Lichat, A Lightweight Chat Protocol

Nicolas Hafner

Shirakumo.org

Zürich, Switzerland

shinmera@tymoon.eu

ABSTRACT

In this paper we present Lichat, a simple and lightweight chat protocol. We take particular interest in how relying on Common Lisp as a base language led to a fast prototyping of the protocol and rapid evolution of its design, and then evaluate the consequences of these decisions when porting the protocol to other languages such as Java, JavaScript, Python, and Elixir.

CCS CONCEPTS

• **Networks** → **Application layer protocols**; • **Information systems** → **Web services**; • **Software and its engineering** → *Software prototyping*;

KEYWORDS

Common Lisp, Chat, Protocol, Design, Networking

ACM Reference Format:

Nicolas Hafner. 2022. Lichat, A Lightweight Chat Protocol. In *Proceedings of the 15th European Lisp Symposium (ELS'22).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.5281/zenodo.2636508

ELS'22, March 21–22 2022, Porto, Portugal

ACM ISBN 978-2-9557474-3-8.
https://doi.org/10.5281/zenodo.2636508

1 INTRODUCTION

The IRC (Internet Relay Chat) protocol has long stood the test of time as a reliable protocol for text chat communication. Large networks and a plethora of mature clients still exist and are in use today. However, The IRC protocol is also plagued by a number of issues that stem from its historical design and largely distributed evolution. In particular, the protocol has many different, incompatible extensions, and features many idiosyncrasies that make it difficult to write new clients or servers.

IRC has also failed to catch up with several changes in user expectations for chat protocols today. It lacks support for custom emoticons, search, formatting, does not permit multiple connections to the same user, and does not even offer an account registration or authorisation process. Instead the onus is on the servers to offer extensions, and on the clients to implement such features completely locally (forgoing synchronisation).

Over time many other chat systems such as AIM, ICQ, Skype, have come and largely gone, though all of these focus on a different model of operation than IRC. They focus on a friends-list based approach where people talk to each other directly, rather than relying on sometimes huge public chat rooms.

More recently, proprietary systems such as Slack, Discord, and Microsoft Teams have sprung up that follow the IRC model more closely, though all focus on a more sheltered experience where communities are gated off via separate "servers" in which the channels reside. People cannot see channels from other servers without being invited. All of these solutions use proprietary, closed-source protocols and often even prohibit the usage of third-party clients, severely restricting user-freedom and endangering the longevity of the platform.

On the open source side, the Matrix (formerly Riot) protocol has been trying to offer a modernised, federated approach to the IRC model. Being federated and open it has a much higher chance at staying around for decades to come than many of the aforementioned proprietary solutions. However, just like the proprietary counterparts, the Matrix protocol is complex and far from trivial to implement.

With Lichat we instead focused on developing a protocol that should be trivial to implement a client for, and not too hard to implement a server for. We also focused on a design that would avoid many of the pitfalls we found in IRC, and in general strives for a few basic properties that ensure reliable and easy to understand communication between the client and server. The core protocol of Lichat is very small, only covering the most basic elements, instead

relying on protocol extensions to provide additional functionality that bring it up to par with competing systems such as Discord or Matrix.

In section 3 we present our initial prototyping for the protocol and illustrate the advantages of picking Lisp as a starting point. In section 4 we discuss how the design evolved after the initial prototyping phase and discuss the porting of the protocol implementations to other languages. Finally, in section 6 we discuss the use and development of extensions to provide additional functionality.

2 RELATED WORK

IRC

Matrix

XMPP

Whatever?

3 INITIAL DESIGN

As a basis for the protocol we chose to use UTF-8 character streams for the transport. This ensures that protocol messages can be read and written by a human, which eases development and debugging. Specifying the character encoding also avoids ambiguity when interpreting the underlying bytes. For the actual data format, we relied on Common Lisp s-expressions, which could simply be `read` and `printed` using the native Lisp functions.

Doing so both eliminated the need to write encoders for the prototype, and ensured that we had a data structure basis that could support more complex data payloads further down the line. The actual message data types communicated with were also directly represented using Common Lisp standard classes, allowing for tight encapsulation of data attributes, and the natural combination thereof through multiple inheritance.

Each message to be sent was trivially encoded by printing a list consisting of the message's type and its construction arguments. The decoding then followed just as trivially. Doing so meant we could set up a server-client communication with as close to no effort as possible while remaining extensible for future changes. Particularly, the choice of symbols for message type names and argument names allowed us to use packages to isolate additions made by extensions into their own namespaces, avoiding future conflicts.

Another of the issues with IRC's design is that it is not strictly possible to verify whether messages arrived at the server at all, and if a response is given, which request that response belongs to. To rectify this issue, we require the inclusion of an `:id` field in every message sent, which the server will either re-use on a mirror-reply, or include as a reference in case of a failure. We also require the server to always reply to a message with some form of a response, even if it is just sending the original message back to the client.

The client should also include a `:clock` field of the local Universal-Time at the time of sending. This ensures that, if distributed to other users, they can get an accurate ordering of events that reflects the sending user's point of view, without potential network delays changing the sequence of events. It also allows the server and clients to estimate lag in the network and issue warnings on unstable connections, or terminate them early, ensuring an ultimately more stable communication.

We also require the inclusion of a `:version` field on connection to allow the server to reject the connection or switch to a compatibility mode. We also include a `:extensions` field to allow the listing of supported protocol extensions from the client, which the server then modifies with its own set of supported extensions when confirming the connection. This way both server and client are fully aware of supported extensions that they should deal with.

On the client to client communication side, we decided to enforce all communication between clients to occur over channels. This simplifies the design on both sides, as there don't need to be any special rules to deal with direct communication. With all of this groundwork laid, we could implement a basic chat server that allowed for communication over multiple channels, as shown in Listing 1.

```
> (connect :id 0 :clock 5243 :from "tester" :version "1.0" :extensions ())
< (connect :id 0 :clock 5243 :from "tester" :version "1.0" :extensions ())
> (join :id 0 :clock 5243 :from "tester" :channel "lichatters")
< (join :id 0 :clock 5243 :from "tester" :channel "lichatters")
< (message :id 0 :clock 5244 :from "shinmera" :channel "lichatters" :text "Welcome!")
> (message :id 0 :clock 5246 :from "tester" :channel "lichatters" :text "Hiya")
< (message :id 0 :clock 5246 :from "tester" :channel "lichatters" :text "Hiya")
```

Listing 1: A basic protocol exchange.

After this most of the protocol design revolved around the implementation of a permissions system to allow channel operators better control over what users are allowed to do, and the ratification of distinct error types for potential failures.

The permissions system in general is very simple: each channel holds a map where each update type is associated with with either a list of permitted users, or a list of denied users. As all communication between users must occur through channels, this provides a simple, but expressive model to handle moderation. For instance, banning a user simply requires removing their ability to send a `join` message to the channel.

For failures, a simple hierarchy was designed:

1. `failure` Any kind of issue
   - 1.1. `malformed-update` The message could not be read at all
   - 1.2. `update-too-long` The message was too long and was skipped wholesale
   - 1.3. `update-failure` An issue with a request the user sent
     - 1.3.1. `insufficient-permissions` The message was denied
     - 1.3.2. `no-such-channel` The requested channel does not exist
     - 1.3.3. ...

Particularly, `update-failures` carry a field called `:update-id` which refers back to the ID of the original request, allowing the client to correlate the failure with the previous request, avoiding the issue of confusing responses should multiple requests and responses be in flight at the same time.

Finally, in order to ensure that permissions tables actually have any grip at all, and to allow users to use multiple connections at once, we introduced an accounts system against which the user is authenticated when connecting.

The initial client and server implementations were fully in Common Lisp, with a JavaScript client intended for public use. However, we ran into scaling and concurrency issues with the Common Lisp server that proved extremely difficult to debug and understand, as they often took days on the public server to be triggered. The lack of primitives for efficient asynchronous programming and lightweight processes pushed us to attempt a rewrite of the server in Elixir instead, once the core protocol was ratified fully.

4 RATIFIED DESIGN

After the initial proof of concept, we decided to simplify the protocol in order to ease the porting process. The first part was to reduce the s-expression syntax to make the writing of a compliant parser easier, and safer. To this end, we restricted the possible data types to the following:

1. `number`
   1.1. `integer`
   1.2. `float`
2. `symbol` (names are case-insensitive, no gensyms)
   2.1. `keyword`
   2.2. `boolean`
3. `list`
4. `string`

Further, we required the use of a null byte as an end of message marker. This is useful so that a server or client can restore the stream of messages should a parser failure occur, or should the update be rejected due to excessive length.

To ease the initiation and maintenance of the connection, the `:from` and `:clock` fields were made optional, instead requiring the server to automatically fill them in should they be missing. This makes it even more trivial to write a simple, first client.

We decided to specify two distinctions in channel use from regular channels: the "primary" channel, and "anonymous" channels. The primary channel is used to keep track of all users on the server, as all users are automatically joined to this channel after connecting. Anonymous channels are used for private "direct" communications between users. They are named such as the name is picked automatically by the server, and they won't appear in channel listings. Otherwise they operate exactly the same as any other channel, making it trivial for clients to implement private user to user communication, which can also be extended to private group chats by inviting other users.

```
(define-package "lichat")

(define-object lichat:update ()
  (:id id)
  (:clock integer :optional)
  (:from string :optional))

(define-object lichat:channel-update (lichat:update)
  (:channel string))

(define-object lichat:text-update (lichat:update)
  (:text string))

(define-object lichat:message (lichat:channel-update lichat:text-update))
```

Listing 2: A section of the machine-readable protocol specification

The primary channel is particularly useful to manage permissions, as all messages that do not target a specific channel instead default to having the access checked against the primary channel.

Finally, we ratified the definition of all message types in a machine-readable format (Listing 2) that can be parsed using the same parser of the wire format. This allows protocol implementations to easily parse up-to-date representations of the entire message type hierarchy.

The core protocol specification has been incredibly stable since these changes, only requiring minor amendments to clarify and disambiguate certain points. Almost all of the active development since happened within extensions and their specifications, instead.

5 PORTING

We implemented client software in JavaScript, Java, and Python, as well as a server in Elixir. For each of those the porting process for the basic syntax of the protocol was fairly simple, only requiring a recursive descent parser that is trivial enough to write by hand. Most of the issues arose when porting the message class hierarchy.

In Python we were able to make use of multiple superclasses as we would in Common Lisp, keeping the implementation quite straightforward.

JavaScript does not support multiple inheritance out of the box, so we implemented our own scheme following the serialised inheritance model of CLOS. This does however still not integrate neatly with the native JavaScript type system, meaning we also require the use of a specialised type test for dispatch.

Similarly to JavaScript, Java does not support multiple inheritance. Instead of defining an ad-hoc system however, we opt for a stubbing approach where we create a single inheritance chain, instead transforming extraneous types into additional class fields. The Java implementation is further restricted by its static typing requirements, forcing additional logic when parsing concrete class instances from the message representation. Finally, Java does not support runtime creation of classes, requiring us to write a code generator that emits static Java code, instead of creating the classes at runtime like the Python and JavaScript implementations.

Our Elixir server implementation eschews inheritance altogether, as the language does not share an object-oriented system at all. We instead manually flatten the inheritance by specifying all inherited fields on each message type. Each message type is represented through a module, with a function to handle the update on the server-side. Thanks to Elixir's macro system we were able to avoid most of the boilerplate overhead of doing this, but we have so far not implemented automated creation of the message representations from the machine-readable specification.

6 EXTENSIONS

7 CONCLUSION

8 FURTHER WORK

On the protocol side, we would like to develop further extensions, primarily to allow for VOIP, federation, and encrypted messaging.

We would also like to develop further client implementations, particularly for C/C++ for use in the multi-protocol client Pidgin.

9 ACKNOWLEDGEMENTS