SETTING UP NAMED DATA NETWORKING IN LOCAL AREA NETWORK

By

YI HUANG

_____

A Thesis Submitted to The Honors College

In Partial Fulfillment of the Bachelors degree
With Honors in

Computer Science

THE UNIVERSITY OF ARIZONA

A U G U S T 2 0 1 3

Approved by:

Dr. Beichuan Zhang
Department of Computer Science

The University of Arizona Electronic Theses and Dissertations
Reproduction and Distribution Rights Form

The UA Campus Repository supports the dissemination and preservation of scholarship produced by University of Arizona faculty, researchers, and students. The University Library, in collaboration with the Honors College, has established a collection in the UA Campus Repository to share, archive, and preserve undergraduate Honors theses.

Theses that are submitted to the UA Campus Repository are available for public view. Submission of your thesis to the Repository provides an opportunity for you to showcase your work to graduate schools and future employers. It also allows for your work to be accessed by others in your discipline, enabling you to contribute to the knowledge base in your field. Your signature on this consent form will determine whether your thesis is included in the repository.

Name (Last, First, Middle)

Huang, Yi

Degree title (eg BA, BS, BSE, BSB, BFA):

BS

Honors area (eg Molecular and Cellular Biology, English, Studio Art):

Computer Science

Date thesis submitted to Honors College:

8/7/2013

Title of Honors thesis:

SETTING UP NAMED DATA NETWORKING IN LOCAL AREA NETWORK

The University of Arizona Library Release Agreement

I hereby grant to the University of Arizona Library the nonexclusive worldwide right to reproduce and distribute my dissertation or thesis and abstract (herein, the "licensed materials"), in whole or in part, in any and all media of distribution and in any format in existence now or developed in the future. I represent and warrant to the University of Arizona that the licensed materials are my original work, that I am the sole owner of all rights in and to the licensed materials, and that none of the licensed materials infringe or violate the rights of others. I further represent that I have obtained all necessary rights to permit the University of Arizona Library to reproduce and distribute any nonpublic third party software necessary to access, display, run or print my dissertation or thesis. I acknowledge that University of Arizona Library may elect not to distribute my dissertation or thesis in digital format if, in its reasonable judgment, it believes all such rights have not been secured.

☒ Yes, make my thesis available in the UA Campus Repository!

Student signature: _____ Date: _____

Thesis advisor signature: Zhang Beichuan Date: 8/6/2013

☐ No, do not release my thesis to the UA Campus Repository.

Student signature: _____ Date: _____

Last updated: 04/01/13

Abstract

IP network is over 40 years old and there are many known problem with it. People are trying to figure out a new way to do networking. Here is one called Named Data Networking (NDN). In this thesis, the advantage of NDN as well as how it works will be discussed. Also, a step by step instruction for how to set up NDN on home routers and how to test-drive NDN with video streaming will be shown. At last, there is a discussion about why NDN content cache in routers is important and how to improve it.

Background

The Internet has played a significant role in the industry over the last 40 years, but its architecture (IP Addressing, Best Effort packet delivery, etc.) remains unchanged during this time. There are many problems discovered such as security, mobility, content distribution, etc. To design a new architecture, many people from all over the world put their efforts in this project called Named Data Networking.

Why NDN

For the Internet architecture we are currently using, the best effort packet structure is the most popular. In the IP header, the most important parts are the source address and the destination address. This architecture looks like an hourglass (See Figure 1 to the left). There are many methods to send packets, many protocols to ensure the format of the content, and many applications that creates and receives packets. In the middle, there is only IP. IP does not provide other services but packets delivery. Note that there is nothing about security or data in IP part. Therefore, it is totally possible that some person could be able to read others' packets and forward them the same packets without being noticed. Since there is no restriction about data encryption, data in packet may be easily read by packets forwarders if the packet creators did not do anything about security.

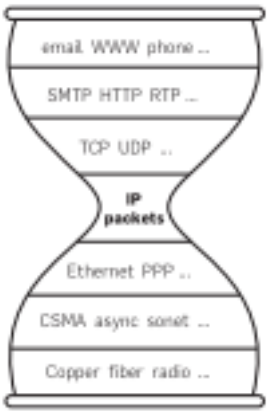

Figure 1 from [1]

There is another significant problem appeared in the current Internet architecture. Traffic near server is incredibly busy. That is because each request has a destination address to a particular server. Once the group of users for a server gets larger, the bandwidth will no longer be able to handle that much traffic. To handle this, Content Distribution Networking is developed. It is still not the best solution.

Named Data Networking (NDN) was designed to solve all those problems. In NDN, clients tell the network what they need instead of using the network to send requests to servers. That is, clients do not need to know server's IP address. Clients ask nearby routers for certain data packets by sending interest packet. "Upon receiving an Interest, a router first looks in its local cache and if a copy of the requested data packet is found, it instantly sends it back. Otherwise the router performs a longest prefix match on its Forwarding Information Base (FIB) and forwards the interest to the next hop towards the data source." [2] After getting the data packet, the router looks up his Pending Interest Table (PIT) to figure out who asked for the packets and forward the data packets.

Using content names instead of IP addresses won't result in confusions. "A consumer asks for content by broadcasting its interest over all available connectivity. Any node hearing the interest and having data that satisfies it can respond with a Data packet. Data is transmitted only in response to an Interest and consumes that Interest." [1] The names are allowed to have multi levels and to be detailed. The application that creates the packet gives the content name. Routers determine whether they have the wanted data packet already or not by examine the details included in the interest packet.

Named Data Networking is secure. Data packets are required to include signature. Applications could verify the signature and decide whether the packet is true or faked.

How NDN Works

In IP network, applications provide their content. Operating systems construct IP packets and send them out. For NDN, applications name its data. Therefore, resources have their names. The idea is, consumer asks network for resource with a specific name, and network gives consumer the data they want.

To do this, we have two kinds of packets in NDN. Consumers construct and send interest packets. The most important part in interest packet is resource name. Since there is no data in interest packets, the size of those packets is relatively small. Once producers with the requested data see interest packets, they reply with data packets.

How do packets travel back and forth between producers and consumers without IP addresses? In NDN, interest packets are routed based on the name of data in the packets. Once a router sees an interest packet and if there is no useful data in its cache, the interest packet will be added to routers pending interest table before the router forward it. When the interest packet reaches appropriate producer, the producer will send data packet to the router who provide the interest packet. Since routers on the path between consumer and producer have the interest packet in their pending interest tables, data packets can track back along pending interest table entries. This approach is different from IP routing. In NDN, producer does not know consumer's address. In other word, producer does not need to know source address because NDN can deliver data packets without source addresses.

To ensure security, every data packet has a signature. Consumers will drop data packets without a valid signature. Compare to NDN, IP does not have any kind of encryption. Applications need to handle security issues in application level which is not very convenient. And developers with less sense of security may produce applications that can easily be hacked. In NDN, data packets have to be signed. This feature is in the protocol and application does not need to handle that.

Steps for Setting up NDN on Home Router

Now we have a common sense of the advantage of NDN. For now, we have an implementation called CCNx which is built on top of IP and can be deployed to home routers. Here is the instruction of how to compile, deploy, and test CCNx on home routers.

How to cross-compile CCNx for DD-WRT and OpenWRT

This is a HowTo for cross-compiling CCNx for DD-WRT and OpenWRT. This process utilizes the DD-WRT/OpenWRT toolchain compiler binaries, so the steps for compiling CCNx for OpenWRT and DD-WRT are exactly the same, only we are using different compiler binaries.

Get Toolchain for DD-WRT

If you are compiling for DD-WRT, you must download DD-WRT's current toolchian package, which is available here:

http://www.dd-wrt.com/dd-wrtv2/downloads/others/sourcecode/toolchains/current-toolchains.tar.bz2

This download contains several different compilers for various architectures, and the toolchain you use depends on the architecture of the router you are cross-compiling for. This HowTo was done while cross-compiling CCNx for an instance of DD-WRT running on an Asus RT-N16 router, which has a Broadcom chipset, so the cross-compile binary used is: current-toolchains/toolchain-mipsel_4.1.1_BRCM24/bin/mipsel-linux-uclibc-gcc

DD-WRT should have an equivalent version of gcc in each of its toolchain directories, but note that for DD-WRT, this process was only tested on a broadcom router. If you are cross- compiling for a non-broadcom based router, it is best to first identify the toolchain you need, then at every step make sure that the analagous thing can be done using your particular toolchain's version of gcc. Also note that the toolchain binaries run on 64-bit linux.

Get Toolchain for OpenWRT

For OpenWRT, getting a cross-compiler is slightly different. Rather than provide a variety of ready-made toolchain compilers, OpenWRT allows you to create your own cross-compiler based on whatever router you are using. You can do this by installing the OpenWRT buildroot. The OpenWRT buildroot is primarily used to build the OpenWRT firmware from scratch, but it also happens to build for you a cross compiler that will output binaries that will run on the router you are building for. The buildroot installation process is outlined here: *http://wiki.openwrt.org/doc/howto/buildroot.exigence*

In step 4, use the command 'make menuconfig' to configure the build for your specific router. I only had to change the 'Target System' and 'Target Profile' variables to match my router. This is enough to configure the build so that it will give us the cross compiler we need. (Don't be concerned about all the other variables and functionality of the menuconfig. It is meant to help you build your own version of the firmware, which we are not concerned with. We only care about the cross-compiler) Now to start the build, run 'make'. (This will take quite some time. I had to leave mine compiling overnight) Once the build is done, you can find the toolchain directory in trunk/staging_dir. The name of the toolchain will depend on the type of router you are compiling for, but my cross-compile version of gcc was called:

trunk/staging_dir/toolchain-mips_r2_gcc-4.6-linaro_uClibc-0.9.33.2/bin/mips-openwrt-linux-uclibc-gcc

Now you have a cross-compiler to building CCNx. For the remainder of the HowTo, replace <path_to_your_toolchain_binary>, with the full path to the cross-compile gcc you are using, and <path_to_your_toolchain's_root> with your toolchain's root directory.

Step to cross-compile CCNx (for both DD-WRT and OpenWRT)

1. Augment the toolchain with openssl:

If we download ccnx-0.6.1 and immediately configure it, then build it using a fresh download of your particular toolchain (The easiest way for me is to run 'make CC=<path_to_your_toolchain_binary>;), the build fails immediately because the toolchain doesn't have certain openssl header files. So first we need to download the openssl dev kit. CCNx can only be compiled with openssl version 1.0.1 or later, so I downloaded opnessl-1.0.1c.tar.gz from: *http://www.openssl.org/source/*, which works for my release of CCNx(0.6.1).

We now need our toolchain to have all the header files in the openssl directroy tree. My linux machine already had openssl v.1.0.1c installed, so I simply copied my /usr/include/openssl directory into <path_to_my_toolchain>/include/, and that did the trick. If you don't already have an opnessl directory on your machine, you can manually create a directory called 'openssl' in your <path_to_your_toolchain's_root>/include directory, and manually copy all of openssl's header files into it.

Once this is done, we can try to build CCNx again, and we see that we can now compile the first few C files, but the build fails once we try to link to the lcrypto library. This is because the DD-WRT toolchain does not have a libcrypto.a file. It is best to check if your toolchain already has this library file in <path_to_your_toolchain's_root>/lib, but if it doesn't we will need to cross-compile libcrypto.a, which is the real reason why we downloaded the openssl dev kit.

Now, we can cross compile libcrypto.a:

a. cd to the root directory of openssl-1.0.1c
b. Open 'Configure' script
c. Change line 361: "linux-x86_64", "gcc:-m64 -DL_ENDIAN -DTERMIO -O3 etc..." to: "linux-x86_64", "",
   (This takes out all of the optimization options that dd-wrt toolchain compiler doesn't recognize)
d. Run './Configure linux-x86_64'. If this step fails, consult the INSTALL file in openssl-1.0.1c, which may have some helpful tips. (I recall having to run './Configure linux-x86_64 no-asm' to get rid of assembler errors)
e. In the root directory of openssl-1.0.1c, run 'make CC=<path_to_your_toolchain_binary>'
f. This will generate a libcrypto.a file in the current directory. Copy libcrypto.a to <path_to_your_toolchain's_root>/lib
g. Run 'make CC=<path_to_your_toolchain_binary>' in the CCNx home directory to see if we can link to libcrypto.a successfully

2. Cross-compile libresolv.a -- NOTE: If you are compiling for OpenWRT, you do NOT need to do this step. Skip to step 3

The CCNx cross-compile in step (f) above should be failing because of an "undefined reference to __dn_skipname". This function should be defined in the toolchain's libresolv.a file, but it's not, so we have to cross-compile our own libresolv.a.

libresolv is a c standard library, and Google led me to a patch of uClibc-0.9.28.1 which adds the __dn_skipname functionality to a source file in the uClibc tree called resolv.c. The patch is found here:

http://lists.uclibc.org/pipermail/uclibc/2011-July/045579.html

It turns out that all we need to do patch resolv.c in the uClibc-0.9.28.1 package (along with some other minor changes), cross-compile it into resolv.o, then add this resolv.o file to the libresolv.a archive file in the toolchain. Here are the exact steps:

a. Download uClibc-0.9.28.1.tar.bz2 from http://www.uclibc.org/downloads/old-releases/
b. Copy the text from the patch into a file called whatever-you-want-to-call-it.patch, and make sure this patch file is in the root directory of uClibc-0.9.28.1 (the patch starts at the line "diff --git a/include/resolv.h b/include/resolv.h", and ends after "#endif /* L_ns_name */").
c. Run 'patch -i whatever-you-want-to-call-it.patch -p1'. For me, the first few patches for resolv.h failed, but the last patch for resolv.c succeeded, which is ok because resolv.c is all we are concerned.
d. The patch added the definitions of two functions, dn_skipname and ns_name_skip, to resolv.c. But dn_skipname also calls a function called labellen, which is not already defined in resolv.c. In fact, I could not find any mention of this function from the webpage which lists the patch that added dn_skipname, even though dn_skipname needs labellen to execute. I eventually found a defintion for lebellen in the source code for eglibc-2.14. To save you the trouble of tracking down this package, I have listed definition of labellen below. Copy this function definition and paste it into resolv.c directly above the definitons that were added in the patch:

```
static int
labellen(const u_char *lp) // copied from eglibc-2.14/libc/resolv/ns_name.c
{
        int bitlen;
        u_char l = *lp;

        if ((l & NS_CMPRSFLGS) == NS_CMPRSFLGS) {
                /* should be avoided by the caller */
                return(-1);
        }

        if ((l & NS_CMPRSFLGS) == NS_TYPE_ELT) {
                if (l == DNS_LABELTYPE_BITSTRING) {
                        if ((bitlen = *(lp + 1)) == 0)
                                bitlen = 256;
                        return((bitlen + 7 ) / 8 + 1);
                }
                return(-1);     /*% unknown ELT */
        }
```

```
        return(l);
} // end of function labellen
```

e. After adding lebellen to resolv.c, add the following two lines to the beginning of resolv.c (right before the first #define):

   #define NS_TYPE_ELT 0x40
   #define DNS_LABELTYPE_BITSTRING 0x41

f. As the last change to resolv.c cut the last line of resolv.c (#endif /* L_ns_name */) and paste it directly above the definition of labellen, this way the changes we made lie outside of the '#ifdef L_ns_name' statement, and they are always defined.

g. Now we need to make some changes to the uClibc-0.9.28.1 tree. As it stands right now, resolv.c will use header files in the uClibc tree that will throw errors in the toolchain compiler. Namely, the header files are:

   uClibc-0.9.28.1/libc/sysdeps/linux/common/bits/kernel_types.h
   uClibc-0.9.28.1/libc/sysdeps/linux/common/bits/wordsize.h
   uClibc-0.9.28.1/libc/sysdeps/linux/common/bits/endian.h

   In order to avoid the errors thrown by these files, change their names:

   uClibc-0.9.28.1/libc/sysdeps/linux/common/bits/kernel_types.h.old
   uClibc-0.9.28.1/libc/sysdeps/linux/common/bits/wordsize.h.old
   uClibc-0.9.28.1/libc/sysdeps/linux/common/bits/endian.h.old

   This will cause the toolchain compiler to not find the files in the uClibc tree, and look for them in the toolchain include tree instead, where the toolchain versions of these files are located, which won't throw any errors.

h. Finally, we can cross compile resolv.c. cd into uClibc-0.9.28.1/libc/inet and run this command:
   <path_to_your_toolchain_binary> -c resolv.c –I
   <path_to_your_toolchain's_root>/include/ -I ../../libc/sysdeps/linux/common/

   This should output a resolv.o binary file with the definitions of the functions we need to compile ccnx. To check if the cross-compile worked, run 'nm resolv.o'. This will show you all the symbol names defined in the file. It should look something like this:

```
$ nm resolv.o
00000378 T __dn_skipname
```

```
                 U __errno_location
00000124 T __ns_name_skip
                 U _gp_disp
00000000 t labellen
```

We see that all three of the functions we added have been compiled, so we should be good to go.

i. Now we have to add this resolv.o file to the libresolv archive file in the toolchain. To do this, run:

   ar -r <path_to_your_toolchain's_root>/lib/libresolv.a resolv.o

   Do NOT create a new libresolv.a using this resolv.o file. You must only add the newly created resolv.o to the existing libresolv.a file that is already in the toolchain lib/ directory.

j. Lastly, try to build CCNx to see if we can link to libresolv.a successfully.

3. Cross-compile libexpat

Now if we try to compile CCNx, the build fails because we dont have an 'expat.h' header file. We will need some expat header files as well as a libexpat.a library file.

a. Download expat-2.0.1.tar.gz from http://sourceforge.net/projects/expat/

b. Copy expat-2.0.1/lib/expat.h and expat-2.0.1/lib/expat_external.h into your path_toolchain/include directory

c. Try compiling ccnx again, and it should now fail because our toolchain compiler cant find '-lexpat', i.e. the toolchain does not have the libexpat.a library file. We need to cross-compile expat-2.0.1 to get this file.

d. To cross compile libexpat, cd to the root directory of expat-2.0.1 and run this command:

   ./configure --build=mipsel-linux --host=<your_host_machine> x86_64-unknown-linux-gnu CC=<path_to_your_toolchain_binary>

   Note that the '--host' variable refers to the machine you are using to run this command, which for me is a Ubuntu-Linux 64-bit machine.
   So I would use '--host=x86_64-unknown-linux-gnu' for the host option. You can use './configure --help' or run './configure' to find out how to specify the host option if your setup is different than mine.

e. After we run the expat configure script with our exact specifications, we can now simply run 'make' from the root directory. This will produce a 'libexpat.a' file, which we can now copy to <path_to_your_toolchain's_root>/lib.

f. Try to build CCNx one last time, to see if we can make it past the step that links to –lexpat.

4. Cross-Compile libpcap

The next error in the CCNx compile should be due to a 'pcap.h' file that the toolchain doesn't have. We will deal with this the same way we did in part 3, by copying the pcap header files and cross-compiling libpcap.a

a. Download libpcap-1.1.1.tar.gz from http://www.tcpdump.org/

b. Copy both the header file libpcap-1.1.1/pcap.h' and the directory 'libpcap-1.1.1/pcap' to <path_to_your_toolchain>/include

c. Try to compile CCNx, and it should fail bacause it can't find '-lpcap', which means we need libpcap.a

d. From the root directory of libpcap-1.1.1, run this command:

./configure --build=mipsel-linux --host=<your_host_machine> CC=<path_to_your_toolchain_binary> --with-pcap=linux ac_cv_linux_vers=2 --target=mipsel-linux

Note again that --host=<your_host_machine> refers the machine you are cross-compiling on. You can most likely set the host to whatever you used for step 3d.

e. Now run 'make' from the libpcap root directory

f. This will produce a 'libpcap.a' file that we can now copy into <path_to_your_toolchain's_root>/lib

DONE!

Assuming that we can now link to the pcap library successfully, we shoud be able to completely compile CCNx without error. Build CCNx one last time to see if we can get all the way through. After compilation, its a good idea to copy the whole ccnx-0.6.1/csrc directory to the router and run some of the test scripts to make sure CCNx is working.

How to Mount a USB pendrive

To mount a USB drive, an attitude adjustment version of openwrt is preferred. For some trunk version, some required kernel modules are not supported. For example, I use openwrt-ar71xx-generic-wzr-hp-ag300h-squashfs-sysupgrade.bin for Buffalo wzr-hp-ag300h.

To flash the router, simply go to router's web interface. System -> Backup/Flash Firmware -> Flash new firmware image.

SSH to your router and run this command:

```
opkg update
opkg install kmod-usb-storage kmod-usb2 kmod-nls-utf8 kmod-fs-ext4 kmod-fs-vfat
```

Then you should be able to mount your USB drive in ext4 format.

Use USB drive as SWAP PARTITION

To use USB drive as swap partition, you need to install block-mount using opkg command. Then do the following:

```
/etc/init.d/fstab enable

/etc/init.d/fstab start
```

The configuration file for fstab is '/etc/config/fstab'. Please disable automount and autoswap by setting 0's on related values. And set option 'enabled' under config 'mount' to 0 as well.

Make your USB drive a swap partition (substitute '/dev/sda1' with your USB drive):

```
mkswap /dev/sda1
swapon /dev/sda1
```

To view your memory and swap space usage, you could simply use free command.

To unmount a existing swap space, run this command:

```
swapoff /dev/sda1
```

Now we have CCNx compiled and ready to run on your home router. The following instruction allows you to play with CCNx using two computers and one router.

Video Streaming

This is an example of using CCNx to stream video on a small network (two laptops and one router).

Before you start, please read:

This particular network consists of two laptops with a router in the middle. The first laptop, which we will call 'laptopA', is serving the video, while the second laptop (laptopB), will be streaming the video. The router will be forwarding all the activity between the two machines, and we will just call it 'router'.

This HowTo is simply a configuration HowTo - i.e. it assumes you have installed NEWEST CCNx on all three nodes of the network, and you have installed vlc and the ccnx plugin for vlc on laptopB. We will only go over how to set up the network so that video streaming will work.

If you did not install VLC and CCNx plugin for VLC, here are the commands to get you a VLC and approprate plugin on Ubuntu (You should have CCNx for ubuntu compiled before doing the following steps):

```
sudo apt-get install vlc libvlc-dev
cd ccnx-master/apps/vlc
mv Makefile.Linux Makefile
make
make install
```

Also, this HowTo uses ccnping to test connectivity. While ccnping is not necessary to stream video, it is a good idea to install ccnping and ccnpingserver on all three machines to make absolutely sure you have a basic connection between each machine.

Setting Up

Here are the configuration commands to be run on each machine:

NOTE: the ip addresses used are specific to my network. You must substitute in your own ip addresses.

LaptopA (10.1.1.116)

./ccndstart
./ccndc add ccnx:/ccnx.org udp 10.1.1.1 9695
./ccndc add ccnx:/ccnx:/my_network/ping udp 10.1.1.1 9695
./ccnpingserver ccnx:/my_network/ping/laptopA &
mkdir repo ./ccn_repo repo/
./ccnputfile ccnx:/my_network/videos/test ../../videofiles/test.m4v

NOTE: If using ccnr instead of ccn_repo, replace "./ccn_repo repo/" with these two commands: export CCNR_DIRECTORY='<absolute path to repository (you could create a new directory before running this command)>'./ccnr

Router (10.1.1.1)

./ccndstart
./ccndc add ccnx:/ccnx.org udp 10.1.1.116 9695
./ccndc add ccnx:/ccnx.org udp 10.1.1.101 9695
./ccndc add ccnx:/my_network/videos tcp 10.1.1.116 9695
./ccndc add ccnx:/my_network/ping/laptopA udp 10.1.1.116 9695
./ccndc add ccnx:/my_network/ping/laptopB udp 10.1.1.101 9695
./ccnpingserver ccnx:/my_network/ping/router &

LaptopB (10.1.1.101)

./ccndstart
./ccndc add ccnx:/ccnx.org udp 10.1.1.1 9695
./ccndc add ccnx:/my_network/videos tcp 10.1.1.1 9695
./ccndc add ccnx:/my_network/ping udp 10.1.1.1 9695
./ccnpingserver ccnx:/my_network/ping/laptopB &

Test Connection

Now, from laptopA, run

./ccnping ccnx:/my_network/ping/router

and

./ccnping ccnx:/my_network/ping/laptopB

to see if we get responses from both of these. And do the same thing on laptopB, but ping laptopA.

Test Repository

From laptopB, run ./ccngetfile ccnx:/my_network/videos ../../MyTestVideo.m4v. When done, verify whether MyTestVideo.m4v has the same size as test.m4v on laptopA.

Start Streaming Video!

To video stream, on laptopB run:

vlc ccnx:///my_network/videos/test

Note that we stored the video file with ccnputfile as 'ccnx:/my_network/video/test' but when we streamed it we used 'ccnx:///my_network/video/test'. The two extra forward slashes are necessary because vlc does not recognize 'ccnx:/...' as a valid host name. It's slightly inconvenient, but that's how it works.

Also note that the video file I used was test.m4v - which is just an arbitrary video file I downloaded from the internet. VLC will play almost any video file, so you just have to get your hands on a video file of any format and put it in the repo with ccnputfile.

Discussions on Cache/Swap Spaces

Content cache for routers is preferred in NDN. If routers can cache some data packets, it could be a “part-time” data source before data packets expire. If there is a data packet in cache, and the router see an incoming interest packet that is asking for the data packet, the router can just reply with the data packet instead of forwarding interest packet to next hop. IP network cannot do the same thing because there is no signature carried by each data packet. If IP routers do the same thing, cached data could be easily compromised or hacked.

Current implementation of router cache is just using router’s memory. This setting brings up a problem. That is, most routers for today do not have much memory since there is no need for IP router’s to cache content.

To make a use of this feature of NDN, we mounted a USB flash drive on a home router and make the flash drive to be swap space. During the experiment, we observed that the router was becoming increasingly slow. That is because flash drive is actually much slower than real memory. Especially when we try to stream video in this situation, the packets timed out much more often. Making flash drive swap space looks like we are giving the router more memory, but router may move CCNX core to flash drive too. That is the reason why the performance goes down when we add a swap space.

When mounting flash drive and expect it to be content cache, to avoid performance issue from happening, an approach is to make content cache configurable. Therefore, content cache can be configured to use memory or any directory in the file system. And we could mount flash drive and configure content cache to use flash drive only. In this approach, flash drive will not be mounted as swap space, so CCNX core will never be moved to flash drive and triggers a performance outage.

Configurable content cache also solves some other problems. Most home routers are i386 and can have no more than 2Gigabytes of memory. Therefore, current CCNX can have no more than 2G content cache. With configurable content cache, this issue is solved by mounting a much larger flash disk and configures it to be content cache.

Implementing configurable content cache is not trivial. Firstly, we will need an indexing plan for managing contents. Secondly, since content store is currently only in memory, the file system part should be implemented separately as well as data indexing and searching. To avoid complication, using Berkeley DB is an acceptable choice. Berkeley DB is portable, fast, and reliable. Also, it is easy to use by other applications. Due to its simplicity and reliability, it is considered to be a good way to implement file system cache.

References

1. Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, Rebecca L. Braynard “Networking Named Content” Magazine Communications of the ACM Volume 55 Issue 1, January 2012 Pages 117-124

2. Christos Tsilopoulos, George Xylomenos (2011). “Supporting Diverse Traffic Types in Information Centric Networks”. Proceeding. ICN ’11 Proceedings of the ACM SIGCOMM workshop on Information-centric networking Pages 13-18