CSE 534 Spring 2014

Project 2

Last updated 5 May, 2014.

Due date Thursday, 8 May.

The purpose of this project is to have you use the ndnSIM module of the NS-3 network simulator to carry out some performance-exploring experiments on the NDN/CCN network architecture.

– You are to work on this Project in groups of two.

– Each group should hand in a hard copy of a report in class on 8 May that addresses the issues highlighted in blue font in the Project description below.

Make sure that your report is structured along the "section" numbers (A1., A2., B1., B2., C1., C2.) of the Project description below.

Also, please email me a soft copy of your report, and include your script files and (samples) of the output trace files from which the results you report on were derived.

– I have set up a Discussion Forum for the Project on Blackboard. Use it to share useful items of documentation, helpful tips, seek advice and guidance from your peers, etc. Also use it to find a partner if you do not already have one.

Installation, Preliminary Experimentation, and Tutorials

ndnSIM module tutorial & installation documentation: http://ndnsim.net/tutorial.html .

Mailing list: http://www.lists.cs.ucla.edu/mailman/listinfo/ndnsim .

– Read the *Introduction* (http://ndnsim.net/intro.html) to get a general idea of ndnSIM.

– Read *Getting Started* (http://ndnsim.net/getting-started.html) to install ndnSIM on your machine.

- Windows users should try installing via VMware Player, VMware Workstation, or VirtualBox, running the required Ubuntu Linux or Mac OS distribution.

  ndnSIM & NS-3 are not officially supported on Windows. I am, perhaps somewhat optimistically, assuming that they will run o.k. if you install a virtualization package that actually runs an image of one of the required OSes (rather than simply "emulate" such an environment using, for example, *Cygwin* – note that NS-3 is known to have problems running under Cygwin). But the truth is that I don't know for sure, which is one reason Windows guys in particular should make a really early start on attempting the installation, and then the preliminary experimentation below, just in case you hit significant problems.

Installation is straightforward if all steps are followed. Questions regarding installation issues to install can be searched for and inquired about on the mailing list. It is not necessary to read through subsection entitled *Simulating using ndnSIM*.

- You have to write scripts to generate the simulation scenarios that define various parameters, such as topology, traffic patterns etc. (this is somewhat similar to NS-2, for those of you familiar with it). To get acquainted you need to go through the following:
  - Read through *ndnSIM helpers* (http://ndnsim.net/helpers.html) in conjunction with *Examples* (http://ndnsim.net/examples.html). In the subsection *Examples*, you should try to understand at least *Simple Scenario*, *9-node grid example*, and *9-node grid example using plugin*.
  - Based on what you have learned above, write a script to create a simple 6-node "dumb bell" topology as follows. For help with the points below, refer to example: http://ndnsim.net/metric.html#example-of-packet-level-trace-helpers.
    - Two "core" router nodes that are attached by a point-to-point link. Call these nodes A and B.
    - Each core node has two "edge" nodes hanging off it (for a total of four edge nodes).
    - Make the two edge nodes hanging off node A a *consumer* nodes, and the two hanging off node B a *producer* nodes.
    - Use the *AnnotatedTopologyReader* to read in the topology file describing the dumb bell (you will have to create this file).
    - Configure the caches, such that they are enabled on the core nodes, but are disabled at the edge nodes. The cache size for the core nodes should be set to 3 objects. The cache policy will be LRU. Use the *SetContentStore* method to set the parameters accordingly (see http://ndnsim.net/metric.html#example-of-content-store-trace-helper).
    - Install a *producer* on each edge node hanging off node B (call these Producer1 & Producer2). Define the object space that a *producer* caters with the *SetPrefix* method.
    - Create two *ConsumerCBR* applications on each *consumer* node, one to request content from Producer1 and the other from Producer2. For the first *consumer* node, have the *ConsumerCBR* application that requests content from Producer1 start at time 0.0; the *ConsumerCBR* application that requests content from Producer2 starts at time 1.0. For the other *consumer* node, on the other hand, we do the opposite. Have the *ConsumerCBR* application requesting content from Producer1 start at time 1.0; the *ConsumerCBR* application requesting content from Producer2 starts at time 0.0.

      Do the above using the *StartSeq* parameter of the *SetAttribute* method.
    - Set the generation rate for interest packets in each *ConsumerCBR* to 1 interest packet per second.
    - Make the forwarding strategy *BestRoute* (http://ndnsim.net/fw.html#forwarding-strategies).
    - Use the *GlobalRoutingHelper* to populate the forwarding tables, which in NDN are called the "Forwarding Index Base" (FIB).
    - Set the experiment to run for 5 seconds.
    - Compile, modify and run your script to make sure there are no errors.

A1. Briefly report on your experience in setting up and getting the dumb bell network going, as per the specifications above. Hand in the script that you developed.

– A simulation generates some output. NS-2, for example generates a trace file. NS-3 has enhanced upon this by creating a tracing subsystem.

- Firstly, familiarize yourself with the kind of tracers that are provided with ndnSIM: http://ndnsim.net/metric.html. There are two of these in particular that will of concern to us for the project:
  - *CSTracer* (content store trace helper): http://ndnsim.net/metric.html#example-of-content-store-trace-helper.
  - *AppDelayTracer* (application-level tracer helper): http://ndnsim.net/metric.html#application-level-trace-helper.
- Run the example scripts associated with these two tracers, and observer the output files.
- The motivation behind, and advantage of, using the tracing subsystem is discussed here: http://www.nsnam.org/docs/manual/html/tracing.html. This discussion should be helpful to you since you need to update the application-level tracer helper in your experiments in order to record a value/metric that is defined by you.
- Now, try to create a Packet Tag, and store that within the data packet. The purpose of the tag will be to track whether the data packet was satisfied by a cache hit or not. This information will then be accessed through the application-level tracer (which will need to be updated accordingly).
  - Unlike NS-2, in which packet objects contain a buffer of C++ structures corresponding to protocol headers, each network packet in NS-3 contains a byte Buffer, a list of byte Tags, a list of packet Tags, and a PacketMetadata object (http://www.nsnam.org/docs/release/3.10/manual/html/packets.html).
  - You have to define and create a new packet tag type object (http://www.nsnam.org/docs/release/3.10/manual/html/packets.html#adding-and-removing-tags). There are several packet tags that have been implemented. For reference, you can look at the packet tag type object which is used to keep track of hop counts (http://ndnsim.net/doxygen/ndn-fw-hop-count-tag_8cc_source.html and http://ndnsim.net/doxygen/ndn-fw-hop-count-tag_8h_source.html). You need to create something similar to that.
  - The new packet tag type object keeps track of when there is a cache hit at a node. The network-level code which deals with this is implemented in the *OnInterest()* method (line 145) of the file *ndn-forwarding-strategy.cc* (http://ndnsim.net/doxygen/ndn-forwarding-strategy_8cc_source.html). This method is responsible for processing the incoming Interest packets. You will need to find the code within this method which deals with a cache hit, create an instance of your packet tag, and append it to the data packet.
  - You also need to modify the application-level tracer so that it accommodates an additional column in its output. At present, it has 9 columns (http://ndnsim.net/metric.html#application-level-trace-helper). The additional column will record a value 1 if the data packet was satisfied from the cache, and a value 0, if it was satisfied from the source/owning node for the object.

- Testing: Install the *AppDelayTracer* in the dumb bell topology script you created. Re-run the experiment, and observe the values in the new column which you have created to record cache hits. This script scenario will have several cache hits, which should be captured with the new packet tag type you implemented, and recorded in the trace file.

A2. Briefly report on the modifications you had to implement in order to introduce the new packet tag. Where and how did you modify the *OnInterest()* method? What about the *AppDelayTracer*? Hand in sample output showing the extra column you introduced.

Project Experiments

For the next set of simulations, we will need to download the following files from the course webpage, and place them into their respective folders within the ndnSIM file hierarchy.

1. Download the script files *ndn-project-1.cc* and *ndn-project-2.cc*. Place these files in the folder *src/ndnSIM/ns-3/scratch*.
2. Download the *ndn-consumer-zipf-mandelbrot-39.cc* and *ndn-consumer-zipf-mandelbrot-39.h* files. Note that these files were modified/updated on 30 April, so please make sure you download the updated copies. Place these files in the *src/ndnSIM/ns-3/src/ndnSIM/apps* folder. These are modified versions of the original *ndn-consumer-zipf-mandelbrot application* files. They will work only with the topology given below. Read about the *ndn-consumer-zipf-madelbrot application* at http://ndnsim.net/applications.html#consumerzipfmandelbrot.

   As of 30 April, there are two "new" files, *ndn-consumer.cc* and *ndn-consumer.h*, which also need to be downloaded and placed in the same folder *src/ndnSIM/ns-3/src/ndnSIM/apps* in which you placed the *ndn-consumer-zipf-mandelbrot-39.cc* & *.h* files above. Versions of these two "new" files already exist in that folder (from when you installed ndnSIM), so the copies I am giving you here should overwrite the ones that came with the installation.

   Note: In *ndn-project-1.cc* and *ndn-project-2.cc*, the *ndn-consumer-zipf-madelbrot application* is instantiated with the following command: *ndn::AppHelper consumerHelpter* ("*ns3::ndn::ConsumerZipMandelbrot39*"). This is to invoke the modified version in *ndn-consumer-sipf-mandelbrot-39.cc* and *ndn-consumer-sipf-mandelbrot-39.h* .
3. Download the topology file *topo-39.txt*. Place this file in the *src/ndnSIM/ns-3/src/ndnSIM/examples/topologies* folder. The illustration of the topology is given in file *topo-39.png*. It consists of 8 core CCN router nodes (nodes 0 – 7) and 31 edge CCN router nodes (nodes 8 – 38). Edge nodes are both consumers and producers. **Note: Do not edit this file.**
4. The main parameter/metrics that you will need to keep track of is the hop count. *AppDelayTracer* keeps track of hop counts, ~~but not in a slightly unusual way~~. ~~Firstly, *AppDelayTracer* keeps track of the round trip hop count (essentially hop count for the entire trip: Interest packet outbound and Data packet inbound). Secondly, in addition to incrementing the count of the packets as they are forwarded hop by hop in the network, *AppDelayTracer* also accounts for an additional hop when the packet is delivered from the network-level to the application-level at the final node. This situation occurs when an Interest packet is satisfied at the producer/repository node that "owns" the data object in~~

~~question, and when a Data packet is received at the requesting node. Thus, for example, when you see a value of 6 hops in the *AppDelayTracer* trace file, this means that the Interest packet took 2 hops in the network, the corresponding Data packet also took 2 hops in the network, and the remaining two hops count the delivery of both the Interest and Data packets from the network-level to the application-level at their respective final nodes (which also implies that the data object was not satisfied from a cache). If you see an odd value such as 5 hops, then this means the Interest packet took 2 hops and the Data packet also took two hops. The remaining 1 hop accounts for the delivery of the Data packet up the protocol hierarchy to the application-level at the requesting node. In this case, the Interest packet was satisfied at the network-level cache of some "intermediate" node, and not at the repository node which owns that object. For further explanation look at the description of hop count in~~
~~http://ndnsim.net/metric.html#application-level-trace-helper~~.

~~The section crossed out above was based on what ndnSIM's documentation of *AppDelayTracer* says. But it turns out not to be true. In one sense, this is fortunate because the actual situation is much simpler and more in line with what one would have expected. *AppDelayTracer* output gives you the total, round-trip number of hops actually travelled in the network by the outgoing Interest packet and its corresponding incoming Data packet. So it is always going to be an even number. No extra "hops" are added for going up the protocol hierarchy at a node, despite what the documentation says.~~

~~This, however, means that for you to find out whether a given data object was fetched: (i) from the repository edge node that "owns" that object; or alternatively, (ii) due to a cache hit at some intermediate node; you will have to depend on the extra column of information output you introduced into *AppDelayTracer* in the context of item **A2.** above~~.

5. You will need to write a script to extract the information you need from the *AppDelayTracer* trace file. You will need the hop count (which you need to readjust based on the explanation above so that you are only counting the actual number of hops in the network, and not the "extra" hops, if any, up the protocol hierarchy at the final node of a packet), as well as the value in the new column you created in the *AppDelayTracer*.

   Note: The *AppDelayTracer* output contains **two** lines for each (Data) packet that is satisfied. These two lines show differences in column 5, and might (but need not necessarily) also show differences in columns 6 & 7. There should be no differences between the two lines so far as the data you need to extract goes. Thus, for your experiments, you can traverse either the odd lines or the even lines of the *AppDelayTracer* trace file.

6. To generate a very light traffic load we have set up a Poisson generation rate of 0.032 interests per second at each edge node. This means that across all 31 edge nodes, we will generate approximately 1 interest per second (0.032 × 31 nodes = 0.992). In our experiments, we want to generate a total number of requests in the network that is equal to 5 times the number of objects, which is 10,000 objects. Therefore, you should let the applications at edge nodes generate Interest packets for 50,000 seconds, which will cause, on average, a total of 50,000 Interest packets to be generated. Run your simulations for 55,000 seconds to give time for all these Interest packets to be satisfied. If you find that not all generated Interest packets are satisfied at the end of 55,000 seconds simulation run time, then you will have to readjust (increase) this time duration. This can be done using the following code: *Simulator::Stop* (*Seconds* (*55000.0*))*; .* The *CSTracer* has been set up to be 1 second less that the simulation stop time:

ndn::CsTracer::InstallAll ("*cs-trace.txt*", *Seconds* (*54999.0*)); . Thus, if you change the time duration of the simulation, do not forget to update the *CSTracer* accordingly.

7. For Experiment Set 2, the traffic load will need to be increased. You will need to go to file *ndn-project-2.cc* and update the *Frequency* attribute of each *ndn-zipf-madelbrot application* there, using the *SetAttribute* method: *consumerHelper.SetAttribute* ("*Frequency*", *StringValue* ("*0.032*")).

   Note that as you increase the generation rate for Interest packets, you will need to decrease the time at which edge nodes stop generating these packets (see *consumerHelper.SetAttribute* ("*StopTime*", StringValue ("50000.0"); in each *ndn-zipf-madelbrot application* in file *ndn-project-2.cc*), and adjust the simulation run time, such that we are still generating and servicing, on average, only 50,000 interest packets. Do not forget to also adjust the *CSTracer*.

Experiment Set 1:

Goal: To observe the effect, in terms of bandwidth savings, of in-network caching in NDN compared to an IP network.

(a) To simulate the IP-network-like instance, we need to run file *ndn-project-1.cc* with all caches disabled. This is the "default" setting in the *ndn-project-1.cc* file. Note that since we are running with a very light traffic load, there will essentially be no PIT aggregation occurring, and all requests will be satisfied from their respective producer repository) edge nodes. This is equivalent to IP network behaviour.

(b) Now we introduce caching. The "object space" (i.e., the total number of objects in the network) is set at 10,000 objects. This is done when we evoke the *ndn-consumer-zipf-madelbrot application* in files *ndn-project-1.cc* and *ndn-project-2.cc*.

We will run the same experiment as in (a) above, but for the following cache sizes: 50 (0.5% of the object space), 100 (1% of the object space), 500 (5% of the object space), 1,000 (10% of the object space), and 2,000 (20% of the object space). File *ndn-project-1.cc* already contains the code to perform these experiments. You will need to comment the line of code for no cache, and uncomment the line that enables caching for size 50, and so on for the other cache sizes in the experiment.

B1. The IP network experiment is the baseline scenario. We shall use hop count as the basic metric by which we indirectly measure bandwidth consumption. Compare and report on the total number of hops saved as we increase the caches sizes. Also report on the number of requests satisfied from cache hits vs. those satisfied from the owning repository nodes. Is there a consistent, significant, monotonic pattern of savings in terms of reduced number of hops? Do you think this trend would continue if we were to further increase the cache size, or would it level off beyond a certain point? Do you think that your results have any correlation with the "popularity profile" defined by the *ndn-consumer-zipf-madelbrot application* (which has been set up with parameter values $q = 0.7$ & $s = 1.5$ in the *ndn-consumer-zipf-mandelbrot-39.cc* file)?

Include tables and figures (graphs, bar charts, etc.) to substantiate your answers and observations.

B2. The *CSTracer* in the file *ndn-project-1.cc* keeps track of cache hits and cache misses. Report, using figures, the cache hits and misses for each node. Do the core nodes have more cache hits compared to the edge nodes? Do you think it would make sense to have larger cache sizes in the core nodes, and smaller ones at the edge; or vice versa? Submit modified versions of the *ndn-project-1.cc* file that implement such configuration choices.

Experiment Set 2:

Goal: To observe the effect, in terms of bandwidth savings, of the interplay between in-network caching and PIT aggregation.

(a) To observe the effect of PIT aggregation, we will need to run experiments without caches, using file *ndn-project-2.cc*. All hops saved can thus be attributed to the effects of PIT aggregation.

The version of *ndn-project-2.cc* that you download is configured for no caches and light traffic. You will need to run a series of experiments in which you gradually increase the traffic load to see the effects of PIT aggregation on hop-savings. In general, the heavier the traffic, the more the PIT aggregation effect. However, you might find that you have to increase traffic quite considerably before ~~significant~~ hop savings from this PIT aggregation effect ~~are~~ might be observed.

For Experiment Set 2 (related to items **C1.** and **C2.** below), we have identified a problem. Unfortunately, the default configuration of ndnSIM does not account for hops saved by PIT aggregation. What this means is that when you run the experiments with heavy traffic there will be PIT aggregation taking place, but it will not show in the hop count column of the *AppDelayTracer* trace output file. We therefore had to modify the ndnSIM simulator code.

Please download the following six files (they are all updated versions of files that came with the ndnSIM installation):

1. *ndn-pit-entry-incoming-face.cc* and *ndn-pit-entry-incoming-face.h*.
2. *ndn-pit-entry.cc* and *ndn-pit-entry.h*.

These four files above need to be placed in folder *src/ndnSIM/ns-3/src/ndnSIM/model/pit*.

3. *ndn-fw-hop-count-tag.h*. This file needs to be placed in folder *src/ndnSIM/ns-3/src/ndnSIM/utils*.
4. *ndn-forwarding-strategy.cc*. This file needs to be placed in folder *src/ndnSIM/ns-3/src/ndnSIM/model/fw*. **But read the important warning below first.**

   WARNING: Note that at this point, you will have already modified the version of the file *ndn-forwarding-strategy.cc* you currently have, when you attached the packet tag in relation to item **A2.** The new *ndn-forwarding-strategy.cc* above is an updated version of the original file that came with the ndnSIM installation. It does not have the packet tag changes you made. Therefore, when you install this new *ndn-forwarding-strategy.cc* file, you will need to re-introduce into it the packet tag changes that you did for the version it replaces.

C1. Report on the effectiveness of PIT aggregation by comparing the baseline IP network to PIT aggregation with no caches over various traffic loads. Include tables and figures as appropriate. Also report on what traffic loads you used; how you adjusted the Interest packet generation times & the simulation end times for these loads, and so on.

(b) Now we introduce caching. You will need to run the same set of experiments as in (a) above, but with cache sizes of 50, 500 and 2,000, say. The *ndn-project-2.cc* file has the required code which needs to be uncommented to run with these cache sizes.

C2. Report on the results you obtained for the cache sizes you used. In particular:

1. part (b) of Experiment Set 1 gave you hop savings that can be attributed purely to the effect caching (under light traffic) with no PIT aggregation;
2. part (a) of Experiment Set 2 above evaluates hop savings achievable entirely due to the effect of PIT aggregation (no caching) under a range of increasing traffic load; and finally,
3. part (b) of Experiment Set 2 above, evaluates hop savings due to the combination of in-network caching and PIT aggregation, also under a range of increasing traffic load.

What can you conclude about the interplay of cache and PIT aggregation from these three sets of results when you compare them to each other (but bear in mind that your results from 1. above are not entirely compatible with those from 2. & 3.)? Please use tables and figures as appropriate to substantiate your answers. You might find the following paper of some help: Kazi & Badr 13, "Pit & Cache Dynamics in CCN".