Taurus Design Document

Table of Contents

1 Introduction and Scope

This is an outline of the major design decisions for Taurus, a client implementing TauNet v.0.2. Its purpose is to explain the struture of the software system and the reasons for that structure. It does not address the TauNet protocol or the requirements of the client, although the accompanying Software Requirements Specification provides important context. Relevant definitions are supplied in that file and not duplicated here.

2 Architecture

2. 1 Overview

The major functionality of Taurus is split into two separate programs: taurusd, a daemon which listens for incoming messages; and taurus, an interactive client for reading and sending TauNet messages. The two do not communicate directly, but they share a directory for storing messages as well as a number of modules implementing different sections of the system's general functionality.

This structure avoids many concurrency issues (although not all—the conversation files still need to be locked before writing, which is handled by python's fcntl module). It also permits the daemon to be left running in the background without any expectation that the user interface is always taking up someone's screen real estate.

2.2 Filesystem

Taurus uses a working directory in the user's home directory, as is typical for applications that need to save user-specific settings and logs in addition to or instead of systemwide ones (such as desktop environments and IRC clients). The user table is stored in that directory in CSV, and a subdirectory contains conversation files. Each conversation file is named for the user on the other end and just contains a timestamped list of messages, interleaving outgoing and incoming ones. This is for ease of display; the conversation-reading screen just does the Python equivalent of tail -f to watch the conversation as messages go back and forth.

2.3 Modules

Apart from the listening and sending compontents, the Taurus system is divided into a number of modules, each oriented around part of its functionality.

The TauNet module handles the TauNet protocol itself, including forming and parsing messages and managing the user table. This means it's the only file that needs to change when the protocol version changes. Everything that reads or writes messages does so by instantiating TauNetMessage using one of its methods for that purpose: incoming() (which takes ciphertext), outgoing() (which takes cleartext and a recipient), or test() (which simply generates a totally empty message, for checking node status with). The UserList class keeps track of TauNetUsers.

Everything related to encryption is in the CipherSaber2 module; this is valuable for a number of reasons, including the ease of swapping it out if the TauNet protocol adopts a different enrcryption scheme in the future. The encryption module can also be run as a standalone CS2 utility with its own command-line arguments, including a modest array of unit tests. This is an artifact of the development process, but make it a potentially valuable utility in its own right.

Finally, the Filesystem module is the only one which interacts directly with the filesystem (not including logging, which it still handles indirectly by providing logger objects to other modules). This means it's the only place where configuration related to Taunet's working directory and conversation file format needs to be stored. It provides the list of open conversations as well as a generator function which yields the backlog and updates to a specified conversation.

2.4 User Interface

To the user, the interface appears as a sequence of screens with different options: a main menu, a selection of conversations to view, a user list, a conversation, or prompts for outgoing message information. These are implemented very simply as standalone functions that call each other in response to commands from the user. The connections between the various screens form a directed acyclic graph; the acyclic part is important to prevent the memory load of arbitrarily nested function calls.

All the actual interaction (display and keyboard input) is handled by curses. This is a shortcut to a clean page-based (rather than line-based) UI, at the relatively small cost of managing screen drawing.

3 Flaws and Opportunities

3.1 Concurrency

Because the listener and user interface both instantiate the taunet module, they read in the user table separately and can get out of sync—especially if it's updated while the daemon is running. This could lead to the situation where we can send messages to other users but will discard their replies. The only reason this was acceptable in this version is that editing the user table was explicitly not a requirement.

3.2 Interface

There are several irregularities in the curses interface, most of which are the result of rushed implementation and not design flaws. The redrawing scheme and settings should be normalized between the different modes, and in general the interface could be more consistent in several places. Using curses also gives us access to a number of other useful features, like text-input boxes and a terminal bell, which might come in handy in future versions.

Having the various screens call each other's functions directly is okay as long as there are only a few of them, and they can only visit a limited set of each other, but if cycles were introduced this could quickly lead to memory problems. A better model might be independent mode objects, which know their own

configuration settings and contents, and a main loop which simply calls the appropriate one. These wouldn't have to literally be objects; functions which return the next mode to visit could accomplish the same thing.

3.3. Daemon

It would be tempting to run the Taurus listener as a system service, but in its current configuration it's incapable of supporting incoming messages for multiple users. This is excusable on the grounds that TauNet is designed for small personal systems (Raspberry Pis). Whether TauNet nodes are really intended to be single-user machines is not part of its specification.

The daemon could easily take over the client's job of tracking the status of remote nodes, and in fact is better suited for it. It loops more frequently and less busily, and could quietly update node status in the background without the user having to trigger updates manually. The hard part of this would be putting the information somewhere the client can get at it, which raises questions about our concurrency model (or concurrency-avoiding model).

It should be possible to manage the daemon from inside the client, and currently isn't. (There's an open issue in the Taurus repository about it, #21.)

3.4. Encryption

Taurus compiles with the TauNet specification for messages sent over the network, but does not encrypt any of the data stored locally. This is excusable only because a) it's a prototype and b) we seem to be presuming that it runs on a single-user system. Future versions of Taurus, probably in step with future versions of TauNet, should make security a higher priority.