Testing Mosh's Assumptions: Evaluating Reference State Selection Strategies

Aaron Hsu, Mimi Liao, Arvindh Manian, John Schappert, Islam Tayeb

Abstract

Mobile shell (Mosh) improves upon SSH for remote terminal access in mobile environments by using the State Synchronization Protocol (SSP), a UDP-based protocol that maintains terminal state across network disruptions and IP address changes. SSP uses the most recently sent state within the retransmission timeout window (the "assumed receiver state") as the reference when generating differential updates. However, this strategy may be suboptimal under high packet loss, where packets can be lost before acknowledgment, potentially causing wasted transmissions. We propose a λ-parameterized extension that probabilistically selects between the assumed receiver state and the last acknowledged state (the "known receiver state"), hypothesizing that a more conservative reference selection strategy could improve Age of Information (AoI) under packet loss.

We implemented a simplified SSP version preserving core mechanisms (RTT estimation, dynamic RTO calculation, differential updates) and developed a Docker-based testbed using Linux traffic control to emulate extreme mobile network conditions with 1000ms base delay and 75% packet loss. Testing three configurations ($\lambda \in \{0, 0.5, 1.0\}$) over 10 iterations each, we found all achieved similar median AoI performance (≈ 1.1s) with comparable high-latency outliers (4.0–14.2s). Notably, $\lambda = 0.5$ exhibited slightly elevated average latency (2.1s) compared to $\lambda = 0$ (1.7s) and $\lambda = 1.0$ (1.8s). These findings suggest that reference state selection strategy has minimal impact on AoI under moderate packet loss, and that SSP's original timeout-based approach may already be near-optimal for this metric in typical mobile environments.

1 Introduction

1.1 The Need for Mosh

Although SSH has provided a secure and reliable de-facto solution for remote terminal access since the early days of computer networking, it operates over TCP and is ill-suited for use by mobile devices. When network connectivity is intermittent or when a client device changes its IP address (both common occurrences in mobile environments), SSH sessions will freeze or terminate entirely. Additionally, SSH operates in a strict character-at-a-time mode, where every keystroke must complete a round trip before being echoed back to the user. In high latency, higher packet loss mobile environments, this can make interactive work nearly impossible.

To address these limitations, Winstein and Balakrishnan introduced Mosh (mobile shell) in 2012 as a solution to mobile remote terminal access [2]. Mosh is built on the State Synchronization Protocol (SSP), a novel UDP-based protocol that synchronizes terminal state between client and server instead of transmitting byte streams [2]. This project proposes an extension to Mosh (the mobile shell proposed in 2012 by Winstein and Balakrishnan). Mosh is a mobile terminal application that supports intermittent connectivity and roaming for clients using SSP. SSP is a UDP-based protocol that associates mon7otonic increasing sequence numbers with states and sends "diffs" between successive states. Two important concepts in SSP are the known receiver state and the assumed receiver state. The known receiver state is the last state that has been acknowledged by the client, and the assumed receiver state is the last state that was sent to the client for which an RTO has not yet expired. In the design presented in the original Mosh paper, the assumed receiver state is always used as the base state for the diff in SSP.

1.2 Proposed Extension

This choice to use only the assumed receiver state is not well justified in the paper. The authors assume that a given state will be received, and thus opt to send diffs based on the assumed receiver state. This does have the benefit of likely minimizing bandwidth usage. But it is not always true (especially in mobile environments) that a sent state will be received. When a packet delivering a state to a client is lost, another packet including a diff starting at that sent state might also be sent due to this assumed receiver state policy. A diff starting at a state that the client is not synced to is not useful by the client and must be discarded. So, when a state packet is lost, an additional useless packet may be sent, causing no progress to be made until an additional packet is sent starting from the correct state after the timeout. To prevent this inefficiency from occurring in higher packet loss environments, we explore the addition of a λ parameter to Mosh with the following effect:

With probability λ, use the known receiver state instead of the assumed receiver state.

Note that the $\lambda = 0$ case is equivalent to the approach suggested in the paper of always using the assumed receiver state.

2 Background

Mobile broadband networks experience significantly higher packet loss compared to stationary scenarios. As shown by Baltrunas et. al in 2016, packet rates in mobile environments can be dramatically elevated, with loss occurring in 30-50% of five-minute measurement intervals under mobility compared to only 2-12% while stationary [1].

The primary reason for mobile packet loss is Radio Access Technology (RAT) changes, which account for approximately 70% of overall packet loss under mobility. These inter-RAT handovers (transitions between LTE, 3G, and 2G networks) involve complex coordination between devices. Additionally, loss in these environments is exacerbated by Location Area Code (LAC) changes as well as coverage gaps and temporary loss of service [1]. The interaction between all of these systems leads to substantially higher loss rates in mobile deployments. These characteristics motivated us to attempt to improve Mosh performance in high packet loss mobile environments.

3 Methods

To evaluate our proposed λ parameter extension, we implemented a simplified version of the Mosh State Synchronization Protocol. Our implementation retains the core mechanisms necessary to test our hypothesis about reference state selection under packet loss while eliminating components orthogonal to this research question.

3.1 Metrics

Our main metric is the Age of Information (AoI) [2]. We measure how age of the information the receiver has, on average. This is a common metric for this kind of sender/receiver use case, and it makes sense to measure for our use case (an interactive terminal, where any delay in a user's command executing is felt directly by the user).

We run a variety of tests and measure the average peak AoI in each.

3.2 Core SSP Components Retained

Our implementation preserves the essential elements of the State Synchronization Protocol:

- **UDP-based datagram layer** with sequence numbers and timestamps
- **RTT estimation** using TCP-style SRTT and RTTVAR calculations with parameters $\alpha = 0.125$, $\beta = 0.25$, $K = 4$, and $G = 0.1$
- **Dynamic RTO calculation** with a minimum threshold of 50ms (following the original Mosh design)
- **Transport instructions** containing `old_num`, `new_num`, `ack_num`, `throwaway_num`, and `diff` fields
- **In-flight state tracking** to maintain dependency graphs and manage unacknowledged states
- **Differential state updates** using `difflib.SequenceMatcher` to generate patches between states

These components are sufficient to evaluate how different reference state selection strategies affect Age of Information (AoI) under varying network conditions.

3.3 State Object Simplification

The original Mosh implementation maintains a complex terminal state including a framebuffer, cursor position, character renditions (bold, underline, colors), Unicode combining sequences, and window dimensions. Our implementation replaces this with simple string states.

This simplification is appropriate for several reasons. First, the behavior of the SSP protocol with respect to reference state selection is independent of the semantic content of the states being synchronized—what matters is the dependency structure between states and the size of diffs. Second, using simple strings allows us to control the diff size distribution precisely. Third, terminal emulation complexity would add implementation burden without contributing to our research question about the λ parameter.

Our test states consist of up to 250-character strings that cycle through a predetermined sequence, providing realistic diff sizes representative of terminal screen updates in interactive applications. The strings are designed to have varying edit distances between successive states, generating diffs of different sizes to simulate the range of update patterns seen in real terminal usage. The diff generation uses Python's `difflib.SequenceMatcher` to produce JSON-encoded patches with operations: `equal`, `delete`, `insert`, and `replace`.

3.4 Elimination of Timing Controls

The original Mosh implementation includes several timing mechanisms to optimize network utilization and user experience:

- A framerate interval (20–250ms) set at half the SRTT to maintain approximately one instruction in flight

- Delayed acknowledgments with a 100ms timeout to allow piggybacking on data packets
- A 15ms collection interval to batch rapid successive updatesk
- Heartbeat packets every 3 seconds to maintain NAT bindings and detect roaming

We omitted these timing controls from our implementation as they were not directly relevant to our research objectives and would have required significant development effort. Our focus was on isolating the effect of the λ parameter on AoI under varying packet loss conditions. The controlled testbed environment and automated testing approach rendered these optimizations unnecessary for our experimental goals.

3.5 Packet Size and Fragmentation

The original Mosh implementation fragments transport instructions exceeding 1400 bytes to accommodate paths with MTUs smaller than the standard 1500-byte Ethernet MTU. Our 500-character state strings and their JSON-encoded diffs fit comfortably within the typical 1500-byte MTU. Even in the worst case where an entire state must be transmitted (diff size equal to state size), the payload remains well under the 1400-byte threshold that Mosh uses to trigger fragmentation.

In our testbed, successive states in the test sequence share substantial common subsequences, producing diffs that are typically much smaller than the full state size. Combined with the standard 1500-byte MTU in our Docker network environment, all transport instructions are transmitted as single unfragmented UDP datagrams. Application-level fragmentation logic is therefore unnecessary for our experiments.

4 Testbed

4.1 Overview and Purpose

To evaluate the performance of our proposed λ-parameterized extension to SSP under varying network conditions, we developed a comprehensive Docker-based testbed that enables controlled, reproducible experiments. The testbed accomplishes three primary objectives: (1) emulating realistic network conditions characteristic of mobile and unstable environments, (2) automating the execution of our mosh-lite implementation under these conditions, and (3) collecting granular latency measurements for analysis.

The testbed is designed to test the state synchronization protocol across a wide range of network parameters including variable latency, packet loss, jitter, and bandwidth constraints. This allows us to systematically evaluate how different values of λ affect the AoI metric under conditions representative of WiFi roaming, mobile networks, and satellite links.

4.2 Architecture

The testbed employs a containerized architecture with network emulation capabilities, illustrated in Figure 1. The system consists of three main components: the container infrastructure, the network controller, and the application layer.

4.2.1 Container Infrastructure

The testbed uses Docker Compose to orchestrate two containers: a client container and a server container, both running Ubuntu 24.04 with Python 3 and network emulation tools. Each container is granted `NET_ADMIN` capability, which is essential for applying traffic control rules to emulate network conditions.

Both containers share three volumes: (1) `scenarios/` for test configuration files, (2) `artifacts/` for control signals and metadata, and (3) `logs/` for application output. The containers communicate over a user-defined bridge network (`netem-net`), allowing us to apply network emulation rules to all traffic between client and server.

The containers include `iproute2` for the Linux traffic control utility (`tc`), which provides the foundation for network emulation via the `netem` (network emulation) kernel module.

4.2.2 Network Controller

The network controller (`netem_controller.sh`) is the orchestration engine that reads JSON scenario files and applies the specified network conditions to both containers. Each scenario file defines a sequence of network states, where each state specifies:

- `delay_ms`: Base round-trip time delay in milliseconds
- `jitter_ms`: Delay variation with normal distribution
- `loss_pct`: Packet loss percentage (0–100)
- `rate_kbit`: Bandwidth limit in kilobits per second
- `duration_s`: Duration to maintain this network state

The controller applies these parameters using two complementary mechanisms. For bandwidth limiting, it configures a Token Bucket Filter (TBF) on the network interface. For delay, jitter, and loss, it uses the `netem` queuing discipline. These can be combined using hierarchical traffic control, where TBF serves as the root qdisc and netem is attached as a child.

Once the network rules are applied to both containers, the controller creates a sentinel file (`artifacts/netem_ready.json`) to signal to the application layer that testing can begin. The controller then progresses through the scenario steps, applying each network configuration for its specified duration while logging all actions to `artifacts/controller_log.txt`.

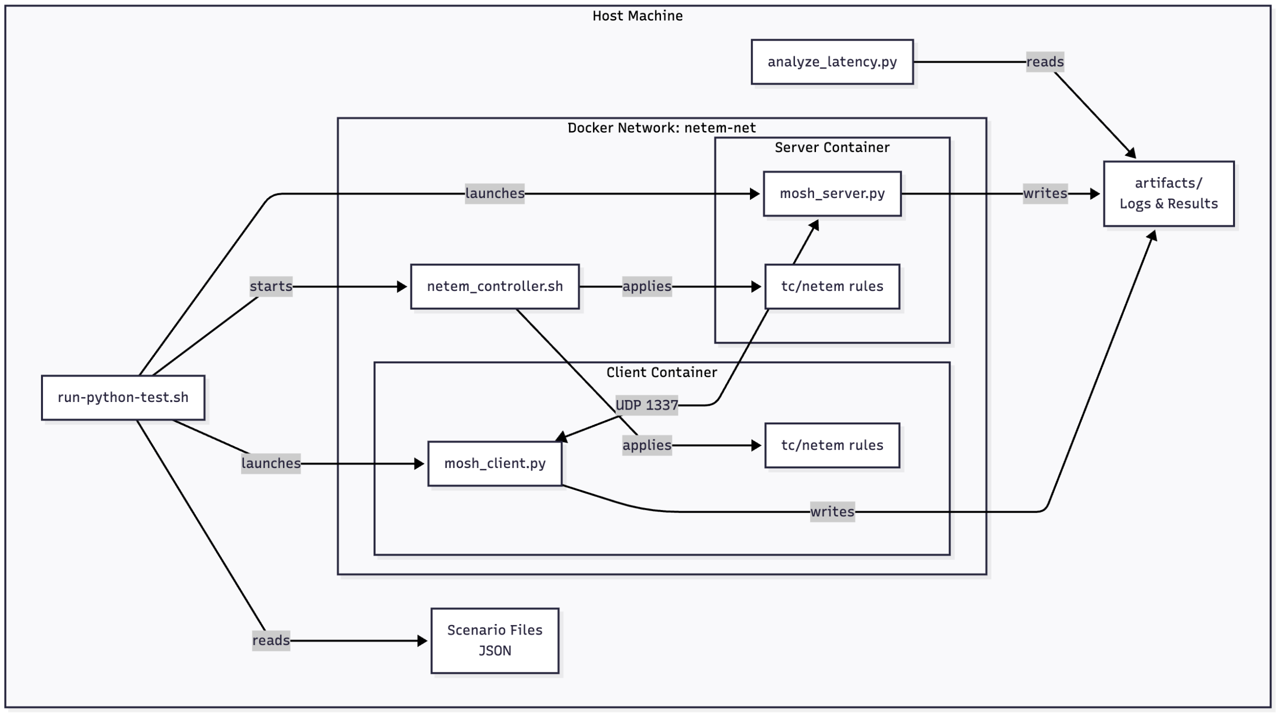

Figure 1: *Testbed architecture showing the Docker-based setup with network emulation components.*

4.2.3 Application Layer

The application layer consists of two components: `mosh_server.py` and `mosh_client.py`, which wrap the core SSP receiver and sender implementations respectively.

The server application waits for the network-ready signal from the controller, then initializes a UDP receiver on port 1337. It logs each received state number and the timestamp of receipt to `/logs/output.log` in CSV format.

The client application similarly waits for the network-ready signal, then begins an automated message generation process. It cycles through a predetermined sequence of states (`["abc", "cde", "edf", "adsfhadsf", "fgi"]`) using `itertools.cycle`, sending a new state every 50 milliseconds (20 messages per second). This sending rate was chosen to be representative of interactive terminal usage while providing sufficient data points for statistical analysis. The client logs each sent state number and timestamp to `/logs/client_out.log`, also in CSV format.

4.3 Test Execution Flow

The complete test execution follows the sequence shown in Figure 2.

The main test runner (`run-python-test.sh`) first builds the Docker images, ensuring all dependencies are installed. It then starts both containers and launches the network controller in the background. The controller reads the scenario file specified by the `SCENARIO` environment variable (defaulting to `scenarios/high_delay_test.json`), applies the initial network conditions, and creates the ready signal.

Once both applications detect the ready signal, the server begins listening and the client begins its automated sending loop. The test continues for the total duration specified in the scenario file (the sum of all step durations). After completion, the runner collects logs from both containers and invokes the analysis script.

4.4 Metrics Collection and Analysis

Our primary metric, AoI of timeline catch-up, is computed from the timestamped logs produced by both client and server. It measures when the receiver's state becomes at least as fresh as each sent packet: if packets 1-100 are sent but only packets 1 and 100 are acknowledged, then packet 1's AoI is its normal one-way delay, while packets 2-99 each have an AoI calculated as the time between when they were sent and when packet 100 arrived (since packet 100's arrival is when the receiver finally caught up past those states). This captures the intuition that even though packets 2-99 were lost, the receiver's display remained stale until packet 100 updated it, making the AoI reflect the actual information staleness experienced by the end user. The analysis script (`analyze_latency.py`) implements the following procedure:

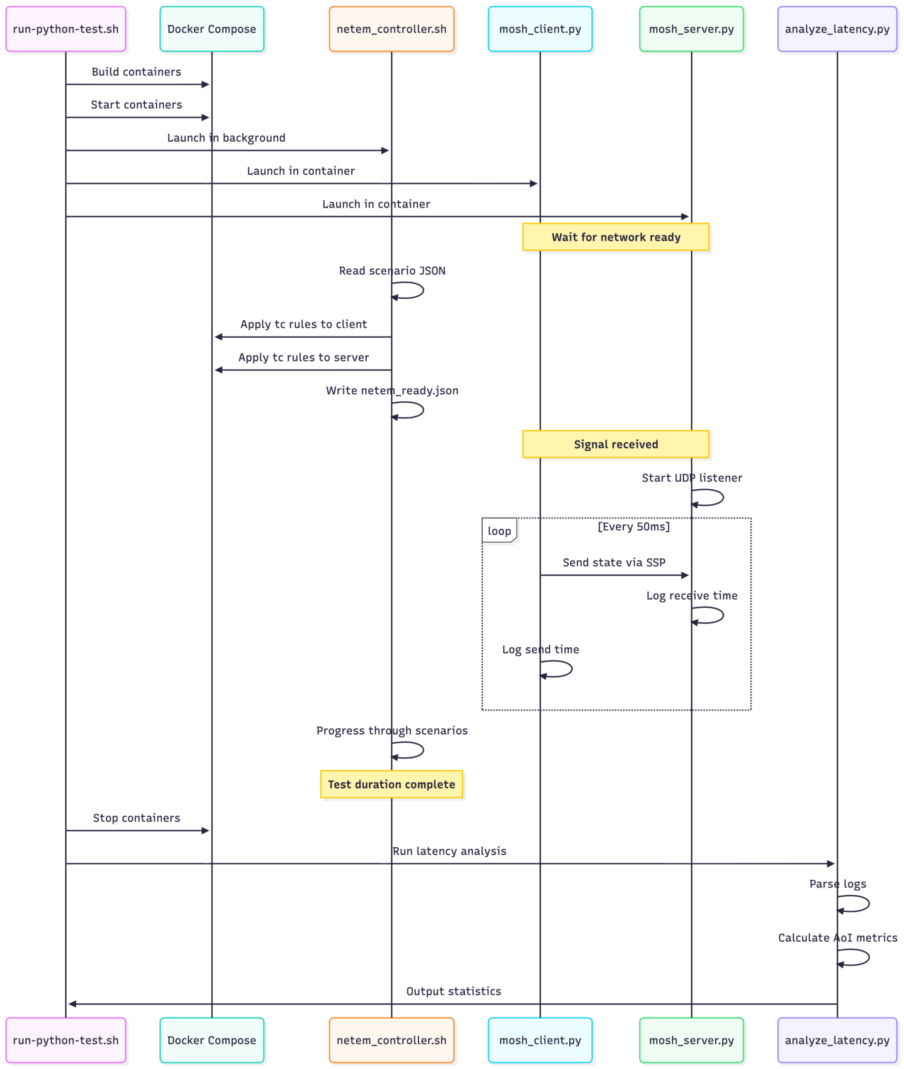

Figure 2: *Sequence diagram showing the test execution flow from initialization through analysis.*

1. Parse `client_out.log` to extract (t_i, s_i) pairs, where t_i is the send timestamp and s_i is the state number
2. Parse `output.log` to extract (r_j, s_j) pairs, where r_j is the receive timestamp and s_j is the received state number
3. For each sent state s_i, identify the containing state c_i – the first received state s_j where $j \geq i$
4. Calculate the AoI as $r_{c_i} - t_i$
5. Apply backfilling for states that were never directly received, using the timestamp of the last known received state. If the server does not receive the last several packets, they are not included in the latency calculation.
6. Compute aggregate statistics: mean AoI, median AoI, and output the full per-state latency data

The script outputs both summary statistics to stdout and a complete CSV file (`latency.csv`) containing per-state latency measurements for further analysis.

4.5 Experimental Scenarios

We designed several test scenarios to evaluate our λ parameter across different network conditions:

- **Baseline**: 10ms delay, 0% loss, 10 Mbps – establishes minimum latency
- **Variable conditions**: Cycles through good (150ms, 0% loss), degraded (200ms, 5% loss), and recovery phases
- **High delay**: 1 s delay with 75% loss – stress test for extreme conditions
- **Packet loss sweep**: Fixed delay (100ms) with varying loss rates (0%, 10%, 25%, 50%)

For each scenario, we tested multiple values of $\lambda \in \{0, 0.5, 1\}$, where $\lambda = 0$ represents the original Mosh behavior (always use assumed receiver state), and $\lambda = 1$ represents always using the known receiver state. The intermediate values provide a probabilistic mix of both strategies.

4.6 Reproducibility and Extensibility

The testbed design prioritizes reproducibility through containerization and declarative scenario files. All network conditions are specified in version-controlled JSON files, and the Docker containers ensure a consistent execution environment across different host machines.

The architecture is easily extensible to test additional parameters. New network conditions can be added by creating scenario files, and modifications to the SSP implementation (such as different λ values or alternative reference state selection strategies) require only rebuilding the containers. The separation between network emulation, application logic, and analysis also allows each component to be modified or replaced independently.

5 Results

The results we report are based on the high delay test. We evaluated our proposed λ-parameterized extension to SSP by testing three configurations: $\lambda = 0$ (original Mosh behavior using assumed receiver state), $\lambda = 0.5$ (mixed strategy), and $\lambda = 1.0$ (always using known receiver state). Each configuration was tested over 10 iterations in a controlled network environment with 1s base delay and 75% packet loss to simulate extreme mobile network conditions.

Figure 3 presents the latency distributions for both median and average AoI metrics across the three λ values. The median latency (left panel) shows remarkably consistent performance across all three configurations, with typical values clustering tightly around 1.1 s. However, all configurations experienced occasional high-latency outliers exceeding 10 s (4.0–13.1 s), indicated by the $\times$ markers. These outliers likely represent instances where multiple consecutive packets were lost, forcing the receiver to wait for a retransmission timeout before state synchronization could proceed.

The average latency (right panel) exhibits greater variability than the median, with interquartile ranges spanning 1.4–2.6 s depending on configuration. Notably, $\lambda = 0.5$ shows the highest median average latency at 2.1 s compared to 1.7 s for $\lambda = 0$ and 1.8 s for $\lambda = 1.0$. All configurations produced high-latency outliers in the 8.1–14.2 s range, with $\lambda = 0$ showing slightly lower outlier values (8.1, 13.8 s) compared to $\lambda = 0.5$ and $\lambda = 1.0$ (both producing multiple outliers above 14 s).

These results suggest that the choice of λ parameter has minimal impact on typical-case performance under even extreme packet loss, as evidenced by the nearly identical median latencies across all configurations. The slightly elevated latencies observed with $\lambda = 0.5$ may indicate that the mixed strategy introduces additional overhead without providing clear benefits in this packet loss regime. The presence of high-latency outliers across all configurations highlights the fundamental challenge of maintaining low AoI under packet loss, regardless of reference state selection strategy.

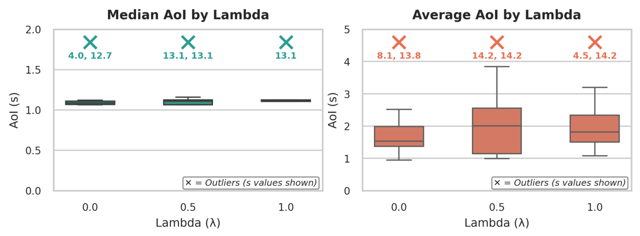

Figure 3: *Latency distribution by λ value showing median (left) and average (right) AoI. Box plots show the interquartile range with whiskers extending to non-outlier extrema. This is for packet loss conditions of 75% (extreme to show difference). Outliers (≥ 2 s for median, ≥ 4 s for average) are marked with $\times$ and their values displayed.*

6 Future Work

There are several promising directions for improving this work. First, the discrepancy between our expected performance gains and the observed results warrants deeper investigation. Our intuition suggested that the proposed modifications should help in high-loss environments, but we were unable to identify the source of the mismatch between theory and measurements.

Second, our prototype was a reimplementation of Mosh rather than a modification of the original codebase. Although we examined substantial portions of the Mosh source, reimplementing the protocol introduced uncertainty regarding behavioral fidelity. Future efforts should build directly on the official Mosh implementation to ensure correctness and to take advantage of its existing timing, rate control, and congestion-avoidance mechanisms, which would likely improve the validity of performance measurements.

Third, we would like a further study of realistic packet loss conditions in roaming scenarios to ensure that we properly analyze the benefits and costs of setting λ. We suspect that there exists an analytical solution to finding the optimal lambda as a function of packet loss rate, but we were unable to derive it.

Finally, several potential enhancements to Mosh itself remain unexplored. The RTT estimator is derived from TCP's algorithm, and tailoring it to Mosh's specific interactive and lossy-network use case may yield benefits. Similarly, Mosh's use of protocol buffers could be revisited: applying compression or more compact encodings may further reduce bandwidth consumption. These ideas merit systematic evaluation in a larger, more realistic test environment.

References

[1] Dziugas Baltrunas, Ahmed Elmokashfi, Amund Kvalbein, and Özgü Alay. Investigating packet loss in mobile broadband networks under mobility. In *2016 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 225–233, 2016.

[2] Roy D. Yates, Yin Sun, D. Richard Brown III, Sanjit K. Kaul, Eytan H. Modiano, and Sennur Ulukus. Age of information: An introduction and survey. *CoRR*, abs/2007.08564, 2020.