol, 6667), m)
    print "running..."
    asyncore.loop()
```

The code imports all the necessary classes that we have developed so far. Then, if it is called directly via **python ved.py** it will create a multiplexer instance, and two factories for protocol interfaces that will listen on the network ports.

This completes startup and the server enters the asynchronous event loop.

4 Final Words

So even if this was just a small prototype for a communication server we are using this type of technology every day as a basic communications tool, just in a lot more evolved form.

At the time of writing Carlo likes to use an enhanced telnet interface to handle all realtime communications, called powwow. Whereas Philipp likes to use a graphical IRC client called xchat. But even now we are also using small perl scripts to gateway incoming e-mail headers immediately into the PSYC world, instead of periodically checking for mail.

So one day we might be using totally different access technologies, like for instance a PSYC messaging tool for our mobile phones. That would be more efficient, more private and egregiously more flexible than the currently upcoming ICQ clients, and immensely cheaper than SMS.

This code was specifically designed to serve as a showcase implementation of what we are talking about. Therefore there is nothing that deals with error handling or other "technical" necessities of writing any non-trivial software, as well as there is no sign of any advanced features.

4.1 Outlook

Since the email system is currently still the most popular short messaging system on the internet, in abscence of a real common standard, it could have been interesting to examine the full integration of SMTP into such a messaging server. This would have bloated our example with a lot of SMTP protocol details as well as being semantically difficult: for example how often should we send out an email log of what has been said? Also, messages received by email would most likely be larger than the 512-byte limit of a normal IRC line, so it would have been necessary to implement some splitting, or to impose some usage rules for the email gateway. Just think of the SMS to mail gateways which only use the Subject header of SMTP for content. What an abuse of SMTP that is! How trivial in PSYC it would be!

If anyone would like to extend the example code, we suggest taking a look at the twisted framework to replace asyncore as it has a lot of classes that implement common protocols and would therefore ease this part of the work. The semantic part of plugging in different protocols, deciding how to present an event is already hard enough. If you are instead interested in working with a seasoned tool which is feature-rich and stable, install yourself a psycMUVE. You will discover that adding your own things to it is easier than you would expect, even if it means getting used to something that looks vaguely like C, but is actually much easier. And it has an unbeatable advantage, it can communicate with other entities worldwide, better than joining an IRC network, so you're not doing a stand-alone thing.

4.2 Conclusion

Let's think of ourselves as a shop selling messaging. We're giving it away for free in fact, but anyhow, to maximize the comfort of our customers' shopping experience we offer as many ways for them to shop with us as we can. Our clients are our customers, our servers are our service. It's so obvious, why are we so new with this approach?

4.3 Feedback

Feedback is always welcome, point your favorite IRC client to ve.symlynX.com 6667, telnet the same host (port 23), use the webchat or java client at http://BRAIN.symlynX.com/ and go for the PSYC room ("/join psyc" will work under most circumstances) where you will normally find some PSYC developers.

References

[1] http://www.irchelp.org/irchelp/rfc/rfc.html

[2] http://www.mozilla.org/unity-of-interface.html

[3] http://www.jabber.org/about/overview.html

[4] http://psyc.pages.de/

[5] http://www.twistedmatrix.com/products/words

[6] http://www.nightmare.com/medusa/programming.html

[7] http://www-106.ibm.com/developerworks/java/edu/j-dw-javachat-i.html

[URL1] http://www.irchelp.org/irchelp/rfc/

[URL2] http://www.netscape.com/assist/net_sites/pushpull.html

[URL3] http://esw.w3.org/topic/JabberChickenEgg

[URL4] http://www.oreilly.com/catalog/jabber/

[URL5] http://muve.pages.de

[URL6] http://psyc.pages.de/keyword-naming.html

[URL7] http://psyc.pages.de/modifiers.html

[URL8] http://twistedmatrix.com/products/spread

[URL9] http://brain.symlynX.com/unix/powwow-1.2.5.tar.gz

[URL10] http://www.kmud.de

[URL11] http://psyc.pages.de/psyc.html