EvLag – A Tool for Monitoring and Lagging Linux Input Devices

Shengmei Liu
sliu7@wpi.edu
Worcester Polytechnic Institute
Worcester, Massachusetts, USA

Mark Claypool
claypool@wpi.edu
Worcester Polytechnic Institute
Worcester, Massachusetts, USA

ABSTRACT

Understanding the effects of latency on interaction is important for building software, such as computer games, that perform well over a range of system configurations. Unfortunately, user studies evaluating latency must each write their own code to add latency to user input and, even worse, must limit themselves to open source applications. To address these shortcomings, this paper presents EvLag, a tool for adding latency to user input devices in Linux. EvLag provides a custom amount of latency for each device regardless of the application being run, enabling user studies for systems and software that cannot be modified (e.g., commercial games). Evaluation shows EvLag has low overhead and accurately adds the expected amount of latency to user input. In addition, EvLag can log user input events for post study analysis with several utilities provided to automate studies and facilitate output event parsing.

CCS CONCEPTS

• **Software and its engineering** → **Input / output**; *Interactive games.*

KEYWORDS

delay, latency, game, lag

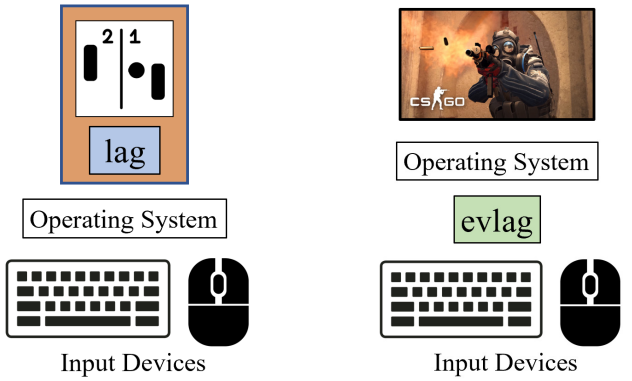
ACM Reference Format:
Shengmei Liu and Mark Claypool. 2021. EvLag – A Tool for Monitoring and Lagging Linux Input Devices. In *12th ACM Multimedia Systems Conference (MMSys '21) (MMSys 21), September 28-October 1, 2021, Istanbul, Turkey.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3458305.3478449

1 INTRODUCTION

Real-time games require players to make many time-sensitive actions that can suffer when the computer responses lag behind player input. Even latencies as small as milliseconds can hamper the interplay between players' actions and intended results. For example, latency when aiming a virtual weapon can make it difficult for a player to hit a moving target in a shooting game, hurting the player's score and degrading the quality of experience. Unlike in traditional network games where a client can potentially act on user input immediately, cloud-based games have latency for all user input by at least a round-trip time to the server [6].

MMSys 21, September 28-October 1, 2021, Istanbul, Turkey

ACM ISBN 978-1-4503-8434-6/21/09…$15.00
https://doi.org/10.1145/3458305.3478449



(a) State of the art – adding delay to game

(b) EvLag – adding delay to game

While there are established methods to compensate for latency [3, 13, 14], an understanding of how latency affects player actions in games is needed in order to choose the most effective latency compensation techniques and, if possible, to develop and apply new techniques. Moreover, understanding the impact of latencies may help better inform competitive gamers about system purchases and guide researchers on building systems to help with game and game-like applications.

The state of the art for studying the effects of latency is typically with a user study. In such a study, participants play a game or use an interactive system with different amounts of latency. The key to a successful study is the ability to add controlled amounts of latency to the base system. Most often, latency is added with custom software built into a small game or interactive application, meaning the needed functionality must be (re)-implemented for each game, or at least for each programming language (e.g., Python, Java, Processing, Javascript) and for each game engine (e.g., Unity or UE4). Even worse, manually adding latency code does not work when studying the many applications where the source code is not available (e.g., commercial games). Solutions that add network latency (e.g., netem [32]) do not work either, since in traditional game architectures the game client can act on input before network latency is added.

Figure 1(a) depicts a summary of the state of the art. In order to add latency to user input, researchers must modify the game (or other software), where the "lag" in the picture represents code that is inside the application. For each application studied, researchers must implement a method to add latency to the user input, typically done by intercepting input events in-game and delaying them in a queue before applying them. Not only does this require the same functionality to be re-implemented each time, but in some

systems (e.g., Javascript [25]), added latency can be highly variable. Moreover, having to modify the code restricts studies to use only custom and open source software, with closed source, commercial applications being unusable even if these are the targets of such research.

Figure 1(b) depicts the use of EvLag – a tool for adding latency and monitoring input for devices in Linux. Instead of modifying the game, EvLag sits between the input devices and the operating system, adding a custom amount of latency to each device (e.g., a keyboard and mouse). This allows researchers to add latency to any application, such as the depicted commercial first-person shooter game *CS:GO* (Valve, 2012), without modification. In addition, for researchers that want to monitor user input, EvLag provides a logfile of all input events (e.g., mouse clicks and timing) for each device. The EvLag repository includes EvParse and EvDetect for help with parsing logfiles and mapping devices, respectively, as well as sample logfiles.

Evaluation shows EvLag has minimal overhead and accurately adds the intended amount of latency. We have successfully used EvLag in a previous user study [15] and provide a virtual machine image that demonstrates EvLag's functionality.

The rest of this paper is organized as follows: Section 2 describes work related to measuring local latency; Section 3 details the implementation of EvLag and accompanying utilities; Section 4 provides a brief evaluation of EvLag's accuracy and overhead, prior use and a demo; and Section 5 summarizes the tool and presents possible future work.

2 RELATED WORK

This section presents work related to EvLag: tools adding latency (Section 2.1), studies of added latency (Section 2.2) and measuring local latency (Section 2.3).

2.1 Tools Adding Latency

There are various tools that add latency to network interfaces. A partial list of free/open source options include: dummynet [4], Mahimahi [18], Clumsy [31] and netem [32]. There are various commercial products that can add network latency, as well. These tools vary in their platforms (e.g., Linux versus Windows) and features (e.g., latency and latency jitter).

We are not aware of any other tools that add latency to computer input devices, in general.

2.2 Studies of Added Latency

There are many studies of the effects of latency on interactive applications that have added controlled amounts of latency and observed the effects on users.

Studies focusing on local latency [7, 8, 13, 16, 17, 20, 21, 29] typically use the technique depicted in Figure 1(a) where a custom application is written/modified to manually intercept and add latency to user input. This same technique is used for studies of latency in cloud-based game systems [26, 30] where, since all player input is delayed by local latency and network latency to the server, adding local latency can emulate network latency.

While such methods are effective for studying latency, these approaches have the significant downsides mentioned in the introduction, including a re-implementation for each study and, worse, the inability to work with closed source software (e.g., commercial games), unlike EvLag.

Studies focusing on network latency [1, 2, 10–12, 22, 28], including some with a focus on cloud-based game streaming [6, 9], typically use a network emulator (e.g., dummynet, Clumsy, or netem) to add delay to game packets sent over the network.

While these techniques avoid custom implementations of added latency and can work with closed-source software, they are only effective when studying network latency or, if used to study local latency, require considerable more complexity than the single-computer system needed by EvLag.

2.3 Measuring Local Latency

There are numerous papers that propose tools and techniques to measure local latency for systems and devices, with most also presenting measurement results using their techniques.

Raeen and Petlund [24] describe a method to measure local system latency using an oscilloscope and a photosensor. Their preliminary test results show local latencies vary by 10s of milliseconds and can be larger for games. Raeen and Kjellmo [23] apply the same measurement technique to virtual reality devices.

Ivkovic and Gutwin [13] use a high speed camera to measure latencies for 16 different game systems. For these systems, local latencies can vary from a low of about 20 milliseconds to a high of about 250 milliseconds.

Casiez et al. [5] present a real-time method for measuring end-to-end latency in graphical user interfaces using an optical mouse. Evaluation of some existing systems shows local latency is affected by the operating system and system load, with substantial differences between different libraries and different Web browsers.

Schmid and Wimmer [27] propose a method for measuring the end-to-end latency using a microcontroller with electrical contact to a mouse and a photo-resistor to capture the screen response. They show their measured local latency values are close to those for a high-speed smartphone camera (240 Hz).

Wimmer et al. [33] propose a method for measuring the latency of input devices and give measurements for 36 keyboards, mice and gamepads connected via USB. They show devices differ in their average local latencies and their latency distributions.

We use a high-speed camera technique to measure local latency, similar to Ivkovic and Gutwin [13], but do so with the base system and with EvLag adding different amounts of latency. Our base system has low latency, but can be used to study arbitrary additional latencies with EvLag.

3 EVLAG

EvLag provides a stand-alone program that adds latency (lag) to input devices in Linux. Features include:

- Adding a fixed (constant) amount of latency to any input device.
- Adding latency to more than one input device at a time.
- Logging input events to a file, one file per device.

Listing 1: Input event

```
struct input_event {
  struct timeval time;
  u16 type;
  u16 code;
  s32 value;
};
```

All code described, as well as examples and documentation, are available in the public EvLag `git` repository:

https://claypool@bitbucket.org/claypool/evlag.git

This section describes the EvLag implementation (Section 3.1), usage (Section 3.2), output (Section 3.3), utilities (Section 3.4) and license (Section 3.5).

3.1 Implementation

EvLag uses the Linux event devices (`evdev`) – interfaces that generalize raw input events from device drivers and make them available as character devices. These character devices are, by default, in the `/dev/input` directory on the Linux system.

EvLag accesses the `evdev` devices via `libevdev`, a user-space library that abstracts IO calls through a type-safe interface. `Libevdev` (and EvLag) sits just above the kernel, between the application and the devices themselves. Communication to and from the input devices makes use of the `input_event` structure, defined in `include/linux/input.h` and shown in Listing 1.

The `struct input_event` members are [19]:

- `time` is the timestamp when the event happened.
- `type` is the event type – for example `EV_REL` for relative movement and `EV_KEY` for a key press or key release. More types are defined in `include/linux/input-event-codes.h`.
- `code` is the event code that accompanies the type – for example `REL_X` for relative mouse movement or `KEY_BACKSPACE` for a key press. The complete list of codes is defined in `include/linux/input-event-codes.h`.
- `value` is the value the event carries, either a relative change for `EV_REL`, absolute new value for `EV_ABS` (such as for joysticks), or 0 for release, 1 for keypress and 2 for autorepeat for `EV_KEY`.

EvLag uses `pthreads`, creating two additional threads – a reader and a writer – for each lagged device. Psuedo-code for the reader thread is shown in Listing 2. The reader thread pulls input events out of the device, adds the desired amount of latency, and puts the input events into a FIFO queue shared by the writer. Psuedo-code for the writer thread is shown in Listing 3. The writer thread pulls input events out of the FIFO queue at the appropriate time and writes them to the device.

A third thread – the main thread – is interrupted at a fixed tick rate and notifies all the writer threads each tick. The timing is from the real time clock `/dev/rtc` via a `read()`, which blocks until the next tick (a timer interrupt) is received. This allows for high frequency polling without being CPU bound, e.g., without continuously polling `gettimeofday()`.

The EvLag repository includes a `Makefile`. To compile EvLag, the `libevdev` package is needed (as well as `gcc`, of course).

Listing 2: Reader thread

```
while (1):

  ev = libevdev_next_event() // Pull event from dev. (blocking)
  ev.time = ev.time + delay  // Add delay to event
  lock FIFO queue
  enqueue(ev)                // Add to queue for writer
  unlock FIFO queue

end while
```

Listing 3: Writer thread

```
while (1):

  if gettimeofday() > ev.time:      // Time for next event?
    libevdev_uinput_write_event(ev) // Push event to dev.
    lock FIFO queue
    ev = dequeue()                  // Get next from reader
    unlock FIFO queue
  end if

  pthread_cond_wait()  // Block until timer interrupt from main

end while
```

Listing 4: Main thread

```
open /dev/rtc      // Open timer
ioctl()            // Set interrupt/polling rate for timer

for each device:

  setup FIFO queue          // For reader-writer communication
  create reader thread
  create writer thread

end for

while (1):

  read /dev/rtc             // Read timer at polling rate
  pthread_cond_broadcast()  // Signal writer(s)

end while
```

3.2 Usage

EvLag is run as a command line tool, indicating the amount of latency to add and the device(s) to add it to. EvLag must be run with root permissions (e.g., through `sudo`), but without any other kernel modules – everything runs in user-space through `libevdev` and `uinput`. Usage is:

```
Usage: evlag [OPTIONS...] --lag <NUM>
             --device <FILE> [--device <FILE> ... (max 10)]

MAIN OPTIONS
-b, --buffer=NUM   Size of buffer (MB)
-d, --device=DEV   Device /dev/input/eventX to lag
-f, --file=FILE    Logfile for events (default none)
-h, --Hz=NUM       Polling rate of device
-l, --lag=NUM      Length of delay (ms)
```

For example, the command:

```
sudo ./evlag -l 75 -d /dev/input/event10
```

adds 75 milliseconds of latency to the `event10` device (e.g., a mouse).

3.3 Output

EvLag can produce logfiles detailing all input events upon request. This is done by using the `-f NAME` (or `-file NAME`) flag, with NAME the file prefix and `.log` as the file suffix. This creates a logfile for each device, with the device type (e.g., mouse) used as a prefix in the name.

The logfiles themselves are in comma separated value (CSV) format, with each row:

millisec, event-type, event-code, event-value

The first line of each file is the header, followed by the events, one per line. The first column is the time the event happened (in milliseconds) relative to the start time, followed by the event type, event code and event value.

Since evdev is a serialized protocol, simultaneous events are indicated by a "synchronous marker" (`EV_SYN`, with values (0,0,0)) to indicate that the preceding events all belong together. For example, the output from a mouse may look like (the '#' and following characters are comments shown here only and are not in the EvLag output file):

```
13, 0002, 0000, 0001    # EV_REL / REL_X   1
13, 0002, 0001, -006    # EV_REL / REL_Y  -6
13, 0000, 0000, 0000    # EV_SYN
```

The output indicates that the mouse moved 1 to the right (on the x-axis) and 6 units up (on the y-axis). The timestamps (the number 13 in the first column) are all the same since they happened at the same time. They belong together as indicted by the last line (`EV_SYN`). As another example:

```
23, 0004, 0004, 589825  # EV_MSC / MSC_SCAN
23, 0001, 0110, 0001    # EV_KEY / BTN_LEFT
23, 0000, 0000, 0000    # EV_SYN
34, 0004, 0004, 589825  # EV_MSC / MSC_SCAN
34, 0001, 0110, 0000    # EV_KEY / BTN_LEFT
34, 0000, 0000, 0000    # EV_SYN
```

The output here indicates that the left mouse button was pressed at time 23 and then released and time 34. For a keyboard example, the output:

```
51, 0004, 0004, 458792 # EV_MSC / MSC_SCAN
51, 0001, 0028, 0001   # EV_KEY / KEY_ENTER
51, 0000, 0000, 0000   # EV_SYN
```

indicates the enter key was pressed down at time 51.

3.4 Utilities

Since the EvLag raw output can be difficult to understand, the EvLag repository also includes EvParse – a utility for parsing EvLag log files. Running EvParse on the above output produces:

```
relative axis event at 0.000013, REL_X , X val: 1
relative axis event at 0.000013, REL_Y , Y val: -6
synchronization event at 0.000013, SYN_REPORT
event at 0.000023, code 04, type 04, val 589825
key event at 0.000023, 110 (KEY_INSERT), down
synchronization event at 0.000023, SYN_REPORT
event at 0.000034, code 04, type 04, val 589825
key event at 0.000034, 110 (KEY_INSERT), up
synchronization event at 0.000034, SYN_REPORT
event at 0.000051, code 04, type 04, val 458792
key event at 0.000051, 28 (KEY_ENTER), down
synchronization event at 0.000051, SYN_REPORT
```

Listing 5: Using EvDetect to find a device number

```
#!/bin/bash

keyboard=$(./evdetect.sh -q "GIGABYTE USB-HID Keyboard")
mouse=$(./evdetect.sh -q "Logitech Gaming Mouse G502")

./evlag -l 50 -d $keyboard -d $mouse &
```

EvParse is written in Python 3 and uses the Python 3 `evdev` package.

Figuring out which device is a mouse or keyboard (or other) can be done by trial and error, checking each input device (`/dev/input/eventX`) one by one. Alternatively, the Linux program `evtest` can be helpful. Sample output of `sudo evtest` on a Linux box with a mouse:

```
/dev/input/event0:    Power Button
/dev/input/event1:    Sleep Button
/dev/input/event2:    Video Bus
/dev/input/event3:    HDA Intel PCH Headphone
/dev/input/event4:    PS/2+USB Mouse
```

In the above example, the mouse is `/dev/input/event4`.

A device that is plugged into a port after booting (e.g., an external mouse for a laptop) and/or moved to a different USB port may be assigned a different `/dev/input/event` number by the OS. For scripts that need to automatically determine the device number, EvLag includes `evdetect` – a utility for finding the number of a particular device. For example, the shell script in Listing 5 finds the device numbers for a Logitech mouse and GIGABYTE keyboard and, then, using EvLag applies 50 ms of latency to each device:

When using EvLag in a virtual machine, some care may be needed when lagging input depending upon how the input devices are shared with the host OS. For example, with Oracle Virtual Box, with the "mouse integration" setting on (the default), EvLag does not receive the mouse movement commands and so cannot lag (or log) the mouse movement. EvLag can still lag (and log) mouse clicks and the keyboard. The "mouse integration" setting can be toggled on and off and when off, EvLag works as expected (i.e., the mouse movements can be delayed and logged). Note, in this "off" mode, the right control key can be used to re-capture the mouse by the host OS.

3.5 License

EvLag is distributed as free software; it can be redistributed and/or modified under the terms of the GNU General Public License as published by the Free Software Foundation. It is distributed in the hope that it will be useful, but without any warranty. See the GNU General Public License for more details.

4 EVALUATION

This section describes evaluation of EvLag, including accuracy and overhead (Section 4.1), use in a user study (Section 4.2) and details on a demonstration image (Section 4.3).

4.1 Accuracy and Overhead

We measure the ability for EvLag to achieve the desired added latency and the overhead by measuring latency without EvLag and then with EvLag with several different added latencies.

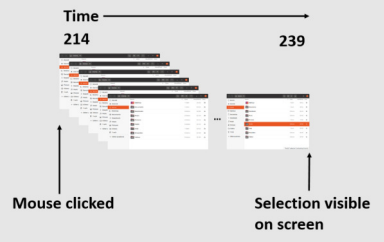


Figure 1: Measuring system latency

We used a Gigabyte Aero 15 laptop with a Logitech G502 mouse. The Aero has an 8-core i7 9750H / 2.6 GHz processor, 16 GB RAM an NVidia GF RTX 2070 graphics card, and a screen resolution of 1920x1080 pixels at 240 Hz. The G502 is a laser mouse with 12k DPI, 300 IPS, and a polling rate of 1 kHz. The laptop is configured with Ubuntu 20.04 LTS, with Linux kernel version 5.4.

We measured latency using the method depicted in Figure 1. A high frame rate camera (a Casio EX-ZR100) filmed at 1000 f/s, capturing the moment a mouse button was clicked inside the file explorer. By manually examining the video frames, the frame number when the mouse was clicked (finger bent, frame number 214 in Figure 1) is subtracted from the frame number when the output was visible based on the user input click (frame number 239 in Figure 1), giving the base system latency (25 milliseconds in Figure 1).

The first condition we tested was the base system without EvLag. Then, we tested EvLag with 0 ms, 10 ms and 50 ms of added latency. The 0 ms condition is to ascertain the latency overhead for EvLag. The measurement method was repeated 10 times on our system for each condition. All EvLag conditions used a polling rate of 8192 Hz.

Figure 2 depicts the results. The x-axis is the added latency and the y-axis is the measured latency from our high-speed camera method. Each point is the mean value for that condition shown with standard deviation error bars. The blue point on the left shows the baseline system latency without EvLag, also indicated by the dashed horizontal line. EvLag with 0 ms has about the same measured latency, suggesting it has minimal observable overhead. The 10 ms and 50 ms conditions closely follow the expected latency, indicated by the dashed angled line.

4.2 User Study

We have successfully used EvLag in a user study assessing the effects of low amounts (under 100 ms) of latency on competitive first-person shooter (FPS) game players [15].

In this study, we configured a high-end gaming laptop (a Gigabyte Aero 15) and gaming mouse (a Logitech G502) to run the FPS game *Counter Counter-Strike: Global Offensive* (CS:GO) (Valve, 2012), scripted so as to take players through game rounds with restricted weapon choices. EvLag was used to control the amount of added latency for the user input for both the keyboard (for avatar movement) and mouse (for aiming and shooting). We replicated this same configuration onto 11 identical laptops, which allowed us to distribute the setup to 43 participants in their homes for testing. EvLag allowed for controlled amounts of local delay without needing manual intervention. EvDetect let us automatically detect the right input devices to lag, even when users rebooted and/or

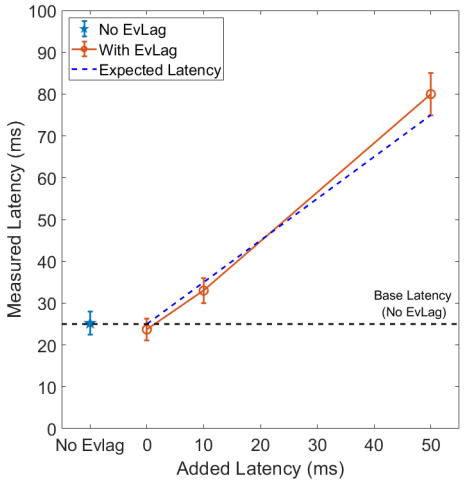


Figure 2: EvLag evaluation

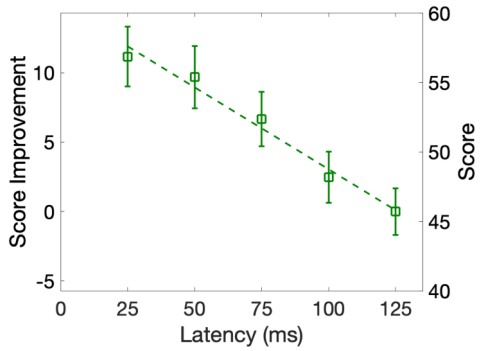


Figure 3: EvLag in a user study – CS:GO score versus latency

plugged in the mouse to different USB ports. After the study, EvParse helped our analysis to ascertain user performance and glean insights from user actions with different amounts of latency.

Figure 3 depicts the results for player score (CS:GO computes score as $score = 2 \times kills + assists$) for 4 minutes of gameplay. The CS:GO log files were mined to determine number of hits, kills and assists by each user for each round, and the EvLag log files were used to determine the number of shots fired based on the number of left mouse button clicks. The x axis is total latency (base plus added by EvLag) in milliseconds (ms). The right y axis is the score and the left y axis is the score increase from the 125 ms latency condition. For example, a score of 15 at 125 ms of latency compared to a score of 20 at 25 ms of latency would be a 5 point improvement on the left y axis. The circles are the means for all users for that latency condition, bounded by 95% confidence intervals. The dashed line shows a linear regression for the mean values.

The linear regression fits the mean scores well with an R^2 of 0.98 and $p = 0.001$. As a take-away, a decrease in latency by 10 ms improves score by 1.2 points per 4 minutes of gameplay. For

reference, often less than a single point separates the scores of top CS:GO players.

4.3 Demonstration

We provide a demonstration of EvLag in a Linux Virtual machine, available for download at:

https://web.cs.wpi.edu/~claypool/papers/evlag/

Upon launching the virtual machine, the demo application shows EvLag's ability to add latency by having the user interact with a paint application with different amounts of added latency. After painting, the demo application shows a logfile produced by EvLag and parsed by EvParse.

5 SUMMARY

Understanding the effects of latency on interactivity in general and computer games specifically is important to build software and systems that better support interactive media applications. Unfortunately, user studies to assess the impact of latency typically must modify the applications being tested. This means re-implementation of latency-inducing techniques each time and, worse, cannot be done for closed-source software, making it difficult to study input latency for commercial games.

This paper presents EvLag, a tool we developed for adding controlled amounts of latency to input devices in Linux, with the option of logging all input events for later analysis. EvLag is implemented in C using the `evdev` device abstraction and the `libevdev` library, providing a lightweight layer of software between input devices and the operating system. This allows it to be used with any Linux application, open source or not, including in user studies with commercial games.

Evaluation of EvLag shows it is effective in adding the desired amount of input latency with low overhead. We have successfully used EvLag in a previous study [15], where EvLag added input delay to a commercial first-person shooter game, and parsed EvLag logfiles to determine player performance and actions with latency.

Future work may include testing EvLag across a larger range of latencies. There are also many more games and interactive applications that might use EvLag in user studies to better understand the effects of latency. EvLag is limited to Linux, so it would be helpful to develop a similar tool for other PC OSes (Windows, Mac OS) and for mobile devices (Android, IOS). Feature extensions to EvLag that may be useful include: support for game controllers in EvParse and the ability to delay multiple devices, each with a different amount of delay. Since EvLag is open source, interested researchers could add such features, and others, as needed.

REFERENCES

[1] R. Amin, F. Jackson, J. Gilbert, J. Martin, and T. Shaw. [n.d.]. Assessing the Impact of Latency and Jitter on the Perceived Quality of Call of Duty Modern Warfare 2. In *Proceedings of HCI – Users and Contexts of Use*. Berlin, Heidelberg.

[2] G. Armitage. 2003. An Experimental Estimation of Latency Sensitivity in Multiplayer Quake 3. In *Proceedings of IEEE ICON*. Sydney, Australia.

[3] Y. Bernier. 2001. Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization. In *the GDC*. San Francisco, CA, USA.

[4] M. Carbone and L. Rizzo. 2010. Dummynet Revisited. *ACM SIGCOMM Computer Communications Review* 40, 2 (April 2010).

[5] G. Casiez, S. Conversy, M. Falce, S. Huot, and N. Roussel. 2015. "Looking through the Eye of the Mouse: A Simple Method for Measuring End-to-end Latency using an Optical Mouse". In *Proceedings of the 28th Annual ACM Symposium on User Interface Software and Technology (UIST)*.

[6] K. Chen, Y. Chang, H. Hsu, D. Chen, C. Huang, and C. Hsu. 2014. On the Quality of Service of Cloud Gaming Systems. *IEEE Trans. on Multimedia* 26, 2 (Feb. 2014).

[7] Mark Claypool, Andy Cockburn, and Carl Gutwin. 2019. Game Input with Delay - Moving Target Selection Parameters. In *Proceedings of ACM Multimedia Systems Conference (MMSys)*. Amherst, MA, USA.

[8] M. Claypool, R. Eg, and K. Raaen. 2017. Modeling User Performance for Moving Target Selection with a Delayed Mouse. In *Proceedings of the 23rd International Conference on MultiMedia Modeling (MMM)*. Reykjavik, Iceland.

[9] M. Claypool and D. Finkel. 2014. The Effects of Latency on Player Performance in Cloud-based Games. In *Proceedings of the 13th ACM Network and System Support for Games (NetGames)*. Nagoya, Japan.

[10] M. Dick, O. Wellnitz, and L. Wolf. 2005. Analysis of Factors Affecting Players' Performance and Perception in Multiplayer Games. In *Proceedings of ACM NetGames*. Hawthorn, NY, USA.

[11] T. Fritsch, H. Ritter, and J. Schiller. 2005. The Effect of Latency and Network Limitations on MMORPGs: a Field Study of Everquest 2. In *Proceedings of ACM NetGames*. Hawthorne, NY, USA.

[12] O. Hossfeld, H. Fiedler, E. Pujol, and D. Guse. 2016. Insensitivity to Network Delay: Minecraft Gaming Experience of Casual Gamers. In *Proceedings of the International Teletraffic Congress (ITC)*. IEEE, Würzburg, Germany, 31–33.

[13] Z. Ivkovic, I. Stavness, C. Gutwin, and s. Sutcliffe. 2015. Quantifying and Mitigating the Negative Effects of Local Latencies on Aiming in 3D Shooter Games. In *Proceedings of the ACM CHI*. Seoul, Korea.

[14] I. Lee, S. Kim, and B. Lee. 2019. Geometrically Compensating Effect of End-to-End Latency in Moving-Target Selection Games. In *Proceedings of the ACM CHI*.

[15] S. Liu, A. Kuwahara, J. Sherman, J. Scovell, and M. Claypool. 2021. Lower is Better? The Effects of Local Latencies on Competitive First Person Shooter Game Players. In *Proceedings of ACM CHI*. Virtual Conference.

[16] M. Long and C. Gutwin. 2018. Characterizing and Modeling the Effects of Local Latency on Game Performance and Experience. In *Proceedings of the ACM CHI Play*. Melbourne, VC, Australia.

[17] M. Long and C. Gutwin. 2019. Effects of Local Latency on Game Pointing Devices and Game Pointing Tasks. In *Proceedings of the ACM CHI*. Glasgow, Scotland.

[18] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan. 2015. Mahimahi: Accurate Record-and-replay for HTTP. In *USENIX Annual Technical Conference (ATC)*. Santa Clara, CA, USA.

[19] V. Pavlik. 1999 - 2001. Linux Input Drivers v1.0. Online: https://www.kernel.org/doc/Documentation/input/input.txt.

[20] A. Pavlovych and C. Gutwin. 2012. Assessing Target Acquisition and Tracking Performance for Complex Moving Targets in the Presence of Latency and Jitter. In *Proceedings of Graphics Interface*. Toronto, ON, Canada.

[21] A. Pavlovych and W. Stuerzlinger. 2011. Target Following Performance in the Presence of Latency, Jitter, and Signal Dropouts. In *Proceedings of Graphics Interface*. St. John's, NL, Canada.

[22] P. Quax, P. Monsieurs, W. Lamotte, D. De Vleeschauwer, and N. Degrande. 2004. Objective and Subjective Evaluation of the Influence of Small Amounts of Delay and Jitter on a Recent First Person Shooter Game. In *Proceedings of ACM NetGames*. Portland, OG, USA.

[23] K. Raaen and I. Kjellmo. 2015. Measuring Latency in Virtual Reality Systems. In *Proceedings of International Conference on Entertainment Computing (ICEC)*.

[24] K. Raaen and A. Petlund. 2015. How Much Delay Is There Really in Current Games? *Proceedings of ACM Multimedia Systems* (March 2015).

[25] B. Ruiz. 2020. The Most Accurate Way to Schedule a Function in a Web Browser. Online: https://tinyurl.com/53xyex6s. (Accessed June 5, 2021).

[26] S. Sabet, S. Schmidt, S. Zadtootaghaj, B. Naderi, C. Griwodz, and S. Moller. 2020. A Latency Compensation Technique Based on Game Characteristics to Mitigate the Influence of Delay on Cloud Gaming Quality Of Experience. In *Proceedings of ACM Multimedia Systems Conference (MMSys)*. Istanbul, Turkey.

[27] A. Schmid and R. Wimmer. 2021. Yet Another Latency Measuring Device. In *Proceedings of the ACM Esports and High Performance HCI Workshop (EHPHCI)*. Virtual Conference.

[28] N. Sheldon, E. Girard, S. Borg, M. Claypool, and E. Agu. 2003. The Effect of Latency on User Performance in Warcraft III. In *Proceedings of ACM Network and System Support for Games Workshop (NetGames)*. Redwood City, CA, USA.

[29] D. Stuckel and C. Gutwin. 2008. The Effects of Local Lag on Tightly-Coupled Interaction in Distributed Groupware. In *Proceedings of theACM Conference on Computer Supported Cooperative Work*. San Diego, CA, USA.

[30] J. Sun and M. Claypool. 2019. Evaluating Streaming and Latency Compensation in a Cloud-based Game. In *In Proceedings of the 15th IARIA Advanced International Conference on Telecommunications (AICT)*. Nice, France.

[31] Chen Tao. [n.d.]. Clumsy. Online: https://jagt.github.io/clumsy/.

[32] The Linux Foundation. [n.d.]. netem. Online: https://wiki.linuxfoundation.org/networking/netem.

[33] R. Wimmer, A. Schmid, and F. Bockes. 2019. On the Latency of USB-Connected Input Devices. In *Proceedings of the ACM CHI*.