===================================

CS 613 - Telnet-NDN Progress Report

===================================

Cody Henderson

Pawan Subedi

Matthew York

Introduction

============

NDN stands for Named Data Networking, a project to redefine fundamental network architecture to meet the networking goals of the 21st century. Named data replaces traditional IP-based routing with a content-centric network model where each piece of information resides at a unique name. A content-centric model of information exchange adds greater opportunities for in-network storage (caching, etc.) and built in trust mechanisms through content signing and encryption.

NDN is a pull based network meaning that the communication between any pair of devices is initiated by a request for data from one of the sides. Further, NDN doesn’t have server-client architecture : the request for the data is quenched by any device in the network that has it. This networking paradigm is useful in some of the application scenarios, but for others, direct communication between two fixed devices might be needed. One example could be remote terminal access. Many devices, particularly in the IOT ecosystem, do not have physical user-interface I/O (i.e. keyboard, mouse, external buttons). Configuration changes with these devices requires logging into them remotely, which presupposes network access to other computers. With current NDN applications, such remote terminal access is not possible, particularly with familiar tools. This project attempts to reverse-engineer a robust client administration tool, Telnet, and its server component, TelnetD, to utilize NDN protocols instead of standard HTTP.

Motivation and Use Case

=======================

Inetutils is the foundational library for countless application around the world. One of the applications bundled in the inetutils library is Telent, a program for remote command of a terminal. It is similar in functionality to SSH, but does not secure the channel. Telnet makes a perfect candidate for adaptation to Named Data Networking protocols due to its simplicity and lack of orthogonal requirements (security, etc).

We propose a minimally-invasive replacement of socket method calls in inetutils with no disturbance to dependent applications. Furthermore, these socket replacement calls will be implemented in telnet as a model for named data networking in other bundled applications such as ftp. In addition, it will provide a useful metric for comparing named data viability versus traditional HTTP networking given that all application operations and constraints are the same, save which type of network is being utilized.

One simple example where remote terminal access is required could be the configuration of RSU and OBU devices in vehicular communication. Most of the RSUs and OBUs run Linux system and are configured for sending and receiving different types of vehicular messages by running the respective applications in them through remote terminal access. In the IP-world, Telnet and SSH are two of the majorly used applications for remote terminal access. Telnet is simpler but lacks security as it sends out the characters in plain text. Thus, in the present times, SSH is largely used in stead of telnet. Security in SSH is achieved by securing the communication channel established between the two communicating parties. In NDN world, each data being sent out is secured within itself thus omits the requirement for securing the channel. Thus, NDN can achieve the security requirement even if it is using Telnet-like communication system. This work, hence, extends the Telnet application into NDN paradigm. Similarly, there are so many applications built on top of these remote terminal access applications. Thus, transferring these applications into newer NDN architecture would be highly supported by the development of this work.

Implementation

==============

The standard inetutils library was forked from its mirrored location on GitHub for its adaptation to NDN. The project is currently a private repository, but will be available publicly at the end of class or upon your request. In addition, a separate repository houses C++ classes built for NDN web calls which mimic the underlying Socket connections inside applications such as Telnet (the focus of this project).

The intermediate goal is to ensure that the interfaces to the NDN web calls were similar to the original ones. In this way, the code changed in applications like telnet is minimized to method signatures and include statements. The long term goal is for the interfaces to be the exact same. If this is achieved, it will mean that using HTTP networking or NDN networking will be a matter of package inclusion (a benefit of the straightforward inclusion style of C/C++) and no telnet application code will need to be changed. In the short term, however, it was deemed beneficial to pre-fix socket calls with "ndn" for ease of development and debugging. These method name changes are reflected in all telnet application files that require socket connections.

The Telnet system works as a team of two app binaries, telnet and telnetd, with the former being the client and the latter being the server. The client negotiates a connection to the server and issues commands to the server to which it attempts to comply. Both telnet and telnetd rely on socket connection calls, with the client connecting directly to TCP sockets and the server using inetd linux daemon to handle incoming connection requests. The client networking logic only required method signature changes (detailed above) at the application level to accept NDN connections. NDN endpoints were simply used instead of DNS. The telnetd server network configuration was replaced wholesale with a socket implementation so that the NDN networking adaptations would work similarly to telnet.

Our NDN socket replacement library, ndn-socket, is written in C++. However, since we do our best to emulate the function signatures found in sys/socket.h, our methods are compatible and can can be linked with C application code. The ndn-socket library had to be able to handle two-way communication, so to help implement the required behavior, we used code from a paper posted on the NDN website (https://named-data.net/2017/09/05/tunnel-ethernet-traffic-ndn/). The original project created a two-way tunnel to act as a VPN for traditional IP traffic. To achieve that, both nodes of the tunnel send interest packets to a NDN name prefix that refers to the opposite node. When the opposite node has data to send, it generates a data packet to fulfill one of the outstanding interests.

In our ndn-socket, library we made use of the same code that created a two-way tunnel. Instead of attaching it to the system's local network interfaces, we attached it to an internal string buffer so we could send or receive characters. The send and recv functions of the linux socket library were fairly straightforward to implement. In the traditional linux socket library, the server must create a socket, bind to an address and port, listen on the port, and accept any incoming connections. The client must create a socket and connect to the server. We did our best to emulate that behavior with ndn-socket.

The socket command allocates internal variables that manage the ndn state. The bind command saves the server's NDN name prefix. The listen command creates an NDN interest packet filter that listens for any interests that are pointed to the server. Any incoming interests are placed in a queue. The accept commands picks the first interest from the listen queue and responds to it with the expected tunnel NDN name. From that point, the server is ready to communicate. When the client calls the connect command, an interest is sent to the NDN namespace that was passed into the connect command. The client's NDN prefix is also attached to the interest. When the server responds to that interest with a data packet, the client extracts the server's NDN name from the data packet and the two-way tunnel is started.

As we were implementing the code, there were a few changes we had to make to the telnet and telnetd code that we had not originally considered. The original bind and connect functions accepted a struct object that held IP specific information, such as IP address and port number. These were modified to pass in a string which contains an NDN prefix. At the moment, the prefixes are hardcoded. More modifications would be necessary to allow a prefix to be passed in as a command line argument to these programs. We also found additional calls relating to the network socket in telnetd that were not a part of the sys/socket.h header. Commands such as read and write were easy to implement since it was nearly identical to send and recv. Other functions, such as select and its related FD\_ISSET calls were more difficult. At the moment, we have working implementations that make sense for telnetd's use, but they will likely need more improvement if we wanted it ndn-socket to more generally work across multiple applications.

Experimental Analysis

=====================

Experimental analysis will be conducted with Wireshark through the Wireshark NDN plugin. This will allow us to analyze network capacity, overhead, etc. relative to the original telnet application. In order to be able to observe packet flow, two Named Data Forwarder instances (or NFDs) are required. Otherwise, the traffic stays inside of the single NFD routing mechanisms and goes undetected by other tools.

The experimental analysis has not yet commenced, but is expected to shortly.

Work in Progress

================

Currently, the telnet client is issuing commands to the telnetd server over NDN using the two-way tunnel implementation. Through various means of debugging, including Wireshark and telnetd's internal logger, we can confirm that data is successfully passed. The current implementation does work for the telnet/telnetd scenario, but we have a couple of ideas that may allow ndn-socket to be used across more applications more easily.

In addition to the two-way tunnel implementation, we want to create an ndn-socket variation that uses the ChronoSync library. We would be able to compare different metrics to determine if ChronoSync is a better choice for this use case than the two-way tunnel implementation. At this point, we know we have modified telnet and telnetd applications that can use our current ndn-socket header. It should be quicker to create a ChronoSync version since we only have to modify ndn-socket.

Related Work

============

https://named-data.net/2017/09/05/tunnel-ethernet-traffic-ndn/

Conclusion

==========

As of now, we do know that it is possible to emulate a traditional IP/TCP socket over an NDN network. Our next steps largely focus on observing the performance of these applications to see how well NDN acts as a replacement to IP/TCP. In our first pass of modifying the code, we did find that a number of changes were required to allow the target program to use our ndn-socket library. We hope to refine our code to further reduce the number of changes required before forming an opinion on how easy it is to migrate old applications to the new ndn-socket library.