TISCaP

TISCaP is Simplicated Chat Protocol.

TISCaP is a client/server protocol for simple, immediate, text-based chat sessions. Its messaging system is designed towards live communication, and has no notion of persistent user presence, or delayed message delivery.

Contents

(clickable!)

Abstract Overview

A Client connects to a Server on TISCaP, establishing a presence on that Server's Chat Circle. A Client may post messages publicly or send messages privately; public-posted messages are delivered to every connected Client, where a privately-sent message is delivered to just one Client.

Terminology on the Parties Involved

(Ready for some obviousness? Okay, here we go…)

A **User** is a human interacting with other humans, via technologies implementing this protocol.

A **Client** is an application acting on behalf of a User, communicating in this protocol.

A **Server** is an application to which Clients connect, and over which Clients exchange messages. Where the Server is the concrete software handling these connections, the **Chat Circle** is a convenience term, meaning the effective community of Clients connected to a Server.

Line Transmission

TISCaP defines communications over a two-way, reliable network stream connection, such as TCP. It usually takes place on port 4020. All communications are text, encoded in UTF-8.

Every interaction between Client and Server takes the form of a **Transmission**. A Transmission consists of a **Command**, sometimes followed by **Associated Data**. Formally,

```
Transmission  :=  (Command)
                  OR (Command + Data)
```

(`:=` means 'is a' or 'consists of')

A Command defines whether or not the receiver must expect Data to follow it. (The default is no Data.) A Command consists of a **Verb**, followed by **Associated Arguments**. Commands are always terminated by `CRLF`.

```
Command  :=  Verb  { + SP + Associated Argument }
             + CRLF
```

Tokens enclosed in {curly brackets} are required in some circumstances and prohibited otherwise.

The particular Verb defines the format of Associated Arguments that follow it (described in more detail below).

A Verb is an alphanumeric, case-insensitive word defining the action of the Command, preceded by an opening character. The opening character is "`/`" on client-to-server Verbs, and "`]`" on server-to-client Verbs.

```
Verb  :=  ("/" or "]")
          + Alphanumeric, case-insensitive word
```

A Verb must be one of the following, from Client to Server:

- /Login
- /Users
- /Public
- /Close
- /Private

or one of the following, from Server to Client:

-]Welcome
-]Disconnected
-]BadSyntax
-]UsernameTaken
-]Private
-]ActiveUsers
-]Connected
-]Public
-]Error

If a Verb is encountered which expects some Associated Argument, the Argument follows the mandatory `SP` after the Verb. The format of the Argument is defined by the Verb that expects it. In general,

```
Associated Argument  :=  Text (UTF-8), excluding CRLF
```

If a Verb is encountered which expects Data, Data follows immediately after the Command's terminal `CRLF`. Data, like all other Transmissions, is expected to be text encoded in UTF-8. It is terminated by the End Of Transmission character, defined as Unicode and ASCII codepoint 4. This character (abbreviated `EOT`) can be expressed in Java strings as `\u0004`, and can often be typed into a Terminal as Control-D.

```
Data  :=  Text (UTF-8), excluding EOT
          + EOT
```

A Server or Client may close the connection at any time. See *Closing a Connection* for semantic details on this issue.

It is a syntax error for any party to send a command which is disallowed at a particular stage of interaction between the Server and Client, as outlined below.

Connection-Establishing Commands

/Login

A `/login` command is the first Client command sent to a Server after establishing a connection. It takes a single Argument, the desired user name, and no Data.

```
Login Transmission  :=  "/Login"
                        + SP + Username
                        + CRLF
```

A **Username** is a case-sensitive identifier for the logging-in User.

```
Username  :=  1 to 16 Alphanumeric Characters
```

]Welcome

A `]welcome` command is sent by the Server to the Client upon successful login. It indicates the Server now associates the Client with its requested Username and is a member of the Chat Circle.

Takes no Argument and no Data.

Until a Client receives a `]welcome` Command, it ***must not*** send any Transmissions to the Server except `/login` commands. Likewise, until a Server sends `]welcome` to a Client, it ***must not*** send any Transmissions to that Client other than `]welcome` or `]usernameTaken`, or other applicable error Commands.

A client may only send `/login` once after connecting, unless it receives `]usernameTaken` and the connection is not closed, in which case it may `/login` only once after each `]usernameTaken`. It is bad syntax to `/login` too many times, especially after receiving `]welcome`.

]UsernameTaken

Sent to the Client to indicate that their requested Username was not available. Takes no Argument and no Data.

Again, the Server may close its connection with a Client any time. After sending a `]usernameTaken` Command would be an especially appropriate time to do such closing, but is not required.

]Connected

Sent to every Client when a user logs in. Takes one Argument, the name of the user who logged in, and no Data.

```
Connected Transmission  :=  "]Connected"
                            + SP + Username
                            + CRLF
```

When a user logs in to the Chat Circle, the Server ***must*** send `]connected` to every Client. A Client may ignore this Command.

]Disconnected

Sent to every Client when a user logs out or is known to have disconnected from the Server. Takes the same syntax as `]connected`.

Messaging Commands

/Users

A Client sends `/users` to the Server to request a list of all users logged in. It merely indicates that the Client is interested in this list. The Server ***should*** respond within a reasonably short amount of time with an `]activeUsers` Command, though the Client cannot expect that the very next Transmission it receives will be this Command.

Takes no Argument and no Data.

It is recommended, though not required, that the Client send `/users` just after receiving `]welcome` confirmation.

]ActiveUsers

Sent by the Server to a user, generally in response to a `/users` Command. This Command takes, as Argument, a list of logged-in Users to the Chat Circle.

```
ActiveUsers Transmission  :=  "]ActiveUsers"
                              + SP + UserList
                              + CRLF
```

The **UserList** Argument is a comma-separated list of Usernames. There are no spaces in this list.

```
UserList  :=  Username { + "," + Username }...
```

Ellipsis (…) indicates 0 or more of the bracketed phrase, depending on how many people there are logged in.

The Server is not expected to broadcast `]activeUsers` Commands without prompting. However, the Client ***must*** accept `]activeUsers` at any time after login.

/Public

Command which the Client uses to send a Message to every logged-in user. Takes no argument, but expects `EOT`-terminated Message Data in the standard format.

```
Public-Send Transmission  :=  “/Public” + CRLF
                              + Data (ends in EOT)
```

]Public

Command which the Server uses to deliver a `/public`-originated broadcast Message to every Client. Takes a single argument, the Username of the sender.

```
Public-Deliver Transmission  :=  “]Public” + SP + Username
                                 + CRLF + Data (ends in EOT)
```

It is presumed that ***every*** logged-in User, even the sender, will receive the Message.

/Private

Command which the Client uses to send a private Message to one particular logged-in user. Takes a single argument, the Username to send to, and `EOT`-terminated Message Data in the standard format.

```
Private-Send Transmission  :=  “/Private” + SP + Username
                               + CRLF + Data (ends in EOT)
```

Upon receiving a Message, public or private, the Server ***should*** deliver the Message, and ***must not*** hold it for later delivery to a User who is not logged in. If the Server is unable to send the message, for some reason, it may return an `]error` to the sender.

]Private

Command used by the Server to deliver a `/private`-originated Message. The syntax is the same as for `]public` Transmissions.

Error Commands

(See also `]usernameTaken` under Connection-Establishing Commands.)

]Error

Indicates to the Client that some general error has occurred. The Client ***should*** notify the User as to the error. This Command takes one argument: a human-readable message indicating what went wrong.

```
Bad Syntax Transmission  :=  “]BadSyntax” + SP
                             + ErrMsg + CRLF
```

The space after the verb is required. Any text between it and the `CRLF` is considered part of the message text.

```
ErrMsg  :=  Text (UTF-8), excluding CRLF
```

The **ErrMsg** may be empty—zero characters—if the Server has nothing to say.

]BadSyntax

Indicates to the Client that a Transmission it sent did not follow the protocol. This should be used for debugging the Client; a Client ***should never*** send any Transmission that would trigger a `]badSyntax` error.

Follows the same syntax as a general `]error` Transmission.

Closing a Connection

A Server may close the transport connection any time, for any reason. If it does so explicitly, the Client may assume that it is logged out. The Server may wish to do so to prune stale connections, for instance, but such behavior and rationale is entirely implementation-dependent.

If the connection is broken inadvertently, either party may drop it, but only the Client may attempt to reconnect (if it wishes).

/Close

The Client ***should*** send `/close` to the Server when the User is ready to log out. After sending `/close`, the Client ***must not*** send any more Transmissions, and ***must*** close the transport connection. The Server, upon receiving `/close`, ***must not*** send any more Transmissions to that Client (though more may arrive due to asynchrony between the two directions of the transport stream).

A Client is not prohibited from terminating its connection without sending `/close`, but is discouraged from doing so.

Glossary

A recap of some of the formal definitions of terms, as defined previously.

Characters

- `SP` is a space character.
- `EOT` is the End-Of-Transmission character; it is Unicode and ASCII codepoint 4, and can be represented in Java as "`\u0004`".
- `CRLF` is a carriage return followed by a line feed ("`\r\n`").

These, as all other text, are UTF-8 encoded.

Broad Linguistic Units

*Here, **`:=`** means 'is a' or 'consists of.' Tokens enclosed in {curly brackets} are required in some circumstances, and prohibited otherwise. Ellipsis (…) indicates 0 or more of the bracketed phrase.*

```
Transmission  :=  (Command)
                  OR  (Command + Data)

Command  :=  Verb  { + SP + Associated Argument }
             + CRLF

Verb  :=  "/Login"
          OR "/Close"
          OR "/Users"
```

```
            OR “/Private”
            OR “/Public”
            OR “]Welcome”
            OR “]UsernameTaken”
            OR “]Connected”
            OR “]Disconnected”
            OR “]Private”
            OR “]Public”
            OR “]BadSyntax”
            OR “]ActiveUsers”
            OR “]Error”          alphanumeric, case-insensitive.

Associated Argument  :=  Text (UTF-8), excluding CRLF

Data  :=  Text (UTF-8), excluding EOT
          + EOT

Username  :=  1 to 16 Alphanumeric Characters

UserList  :=  Username { + “,” + Username }...
```

Authors

Prepared by *Talus Baddley*.

Protocol Designed by

Daniel Rodosky
Sean Groathouse
Talus Baddley
Taz Chapman
Shane McKee
Jace Maxfield
Patrick Hanna
Isaiah Erichsen
Aprille Hanley
Christopher Fowles
Nicole Thomas
Daniel Barton
Morgan McCoy
Bart Joseph Nadala
Jose Garcia
Kyle Swanson
Charles Bailey
Jennifer Giere
Tanner Paige Floisand
Darrell Henley
Jonathan Dolan