Mosh: An Interactive Remote Shell for Mobile Clients

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

Abstract

This paper describes Mosh, a mobile shell application that supports intermittent connectivity, allows roaming, and provides speculative local echo of user keystrokes. Mosh is built on the State Synchronization Protocol, a new UDP-based protocol that securely synchronizes client and server state, even across client IP address changes. Mosh uses SSP to synchronize a character-cell terminal emulator. By maintaining the terminal state at both client and server, the Mosh client predicts the effect of user keystrokes and speculatively displays many of its predictions without waiting for the server to echo.

For mobile clients, Mosh is considerably more usable than the Secure Shell (SSH) protocol for two reasons. First, unlike SSH, it maintains sessions across periods of disconnection and changes in network address. Second, by speculatively echoing keystrokes locally, Mosh improves interactivity over high- or variable-delay network paths.

Our evaluation analyzed keystroke traces from six different users covering a period of 40 hours of real-world usage and including 9,986 keystrokes. Mosh was able to display immediately the effects of 70% of the user keystrokes. Over a commercial EV-DO (3G) network, median keystroke response latency with Mosh was 4.8 ms, compared with 503 ms for SSH. Mosh erred in predicting the keystroke response 0.9% of the time, but removed the error from the screen after at most one round-trip time.

1 Introduction

The character-cell terminal is a venerable and wildly popular interface for the remote use and administration of networked computer systems. Dating back to the development of TELNET [18] and SUPDUP [7, 23] in the 1970s, users have relied on text-based remote login protocols to control servers and supercomputers and access faraway software and resources.

Nowadays, the prevalent approach is the Secure Shell (SSH) protocol [25], which has been implemented for all major architectures and operating systems, including smartphones and tablets, running inside a terminal emulator. Because the basic language for drawing text and lines in a character terminal has not changed in over twenty years [1], the remote terminal has become the lingua franca among diverse computer systems on the Internet.

System administrators use SSH to control servers, routers, and almost every other piece of equipment in a data center. Developers use SSH to access servers in the "cloud." Users rely on SSH for access to desktops, databases, chat rooms, e-mail, and other remotely-installed software. Much of this access now occurs from mobile devices—including smartphones, tablets, and laptops connected via Wi-Fi or cellular-based networks. There are several remote terminal applications available in the stores for iOS and Android devices.

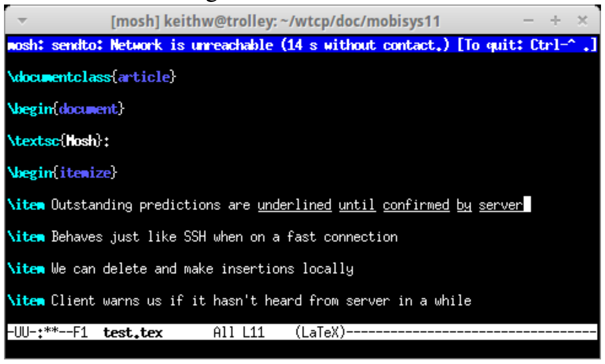
Unfortunately, SSH has two major weaknesses that make it unsuitable for mobile use. First, because it uses a single TCP connection, SSH does not support roaming among IP addresses, intermittent connectivity while data is pending, or marginal links with packet loss. A laptop cannot switch Wi-Fi networks, or from Wi-Fi to a cellular data connection, and expect to keep its connection. Nor can a smartphone travel in and out of range of a carrier's signal while maintaining an SSH session.

Second, SSH operates strictly in character-at-a-time mode, with all echoes and line editing performed by the remote host. On today's commercial EV-DO and UMTS (3G) mobile networks, round-trip latency is typically in the hundreds of milliseconds when unloaded, and on both 3G and LTE (pre-4G) networks, delays reach several seconds when buffers are filled by a contemporaneous bulk transfer. Such delays often make SSH painful for interactive use on mobile devices.

This paper describes a solution to both problems. We have built Mosh, a mobile shell application that supports IP roaming, intermittent connectivity, and marginal network connections, and performs predictive client-side echoing and line editing without any change to server software, and without regard to which application is running on the server. Mosh makes remote servers feel more like the local computer, because most keystrokes are reflected immediately on the user's display—even in full-screen programs like a text editor or mail reader.

These features are possible because Mosh operates at a different layer from SSH. While SSH securely conveys an octet-stream over the network and then hands it off to a separate client-side terminal emulator to be interpreted and rendered in cells on the screen, Mosh contains a server-side terminal emulator and uses a new protocol to synchronize terminal screen states over the network. Because both the server and client maintain an image of the screen state, Mosh can support disconnected operation and predictive client-side local editing.

Figure 1: Mosh in use.



```
mosh: sendto: Network is unreachable (14 s without contact.) [To quit: Ctrl-^ .]
\documentclass{article}
\begin{document}
\textsc{Mosh}:
\begin{itemize}
\item Outstanding predictions are underlined until confirmed by server
\item Behaves just like SSH when on a fast connection
\item We can delete and make insertions locally
\item Client warns us if it hasn't heard from server in a while
-UU-:**--F1  test.tex      All L11    (LaTeX)--------------------------------
```

Because the server can skip past intermediate screen states, the system can adjust its network traffic so as not to fill up network buffers on slow links. By contrast, SSH has to send all application data because it does not know what effect any byte will have on the client's state. As a result, unlike in SSH, in Mosh "Control-C" always works to cease a runaway server-side process within an RTT.

Mosh's design makes two principal contributions, based on important principles that could apply to other applications:

1. **State Synchronization Protocol, or SSP:** A new secure object synchronization protocol on top of UDP to synchronize abstract state objects between a local and remote host in the presence of roaming and intermittent connectivity.

2. **Speculation:** Mosh has a "split" terminal emulator that maintains images of the screen state at both the server and client and uses the above protocol to synchronize them. The client is free to make guesses about the effect a new keystroke will have on the screen state and when confident will render such effects immediately. After one RTT, the client can check if its speculation was correct and can repair the screen state if it made a mistake.

We have implemented Mosh in C++ for Linux and Unix-like systems and have experimented with it across various wireless networks and across disconnections. Mosh is free software and is available at `http://mosh.mit.edu`. An example of Mosh's interface is shown in Figure 1.

We describe SSP in Section 2. Section 3 discusses the split terminal emulator and speculative local echo procedure. In Section 4, we present experimental results on the accuracy of Mosh's predictions and the resulting improvement in interactivity over commercial cellular networks.

2 State Synchronization Protocol

The design of Mosh reflects our assumptions about the user's desired behavior from a mobile shell application. Figure 1 shows a typical use case for the application: the user in a full-screen editor window working on a document. We treat the problem of conveying the screen state from server to client almost as we would a videoconference: the goal is to convey the *most recent state of the screen* to the client. (The reverse direction is more restricted because the client must send every keystroke typed to the server.)

To illustrate the consequences of this decision, if the user is playing an ASCII animation and the connection drops for five minutes, after resuming we assume the user is only interested in the current frame on the screen—not all the intermediate frames that the terminal displayed in the meantime.

This choice is appropriate for tasks like editing a document or using an e-mail or chat application, which control the entire screen and provide their own means of navigation through a document or chat session. But it causes trouble for a task like "cat"-ing a large file to the screen. If the connection drops in the middle of the cat, upon resume the user will see only the lines contained on the screen near the end of the file, and will not have the rest of the file in their local terminal's scrollback buffer. We think that this behavior is reasonable in most cases; our design favors interactive response over ordered delivery of all the "older" updates.

If these semantics were a problem, the user could use either a full-screen application that maintains its own buffer and means of navigation, e.g., a pager such as `less` or `more`, or can use the `screen` or `tmux` utilities, which are essentially pagers for the entire terminal and maintain a navigable scrollback buffer on the server side.

2.1 Protocol design goals

Underlying Mosh is SSP, a lightweight, datagram-based protocol to synchronize the state of abstract objects between a local node, which controls the object, and a remote host that may be only intermittently connected.

The Mosh system runs two instances of this protocol, one in each direction, instantiated on two different kinds of objects. From client to server, the objects being synchronized represent the history of the user's input. That input comes in two forms: keystrokes and requests to resize the terminal window. From server to client, the objects represent the current state of the terminal window.

SSP's design goals were to:

1. Leverage existing infrastructure for user and host authentication and login, e.g., SSH.

2. Not require any privileged code.
3. At any given time, take the action best calculated to fast-forward the remote host to the sender's current state.
4. Accommodate a roaming client whose public IP address changes, without the client's having to know that a change has happened.
5. Preserve the confidentiality and authenticity of the session against an active attacker in the network.
6. Recover from any sequence of dropped or reordered packets, no matter how long the connection has been interrupted.

We use the existing infrastructure for authenticating hosts and users. To bootstrap an SSP connection, the user first logs in to the remote host using conventional means, such as SSH or Kerberos. From there, the user or her script runs the server: an unprivileged process that chooses a random shared encryption key and begins listening on a UDP port. The server conveys the port number and key over the initial connection back to the client, which uses the information to start talking to the server over UDP.

Because Mosh does not use any privileged code and doesn't authenticate users, its security concerns are simplified. Mosh only needs to ensure the confidentiality and authenticity of a single terminal session running between two processes started by the same unprivileged user.

SSP is organized into three modules. A cryptographic module provides confidentiality and authenticity of arbitrary messages. A datagram layer sends UDP packets over the network. And a "transport" layer is responsible for conveying the current object state to the remote host in an efficient manner. It can be instantiated to synchronize any object that supports a particular interface.

2.2 Cryptographic Module

The security of the system is built on AES-128 in the Offset Cookbook (OCB) mode [11], which provides confidentiality and authenticity with a single 128-bit secret key. We use Krovetz's optimized reference implementation.

OCB requires that each plaintext be paired with a unique 128-bit nonce over the life of the encryption key. We use an incrementing 63-bit sequence number and direction flag for this purpose. That is enough to guarantee that each packet successfully received was sent by the remote host at some point.

To handle reordered and repeated packets, SSP relies on the principle of idempotency. Each datagram sent to the remote site represents an idempotent operation at the recipient—a "diff" instructing the remote site how to construct state m from some prior state $n < m$. As a result, unlike Datagram TLS and Kerberos, SSP does not need to maintain a replay cache or other message history state, simplifying the design and implementation.

Figure 2: Datagram format

Name	Format	Purpose
`Direction`	bool	High bit of nonce
`seq`	63-bit uint	Increments with every outgoing packet.
`timestamp`	16-bit uint	Lower 16 bits of ms timer.
`timestamp_reply`	16-bit uint	Last timestamp received.
`payload`	string	From transport layer.

2.3 Datagram Layer

The datagram layer maintains the encrypted connection to the remote host. It accepts opaque payloads from the transport layer, places them in packets, delivers the packets to the cryptographic layer, and sends the resulting ciphertext as the payload of a UDP datagram. It is responsible for estimating the timing characteristics of the network path and keeping track of the client's current public IP address and port number.

The datagram format is shown in Figure 2. All integers are in network byte order. The `seq` field increments with every outgoing packet. Together the `Direction` and `seq` form the nonce, ensuring that we do not repeat the same nonce-key combination.[1]

Client roaming. The client's datagram layer maintains the same socket address for the server throughout the life of the connection. Thus, the server is not permitted to roam. However, the client *is* permitted to change its IP address (and public port number, e.g. if it has roamed to or from a NAT). The server's datagram layer maintains the current socket address of the client. Whenever an authentic datagram arrives at the server from the client with a sequence number greater than any previously received, the server stores the packet's source address and port number as the client's new socket address.

As a result, client roaming happens automatically, without the client's necessarily even knowing that it has changed public IP addresses. It is the responsibility of the transport layer to send occasional heartbeats to make sure the server is informed of the client's new address even if no data are pending.

Because the server considers only packets with sequence numbers that surpass the greatest previously received, the roaming operation is invariant to reordering

[1]In the unlikely event that the 63-bit packet sequence number rolls over, the connection is terminated.

and an attacker cannot confuse the server by replaying an old packet from an old client address.[2]

Estimating round-trip time (RTT) and RTT variation. The datagram layer is also responsible for estimating the smoothed round-trip time (SRTT) and RTT variation (RTTVAR) of the connection,

When sending a datagram, SSP puts a millisecond timestamp into the `timestamp` field. If it has received a `timestamp` from the remote site within the last second, it encodes the `timestamp_reply` field with a copy of that most-recently-received timestamp, adjusted forward for how long it has been since it received the timestamp.

Because every datagram has a unique sequence number, unlike with TCP there is no ambiguity between the timestamps of different datagrams where one may be a retransmission of the same payload.

On receipt of a `timestamp_reply`, SSP updates its RTT and RTTVAR estimates using the equations from TCP [15]. It ignores out-of-sequence datagrams.

SSP also calculates a retransmission timeout using the same formula as TCP, but alters the lower limit from 1 second to 50 ms. In an interactive session, the timeout is generally the only mechanism that detects a dropped packet. Because there is typically only a small amount of data pending at a time, TCP will not receive a triple duplicate acknowledgment to inform it of packet loss. Thus, because of the lower limit on RTO, SSH cannot usually detect a dropped keystroke faster than within one second, irrespective of RTT, making it perform poorly on paths with non-trivial packet loss rates.

2.4 Transport Layer

The transport layer is responsible for synchronizing the current contents of the local state to the remote host and vice versa.

The implementation is divided into two parts: the sender and receiver. Each instance includes one of each. For example, in Mosh, the server is the sender for screen states and the receiver for user input. The client is the sender for user input and the receiver for screen states.

SSP itself is agnostic to the type of objects sent and received. An object to be conveyed must only support the four-function interface shown in Figure 3.

Figure 3: Transport state object interface

Method	Purpose
`string diff_from( Obj )`	Find difference between two objects.
`apply_string( string )`	Apply `diff` to object.
`bool operator==( Obj )`	Compare for equality.
`subtract( Obj )`	Remove common prefix.

Figure 4: Transport Instruction (protocol buffer)

Name	Format	Purpose
`old_num`	variable uint	Reference state
`new_num`	variable uint	`diff` brings reference to this target
`ack_num`	variable uint	Latest state received
`throwaway_num`	variable uint	Oldest state receiver must save
`diff`	string	Input to `apply_string()`

Transport sender behavior

The transport sender is implemented as an event-driven automaton. When woken up, its goal is to bring the receiver from its current opinion of the receiver's state to the actual state of the object. It does this by sending an "Instruction": a self-contained document that describes the identify of the source and target states and the binary difference (`diff`) required to get the receiver from source to target.

Instructions represent idempotent operations on the receiver: no matter how many times the `diff` from one state to the next is received, it can only do the same thing each time. The format is described in Figure 4. When the length of an instruction exceeds 1,400 bytes, the transport sender fragments it before passing to the datagram layer.[3] A trial instruction, with an empty `diff`, serves as an acknowledgment without accompanying data.

Which instruction to send?

When preparing a new instruction to update the receiver, the target state is foreordained: it's the most recent state of the object to be synchronized. But the source state is a different story. In the presence of packet loss, the transport sender does not know the current state of the receiver, and therefore cannot know the most appropriate state to serve as a reference for the `diff`.

Instead, the transport sender and receiver cooperate to give each other options. The sender maintains a rolling list of the states it has told the receiver how to create, and the receiver keeps a rolling list of the states it has been able to construct, any of which can serve as the reference for an incoming instruction.

In the receiver's own instructions to the sender, it includes an acknowledgment of the greatest-numbered state it has been able to construct. The highest-numbered acknowledgment received by the sender becomes the *known receiver state*. The sender discards its copy of states it has sent prior to the known receiver state from the rolling list. In its own outgoing instructions, the

[2]We do not prevent against a denial-of-service attack where an active attacker intercepts packets and resends them under its own IP address to fool the server's roaming detection. Such an attack would not compromise the confidentiality of the connection but would disrupt it.

[3]Originally, SSP used path MTU discovery to find the fragment size. In testing, we found several home Internet connections with sub-1,500 MTUs that did not support PMTU discovery, so SSP now sends fragmentable IP datagrams with a maximum length of 1,400 bytes.

sender includes the known receiver state as the "throwaway number," telling the receiver to discard all states older than it.

The known receiver state can always serve as the reference for the sender's next instruction. However, it would not be efficient to keep using the known receiver state as the reference for all instructions until a new state is explicitly acknowledged. Ideally, instructions arriving in series would build on each other instead of each repeating longer and longer `diffs` from a faraway reference.

Instead, the transport sender assumes that any instruction it sent recently enough will eventually be received and can serve as the reference state for its next instruction. How recent is enough? This is the purpose of the retransmission timeout. The most recent state for which the RTO has not yet expired is known as the *assumed receiver state*. It is this state that the transport layer uses as the basis for its instruction.

If the sender drops out of contact with the receiver for too long and its list of sent states after the known receiver state grows to more than 32, the sender begins discarding states to save memory. When the receiver returns, the sender can always use the known receiver state as the reference for the next instruction.

Transport sender timing

Because the goal of Mosh is to produce a mobile remote shell that makes foreign hosts feel as responsive as the local computer, great attention was paid to timing subtleties in the implementation.

The transport sender uses a number of timers to behave efficiently. The "framerate interval" between instructions sent to the receiver is set at half the smoothed RTT estimate, constrained to lie between 250 ms (a 4 Hz frame rate) and 20 ms (50 Hz). As a result, there is about one instruction in flight to the receiver at a time. Network buffers do not fill up and increase latency, even when a process goes haywire and floods the terminal.

This rate control strategy is what allows Control-C and other interrupt sequences to work consistently on "runaway" programs with Mosh. On some existing systems, such as TELNET and RLOGIN [10], the TCP urgent feature is used to signal interrupts to the remote side. This provides a partial solution to runaway processes, as the urgent-data pointer can leapfrog host buffers on the endpoints, but doesn't leapfrog network queues. SSH does not use the urgent feature and is susceptible to runaway processes.

The transport sender uses delayed acks, similar to TCP, to cut down on excess packets. After receiving an instruction from the other side of the connection, it waits up to 100 ms to acknowledge it, in the hope that it will have its own data to send in reply on which the ack can piggyback. When sending a delayed ack, the sender adjusts the reply timestamp contained in the datagram to account for the time the acknowledgment was delayed, so the practice does not confuse the remote party's RTT estimate.

The sender also delays its own outgoing instructions up to 15 ms from the first time its object has changed, in order to collect future updates that may be following in close succession. We assume that these updates tend to clump together (for example, a string being written to the terminal), and it would be wasteful to send off an instruction containing only the first byte of a long string. Additionally, because the next instruction will have to wait at least the full framerate interval (which can be up to 250 ms), a failure to pause to collect all the proximate updates can result in a larger delay overall.

The sender wakes up to send a heartbeat at least every 3 seconds if it hasn't sent a packet for other reasons. This allows the server to learn when the client has roamed to a new IP address, and it lets the client warn the user when it hasn't heard from the server in a while. It also keeps the connection open when the client is behind a network address translator.

Transport receiver behavior

Upon receiving an instruction, the receiver searches for the reference state in its own rolling list of received states, and applies the `diff` to produce the target state, which it saves. If the target state ID is greater than any state the receiver has previously constructed, the target becomes the receiver's new image of the sender's state, and the receiver discards all saved states earlier than the `throwaway_num` given in the instruction. The sender must not use a state earlier than the `throwaway_num` as the reference for a future instruction.

3 A Remote Terminal with Speculative Local Echo

To support the Mosh application, we implemented a terminal emulator from scratch that obeys the SSP state-synchronization interface (Figure 3). The client sends all keystrokes to the server, which applies them and maintains the authoritative reference of the terminal state, which it in turn synchronizes back to the client.

The client also takes the opportunity to intelligently guess the effect that local keystrokes will have on the terminal, and in most cases can speculatively apply such keystrokes immediately. The client observes the success of its predictions to decide how confident to be and whether to actually display the predictions to the user. (When predictions are outstanding for more than 100 ms, we underline them so the user does not become misled.)

Occasional mistakes can be removed within an RTT and do not cause lasting effect.

Although the same network protocol is run in both directions, the client-to-server and server-to-client connections have different semantics because the objects synchronized in each direction have different implementations of `diff_from()`. The client-to-server states represent the history of keyboard input, and `diff_from()` gives the keystrokes typed in between the two states. Nothing is omitted. The server-to-client synchronized objects represent the state of the screen, and `diff_from()` gives the shortest sequence that conveys the terminal between two states, possibly skipping over intermediate screen states.

3.1 Implementing the terminal emulator

Mosh's terminal emulator implements the subset of the ECMA-48/ANSI X3.64 language [1] used by typical terminal emulators, including the `xterm`, `gnome-terminal`, `Terminal.app`, and `PuTTY` programs for X11, OS X, and Windows. This protocol was popularized by Digital Equipment Corp. in the 1970s and 80s and specifies a series of escape sequences to move the cursor around, render characters in bold, underline, and various foreground and background colors, erase areas of the screen, set and remove tab stops, sound the teletype bell, define subregions of the screen as scrolling areas, change the title of the window, etc. The protocol is bidirectional; the host can query the terminal for its current character position and ask it to identify itself.

The terminal manipulates the screen state, which is the object conveyed with the State Synchronization pPotocol. Formally, the screen state includes (1) the window title and (2) a framebuffer containing the contents of the window.

The framebuffer in turn consists of:

- A width and height.
- A array of **rows**, with length equal to the height.
- The current cursor position and visibility.

A row contains (1) an array of **cells** with length equal to the width, and (2) a flag indicating whether the row wraps over to the next line. The latter is important for copy-and-pasting so the local terminal can know whether copied text should include a carriage return at the end of the row and whether a double-click should select a contiguous word spanning the wrap.

Each cell contains:

- An array of Unicode scalar values, representing a combining character sequence. For example, the letter "a" may have several accents and other diacritic marks attached to it, each accent represented with a Unicode code point.
- A width indicating whether the cell takes up one or two columns of the terminal. Many Chinese, Japanese, and Korean characters scripts take up two cells.
- A bitmap of "renditions" indicating how the character should be displayed: bold, underlined, blinking, inverse-video, invisible.
- A numeric foreground and background color as specified by the ECMA-48 standard.

Implementing a terminal emulator to alter this screen state in response to input from the host application was largely straightforward and followed the lines of many similar implementations, but there were a number of deep subtleties involving the correct use of Unicode on POSIX-like systems, including Mac OS X and Linux. Mosh corrects a number of bugs and design problems encountered when using SSH on those systems (see §3.3). Before describing our implementation of these issues, we describe how Mosh speculatively echoes keystrokes locally to improve interactivity.

3.2 Speculative local echo

Because Mosh operates at the terminal emulation layer and maintains an image of screen state at both the server and client, it is 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 operate essentially identically in response to user keystrokes. In most cases, they either echo it at the current cursor location or they don't echo at all. As a result, it is 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 a high-latency network connection or one with packet loss.

Our general strategy is for the Mosh client to make predictions in the background whenever the user hits a keystroke. These predictions are only shown to the user when they have been confirmed—meaning that a letter we hypothesized to appear in a particular cell did appear there. At that point, we can start showing our predictions to the user immediately, but only for the current row of the terminal and only until the user does something that causes us to lose confidence. The speculative engine must prove itself anew on each row of the terminal before showing its predictions to the user.

To achieve this, the Mosh client maintains three state variables:

1. The current "prediction epoch," an integer that represents a family of predictions that will live or die together (for example, when the user starts typing on a new line and we are not sure whether all the characters will be echoed or none will).

2. The current "confirmed epoch," an integer that corresponds to the most recent epoch that includes at least one prediction that has been verified to be correct.

3. An array of predicted overlay cells and cursor locations, indicating predictions for what we think the server will put in certain cells on the screen. Each such prediction contains:

   (a) its prediction epoch, and

   (b) the local state ID that first includes the keystroke that formed this prediction. After the server has acknowledged this state, we can check if the prediction was correct.

The strategy is to update the speculative cursor location locally, moving it to the right on each keystroke and wrapping at the margin. When the user hits a key, we make a prediction that the key will be echoed at the predicted cursor location. We also handle the backspace key and left- and right-arrow keys locally.

Some events cause the server to become tentative in its predictions—in other words, to increment the prediction epoch. For example, when the user hits a control character such as carriage return, an up- or down-arrow key, or the tab character, or wraps to a new line, we don't know whether the server will continue its current behavior. The user might have typed "ssh remotehost" and a carriage return, and the server has replied with "Password:" and has turned off echoing. We do not want to echo the user's characters in that case. By incrementing the prediction epoch, we can continue to make predictions in the background along what we think is the server's most likely course, but this new epoch of predictions will only be displayed to the user when the first of them has been confirmed.

Upon receipt of a new instruction from the server, the client checks its past predictions to see how well it did. Any prediction whose corresponding keystroke was included in outgoing transport state *n* should be reflected on the screen by the time the server is acknowledging having received state *n* from the client. (Because of delayed acks, the server's echo generally arrives in the same instruction as its acknowledgment.)

If a prediction turned out to be incorrect, we erase all predictions made in the same prediction epoch. If a prediction was correct, we advance the "confirmed epoch" to the prediction epoch of the confirmed prediction. At this point, we show the rest of the predictions made in that epoch to the user.

We find that Mosh can display immediately the effects of more than two thirds of user keystrokes in typical use and dramatically improves the perceived responsiveness of the remote terminal over a mobile connection.

3.3 The challenge of Unicode

Unicode has become commonplace in representing coded characters in a computer system, especially on the Web, but we found that its implementation in a character terminal was not straightforward. This section summarizes the most significant issues we encountered in our implementation of Mosh and how we resolved them.

On POSIX systems, the `TERM` environment variable typically corresponds to an entry in a terminfo database that specifies the particular set of escape sequences recognized by the terminal emulator (e.g., `vt220` includes many sequences not recognized by a `vt100`, and `xterm` includes many more capabilities). Mosh uses the `xterm` `TERM` type.

The TELNET protocol uses the TERMINAL-TYPE option [24] to convey the local terminal type to the remote host so that it can send the appropriate control sequences. The SSH protocol sends the `TERM` environment variable during the connection initialization (§ 6.2 of [26]).

Formally, the lowest layer of the protocol interpreter is a finite state machine whose input sequence is a series of 8-bit values. Because the ECMA-48 standard does not fully specify the behavior of this state machine in the presence of pathological inputs, we implemented the state machine of a DEC VT500 terminal reverse-engineered and documented by Paul Williams.[4]

Between about 1998 and 2003, popular terminal emulators began to support internationalized text in the UTF-8 character encoding scheme. To extend the ECMA-48 framework to support UTF-8, `xterm` and other programs effectively modified the underlying state machine to operate on Unicode scalar values encoded in UTF-8 instead of on 8-bit quantities.

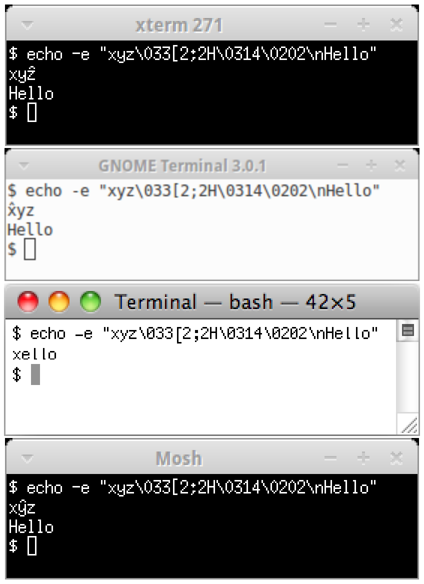
This transition did not necessarily happen rigorously, and terminal emulators today disagree about how to interpret various pathological sequences and when to apply normalization rules (see Figure 5).

Terminal authorities also disagree about whether the move to UTF-8 should disable ISO 2022 shift sequences,

[4]http://vt100.net/emu

Figure 5: Unicode disagreements

Four distinct interpretations of the same UTF-8 sequence from four terminal emulators, because of disagreement about when to apply normalization rules and where to apply diacritics after cursor motion.



```
$ echo -e "xyz\033[2;2H\0314\0202\nHello"
xyẑ
Hello
$
```

```
$ echo -e "xyz\033[2;2H\0314\0202\nHello"
x̂yz
Hello
$
```

```
$ echo -e "xyz\033[2;2H\0314\0202\nHello"
xello
$
```

```
$ echo -e "xyz\033[2;2H\0314\0202\nHello"
xŷz
Hello
$
```

which switch the terminal from text mode into a palette of graphics characters. The historic problem with the use of ISO 2022 was that it used "locking" shift sequences, so a single misbehaving program can put the terminal permanently into hieroglyphs until the user resets it. UTF-8 is a self-synchronizing encoding that makes shift sequences unnecessary, theoretically immunizing the terminal to this problem. However, some applications continue to emit ISO 2022 sequences to draw lines and symbols on the screen. As a result, most popular terminal emulators continue to support these locking shift sequences.

The treatment of wide characters (e.g., Chinese, Japanese, and Korean characters that take up two columns) is another problem area. One difficult case concerns a two-cell wide character that arrives in the last column of the screen. Should the terminal wrap the entire character to the next row (since a two-cell character won't fit in the last cell) and then display it? Or should it display the left half of the character in the last cell? We found that `xterm` 271 exhibits both behaviors, depending on the timing of the received characters and how they are chunked across calls to `read()`.

In implementing Mosh, we attempted to follow the prevailing convention when there was one, and otherwise define and follow consistent behavior that we believe is most sensible. In practice, Mosh has good compatibility with existing terminal emulators on non-pathological inputs.

We also attempted to cut the Gordian knot of two vexing issues concerning the implementation of Unicode on POSIX systems: communicating the character encoding across the network, and handling the kernel's need to know the encoding.

Communicating the encoding across the network

In moving to UTF-8, the `TERM` environment variable retained the same values. To express the difference between an 8-bit `vt220` and a UTF-8 `vt220`, today's systems use the locale functionality of the ISO C and IEEE POSIX standards. For example, on a current Mac OS X or Linux system, a user will typically have a `TERM` environment variable of `xterm` or `xterm-color` and a LANG environment variable of `en_US.UTF-8`.

This creates two challenges to a remote shell application. First, the `LANG` environment variable now needs to be conveyed over the connection to the remote side.[5] Second, to receive the correct character encoding through the connection, the remote host must now support not only the UTF-8 character, set, but the particular *language dialect* of the local user.

For example, SSH connections from a British user (with `LANG` set to `en_GB.UTF-8`) to an American's computer often do not work properly and can produce garbage on the screen, because the en_GB locales are typically not built on American installations. When the locale cannot be found, applications default to the C/POSIX default locale, whose character set is U.S. ASCII. This leads to incorrect results even when UTF-8 is supported by the remote host.

Such issues can become fiendishly difficult to debug when there are several layers of terminal emulation in the mix; for example, a GNU screen process. Mosh avoids confusing the user by simply refusing to start up unless the local and remote hosts both use the UTF-8 character set in the current native locale (recall that Mosh typically inherits a session initialized by SSH). This avoids a substantial number of hard-to-debug misconfigurations at the expense of some pain upfront to properly configure a UTF-8-clean environment.

Informing the kernel of the encoding

Another difficulty with the use of the `LANG` environment variable to signal the presence of UTF-8 is that environment variables are not available to the kernel. In a typical POSIX system, the kernel also needs to know the character encoding, because when the terminal is in "cooked"

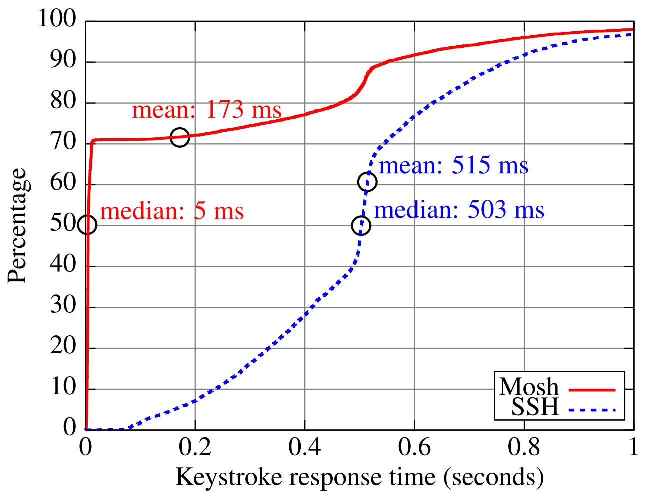
[5]SSH is typically now configured to do this as part of a separate environment-variable exchange that occurs in addition to the mandatory `TERM` field.

mode, the tty driver itself processes delete characters by erasing the previous character before it is read by an application.

To do this, the kernel needs to know how many bytes to delete in the input buffer, which means it needs to be able to interpret the incoming character sequence. POSIX does not provide a facility to convey this information to the kernel, but Mac OS X and Linux both use the IUTF8 `termios` flag to instruct the kernel to interpret incoming text as UTF-8.

Unfortunately, there is no standardized mechanism to convey the IUTF8 flag over a remote terminal connection, and SSH does not do this. As a result, today, neither Mac OS X nor Linux properly handles the case of typing and then deleting a UTF-8 character over an SSH connection. If the last character has a UTF-8 encoding longer than one byte, the operating system deletes it on the screen but produces an invalid UTF-8 sequence in memory or on disk. Mosh corrects this by setting IUTF8 on the server side and using raw mode on the client side.

Figure 6: Cumulative distribution of keystroke response times with Sprint EV-DO (3G) Internet service



4 Results

We evaluated Mosh using traces contributed by six users, covering about 40 hours of real-world usage and including 9,986 total keystrokes. These traces included the timing and contents of all writes from the user to the host and vice versa. The users were asked to contribute "typical, real-world sessions." In practice, the traces include use of popular programs such as the `bash` shell and `zsh` shells, the `alpine` and `mutt` e-mail clients, the `emacs` and `vim` text editors, the `irssi` and `barnowl` chat clients, the `links` text-mode Web browser, and several programs unique to each user.

To evaluate typical usage of a "mobile" terminal, we replayed the traces over an otherwise unloaded Sprint commercial EV-DO (3G) cellular Internet connection in Cambridge, Mass. A client-side process played the user portion of the traces, and a server-side process waited for the expected user input and then replied (in time) with the prerecorded server output. We speeded up long periods with no activity. The average round-trip time on the link was about half a second.

We replayed the traces over two different remote shell applications, SSH and Mosh, and recorded the user interface response latency to each simulated user keystroke, as seen by the user. The Mosh predictive algorithm and SSP were frozen prior to collecting the traces and were not adjusted in response to their contents or results.

The cumulative distributions of keystroke response time are shown in Figure 6. Mosh reduced the median keystroke response time from 503 ms to nearly instant (because more than half the keystrokes could be immediately displayed), and reduced the mean keystroke response time from 515 ms to 173 ms.

When Mosh was confident enough to display its predictions, the response was nearly instant. This occurred about 71% of the time. But many of the remaining keystrokes were "navigation," such as moving to the next e-mail message, and Mosh cannot make a prediction in these cases. For keystrokes it could not predict, Mosh's latency distribution was similar to that of SSH.

Mosh displayed an erroneous prediction 0.9% of the time. Some of these occurred because of a timeout (the keystroke eventually *was* echoed by the server), and the rest generally occurred because Mosh predicted an insertion when the user was in an overstrike mode, or was in a limited-size input window that did not extend to the edge of the screen (as displayed by the `links` Web browser), a condition not anticipated in our own testing before we saw the user traces. In future work, we plan to make Mosh more robust against these scenarios. In each case, the erroneous prediction was removed from the screen within a round-trip time.

4.1 Appropriateness of timing parameters

We also used the user traces to examine our choice of timing parameters for the SSP sender—in particular, the 100 ms timeout for delayed ACKs and the 15 ms interval after receiving a write from the host to collect writes that may be following in close succession. For this analysis, we disregard the possible benefits of speculative local echo and focus on network performance.

Figure 7 shows the distribution of the interval between when a user keystroke is supplied to the host and when

Figure 7: Cumulative distribution of server-side host response time to received user keystroke

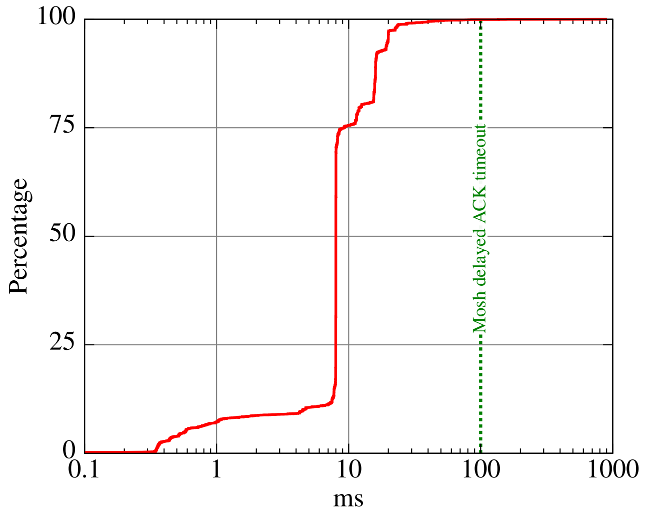
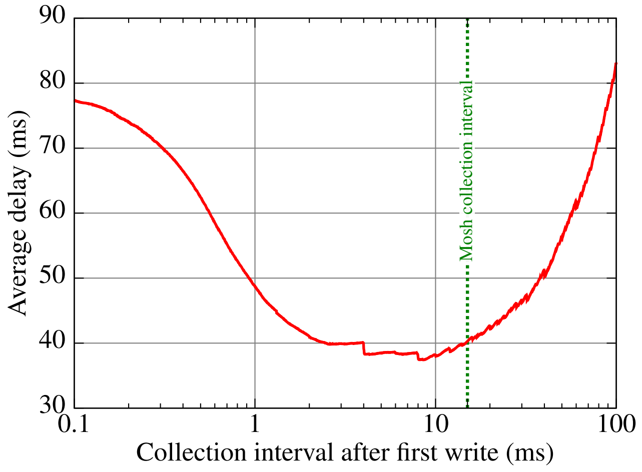


Figure 8: Average protocol-induced delay from varying collection interval (with frame interval of 250 ms)



it first responds—either an echo from the tty driver itself or from a host application. More than 99.9% of the time, the host has data ready to send to the client before 100 ms have elapsed, meaning that the server's ACK was able to piggyback on outgoing data.

Figure 8 shows the artificial delay introduced by the Mosh server on the host's screen updates in our traces. Recall that the server obeys two rules: always wait at least the framerate interval after a previous frame, and always wait at least the "collection interval" after receiving an initial write from the host. The server waits 15 ms after an initial write to collect the host's data before sending an instruction to the user. This parameter was chosen to be just smaller than the minimum frame rate interval (20 ms) and represents a tradeoff. Too short could cause the server to send a tiny initial instruction and then wait for the full framerate interval to elapse before sending more data.[6] But too long would hurt the responsiveness of a typical session. Based on these results, the choice of 15 ms appears to be near the sweet spot.

5 Related Work

TCP vs. an application-layer protocol atop UDP. Our view is that interactive applications that don't involve large data transfers are better served by a UDP-based protocols rather than TCP. The reason is that these applications do not usually have an offered load that causes the number of unacknowledged TCP segments to grow larger than a small number of segments, so most packet losses end up requiring a long (1 second or worse) timeout for the retransmission; fast retransmissions using triple duplicate ACKs [8] and selective ACKs [12] are not useful for such "small window" situations. One would do better using a TCP with "limited transmit" capabilities (RFC 3042) [2, 3], but even that will not be able to avoid the long timeouts. In contrast, our approach uses short timeouts, which can be as small as 50 ms if the round-trip time and deviation indicate that such a small value is unlikely to lead to a spurious retransmission. One might be tempted to introduce such short timeouts into TCP, but that would be problematic if the connection were being used for a bulk transfer. Because it runs at the application layer, SSP knows much more the workload, and is able to tune its timers to support its needs without burdening the network. By contrast TCP is far more general, and therefore has to be conservative in its operation.

Network protocols for mobile and intermittently connected hosts. Support for host mobility and migration in the face of changing IP addresses and intermittent connectivity is a problem that has received significant attention in the mobile computing and networking communities over the past two decades. Mobile IP [16, 17] introduces a level of indirection in packet forwarding, requiring each host to have a permanent "home" address, and arranging for all packets to be forwarded to the mobile host's current network location via this home agent. It achieves continuous TCP connectivity in the face of changes to a host's IP address, but does not handle intermittent connectivity, particularly multi-second or longer

[6]On the link in our tests, that interval was approximately 250 ms (4 Hz), because the average RTT was about 500 ms, and the frame interval is set at half smoothed RTT estimate. Additionally, 4 Hz is the smallest frame rate we allow.

outages.

Several network architectures propose the use of network-independent invariant end-point identifiers (EIDs) as a way to handle network mobility, including NIMROD [5] and HIP [14]. The TCP Migrate scheme [21] developed an end-to-end approach to host mobility, requiring no changes to the routing infrastructure. All these efforts provide semantics similar to Mobile IP, but using a different set of protocols and a different architecture.

In terms of intermittent connectivity, Snoeren et al. developed a session-layer approach, showing how SSH sessions could work across long periods of disconnection [20, 22]. This work required changes to the kernel at both ends, as well as application modifications. In contrast, Mosh solves the problem for a specific application class in a more efficient and simpler fashion.

REX [9], a secure remote execution protocol built on the Self-certifying File System [13], supports client and server roaming and intermittent connectivity over TCP by way of resumable connections carrying remote procedure calls. If the underlying TCP connection aborts or times out, the client attempts to initiate a new TCP connection to the same DNS hostname as it originally requested. Upon re-establishment, each side resends all unacknowledged RPCs, which are kept in a replay cache. REX requires a TCP timeout or error to activate its roaming mechanism, unlike SSP, which roams automatically to whichever address the server has seen most recently from the client.

Datagram Transport-Layer Security (TLS). Mosh uses an application-layer protocol over UDP, SSP, to synchronize state between the client and server. One might ask why our system incorporates its own cryptography module, rather than using Datagram TLS [19]. The main reason is that DTLS does not support roaming, especially if the client is not required to know it has roamed. In addition, in contrast to our approach, DTLS requires public-key cryptography and a replay cache, which our approach is able to avoid, leading to a simpler design and implementation.

On speculation: Some BSD-style operating systems support the LINEMODE option [4] for TELNET, in which character echoing and line editing is performed by the client. Unfortunately, this feature has fallen out of use for several reasons. LINEMODE doesn't work with programs that put the terminal into "raw" mode, including popular libraries like readline, shells like bash, and full-screen applications like text editors and e-mail readers. The Linux kernel did not add the necessary support for TELNET LINEMODE until 2010 [6], and SSH does not have an equivalent of LINEMODE.

SUPDUP [23] included an elaborate Local Editing Protocol, in which an entire text editor session could be executed locally and uploaded to the server in batches. SUPDUP required the host application to encode its interactive functionality in the SUPDUP language to instruct the terminal what to do in response to each user keystroke, which characters should be considered white space for purposes of word wrapping, etc. In practice, only EMACS on ITS ever implemented the SUPDUP Local Editing Protocol. Mosh's advantage is that it does not require modifications to host applications and nonetheless handles most typing and cursor movement keystrokes immediately.

6 Conclusion

This paper presented the design, implementation, and evaluation of Mosh, a mobile shell that performs well over high- and variable-delay network paths and is a better solution for mobile clients than current approaches such as SSH. Mosh handles intermittent connectivity and changes in IP addresses without losing the terminal session; for example, one can use it to resume a terminal session on a laptop that was initiated at home, even when the user suspended her laptop and reconnected at work from a different IP address. The session will resume correctly even if there had been unacknowledged session data when the laptop was suspended.

Mosh also provides good interactive performance when used over long-delay network paths. In our empirical evaluation of about 40 hours of keystroke activity from six users, we found that over a Sprint EV-DO (3G) connection, the median response time to a keystroke was 503 ms with SSH, but only 4.8 ms with Mosh. The mean response time was 515 ms with SSH, and 173 ms with Mosh. The reason for these improvements in interactivity is Mosh's speculative local echo of keystrokes, which accurately predicted the response to 70% of user keystrokes without needing a response from the server. Mosh displayed incorrect predictions 0.9% of the time, which it corrected after at most one RTT.

In conclusion, we believe that the ideas described in this paper extend beyond the interactive mobile shell and terminal application. The underlying principle is that by using a *state-oriented* transport layer, as opposed to a reliable octet-stream layer, one can embed application semantics into the protocol without making the protocol itself application-specific. The API we advocate provides the ability to *synchronize state* between client and server (and vice versa), rather than an abstraction that fully decouples the transport mechanism (SSH-over-TCP) from the application (the remote terminal), with the application treating the transport as a black box.

Our results with Mosh suggest that this idea of decomposing the problem into a lower layer that handles state synchronization is well-suited for mobile applications over "challenged" networks, and is potentially superior for interactivity to current approaches that use a reliable octet-stream (such as TCP or SSH-over-TCP). Other mobile applications with similar goals may benefit from a similar approach.

7 Acknowledgments

We thank Nickolai Zeldovich for helpful comments on this work, and Richard Stallman for helping us understand the capabilities of SUPDUP. We also thank the users who provided us with keystroke traces used to evaluate Mosh. This work was supported in part by NSF grants 1040072 and 0721702.

References

[1] *Control Functions for Coded Character Sets.* ECMA-48 (1991); ISO/IEC 6429:1992.

[2] M. Allman, H. Balakrishnan, and S. Floyd. Enhancing tcp's loss recovery using limited transmit. Technical report, RFC 3042, January, 2001.

[3] H. Balakrishnan. *Challenges to reliable data transport over heterogeneous wireless networks*. PhD thesis, University of California, Berkeley, 1998.

[4] D. Borman. Telnet linemdoe option. RFC 1116, 1990.

[5] I. Castineyra, N. Chiappa, and M. Steenstrup. Rfc 1992: The nimrod routing architecture. 1996.

[6] H. Chu. tty: Add extproc support for linemode. Linux Git commit 26df6d13, 2010.

[7] M. Crispin. Supdup display protocol. RFC 734, 1977.

[8] V. Jacobson. Congestion Avoidance and Control. In *SIGCOMM*, 1988.

[9] M. Kaminsky, E. Peterson, D. B. Giffin, K. Fu, D. Mazières, and M. F. Kaashoek. REX: Secure, Extensible Remote Execution. In *Proceedings of the 2004 USENIX Annual Technical Conference (USENIX '04)*, pages 199–212, Boston, MA, June 2004.

[10] B. Kantor. Bsd rlogin. RFC 1282, 1991.

[11] T. Krovetz and P. Rogaway. The software performance of authenticated-encryption modes. In *Proceedings of the 18th international conference on Fast software encryption*, FSE'11, pages 306–327, Berlin, Heidelberg, 2011. Springer-Verlag.

[12] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. *TCP Selective Acknowledgment Options*, 1996. RFC 2018.

[13] D. Mazieres. *Self-certifying File System.* PhD thesis, Massachusetts Institute of Technology, May 2000.

[14] P. Nikander, T. Henderson, C. Vogt, and J. Arkko. End-host mobility and multihoming with the host identity protocol. *RFC 5206*, April 2008.

[15] V. Paxson, M. Allman, J. Chu, and M. Sargent. Computing tcp's retransmission timer. RFC 6298, 2011.

[16] C. Perkins. Mobile ip. *Communications Magazine, IEEE*, 40(5):66–82, 2002.

[17] C. Perkins et al. Rfc 3220: Ip mobility support for ipv4. *IETF, jan*, 2002.

[18] J. Postel and J. Reynolds. Telnet protocol specification. RFC 854, 1983.

[19] E. Rescorla and N. Modadugu. Datagram transport layer security. RFC 4347, 2006.

[20] A. Snoeren. *A session-based approach to Internet mobility*. PhD thesis, Massachusetts Institute of Technology, 2002.

[21] A. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 155–166. ACM, 2000.

[22] A. Snoeren, H. Balakrishnan, and M. Kaashoek. Reconsidering internet mobility. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 41–46. IEEE, 2001.

[23] R. M. Stallman. The supdup protocol. Technical report, MIT AI Memo 644, 1983.

[24] J. VanBokkelen. Telnet terminal-type option. RFC 1091, 1989.

[25] T. Ylonen. Ssh–secure login connections over the internet. In *Proceedings of the 6th USENIX Security Symposium*, pages 37–42, 1996.

[26] T. Ylonen. The secure shell (ssh) connection protocol. RFC 4254, 2006.