Austin Eversole

6/3/2022

Computer Networks

Kaiwa Chat Protocol

Service Summary:

Kaiwa Chat, KaiChat for short, is a stateful application layer network protocol to facilitate a chat capability between two or more hosts. It follows the client-server network model, where a server can take connections from multiple clients concurrently.

Kaiwa Chat gets its name from the Japanese word for conversation: “Kaiwa”. KaiChat’s main function is to allow its users the ability to chat with each other across the internet. It accomplishes this by designating each user as a client, and having each client connect to one central server that can manage each socket connection for the client. The server will deliver messages to and from the client. KaiChat uses separate chatrooms within the KaiChat network for conversation, called channels. Users can chat with each other by joining an existing channel, or users can create a new channel and wait for other users to join the channel. The KaiChat protocol uses Transport Layer Security (TLS) to secure its communications.

Definitions:

Clients:

A client is anything that can connect to a server. To distinguish between clients, each client is given a username that can be up to ten characters long. The client username also must not be a duplicate to a client username that already exists in the server. The username must not be “SERVER”, which is a reserved username.

Servers:

A server is a device attached to the network that provides the chat service for clients. The server forms the backbone of KaiChat. There is only one server per KaiChat network allowed in the current implementation. It forwards messages that it receives from clients to all other clients. It also keeps track of all chat messages so it can keep clients up to date on chats in their channel. In order for the server to accept a new connection from a client, version negotiation must be performed and the client must send its preferred version number (and the user behind the client must specify a username).

Channels:

A channel is a group of one or more clients. When a message is sent on a channel, all clients on that channel will receive the message. When a client wants to join a channel, that channel is created if the channel doesn’t already exist. If the channel does exist, the client joins the existing channel. Channels are named, and there may not be more than one channel with the same name at a time. Channel names are strings that start with @ and may be up to fifty characters. Once a client joins a channel, they are able to chat with other clients that may be in the channel. Clients can only be in one channel at a time. When there are no more remaining clients in a channel, the channel closes. The channel name is then freed up to be created again.

Starting KaiChat service:

The client initiates the first connection in the KaiChat network by sending a ping to the server. The server responds to this ping from a client with a pong message, after which the server sends the highest supported version number of the KaiChat Protocol to the client. Once the client selects a version, it sends this version to the server and version negotiation is complete. The client then sends client information (which in the current implementation is the client username). After receiving the client information, a full connection is established between the client and the server from which the client can join a channel.

Maintaining KaiChat service:

KaiChat runs over the TCP transport protocol. Specifically, Kaichat is configured to run on TCP port 29200 on both the server and client. Once a client has joined a channel, the client sends a query message every 3 seconds to query the server for any new messages. The server sends a message back to the client for each new chat until the client is caught up. To do this, the server stores a running history of all chat messages. It relays chat messages to all clients (except for chat messages sent by the clients themselves) to keep clients up-to-date, and leaves it to the client application to ignore chat messages from channels it has not joined. If there are no new chat messages, the server replies to the clients query using a special up-to-date flag (see query section of message definition) to indicate that the client is up to date. If a client sends the quit command (see the KaiChat specification for more details), the KaiChat server closes the socket used in the TCP pipe with that client To do this, the user should enter *!quit* at any prompt once a full connection is established between the client and server.

The KaiChat Specification:

KaiChat uses a standard message format for all messages. This standard message format consists of three parts: the prefix, the command, and the command parameters. All characters in this message definition use ASCII encoding.

PDU Definition:

Prefix	256 bytes representing a string (multiple ASCII characters)
Command	4 bytes representing an integer
Command parameters	2 KB big-endian bytestream (multiple data types can be represented in this bytestream – more details in command summary list)

Representation Notes:

The PDU definition components will follow the endianness of the computer and will be converted as needed (see end of message definition for more details). A big-endian integer in this context represents numbers by powers of 2 where the most significant byte is the farthest to the left (the byte transmitted first). For example, a 1 byte big-endian integer ‘3’ would be represented by the byte 0011.

A character in this context is represented by a 1-byte value, using ASCII encoding. A string is represented by multiple characters.

Message Definition

The prefix is a 256 bytes and represents a string containing the client username and the channel name separated by a space character (if applicable) when the message is being sent by the client. The prefix will contain the name “SERVER” and the channel name separated by a space character (if applicable) when the message is sent by the server. There are many cases where the prefix will be empty (see below table for all use cases of the prefix). The command section will contain one of nine commands which are of 8-byte integers. The command parameters section is 2 kilobytes, and it can represent multiple data types which will be described in detail in the commands summary table below.

A summary of each command and what the prefix and command parameter sections will contain follows:

1 – VERSION_NEGOTIATION	The first message sent after ping/pong. Prefix will contain SERVER when server sends highest support version number. It will be empty when client selects the version. [6 bytes – ‘SERVER’ or empty] [200 bytes – empty] Command parameters must be the version number (2 bytes representing an integer in big-endian – can represent version 1 through 256). The representation of the command parameters byte stream for this command follows: [4 bytes - Version Negotiation] [1998 bytes – empty] (empty refers to a 0-byte where all 4 bits are 0).

2 – CLIENT_INFO	The client sends the username desired by the client and the client hostname. Prefix will be empty. [256 bytes – empty] Command parameters must be the username (10 ascii characters max) and hostname (represented as ascii characters), separated by a space character. The representation of the command parameters byte stream for this command follows: [10 bytes – username] [1 byte – space ‘ ‘] [16 bytes – host name] [1973 bytes – empty]
3 – JOIN CHANNEL	The client indicates that it wants to join a specific channel. Prefix must contain the username and channel name, separated by a space character. [10 bytes – username] [1 byte – space ‘ ‘] [50 bytes – channel name] [229 bytes – empty] Command parameters will be empty. The representation of the command parameters byte stream for this command follows: [2000 bytes – empty]
4 – SEND_MESSAGE	The client indicates that it wants to send a message Prefix will be empty [256 bytes – empty] Command parameters will contain the message to be sent, ending with the ASCII nul character. [2000 bytes – message, followed by empty memory after the nul character]

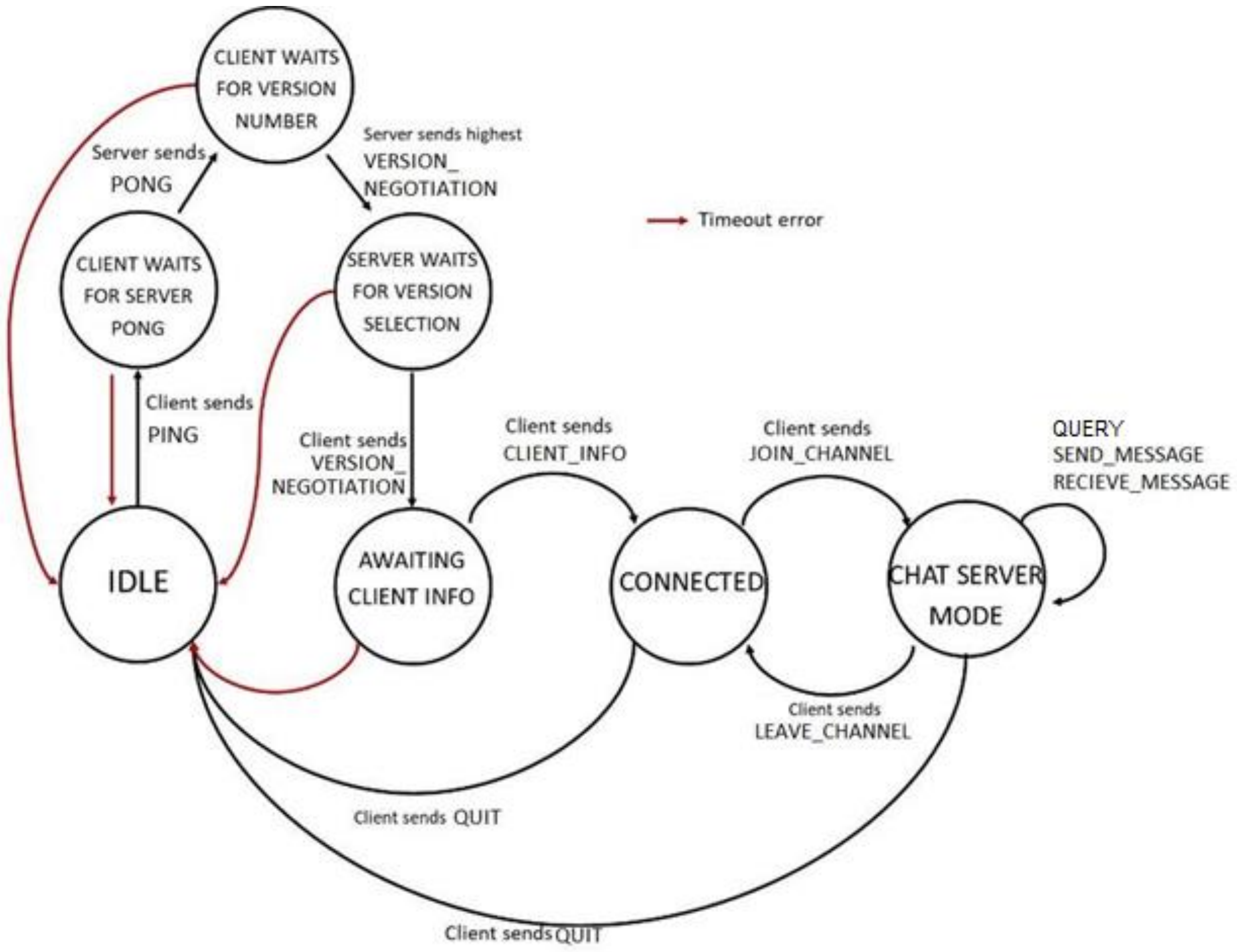
5 – RECIEVE_MESSAGE	The server delivers a message to the client. It is sent in response to a client query. Prefix will contain the username that sent the message, the channel name, and the up-to-date flag, all separated by a space character. If there are no new messages in response to the query, up-to-date flag will be 0. [10 bytes – username] [1 byte – space ' '] [50 bytes – channel name] [1 byte – Up-to-date status] [228 bytes – empty] Command parameters will contain the message to be received, ending with the ASCII nul character. [2000 bytes – message, followed by empty memory after the nul character]
6 – LEAVE_CHANNEL	The client leaves the channel Prefix will be empty [256 bytes – empty] Command parameters will be empty [2000 bytes – empty]
7 – PING/PONG	Ping/Pong message. It is called ping (the query) if sent by client, and pong (the response) if sent by the server The prefix will be empty. [256 bytes – empty] The Command parameters will be empty. [2000 bytes – empty]
8 – QUERY	The client is querying the server for any new messages (when user is in a channel) The prefix will be empty. [256 bytes – empty] The command parameters will be empty. [2000 bytes – empty]

0 – QUIT	The user quits the chat client Prefix will be empty [256 bytes – empty] Command parameters will be empty. [2000 bytes – empty]

When an element of the message section in the summary above is described as a "must" have, then if any of those elements in the message section are lacking, it is an invalid message and is ignored by the server or client.

The KaiChat Protocol is assumed to be run with little-endian byte ordering in mind, but can run on both little-endian and big-endian processors. It would be preferred for the KaiChat server to run on a little-endian processor to minimize conversions. If a client on a KaiChat network uses byte ordering that the other clients and server don't use, this would cause a problem because as the endianness doesn't match. To solve this, the server determines what endianness the client is using during version negotiation. When the server reads the message to decide what command should be executed, it will check the command section of the message against both the little-endian and big-endian form of the 32-bit integer '1' as an entry into the version negotiation logic (The command '1' is the version negotiation command). If the big-endian form of the 32-bit integer '1' is detected, then the server will flag this client as using a big-endian processor and interpret all message data received from here on out as having big-endian endianness and convert incoming data to little-endian before processing. If the little-endian form of the 32-bit integer '1' is detected, the server will do nothing as it runs natively with little-endian endianness so the endianness is compatible. After determining the endianness of the client and setting the flag if applicable, the server will continue with version negotiation.

DFA


CLIENT WAITS FOR VERSION NUMBER
Server sends PONG
Server sends highest VERSION_ NEGOTIATION
Timeout error
CLIENT WAITS FOR SERVER PONG
SERVER WAITS FOR VERSION SELECTION
Client sends PING
Client sends VERSION_ NEGOTIATION
Client sends CLIENT_INFO
Client sends JOIN_CHANNEL
QUERY
SEND_MESSAGE
RECIEVE_MESSAGE
IDLE
AWAITING CLIENT INFO
CONNECTED
CHAT SERVER MODE
Client sends LEAVE_CHANNEL
Client sends QUIT
Client sends QUIT

Extensibility

The KaiChat protocol is designed to accommodate extensions and additions to it in the future. It does this by using a version negotiation system, of which the protocols current implementation is version 1 of the KaiChat protocol. The server sends its highest version in a VERSION_NEGOTIATION message to the client, after which the client selects a version that it is compatible with and sends that version in a VERSION_NEGOTIATION message back to the user. At this point, the server and the client both know the selected protocol version and they can communicate.

The KaiChat message specification also dedicates unused memory in the prefix section of the message definition for future use. These could be used as options in future extensions. The prefix section is a 256 byte string, of which the client username can only be 10 characters (10 bytes), the channel name can only be 50 characters (50 bytes), and a space character servers as a separator (1 byte). That leaves 195 bytes for future use in the prefix.

Future extensions to the KaiChat protocol could include a new command that informs clients of all existing channels on the KaiChat network, the ability to ban obnoxious or malicious users, and introducing a permission system that would be required to elevate user privileges to ban users.

Although it is incredibly rare, some computers store integers as 2 or 8 bytes rather than 4 bytes. In the future, this protocol could be extended to incorporate the potential for a variable int size. The most efficient way to do this would likely be to make the command component of the KaiwaChat message PDU 8 bytes rather than 4 bytes.

Performance

KaiChat takes advantage of the reliability of TCP to ensure that message PDU's get from the client to the server, and from the server to the respective clients. The reliability guarantees of TCP can slow performance of KaiChat because of TCP features such as sliding window protocols, retransmission, and congestion control. However, the KaiChat protocol does not require a degree of performance substantial enough such that these features would detrimentally affect the KaiChat service.

In the Chat Server Mode state, the client sends a query to the server ever 3 seconds to ask for any new messages. This may be a factor to analyze when hosting a large KaiChat network with thousands of clients if the server is not supported by capable hardware, but it should not cause any particularly noticeable performance issues in servers using modern hardware. Where relevant, a new KaiChat network can be set up with another server if noticeable performance issues are encountered using one network.

Security Implications

The KaiChat service uses Transport Layer Security (TLS) as an application protocol secure channel. Outside of TLS, KaiChat does not have extensive security features in its current implementation. This is because there are no user permissions or private channels – everyone is allowed to join any channel as long as the client knows of the channels existence.

If an attacker knows the protocol, then depending on the attack vector there is potential that an attacker could impersonate the server by crafting a PDU with username 'SERVER' in the prefix. However, because TLS encrypts the application layer KaiChat protocol during transport, this provides a layer of security to prevent attacks such as this.

A timeout serves as a security feature to prevent denial of service attacks, which attackers could initiate by starting a large number of connections. An attacker could do this by pinging the server but never sending a selected version (for version negotiation) or client information. To prevent this, the server will timeout if it doesn't receive a VERSION_NEGOTIATION message after 30 seconds from sending the PONG response (the times between the server and the client are not synchronized, and the 30 seconds are measured from the server side). Similarly, the server will timeout if it doesn't receive a CLIENT_INFO message after 30 seconds from receiving the VERSION_NEGOTIATION message. When the server timeouts, the connection to the client is ended and the server releases the thread dedicated to that client. The client also has a timeout if it does not receive a PONG or a VERSION-NEGOTIATION message from the server. However, from a server security standpoint this is not as relevant as the server timeouts are.

Differences and explanations of differences in the 2nd version of the Protocol Design Paper:

References to both the server storing hostname and the client sending hostname to the server in the CLIENT_INFO messages were removed. in the current implementation, the server does not store the hostname. This is because that is taken care of when socket is created server-side before version negotiation even beings.

The ability for a user to enter !quit at anytime in any prompt (once connected to a server) to end the connection to the server was added.

Added a new message command type – QUERY. A client sends a query to the server every three seconds. The server then sends RECEIVE_MSG PDU's for each new message in the server's chat history that the server has not sent to the client yet. If there are no new messages in the queue to the send to the client then an up-to-date flag will be set to true.

The QUERY command has also been added to the DFA in the Chat Server Mode Node.

Requirements to send both the username and channel in the prefix of every PDU was removed. This is because it became redundant with the threaded nature of the concurrent chat server - the server keeps a thread open for each client wherein the current username and channel can be stored. It is simpler to just use reuse the username and channel for server-side operations than pull the username and channel from the client PDU prefix each time.

What the server stores in terms of chats, users, and channels has been changed. In the current implementation, the server will be keeping track of all chat. It will not be storing a list of users that are active, channels that are active, or what channel a user has joined.

Username requirements that were unnecessary have been removed. The server won't confuse a space in a username for a delimiter because it reads the first 10 bytes of the prefix as the username, rather than searching for a delimiter.

Removed silently ignoring PDU's received that do not follow the strict requirements for elements in message sections. Although it could be a good security, it could give deadlock in some places.

Rephrased the endianness section.

Added section about some computers that store integers as something other than 4 bytes

Fixed several typos and grammatical errors.

Made client, server, and channel definitions more concise.

Updated “Maintaining KaiChat Service” section to reflect recent changes cited above.