Mosh: A State-of-the-Art Good Old-Fashioned Mobile Shell

Keith Winstein and Hari Balakrishnan
M.I.T. Computer Science and Artificial Intelligence Laboratory, Cambridge, Mass.
{keithw,hari}@mit.edu

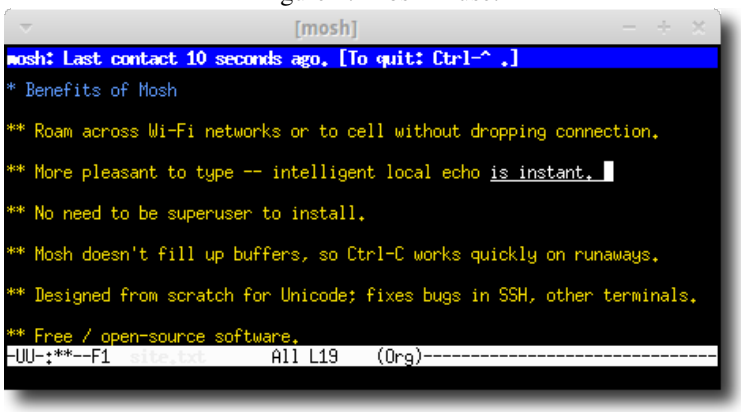
Remote terminal applications are almost as old as packet-switched data networks. Starting with RFC 15 in 1969, protocols like TELNET, SUPDUP, and BSD's rlogin and rsh have played an important role in the Internet's development. The reader of *;login:* has undoubtedly used the most popular of these: the Secure Shell, or SSH, which since 1995 has ruled the wires. This article describes a successor to these venerable applications that was designed to work better in our increasingly mobile and wireless world.

SSH has two weaknesses that can make it unpleasant today. First, because SSH and previous remote terminals run over TCP, they don't support roaming between IP addresses, or always preserve sessions when connectivity is intermittent. A laptop will happily suspend for a commute to work, but after wakeup its SSH sessions will have frozen or died.

Second, SSH operates strictly in character-at-a-time mode, with all echoes and line editing performed by the remote host. As a result, its interactive performance can be poor over wide-area wireless (e.g., EV-DO, UMTS, LTE) or transcontinental networks (e.g., to cloud computing facilities or remote data centers), and sessions are almost unusable over paths with non-trivial packet loss. When loaded or when the signal-to-noise ratio is low, delays on many wireless networks reach several *seconds* because of deep packet queues ("bufferbloat") or over-zealous link-layer retransmissions. Many home networks also suffer from multi-second delays under load. Trying to type, or correct a typo, over such networks is unpleasant.

These problems affect users to varying degrees. For us after fifteen years, they eventually bubbled over from the cauldron of smouldering frustration that produces software: in this case **Mosh**, the mobile shell (http://mosh.mit.edu). Mosh is free and open-source software and is available for most major operating systems.

Figure 1: Mosh in use.



Mosh fixes these issues. A user who switches IP addresses (e.g. from Wi-Fi to cellular while leaving a building, or from home to work) keeps the session without thinking about it. Ditto for suspend and resume. Mosh is careful about flow control and doesn't fill up network buffers, so for example "Control-C" works right away to halt output from a runaway process. Mosh reacts to packet loss intelligently.

In addition, the Mosh client runs a predictive model of the application in the background, and uses the model to do intelligent client-side echoing and line editing. According to data we have collected from contributing users, more than two-thirds of the keystrokes in a typical Unix session can be displayed instantly with a conservative model of application behavior. Mosh's empirical approach to local echo works in full-screen programs like a text editor or mail reader as well as at the command line, and doesn't require a change to server-side software.

We announced Mosh by accident in April 2012, or to be fair, somebody announced it for us with a post on Hacker News (http://news.ycombinator.com). Over the next 48 hours, Mosh's hurriedly-completed Web page received more than 100,000 views; not an indication of merit, but an unexpected amount of attention for a Unix system utility. The user community has gone from 8 users at M.I.T. to more than 30,000 users.

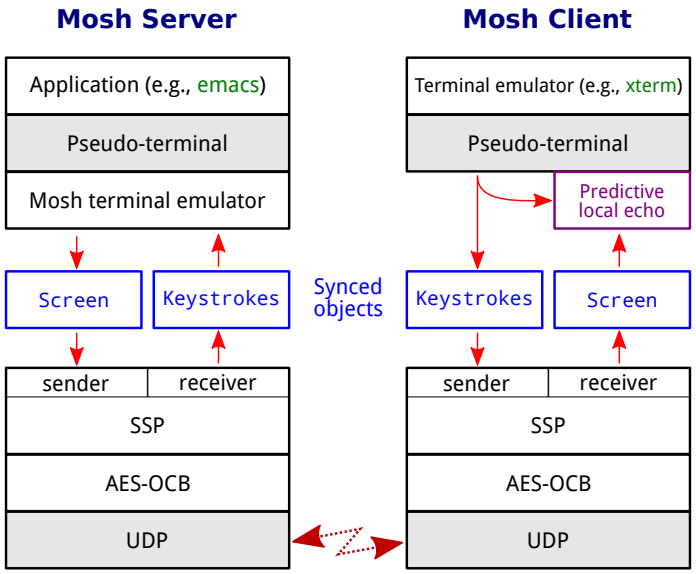
This level of interest may be because we scratched an itch: Mosh is a rare example of a gracefully-mobile network application. Today, many programs intended for mobility, including e-mail clients and Web browsers on popular smartphones, can't cope gracefully if the client switches network interfaces, roams, or has intermittent connectivity—the very conditions presented by mobile networks.

Here's our view of the ideas behind Mosh, with the hope that new applications could benefit from the same principles. We plan to investigate this hypothesis in future work.

1 Choose a Good Abstraction

Traditional remote-shell protocols such as TELNET, rlogin, and SSH work by reliably conveying a byte-stream from the server to the client, to be interpreted by the client's terminal. Mosh works at a different layer and treats the remote terminal more like a videoconference. With Mosh, the server runs a terminal emulator, and the server and client each maintain a snapshot of the current screen state. The problem becomes one of state-synchronization: fast-forwarding the client to the *most recent* screen as efficiently as possible.

Figure 2: Mosh's design.



This synchronization is accomplished using a new protocol we call the *State Synchronization Protocol* (SSP). SSP runs over UDP, synchronizing the state of an object from one host to another. Datagrams are encrypted and authenticated using AES-128 in the Offset Codebook mode [1], which provides confidentiality and authenticity with a single secret key.

Because SSP works at the object layer and can control the rate of synchronization (i.e., the frame rate), it does not need to send every byte it receives from the application. Mosh regulates the frames so as not to fill up network buffers, retaining the responsiveness of the connection. Mosh sets the minimum time between frames to half the smoothed round-trip time (RTT) of the path. By contrast, SSH doesn't know how the client will interpret each octet, and so must send everything the application generates.

A schematic of Mosh's design is shown in Figure 2. Mosh runs two copies of SSP, one in each direction of the connection. The server-side terminal emulator exports an object called the `Screen`, containing the contents of the

terminal display. This is the object that SSP synchronizes to the client. Meanwhile, the client records the user's keystrokes in a verbatim transcript, and synchronizes this object to the server.

Why TCP is the wrong abstraction. TCP presents a reliable, in-order, byte-stream abstraction and assumes continual connectivity between a pair of IP addresses. For applications like Mosh, this interface is problematic: In marginal conditions or after a period of disconnectivity, the server ought to try to "fast-forward" the client to the current screen state, not resend old data that has been queued. Even if the server application knows that much of the data queued up isn't useful, it is not possible to "pull back" stale data from a kernel socket buffer, much less from the network.

TCP doesn't allow intermittent connectivity and roaming. Previous work in this area, including proposals for TCP connection migration and Mobile IP, have not been widely deployed; TCP migration requires kernel modifications, and Mobile IP third-party home agents, and neither supports intermittent connectivity.And TCP's minimum retransmission timeout is at least one second: fine for bulk transfers, but not for human-generated interactive flows.

2 Idempotency for Security and Roaming

SSP is a novel secure-datagram protocol with a design considerably simpler than previous work. We agree this is just cause for skepticism. It will take time for the security community to become comfortable with SSP; protocols like SSH, Kerberos, and TLS have had security holes and design weaknesses surface only after years of scrutiny. (We didn't use Datagram TLS because it doesn't support roaming and requires the endpoints to generate public key pairs to authenticate each other.)

The security of SSP rests on the principle of *idempotency*. Each datagram sent to the remote site is encrypted and authenticated with AES-OCB, and represents an idempotent operation at the recipient—a "diff" between a numbered source and target state. The diff is a *logical* one: the object itself calculates the diff between itself and a future object of the same type, and an object on the other end of the connection "applies" the diff.

Figure 3: Example "diff"

Field	**Value**	**Explanation**
type:	`Screen`	*type of object*
protocol version:	2	
source state:	#17	*what state diff is coming from*
target state:	#20	*will be created when diff is applied to source*
ack:	#6	*latest state sender has constructed from other side*
received ack:	#17	*latest state other side has constructed from sender*
contents:	`"1st\r\n2nd ln"`	
random chaff:	...	*frustrates packet-length analysis*

For example, a diff might tell the receiver how to get from frame #17 to frame #20. An attacker who repeats the datagram containing this diff, or changes the ordering of datagrams, won't compromise the security of the system.

Roaming becomes simple to accommodate. Every time the server receives an authentic datagram from the client with a sequence number greater than any before, it sets the packet's source IP address and UDP port number as its new target for future outgoing datagrams. As a result, client roaming happens without any timeouts or a notion of reconnection. The client doesn't even know (or need to know) it has roamed. This is helpful when the client is behind a network-address translator (NAT) and it is the NAT that has changed public-facing IP addresses (common when "tethering" to a smartphone).

3 Make Each Packet Your Best Packet

SSP tries to wring the most benefit out of each packet and get the receiver to the current object state as efficiently as it can. The sender can formulate diffs between whatever pairs of states it thinks make for the swiftest way to accomplish

that goal. For lossy links, one technique we have developed is the *prophylactic retransmission*, or p-retransmission. We illustrate this technique with an example:

1. Consider a situation where the receiver has acknowledged the sender's state #3. Then the application changes the object state (e.g., changes the contents of the screen) to a new state, #4.

2. The sender knows that the receiver already has state #3, so it creates a diff from #3 $\rightarrow$ #4 and sends it.

3. Soon after, and before the diff is acknowledged, the object state changes again to #5.

4. If the previous diff hasn't timed out, the sender will formulate a diff from state #4 $\rightarrow$ #5, with the assumption that both diffs will arrive and be applied. This is the "normal transmission." If the #3 $\rightarrow$ #4 packet was lost, the receiver won't be able to apply the new diff and will stall until the sender times out and retransmits.

5. But another option is to formulate a diff all the way from state #3 $\rightarrow$ #5: the p-retransmission. Sometimes this diff will actually be shorter or the same size as the normal transmission, and it's more likely to be useful to the receiver because it doesn't have state #4 as a prerequisite. (It's a retransmission of sorts, because implicitly we're re-sending the diff between state #3 and #4 before it has timed out.)

The algorithm says that if the p-retransmission is shorter or only slightly longer (relative to packet overhead) than the normal transmission, we send it instead. The advantage is that if a loss occurs, we get the benefits of resending without necessarily repeating anything and without a timeout and stall, with a tunable maximum overhead.

4 Speculate, but Verify

Because Mosh operates at the terminal emulation layer and maintains the screen state at both the server and client, it's possible for the client to make predictions about the effect of user keystrokes and later verify its predictions against the authoritative screen state coming from the server.

Most Unix applications react similarly in response to user keystrokes. They either echo the key at the current cursor location or not. As a result, it's possible to approximate a local user interface for arbitrary remote applications. We use this technique to boost the perceived interactivity of a Mosh session over network paths with high latency or high packet loss. When the RTT exceeds a threshold, we underline unconfirmed predictions so the user doesn't become misled. This underline trails behind the user's cursor and disappears gradually as responses arrive from the server. Occasional mistakes are removed within one RTT.

Previous work in this area, such as TELNET's LINEMODE option, only works when the kernel itself is echoing user keystrokes. This is rare today, as full-screen applications (like `emacs` and `vi`) and even command-line shells now disable these kernel echoes and process keystrokes within the application. Mosh's approach is more general and doesn't require a change to server software.

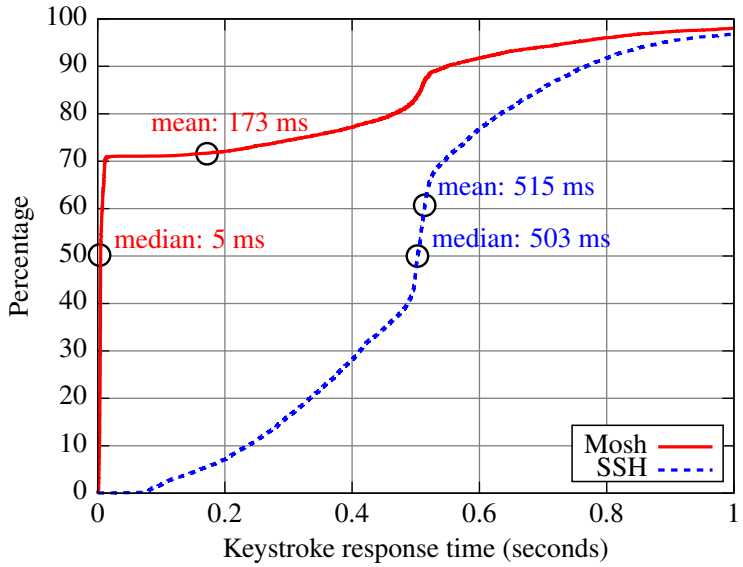
The challenge is that about a third of user keystrokes are "navigation" (such as going to the next e-mail message, or changing modes in `vi`) and shouldn't be echoed. Of course, password entry shouldn't be echoed either.

Mosh deals with this conservatively. The client makes predictions in groups known as "epochs," with the intention that either all of the predictions in an epoch will be correct, or none will. An epoch begins tentatively, making predictions only in the background. If any prediction from a certain epoch is confirmed by the server, the rest of the predictions in that epoch are immediately displayed to the user, along with any future predictions in the same epoch.

Some user keystrokes are likely to alter the host's echo state from echoing to not, or are otherwise hard to predict, including the up- and down-arrow keys and control characters. These cause Mosh to lose confidence and increment the epoch, so that future predictions are made in the background again.

Experimental results. We evaluated Mosh using traces contributed by six users, covering about 40 hours of real-world usage and including 9,986 total keystrokes interacting with a variety of Unix applications. These traces included the timing and contents of all writes from the user to a remote host and vice versa. The cumulative distributions and statistics of keystroke response time are shown in Figure 4. When Mosh was confident enough to display its predictions, the response was nearly instant. This occurred about 70% of the time. Our USENIX ATC paper [2] reports similar results on other Internet paths.

Figure 4: Cumulative distribution of keystroke response times with Sprint 1xEV-DO (3G) Internet service

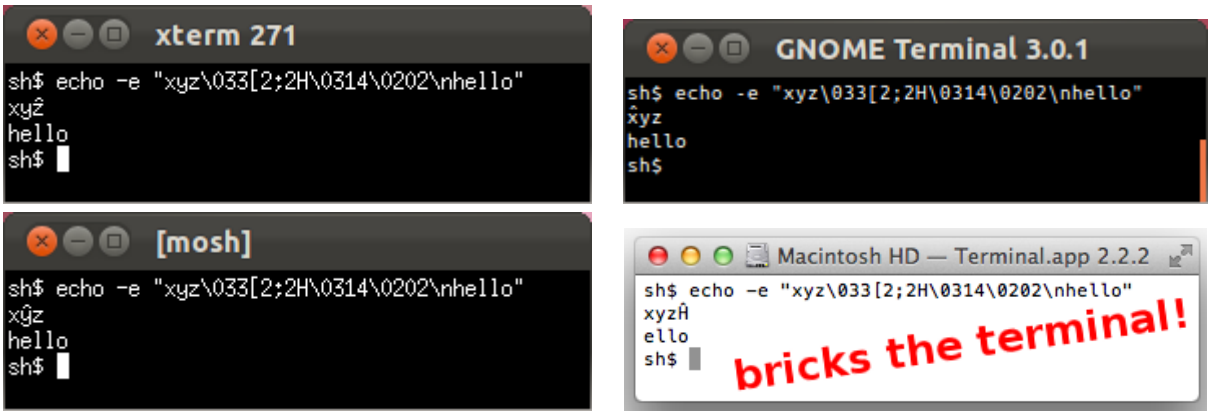


5 Get the Details Right, or if There is no "Right," Defensible

Character set handling on Unix—maybe on any operating system—remains a dark and under-specified area. We learned, for example, that there is no specification for a Unicode terminal emulator (the ECMA-48 specification was last revised in 1991). Existing terminal emulators show a variety of interpretations on issues like, e.g., the order of normalization and the placement of combining accents when mixed with control characters (such as newline) and escape sequences.

Figure 5 illustrates this problem. The same sequence, in which a combining accent is mixed with an escape sequence and newline, gives rise to four different interpretations in four popular terminal emulators. The most interesting of these is the Mac OS X `Terminal.app`, which normalizes the input before parsing the escape sequences, then triggers a bug that freezes the terminal.

Figure 5: Same string, four interpretations.



We can't argue that Mosh's interpretation of Unicode corner cases is *correct*, because there is no authority on such

matters, but we did put effort into making it *defensible*.

We also learned that POSIX and the Single Unix Specification don't provide a mechanism to delete multibyte characters (including UTF-8 sequences) in "canonical mode," because the kernel doesn't know how many bytes to delete, or even what character set the user is in. Linux and Mac OS X have responded by creating an IUTF8 termios flag to tell the kernel that it should interpret input as UTF-8.

Mosh sets this flag where available, but that only fixes part of the problem: the kernel still doesn't know how many columns the character occupies on the display and how many backspaces to send. With IUTF8 set, deleting wide Chinese, Japanese, and Korean characters won't leave garbage in memory, but it still leaves garbage on the screen. There's no easy solution to this problem—we hope CJK users aren't often trying to type and then delete wide characters in canonical mode.

6 Solve a Small Problem

With Mosh, we tried not to solve any problems we could avoid. Mosh doesn't authenticate users, run a daemon, or contain any privileged (root) code. Users don't need root to install Mosh on the client or server. Mosh doesn't use public-key cryptography. Users continue to authenticate and log in through their existing means: probably SSH with passwords, public keys, or Kerberos.

The `mosh` program is a script that uses SSH to log in to the server and execute an unprivileged `mosh-server` process, which prints out an AES key and binds to a high port number. The script then shuts down the SSH connection and executes the `mosh-client` with the supplied key and port. The client contacts the server directly over UDP.

Mosh doesn't support multiple windows, split-screen modes, multiple clients connected to the same server, or reattaching if the client has rebooted or the user has moved to a different machine. For these features, users often run terminal multiplexers like GNU `screen` or OpenBSD `tmux` inside a Mosh session.

Limitations and future work. The current version of Mosh (version 1.2.1) has several limitations:

- Because Mosh only synchronizes an object representing the current contents of the screen, there is no guarantee that the user will be able to scroll back to history that was missed. If the application is sending large volumes of output or the user was disconnected, the history could have been in lines that were skipped over. The user must use a server-side pager, such as `screen` or `less`, to have accurate scrollback.
- Mosh doesn't tunnel X11 sessions or `ssh-agent` requests.
- Mosh doesn't work through TCP-only proxy servers.
- The Mosh server requires a separate UDP port for each concurrent client—when the client's IP address can vary, the server-side UDP port is the only invariant connection identifier. Some users have objected that opening the default port range of 60000–61000 is not a realistic request of security-conscious network administrators.
- Mosh clients for Android and iOS are still in development.
- Mosh doesn't support IPv6, or roaming from IPv6 to IPv4 addresses for the same server.
- Mosh provides transport-layer security, but doesn't attempt to hide the existence of a session.

Mosh has attracted a cadre of online contributors, and we plan to address some of these issues in future versions. Limited as Mosh is, we've received valuable and encouraging feedback from users. Releasing early and often was the right decision.

References

[1] T. Krovetz and P. Rogaway. The software performance of authenticated-encryption modes. In *18th Intl. Conf. on Fast Software Encryption*, 2011.

[2] K. Winstein and H. Balakrishnan. Mosh: An Interactive Remote Shell for Mobile Clients. In *USENIX Annual Technical Conference*, Boston, Mass., June 2012.