ndnMouse
Secure Control Interface for a PC Using a Mobile Device

Master's Capstone Project Report

Wesley Minner
Computer Science M.S. Student
wesleyminner@gmail.com

ABSTRACT

This report outlines the results of my efforts on ndnMouse, my Master's Capstone project, as well as my class project for CS 217B. It is open-sourced on Github [4] under the GNU public license, and freely available on the Google Play App Store [5]. NdnMouse provides secure and efficient control of one or more personal computers via remote mouse movement and rudimentary keyboard commands from the user's phone, running the communication protocol over Named Data Networking (NDN) [19]. Benefiting from NDN's efficient multicasting performance, ndnMouse allows multiple PCs to query real-time data from a low-powered Android phone without a drop in performance. While at this time there is no immediate advantage for the user to connect multiple computers to ndnMouse, the long term goal is to support seamless mouse movement across multiple monitors on two or more computers. In order to judge the design and performance benefits of NDN, ndnMouse also supports communication over UDP. Implementation of both protocols resulted in fairly unique server and clients designs. The pros and cons of each will be explored in this report.

Keywords

NDN, Mouse, Keyboard, Remote Access, Security

1. OVERVIEW

NdnMouse provides remote access to the mouse and keyboard of a PC in order to provide a simple and convenient way to wirelessly interface with your computer from across the room. Wireless slideshow/powerpoint control represented the main use case, removing the need for a proprietary piece of hardware (such as a USB/bluetooth clicker device). Anecdotal evidence suggests that users of these devices encounter a variety of issues such as: drained batteries, poor connection, and general hardware failure. Additionally a presenter must also carry around the clicker hardware, which has no other purpose than slideshow control. NdnMouse removes the need for a specialized clicker and adds remote slideshow control to something most people already carry around in their pockets, a smart phone.

By taking advantage of the local WiFi access point or the phone's WiFi hotspot feature, a user can use their Android smart phone as a virtual touchpad and simple keyboard, providing the functionality needed to step through a powerpoint presentation without any additional hardware. NdnMouse also may be of use during times of hardware failure, providing an emergency backup mouse or keyboard when the user has limited options to interface with their PC.

Both NDN and UDP/IP transport/routing protocols are supported, encrypting packets with AES if a password is provided by the user. Additionally NDN can provide some extra features in the form of interest (query) condensing, allowing the phone server to better scale its performance when multiple PC clients are connected. More on this idea will be described in Section 5. However the protocol choice should ideally be transparent to the user, as all ndnMouse features are supported on both NDN and UDP.

For the rest of this report, Section 1 review the features ndnMouse, with more detail of the command protocol in Section 2. Details on the security features will be given in Section 3, with challenges and trade-offs of ndnMouse listed in Section 4. Performance analysis comparing NDN and UDP implementations is in Section 5. Section 6 explores extensions and future use cases for ndnMouse, and Section 7 looks at some of the related work in this area. Finally Section 8 concludes this report. Additionally screenshots of the application are in Section 11, the Appendix.

1.1 Features

NdnMouse supports full mouse control: relative cursor movement, left click, right click, and a shortcut allowing tap-to-click on the touchpad. Sensitivity and precision settings are also provided, for fine tuning the feel of the movement. Two-finger scrolling works similarly to Apple laptops, with inversion and sensitivity settings as well. Rudimentary keyboard support allows the user to execute common slideshow control commands, such as using the arrow keys or the spacebar to easily change slides. Additionally custom typed messages are supported, allowing the user to type any message using the built-in Android keyboard. Upon sending the message, all receiving client PCs will then virtually type the characters out instantly on whatever program window is selected at that time.

The security for ndnMouse was designed to defend against packet snooping, replay attacks, privacy attacks, and brute force attacks. More details will be given in Section 3. The user may also choose to not use a password, which sends the communication protocol in cleartext and avoids a minor amount of encryption/decryption overhead.

1.2 Supported Platforms

NdnMouse is composed of two applications: the server/producer Java application (running on the Android phone), and the client/consumer Python application (running on the PC). Any relatively modern Android phone is supported

(Android 4.1 and up), and basically any PC that can run NDN's Network Forwarding Daemon (NFD) [7] and Python3 [9]. Since Python runs on the three major operating systems (Linux, OSX, and Windows), the limiting factor is NFD, which only currently runs on Linux and OSX. However Windows users can still use ndnMouse by limiting their protocol choice to UDP only.

Other dependencies include a few Python libraries needed for the PC client, specifically PyAutoGUI [11], PyCrypto [3], and PyNDN [14]. PyAutoGUI provides mouse and keyboard control, as well as some simple dialog boxes for collecting user input. PyCrypto handles all the cryptography operations, and PyNDN provides the API for using NDN and for interfacing with NFD. The Android application uses jNDN [15] to access the NDN API, but this library comes compiled into ndnMouse's APK and requires no outside installation by the user. Both the PC and Android phone must have NFD installed and running to communicate over NDN.

2. PROTOCOL

NdnMouse can run its communication protocol four different ways, depending on the transport protocol used (UDP or NDN) and if a password is provided by the user (security on or off).

2.1 UDP

UDP was chosen as the baseline, IP-based, transport protocol because of its simplicity and performance. NdnMouse does not require the reliable data delivery or the in-order packet processing that TCP guarantees. Reliable data delivery would only delay the real-time movement commands of ndnMouse, and in-order processing is not needed as ndnMouse silently discards late or out-of-order packets. This avoid unexpected mouse movement and jitter. Also by being connectionless, UDP makes it easy to recover from temporary network loss.

Only a minor amount of session upkeep is needed by the ndnMouse client, which sends periodic heartbeat messages, ensuring that the server is still alive and sending unsolicited mouse command data. Lastly, UDP is more suited than TCP to be compared with NDN's relatively featureless transport layer, which will be a focus in Section 5. While ndnMouse's UDP communication protocol does use some similar session establishment strategies as a TCP connection, it sheds all the other features of TCP to be as lightweight as possible.

Without a password (all security off), packets are transmitted in cleartext and are at most 16 bytes. The ndnMouse UDP communication protocol uses connection-oriented communication by having the client send an `OPEN` message to the server, similar to a TCP `SYN` packet. This asks the server to establish a *session* between the client and itself. An `OPEN-ACK` reply message is used to respond to the `OPEN` message and completes the connection setup.

Once a session is established, the server will send unsolicited mouse commands[1] to its clients. The data format is not strict across all possible mouse commands, but generally takes the form of one character dictating the type of command, followed by any additional information for that command. For example, a mouse movement command is of the form `M<x-4B><y-4B>`. I denote the x and y values each have a fixed length of 4 bytes by appending "`4B`." See Table 1 for the supported, non-secure message formats.

Message Type	Message Format
Movement	`M<x-4B><y-4B>`
Click	`C<click_message>`
Scroll	`S<x-4B><y-4B>`
Keyboard	`K<keyboard_message>`
Typestring	`T<type_string-10B-max>`

Table 1: Non-secure message formats

Message Type	Secure Message Format
Movement	`<seq-4B>M<x-4B><y-4B>`
Click	`<seq-4B>C<click_message>`
Scroll	`<seq-4B>S<x-4B><y-4B>`
Keyboard	`<seq-4B>K<keyboard_message>`
Typestring	`<seq-4B>T<type_string-10B-max>`

Table 2: Secure message formats

Heartbeat messages are sent every one second to keep the connection alive when no other mouse command packets are being sent. The client query packet contains the message `HEART`, and the server reply contains the message `BEAT`. The main goal of heartbeats is to detect when the server has lost its session with the client, such as during a server reset or crash. In these cases, the client will miss a certain number of heartbeat replies from the server, triggering a session restart on the client side. The client will then try to open a new session using `OPEN` messages. After receiving a `OPEN-ACK`, the session will resume normal activity.

Lastly a `CLOSE` message is sent by the client upon exiting the application. This lets the server know that it can stop sending unsolicited mouse command messages. Note that the simple, connection-oriented communication I implemented on top of UDP does not provide reliable delivery or in-order processing of packets. Its only purpose is to allow ndnMouse to gracefully recover from server/client connection problems. For example, if the server (phone) silently dies or resets, then without some kind heartbeat the client would never find out, and would not receive any further unsolicited mouse commands. By having a simple heartbeat, the client can detect a service disruption and quickly resolve it by establishing a new session. This provides a better user experience than having to restart both the server and client when something goes wrong.

When security features are turned on, the UDP communication protocol becomes slightly more complex to prevent specific types of malicious attacks. Both the packet format and the payload message require changes. In Table 2, the secure message formats are listed. A four byte sequence number is added to the front of each message, and then the entire message will then be encrypted, as described in Section 2.3 on Mouse Packets. The message length is also fixed to 16 bytes, using PKCS5 padding [6].

[1]I will refer to all commands ndnMouse can send as mouse commands, though this includes supported keyboard commands as well.

Interest Name	Purpose
`/ndnmouse/move`	Movement Data
`/ndnmouse/command`	Command Data
`/ndnmouse/seq`	Sequence Number Sync
`/ndnmouse/salt`	Password Salt Data

Table 3: NDN served interests

2.2 NDN

When designing the NDN communication protocol, I found that the architecture would have to differ significantly from the UDP communication style. The consumer/producer[2] relationship encourages the use of connectionless communication. In this way, the consumer can easily get the data it requests without worrying about where it comes from (location), as NDN data packets are immutable and authenticated via a signature. However note that I do not validate the signature on my NDN produced packets, and my reasons for this are given in Section 4.3. NdnMouse's NDN communication protocol also uses the same message formats given in Table 1 and 2. The only difference is how they are retrieved by clients/consumers.

To allow multiple consumers (PCs) to share the same data packets (from the Android phone producer), I chose to use a connectionless design. That is, each data packet contains no consumer-specific state. All consumers can share and process the same real-time, mouse command data. The connectionless design also fit well with the NDN callback-oriented library, where all communication activity on the server/producer is handled by a single main thread, set up beforehand to serve particular named interests. Contrast this to ndnMouse's UDP communication protocol, which spins off worker threads to handle each client separately. There is no need to do this with NDN as the connectionless design lets the producer be consumer agnostic: all consumers receive the same data from the producer for any given interest. The NDN producer is set up to serve the interest names given in Table 3. The sequence number and password salt interests are only used when ndnMouse has security turned on.

While the NDN producer may run a connectionless communication protocol when security is turned off, an exception had to be made to support sequence numbers when security is turned on. The sequence number primarily helps prevent replay attacks by functioning as a nonce. Each consumer must have some idea of the largest sequence number they have witnessed in returned data from the producer. This means that each consumer must maintain a state and cannot run a completely connectionless communication protocol. More details will be given in the Section 3.

2.3 Mouse Packets

For non-secure communication, mouse packets are exactly the same as the message formats listed in Table 1. In those cases, the payload is the entire packet. When security is on, then additional information must be carried with the payload, and encryption is performed on select segments of the mouse packet. Specifically a 16 byte initialization vector (IV) is prepended to the front of every message, which will be used with AES encryption. More information on this will be given in security section 3. Additionally the total packet length is fixed at 32 bytes, and encryption is only performed on bytes 17-32. See Figure 1 for a visual representation of the secure mouse packet.

[2]For NDN, the consumer can be thought of as the client, and the producer as the server. I may use these terms interchangeably as they are closely related.

3. SECURITY

Creating a secure communication protocol was one of my top priorities for ndnMouse. Few NDN applications thus far have implemented any meaningful security, and the scope of my project allowed me to spend the time to properly design a secure protocol. There are three major components to the security architecture: data encryption, sequence number validation, and password salting. Data encryption also conveniently provides authentication as only authorized users have the proper decryption key to read ndnMouse data. These security components are supported by both the UDP and NDN communication protocols of ndnMouse, though there exist a few minor implementation differences which will be explained below.

3.1 Data Encryption

Data encryption is handled by standard AES with cipher block chaining (CBC). CBC requires a new, random IV for each packet to ensure that the first block of a known piece of data always encrypts to a different piece of ciphertext. Remaining blocks in the data (if any) then use the prior block to act as the IV, which creates a chain of block encryptions. I chose this method because the alternative, electronic codebook, is known to be relatively weak for encrypting data with similar or the same payloads. NdnMouse is expected to send many click messages with the same data content, so I did not want these messages to encrypt to the same ciphertext for privacy reasons.

A 16 byte block size was used for CBC, which meant that packets only contained one block to be encrypted. While this means that CBC provides little benefit to the AES encryption, ndnMouse packet sizes may grow in the future to accommodate more data per packet. Then CBC may be needed to encrypt multiple blocks of data. Additional thoughts on ndnMouse feature extensions that may require larger data packets are given in Section 6.4.

To ensure that the client/consumer can always decrypt a secured packet, the cleartext of the unique IV used for each packet's encryption is prepended to the front of the ciphertext. It is common to transmit IVs in cleartext, as knowledge of the IV alone cannot help decrypt an encrypted piece of data. The IV merely helps the final ciphertext to look unique, regardless of the underlying cleartext data.

3.2 Sequence Number Validation

Even though each secured packet is unique via the IV and AES encryption, a replay attack could be used to maliciously force the client to perform an mouse command. To prevent this, a sequence number was added to the front of each message (underneath encryption), which could then be validated on both the client and server side before executing any mouse or protocol commands. The sequence number policy of ndnMouse enforces that no command should be executed which contains a sequence number lower than the largest sequence number witnessed by the device.

Upon opening a new ndnMouse session, the client sends the initial `OPEN` message with a sequence number of zero, and

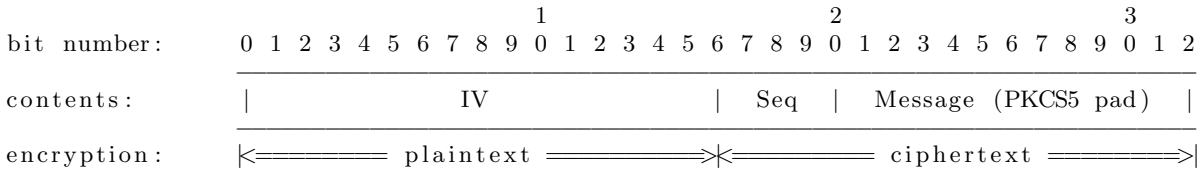

Figure 1: Secure mouse packet description

the server replies with the `OPEN-ACK` message and a sequence number of one. Thereafter each mouse or protocol command uses the current sequence number incremented by one. Since each sequence number is only used once, this field can be thought of as a nonce. Additionally since ndnMouse uses in-order sequence numbers, it is easy to determine if a number was previously used. So if an attacker recorded a set of encrypted packets and attempted to replay them at a later time to the client, the client would find old sequence numbers and simply ignore the commands.

The question then arises whether packets delivered out-of-order would cause a problem for ndnMouse. This is not the case as both the server and client have a catch-up mechanism, which allows them to set their sequence number to any number higher than what they are currently set to. If packets were to arrive with sequence number order 2, 4, and 3, the last packet command would be ignored. The catch-up mechanism would assume that sequence number 3 was lost or delivered late, skipping ahead to a current sequence number of 4. For a latency-sensitive, mouse control application, late packets are useless anyway!

3.3 Password Salting

Users typically resort to using the same password for a given application. With this in mind, if packets were captured by a malicious subject during a given session of ndnMouse, the decryption key[3] would be the same as any future session. Then replay attacks would still be possible *intra*-session[4] (as long as the sequence number was not relatively old). This is different than the prevention of *inter*-session[5] replay attacks that sequence numbers have provided us. To resolve this security hole, random password salts were added to ndnMouse.

For ndnMouse, a password salt is 16 random bytes generated by the client or server, which will be appended to the user password before hashing occurs to generate the key. The UDP communication protocol uses the initial IV on the client's `OPEN` message to use as the password salt for the remainder of the session[6]. Like IV's, a password salt may be passed in cleartext as it does not help decrypt an encrypted message or reveal the user password in any way.

For the NDN communication protocol, all interest names are given in cleartext, so no IV can be used from the client/-consumer as the password salt. To resolve this, the server/producer will create a random password salt on session startup. The consumer can simply retrieve this data by sending an interest for `/ndnmouse/salt` as seen in Table 3. All other data returned for interests will be encrypted using the key generated with the salted user password.

Traditional uses of a password salt are for storing user passwords securely in a database. Passwords are salted, then cryptographically hashed, with the database only storing the resulting hash and the password salt in cleartext. In this way, a compromised database will not reveal the actual user passwords to the attackers. To retrieve user passwords from the corresponding hashes/salts, an attacker would need to put forth a significant brute force effort of comparing a password rainbow table using the cleartext salts provided for every user, resulting in a table several magnitudes larger than a rainbow table needed for unsalted passwords.

NdnMouse does not store any form of the user password in persistent memory. However it should not be susceptible to intra-session replay attacks. Therefore by simply using a password salt, each session will have a unique key for encryption/decryption even if the same user password is used. Any replayed packets from previous sessions will not decrypt correctly on the client side, and will be thrown away silently.

3.4 Attack Types and Defenses

As stated in the features section, security was designed to defend against packet snooping, replay attacks, privacy attacks, and brute force attacks. Each of the security features mentioned above target one or more of these malicious behaviors. Encryption prevents malicious subjects from snooping data from packets. Replay attacks are prevented both inter-session and intra-session via sequence number validation and the use of password salts respectively.

The effectiveness of privacy attacks is decreased by using new, random IVs on each individual packet. This prevents two packets with the same data being encrypted to the same ciphertext, reducing the amount of inference an attacker can gain by analyzing the data flow. Finally the efficiency of brute force attacks is limited by using a new password salt for each session. NdnMouse sessions are expected to be relatively short-lived, as slideshow presentations represent the primary use case. As a result, a relatively small amount of packets using the same key will be exposed for a brute force attacker to collect.

4. CHALLENGES AND TRADE-OFFS

NdnMouse met with several challenges and trade-offs during implementation. Development of the four different protocol styles (UDP or NDN, security on or off) flowed in a serial manner. The UDP server implementation was built first, then ported to NDN. After that, the server classes were extended to support security, first in UDP and then in NDN. It was more challenging that I initially anticipated

[3]For this report, the term *key* refers to a cryptographically hashed user password

[4]between two different sessions

[5]within the same session

[6]The UDP `OPEN` message is encrypted with the unsalted password hash, but there after all other messages will use the salted password hash.

to port the UDP server to NDN, as the way of thinking for each style of communication differed greatly. UDP took on a traditional worker thread design, while NDN lent itself to a more simple, single-threaded, callback approach. Other challenges included the lack of unsolicited data support in NDN, the limitations of addressing a device through NDN, and my decision to use a shared user secret rather than NDN's built-in signature validation.

4.1 Unsolicited Data

UDP can easily send unsolicited data through a socket, but NDN cannot send unsolicited data through a face. By design, it must receive an interest packet first. For my application, unsolicited data is useful to send unpredictable mouse commands, like mouse clicks, avoiding the need for continuous polling from the client side. Mouse movement, on the other hand, must be continuously polled at a constant rate, which translates nicely to the NDN interest/data model.

To handle mouse clicks and other commands using NDN, I created a separate interest that would ask the producer for mouse command data at a constant rate. A majority of the packets time out due to no available data, but when the user does execute a mouse click or a special keyboard keypress, the consumer side will still receive the data in a timely manner. Though this method is not as efficient as UDP, the latency is still low enough to be nearly imperceptible by the user.

4.2 Addressing Devices Using NDN

In NDN, devices no longer have IP addresses. Instead each producer registers a set of prefixes for which they produce data. Consumers do not retrieve data by a producer's address, but instead they simply ask for desired data by name. Currently NDN's forwarding daemon, NFD, does not provide routing/propagation of registered name prefixes. This means that when a producer registers the prefixes it is willing to serve to the local NFD, the registration stops there. NFDs that are one or more hops away do not get this information, and do not know to forward relevant interests in that direction.

Since ndnMouse was designed to run over a local WiFi access point, there are two NFD hops that the consumer interests must pass through to reach the producer: NFD on the consumer (PC) and NFD on the producer (phone). In order to effectively route interests from the consumer to the producer, ndnMouse must preemptively set up an interest route, forwarding interests with the `/ndnmouse` prefix to the IP address[7] of the Android phone's NFD. The routing protocols are not yet mature enough to do this automatically, so a shell command is used to create this route: `nfdc register /ndnmouse <producer-ip-addr>`. For this reason, the user is required to enter the phone's IP address on consumer application startup (on the PC). See Figure 2 for a visual representation of route setup. At this time, the only way to completely eliminate the need for IP addresses in NDN applications is to use a more direct method of connecting two devices, such as WiFi Direct. However WiFi Direct is not yet supported on NFD for Linux PCs. For this reason, a trade-off was made to use IP address to perform connection setup, even though all communication is being run over NDN.

[7]NDN currently uses IP tunneling to do routing of interests and data.

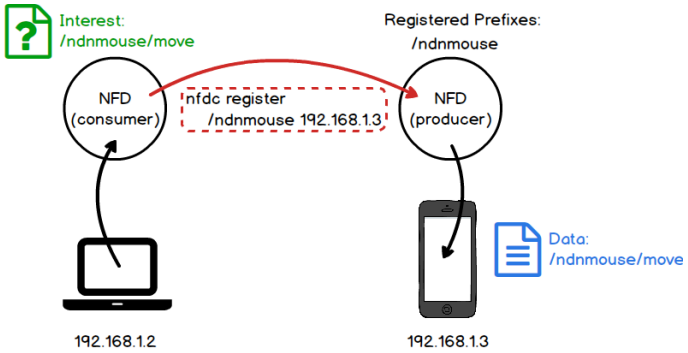

Figure 2: Setting up an NFD route

4.3 Signature Validation vs Shared Password

Enforced packet signatures is one of the most important features of NDN, as it allows a consumer to acquire data from any location, independent of where it was originally produced. A consumer can always verify if the data originates from the producer they expect by validating the data's signature. I originally intended to use this method to validate the consumers (PCs) that try to get data from the producer (phone). However due to time constraints and ease-of-use considerations, I eventually decided to forgo this idea for a simpler strategy that relied on a shared user password.

Signature validation presented a complex problem to solve in order to simply trade a shared symmetric key. Since ndnMouse is meant to be set up by the same person on at least two local devices, typing in a simple user password allowed both devices to have the same symmetric key without any additional work. If ndnMouse had to use signature validation to trade a symmetric key, there would then be additional user-workflow problems with deciding who could receive that symmetric key from the producer and who could not. Even with proper NDN certificates installed on each device, the user would likely need to whitelist devices that were allowed to act as consumers for ndnMouse.

By having the user decide the shared secret offline and pass it as a parameter for each controlled device, ndnMouse can provide both consumer authentication and a shared symmetric key for encryption/decryption of data. Also I assume that most users are familiar with how a shared password would work for this type of application. If ndnMouse required a whitelist of authenticated NDN devices, it might burden the user with the need to understand more of NDN's inner workings.

While these arguments may not fully justify stepping away from the ideal NDN workflow (which uses signature validation), I believe that a shared password works best for ndnMouse's use cases and fulfills its ease-of-use goal. However future extensions of ndnMouse may be able to add in a module to collect a shared secret via signature authentication, making use of all the features NDN has to offer.

5. PERFORMANCE ANALYSIS

During my development of ndnMouse, I noticed a few key performance differences between the NDN and UDP communication protocols. This could be due to a variety of rea-

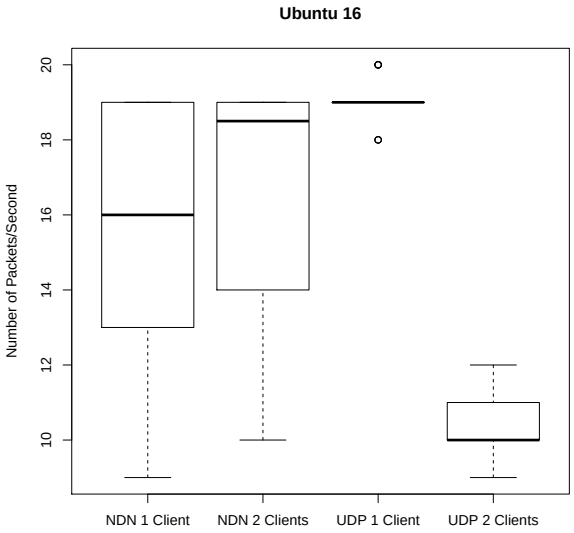

Figure 3: Ubuntu mouse movement performance benchmark

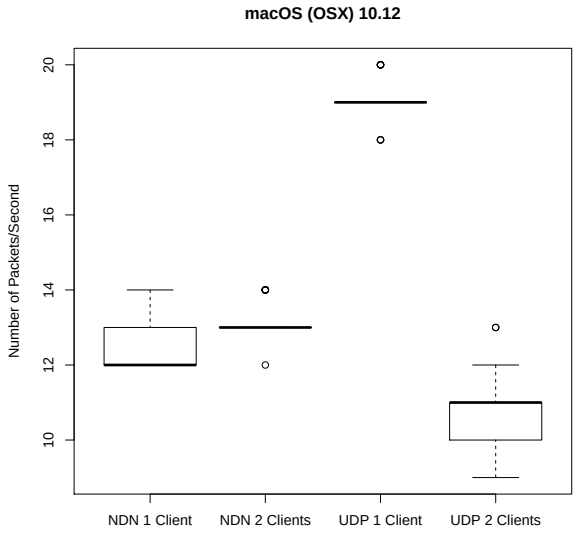

Figure 4: OSX mouse movement performance benchmark

sons, such as the ndnMouse server/client implementations, the core performance of NFD, the kernel integration of UDP, etc... This section will explore how ndnMouse performs in a realistic scenario, and try to explain any performance differences between the NDN and UDP communication protocol.

Two PCs with two different operating systems (Ubuntu 16, and macOS[8] 10.12) were used for testing, along with a Nexus 5X Android phone running Android 7.0. The Ubuntu 16 PC is a custom-built desktop, with an Ivy Bridge i5 Intel CPU. The Macbook Pro laptop running macOS was built at the end of 2013. All benchmarks were performed on the same wireless access point with security enabled, in order to be as close as possible to the primary use case: slideshow control. Wired and non-secure communication was also tested briefly, and no significant improvement was noticed in the movement update frequency. So only wireless, secure benchmarks are analyzed in this section.

The benchmarking tests are segregated by PC due to hardware differences, and show the performance of both NDN and UDP communication protocols. The tests were designed to quantify the smoothness of mouse movement on each communication protocol, and understand how the performance would scale with the number of clients connected to the ndnMouse server. The number of mouse movement update packets per second gives us a decent indication of how smoothly the mouse was moving on the client at any particular time. Each test lasted for 30 seconds, sending as many mouse movement updates as possible during that time. Note that the ideal update frequency is programmed to be 20 Hz. Anything less than 20 packets per second means that overhead/load is significant enough to drag down the performance.

Each resulting dataset is summarized with a separate box plot on Figures 3 and 4. The multi-client tests were performed at the same time. That is, both the Ubuntu and OSX PCs were attached to the server/producer during the multi-client benchmark, and the resulting dataset was used for both PC's 2-client box plot. The single client datasets were all collected separately.

[8]Operating system names MacOS and OSX are used interchangeably in this report.

5.1 NDN vs UDP

Figures 3 and 4 compare the performance of NDN to UDP for two different scenarios: single client and multi-client (2 clients). For single clients tests, NDN seems to be inconsistent in its frequency of movement updates relative to UDP, which seems to have very few points outside outside the median line markers. Additionally NDN has more outliers than UDP and a much wider range of update frequencies were observed in its 30 second test window. This correlates well to the perceived user experience of the NDN communication protocol on ndnMouse. In practice, NDN mouse movements are often times jagged and slow to update. UDP, on the other hand, performs much more consistently, updating frequently enough to produce smooth and easy to control mouse movements.

A glaring concern comes from the large performance gap between the Ubuntu benchmark and the OSX benchmark. This could be due to the hardware advantage of the Ubuntu desktop. However the proposed hardware advantage looks less likely when UDP is observed to be equivalent in performance for both Ubuntu and OSX. Another potential explanation could suspect degraded performance of NFD on the OSX kernel vs a native Linux environment.

The performance of NDN redeems itself when we examine how the implementations scale with additional client/consumer load. Oddly enough, the NDN performance becomes more consistent, with smaller box plot quadrants on both Ubuntu and OSX. Other than that, there is no noticeable performance degradation relative to the single client NDN benchmark. This was verified by viewing the data and by simply examining the smoothness of the mouse movements

on-screen during the test.

UDP does not fair so well in the multi-client benchmark, dropping its single client performance from a consistent 19 packets per second, to about half that rate for multiple clients. The same performance drop was found on both Ubuntu and OSX. The visual mouse smoothness also correlated this performance loss, with very jerky and inconsistent mouse movements on both attached clients. The following sections will examine how the NDN and UDP ndnMouse implementations can help explain these benchmark performance characteristics.

5.2 Event-Driven Architecture

The NDN library asks the application writers to implement their producers as event-driven. Callbacks are setup ahead of time for specific interest names (events), then the producer simply waits for interests to come in to handle them serially. In this way, a single thread is needed for the producer to handle all incoming interests, similar to popular event-driven frameworks like Node.js [8] and Twisted [16].

This type of architecture is known to scale well, as adding more clients does not increase the overhead of thread management. Additionally the callback setup allows for a much simpler implementation in ndnMouse's case. The server is nearly stateless, with the exception of the sequence number nonces. When comparing the implementation classes for NDN and UDP, it is clear that the NDN producer code is shorter and easier to read. Using descriptive interest names also makes it clear to the programmer what is intended to be done with each handler.

Interest collapsing by NFD may be another contributing factor for the enhanced scaling performance of ndnMouse's NDN communication. As multiple consumers send the same interest to the ndnMouse producer, the Android phone's NFD will condense queries with the same interest name, only forwarding a single interest to the ndnMouse application. The pending interest table (PIT) of NFD will keep track of all the consumers who asked for the data. After the interest is handled by ndnMouse and data is returned to NFD, the PIT is used to distribute the data appropriately to all consumers who originally sent interests. This one level of abstraction between the consumers and the producer greatly reduces application effort required to multicast data back to interested consumers.

5.3 Multi-Threaded Architecture

UDP's implementation took on a traditional multi-threaded approach, spinning off a new worker thread for each additional client. While this approach worked well for a single client, even just adding a second client increased the thread management overhead significantly, dropping the performance by half. The UDP clients must also keep track of more state than their NDN counterparts (like using heartbeats), in order to reap the performance benefits of unsolicited data.

There may be performance optimizations that could help the UDP server scale better on cheap Android phones. It might be possible to push more responsibility on a single worker thread, rather than spinning off new threads each time another client arrives. However this would also increase the complexity of the implementation, by forcing one thread to juggle several different states for different clients.

6. EXTENSIONS

NdnMouse was primarily developed in seven weeks, starting from the beginning of April 2017. There are still many unimplemented features and optimizations that could benefit the application, but were outside the scope of the project's time frame. As such, extensions to ndnMouse will be listed here to show how this project could continue to evolve in the future.

6.1 NDN Performance Optimization

As shown in the performance section, ndnMouse's NDN communication protocol does not perform as well as its UDP sibling, for a single connected client. This could be due to a variety of reasons, such as NFD running as a software layer instead of being integrated into the kernel. However this cannot easily be changed, so instead we look to ndnMouse's server implementation for NDN optimization opportunities.

Mouse movement update consistency seem to be the main performance issue for the NDN communication protocol. Currently the NDN server architecture has only one outstanding (pending) interest at all times for collecting movement data. This means that an interest must be sent out across the network, be processed and fulfilled by the producer, then the resulting data must traverse back through the network, following the PIT, to the consumer. Only upon receiving the data back, can the NDN consumer send another movement interest.

In order to smooth out mouse movement, data must be returned faster and at a more consistent speed. There may be a performance gain if ndnMouse consumers sends two or more outstanding movement interests at any one time (instead of just one). Having parallel pending interests could loosen the movement update bottleneck, as long as NFD is not causing the issue. However more experimentation would be required to gain any additional insight on this idea.

Another idea for better NDN performance could be condensing the number of interests sent by the consumer. If the producer packed all its data into a single larger data packet (containing movement, clicks, commands, key presses, etc...), then this would reduce the number of interests the producer would need to process. This uses a well known operating system concept called batching to reduce overhead. Decreasing overhead on the producer's side may be a practical idea, but the complexity of data packing and unpacking would certainly increase as a trade-off.

6.2 Many Mice, One PC

A common feature request[9] for ndnMouse asks for support of many mice controlling one computer. This would allow local collaboration on a single PC, letting each user have a say in what gets shown on-screen. For example, the speaker may click quickly through a slide with an important point, leaving some viewers confused. Any person using the control application, may then click backwards to the important slide and ask their question aloud. This avoids the awkward feedback loop of the viewer asking the speaker to click back once more, and once more, and once more, etc... to reach the slide in question.

In order to support this feature, ndnMouse would require an architectural rework. Currently the phone acts as the server/producer, and the multiple PCs connecting to it act

[9]Idea credit goes to Dr. Lixia Zhang.

as the clients/consumers. Each PC must reach out to connect to the desired phone (via IP address). The many mice to one pc relationship inverts this, so that the PC is now the server/producer, serving the many mice connecting to it. Mice would have to explicitly reach in to connect to single PC, entering its IP address to properly connect. There may be a feasible solution that allows both relationships to work (one mouse to many PCs, or many mice to one PC), but more architectural design work is required to understand the additional requirements this would impose on ndnMouse.

6.3 Additional Wireless Interface Support

NdnMouse currently only supports communication over a local WiFi access point or via the phone's WiFi hotspot feature. However users may wish to use other wireless interfaces, like Bluetooth or WiFi Direct. At the time of ndnMouse's design, NFD only supported standard Ethernet and WiFi interfaces. However during this quarter, two students in CS 217B have been working on NFD support for Bluetooth and WiFi Direct (on PCs). When support for these interfaces is complete, it would be greatly beneficial to ndnMouse users to receive additional wireless interface options to simplify connection setup in times when local WiFi access points are not available.

6.4 Cross-Computer Mouse Support

In my view, the long term goal for ndnMouse was to create an application that could eventually control multiple PCs seamlessly. For example, moving your mouse from the screen of a laptop, to another monitor of a completely separate desktop, as if they were the same computer. Sharing a mouse and keyboard between computers over the network is not a new concept. The desktop application Synergy [12] has been around since 2001, and runs on all major desktop operating systems. It basically allows multiple computers with different operating systems to share the same mouse and keyboard, seamlessly transitioning the mouse across screens as if all the monitors were connected to a single computer.

NdnMouse could fill a similar niche as Synergy, but also bring additional features and security with it via NDN. Communication between the server and clients would be more efficient with NDN's built-in multicasting support. NDN packet signatures could be optionally validated for extra confidence when verifying identities. Lastly, the Android-based server front end of ndnMouse could also be ported to a desktop application to take advantage of real mouse and keyboard hardware (similar to Synergy).

These are just some of the extension ideas that show the promise of ndnMouse, and how it could continue to evolve and provide the benefits of NDN to more users.

7. RELATED WORK

There are no known NDN applications that serve as real-time control applications. Some tools are available that support that real-time synchronization of data, such as ChronoSync [20]. However applications that use this tool, like NDN Whiteboard [2], are more latency tolerant than ndnMouse. NDN Whiteboard propagates changes on a shared drawing canvas to other users viewing the same whiteboard. Lower latency provides a better collaboration experience, but eventual consistency is good enough for this application. The same cannot be said for ndnMouse, which requires a low enough latency to allow remote control of a mouse pointer, displayed on different screen than the one taking user input.

As I mentioned in the Extensions section, Synergy [12] provides cross-computer mouse support by having the server PC sending control messages to all the client PCs. Synergy does not support remote control via a mobile device however. NdnMouse is a step in that direction, ideally using NDN to provide a more efficient method of multicasting to multiple clients.

There is plenty of related work in the field of remote mouse and keyboard control (though not communicating over NDN). Searching Github, I found many open-source mouse control applications, but none of them I found implemented any security features. Two relatively popular Github repositories, Android Desktop Remote Control [13] and Mobile Mouse [18], had basic mouse and keyboard control features, but no security features were included. Neither of them appeared to be released on the Google Play store.

Android Desktop Remote Control uses UDP datagrams, while Mobile Mouse uses a TCP socket. Both applications use a different server/client setup than ndnMouse. The PC acts as the server, while the Android phone acts as the client (ndnMouse has the opposite configuration). The advantage of the server-PC/client-phone configuration is that it may allow multiple phones to control a single PC. However, after examining the code of these two Githubs, it does not appear that the code architecture would allow this. Only one thread is created on server instantiation, which can only serve one client phone at a time.

As for closed-source, mouse control applications, there are numerous free Android apps on the Google Play store. I chose two of the most popular ones (reported to have 5-10 million installs) and tested them out. Remote Mouse [10] is ad-supported with additional in-app purchases to unlock extra features. The base model is still well-featured, with many additional shortcuts and PC control options. The architecture defines the PC as the server, with connecting Android phones as the clients. Multiple phones may connect to control a single PC server. By using Wireshark [1] to snoop packets between my phone and PC, I was able to analyze some of their communication protocol. Remote Mouse uses both UDP and TCP packets, where TCP is primarily used for setup and teardown of sessions. The UDP control packets were resistant to temporary network loss, as the session resumed control after WiFi was restored on the phone client.

Security is laughable on Remote Mouse, in that it provides almost no protection from privacy attacks or replay attacks. The phone client always responds to the password authorization challenge with the same hash (for a given password). This could easily be replayed to gain false authorization. No encryption is used for mouse commands, and keyboard typing uses some form of a Ceasar Cipher, shifting the character ASCII-encoding by a constant amount. It would be trivial to perform a man-in-the-middle attack and decode all typed characters sent from the phone.

The other popular Android app, WiFi Mouse [17], has a similarly free base application, with ads and paid additional features. Multiple phone clients can control a single PC server. TCP packets are used for all communication, but sessions cannot survive temporary network loss. No errors were reported by the client or server, but mouse and keyboard control no longer worked after toggling WiFi. Security is abysmal on this app, as a non-configurable 4 digit

Figure 5: Snooping "secure" data from WiFi Mouse

password is set automatically by the server. Authorization is needed to establish the initial TCP connection, but the challenge response hash sent by the phone never changes (for a given password). All TCP data is cleartext, including keyboard messages, making it extremely easy to snoop sensitive information by following the TCP stream in Wireshark. Lastly, the server application reports currently running applications on your PC to the phone in cleartext. See Figure 5 for an example. I can only guess this is some violation of privacy, as their privacy policy does not state they collect this type of information.

After completing this related work research, I would recommend staying away from closed-source, mouse control Android apps. It is highly likely that no real security is implemented, and they may even be violating your privacy. Contrast that to ndnMouse, which implements powerful security and presents its code freely and openly to the community.

8. CONCLUSION

This report has presented a new application for NDN, enabling users to securely control their PCs from across the room. NdnMouse provides all the features one would expect from a laptop trackpad, and adds supplementary keyboard support for slideshow presentations. Security was a focus of this application during development, and ndnMouse runs its own custom, secure communication protocol to thwart a majority of common attacks. By pressing forward with new NDN applications, it becomes easier to see how NDN can improve the overall security and efficiently of the Internet.

NdnMouse still has room to grow and this reports shows there are plenty of potential extensions which would help ndnMouse serve more users. Additionally ndnMouse helps uncover areas where NDN could use more optimization, demonstrating the performance differences between UDP and NDN server/client implementations. Overall the performance of single client NDN is still respectable compared to that of UDP, considering the additional layers of software and translation NFD adds to the stack. However the scaling performance of NDN is the real star of the show, and demonstrates how next-generation multicast applications can benefit from having a new "narrow waist of the Internet."

9. ACKNOWLEDGMENTS

Thank you to Alex Afanasyev for providing valuable feedback during the writing of this paper. Thank you to Lixia Zhang for advising me many times in office hours throughout this project.

10. REFERENCES

[1] G. Combs et al. Wireshark. https://www.wireshark.org/. Accessed: 2017-5-26.

[2] S. Gouthaman, P. Huang, A. Afanasyev, and P. Bankole. NDN Shared Whiteboard Project. https://github.com/named-data-mobile/apps-NDN-Whiteboard. Accessed: 2017-3-5.

[3] D. Litzenberger. PyCrypto - The Python Cryptography Toolkit. https://www.dlitz.net/software/pycrypto/. Accessed: 2017-5-19.

[4] W. Minner. ndnMouse - Github. https://github.com/wminner/ndnMouse. Accessed: 2017-5-23.

[5] W. Minner. ndnMouse - Google Play Store. https://play.google.com/store/apps/details?id=edu.ucla.cs.ndnmouse. Accessed: 2017-5-23.

[6] K. Moriarty, B. Kaliski, and A. Rusch. PKCS #5: Password-Based Cryptography Specification Version 2.1. RFC 8018 (Informational), Jan. 2017.

[7] Named Data Networking. NFD - Named Data Networking Forwarding Daemon. https://named-data.net/doc/NFD/current/. Accessed: 2017-5-19.

[8] Node.js Foundation. Node.js. https://nodejs.org/en/about/. Accessed: 2017-5-23.

[9] Python Software Foundation. Python 3.6.1 documentation. https://docs.python.org/3/. Accessed: 2017-5-19.

[10] Remote Mouse. Remote mouse. https://play.google.com/store/apps/details?id=com.hungrybolo.remotemouseandroid. Accessed: 2017-5-26.

[11] A. Sweigart. PyAutoGUI. https://pyautogui.readthedocs.io/en/latest/. Accessed: 2017-3-5.

[12] Symless. Synergy - Mouse and keyboard sharing software. https://symless.com/synergy. Accessed: 2017-5-22.

[13] J. Taylor. Android desktop remote control. https://github.com/justin-taylor/Android-Desktop-Remote-Control. Accessed: 2017-5-26.

[14] J. Thompson, A. Bannis, spencerucla, and P. Batista. PyNDN: A Named Data Networking client library with TLV wire format support in native Python. https://github.com/named-data/PyNDN2. Accessed: 2017-3-5.

[15] J. Thompson and A. Brown. jNDN. https://github.com/named-data/jndn. Accessed: 2017-4-30.

[16] Twisted Matrix Labs. Twisted. https://twistedmatrix.com/trac/. Accessed:

2017-5-23.
[17] WiFi Mouse. Wifi mouse. https://play.google.com/store/apps/details?id=com.necta.wifimousefree. Accessed: 2017-5-26.
[18] L. Zhang. Mobile mouse. https://github.com/tuesda/mobilemouse. Accessed: 2017-5-26.
[19] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, k. claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang. Named data networking. *SIGCOMM Comput. Commun. Rev.*, 44(3):66–73, July 2014.
[20] Z. Zhu and A. Afanasyev. Let's chronosync: Decentralized dataset state synchronization in named data networking. In *Network Protocols (ICNP), 2013 21st IEEE International Conference on*, pages 1–10. IEEE, 2013.

11. APPENDIX

Screenshots of the Android and PC applications follow. The Android app can be downloaded directly from the Google Play App Store, and the PC client application can be downloaded from the Github repository. A demo of ndnMouse can be viewed here: https://youtu.be/ZNNqTG2ha6s

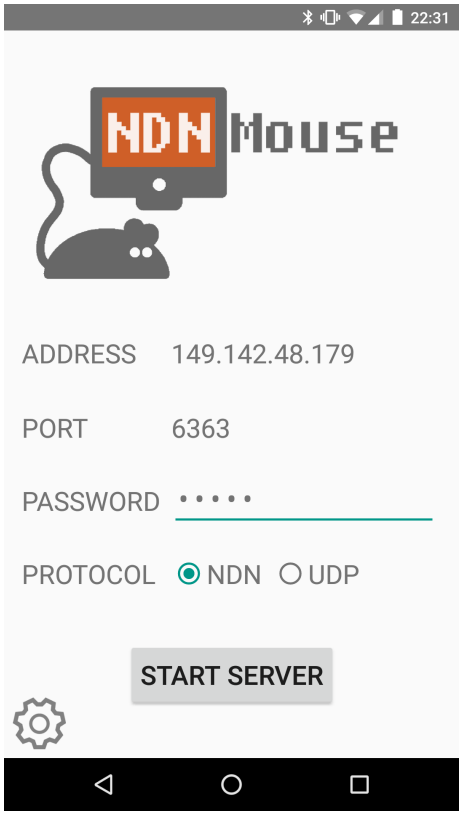

Figure 6: Start screen

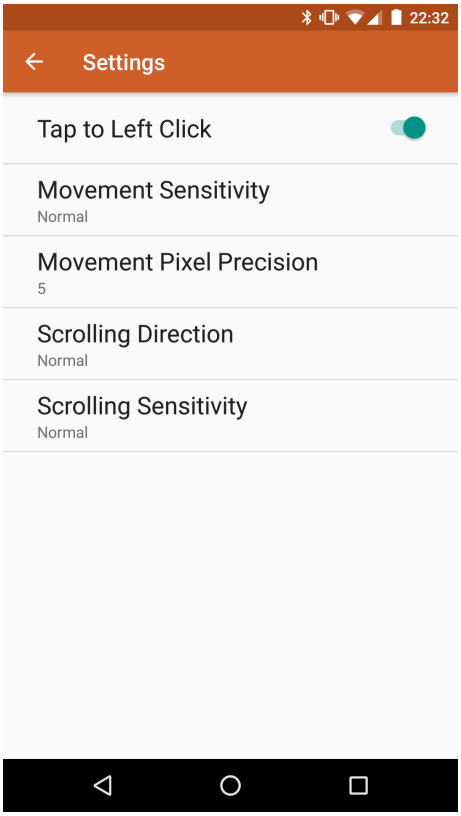

Figure 7: Settings screen

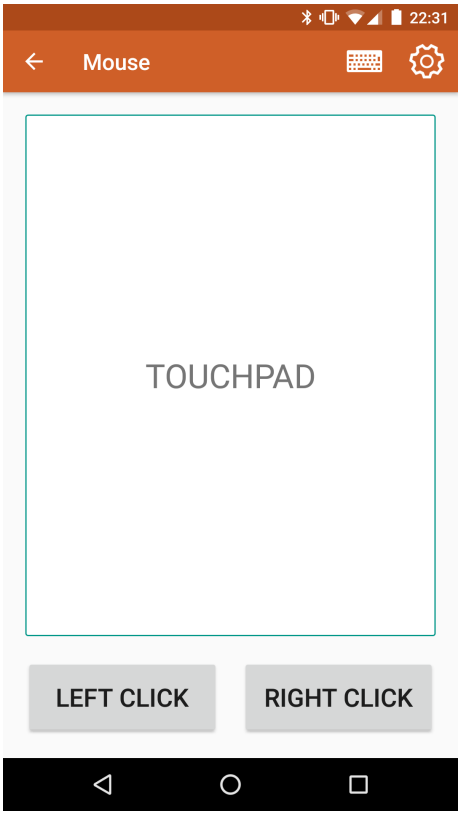

Figure 8: Touchpad screen

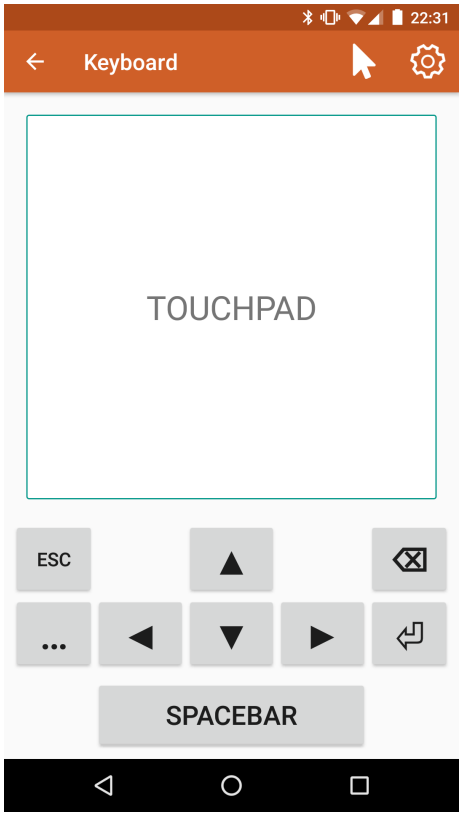

Figure 9: Keyboard screen

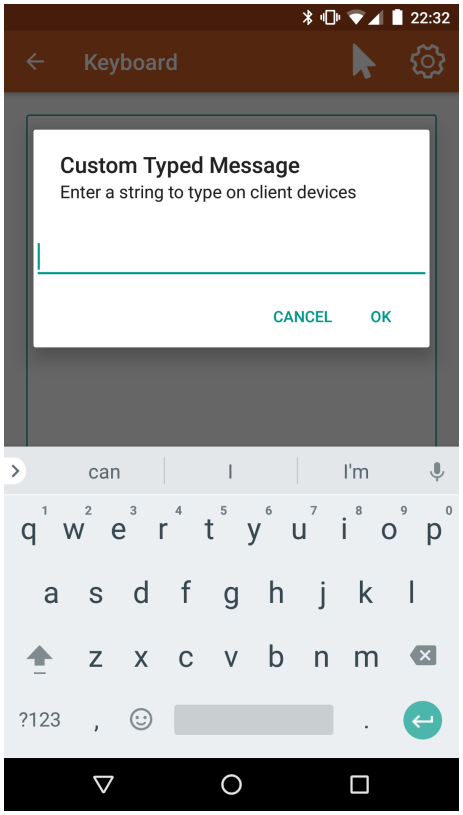

Figure 10: Custom typed message entry

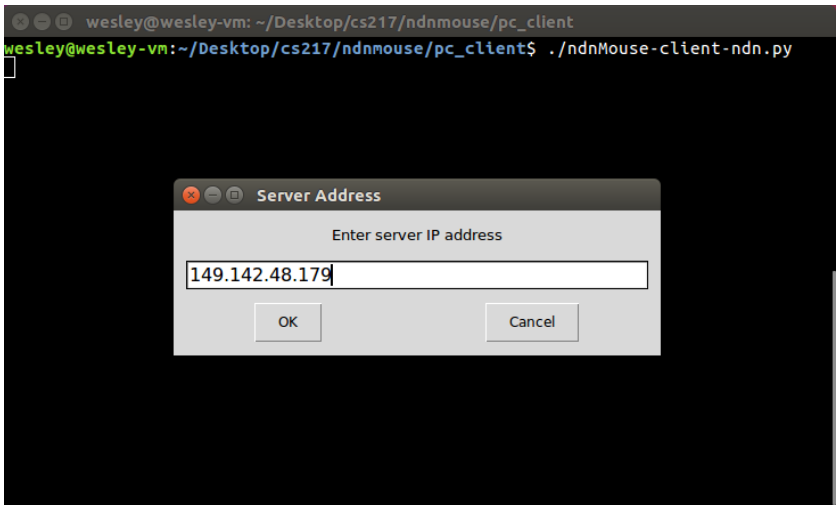

Figure 11: PC program IP address entry

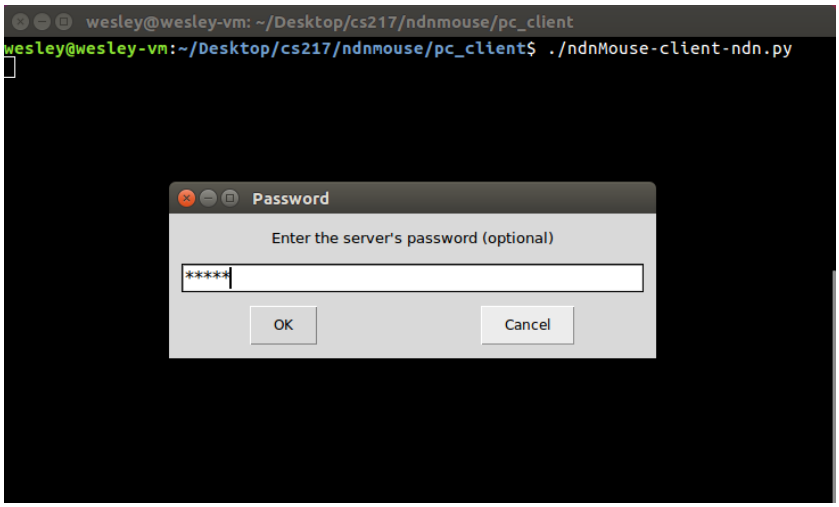

Figure 12: PC program password entry

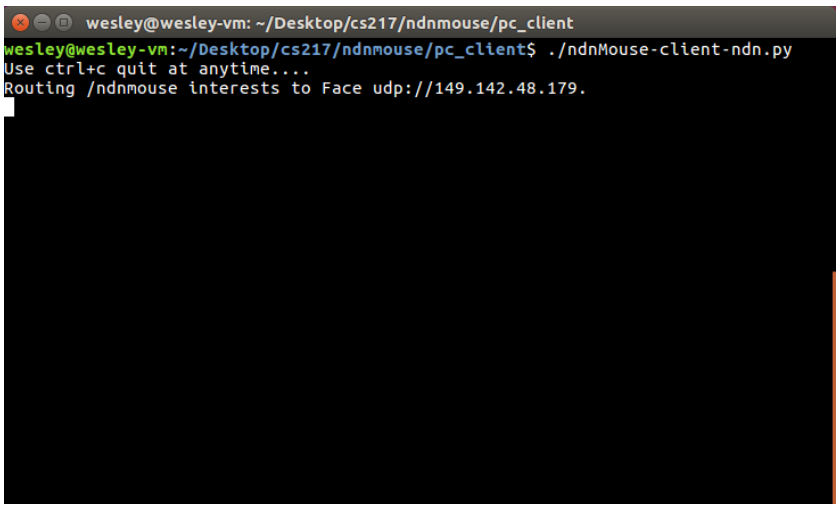

Figure 13: PC program running