TU Delft Delft University of Technology

Education/ET4285.Group32012

Generated: 08 Feb 2013 - 11:44

Table of Contents

Deployability of Content-Centric Networking

Group members

Member name	Email	Student number
Mani Prashanth Varma Manthena	M.P.V.Manthena@student_NO_SPAM.tudelft.nl	4243846
Michal Golinski	M.Golinski@student_NO_SPAM.tudelft.nl	4253590

Abstract

Most of the network traffic in the internet is being dominated by ever increasing content accessing and delivery services which are inducing large amounts of delays and bandwidth consumption. Further, the present internet architecture (TCP/IP) which revolves around a host - based communication model was not designed to support and effectively handle such high traffic flows. Content-Centric Networking (CCN) is a future internet architecture proposed to solve the above problems by naming the content itself instead of the content source and caching the content in the intermediate routers for effective content distribution and retrieval.

In this paper, we investigate the factors and parameters like content to cache, cache size, content caching under changing link conditions, etc. that must be taken into account while deploying CCN in the present internet. We also compare CCN with the present internet architecture (TCP/IP) under different real world scenarios. Different tests and measurements were performed on both CCNx (CCN based open source experimental project) and HTTP (TCP/IP based network communication protocol) under different scenarios by simulating real world networks in our measurement setups.

Our measurements showed various interesting results along with some loopholes in the present implementation of CCNx like poor caching strategy, unable to handle packet losses, and implementation flaws in MTU (Maximum Transmission Unit) size. For most of the real world scenarios, HTTP still has better performance compared to CCN but, CCN proved to outperform HTTP in the case of large cache sizes in the intermediate CCN routers and multiple users/hosts requesting the same content data (content caching). We also proposed some smart caching strategies and other implementation changes that are required in the present form of CCNx for large scale deployability of CCN architecture in the near future.

Introduction

TCP/IP (Transmission Control Protocol / Internet Protocol) is a current internet protocol suite which is also the most popular protocol suite to be used in computer networks till date. It is a host - based communication model involving conversation between only two machines which are content/information source and destination. It involves routing of IP packets which contain two identifiers that are source and destination addresses. Due to this host - based communication model, one has to get the content data each time from the content server even though nearby hosts may already have the same data. In other words, multiple users/hosts requesting same content data from a content server must obtain the data each time directly from the content server individually, which may lead to wastage of scarce and valuable resources like network bandwidth. Furthermore, the recent tremendous growth of content/data demand in the internet resulted in a number of issues like content access delays, bandwidth consumption, unreliable content access, content security issues, etc. Although TCP/IP has some inherent problems and issues, its design is still simple, robust and scalable compared to other network architectures.

Internet users today are mostly concerned about what content they get and not from where they get it. So, replacing named hosts with named content may solve the above communication problems in the present internet. Content-Centric Networking (CCN) is a proposed future internet architecture built on named data. CCN allows content to be cached at the intermediate routers. Unlike the present internet architecture, multiple users requesting the same content data may obtain the data from nearby routers where the requested content is cached instead of obtaining it each time from the content server individually. This may result in the reduction of bandwidth consumption and content access delays. and may result in improving overall network performance. Cached content at the router is duplicated and sent to all the users requesting that content. Content is requested by the users and the intermediate routers based on content name and not on the content location as in IP. Content requests are sent on all out going faces i.e. broadcasting by the intermediate routers instead of sending them only on a minimum spanning tree as in TCP/IP.

Our main aim of the project is to study deployability considerations like content to cache, cache size, content caching under changing link conditions, etc. of CCN and to compare it with the present TCP/IP architecture under various simulated real world scenarios.

Related work is presented in the following section which is then followed by a brief introduction to CCNx (CCN based open source experimental project) [2]. Our measurement scenarios are discussed and diagrammatic representations of different measurement setups and scenarios are also included in the later section which is then followed by our measurement results where we present our results and explain them. Finally, we conclude the report by presenting our main conclusions with our proposed future work, references, tabular forms of measurement results, example measurement setup scripts and important fragments of python script which we wrote for measuring various parameters like mean packet delay, standard deviation of packet delay (jitter), bandwidth, download time, etc. of the downloaded content data packets from the content server which were captured by wireshark (packet sniffer tool) at intermediate routers/nodes in the measurement setups.

Related work

There are lots of proposed models on cache size implementation and memory management of CCN. Each of them suggesting optimum caching strategeis and cache size that can be used in the intermediate CCN routers based on various aspects like memory blocks price, efficiency of memory storage and buffering, amount of traffic in the network, etc.

Further, There had been lot of research done on performance comparison of CCN with other network architectures like TCP/IP. Data transfer efficiencies of CCN, HTTP and HTTPS are compared in *Van Jacobson et al*. [1] for an experimental measurement setup on a campus Ethernet. The results showed that CCN matches the performance of unsecured HTTP and substantially outperforms secure HTTPS. Further, the content transfer via CCN is always secure. Several real time services and applications like live streaming, video and audio distribution, etc. are also implemented in the CCNx framework and their performance had been compared with their counterparts in other network architectures like HTTP in TCP/IP.

All of the above approaches more or less sum up the CCN extent of deployability. So, in this paper we tried to study the deployability of CCN by considering its design parameters like what content to cache, cache size, content caching under changing link conditions, etc. and we also compared the performance of CCN with the present internet architecture (TCP/IP) under different simulated real world scenarios.

The Project CCNx

CCNx (CCN based open source experimental project) was implemented in our project by installing it on Linux machines which are then used in our measurement setups and scenarios as CCN routers/nodes. The full software protocol suite and documentation of CCNx can be found and downloaded at the CCNx [2] and Git Hub [3] websites. Project CCNx is sponsored by the Palo Alto Research Center (PARC) and is based upon the PARC Content-Centric Networking (CCN) architecture. Project CCNx is still in its early development stage and only an architectural backbone is provided for researchers and students to perform some experiments on it and to develop new applications using the backbone. Project CCNx was designed in such a way that it can be used as an overlay network over existing network architectures like TCP/IP.

We hereby present only the outline of the CCNx - CCN architecture. A full explanation of the architecture can be found in *Van Jacobson et al*. [1].

- Firstly, most of the layers in CCN and IP Protocol stacks are similar. Layer 3 i.e. network layer in IP has a power of simplicity due to weak demands it makes on the layer 2 and thus this layer is shown (as a thin waist in the IP protocol stack) in Figure 1. There is a similar layer 3 i.e. network layer in CCN protocol stack which is represented by content chunks. The CCN protocol stack has two layers that are different from the corresponding layers in the IP protocol stack. These layers are called strategy and security layers. Strategy layer makes optimal choices between multiple links/connections in case of changing link conditions. Thus CCN has an advantage of supporting multiple links and can make full use of them for optimal network performance. Security layer provides security to content rather than the connections through which the content travels. Further, in CCN all content is authenticated with digital signatures and private content is protected with encryption. Thus it avoids many host based security vulnerabilities as in IP networking.

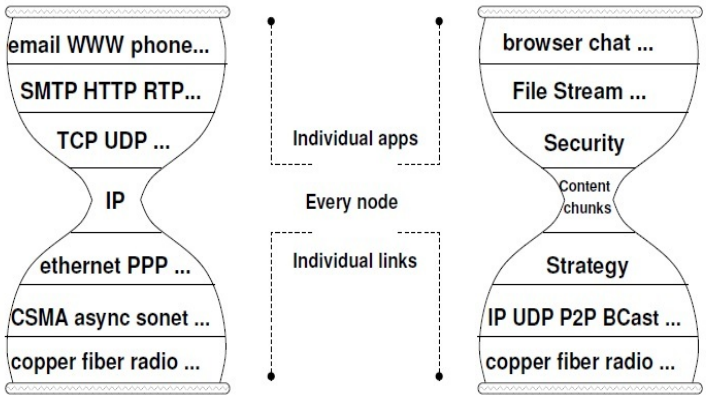


Figure 1: IP and CCN Protocol Stack [1]

- CCN names the content instead of the host or content source. CCN has two packet types, Interest and Data. A user requests a particular content data by sending CCN interest packets over all his outgoing faces i.e. broadcasting. Any node/router which gets the interest packet and has the requested content data by longest - match look-up of the content name in the interest packet responds to the interest by sending a data packet to the user. Data packet consumes the corresponding interest packet at a given node/router. Interest and data packets are similar to data and ack packets in TCP/IP.
- CCN has three main data structures: FIB (Forwarding Information Base), Content Store (Buffer Memory) and PIT (Pending Interest Table). FIB is used to forward interest packets to a list of outgoing faces instead of single face as in IP FIB. Content Store is used for content caching and stores the data packets as long as possible. So, that multiple users requesting the same content may benefit from the stored/cached data and may result in decreasing bandwidth consumption and content access delay. PIT keeps track of all the pending interests and upstream forwarded interests. CCN interests

which are forwarded upstream leave 'bread crumbs' or trails at each node/router they pass through, so that the corresponding data packet can go downstream to the original requester(s). PIT uses soft state model i.e. interests that don't have any matching data after a certain time are timed out and are removed from the PIT entries list.

- When an interest packet arrives on one of the faces of a node/router, a longest - match look-up is done on its content name and the index structure used for look-up is done in such a way that Content Store match will be preferred over a PIT match which in turn will be preferred over a FIB match. Figure 2 shows the CCN forwarding engine model.

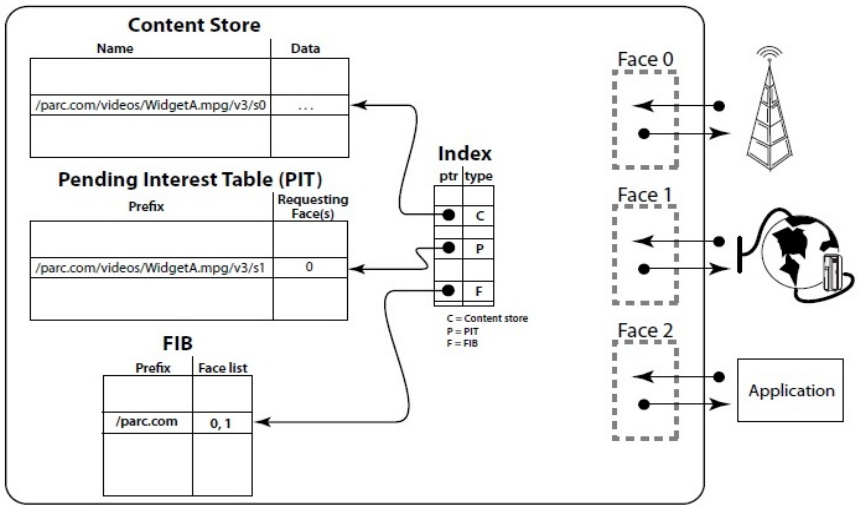


Figure 2: CCN Forwarding Engine Model [1]

- For reliable and loss-less content delivery, CCN interests which are timed out and removed due to non available matching data must be re-transmitted by the application which originated the initial interests, if it still wants the requested data. Data packets cannot loop but CCN interest can loop. In order to detect, discard and prevent duplicate interest packets each interest packet has a random nonce value. Further, CCN flow balance is maintained at each hop in order to control congestion in the network.
- Content names are hierarchically structured, so that a individual name is composed of several components. For convenience, names are represented as URIs with "/" character separating any two components.
- The CCN forwarding model is a strict super-set of the IP model with less restrictions on multi - source and multi - destination scenarios. So, any routing algorithm or scheme (both intra - domain and inter - domain) that performs well in IP should also perform well in CCN.

Project CCNx allows users to change several CCN protocol parameters like cache size, MTU (Maximum Transmission Unit), etc. in the protocol open source configure file. We used this feature of CCNx to perform different types of tests and measurements by changing the above parameters under different real world scenarios.

Measurement scenario

Our measurement setup involves Linux machines with CCNx installed. They perform operations of content server, intermediate CCN routers and clients in the measurement setup. Measurements are taken for four different setups as shown below in Figures 3,4,5,6.

Bad links between the two intermediate CCN routers in measurement setups - B and D have been generated by delaying the packets on the outgoing links by 100ms, 10ms jitter and 200ms, 20ms jitter (in total three scenarios including 0ms, 0ms jitter delay scenarios i.e. no delay scenario) and an other scenario of dropping packets (~ 20% of the total packets on the outgoing links) for which we used Linux netem commands [4] which provide network emulation by simulating the properties of a wide area network. Netem simulates variable delay, loss, duplication and reordering.

Different measurements are carried out in each measurement setup by varying the intermediate CCN routers cache size as 50000 (default), 20000, 10000, 5000 and 10 (Minimum) content objects and the same set of measurements are carried out by varying the MTU (Maximum Transmission Unit) size as 150, 300, 750, 1500, 3000 bytes.

A large file (50 MB) has been uploaded into the content server repository and is being downloaded at the other end by the clients. In case of measurement setups - C and D, the client 2 downloads the same file which client 1 has downloaded earlier. Such measurements are performed in order to analyze the caching properties of CCN. Example measurement setup scripts are included below (last section).

Wireshark - a packet sniffer tool [5] installed on the Intermediate CCN routers displays and captures the network packets. We wrote a python script to analyze the network packets captured by wireshark which would compute and plot mean packet delays, jitter, bandwidth, etc. We included the python code below (last section).

We also performed the same set of measurements by setting up an HTTPServer using Python simple HTTPServer [6] and wget to download the files at the client side from the content server. This is done in order to compare CCN and HTTP functionality under various real world scenarios.

Measurement Setup - A

Figure 3: Measurement setup - A

Measurement Setup - B

Figure 4: Measurement Setup - B

Measurement Setup - C

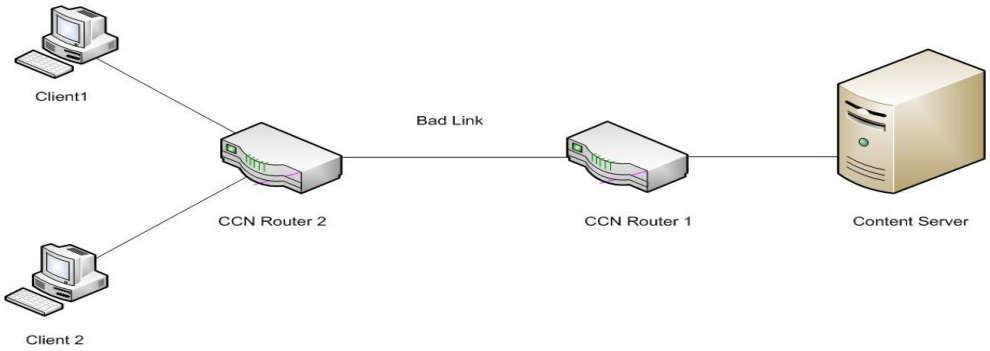


Figure 5: Measurement setup - C

Measurement Setup - D

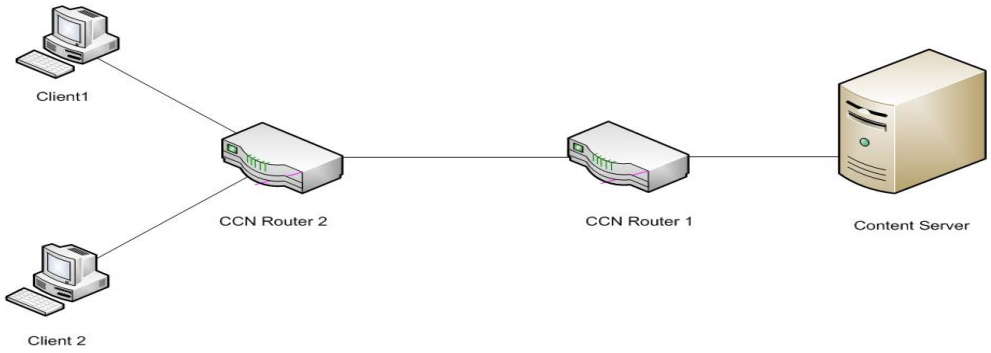


Figure 6: Measurement setup - D

Based on our scenarios, we expected that larger cache sizes would decrease overall bandwidth, delay, jitter, etc. in the case of measurement setups C and D due to local caching of the content first downloaded by the client 1. We also expected that CCN would out perform HTTP in that case (i.e. measurement setups - C and D with larger cache sizes) and it would approximately match the performance of HTTP in other measurement scenarios and setups. We also expected that changing MTU (Maximum Transmission Unit) size could fix the problem of decrease of bandwidth, because with higher MTU size, more interests can be sent (interest stuffing is allowed within MTU limit) and less packets traverse the network. Finally, we expected that CCN could handle packet losses like TCP/IP and support unreliable content data transmissions as in TCP/IP.

Measurement results

During the measurements a file of size 52MB was being downloaded from the network through CCN (with different internal settings and scenarios) and HTTP. At the beginning of each CCN measaurement, caches are full, but all the content stored in a cache is marked stall (it is treated as if the content is not cached). Further, if a chunk (group of content objects) in a node/router that is marked stall is requested, the request will be forwarded further to a node, where this content is not marked as stall.

Influence of cache size on delay

The first parameter we wanted to observe is the influence of cache size on delay and jitter in the network. Larger cache size could increase delay due to longer look-up time. However, we did not find that correlation. For different cache sizes and first download of a given content the delay is the same for all the cache sizes. This result is expected, since larger cache size (resulting in possibly large filled cache) should not increase delay between intermediate nodes. For a scenario where the same content is downloaded for the second time there is no change in delay up to a certain size of the cache. Above this threshold, mean delay is considerably decreased and equals to the one from the scenario without any introduced delay. It means that with the increased size of the cache bad quality link is being bypassed. This result is explained in the next section "Influence of delay on a bandwidth". Results are depicted on Figure 7.

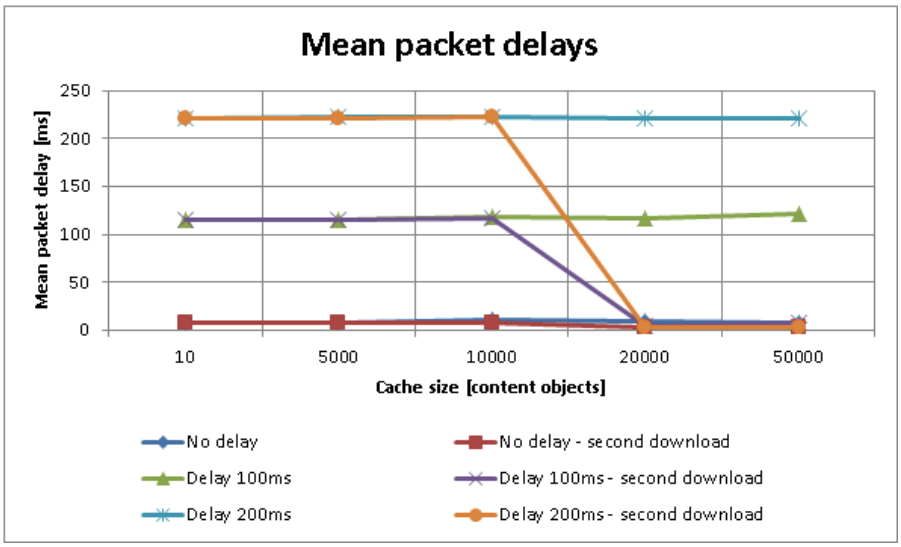


Figure 7: Mean packet delays

With the use of caching, jitter in such a network could be reduced. In a measurement scenario, with bad quality links induced first with 100ms delay, 10ms jitter and then with 200ms delay, 20ms jitter. Due to caching of the content the access to it could be given not from a server that is behind bad quality link, but from a locally stored copy. Such a behavior could reduce jitter. However, as it can be seen in Figure 8, there is no correlation between jitter and cache size. Using cache to reduce jitter could be implemented as an extension to the current CCNx implementation (for example through token buckets at the input of the CCN faces i.e. input control).

There are some interesting results depicted on Figure 8. For the delay 100ms and cache size 50000, jitter of the incoming packets is dramatically increased. However, this is not the case for delay of 200ms. It may suggest that this measurement is either wrong or was taken during increased activity of a machine processor used in the measurement setups (current CCNx implementation is very dependent on the hardware used). Moreover, for the second downloads, for 100ms case jitter increases while for 200ms jitter decreases. These changes are however minor (only few ms) and may be of random nature.

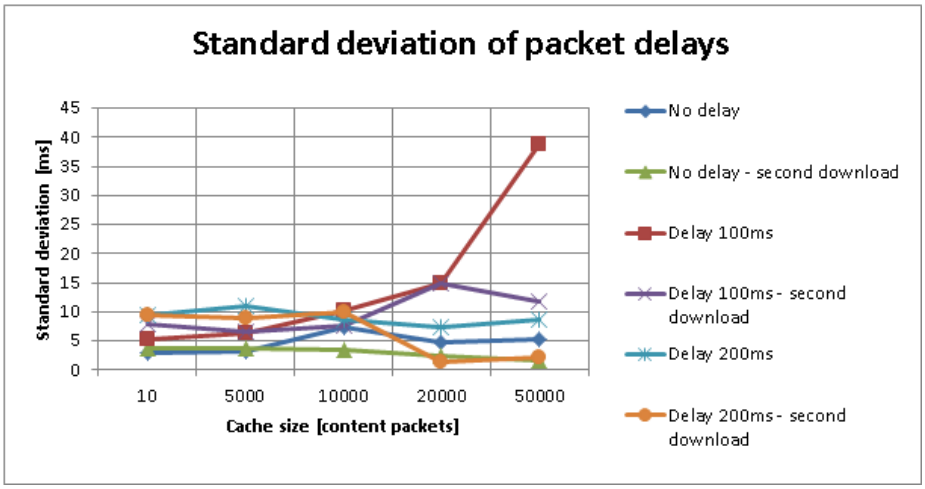


Figure 8: Standard deviation of packet delays

Influence of delay on a bandwidth

One of the conducted tests and measurements investigated the influence of delay introduced on a bad quality link to the bandwidth of overall connection (file download). The expected result was no or small influence of link delay on bandwidth. The reason for that would be, even though the packets are delayed, they are transmitted with the same "speed" over the link, so the download time should be the same (even though download in itself will be delayed by the value of bad quality link delay). On the contrary, measurements showed drastic influence of increased delay on overall available bandwidth. Compared to HTTP download (with the usage of TCP protocol) bandwidth of CCN (over UDP) is much lower (Figure 9) and it lowers down more with the introduction of delay (Figure 10).

Figure 9 shows the comparison between HTTP download and a few chosen CCN downloads (which took place one after another, so that in a second case, marked by "second download", we can make use of the caching property of CCN). The CCN scenario presented in Figure 9 is a scenario with cache sizes set to 10, 10000 and 20000 content objects. After careful analysis of all the conducted experiments (Figure 11), we noticed that for a given 52MB file only cache size above 20000 is sufficient to benefit from caching (in case of lower cache sizes whole file is not stored locally and as an effect, whole copy of the file is downloaded again from the server - as described in subsection "Smart caching strategies"). From the Figure 9 it can be seen that bandwidth of HTTP download is higher than CCN download (to better depict the big difference, bandwidth has been depicted using logarithmic scale). For cache sizes of 10 and 10000, for the second download bandwidth is comparable to the first download case. However, positive effect of caching is noticeable for 20000 cache size, since for this case a cache that is placed before bad quality link is used to retreive the file, so that bad quality link is not used at all.

The reason of bad performance of CCN compared to HTTP has to be investigated further. The main reason noticed so far is its dependence on hardware configuration. Different tests conducted using different hardware configurations showed big differences in CCN performance. The reason for that may be there is a lot of logic implemented on top of UDP protocol, that was used during the test, that has to be performed before packet transmission.

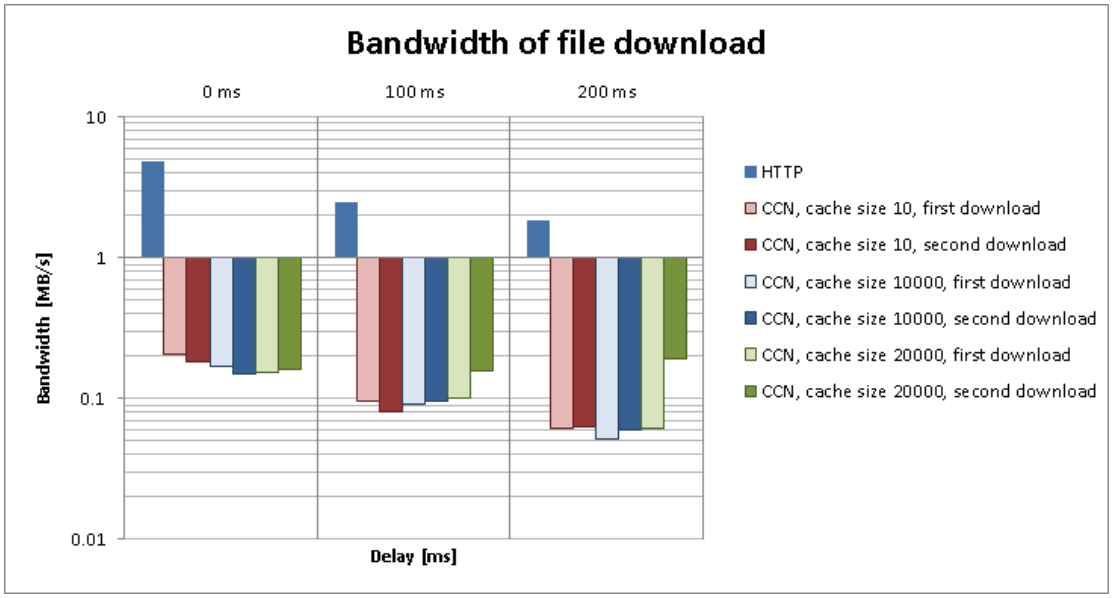


Figure 9: Bandwidths of file download

What is interesting to investigate is how much the service deteriorates. Figure 10 presents relative bandwidth loss for three values of delay on a bad quality link. We defined relative bandwidth loss as the highest bandwidth in a given category divided by actual bandwidth. For example, the highest bandwidth for CCN was achieved for the following parameters: delay 0ms, cache size 10, first download. The relative bandwidth loss of this value is 1. The relative bandwidth loss of value *x* means that for a given delay the achieved bandwidth is *x* times lower than the maximum bandwidth achieved during our measurements in a given technology (HTTP or CCN). Introduction of such a relative parameter allows us to compare how much HTTP and CCN deteriorates when we introduce delay. Provided that we have hardware that is fast enough to perform CCN operations as fast as HTTP download, it should be possible to achieve similar factors of loss for bad quality links. What can be noticed from the graph is usually with the introduction of delay on a bad quality links, CCN suffers higher deterioration of a service than HTTP. However, for a sufficiently large cache size CCN may outperform HTTP - for a cache sizes of 20000 and 50000 and second downloads (when the content is cached) the relative loss is smaller than that of HTTP.

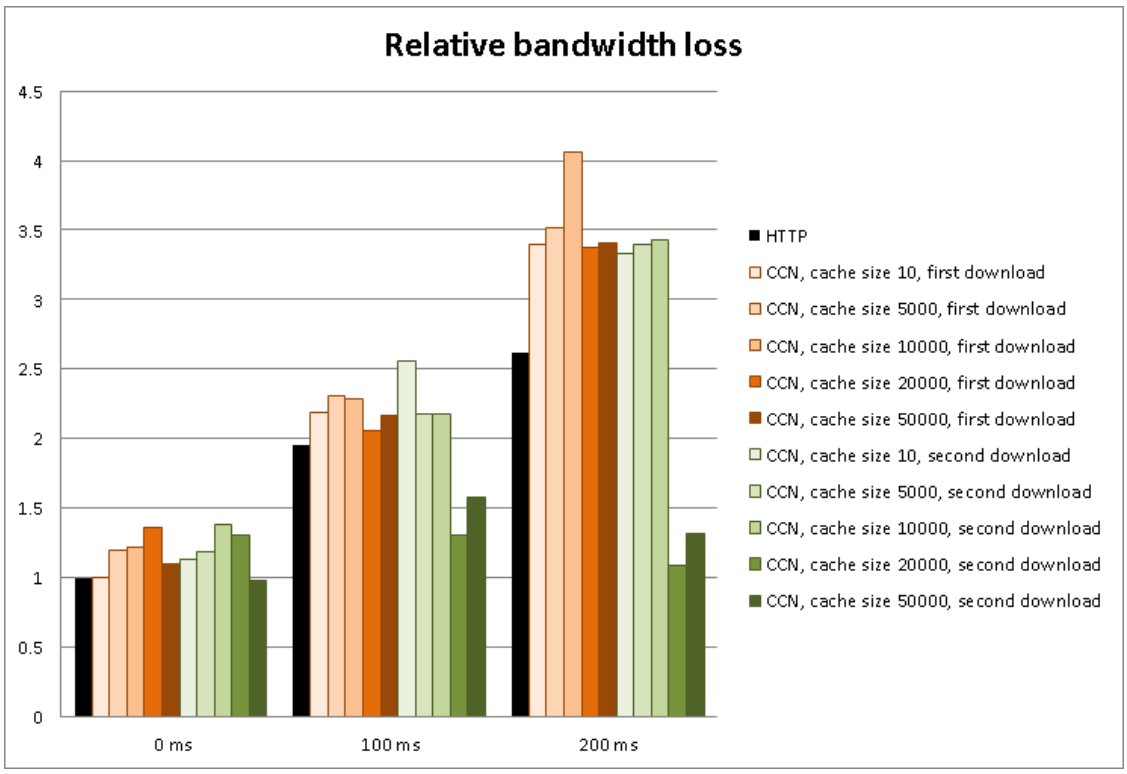


Figure 10: Relative bandwidth loss

Figure 11 shows the summary of all taken measurements. Some interesting conclusions may be drawn from this graph. What can be noticed is that for a small value of cache size second download takes longer than the first one. It may be connected with a longer look-up time in the local cache, because some content is already cached - unfortunately it does not match requested content. For sufficiently large cache size (cache size 20000 and larger) we notice improvement of the bandwidth between first and second downloads.

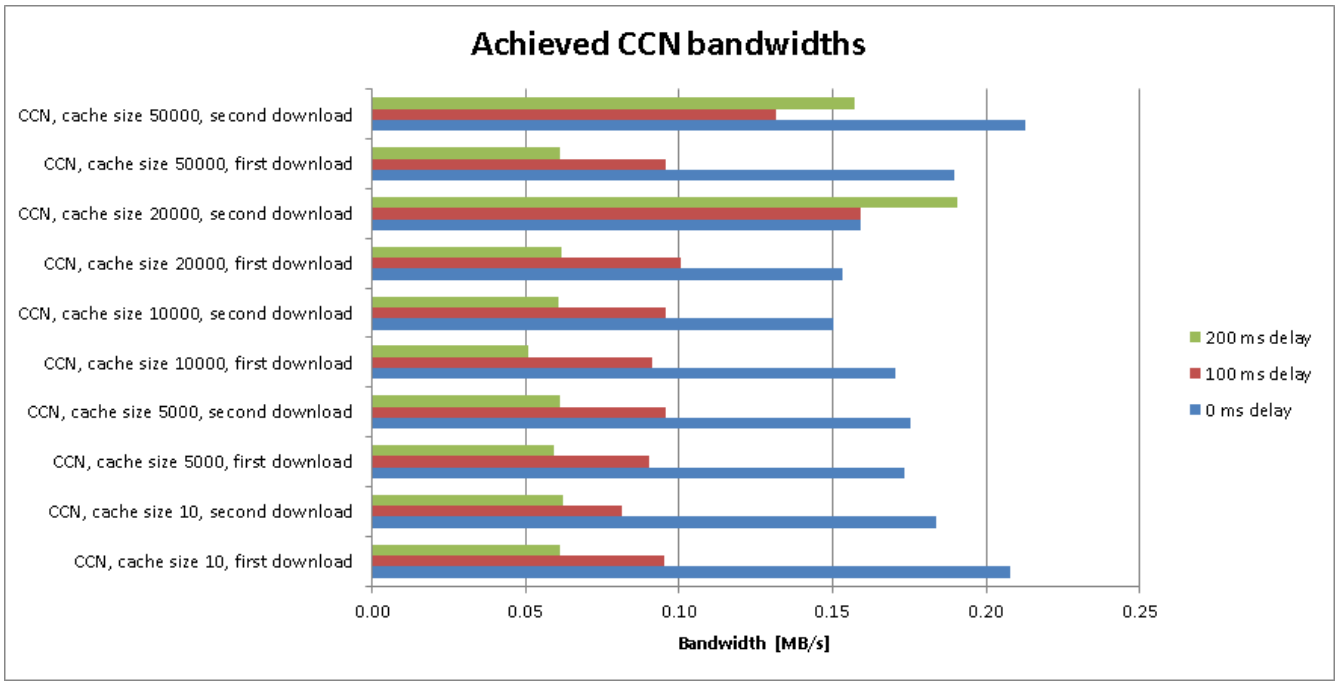


Figure 11: Achieved CCN bandwidths

MTU size and download time

Changing MTU (Maximum Transmission Unit) size could fix the problem of decrease of bandwidth, because with higher MTU size, more interests can be sent (interest stuffing is allowed within MTU limit) and less packets traverse the network. However, CCNx seems not to react on change of CCND_MTU parameter responsible for MTU size. It may be due to problems with CCNx implementation. We did not notice meaningful influence of this parameter change to the performance of the network, as it is shown in Figure 12.

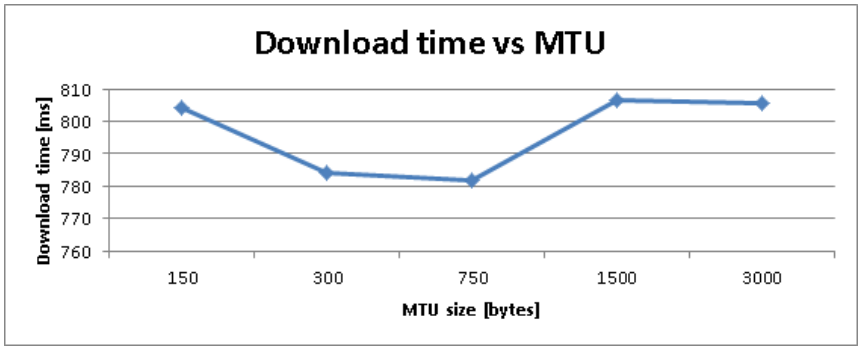
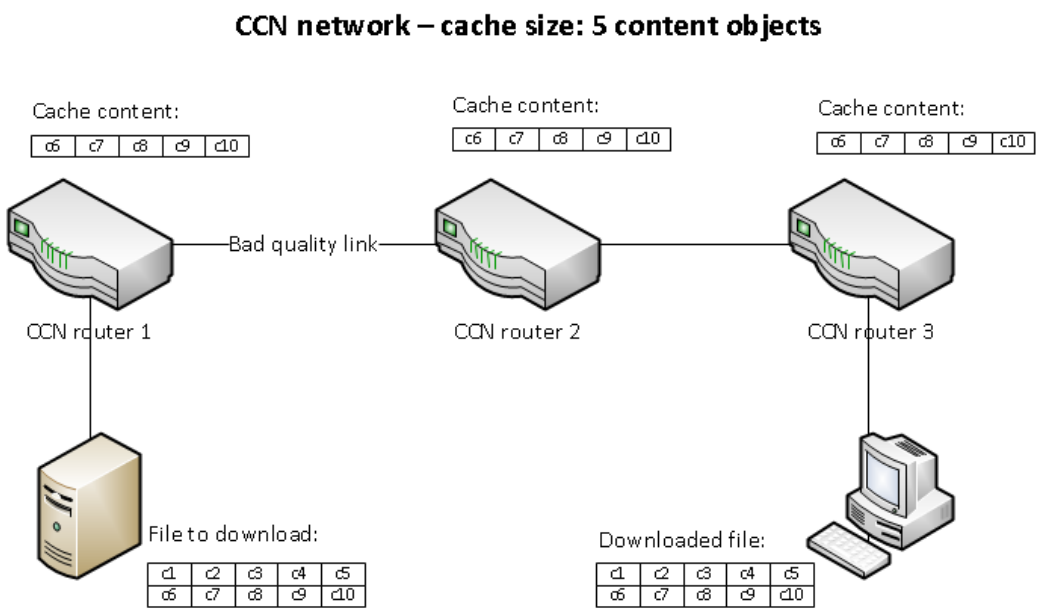


Figure 12: 200 ms delay on a bad quality link, cache size 50000

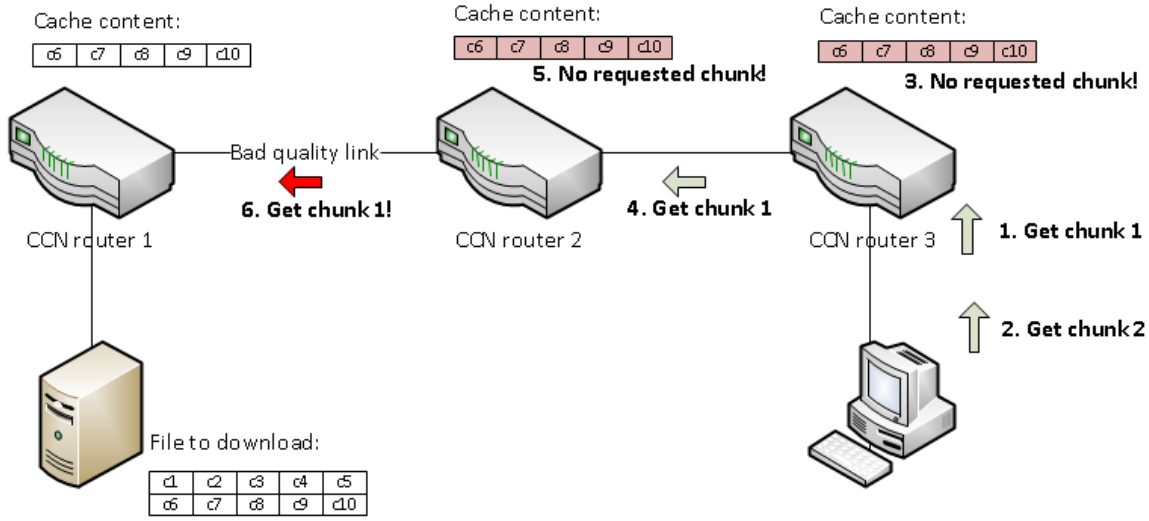
Smart caching strategies

One of the behaviors to investigate was caching strategy in case of presence of bad quality links. We considered a simple scenario of content caching in a line network, as presented in Figure 13. In the current

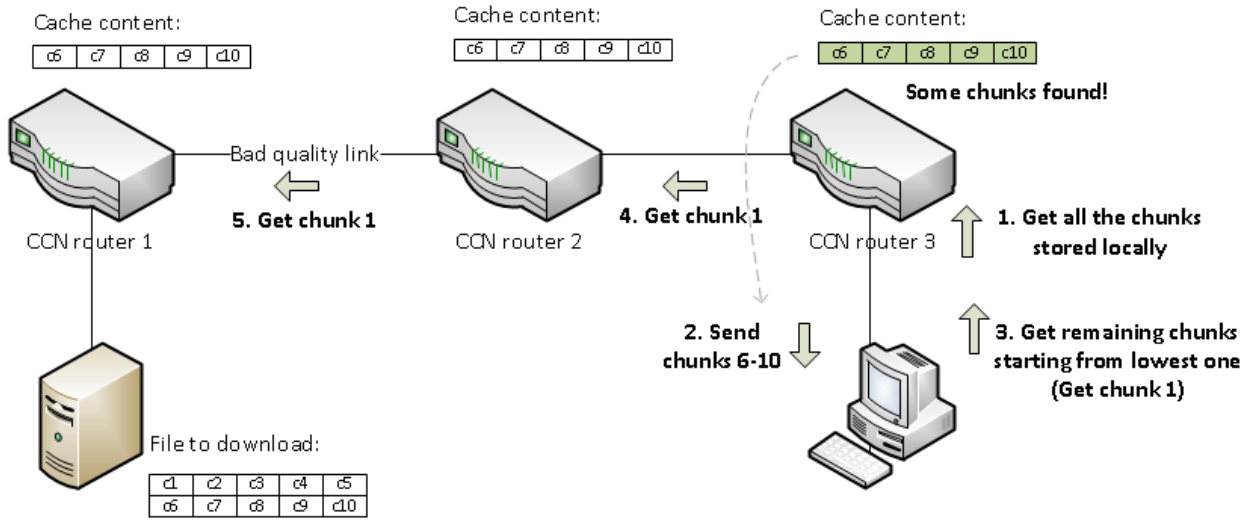
implementation, CCNx cache works like a FIFO queue. Let the cache size be *k* content objects and a file can be divided into *n* content objects. For a linear topology, if *k*>=*n*, we benefit from caching as content will be downloaded from the closest node/router. If, on the other hand i.e. *k*<*n*, in the current implementation we do not benefit from caching at all. The reason for this is that in such a case, content objects *n-k+1* to *n* (chunk 2) will be cached at all the nodes in our topology (Figure 13a). When next user tries to download the file, content objects *1 to n-k* (chunk 1)will be requested, thus the requests will propagate throughout the whole network (Figure 13b). If, in the case of subsequent requests, all the content objects residing on the closest nodes would be send (content objects n-k+1 to n i.e. chunk 2), and afterwards only content objects *1* to *n-k* (chunk 1) would be requested from the content provider, the idea of caching would be utilized in a better way (Figure 13c). Moreover, if we would use intelligent caching, that would cache different content objects at different nodes, then in the case of *k*<*n* we could still bypass the bad quality link provided we are sufficiently far away from it, so that all of the content objects required are cached at the nodes before bad quality link.



a) State of the network after file download



b) Start of second download – measured case



c) Start of second download – expected case

Figure 13: Smart caching example and measured scenario.

Behaviour described above would result in two groups of packets appearing while the file is being downloaded - low delay packets with chunks stored locally and high delay packets with chunks downloaded through bad quality link (Figure 14). With the increase of cache size, the size of first group of packets (low delay packets) would increase as more chunks would be downloaded from a locally cached copy.

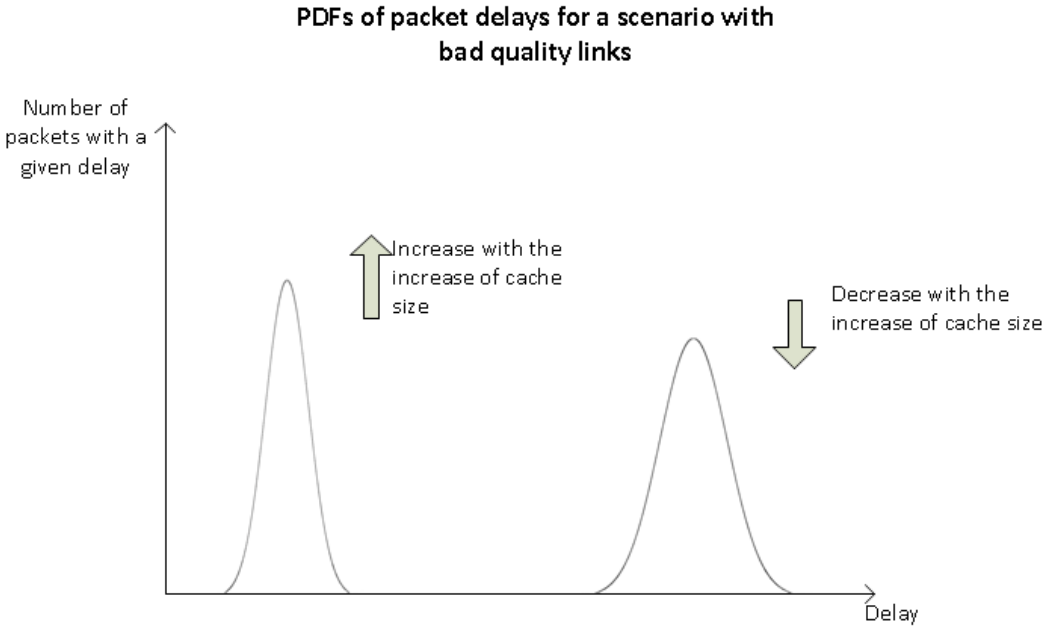
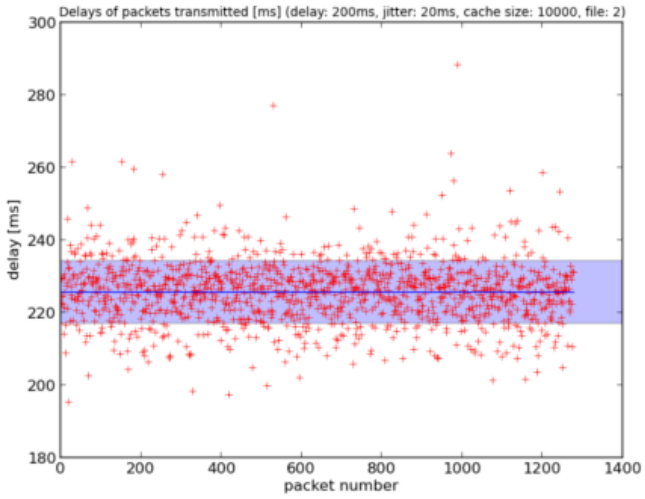


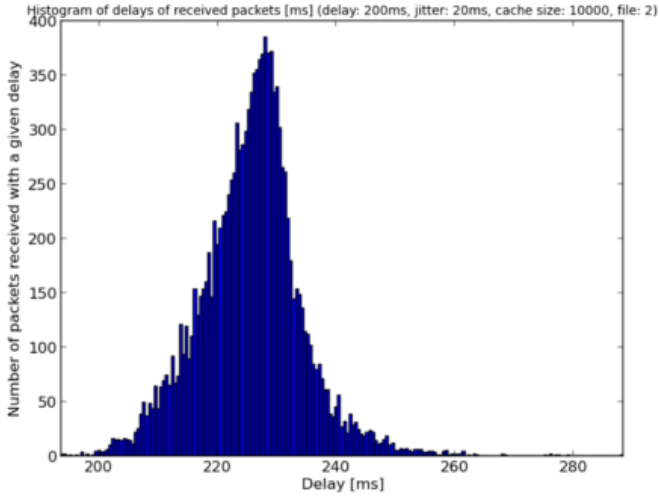
Figure 14: Packet delays in a scenario of smart caching

To implement smart caching, control protocol should be implemented, that will investigate, what chunks are cached at which nodes/routers. One of the principles should be that the same chunks should not be stored at the neighboring nodes. In the case of k<n and a line network, such a smart caching protocol should be able to detect bad quality links. If link quality is good enough (there is no situation of a bad quality link in a close nieghbourhood), smart caching strategy can be employed. A node, which is receiving content objects from the source should inform following nodes, if the contents that are being forwarded have been cached in that node. If they have been cached, the next node will not cache it. If they had not been cached (for instance because link quality is bad or cache is full), then the next node will cache the content and mark it cached while forwarding further. While downloading the file, we should be able to request **any** of the chunks in any given order, and then assemble it after we have all the chunks instead of requesting chunks in a given order as implemented now.

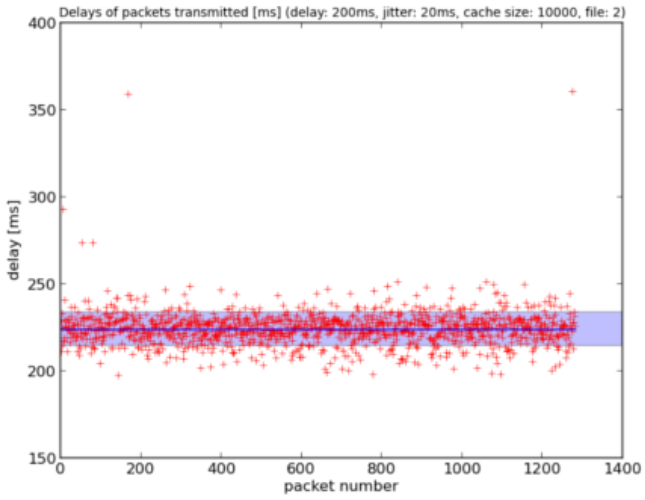
Measured scenarios have showed simple behavior of caching algorithms. They do not take into consideration a situation described above. So, in a scenario where not a whole file is cached locally, the whole file will be downloaded again. Both behaviors can be seen below:

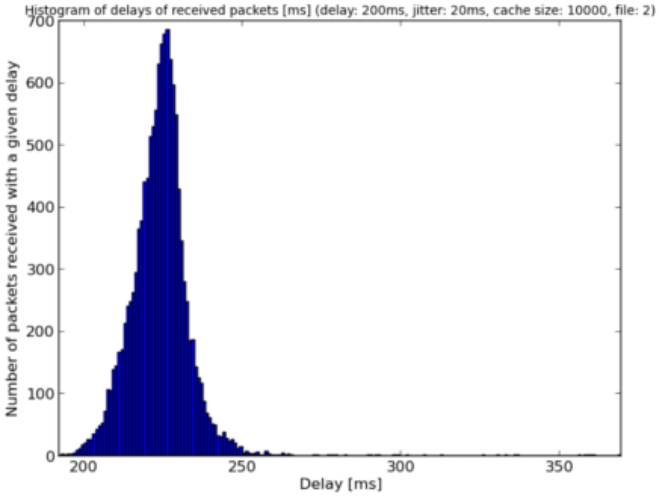
- Cache size 10000 (Figure 15) - First and second download do not differ in mean delay and jitter.





a) 200ms delay, cache size 10000, first download

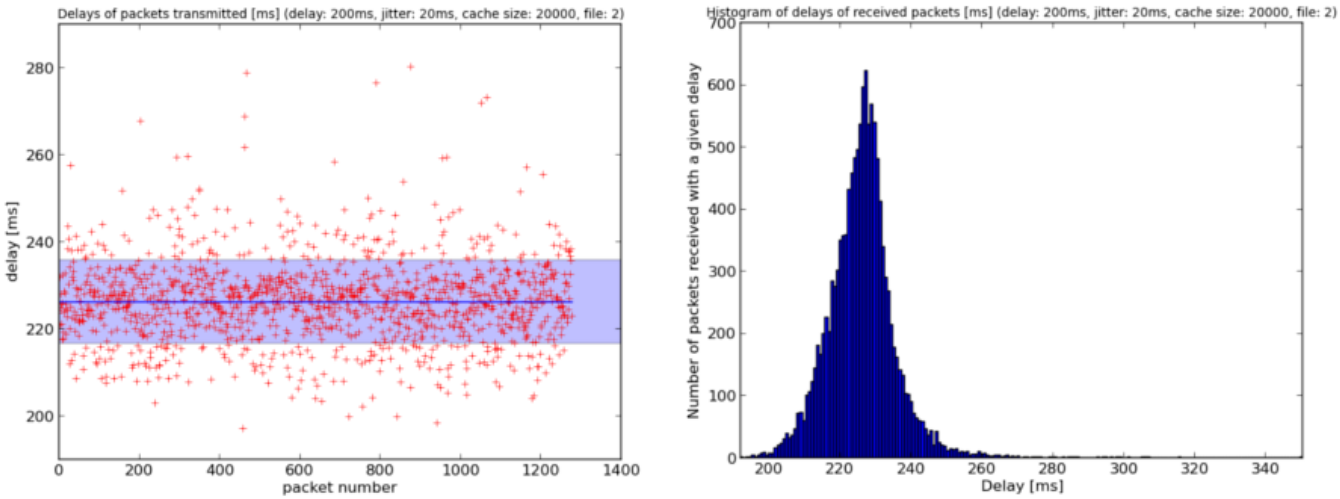




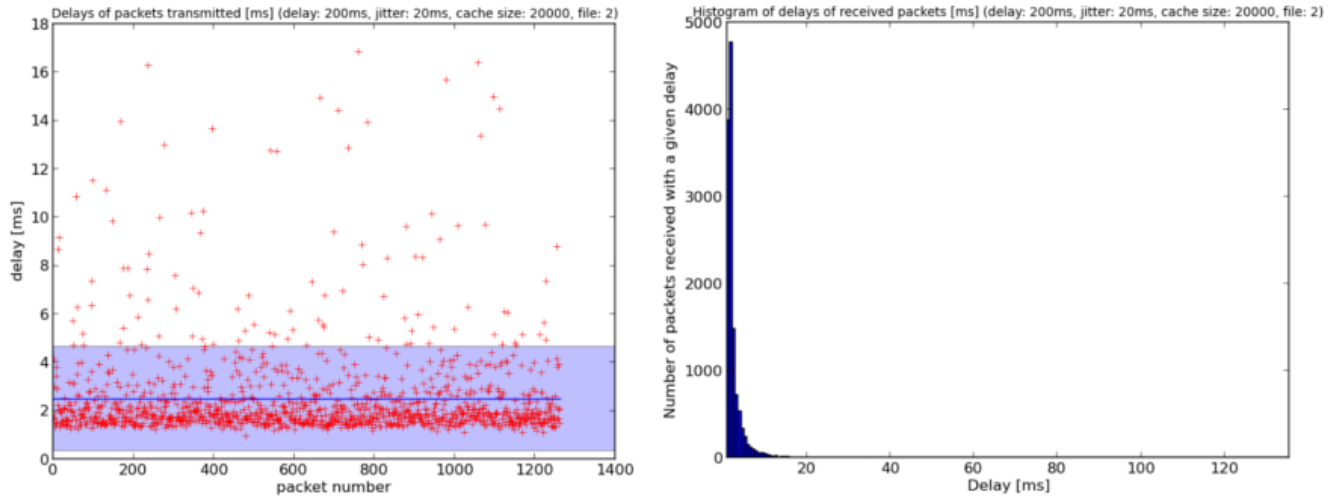
b) 200ms delay, cache size 10000, second download

Figure 15

- Caching strategies (Figure 16) - cache size 20000: First and second download differ much in mean delay and jitter due to the fact that in second download locally cached copy of the file is being used.



a) 200ms delay, cache size 20000, first download



b) 200ms delay, cache size 20000, second download

Figure 16

Links with losses

During our measurements we used UDP as a transport protocol for CCN packets. After introducing losses on a bad quality link (losses of ~20% of the total number of packets), CCN network was unable to handle download of the file. It means that the strategy layer of CCNx does not implement handling packet losses. Such a feature should be implemented in the future, since CCNx should be able to run using many different technologies, which may not support reliable transfer, as, for instance, TCP does.

Conclusions and future work

Most of the network traffic in the internet is being dominated by ever increasing content accessing and delivery services which are inducing large amounts of delays and bandwidth consumption. Further, the present internet architecture (TCP/IP) which revolves around a host - based communication model was not designed to support and effectively handle such high traffic flows. Content-Centric Networking (CCN) is a future internet architecture proposed to solve the above problems by naming the content itself instead of the content source and caching the content in the intermediate routers for effective content distribution and retrieval. In this paper, we investigate the factors and parameters like content to cache, cache size, content caching under changing link conditions, etc. that must be taken into account while deploying CCN in the present internet. We also compare CCN with the present internet architecture (TCP/IP) under different real world scenarios. Different tests and measurements were performed on both CCNx (CCN based open source experimental project) and HTTP (TCP/IP based network communication protocol) under different scenarios by simulating real world networks in our measurement setup.

Our measurements showed various interesting results along with some loopholes in the present implementation of CCNx like poor caching strategy, unable to handle packet losses, and implementation flaws in MTU (Maximum Transmission Unit) size. For most of the real world scenarios, HTTP still has better performance compared to CCN, but CCN proved to outperform HTTP in the case of large cache sizes in the intermediate CCN routers and multiple users/hosts requesting the same content data (content caching). We also proposed smart caching strategies and other implementation changes that are required in the present form of CCNx for large scale deployability of CCN architecture in the near future.

Our measurements were performed on a small scale network of five nodes/routers. We have simulated a wide area network between the two intermediate nodes/routers by inducing delays, jitters, packet losses, etc. in the link between them. Such a simulation can not truly represent a wide area network e.g. internet because network traffic parameters like delays, jitters, packet losses, etc. in such networks are not constant and may change with time and they also depend on other factors like distance between the nodes/routers, type of links, etc. So, we would like to extend our measurements and tests on more of a real world, wide area network (Internet) with long distant nodes/routers by using something like research test-beds as in Planet Lab [7], as our future work.

References

[1] V. Jacobson et al. (PARC). Networking Named Content, Co Next - 2009, Rome, December, 2009.

[2] Project CCNx, https://www.ccnx.org/, Jan. 2013.

[3] Github - Online Project hosting, /ccnx, https//github.com/ProjectCCNx/ccnx, Jan. 2013.

[4] Linux Foundation, netem, http://www.linuxfoundation.org/collaborate/workgroups/networking/netem, Jan. 2013

[5] Wireshark, Network Sniffer Tool, http://www.wireshark.org/, Jan. 2013.

[6] Python, Simple HTTPServer, http://docs.python.org/2/library/simplehttpserver.html, Jan. 2013.

[7] Planet Lab - Global Research Test-beds, http://www.planet-lab.org/, Jan. 2013.

Appendix A - Measurement tables

	No delay		No delay - second download		Delay 100ms		Delay 100ms - second download		Delay 200ms		Delay 200ms - second download	
Cache size	Mean delay [ms]	Standard deviation [ms]	Mean delay [ms]	Standard deviation [ms]	Mean delay [ms]	Standard deviation [ms]	Mean delay [ms]	Standard deviation [ms]	Mean delay [ms]	Standard deviation [ms]	Mean delay [ms]	Standard deviation [ms]
10	7.50	2.90	8.34	3.73	115.22	5.38	115.68	7.80	221.23	9.29	221.04	9.34
5000	8.14	3.31	8.47	3.80	115.09	6.30	114.70	6.51	222.11	10.86	221.17	8.85
10000	10.51	7.23	8.51	3.42	119.11	10.09	117.09	7.51	222.40	8.73	222.93	9.97
20000	9.06	4.86	3.37	2.27	116.22	14.80	4.17	14.78	220.80	7.46	2.97	1.40
50000	8.74	5.35	3.08	1.70	122.35	38.70	7.48	11.81	221.25	8.51	3.30	2.15

CCN			
0 ms delay	Download time [s]	Bandwidth [MB/s]	Relative bandwidth loss
CCN, cache size 10, first download	252.67	0.21	1.00
CCN, cache size 10, second download	285.94	0.18	1.13
CCN, cache size 5000, first download	302.98	0.17	1.20
CCN, cache size 5000, second download	299.66	0.18	1.19
CCN, cache size 10000, first download	308.03	0.17	1.22
CCN, cache size 10000, second download	349.05	0.15	1.38
CCN, cache size 20000, first download	342.73	0.15	1.36
CCN, cache size 20000, second download	329.73	0.16	1.30
CCN, cache size 50000, first download	277.26	0.19	1.10
CCN, cache size 50000, second download	247.01	0.21	0.98
100 ms delay	Download time [s]	Bandwidth [MB/s]	Relative bandwidth loss
CCN, cache size 10, first download	551.75	0.10	2.18
CCN, cache size 10, second download	645.24	0.08	2.55
CCN, cache size 5000, first download	582.12	0.09	2.30
CCN, cache size 5000, second download	549.67	0.10	2.18
CCN, cache size 10000, first download	576.50	0.09	2.28
CCN, cache size 10000, second download	550.32	0.10	2.18
CCN, cache size 20000, first download	520.98	0.10	2.06
CCN, cache size 20000, second download	329.97	0.16	1.31
CCN, cache size 50000, first download	548.57	0.10	2.17
CCN, cache size 50000, second download	399.72	0.13	1.58
200 ms delay	Download time [s]	Bandwidth [MB/s]	Relative bandwidth loss
CCN, cache size 10, first download	858.71	0.06	3.40
CCN, cache size 10, second download	842.70	0.06	3.34
CCN, cache size 5000, first download	888.08	0.06	3.51
CCN, cache size 5000, second download	859.32	0.06	3.40
CCN, cache size 10000, first download	1027.16	0.05	4.07
CCN, cache size 10000, second download	867.19	0.06	3.43
CCN, cache size 20000, first download	852.20	0.06	3.37
CCN, cache size 20000, second download	275.32	0.19	1.09
CCN, cache size 50000, first download	860.87	0.06	3.41
CCN, cache size 50000, second download	334.22	0.16	1.32

Appendix B - Example set up script

```
ip_addr=10.10.0.6
net_mask=255.255.255.0

ip_addr_next=10.10.0.2

ccn_prefix=ccnx:/

echo ---configure interface ip $ip_addr mask $net_mask
sudo ifconfig eth1 $ip_addr netmask $net_mask up
echo ---configured
sleep 2

echo  ---start ccnd daemon
ccndstart
echo  ---ccnd daemon started
sleep 2

echo  ---adding route to $ip_addr_next for $ccn_prefix
ccndc add $ccn_prefix udp $ip_addr_next
echo  ---route added
sleep 2

#echo  ---trying to get $filename for $save_filename
#ccngetfile $filename $save_filename
#echo  ---success
```

Appendix C - Restart measurement script

```
ccnrm ccnx:/
ccndstop

echo setting cache size to $1
export CCND_CAP=$1
echo cache size set to $CCND_CAP

echo setting mtu size to $2
export CCND_MTU=$2
echo cache size set to $CCND_MTU

./set_up2

echo ---CHECK CACHE CONTENT!
ccnls ccnx:/
echo ---END CACHE CONTENT CHECK
```

Appendix D - Python measurement script fragments

```
CCN_TYPE_INTEREST = '\x01\xd2'
CCN_TYPE_CONTENT = '\x04\x82'

HEADER_NAME1 = '\xf2\xfa\xcd'
HEADER_NAME2 = '\x00\xfa' #\xc5 or \xd5
HEADER_NAME3 = '\x00\xfa\xbd'
HEADER_CHUNK_NR = '\x00\xfa' #\x95 \x9d
HEADER_END = '\x00\x00'

#only for content packets
HEADER_SIGNED_INFO = '\x01\xa2\x03\xe2\x02\x85'
HEADER_CONTENT = '\x01\x9a'

class CcnPacket:
    def __init__(self, data):

        self.type = data[:2]

        if self.type  == CCN_TYPE_INTEREST:
            self.endOfHeader = self.processHeaderName(data)

        elif self.type  == CCN_TYPE_CONTENT:
            self.endOfHeader = self.processHeaderName(data)
        else:
            raise Exception('wrong type for CCN packet')


    def processHeaderName(self, data):
        tmp = data.split(HEADER_NAME1, 1)
        if len(tmp) == 2:
            tmp = tmp[1].split(HEADER_NAME2 ,1)
            self.name1 = tmp[0]
        else:
            raise Exception('Wrong format level1')

        if len(tmp) == 2:
            tmp = tmp[1].split(HEADER_NAME3 ,1)
            self.name2 = tmp[0]
        else:
            raise Exception('Wrong format level2', tmp)

        if len(tmp) == 2:
            tmp = tmp[1].split(HEADER_CHUNK_NR ,1)
            self.name3 = tmp[0]
        else:
            raise Exception('Wrong format level3')

        if len(tmp) == 2:
            tmp = tmp[1][1:].split(HEADER_END ,1)
            self.chunkNr = tmp[0]
        else:
            raise Exception('Wrong format level4', tmp)

        if len(tmp) == 2:
            return tmp[1]
        else:
            raise Exception('Wrong format level4')


def bits(i,n):
    return tuple((0,1)[i>>j & 1] for j in xrange(n-1,-1,-1))

def processIpPacket(ipPacket):
```

```
    global lastContentAdded

    CCN_TYPE_INTEREST = '\x01\xd2'
    CCN_TYPE_CONTENT = '\x04\x82'

    flagbits = bits(ipPacket[0].off, 16)

    df_flag = flagbits[FL_DF_POS]
    mf_flag = flagbits[FL_MF_POS]

    if df_flag and ipPacket[0].data.data[:2] == CCN_TYPE_INTEREST:
            ccnPacket = ccn.CcnPacket(ipPacket[0].data.data)
            chunkNrHex = ccnPacket.chunkNr.encode('hex')
            interestsAndContents[chunkNrHex] = ((ccnPacket, ipPacket[1]), (0, 0))
    elif ipPacket[0].data.data[:2] == CCN_TYPE_CONTENT:
        ccnPacket = ccn.CcnPacket(ipPacket[0].data.data)
        if df_flag:
            chunkNrHex = ccnPacket.chunkNr.encode('hex')
            if interestsAndContents.has_key(chunkNrHex):
                icTupleToodify = interestsAndContents[chunkNrHex]
                interestsAndContents[chunkNrHex] = (icTupleToodify[0], (ccnPacket, ipPacket[1]))
        else:
            chunkNrHex = ccnPacket.chunkNr.encode('hex')
            if interestsAndContents.has_key(chunkNrHex):
                icTupleToodify = interestsAndContents[chunkNrHex]
                interestsAndContents[chunkNrHex] = (icTupleToodify[0], (ccnPacket, 0))
                lastContentAdded = chunkNrHex
    elif (not df_flag and not mf_flag):
        if interestsAndContents.has_key(lastContentAdded):
            icTupleToodify = interestsAndContents[lastContentAdded]
            interestsAndContents[lastContentAdded] = (icTupleToodify[0], (icTupleToodify[1][0], i


filename = '../../measurements/big_file_delay_filtered'

f = open(filename)
pcap = dpkt.pcap.Reader(f)

ipPackets = []
for ts, buf in pcap:
    eth = dpkt.ethernet.Ethernet(buf)
    ip = eth.data
    ipPackets.append((ip, ts))

interestsAndContents = {}
lastContentAdded = 0
for ipPacket in ipPackets:
    try:
        processIpPacket(ipPacket)
    except Exception as e:
        print e

delays = []
for key in interestsAndContents.keys():
    iacTuple = interestsAndContents[key]
    interest = iacTuple[0]
    content = iacTuple[1]
    interestPacket = interest[0]
    interestTs = interest[1]

    contentPacket = content[0]
    contentTs = content[1]

    if contentPacket == 0 or contentTs == 0:
        continue

    delay = contentTs - interestTs
    delays.append(delay)
```

This topic: Education/ET4285 > Group32012
Topic revision: r32 - 08 Feb 2013 - 11:44:00 - ManiPrashanthVarmaManthena