XJCP – Specification

XJCP = Extensible JSON Chat Protocol

Note: One day SXJCP (SecureXJCP) will be specified , which will extend and replace XJCP.

Content

Communication Channel

The communication must be done over HTTPS. A socketconnection is optional and very uncommon. All clients and servers must support https.

You may send data to the server as POST or GET variable „msg“ for the normal protocol or „m“ for the minified protocol. When both variables are declared the normal protocol will be prefered and the minified protocol ignored.

The JSON-Object should be encoded using the Javascript escape method. After receiving the message must be unescaped by the following method and used as a JSON-Object.

```
function unescapeJS(str) {
      return preg_replace_callback( "(\\\\x([0-9a-f]{2}))i", function (a) {
            return chr(hexdec(a[1]));
      }, str );
}
```

The servers have to encode texts using:

```
htmlspecialchars(str);
```

Therfore clients must support HTML amp-tags. However clients can rely on the assumption that a message does not contain html code.

JSON-Objects

The minified protocol works the same but the identifiers are different. Look at the overview.

Client-To-Server

The client sends json objects to the server in order to request information. The request for events is inherited in every message and does not need explicit requesting.

A request should only contain login information if the client does not posess a valid id to use.

This results in the following skeleton for JSON-Objects.

{id:123, ...}

or when no id is availible yet:

```
{user:“foo“, pw:“bar“, ...}
```

As a client you are allowed to set variables in the json object that you do not request, however they must be set null.

Server-To-Client

The server sends back json objects depending on the request. Some servers send back all results and put null where nothing was requested while others only set those results that were requested. The client needs to be capable of both.

A typical answer on a send message request could look like this:

```
{id:123, events:[], onMessage:“Success“}
```

But maybe also like this:

```
{id:123, events:[], onMessage:“Success“, onError:null,
onData:null,...}
```

In-Out Types Overview

The following table shows which input attributes you can set and what objects are required as well as what output attribute the server will set and what objects that attribute will contain.

Mini is the identifier used in the minified protocol. The minified protocol may only be used on mobile devices.

Mini	Input-Attribute	Output-Attribute	Input Object	Output Object
i	id	id	String	String
u, p	user, pw	user, pw	String	String
e	-	events	-	[Event]
c	getChats	onGetChats	{}	[ChatObject]
k	getContacts	onGetContacts	{}	[ContactObject]
h	getHistory	onGetHistory	[{conversation, count}]	[History]
m	message	onMessage	[{conversation, message}]	[String]
d	removeEvent	-	[{conversation}]	-
n	newConversation	onNewConversation	[{conversation}]	[String]
r	renameConversation	onRenameConversation	[{conversation, name}]	[String]
sN	setName	-	String	-
aF	addFriend	onAddFriend	[String]	[String]
sS	setStatus	-	Integer	-
sP	setProfileImage	onSetProfileImage	{}	String
sG	setGroupImage	onSetGroupImage	{conversation}	String
iE	injectEvent	onInjectEvent	[{conversation, type, message}]	[String]
da	data	onData	{conversation}	String

{} means that this must be not null
[] means an array of the object inside. When there is an array on the input object and output object this is for batch mode. Further information in the section about batch processing.

The id is linked to a device and is globaly unique. It should be bound to an IP address for security.

User and pw are only required, when the id is not accepted. They are used for login.

Events can be any type of events. (There should be onMessage events for messages a user receive.)

I/O-Details

Id

The id must be passed in every request in order to identify and authentificate the user.

user,pw

The user and pw must be set in requests when there is no valid id known to the client. It is used to login.

Events

This is an implicit request. Every result contains a list of events that occured since the last request.

GetChats

Get the chats of a specific user. Returns a chat list.

GetContacts

Get The contacts of a user. Returns a contact list.

GetHistory

Get the history of a specific conversation. You must pass an object containing „conversation:'...'“. A result will be the last „count“ messages of the chat if count is set or the last 60 otherwise. The result will be sort by timestamp. The server does not guarantee that it has more than the last 60 Messages of a conversation stored. There may be implementations where only the last 60 Messages per conversation are stored.

Message

Send a message in a given conversation. The result is either „Success“ or an errormessage.

RemoveEvent

Trigger a special event that tells other clients of the same user, that all notifications for a certain conversation have been consumed and processed. Generates an „onRemove“.

NewConversation

Create a new conversation.

RenameConversation

Rename a conversation to the given name. A rename is for the user only. Others cannot see the name a user sets for a conversation.

SetName

Set the name that is displayed for a user.

AddFriend

Add someone as a friend. It's nickname is required. The result will be either „Success“ or an error message.

SetStatus

Set your own status. The status is an integer where 0 means offline 1 means online and values between 2 and 99 are something inbetween in % from 2 (online) to 99 (offline). Warning: Most clients will only support the states 0 and 1. Actually you can implement your client to test if the status is 0 (offline) or non zero (online).

SetProfileImage

Set your profile image. The image must be send as „uploadedimage“.

SetGroupImage

Set the image of a conversation (globaly). The image must be send as „uploadedimage“ and the conversation in the attribute.

InjectEvent

Inject a custom event send to other clients. You can use this for webrtc. Conversation, type and message of the event are required.

Data

Send data to a conversation. You must specify the conversation and submit the file via „uploadedfile“. Only files of the following type are accepted: png, jpg, avi, mp4, txt, zip, java, rb, js, c, cpp, h, obj, blend, fsh, vsh. The type is determined by the fileending. Png and jpg are interpreted as images and avi and mp4 as videos. The rest is interpreted as files.

A file upload generates messages that start with #image, #video or #file and then are followed by a link to a file. Users can link external files directly by using thes tags, too.

Helper-Classes

```
class Event {
     type = "onMessage";
     msg = "nick1,nick2";
     nick = "nickname";
     text = "The actual chat message!";
}

class History {
     messages = array();  // of ChatMessages
     conversation = "nick1,nick2";
}

class ChatObject {
     conversation = "nick1,nick2";
     name = "Demo Chat";
     time = "1900-01-01";
}

class ContactObject {
     nick = "nick1";
     name = "Tester";
     status = "0";
}

class ChatMessage {
     author = "Tester";
     nick = "nick1";
     time = "1900-01-01";
     text = "The actual chat message";
}
```

Minifying helper classes

type = t

msg = m

nick = n

text = x

messages = m

conversation = c

name = e

status = s

time = d

author = a

Batch-Processing

There are a few requests that can be passed as a batch of objects.

Input and output objects are in the same order. So input[0] => output[0] and input[9] => output[9].

// TODO further specification

Multi Server Support

Servers must not communicate with other servers. This is against the design idea of XJCP. The idea of xjcp is to create local chat networks with a closed userbase.

However, clients must support multiple servers to be connected to. The client will determine on which server a given nick is used for by either splitting the username as „nick@server“ or by seperately asking for a username and a server.

Clients should make transparent to the user to which XJCP-Server a chat/contact belongs.