NetChat Modular Communications System TJHSST Computer Systems Lab 2006-2007

Barnett Trzcinski
Steven Fuqua
Andy Street

January 24, 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.

Keywords: Network, Module, Modular, Communications, Protocol, Server-Client

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 simplfying the entire system and making the integrated approach normal functionality, not a special form of client.

3 Base Modular Framework

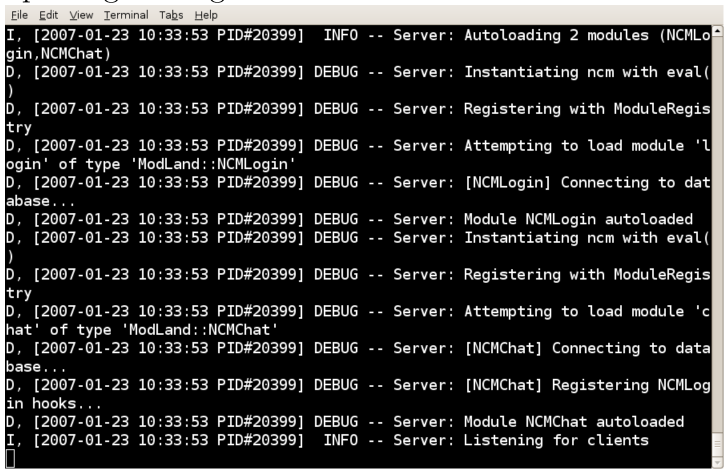
Before any meaningful work on end functionality could be achieved, the basic framework from which the functionality would later derive needed to be developed first. This involved defining a new standard for message passing using XML, called the NetChat Protocol (NCP), which handles all communication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

3.1 Defining NCP

Using the Trac system implemented at 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...

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.



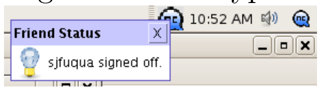
```
I, [2007-01-23 10:33:53 PID#20399]  INFO -- Server: Autoloading 2 modules (NCMLo
gin,NCMChat)
D, [2007-01-23 10:33:53 PID#20399] DEBUG -- Server: Instantiating ncm with eval(
)
D, [2007-01-23 10:33:53 PID#20399] DEBUG -- Server: Registering with ModuleRegis
try
D, [2007-01-23 10:33:53 PID#20399] DEBUG -- Server: Attempting to load module 'l
ogin' of type 'ModLand::NCMLogin'
D, [2007-01-23 10:33:53 PID#20399] DEBUG -- Server: [NCMLogin] Connecting to dat
abase...
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 ModuleRegis
try
D, [2007-01-23 10:33:53 PID#20399] DEBUG -- Server: Attempting to load module 'c
hat' of type 'ModLand::NCMChat'
D, [2007-01-23 10:33:53 PID#20399] DEBUG -- Server: [NCMChat] Connecting to data
base...
D, [2007-01-23 10:33:53 PID#20399] DEBUG -- Server: [NCMChat] Registering NCMLog
in 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

NCController is the main backbone for the J-Client framework in that it not only houses all the server commands and maps of loaded and unloaded modules, but also routes the majority of client traffic coming from the socket. Incoming messages, after being parsed by the NCXMLParser into a NCXMLData object, are then sent to be handled by the Controller, which ultimately decides what to do with the message. If a message is a module message, the Controller will find the module in the map of loaded modules, and route it to that module. If it is a server message, the Controller deals with it right there, using reflections to call the appropriate method for the message type NCAbstractModule is the class that all user created modules must implement. The AbstractModule class handles all message routing and reflection, allowing users to just implement a method with the name moduleCommand_[Module Message Command Type] and know that the method will be called and passed a content tree whenever a message with that module message command type is received.



sjfuqua signed off.

Implementing subclasses must implement:

1. A constructor that is passed an NCController
2. The cleanUp() method, called when the module is being unloaded

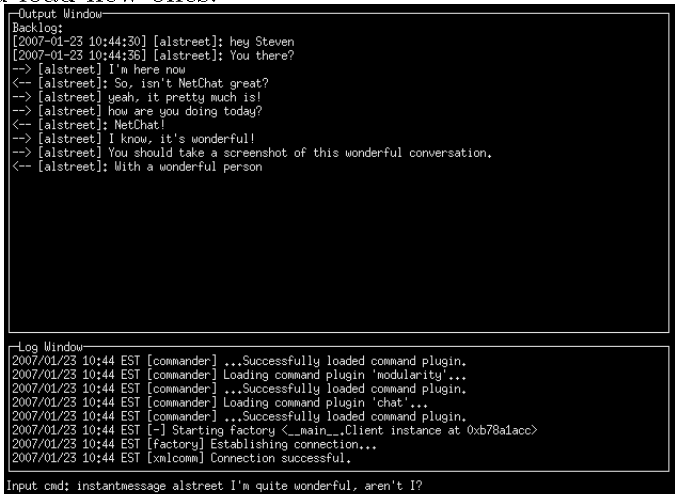
3. The getName() method, which returns the generic name of the module (i.e. for NCLoginModule, the String "login", or whatever is defined in etc/modules.conf)

4. The getProtocolVersion() method the returns the NetChat Protocol version number the module is using (currently 0.1a)

5. The getVersion() method that returns the version number of that particular module (used for versioning control and automatic updates.)

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
Backlog:
[2007-01-23 10:44:30] [alstreet]: hey Steven
[2007-01-23 10:44:36] [alstreet]: You there?
--> [alstreet] I'm here now
<-- [alstreet]: So, isn't NetChat great?
--> [alstreet] yeah, it pretty much is!
--> [alstreet] how are you doing today?
<-- [alstreet]: NetChat!
--> [alstreet] I know, it's wonderful!
--> [alstreet] You should take a screenshot of this wonderful conversation.
<-- [alstreet]: With a wonderful person

Log Window
2007/01/23 10:44 EST [commander] ...Successfully loaded command plugin.
2007/01/23 10:44 EST [commander] Loading command plugin 'modularity'...
2007/01/23 10:44 EST [commander] ...Successfully loaded command plugin.
2007/01/23 10:44 EST [commander] Loading command plugin 'chat'...
2007/01/23 10:44 EST [commander] ...Successfully loaded command plugin.
2007/01/23 10:44 EST [-] Starting factory <__main__.Client instance at 0xb78a1acc>
2007/01/23 10:44 EST [factory] Establishing connection...
2007/01/23 10:44 EST [xmlcomm] Connection successful.

Input cmd: instantmessage alstreet I'm quite wonderful, aren't I?
```

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 ("authorize_login"). At this point, Python reflection is used to probe the Login Module class for a function called "authorize_login", which is called and passed the content of the message, which it can then parse independently. 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 specific 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.

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, though powerful, is currently unsuitable for use as a release client. It was originally created to be a testing bed for new features, but the focus has since shifted as a comparison between coding paradigms: Java versus Python, GUI versus text. Experience thus far has shown that implementations of modularity and interface are remarkably similar between the two clients, though due to the inherent difference in interface, some functional differences exist. The nature of the curses interface library makes it difficult to create a perfect client, and additional emphasis needs to be placed on ease of use (for example, tab completion of usernames has been implemented).

5 Appendix A: Server Code Samples

5.0.3 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
...
```

5.0.4 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 overriden 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
...
```

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.