The tmux terminal multiplexer

Nicholas Marriott, 2011

tmux is a program which allows multiple text terminal (TTY) processes to be created and managed from a single text terminal. The set of processes running in tmux may be detached from the terminal, continue running in the background and reattached to a different terminal without interruption. It provides the ability to display output on disparate terminals simultaneously and a large set of features for management of child terminals.

In development since 2007, from the outset tmux was primarily developed on OpenBSD and since mid-2009 has been part of the OpenBSD base system.

This paper presents a brief summary of the motivation for and history of the tmux program, an overview of the tmux design and main data structures, a discussion of the design decisions behind some of the components, and a summary of major limitations and areas for improvement.

History

tmux was started in 2007 with the aim of producing an alternative to the GNU screen program with some fundamental design differences.

GNU screen is a "full-screen window manager"[1] under sporadic development since the 1980s[2].

tmux was created with the following differences in mind:

- For a server-client design with all windows managed by a single server.
- To be fully BSD-licensed.
- To aim for a consistent, modern code style that would be easy to extend.

In addition, the following are major goals:

- Try to have sensible defaults.
- Be conservative and correct: fix bugs quickly, fail fast on errors and limit incorrect output to terminals.
- Provide a full command interface and prefer the shell for scripting.
- Strive for a consistent interface: command syntax should be identical whether used from the shell or a key binding.
- Try to be well documented in manpages.
- Attempt to be portable but minimise dependencies outside the OpenBSD base system.

tmux and OpenBSD

tmux was imported into the OpenBSD base system in July 2009, a few months before the 4.6 release, to replace the window(1) program and is now installed as /usr/bin/tmux.

From the outset, tmux was written with OpenBSD code style and practices in mind and with OpenBSD as the primary development platform, so import into OpenBSD required relatively few changes, mainly to strip out portability code and add an OpenBSD-style Makefile.

Since then, tmux has changed to reuse existing OpenBSD code where appropriate, notably libevent and the imsg framework discussed in later sections. OpenBSD principles that tmux attempts to follow include:

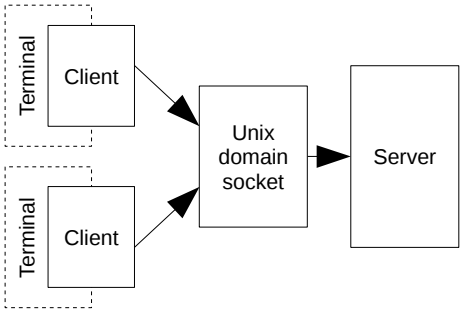
- Use imsg for IPC.
- Maintain a consistent code style (largely following style(9)).
- Strive for code correctness; use safe functions such as strlcpy(3), strlcat(3) and strtonum(3).
- Use the ISC license.
- Attempt to provide sensible defaults; avoid arbitrary changes to defaults without good reason.
- Undertake code inspection and auditing both manually and with automated tools such as lint.

Import into OpenBSD has brought tmux to the attention of a large and technically-proficient user and developer base as well as allowing its developer to work on a highly enjoyable and interesting project beyond tmux itself.

Design Overview

This section gives a description of the tmux design in terms of the primary data structures and the event loop.

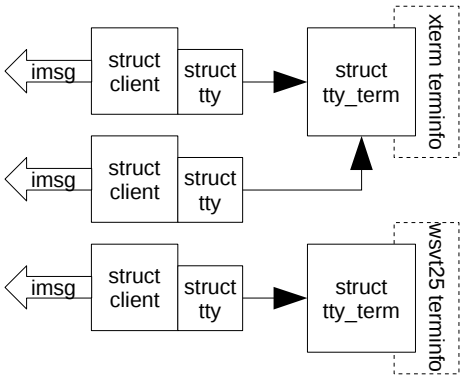
tmux is a server-client system. All child processes, pseudoterminals, configuration and data structures are managed by one server process. The tmux server process may be *attached* to by one or more client processes. Clients have two primary roles: firstly, to transfer the environment and terminal file descriptors to the server; secondly, to occupy the terminal and prevent other processes from using it until the client exits (that is, it is *detached*). Clients attach to a server through a Unix domain socket and uses libevent[3], the OpenBSD imsg framework and a simple binary protocol to communicate.



Attachment of a client to a tmux server

On attachment a client sends a series of messages which transfers command line arguments, the environment, and the stdin, stdout and stderr file descriptors to the server. The terminal (the client terminal) is then managed by the server.

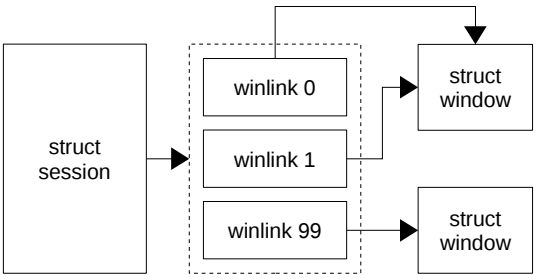
This design permits an important feature: all child processes inside tmux may be displayed on any client terminal or on multiple terminals; this would pose considerable problems were the processes owned by multiple server processes.



Client data structures in the server

In the server, a client and its imsg channel are represented by a *client* structure which is associated with a *tty* structure which represents the client terminal. Each tty structure links to a *struct tty_term* which is a container for the terminfo(5) data required to correctly write to a terminal. tmux intentionally limits use of the curses API to reading terminfo entries: although the curses API has many uses, it suffers from a poor namespace and does not lend itself well to use in applications managing multiple terminals.

Each client is associated with a *session*: this is the client's *current session*. In the tmux server, a session is a collection of links to *windows* and some associated state. Sessions are linked to windows via a tree of intermediary structures, *winlinks*, which permit a window to be linked to multiple sessions

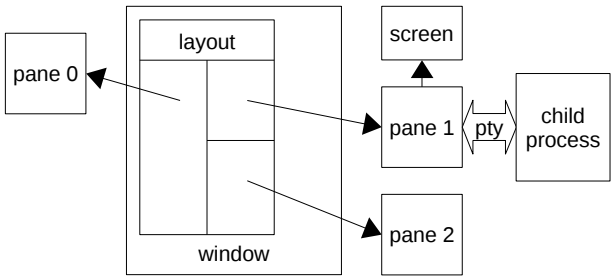


Session and window structures

As all sessions are within a single server, a client may be moved between several sessions during its lifetime.

A window in tmux is a collection of rectangular *panes* each of which is associated with a *cell* in the window's *layout*. Each pane is connected to the master side of a pseudoterminal and associated with an internal *screen* which represents the current pseudoterminal state. The screen is a grid of text cells and associated state (such as the cursor position) which is updated as the child process produces output and is used to display full pane state when required, such as

when a pane is first shown on a terminal after a client is attached. This screen together with the pseudoterminal used to update it is the primary mechanism allowing a child process to continue running without any client terminals.

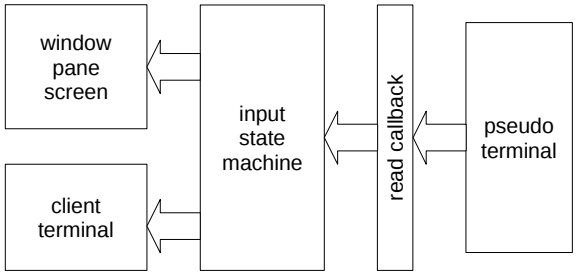


Window and pane structures

The main consequences of this design is that panes are present only in one window at a time; this was omitted in the interests of code simplicity when panes were implemented.

The tmux server is driven by the libevent event-driven programming library. Clients, their associated terminals, and window panes have buffer events associated with them. Buffer events are a libevent construct which automatically read and write between a buffer and a file descriptor. In addition, events are recorded for a variety of timers and other purposes.

For example, the main flow of input data is driven by a libevent buffer event callback on the master side of the pseudoterminal associated with a pane.



Flow of input data

This delivers the data into a state machine parser for vt100 escape sequences based on the design published by Paul Williams[4]. The output from the state machine is series of function calls to update the screen associated with the pane and any client terminals currently displaying the pane.

tmux and libevent

libevent is an event-driven programming library originally written by Niels Provos. The project is now maintained by Provos and Nick Mathewson. Currently OpenBSD includes libevent 1.4 in the base system.

tmux switched to libevent in late 2009 from a custom poll(2) loop. The main reason for this was to simplify the code - the poll loop was becoming increasing complex to deal with a large number of clients, windows, timers and other events. tmux is naturally event driven and fits well with the libevent model - much of the work is reading and writing to file descriptors so buffer events, previously mentioned, were very attractive and replaced custom buffer functions.

Use of libevent has been largely successful but it has had some difficulties. Many of these have been related to portability.

tmux uses libevent primarily on pseudoterminal file descriptors or those of other special devices and these have been found not to be well supported by the high-performance event mechanisms preferred by libevent. For example:

- epoll on Linux doesn't support /dev/null. This is a difficulty as tmux now passes all client stdio file descriptors into the server and adds them to libevent.
- Likewise, kqueue on OpenBSD didn't support /dev/null.
- OpenBSD and FreeBSD kqueue had a bug when used on terminals (some missing wakeup calls). OpenBSD also lacked support for FIONREAD which is used by libevent.
- OS X kqueue and poll do not support anything except socket file descriptors.

To work around this, the portable version of tmux forces libevent to revert to poll or select on platforms with known problems.

The imsg Framework

The OpenBSD imsg framework[5] is a small API intended to provide reliable and secure IPC over Unix domain sockets. First written for the OpenBSD bgpd program and later moved to libutil, it is widely used by OpenBSD privilege separated daemons. tmux switched to imsg for message passing between client and server after import into OpenBSD for several reasons:

- Code reduction. imsg allowed custom code for IPC to be eliminated.
- Reliability. imsg is well established and tested.
- A simple API for file descriptor passing.

Portability

As well as OpenBSD, tmux also runs on Linux, Solaris, FreeBSD and NetBSD, HP-UX and AIX. A portable CVS tree is maintained on SourceForge and synced regularly with the OpenBSD tree by a long time tmux contributor (Tiago Cunha). SourceForge also hosts the portable tmux mailing lists and file repository.

tmux has not presented major portability challenges so far - the codebase is relatively small and has dependencies which are themselves portable (such as libevent) or small and easy to emulate (such as the strlcpy and asprintf functions). Even pty allocation has not been a major problem - the *BSDs and Linux include the forkpty function and this is easily emulated on other platforms (/dev/ptm on HP-UX and Solaris and /dev/ptc on AIX).

Up until early 2010 the tmux portable build system was a pair of custom makefiles, one for GNU make and one for BSD make, and a custom shell script for configuration. This was replaced by autoconf, mainly as providing portibility and the build interface that users expect was becoming complex with custom scripts.

The portable tmux contains about 4000 lines of compatibility code, most of which was copied from OpenBSD or portable OpenSSH with minimal changes, and about 700 lines of the build system.

Specific Design Examples

This section focuses on the design and implementation of a few notable tmux components or features.

Commands

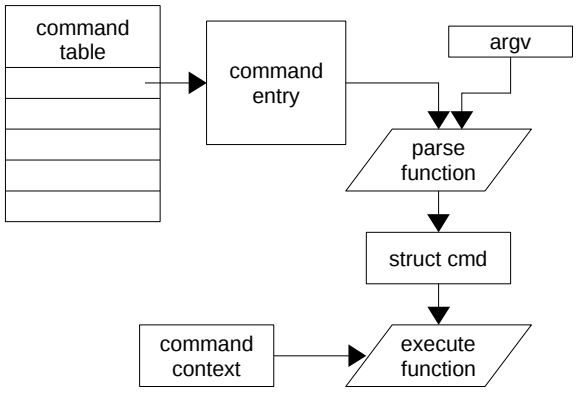
tmux provides a large command set. Commands are modelled after the Unix command syntax, for example:

```
tmux new-window -d "emacs a_file"
```

Most commands have both a verbose name (such as "new-window") and a shorter alias ("neww").

In early tmux versions, commands were matched using a series of string comparisons; however, it quickly became clear that this did not scale well to the number of commands being added and a new approach was required. The principles of this new approach were to maximise code reuse and as far as possible enforce user interface consistency - the same code should be used to parse and execute commands no matter which mechanism was used to activate them (be it key binding, the shell, a configuration file, or another).

An object oriented approach was developed: each command has a descriptor which includes the name, alias, some flags and a set of function pointers. The descriptors are stored together in a lookup table; each points to a set of functions specific to the command. When a command is processed, the descriptor and argument vector are passed into a factory method which parses the arguments and allocates and returns a *struct cmd*. The struct cmd contains all the information necessary to execute a command - it is an instance of the command. Separating parsing and execution permits validation of syntax to take place before the command is executed, and allows a lot of common parsing code to be reused.



Command execution process

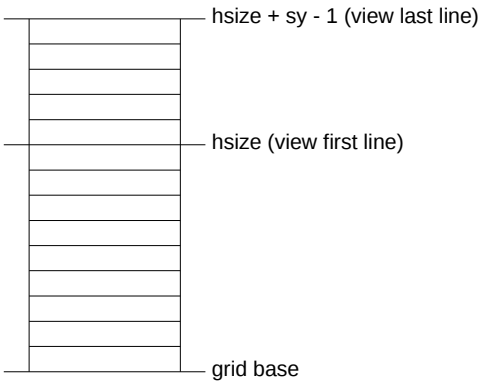
Commands are executed by calling the execute function in their descriptor with the previous created struct cmd and a *command context*. This context provides the environment in which a command executes, including the client which triggered the command and a set of functions which may be used by the command to output information or errors. These functions allow, for example, a command triggered from a key

binding to report errors in the tmux status line, while the same command issued from the shell prints them to stderr.

Screens and UTF-8

A screen is a critical data structure in tmux. It is a representation of a terminal state in a form which can be redrawn onto another terminal. A screen is made of ancillary information (cursor and scroll region position, tab indexes) and a rectangular set of cells in a *grid* structure organised as an array of lines. To minimise memory use, the size is stored with each line and it is expanded only as far as the last used cell. Each grid cell consists of five bytes containing the cell attributes, flags, colours and data.

Each grid structure is split into a visible portion - the *view* - and a *history* portion.



Grid data layout

This layout permits efficient scrolling and for the same primitive operations to be used on lines in both the history and the view. Operations on the screen are largely encapsulated into primitive operations on the grid: line scroll, move, insertion, deletion, and so on.

Implementation of UTF-8 presented some problems for efficient storage of the screen. UTF-8 characters are multibyte and vary in length, but maintaining variable length data in grid lines would introduce considerable code complexity. Equally, extending the data member of each grid cell would increase memory consumption for non-UTF-8 users.

The chosen solution was to introduce a parallel grid for UTF-8 characters. Each cell in the UTF-8 grid consists of a byte containing the width of the UTF-8 character and a nine byte char array, sufficient to contain UTF-8 sequences for any combined character made up of three-byte UTF-8 characters.

This solution introduces no additional overhead for non-UTF-8 terminals but chooses efficiency and code simplicity over memory use for UTF-8 terminals. For example, although the nine bytes per cell seems considerable, it allows cells to be allocated without complex variable size logic, simplifies the code for combined characters (an append is all that is necessary) and allows complete combined characters to be written to a terminal as one.

Input Parser

tmux uses a parser based on the state machine designed by Paul Williams[4], with alterations to limit it to 7-bit only, to support UTF-8, and for some minor quirks seen in modern terminal emulators. This replaced the original parser used in early versions due to concerns about the correctness of corner cases.

The purpose of the tmux input parser is to take a stream of data in a form described by the "screen" terminfo(5) entry and correctly update both a pane's screen and any clients on what the pane is visible. tmux reuses the "screen" terminfo(5) entry as there is limited value in creating a custom entry and there would be considerable lag before it was available on all supported platforms.

Each pane stores context associated with the parser and as data is read from the pseudoterminal, the state machine is transitioned through a set of state tables to end up at dispatch functions for C0 and C1 control sequences, CSI escape sequences, the APC and OSI sequences, and UTF-8 character input. In most cases, these update some state on the screen which then forwards as necessary to a tty command function which updates client terminals as necessary.

The tty command functions use the terminfo(5) sequences from the tty_term structure to update the terminal. Where sequences are not available, they are emulated. For example, on terminals which do not support the insert character sequence, tmux emulates it by redrawing the entire terminal line from the saved screen data.

Background Jobs

tmux has support for executing arbitrary shell

commands. This is used for two purposes:

1. The if-shell and run-shell tmux commands, which allow the user to execute a shell command.
2. The tmux status line and some other strings allow insertion of shell command result with a #() sequence.

When this was first implemented, the tmux server would block until the shell command completed. This had a serious problem: if the command itself executed tmux (creating a client), the blocked server could not respond to allow the new client to exit, leading to deadlock.

To solve this problem, the concept of background jobs was introduced. A job is a container for a shell command and an event. When the job is run, the shell command is forked and allowed to run in a separate process. If a SIGCHLD is received, the jobs list is searched for a matching process: if found, the job is complete and an internal callback attached to the job is fired. Similarly, if the event shows that the job's stdout has closed, it is treated as complete.

Although this permits tmux commands to be issued from jobs without blocking the server (an important ability), it introduces an unfortunate side effects: unintuitive behaviour when used from the configuration file. For example, a user might expect the sequence:

```
if-shell 'true' set -g default-terminal "foo"
new-window
```

To change the default-terminal option before creating a new window. However, as jobs execute asynchronously, there is no guarantee the "true" command will be complete and the "set" command executed before the configuration file parser has proceeded to the next line and parsed the next window.

Automatic Rename

Automatic window renaming is a tmux feature which uses uses a platform-specific method to establish the name of the process running in each window. Compatibility wrappers aside, this is the one piece of operating system dependent code in tmux.

On *BSD and OS X tmux uses sysctl to retrieve the list of processes associated with a terminal and a set of heuristics to work out the currently active process. On Linux and Solaris it reads the name or command line of the process directly from /proc.

This feature is more reliable than parsing terminal output to guess the command the user is running and has the advantage that it works even for processes not started from a shell prompt. However, it has the disadvantage of not following remote processes, such as after ssh. In practice this has not proven to be a major source of complaint.

Future Work

After three years of work, tmux is a mature and stable piece of software in wide use both as part of OpenBSD and with the portable release.

However, there are a few major outstanding items:

1. *Panes are fixed to one window*

This was a design choice made to simplify the implementation of panes. However, this lack limits the usability and flexibility of panes and is frequently queried by users.

Solutions may be to collapse the pane and window structures into one, or to introduce an additional intermediate data structure between windows and panes analogous to the winlink structure between sessions and panes. However, neither of these are trivial changes and either would have implications for the command interface.

2. *Windows can't be bigger than smallest terminal*

tmux limits the displayed size of a window to the size of the smallest client attached to the session containing it - that is, it will never attempt to make a window smaller than can be displayed by all clients that could show it.

This requires smaller clients to be detached before a larger can use the entire screen - an irritation for users.

A superior solution would be to permit smaller clients to display only a section of a larger window and to pan with the cursor.

3. *UTF-8 improvements*

Although tmux supports UTF-8 terminals, its

UTF-8 support is still primitive in some other areas, particularly of the user interface.

For example, UTF-8 support is not well supported by the interactive command prompt and copy and paste functions.

4. *More...*

As of late 2010, there are still 130 items, large and small, on the tmux todo list, so it is certain there will be work to do well into the future.

Notes

[1] GNU screen is available from http://www.gnu.org/software/screen/.

[2] See the GNU screen change log at http://git.savannah.gnu.org/cgit/screen.git/tree/src/ChangeLog.

[3] The libevent site is at http://monkey.org/~provos/libevent/.

[4] Full details of Paul Williams' parser are at http://vt100.net/emu/dec_ansi_parser.

[5] For information see the imsg_init man page, available online at http://www.openbsd.org/cgi-bin/man.cgi?query=imsg_init.