CSE534 Project #2 Report **Sai Chaitanya Mallampati [109597269]**
5th May, 2014 **Paavan Kumar Sirigiri [109596437]**

Answer [A 1]

Prior to setting up the dumb bell network, we installed ndnSIM on our machine. We did the following in order to install ndnSIM -

1) To install the boost libraries we ran the following command -
sudo aptitude install libboost1.48-all-dev

2) To run the visualizer we ran the following commands -
sudo apt-get install python-dev python-pygraphviz python-kiwi
sudo apt-get install python-pygoocanvas python-gnome2
sudo apt-get install python-gnomedesktop python-rsvg ipython

3) To download the ndnSIM source code we ran the following commands -
mkdir ndnSIM
cd ndnSIM
git clone git://github.com/cawka/ns-3-dev-ndnSIM.git ns-3
git clone git://github.com/cawka/pybindgen.git pybindgen
git clone git://github.com/NDN-Routing/ndnSIM.git ns-3/src/ndnSIM

4) To compile and run ndnSIM we ran the following commands -
cd ns-3
./waf configure –enable-examples
./waf
./waf –run=ndn-grid –vis

Now, to set up the dumb bell network we did the following -

1) We created a network topology with two core router nodes attached by a point-to-point link. These nodes are named A and B.

2) To the core router node A, we hung two edge nodes. Each of these edge nodes is made a consumer node. These nodes are named Consumer1 and Consumer2.

3) To the core router node B, we hung two edge nodes. Each of these edge nodes is made a producer node. These nodes are named Producer1 and Producer2.

4) We used the *AnnotatedTopologyReader* to read the topology file describing the above topology.

5) After this, we configured the caches such that they are enabled at the core nodes and disabled at the end nodes. We also set the cache size for the core nodes as 3 objects.The cache policy is Least Recently Used. All this coding is done in the *SetContentStore* method.

6) Next, on each of the producer node, we defined the object space the producer caters to. This coding is done in the *SetPrefix* method.

7) Next, on each of the consumer node, we created two ConsumerCBR applications - one to request content from Producer1 and the other from Producer2. In Consumer1, the application is configured to request content from Producer1 starting at time 0.0 and to request content from Producer2 starting at time 1.0. Similarily in Consumer2, the application is configured to request content from Producer1 starting at time 1.0 and to request content from Producer2 starting at time 1.0. This

coding is done using the *StartSeq* parameter of the *SetAttribute* method.

8) After this, we set the generation rate for interest packets in each of the ConsumerCBR to 1 interest packet per second.

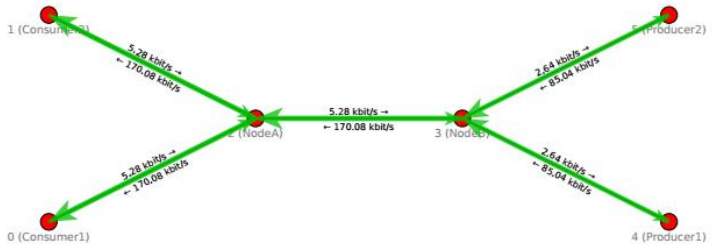
9) We made the forwarding strategy *BestRoute*.

10) Next, we made use of the *GlobalRoutingHelper* to populate the Forwarding Index Base(FIB).

11) We set the experiment to run for 5 seconds.

On doing this, the dumb bell network was running successfully.

The topology of the dumb bell network looks like below -



Answer [A 2]

To introduce the new packet tag, we made the following code modifications -

1) Added *ndncachehittag.cc* class. This class has a parameter named *cachehit* which is set to 1 when the object is found in the cache.

2) Modified the *appdelaytracer.cc* class. We added code to add an extra column in the output generated by the class. This column is labeled as *cachehit*. If the object is present in the cache, *cachehit* is shown as 1 and if the object is absent in the cache, *cachehit* is shown as 0.

3) Modified the *ondata* method in the *ndnconsumer* class. We added code to process the cache hit tags in the incoming packets.

4) Modified the *oninterest* method in the *ndnforwardingstrategy* class. We added code to create a cache hit packet tag and set it to 1 whenever the object is found in the content store.

For each of this class, we made the corresponding changes in the header files as well.

The sample output looks like below -

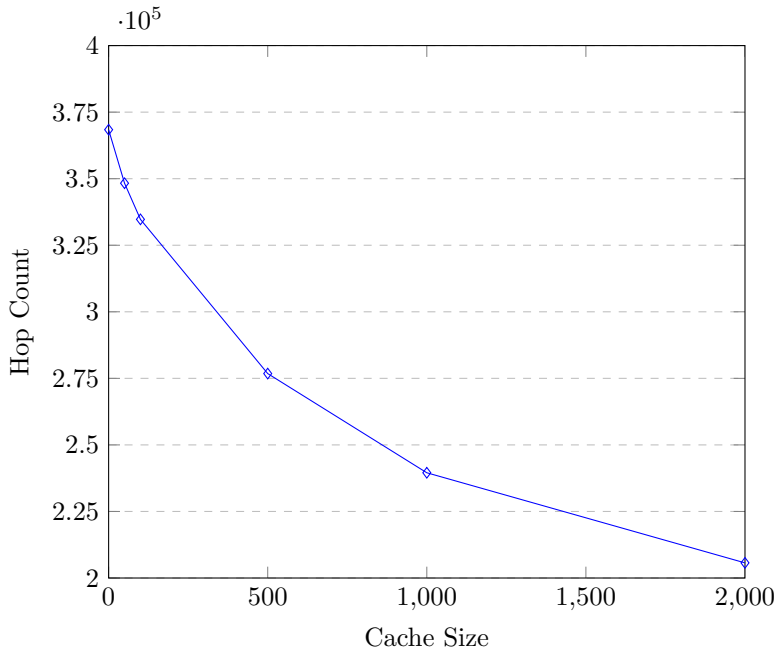
Answer [B 1]

We set up and ran the experiment with cache disabled as well as for various values of cache. Complete results of our experiments can be seen in the table below -

Cache Size	Interest Packets	Cache Hits	Hop Count	Cache Misses
No Cache	49358	0	368402	49358
50	49358	12448	348302	36910
100	49358	17765	334722	31593
500	49358	34096	276790	15260
1000	49358	39205	239534	10153
2000	49358	40137	205686	9221

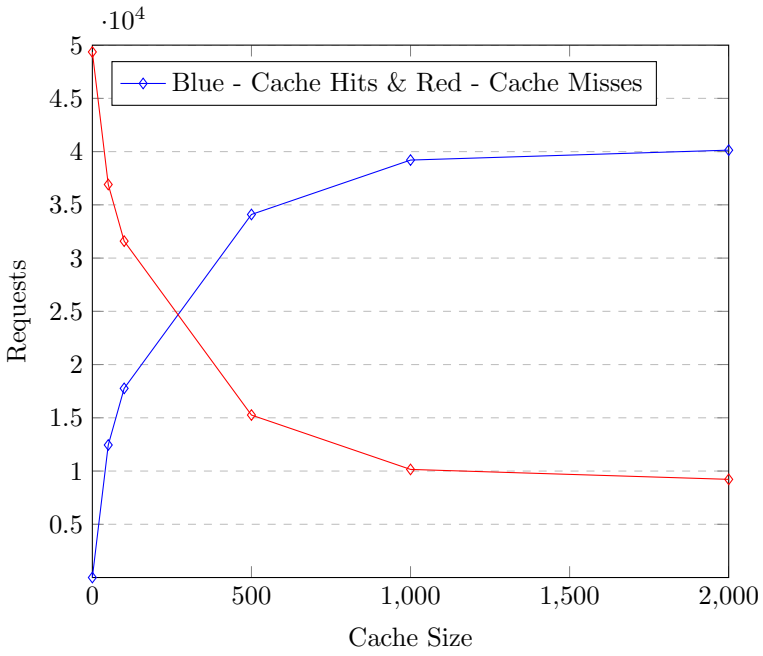
As we increased the cache size the total number of hops decreased.

Graphically looking at the variation of hop counts with the variation of cache size,



As the cache size increased from 0 to 50, hop count decreased from 368402 to 348302 [a total save of 20100 hops]
As the cache size increased from 50 to 100, hop count decreased from 348302 to 334722 [a total save of 13580 hops]
As the cache size increased from 100 to 500, hop count decreased from 334722 to 276790 [a total save of 57932 hops]
As the cache size increased from 500 to 1000, hop count decreased from 276790 to 239534 [a total save of 37256 hops]
As the cache size increased from 1000 to 2000, hop count decreased from 239534 to 205686 [a total save of 33848 hops]

Number of requests satisfied from cache hits vs those satisfied from the owning repository nodes - Initially when the cache is disabled, all the requests are satisfied from the owning repository nodes. As we kept increasing the cache size, the number of requests satisfed from cache hits increased and correspondingly the number of requests satisfied from the owning repository nodes decreased. The number of requests always remain constant. Graphically speaking,



In the following table, we look at the hops change per unit change in cache size.

Cache Size Change	Hops saved	Hops Change/Cache Change
0 to 50	20100	402
50 to 100	13580	271.6
100 to 500	57932	144.83
500 to 1000	37256	74.512
1000 to 2000	33848	33.848

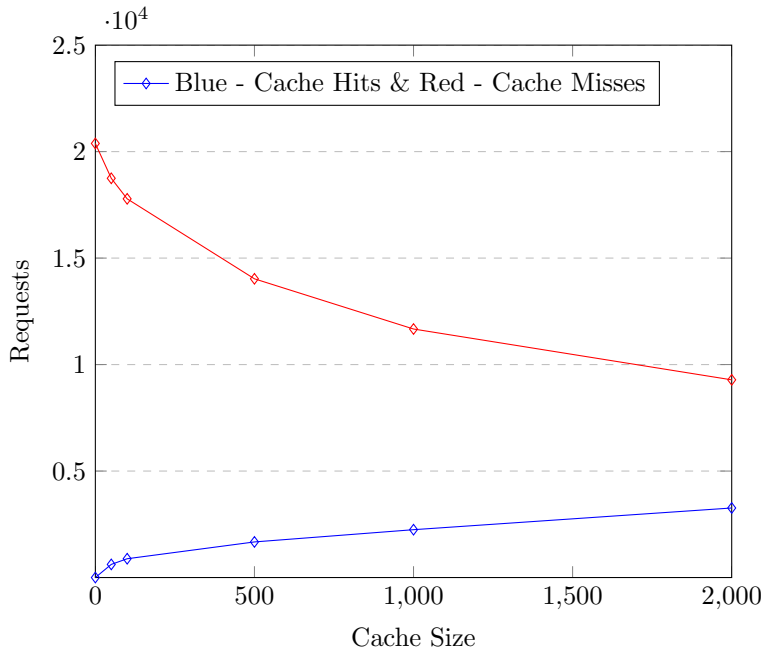
Is there a consistent, significant, monotonic pattern of savings in terms of reduced number of hops? As we can see from the table above, the hops changed per unit change of cache size was significant for lower frequecies. As we kept increasing the frequency, the hops saved per unit change of cache size kept decreasing. The change is neither consistent nor monotonic.

Do you think this trend will continue or level off beyond a certain point? From the above pattern we can say that the number of hops saved per unit change in cache size will keep decreasing until a certain point. After that, it will most likely level off.

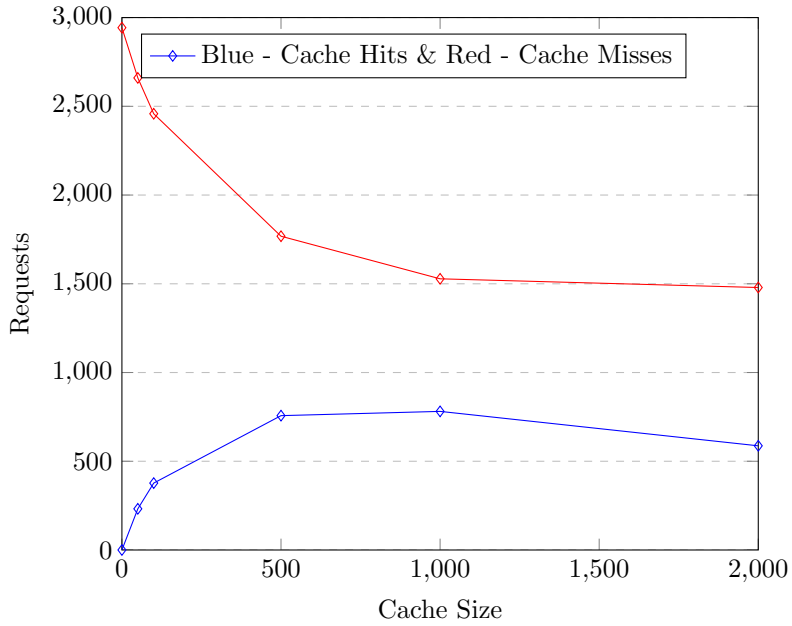
Do you think that your results have any correlation with the popularity profile? The popularity profile shows how often a particular object is requested and it does have a direct correlation with the results we obtained above. If the values of the parameters q and s are increased, the existing objects will be requested more often thus resulting in higher cache hits. If the values of the parameters q and s are decreased, the existing objects will be requested less often thus resulting in lower cache hits.

ANSWER [B 2]

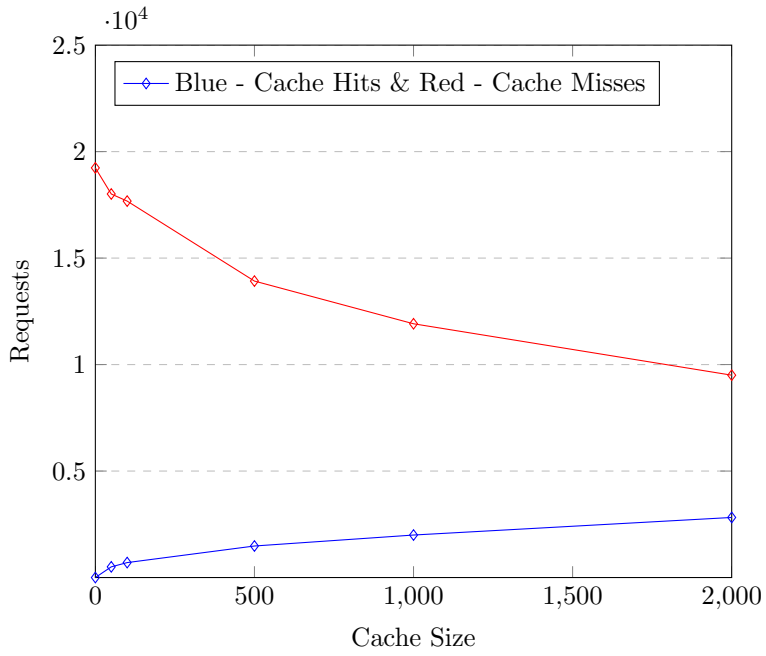
We used the CSTracer to find the number cache hits and cache misses for a core node (in the case of the graph below we considered node 0)



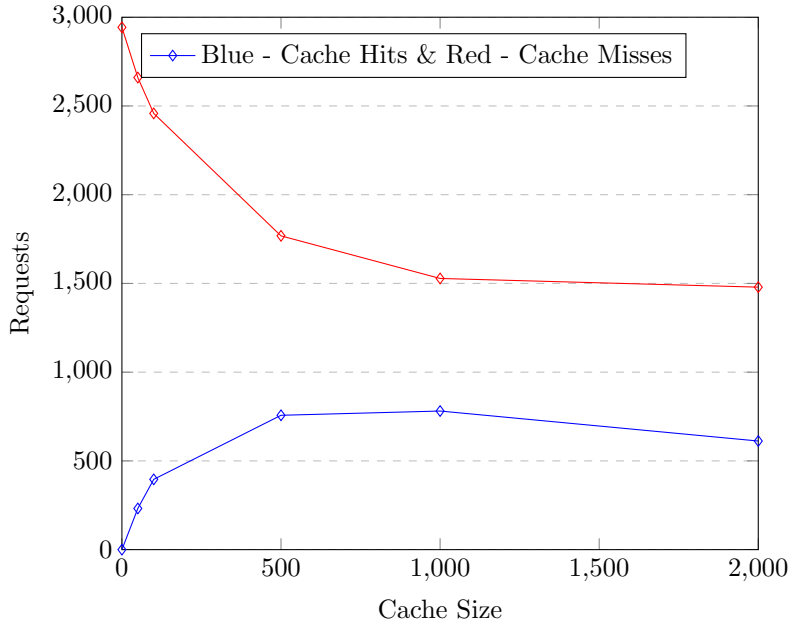
We also used the CSTracer to find out the number of cache hits and cache misses for a edge node (in the case of the graph below we considered node 8)



Just like above, we now plot the graphs for cache hits and cache misses on an average for core nodes



We now do the same for edge nodes



Observations :

1) In the case of core nodes, as the cache size increased the cache hits increased and the cache misses decreased.

2) In the case of edge nodes, as the cache size increased the cache hits increased until certain cache size but for cache size 2000, the cache hits actually decreased when compared to cache size 1000. The cache misses decreased just like in the case of core nodes.

3) As the cache size increased, the cache misses decreased at a steeper rate in the case of core nodes as compared to edge nodes. Similarly, the cache hits increased at a steeper rate in the case of core nodes as compared to edge nodes.

Do you think core nodes have more cache hits compared to the edge nodes? From the above graphs and observations, the core nodes do have more cache hits than edge nodes.

Do you think it would make sense to have larger cache sizes in core nodes and smaller ones at the edge or vice versa ? Yes, because from the graphs above we can say that smaller cache sizes in end nodes doesn't have much negative impact on the overall performance of the network. Also, larger cache size in the core nodes result in high cache hits and low cache misses which is good for the performance of a network.

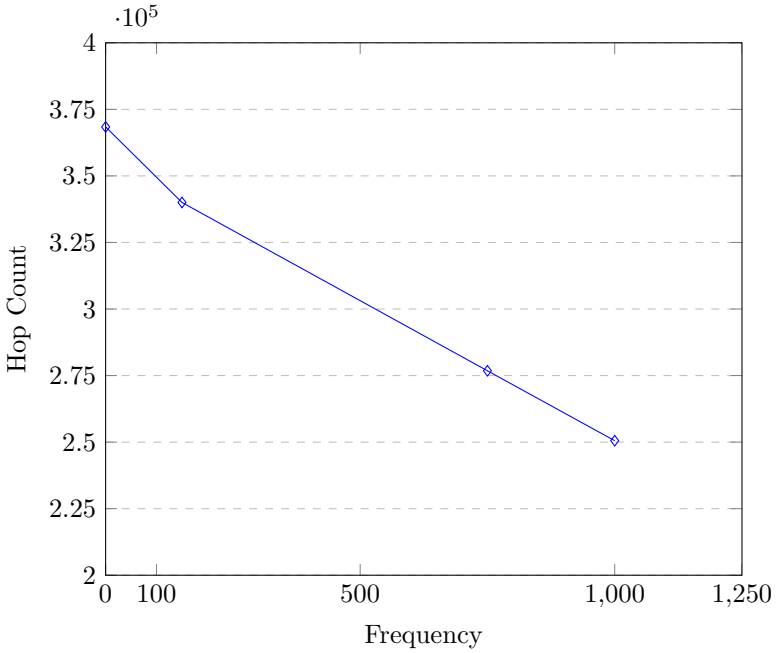
ANSWER [C 1]

Throughout the experiments on the effectiveness of PIT aggregation, we kept the number of interest packets as 50000 i.e., a constant. We kept changing the frequencies and correspondingly the simulation times changed in order to keep the number of interest packets.

We used the formula, $time = \frac{50000}{frequency*31}$

Frequency	**Simulation Times**
0.032	50000
150	10.75
750	2.15
1000	1.61

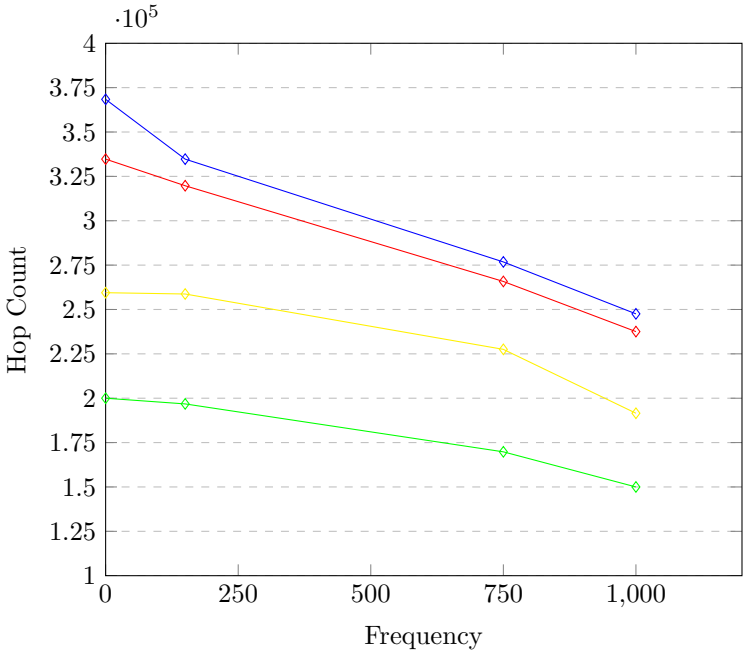
Plotting the graph for all of these frequencies against all number of hops -



From the graph above, we can see that the hop count decreases linearly as we kept increasing the frequency.

ANSWER [C 2]

Now, we analyze how the hop count changed with changing frequency and cache size. We used cache size of 0, 50, 500 and 2000. Also we used the same frequency set we used above. Graphically,



In the graph above, blue plot corresponds to 0 cache size. Red plot corresponds to 50 cache size. Yellow plot corresponds to 500 cache size and green plot corresponds to 2000 cache size.

We know that the plot of hop counts against cache size was exponentially decremental levelling off beyond certain point. This is the effect of cache size on the overall hop count.We also know that the plot of hops counts against frequency was almost decrementally linear. This is the affect of PIT aggregation on the overall hop count. The above graph depicts the cummulative affect of cache size and PIT aggregation. Although, cummulative affect is being measured the results aren't cummulative.

Another key observation is that increase in cache size at lower frequencies results in a large hop saves. This is clearly evident from the graph above.