

# NetChat Modular Communications System

## TJHSST Computer Systems Lab 2006-2007

Barnett Trzcinski

Steven Fuqua

Andy Street

March 30, 2007

### **Abstract**

NetChat revolutionizes the mundane system of communications of data, be they simple chat messages and e-mail to the transfer of mission-critical data. A modular framework ensures that nearly every type of communication can be tunneled through the same network, the same systems, and the same software. Not only does this simplify the experience for the end-user, but it accelerates development of new methods of communication and enables innovators to deliver these methods to the public faster than ever before.

## **1 Introduction**

Ever since the advent of the Internet, countless innovations have been made in the area of communications. As new ideas are constantly researched and explored, people discover new ways to communicate that they have never seen before. Unfortunately, as these ideas are disseminated, their organization becomes looser and looser. For example, mail, web pages, and news are all communicated using entirely different protocols, and in some cases (as with instant messaging) there are several competing protocols, each improving on the last's inefficiencies and issues.

Although this was necessary in the past when global socket communication via TCP/IP was a new idea, with more exploration into the encapsulation of a variety of data these numerous protocols become unnecessary.

In particular, all of the aforementioned communications utilities (mail, web pages, news, and instant messaging) all primarily involve the exchanging of simple data (text, images, files), but in a different manner. In essence, lots of debugging effort goes into improving and working on each of these separate protocols, when in the end they all accomplish the same end result. This is a waste of developers' time as well as users' patience, as many times arbitrary incompatibilities can surface through testing and simple end-user interaction with the services.

Rather than continue to develop separate protocols, there is a simpler solution. With the coming of Extensible Markup Language, or XML, the ability to represent diverse forms of data within one standard and using one parser is finally here. So far, the primary use of XML has been in the representation of stored, static data: configuration files, some web documents, and definitions. Other attempts to create unified communication protocols have resulted in cluttered definitions and unexpandable efforts for future uses [1]. Our aim is to create a new higher-level protocol, the NetChat Protocol (NCP), which uses XML as a base to represent numerous communications protocols. We are not trying to integrate diverse protocols into one system, as through middleware [2]; rather, we will re-implement existing communication systems to run through this same protocol, using rapid development techniques. Through this system, despite the different real applications of the data, it can all be transferred the same way, via the same routing, server network, and client.

## 2 Background

This project should not be confused with simple integration techniques. Large projects (such as AOL) and some small projects (such as web page portals) have tried to integrate diverse services such as mail, web, and news, and consolidate them into one interface. The primary problem is, however, left unresolved in these systems: the underlying communication for each service is quite different, and separate effort is required to maintain each function. NCP, on the other hand, is able to unify these forms of communication, thus simplifying the entire system and making the integrated approach normal functionality, not a special form of client.

## 3 Base Modular Framework

Before any meaningful work on end functionality could be achieved, the basic framework from which the functionality would later derive needed to be developed first. This involved defining a new standard for message passing using XML, called the NetChat Protocol, which handles all communication between server and client.

### 3.1 Defining NCP

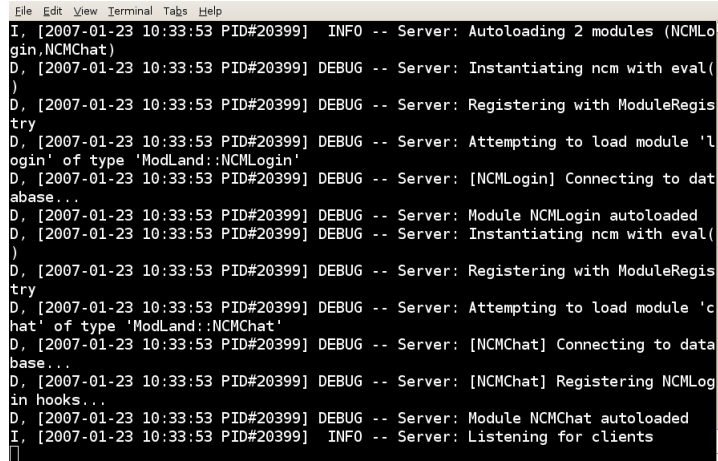
Using the Trac system implemented at [netchat.tjhsst.edu](http://netchat.tjhsst.edu), we were able to coordinate our efforts in developing this simple yet highly effective protocol for generic communication of data. Every message is composed of several XML tags. The root 'message' tag encompasses two others: 'header' and 'content'. The 'header' tag contains a 'global' tag, which might contain information such as specific properties that each NetChat Message specifies individually. Following the global information comes data specific to the type of message, such as tags defining the specifics of which Module to route the information to. Finally, the content tag can contain anything that a developer wishes to specify in his parser: there is no standard for content, and it is available for any data in any format that the developer specifies.

The server and clients communicate using these messages, divided into two supercategories known as 'server messages' and 'module messages'. Server messages deal with private communication between client and server, such as handshakes and stay-alive communication. Module messages are significantly more complicated, and are further subdivided based on modifying specific attributes of a given module. For example, the Chat module may receive module messages ranging from retrieving a buddy list to an alert that a message could not be sent.

### 3.2 Server Modules

The server is written in the highly dynamic Ruby language, which makes creating the modular framework for specific communications applications relatively easy. All that was needed was to define a generic base module, which utilizes the core framework, and create the documentation that users can use to write their own modules. Appendix A-1 shows the NCMBase

module's parse method, exposing the reflection and dynamic code there in interpreting messages.



```

File Edit View Terminal Tabs Help
I, [2007-01-23 10:33:53 PID#20399] INFO -- Server: Autoloading 2 modules (NCMLogin, NCMChat)
D, [2007-01-23 10:33:53 PID#20399] DEBUG -- Server: Instantiating ncm with eval(
)
D, [2007-01-23 10:33:53 PID#20399] DEBUG -- Server: Registering with ModuleRegistry
D, [2007-01-23 10:33:53 PID#20399] DEBUG -- Server: Attempting to load module 'login' of type 'ModLand::NCMLogin'
D, [2007-01-23 10:33:53 PID#20399] DEBUG -- Server: [NCMLogin] Connecting to database...
D, [2007-01-23 10:33:53 PID#20399] DEBUG -- Server: Module NCMLogin autoloaded
D, [2007-01-23 10:33:53 PID#20399] DEBUG -- Server: Instantiating ncm with eval(
)
D, [2007-01-23 10:33:53 PID#20399] DEBUG -- Server: Registering with ModuleRegistry
D, [2007-01-23 10:33:53 PID#20399] DEBUG -- Server: Attempting to load module 'chat' of type 'ModLand::NCMChat'
D, [2007-01-23 10:33:53 PID#20399] DEBUG -- Server: [NCMChat] Connecting to database...
D, [2007-01-23 10:33:53 PID#20399] DEBUG -- Server: [NCMChat] Registering NCMLogin hooks...
D, [2007-01-23 10:33:53 PID#20399] DEBUG -- Server: Module NCMChat autoloaded
I, [2007-01-23 10:33:53 PID#20399] INFO -- Server: Listening for clients

```

Essentially, the primary method of defining the behavior of the module is in writing several methods, whose names correspond to the type attribute in the header of each message. Each method is passed the client identifier which sent the message, the module-specific header, and the content of the message (if any). For example, to respond to a message type hello, the method `msg_hello` is implemented in the server module. How these methods are called is hidden from the module writer behind the scenes; at most, the writer can add a few pre-processing commands to the overall parse method which is responsible for using reflection to call the message methods before calling super and continuing the old method. Appendix A-2 shows how NCMChat defines the response to a particular type of message as well as a pre-authorization line added to the parse method.

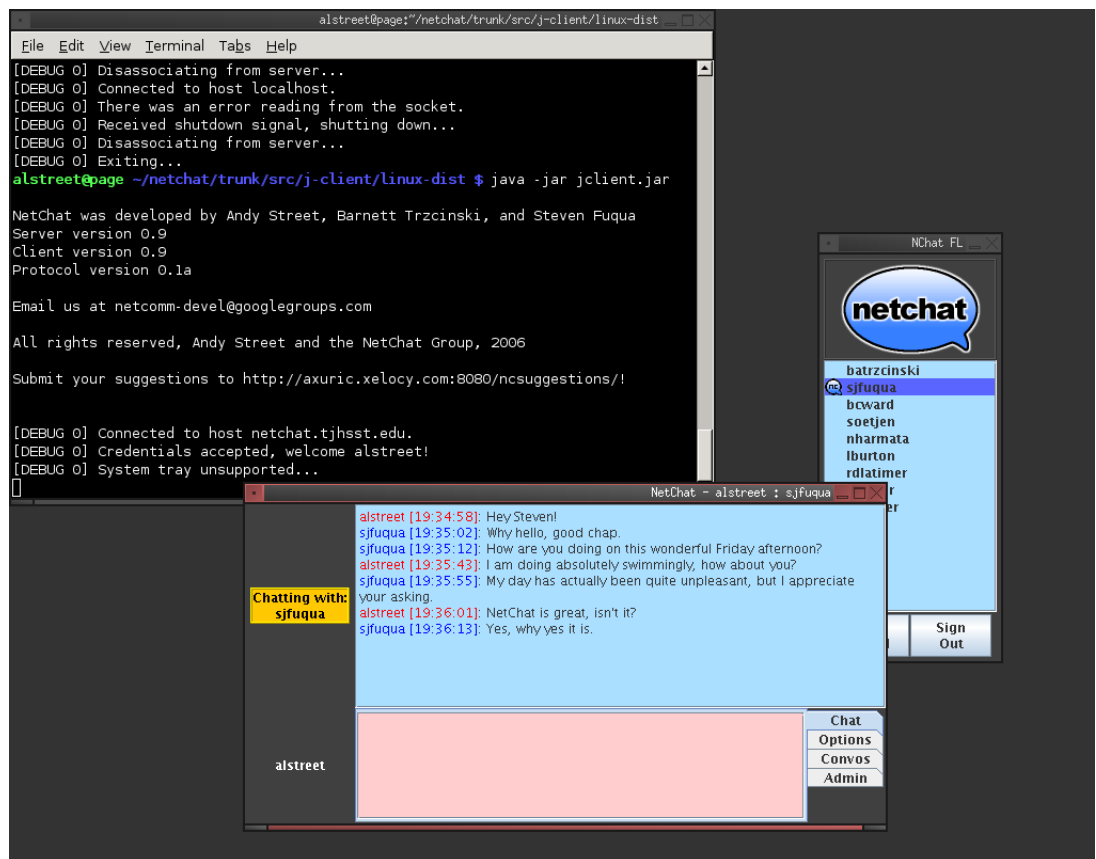
## 3.3 Client Modules

### 3.3.1 Graphical: Java

The J-Client, written in Java, is the main distribution client for NetChat. It provides a graphical interface written with the Eclipse Foundation's Standard Widget Toolkit (SWT), a portable widget toolkit that uses native back-end code to render native widgets for Java supported operating systems, and JFace, a data binding UI model framework, both of which allow this client to run on a variety of graphically-enabled systems. The J-Client uses a static parsing system to turn incoming NCP into easily accessible and organized

data objects. After getting passed a message from the socket by the Connector class, the XMLParser uses callbacks to construct an XMLData object, a standardized, tiered dictionary structure that is then passed to the Controller for routing. The Controller uses the routing header stored in the Data object to determine the type of message (module or server) and the module it should be routed to if any. If it is a server message, it is dealt with immediately, calling the proper method according to the data and passing a Tree version of the content tree. Otherwise, as a module message, it is passed to the correct module which handles it accordingly.

The main modular utility of the J-Client is the `java.lang.reflect` package, Java's implementation of reflections. Reflections are used to call server and module messages based on incoming NCP, as well as to dynamically load modules. Reflections are used to access fields, call methods and even dynamically instantiate Objects based on Strings. Once a message has been routed to the correct location, i.e. the correct module, the handling method will look at the `message_type` attribute of the Data object, append an identifying String to it, and then use the reflections package to call the method by that name, passing a content tree. Reflections are also used to create and load modules: once a module is authorized, the Controller will look up the corresponding class name and dynamically create an instance of the class. See Appendix B for code samples.



### 3.3.2 Console: Python

The console client is written in the Python scripting language, which allows for ease of expansion and flexibility of coding. Once a basic system for creating and loading modules was established, it became simple to create and load new ones.

```
-Output Window-
No backlog.
The following modules are active:
- base (7 commands/aliases)
- chat (9 commands/aliases)
- modularity (14 commands/aliases)
- screen (2 commands/aliases)
--> [batrczinski] hey
NCChat Warning: Your buddy is not online, but they will receive your messages when they
sign on.
Affirmative.
--> [batrczinski] hey?
NCChat Warning: Your buddy is not online, but they will receive your messages when they
sign on.
--> [batrczinski] where are you :(
NCChat Warning: Your buddy is not online, but they will receive your messages when they
sign on.

-Log Window-
2007/03/28 14:15 -0400 [commander] *Registering command 'in'.
2007/03/28 14:15 -0400 [commander] *Registering command 'friend'.
2007/03/28 14:15 -0400 [commander] *Registering command 'msg'.
2007/03/28 14:15 -0400 [commander] *Registering command 'fl'.
2007/03/28 14:15 -0400 [commander] *Registering command 'instantmessage'.
2007/03/28 14:15 -0400 [commander] *Registering command 'unfriend'.
2007/03/28 14:15 -0400 [commander] ...Successfully loaded command plugin.

NetChat Py-Client 1.0b                                     sifuqua@localhost
> in batrczinski nooo come online :([]
```

Upon receiving a Module Message from the server, the console client will begin analyzing the message itself. Once determining the appropriate module (for example, "login"), the parser will delve deeper and extract the specific type of login-related message ("accept\_login"). At this point, Python reflection is used to probe the Login Module class for a function called "accept\_login", which is called and passed the content of the message, which it can then parse independently. The code for this is shown in Appendix C-1. To define a new Module, one must only create a file "module\_name.py" in the appropriate directory, add the file to the configuration script as a default module. To allow for the module to load appropriately, a class within the file must exist that extends Module and has appropriate message hooks as specified by the NetChat Protocol. After the module is finished parsing content in its own unique manner, it is free to send response data to the server or to modify the client's internal state. The basic Module class is located in its entirety in Appendix C-2, with the Module parser in Appendix C-3.

## 4 Results and Discussion

The primary purpose has been accomplished. However, some changes should be made to make the product viable for widespread distribution.

First of all, the server right now is written in Ruby, a highly dynamic interpreted language. Although its performance is quite good, writing a streamlined, multithreaded server in C would vastly improve performance with large amounts of clients. Although some of the programmatic concepts would have to change (for example, some of the flexibility accomplished

through reflection), the functionality overall could remain intact quite well due to the high availability of XML parsers and required technology through C.

The Python client is not quite ready to be used as a release client. Due to its original intention of being a testing bed for new features, only recently has work commenced on establishing a flexible user interface through the curses terminal library. While sweeping progress is being made in developing the text-oriented Python client into a easily usable communications client, work remains to be done. It does, however, remain powerful in testing new features, as the simple interface and lightweight design aid in implementing innovative code as quickly and efficiently as possible.

## 5 Appendix A: Server Code Samples

### 5.1 Code Listing 1: NCMBase

[NCMBase.rb]

```
...
def parse (client, header, content)
  \ $log.debug "#{@name} instance has received data" unless \ $log.nil?

  # default behavior, route to a reflected method based on message type
  type = header.elements['properties'].attributes['type']
  response = self.__send__("msg_#{type}".to_sym, client, header, content)
  unless response.nil?
    self.communicator.send_message client, response[:header], response[:content]
  end
end
...
```

### 5.2 Code Listing 2: NCMChat

[NCMChat.rb]

```
...
```



```

# Handles the type 'backlog_request'.
# * Every backlog message is sent back to the client, sorted in the order
#   they were received, and cleared from the database.
# * An empty <content/> section is sent back if there are no backlogged messages.
def msg_backlog_request (client, header, content)
  m = make_skeleton_message
  response_header, response_content = m[:header], m[:content]
  response_properties = response_header.elements['properties']

  response_properties.attributes['type'] = 'backlog'
  username = self.moduleaccessor.access('login').get_username client
  q = @mysql.query("SELECT * FROM chat_backlog WHERE destination=
    '#{Mysql.quote(username)}' ORDER BY sent ASC")
  q.each_hash do |row|
    m = REXML::Element.new 'message'
    m.attributes['src'] = row['source']
    m.attributes['sent'] = row['sent']
    m.text = row['message']
    response_content.add m
  end
  q.free
  @mysql.query("DELETE FROM chat_backlog WHERE
    destination=#{Mysql.quote(username)}")

  return {:header => response_header, :content => response_content}
end

# Partially overridden to force an authentication check before processing _any_ mes
# The original functionality is kept assuming that check succeeds.
def parse (client, header, content)
  return nil unless checkauth(client) # prevents any nefarious message
                                       # handling if unauthorized

  # proceed with base functionality
  super(client, header, content)
end
...

```

## 6 Appendix B: Java Client Code Samples

### 6.1 Code Listing 1: Controller Method Reflections

[netchat.system.NCController.java]

```
...
/**
 * Routes an XML message to be handled by wither a module or a client.
 */
public void handleXMLData(NCXMLData d)
{
    HashMap data = d.h;
    String type = ((String)(data.get("type"))).toLowerCase();

    Debug.println("Handling XML message...", 2);

    if(type.equals("modulemessage"))
    {
        NCAbstractModule m;
        String name;

        if((m = modules.get((name = ((HashMap<String,String>)
            (data.get("properties"))).get("name")))) != null)
        {
            Debug.println("Handling " + type + " for module " + name + "...", 2);

            m.handle(d);
        }
        else
            Debug.println("Module " + name + " is not loaded.", 0);
    }
    else if(type.equals("servermessage"))
    {
        String msgType = ((HashMap<String,String>)
            (data.get("properties"))).get("type").toLowerCase();

        Debug.println("Handling " + type + " " + msgType, 2);
    }
}
```

```

        try
        {
            String method = "serverCommand_" + msgType;
            Method servMethod = NCController.class.getMethod(method,
                new Class[] {NCXMLElement.class});
            servMethod.invoke(this, new Object[] {d.getContent()});
        }
        catch (Exception e) { ... }
    }
}

```

## 6.2 Code Listing 2: Controller Module Loading

[netchat.system.NCController.java]

```

...
/**
 * Called upon receiving the server message 'authorize_module,' uses reflections
 * to create an instance of the module and loads it.
 */

//Called upon receiving "authorize_module"
public void serverCommand_authorize_module(NCXMLElement content)
{
    String modName = (String)(content.getElement("name").getSubObjects().get(0));

    if(modules.get(modName) != null)
    {
        Debug.println("Module " + modName + " already loaded, aborting...", 0);
        return;
    }

    boolean foundMod = false;
    for(int i = 0; i < modulesAwaitingAuth.size(); i++)
        if(modName.equals(modulesAwaitingAuth.get(i)))
        {
            foundMod = true;

```

```

        modulesAwaitingAuth.remove(i);
        break;
    }

    if(!foundMod)
    {
        Debug.println("ERROR: Module " + modName + " not awaiting
            authorization! Aborting...", 0);
        return;
    }

    Class modClass = moduleClassnameMap.get(modName);
    if(modClass == null)
    {
        Debug.println("ERROR: Module " + modName + " is not defined!
            Aborting...", 0);
        return;
    }

    Debug.println("Creating instance of " + modClass + "...", 2);

    try
    {
        Class cont = Class.forName("netchat.system.NCController");
        Constructor c = modClass.getConstructor(new Class[] {cont});
        NCAbstractModule mod = (NCAbstractModule)
            (c.newInstance(new Object[] {this}));

        Debug.println("Loading module " + modName + "...", 2);

        loadModule(mod);
    }
    catch (Exception e) { ... }
}

```

## 7 Appendix C: Python Client Code Samples

### 7.1 Code Listing 1: Login.accept\_login

[netclient.modules.login.py]

```
...
def accept_login(self, content):
    self.logger.log('Login accepted.')
    self.online = True
    tags = list(iter(content))
    try:
        uname = tags.pop(0)
        if tags:
            raise IndexError
    except IndexError:
        raise XMLError, 'Received NCLogin Message with wrong # of content subelements'

    if uname.tag != 'username':
        raise XMLError, 'Received NCLogin Message with invalid content tag.'

    self.username = uname.text
    mmanager.queue_modules(cmanager['config'].find('modules', 'default_modules') +
                           cmanager['terminal'].set_dialog('parser'))
...
```

### 7.2 Code Listing 2: Module

[netclient.extensibles.py]

```
...
class Module(Component):
    """
    A Module used by the netclient.mmanager.ModuleManager.
    """

    def __init__(self):
```

```

        self.opened = False

    def open(self):
        self.opened = True

    def close(self):
        self.opened = False

    def parse_content(self, cont, typ):
        """
        Given a Module command and the content of an NCP message,
        the Module will react in a specified manner.
        """

        return getattr(self, typ)(cont)

    def get_tabbed_list(self):
        """
        Called by the curses screen when 'tab' is hit, such that each
        module can implement it differently.
        """

        return []

...

```

### 7.3 Code Listing 3: ModuleManager.load\_module

[netclient.mmanager.py]

```

...
def load_module(self, name, q=False):
    d = self.queue if q else self.modules
    if name in d:
        return True
    module = dynamicLoad(self.path, name)
    if module is not None:
        reload(module)

```

```

        mclass = getattr(module, module.moduleclass, None)
        if not issubclass(mclass, Module):
            raise ModuleError, 'Modules must extend Module.'
        minstance = mclass()
        version = getattr(module, 'version', None)
        if not isinstance(version, str):
            raise ModuleError, 'Module %s does not have a valid version.' % name
        self.add_module(name, minstance, version, q)
        return True
    else:
        return False
...

```

## References

- [1] S. A. Moore, “A Communication Framework for Applications”, *Proceedings of the 28th Hawaii International Conference on System Sciences*, pp. 330-341, 1995.
- [2] S. A. Gutierrez-Nolasco and N. Venkatasubramanian, “A Composable Reflective Communication Framework”, *Proceedings of IFIP/ACM Workshop on Reflective Middleware 2000*, 2000.
- [3] DJ Adams, “Programming Jabber: Extending XML Messaging”, O’Reilly & Associates, 2002.
- [4] M. E. Fayad and D. C. Schmidt, “Object-Oriented Application Frameworks”, *Communications of the ACM 10*, Vol. 40, October 1997.
- [5] A. Denis, C. Pérez, and Thierry Priol, “PadicoTM: an open integration framework for communication middleware and runtimes”, *Future Generation Computer Systems 19*, pp. 575-585, 2003.