Fire Message Handling Architecture: A Proposal

Revision 1.0
February 18, 2003

Alan Humpherys
alangh@users.sourceforge.net
http://fire.sourceforge.net

Introduction

Message handling is the process by which the user messages to or from various Instant Messaging Services ("Services") are sent, received, processed, and displayed. This does not cover the other types of messages to or from Services (such as buddy list handling, file transfer, conferencing, or session management).

The current message handling architecture of Fire has grown in an ad-hoc fashion as new features and functions have been added to the processing stream over time. As a result, it is not possible to easily add new functionality to the message-handling stream or fix some of the existing bugs or deficiencies.

Current control of message handling is distributed among several objects in Fire including the various CommunicationControllers, ChatWindow, NSAttributedStringAdditions, and others.

This proposal outlines a new userMessage class that centralizes, standardizes and encapsulates the processing of user messages between the User Interface and the CommunicationControllers of the various Services.

CommunicationController Role

The role of the individual CommunicationControllers will remain very much as it is today. For user messages this object has the responsibility for properly receiving and sending the messages on the service transport, as well as aid in the processing of certain service specific portions of the message (such as font and attribute encoding or decoding). It is anticipated that the changes to the CommunicationControllers will be minimal, with the largest changes being in the movement of various message-processing steps into separate methods that will be called by the new UserMessage object.

Receiving Messages

When any message is received from the service, it is the duty of the CommunicationController and its various helper classes and libraries to decode the message and determine message type. Once a particular message has been found to be a user message, a userMessage object is created with the raw message contents payload (represented as a NSData object) used as the initialization parameter.

```
msg = [UserMessage initWithData:(NSData *)rawMessage
fromBuddy:(BuddyItem *)buddy forAccount:(Account *)account];
```

For group chat sessions, this would have an additional parameter indicating the chat room name for which it is intended.

```
msg = [UserMessage initWithData:(NSData *)rawMessage
fromBuddy:(BuddyItem *)buddy forAccount:(Account *)account
forChat:(NSString *)chatName];
```

Each of these initializers would also have a form containing a final timestamp: parameter that would allow the CommunicationController to assign a timestamp to the message. This would be used by those services allowing offline messages to be stored and delivered at a later date when the user comes online. Without this optional parameter, the current system time is assigned as the timestamp.

The CommunicationController then invokes the processInbound method on the userMessage object.

```
[msg processInbound];
```

This method then ensures that all proper steps are taken to process and display this message on the user interface.

Sending Messages

The role of the CommunicationController in sending a message will consist of implementing a method that takes a UserMessage object as a parameter and uses its access methods to extract the data from the UserMessage object and send it to the service in the proper packet form.

```
- (void)sendMessage:(UserMessage *)msg;
```

Message Processing

Certain aspects of processing a message are very dependent upon the service. Much of this processing is currently taking place at various stages in the handling of a message, unfortunately, in order for all types of messages to be sent and all functions performed, more strict control over the ordering of processing needs to take place.

Apart from the sendMessage method mentioned above, the CommunicationController also needs to implement several methods that will be used at various stages of processing the message. Many of these can be defined in the CommunicationController superclass, but some may have to be implemented in the subclass for some Service types.

```
// Get the default encoding type for messages on this
// service.  This may be overridden by end-user
// selections on the application or buddy level.
- (CFStringEncoding)defaultCharacterEncoding;

// Get the default smiley dictionary for this service
- (NSDictionary *)smileyMenuDictionary;

// Add or remove the user name from the end of the
// message (for ICQ/IRC?)
- (NSString *)extractUserName:(UserMessage *)msg;
- (NSString *)addUserName:(UserMessage *)msg;

// Convert the string to/from an attributed string
// according to the markup language for this service
- (NSAttributedString *)createAttributedString:(UserMessage *)msg;
- (NSString *)encodeAttributedString:(UserMessage *)msg;
```

```
// Parse Service Specific Control messages
- (BOOL) parseServiceControlMessages:(UserMessage *):msg;
```

All of the above should be fairly self-explanatory with the exception of parseServiceControlMessages. This method provides an opportunity for the CommunicationController to parse and handle messages that are commands, actions or controls for that service type. This is done at an appropriate time in the message handling so that other functions such as encryption, encoding, and translation do not interfere with these message types. Some examples of this type of function are the /xxx commands in IRC (e.g. /me does something) or the action commands ("emotes") of Yahoo Chat. If this message is found to contain a service control message or action, this method takes appropriate action and returns YES if further processing of this message is to continue. If NO is returned, no further action (or display) of this message will occur. A NO return would be used if the message were fully handled by this method including all appropriate end user display.

ChatWindow role

Currently, the ChatWindow methods contain most of the message processing code. It is proposed that this code be moved out of the ChatWindow object (mostly out of the various flavors of addAttributedMessageToDisplay), and into the UserMessage object.

The result would be a single method of:

```
- (void)addMessageToDisplay:(UserMessage *)msg;
```

This method would display the attributed message in its window and nothing more. All other processing currently done in the following methods would be moved into the UserMessage class:

```
- (void)addAttributedMessageToDisplay:(NSAttributedString
*)anAttributedMessage fromUser:(NSString *)userName;
- (void)addAttributedMessageToDisplay:(NSAttributedString
*)anAttributedMessage fromUser:(NSString *)userName
translateIncoming:(BOOL)needToTranslate;
- (void)addAttributedMessageToDisplay:(NSAttributedString
*)anAttributedMessage fromUser:(NSString *)userName withDate:(NSDate
*)aDate;
- (void)addAttributedMessageToDisplay:(NSAttributedString
*)anAttributedMessage fromUser:(NSString *)userName withDate:(NSDate
*)aDate translateIncoming:(BOOL)needToTranslate;
- (void)addAttributedActionToDisplay:(NSAttributedString
*)anAttributedMessage fromUser:(NSString *)userName;
- (void)addActionToDisplay:(NSString *)action fromUser:(NSString
*)userName;

- (void)addStatusMessageToDisplay:(NSString*)message
fromUser:(NSString*)who;
```

This change will allow the ChatWindow object to encapsulate the user interface and UserMessage to encapsulate the message data and processing.

UserMessage Role

The new UserMessage object will encapsulate all of the data and processing for a user message. As discussed above, this will consist of functionality currently in several objects throughout Fire. This will continue to be done in conjunction with other objects as described in their other sections, but with the processing done in a much more structured manner.

The following table shows the different steps in the message handling process for current Fire functionality. Other items (such as message forwarding, abbreviation expansion, etc.) may be added to this processing in the not to distant future:

Incoming Message Handling

Order	Function	Location	InputType	OutputType	Special
1	Message Transmission	Service	-	NSData	
2	Fire Binary message decapsulation	General+Svc	NSData	NSData	
3	Signing Validation	General	NSData	NSData	
4	Decryption	General	NSData	NSData	
5	Fire<->Fire Control msg parsing	General	NSData	NSData	may branch
6	Character Set Decoding	General+Svc	NSData	NSString	
7	User name extraction	Service	NSString	NSString	
8	Service Control Msg parsing	Service	NSString	NSString	may branch
9	Translation	General	NSString	NSString	creates 2 messages
10	Profanity Filter	General	NSString	NSString	
11	Speak Message	General	NSString	- (sound)	
12	Font/Atribute Decoding	Service	NSString	NSAttributedString	
13	Smiley parsing	General+Svc	NSAttributedString	NSAttributedString	
14	URL/Email detection markup	General	NSAttributedString	NSAttributedString	
15	Message Display	General	NSAttributedString	-	
16	History/Logging	General	NSAttributedString	UTF8 in file	
17	Notifications/Away Processing	General	-	-	may create outgoing

Outgoing Message Handling

Order	Function	Location	InputType	OutputType	Special
1	Font/Atribute Encoding	Service	NSAttributedString	NSString	
2	Translation	General	NSString	NSString	Creates 2 messages
3	Message Display	General	NSString	-	Do incoming steps 11-17
4	User name addition	Service	NSString	NSString	
5	Character Set Encoding	General+Svc	NSString	NSData	
6	Encryption	General	NSData	NSData	
7	Signing	General	NSData	NSData	

8	Fire Binary message encapsulation	General+Svc	NSData	NSData
9	Message Transmission	Service	NSData	-

As demonstrated by the preceding table, the functions performed at a service specific level are spread throughout the processing stream, thus the encapsulation of control of this processing should be moved to an external object, rather than contained within CommunicationController or ChatWindow objects.

In the above data flow model, any message, for any service can have all optional functions applied to it. In today's model, things such as signing and encryption are exclusive to other functions such as translation and character encoding, and may not work on particular services. Also today's model does not lend itself to things such as having non-blocking mechanisms to handle processing steps that are time consuming (such as translation).

Impact on Other Objects

It is anticipated that the impact upon other objects will be minimal. Some methods (or functionality) may be moved into the UserMessage object, and the translation currently in the NSAttributedStringAdditions will probably move as well.

Appendix:

UserMessage.h header file (partial listing)

```
@interface UserMessage : NSObject
{
      NSData                  *rawMessage;
      NSString                *msgString;
      NSAttributedString      *attributedString;
      Account                 *account;
      BuddyItem               *buddy;
      NSString                *chatName;
      time_t                  timestamp;
      … other instance variables for state
}

// Convenience Initialization methods
+ (UserMessage)initWithData:(NSData *)rawMessage fromBuddy:(BuddyItem
*)buddy forAccount:(Account *)account;
+ (UserMessage)initWithData:(NSData *)rawMessage fromBuddy:(BuddyItem
*)buddy forAccount:(Account *)account timestamp:(time_t)time;

// Processing action methods
- (void)processInbound;
- (void)processOutbound;

// Access Methods
- (NSData *)getMessageData;
- (NSAttributedString *)getAttributedMessage;
- (NSString *)getMsgString;
- (time_t)getTimeStamp;
- (Account *)getAccount;
- (Buddy *)getBuddy;
- (NSString *)getChatName;

// "Setter" equivalents of the above will be
// implemented as needed, for example:
- (void)setAccount:(Account *)newacct;

// State Testing Methods
- (BOOL)isGroupChat;
- (BOOL)isOutbound;

@end
```