tmate: A scalable UNIX terminal sharing/management tool

Paper # 238

1 Abstract

tmate is a remote terminal application that supports easy session sharing, NAT traversal, and both SSH and HTML5 accesses. It augments the tmux terminal multiplexer and extend its features to a cloud based service. We made our geo distributed deployemnt a public service. It coordinatates more than 4000 paired sessions per month.

2 Introduction

Screen sharing technologies and remote desktop applications have been developed in the past two decades to provide numerous use cases, ranging from demoing a product, to performing troubleshooting. In the UNIX world, remote terminal applications have been around for much longer than that, the most popular being Secure Shell (SSH). However, SSH do not allow an easy way to share the same terminal with other parties. Terminal multiplexers such as tmux [**?**] provide terminal sharing capabilities, but lack remote connectivity features leaving users having to setup SSH tunnels.

This paper describes tmate, a system that allow users to export UNIX terminals into the web seamlessly. tmate solves the problem of remotely sharing terminals elegantly by offering both HTML5 browser and SSH accesses to a given terminal. The predominant usecases of terminal sharing is similar to the screen sharing ones, namely, doing pair-programming, troubleshooting issues, or managing servers from a remote location. Without tmate, users typically do the following steps: 1)

1) help program maintainers troubleshoot non-reproducible issues.

It happened many times that when someone reports an issue on one of my open-source project, I cannot reproduce the bug locally. In this case, I ask the user to install tmate and share their terminal with me which allows me to quickly jump in their environment and quickly resolve the issue. Often, resolving the issue would have been much more difficult and time consuming if done through standard communication channels as it would not allow interactive debugging.

I actually found that people would be much more comfortable sharing their terminal and allow me to see proprietary source code they are working, than sharing their code directly for IP/NDA reasons.

2) remote pair-programing

a) Remote pair programming can be difficult to set up. It often involves dealing with creating local accounts, managing SSH keys, port forwarding router configuration. This is cumbersome to manage. tmate allow terminal sharing to a friend in a few keystrokes (8 to be exact), providing that SSH connections are allowed to the outside world (which is most likely true for a developer).

Right now, tmate is seeing 1000 sessions per week. Most of them are seen Mon-Fri, 10am-6pm.

b) tmate could tunnel tcp/udp traffic, to expose local http servers so the remote pair could access the host http servers through its own browser. pairing extensions could be made in the future to synchronize browsers.

c) Perhaps tmate could provide integration with google hangouts / skype.

d) since tmate is a fork of tmux, and tmux allow splitting the terminal in multiple panes, tmate allow it too. in the future, we might want to allow ”rogue mode” where two different user can type in two different panes at the same time without interfearing with each other. This could be also useful to implement some sort of chat window.

3) server management and administration

In small and medium sized startups, sysadmins have the task of overseeing all the production servers. tmate allows the sysadmin to manage terminals in its fleet of servers. There are a few useful use cases for the sysadmins:

a) First, by ensuring that all developers only go through tmate when accessing a terminal in production, tmate can record who did what on which server accurately. This complete auditing feature is not available in other systems. This feature can be useful in critical situations. For example, when the production system encounters a failure, down, the sysadmin has to fix the issue and bring the system back online as soon as possible. In its debugging task, the sysadmin often needs to have visibility over recent human intervention and see at a glance who did what.

b) tmate can act as an initd daemon to launch all services on the production servers. Typically, services (databases, web servers, etc.) can log their output to STDOUT. Each service would be spawned in an interactive terminal on boot. This way, tmate can track the log output of all services, avoiding the need for sysadmins to ship the logs to some other server. tmate can provide all features 3rd party logging systems can do, while providing a way for sysadmins to intervene directly in terminals, and inspect, for example the file system. A sysadmin may also triggers actions against specific regular expressions when seen on the output.

c) Further, tmate could detect applications and parse their output content. For example, suppose you have three mongodb instances, each running in their respective tmate terminal. The tmate dashboard (through the tmate browser http interface) can display custom styling for the terminal thumbails. e.g. green borders for the primary database, and yellow borders for the secondary databases. We can even do heavy terminal interpretation. for example, if tmate detects top/htop running in the terminal, it may provides cpu/mem graphs, and provide history of the resource utilization of the machine, which can be useful for aggregating data on a large amount of machines. These visual queues allow sysadmins and developers to quickly know which terminal to go to when problems arise in their system allowing them to take action faster.

4) augment terminals on personal computers

a) tmate provides a recording of all terminal input/output on the local machine. This is great for a number of reasons: 1) The user can go back in time to see exactly how he performed a certain task. Looking at the bash history is often not good enough, esp. when files are edited. tmate can provide full-text search to help users locate certain moment in their terminal history. tmate can also provide statistics over terminal usage and hints the user with useful command aliases and tips to improve workflow.

b) tmate can interpret the terminals and the inputs. with the use of community provided plugins, tmate could suggest solutions for the issues someone may be facing. For example, when seeing certain error messages like ”Error: Unknown method abc() in module Blah”, tmate could match this error and suggest a workaround based on stackoverflow, or whatever database user may code. directly at the location of the error. tmate could also (but we would not do it because it's a little evil) provide targeted ads directly to developers. For example, if tmate detects a fresh new ruby-on-rails project, it could suggest the user to install newrelic, heroku, etc. in the status bar.

2.1 User Experience and Challenges

When designing tmate, we were facing the following challenges:

1) Getting the pairing setup user experience right. - We want to get users (hosts) to share terminals with a minimal amount of steps and keystrokes. - We want other users (friends) to connect to shared terminals without having to install any additional clients. With these constrains, we want to maximize the amount of security we can have. This means that friends must use SSH or a web browser with HTTPS. This also means that if the host is behind a NAT, we somehow need to traverse that when clients connect to the host. We deal with this by having an external server to connect the two parties.

We got the current workflow down to three steps: a0) (install, one time only) run ”brew install tmate” a1) run ”tmate” b) copy/paste the connection string to the friend c) which he then copy/paste in his terminal or web browser

2) Get the highest availability we can. tmate should always start whether the connection is slow, interrupted, or disconnected. Whatever happens in the terminal should be unaffected by the network connection. Whenever a disconnection happen (tmate -¿ remote tmate, or remote tmate -¿ proxy, or proxy -¿ master), things should reconnect automatically and sync back. Right now, we use a postgres DB, which is a single point of failure, but if it fails, the tmate service degrades well enough. The html5 connections would go out of service, but the SSH connections remain available.

3) Get the lowest latency possible, and uninterrupted service To get the lowest latency possible, we provide servers across the globe. The ultimate performance would be to do peer to peer, which would be an avenue to explore if we could afford to have another client as SSH. Any server-side code upgrade should not disconnect any of the existing connections (hence erlang, and its hot code swap features). Any server-side code upgrade should maintain backward compatibility with older tmate clients (sometimes quite hard).

4) One of the decision that was difficult to make was to fork tmux, or do a standalone client. The reasons to use tmux were: a) It already provides an overlay on top of a terminal with status bars and everything. b) Users are familiar with tmux, it's a very popular tool. So if we

want to interpret terminals (and we do), it's easier to go in that direction. Note: the html5 interface already benefits from this decision as it understands the panes and windows semantics. Using a standalone client would not be able to interpret a tmux session running inside of it. The reasons to not use tmux were: a) the tmux codebase is large, and is difficult to understand. Synchronizing its state require a high level of C proficiency.

5) Provide privacy garantees and provide end-to-end encryption.

3 Related Work

In the environment sharing realm, numerous tools exists. For example rdesktop, VNC, and more comprehensive solutions like Citrix, WebEx, GoToMeeting, twitch.tv all provide desktop sharing capabilities. However all these tools more or less stream videos. Streaming videos may have high bandwidth requirements, and do not offer necessarily good quality for text only environments.

Other tools provide sharing capabilities such as vanilla tmux, or wemux, but require SSH setup for others on the host machine, and do not provide any solutions with respect to NAT traversal.

Others have provided HTML5 terminal sharing clients such as gotty, tty.js, wetty, or termshare. But these only function as a regular web server, and provide no NAT traversal capabiltiies, provide no high availability feature, nor dual SSH/HTML5 access.

- examples of web based terminals integration: * c9.io, nitrous.io, google cloud engine

- low latency ssh like: * mosh

- CI tools: * circle-ci, offers SSH access. * codeship, offers SSH access. * jenkins

terminals in the cloud: * auditd * bashhub * papertrail

4 Architecture

TODO diagram inside the servers with the different components

related work

On a high level, the architecture is the following: - Users run the tmate client (host) on their machine. tmate is a fork of tmux which is a terminal multiplexer. - The state of the tmate host is replicated to a remote tmate daemon on tmate.io. - The remote tmate is also a fork of tmux, and has been modified to accept SSH connections. - SSH clients connect to the remote tmate server, and since tmux is a terminal multiplexer, clients attach to the terminal as a tmux feature. However keystrokes are sent direcly to the tmate host, which then replicates its effects to the remote tmate daemon. - Additionally, the remote tmate daemon forwards the replication log stream to a daemon on the same machine, the proxy service. - The proxy implements websockets to serve html5 clients the replication stream - The pairs (remote tmate daemon, proxy) are geographically replicated to allow best latencies for clients. - The proxy sends events to a master service, which sits on top of a SQL database and serve the web interface. - The proxy also send events to user-defined webhooks. - The master is a traditional web server in the sense that it manages users, sends emails, etc. However, it does not serve HTML, but only provides JSON APIs. The web user-interface is a separate javascript frontend application that communicates with these APIs. - The web interface is a javascript application served from a CDN.

4.1 tmate client on host machine

The host is the machine that runs the actual programs in the terminals. On the host, the user launches tmate. Since tmate is a fork of tmux (terminal multiplexor), the same terminology and architecture applies. tmux itself follows a server-client architecture, which tmate reuses. - the local tmate server runs all the sessions. A session have many windows. A window have many panes. A pane is essentially a pty that runs a shell or some other program. - on the host, the user runs a local tmate client which connects to the local tmate server to display a single session through a UNIX socket. The client provides its terminal pty file descriptor to the server, which then directly renders on the client's screen. This is why the tmux client/server architecture is not compatible across an IP network.

tmate adds non trivial modifications to the tmux codebase. (43 files changed, 2894 insertions(+), 105 deletions(-)) tmate adds the following behavior to tmux: 1) when starting, tmate reads a local /.tmate.conf in addition to the local /.tmux.conf that allow users to specify which tmate server they want to replicate to. A custom replica host and its SSH fingerprints, and user-defined webhooks may be specified. 2) The replication stream buffer is initialized. We need to do this early because we do not block on the connection to be established before allowing terminal commands to run. Further, the replica needs all the tmux commands specified in the configration files to be replicated, which is done before connecting to the replica. 3) Once the local tmate server is running, it connects to each of the IP that ssh.tmate.io resolves to (or the user configured host). The connection follows the SSH protocol through libssh. This allow authentication of both the server and the clients, provides encryption, and channel windowing. If the user has ssh keys that need a passphrase, the user is asked for the passphrase in the bottom of the terminal. 4) Once the local tmate server has completed an SSH handshake with one of the replica server, that replica is elected for being the replica to be used, and all the other pending connections to other IPs are killed. This is useful for a) fault tolerance and b) selecting the remote tmate server with the lowest latency. 5) Once a successful SSH connection has been established to a remote tmate, the remote tmate gets assigned a session token and forwards a connection string to the local tmate (e.g. ssh SESSION_TOKEN@ny.tmate.io), which is displayed in the status bar.

4.2 remote tmate replica

The original tmux codebase is modified to accomodate our needs (59 files changed, 4082 insertions(+), 186 deletions(-))

The remote tmate daemon accepts connections on the SSH port. Each new SSH connection gets forked into a new process. A connection to the tmate proxy is established right away. If this connection fails, the process dies. This abords the SSH handshake, which is important for the tmate client so it continues trying other servers. If the SSH handshake has not completed after 60 seconds, the process dies. Note that the forked processes create their own process group with setpgid(0,0) to allow the parent to be restarted through upstart without killing its children. We force compression on the SSH protocol to save bandwidth, as text editors such as VIM can be very verbose. After doing the SSH handshake, three things may happen: a) Either the SSH client asks for the tmate subsystem, which would be a local tmate client asking for replicating its state. b) Or the SSH client requests a pty and a shell, in which case this would be a request to attach to an existing remote tmate server. c) Or the SSH client requests to execute a command, in which case the command to be executed is forwarded to the proxy.

4.2.1 SSH tmate subsystem

a) In the case the SSH client asks for the tmate subsystem: 2) a session_token and a read-only session token are generated through /dev/urandom. These tokens are comprised of 25 alphanumeric characters. 3) The tmux server is set to operate on the /tmp/tmate/¡session_token¿.sock unix socket, and a symbolic link of the name /tmp/tmate/¡ro-session_token¿.sock is created to point to the real unix socket. 4) Because the tmux codebase is large (45,000 LoC) we have to assume that an attacker can exploit bugs in the tmux codebase to gain arbitrary code execution. To alleviate this issue, we take a few precautions by isolating the tmate forked process into a tight jail. The steps are the following: a) All necessary file descriptors are opened (e.g. ncurses terminal info, tmux unix socket, log file) b) get into an empty chroot. c) we enter a pid,ipc,network namespace. d) and finally the uid,gid is switched to nobody The network namespace is important as we do not want the attacker to connect to various of our services and databases. Note that the attacker can still communicate with the proxy as he has full access to the proxy connection. 5) the tmux server is finally started and replicates the host tmate state.

4.2.2 SSH pty request

b) In the case the SSH client requests a pty: 1) the username is the session token. The corresponding UNIX socket is opened in /tmp/tmate/¡session_token¿.sock. If it is a symbolic link (through stat()), the client is set to be a read only client (all of its inputs are discarded). Note that if the socket cannot be found, a random sleep is inserted to make timing attacks harder. 2) The process is jailed 3) A ”tmux attach” command is performed on the tmux socket, and the ssh pty is given to the remote tmate server for rendering. 3) pressed keys are interpreted by the remote tmate server. Key bindings are the same as the local tmate server as we replicated the tmux configuration. These key bindings triggers tmux commands (e.g. ”split-window -v”), which are forwarded to the tmate host. 4) client pty sizes are also forwarded to the local tmate server. This way, the local tmate server can choose an appropriate window size that suit all clients.

In the future, we want to also accept mosh clients. This would allow slow clients to join a session. However, this would not make the local to remote tmate connection tolerance for slow network.

4.2.3 SSH exec

c) In the case the SSH client requests a command execution: The command the client runs is typically of the form ”ssh ssh.tmate.io identify TOKEN”. That command is an example of how a user would login on the web interface by using his SSH credentials. 1) The command and its arguments are sent to the proxy 2) When the proxy returns a message and an exit code, they are forwarded to the STDOUT and exit status of the SSH client.

the local and remote tmate servers are coded in C, and use the reactor pattern with libevent.

4.3 Protocol between the host and the replica

The communication between the tmate host and the tmate replica uses msgpack, an encapsulation format similar to JSON, but a little faster and more compact as it’s all binary.

The host sends a stream of messages to the replica. These messages are defined as follow:

4.3.1 host to replica

* `[OUT_HEADER, int: proto_version, string: version]` This is the first message sent to the replica. This announces the current protocol version, and the client version, useful as we want to maintain backward compatiblity against old clients.

* `[OUT_SYNC_LAYOUT, [int: sx, int: sy, [[int: win_id, string: win_name, [[int: pane_id, int: sx, int: sy, int: xoff, int: yoff], ...], int: active_pane_id], ...], int: active_win_id]` This message sends the current layout of the session. It sends the current terminal size, and all the attributes of the windows. For each window, the window title is serialized, and a list of panes is sent. For each pane, we send the cursor position, and the position within the window. When receiving such message, the replica figures out which window, or pane, that needs to be added or removed. Note: this may be CPU intensive as we get a full layout sync whenever a window title changes, which happens everytime a shell command is run (the window title takes the shell command that is being run), so executing a trivial command like "ls" would trigger two layout sync.

* `[OUT_PTY_DATA, int: pane_id, binary: buffer]` This message indicates new data to be displayed on a pane. The data is raw and non interpreted.

* `[OUT_EXEC_CMD_STR, string: cmd]` This message instructs the replica to run a specific command (e.g. `bind -n M- split-window -h -c "#pane_current_path"`) Note that this is a legacy message. New version use `OUT_EXEC_CMD`.

* `[OUT_EXEC_CMD, string: cmd_name, ...string: args]` Same as above, except that we have the arguments parsed properly.

* `[OUT_FAILED_CMD, int: client_id, string: cause]` This message is used as a reply for `IN_EXEC_CMD_STR`.

* `[OUT_STATUS, string: left, string: right]` This message sets the current status bar.

* `[OUT_SYNC_COPY_MODE, int: pane_id, [int: backing, int: oy, int: cx, int: cy, [int: selx, int: sely, int: flags], [int: type, string: input_prompt, string: input_str]]]` This message synchronizes the current copy mode. This is done when the user scrolls within a pane, selects text, or search for some text within a pane.

* `[OUT_WRITE_COPY_MODE, int: pane_id, string: str]` Not sure what this one does anymore.

* `[OUT_FIN]` Seals the session. That means that we won't accept reconnections.

* `[OUT_READY]` Ready means that the initialization phase is complete, and the configuration files have been processed. This is a trigger for the proxy to register the session and trigger webhooks. (Since webhooks are configurable, it's important to wait that we have all the configuration files processed, which we replicate with `OUT_EXEC_CMD`)

* `[OUT_RECONNECT, string: reconnection_data]` Sent right after the `OUT_HEADER`. The reconnection_data is an opaque string passed from the proxy.

* `[TMATE_CTL_SNAPSHOT, [[int: pane_id, [int: cur_x, int: cur_y], int: mode, [[string: line_utf8, [int: char_attr, ...]], ...], ...], ...]]` Sent right after the `OUT_RECONNECT` message. The snapshot contains each pane's content. For each pane, we send each parsed lines in utf8, each with all their attributes (color, blink, etc.). A configurable amount of history can be retreived for each pane, to allow scrolling up. We also send the alternate vt200 screen.

4.3.2 replica to host

The replica sends the following messages to the host:

* `[IN_NOTIFY, string: msg]` This prints the message to the status bar so we can communicate with users. All these messages are retrievable by running "tmate show-messages" in a shell. For example:

```
$ tmate show-messages Connecting
to ssh.tmate.io... Note: clear your
terminal before sharing readonly
access web session read only:
https://tmate.io/t/ro-hoh9SSovQ8zlc0AAxfOkVZGZu
ssh session read only: ssh
ro-hoh9SSovQ8zlc0AAxfOkVZGZu@ny2.tmate.io
web session: https://tmate.io/t/5F1caD9FOvYCD6Nk3cQpkI40
ssh session: ssh 5F1caD9FOvYCD6Nk3cQpkI40Q@ny2.tmate.io
```

* `[IN_PANE_KEY, int: key]` This instructs the host to direct a keystroke to the active pane.

* `[IN_PANE_KEY, int: pane_id, uint64 keycode]` Same as above, except the target pane is specified, and the keycode is parsed to the new tmux format. Note: Mouse support is experimental.

* `[IN_RESIZE, int: sx, int: sy]` This instructs the host to resize its window to a specified size.

* `[IN_EXEC_CMD_STR, int: client_id, string: cmd]` This instructs the host to run a specific command. (e.g. $split-window-h$)

* `[IN_EXEC_CMD, int: client_id, ...string: args]` Same as above, except the arguments are split into an array.

* `[IN_SET_ENV, string: name, string:`

value] Used to send environment variables to the host. Currently, the following environment variables are defined: tmate_ssh, tmate_ssh_ro, tmate_web, tmate_web_ro: each of these variable contain the connection string to connect to the remote session either via ssh, or the web interface. Further, the variable reconnection_data may be set, which enables the reconnection protocol. On the host, the user can read the value of a variable with such shell command: $ tmate display -p '#tmate_ssh'

* [IN_READY] This instruct the host that the environement variables are set. The user can run the following command to wait for this message: $ tmate wait tmate-ready This way, the user can wait before printing the environment variables.

4.4 Proxy

We mentioned earlier that the remote tmate server connects to a proxy. The proxy is coded in Elixir, and runs on the Erlang VM. The proxy lies on the same machine to reduce latencies, but serve different functions: a) provide websocket access to terminal sessions for HTML5 clients b) emit events to the master (we'll talk about the master later) c) do RPC calls to the master due to SSH exec calls d) send webhooks events e) handle reconnections f) notify the host of any clients joining or leaving the session g) gather latency measurements

When a tmate replica connects to the proxy, the proxy instantiates a new Erlang process. This process is isolated from the other processes on the VM. During its lifetime, the proxy receives a copy of the stream that the host sends to the replica. The initialization process of the session process is the following:

4.5 when the replica connects to the proxy

1) Before accepting the TCP connection from the replica, we ping the master server. If the ping fails, then the connection is refused, which in turns rejects the connection to the tmate host. This is required due to our high availability protocol. 2) Session parameters from the replica are received, including the session tokens, and the public key of the host. 3) tmux commands from the host configuration files are received and parsed to locate any webhook configurations. 4) Once the OUT_READY message is received: a) If we got any reconnection data, we proceed with the reconnection protocol (more details below). b) the session is registered to the local session registery (another erlang process), which is essentially a map from the session tokens to the pid of the session. c) The webhook process is started with the configured webhooks. d) the environment variables (ssh_cmd, ssh_cmd_ro, etc.) are sent to the host and the user notified with connection messages. e) The session_open event is emitted to the master and to the webhooks. f) We notify the user to upgrade his client if it's outdated.

4.6 when an HTML5 client connects to the proxy

When an HTML5 client connects to the proxy, it does the following: 1) Dispatch the HTTP request through the cowboy library (an HTTP server). 2) For the route /ws/session/:stoken, we lookup the provided session token from the local session registery. When found, the connection is upgraded to a websocket and the session process is monitored for errors. 3) The session cookie containing the user_id is decrypted and validated. 4) Instruct the session process to request a snapshot to the remote tmate server. A snapshot contains, for each panes: a list of all the rows in utf8 with the attribute of each character (fg/bg color, bold, blink, reverse), the cursor position, and the terminal mode (cursor hide/show, vt200 mouse enabled, etc.) The snapshot sends not only the visible lines of each panes, but 300 (configurable) lines of history per pane. This allow the HTML5 client to scroll up, and see some history. 5) Once the session process receives a snapshot, it sends to the websocket the latest known layout, the snapshot, and then register the snapshot to receive all pane data as a stream. This is done atomically: once the websocket receives a pty data for a pane, the pane state must have been restored to correspond to that point in time. 6) And then the websocket gets a copy of the data as the remote slave receives it.

4.7 when clients join and leave a session

When SSH clients join or leave the session on the tmate replica, the proxy is notified. This allow the proxy to maintain a list of connected clients. The proxy session process also tracks websocket clients. When a client join or leave the session, the master is notified via session_join and session_left events respectively. These events are also sent to webhooks. Each client screen size is tracked so that when any screen size change, the proxy can compute the minimum size and send it to the tmate host.

4.8 Webhooks

Webhooks allows service integration with tmate.io. When one of those events is triggered, a HTTP POST JSON payload is sent to the webhook's configured URL. The webhooks are configured via two configuration directive: tmate-webhook-url and tmate-webhook-userdata. The url corresponds to the endpoint tmate should send events to, and the userdata value gets to be included in every event payload.

All events are sent with the following generic payload structure:

{ type: "event_type", entity_id: "d8d1cbb2-f5d6-11e5-b8f0-888888888788", userdata: "some private data", params: ..., }

The type denotes the event type (e.g. session_open), entity_id is the corresponding event entity (e.g. the session_id), userdata corresponds to the string from the /.tmate.conf file, and params is a hash containing additional event data.

For each event, the proxy sends the JSON payload to the configured endpoint, and expects a 2xx HTTP response code. On failure, it will retry sending that event 5 times maximum, sleeping 3 seconds between tries. If the maximum amount of tries is reached, the failed event is discarded, and the next event is sent over.

The following describes the different events.

4.8.1 session_open

{ type: "session_open", userdata: "some private data", entity_id: "d8d1cbb2-f5d6-11e5-b8f0-888888888788", params: { stoken: "gX5RFpICOE9n0tLMDOWDrOQeO", stoken_ro: "ro-QX35Q8ukrxz2MlDz7PewujUQ9", ssh_cmd_fmt: "ssh web_url_fmt: "https://tmate.io/t/reconnected: false, pubkey: "AAAAE2VjZHNhLXN...", ip_address: "74.64.123.124", client_version: "2.2.1" } }

The session_open event is sent when a new session is created or when a session has reconnected after a network failure. entity_id is the session id. stoken and stoken_ro are the read/write and readonly session tokens. ssh_cmd_fmt and web_url_fmt are the connection strings where %s must be replaced with stoken or stoken_ro depending on the desired access. reconnected is a boolean to denote if the session was created, or got reconnected. In the case of a reconnection, The entity_id will have the same value as the previously received session_open event. However, the connection strings may be different as the SSH server might be different. However the tokens will remain the same. You may assume all connected clients (see below) have left the session when receiving such reconnection event. pubkey is the SSH public key of the tmate host. ip_address is the IP address of the tmate host. client_version is the client version of the tmate host.

4.8.2 session_close

{ type: "session_close", userdata: "some private data", entity_id: "d8d1cbb2-f5d6-11e5-b8f0-888888888788", params: {} }

The session_close event is sent when the host closes the session. No reconnection may be expected at this point. * entity_id is the session id.

* session_join { type: "session_join", userdata: "some private data", entity_id: "d8d1cbb2-f5d6-11e5-b8f0-888888888788", params: { id: "76ee3600-f5dc-11e5-a64d-04018f4b2301" readonly: false, type: "web", identity: "76ee0360-f5dc-11e5-bed2-04018f4b2301", ip_address: "74.64.126.154", } }

The session_join event is sent when a client connects to a tmate session. * entity_id is the session id. * id is the client id, a UUID corresponding to that client connection. * readonly is true when the client connected via the readonly token. * type can be either ssh or web. * identity is the SSH public key of the client when type is ssh. If type is web, the identity correspond to a meaningless UUID. * ip_address is the IP address of the client.

4.8.3 session_left

{ type: "session_left", userdata: "some private data", entity_id: "d8d1cbb2-f5d6-11e5-b8f0-888888888788", params: { id: "76ee3600-f5dc-11e5-a64d-04018f4b2301" } }

The session_left event is sent when a client disconnects from a tmate session. * entity_id is the session id. * id is the client id.

4.8.4 session_stats

{ type: "session_stats", userdata: "sometoken", entity_id: "edd54332-f5da-11e5-b01e-04018f4b2301" params: { id: "76ee3600-f5dc-11e5-a64d-04018f4b2301" latency: { n: 53, mean: 26.11320754716981, stddev: 3.3262382157076864, median: 25, p90: 31, p99: 35 } } }

The session_stats event is sent to provide aggregate statistics on service quality for each client. * entity_id is the session id. * id is the client id. If id is null, the statistics corresponds to the tmate host. * latency gives aggregate statistics on latency, measured every 10 seconds by sending a ping packet to the clients and host. If the client id is not null, the latency corresponds to the end-to-end latency, that is client `latency + host latency`, which is the perceived latency by the client. If the client id is `null`, the latency corresponds only to the host latency. * `n` is the number of samples acquired. Samples are taken every 10s. * `mean`, `stddev`, `median`, `p90` and `p99` are the mean, standard deviation, median, 90th percentile, and 99th percentile.

4.9 Latency measurements

To monitor the quality of service, tmate measures the latency of its clients. The latency is measured for the three kind of clients: 1) the tmate host 2) the SSH pty clients 3) the websocket clients. Both 1) and 2) are using an SSH connection. To measure the latency, we send keepalives by sending global requests with "keepalive@openssh.com" as data with a request for reply. Different clients reply differently. libssh (1) sends an unimplemented packet reply to keepalive requests and other clients like openssh (2) sends a request denied packet to keepalive requests. By measuring the time difference from the keepalive request and the reply, we are able to measure the latency of the client. With websocket, it's easier as the websocket protocol includes a PING/PONG protocol. We measure the latency of clients every 10 seconds. The proxy aggregates these latencies for each client to compute the mean, standard deviation, median, 90th percentile, and 99th percentile. The percentile are computed efficiently with a tree.

4.10 Protocol

4.10.1 Replica to Proxy

The following describes the messages send from the replica to the proxy.

* `[CTL_OUT_HEADER, int: ctl_proto_version, string: ip_address, string: pubkey, string: session_token, string: session_token_ro, string: ssh_cmd_fmt] string: client_version, int: client_protocol_version]` This message is the first message the replica sends to the proxy.

* `[CTL_OUT_DEAMON_OUT_MSG, object: msg]` This message is a wrapper around the stream of messages the host sends to the replica.

* `[CTL_OUT_SNAPSHOT, [[int: pane_id, [int: cur_x, int: cur_y], int: mode, [[string: line_utf8, [int: char_attr, ...]], ...], ...], ...]]` This message represents a terminal snapshot.

* `[CTL_OUT_CLIENT_JOIN, int: client_id, string: ip_address, string: pubkey, boolean: readonly]` This message indicates that a SSH client has joined the session.

* `[CTL_OUT_CLIENT_LEFT, int: client_id]` This message indicates that a SSH client has left the session.

* `[CTL_OUT_EXEC, string: username, string: ip_address, string: pubkey, string: command]` This message is sent when a SSH client request an SSH command to be executed.

* `[CTL_OUT_LATENCY, int: client_id, int: latency_ms] // client_id == -1: tmate host` This message is sent to report latencies.

4.10.2 Proxy to Replica

* `[CTL_IN_DEAMON_FWD_MSG, object: msg]` This message tells the replica to forward the msg to the host.

* `[CTL_IN_REQUEST_SNAPSHOT, int: max_history_lines]` This message instruct the replica to generate a snapshot.

* `[CTL_IN_PANE_KEYS, int: pane_id, string: keys]` This message sends a string to a specfic pane. The replica translate this into the relevent keystokes to the host.

* `[CTL_IN_RESIZE, int: sx, int: sy] // sx == -1: no clients` Indicates the desired terminal size.

* `[CTL_IN_EXEC_RESPONSE, int: exit_code, string: message]` Sends a reply to the SSH client requesting an command exec.

* `[CTL_IN_RENAME_SESSION, string: stoken, string: stoken_ro]` Rename the session name after reconnecting (visible on the $/proc/self/cmdline$).

4.10.3 Proxy to HTML5

* `[WS_OUT_DAEMON_OUT_MSG, object: msg]` This message is a wrapper around the host stream.

* `[WS_OUT_SNAPSHOT, [[int: pane_id, [int: cur_x, int: cur_y], int: mode, [[string: line_utf8, [int: char_attr, ...]], ...], ...], ...]]` This message contains the terminal snapshot. Proper care

4.10.4 HTML5 to Proxy

* `[WS_IN_PANE_KEYS, int: pane_id, string: keys]` This message instruct the proxy to send the specified keys in a specific pane.

* `[WS_IN_EXEC_CMD]` This message instruct the proxy to send the host a command to execute.

* `[WS_IN_RESIZE]` This message instruct the proxy the current window size.

4.11 Reconnection protocol

Tmate supports a reconnection protocol from the host to the replica in case of network failures. We want the host to be able to reconnect to a different replica server, in case some tmate.io servers go down. However, we do not want users to be able to forge a session token. Thus, host must be able to prove that the session token they are trying to reconnect to is legitimate. To this end, the proxy server packs the session id, the session token, and the readonly session token. It then uses an HMAC to sign the data, and sends it to the host as an environment variable. When the host detects a connection failure (with a TCP keepalive failure for example), it tries to reconnect to a replica. Once the connection is successful, it sends its header, reconnection data, all tmux commands that are useful for the state replication, a layout sync, and a snapshot of the current pane. The proxy gets that reconnection data, verify its integrity, and renames the tmux sockets accordingly as the replica server cannot rename sockets at this point since its in a tight jail.

4.12 High Availability

To maximize user experience, we ought to provide a high availability service. Our requirements are the following:

1) the tmate host local interactions should never be impaired due to any networking problems (e.g. slow, interruputed, choppy) regarding the tmate.io service. This is dealt with the design of the tmate host as all network related mechanisms are asynchronous to the terminals. There is no RPC towards tmate.io.

2) When a tmate.io server fails, the infrastructure should heal, and clients should experience a minimal interruption. This is dealt with various places: a) when the tmate host detects a connection problem towards the replica, the host initiates the reconnection protocol as explained above. b) when the tmate replica detects a connection problem towards the host or the proxy, the replica dies, killing any SSH clients, and disconnects abruptly from the proxy, in the hope that the host triggers the reconnection protocol. c) when the proxy detects an issue with the tmate replica, it shuts off, killing any HTML5 clients, and notify the master that the session is experiencing a loss of availability, in wait for the tmate host to initiates the reconnection protocol. d) when the proxy detects an issue with the master, the tmate.io runs in a degraded state: new HTML5 clients may no longer connect as the HTML5 clients rely on the master to get the URL of the proxy websocket. Note that the proxy does not do any RPC towards the master, but only emits events, so if the master is not responding, the proxy survives.

3) tmate.io should scale accordingly to maintain a good performance SLA. a) first, it would be a good thing to use cgroups to manage machine resources per session. This way, if there is a very noisy session, other sessions on the same host would remain unaffected. b) In case the number of connected clients become very large on a single session, we would have scaling issues with the current infrastructure. We would need to replicate the session to other machines. This is easy to do as we would just need to forward the replication log file.

4.13 Master

The master service is coded with elixir, using the phoenix framework (equivalent of ruby-on-rails in the Elixir world). The master is a traditional web application with a postgres database. The master is not geographically distributed as opposed to the other pieces in the system that we've seen so far.

The master service does not serve HTML, but only provides JSON APIs. The web user-interface is a separate javascript frontend application that communicates with these APIs.

The proxies communicate with the master service via the Erlang RPC mechanism. Some additional glue was put in place to retry requests in case of network failures, but essentially, the master service responds to the events that the proxies send, store them, and projects them into its postgresql database. The projections are coded in a way that tolerate duplicate events. For this, the master service leverages uniqueness indexes on postgresql to get clean $getorcreate$ semantics. For example, we store SSH identities in a table, each row has a unique public key that follow a singleton pattern.

TODO: Explain data structures that we use in postgresql.

4.14 Web Frontend

The frontend is coded with numerous plugins. First, we use babel to transpile ES7 javascript to regular javascript. We use bootstrap for the style. jQuery for AJAX helpers. We use React for composing our UI elements, with various plugins. To package all of our javascript, we use webpack, which allow very powerful workflow such as react component hot loader, or advanced javascript chunking to optimize load times client side.

To display the terminals, we use term.js, original work of Fabrice Bellard. We had to fix a few bugs and do some adaptation (focusing terminals needs to be synchronized across clients for example). Some control sequences were not properly recognized due to the terminal being a screen-256colors instead of an xterm. Mouse coordinates were not properly done.

The application uses react router to provide instant navigation in the application. When loading a

`/t/session_token` URL, The application renders a session react component. The life cycle of a session component goes as follow: First, a request to `/api/t/session_token` is sent to the master service, which replies with a websocket URL. A connection is then made to that websocket. The UI component then waits for the session layout and snapshot. Upon a layout sync, the appropriate window and pane UI component are created and destroyed via React engine. Each pane snapshot is routed to the approriate pane UI element, and translated accordingly to the internal representation of term.js. Once done, the session UI component enters its normal state, and recieved pty data is routed to the proper pane component, which feeds its term.js instance.

When the user presses keystrokes on the window, we intercept these keystroke and send them via the websocket. When the user clicks on a pane, the pane is selected on the host by sending a tmux command `select-pane -t #pane-id`. We could also implement pane resizing and window splitting in a similar fashion, but such buttons have yet to be implemented.

If the tmate host was to disconnect, the websocket connection would then close, and the HTML5 client would automatically reconnect seamlessly. A feature not possible with SSH.

4.14.1 Threat Model

We investigate two different thread models against different architectures.

The first threat model *T1* assumes an attacker that can perform man-in-the-middle attacks between tmate.io and the host, or clients. The attacker can also trigger an exploit in the tmux replica codebase. (But We assume a bugfree kernel, erlang vm, ssh implementation)

The second threat model *T2* assumes a malicious or compromised tmate.io service.

The question is: can the attacker gain access to an unauthorized session terminal and see some content of its panes?

1) The first architecture is the current architecture. Under T1, the attacker cannot gain access to unauthorized sessions. SSH/HTTPS provides authentication against MitM attacks, and the tmux replica codebase runs in a jail.

Under T2, the attacker can gain access.

2) The second architecture assumes that clients no longer connects via SSH, but a custom tmate remote viewer to perform end-to-end encryption. This custom viewer would be a client side tmate replica at its heart, while tmate.io acts as an evolved proxy. The tmate host would specify a secret key, unknown to tmate.io. Clients would be required to use this secret key, in addition to the usual session token. This secret key may be communicated via a side channel.

The host would send all messages encrypted with the secret key except for the following message: `OUT_HEADER`, `OUT_FIN`, `OUT_READY`, `OUT_RECONNECT`. All received messages the host would receive would be encrypted except for `IN_NOTIFY`, `IN_SET_ENV`, and `IN_READY`. This way, tmate.io can still operates correctly and provides all the usual features.

All messages would be encrypted with a counter encryption (CTR) and signed/verified with an HMAC.

Note that the tmate host may set, change, or unset the secret key at any time during a session. Users may go off-the-record for a period of time they chose without having to stop or restart a session.

The down side of this approach are: a) clients would be required to use a special client as opposed to using SSH. the HTML5 interface would require a custom extension not provided by tmate.io. b) No data can be used by tmate.io.

The good side of this approach are: a) privacy concerns are addressed under T1 and T2. b) maybe that could be a feature users would pay for.

4.14.2 User Authentication

tmate.io supports three ways to authenticate users:

1) OAuth through GitHub 2) Email 3) SSH exec command

TODO.

4.15 Data Flow

The system follows an event sourcing pattern. Similarly to Synapse, the system has publishers, a common data bus, and subscribers. Contrary to Synapse, the publishers to do have any database, they write directly events to the data bus, and views are projected accordingly. This has a number of benefits: - writes are highly availaible, scalable, and have low latency. - debuggability is improved as all application state mutation are persisted. - data loss is greatly reduced as the only possible write operation is ”insert”. The drawbacks of event sourcing are: - no guarantees about reading your own writes. Not sure how big of a problem this is yet. - we must use UUIDs everywhere.

4.16 Monitoring

For the monitoring, we use a stack of statsd, collectd, graphite so we can collect statistics around usage of tmate.io throughout the day. The proxies also send latency metrics to statsd.

The C codebase catches SIGSEGV and prints the backtrace in the logs. In the past year, crashes were not the issue, but process hanging was. For example, we had a bug where a SIGCHLD signal handler would call printf(), which internally calls malloc(). When a signal was triggered during normal code execution interrupting a malloc() call, a deadlock would occur due to a double malloc() call. In these cases, analyzing a core dump is more useful.

We use rollbar, a 3rd party service, to report errors from the Elixir codebase. We also use rollbar to report errors from the Javascript codebase. Note that we also provide javascript mapping from the compiled javascript to our source code so we can have readable backtraces.

4.17 Deployment

4.17.1 Server side

All of our infrastructure is deployed with chef. To upgrade the tmate replica codebase, a recompile and a service restart ensures that new sessions run the new code. However, the existing running session are not migrated to the new codebase. However, with Elixir, we use exrm and edeliver to compile a bundle that allow a hot patch of the running Elixir services. This way, we can upgrade the codebase without having to restart the daemon, we don't close any running connections, and all the existing application state is migrated to the new codebase.

4.17.2 Client side

When we release a new version of the tmate host, we need to distribute on various platforms. 1) tmate is in the official tree of homebrew. I have to make pull requests to get it upgraded. Their build bot compiles tmate for the three versions of Mac OSX. 2) For Ubuntu, A contributor Javier Lpez creates the ubuntu releases. 3) On Gentoo, tmate is in the official tree. 4) On Arch, contributors get it updated 5) I also provide static builds for amd64, i386, and armv7l.

4.18 User Manual

4.18.1 Installation

You may find tmate in your package manager. Otherwise, you may install the static binaries.

4.18.2 Running tmate

1) Make sure you have ssh keys available on your machine. Run ssh-keygen otherwise 2) Launch `tmate`. You should see a connection string in the form "ssh XXX@xx.tmate.io" at the bottom of the screen. With this connection string, people can connect to your terminal. 3) If you need additional connection accesses, you may run `tmate show-messages`. You will see something of the form:

```
   Connecting to ssh.tmate.io...  Note:
clear your terminal before sharing
readonly access web session read only:
https://tmate.io/t/ro-DYok0JDncSXywEAhNyPzHZPS6
ssh session read only:  ssh
ro-DYok0JDncSXywEAhNyPzHZPS6@ny2.tmate.io
web session:  https://tmate.io/t/1Ir20RWWFkkFqzddq39R5RM6
ssh session:  ssh 1Ir20RWWFkkFqzddq39R5RM66@ny2.tmate.io
```

You may share any of these connection accesses depending if you want web or ssh access, or read/write or read-only access.

4.18.3 Configuring tmate

As tmate is a fork of tmux, it reuses the tmux configuration file, ` /.tmux.conf`. Additionally, tmate reads the ` /.tmate.conf` where the following options can be specified:

* tmate-identity: use a specific SSH key, instead of the default one. * tmate-server-host: Use a custom tmate remote server instead of the official ssh.tmate.io. * tmate-server-port: tmate server port. * tmate-server-rsa-fingerprint: SSH fingerprints of the tmate remote server. * tmate-server-ecdsa-fingerprint: SSH fingerprints of the tmate remote server. * tmate-display-time: Configures how long tmate status messages stays. Defaults to 10 seconds. * tmate-webhook-userdata: Configures a custom webhook (see above for more information). * tmate-webhook-url: Configures a custom webhook (see above for more information).

Since tmate is a fork of tmux, it supports all of its 85 commands (e.g. new-session, split-window, pipe-pane). See the man page of tmux for a complete reference: http://manpages.ubuntu.com/manpages/trusty/man1/tmux.1.html

4.18.4 Reverse Tunnel example

The following shows an example of how to run tmate as a daemon:

```
   # Launch tmate in a detached state
$ tmate -S /tmp/tmate.sock new-session
-d # Blocks until the SSH connection is
established $ tmate -S /tmp/tmate.sock wait
tmate-ready # Prints the SSH connection
string $ tmate -S /tmp/tmate.sock display
-p '#tmate_ssh'
```

4.19 Future Work

* There's some non trivial graphic bugs to fix with the use of term.js. Instead of streaming the pty data to the

browser, we might want to do the rendering server side (which is done anyways), and send some smart updates to the browser (a bit like the mosh protocol). So maybe we should remove term.js and replace it with our own thing to get something really robust.

* Provide a mosh client

* send the session stream to cassandra to record the sessions. Each column would be keyed by a timestamp, and the value would be a frame. a frame would start with a snapshot, and 30 seconds of session data (or 1Mb, whatever comes first). This would allow easy scrolling in time like a youtube video.

* We also need to interpret the terminal line by line (not row by row, which can wrap), keep only the raw text, and ship it to elasticsearch for indexing and full-text search. This would also be useful for regular expression triggers on text.

* The HTML dashboard: - it would present all the sessions associated with the account. - it needs to present 100's of terminals potentially. The thumbnails need to be something relevent. Perhaps rendering session snapshots into some small .png would be a good idea. - we also need some "new terminal" button next to a host name.

* We also need to push the initd function of the local tmate daemon. We want to be able to make it work in a Docker environment to launch the processes.

5 Evaluation

We deployed tmate is three different environments: a public geographically distributed service for remote pairing, a continuous integration service integration for debugging purposes, and a test bed environment for scalability testing on Amazon AWS.

5.1 AWS deployment

We ran experiments on Amazon AWS with up to 100 c3.large instances (2-core, 4GB) running simultaneously to saturate tmate. To evaluate tmate throughput and latency under high load, we developed a stress-test microbenchmark simulating N simultaneous sessions running. A session is comprised of one tmate host connected to one tmate daemon. tmate daemons run on k dedicated servers. We connect *s* ssh clients, and *h* html5 clients to each session. The tmate host streams 10kb/s of pty data to the tmate daemon, which in turns is streamed to the clients.

Figure1 shows scaling properties with many sessions. Shown on the x-axis the number of sessions, and on the y-axis the 95th percentile of the latency perceived from the clients, sampled every second for a minute. We connect one ssh client and one html5 client (s=1,h=1). The latency is measured by the clients. Each client periodically writes a character into a pane, and measures the time it takes to receive an echo back. The figure shows various cluster sizes with k=1,3,10,30. We see that the service capacity grows linearly w.r.t. the cluster size, at a rate of X sessions/server.

Figure2 shows a single session with many clients. Shown on the x-axis the number of clients, and on the y-axis the perceived latency. We scale the number of ssh clients separately from the number of html5 clients. The tmate daemon bottlenecks at X ssh clients with no html5 clients, and Y html5 clients with no ssh clients. A host can accomodate many more html5 clients compared to ssh clients because each ssh client consumes a fair amount of CPU due to pty rendering, while html5 clients impose only modest overhead on the server as html5 clients receive the same data stream.

We also wanted to evaluate how a noisy session can affect other sessions on the same server. On a machine with X running sessions, we took X/2 sessions to make stream as much data as they could. We measured the perceived latency of the remaining session. We found that X. (should we use cgroups and throttle tasks?)

Note: Should we evaluate a graphical / VNC type of solution? Even though these sort of solutions are not really practical as they require the sharing of a whole desktop screen, and not just share a window.

5.2 Public deployment (tmate.io)

We provide a public geographically distributed environment for people to use. Our public deployment is deployed at seven different locations: sf, to, ny, ln, am, fk, sg on the DigitalOcean cloud. In the past month (March 26 to April 26 2016), there was 31805 sessions created. If we only look at the sessions that had at least one client connected, this leaves us with 3939 sessions (12%). The rest of the discussion focuses on these sessions. The mean and median session duration was 260min and 30min respectively. 10% of the sessions had at least two different clients connected (identified with their SSH keys) and 2% had at least three clients connected. The maximum was 6 clients connected. The sessions were created by 1409 different hosts, and we saw 1496 unique SSH clients connecting to these sessions. We saw 6033 different client connection, 11% of them were through the HTML5 interface. Among all remote clients, we measured the end to end latency (RTT of client to tmate.io + RTT tmate.io to the host). For each client connection, we recorded the mean, median, p90 and p99 of the end to end latency. The mean of the means is 420ms. The median of the mean is 168ms. The median of the median is 134ms. The median of the p90 is 228ms. The median of the p99 is 444ms. These latencies are low enough to make remote pairing a pleasant experience. tmate is typically used on week days between 10am and 6pm.

5.3 Travis-CI integration

Travis-CI [1] is a well-known continuous integration service used by more than 200,000 projects. Travis-CI monitors a project's git repository and runs the test suite when the repository receives new commits. This way, regressions are caught early when running code. When the test suite fails, it can be for a variety of reasons. Some failures may be hard to diagnose. Unfortunately, Travis-CI did not offer any ways for users to interactively inspect and debug their tests. tmate was integrated to Travis-CI to allow users to interact directly with the machine that runs their test suite.

At the moment, users can insert a ”debugging” statement in their test pipeline, which spwan a new SSH connection string for users to connect and debug their test suite. Without tmate, implementing this feature would be difficult. An SSH server would need to be spawned in the test running container, which port should be properly rounted on the VM host. with tmate, this feature is easy to implement.

Not sure what to say in term of evaluation for the moment, as we won't do any crazy load for the moment.

6 Conclusions

tmate conclusion.

References

[1] Travis-CI. `https://travis-ci.org`.