The CONET solution for
Information Centric Networking

Technical Report - Version 0.4, January 2012

N. Blefari Melazzi(1), M. Cancellieri(1), A. Detti(1), M. Pomposini(1), S. Salsano(1)

(1)University of Rome “Tor Vergata”

Available at: http://netgroup.uniroma2.it/CONET/

Contact: stefano.salsano@uniroma2.it

DISCLAIMER

This work has been partly funded by the EU in the context of IST CONVERGENCE (http://www.ict-convergence.eu/) and OFELIA projects (http://www.fp7-ofelia.eu/), but it does not necessarily represent the official position of the project itself and of its partners. Authors are solely responsible for the views, results and conclusions contained in this work.

COPYRIGHT

"Copyright and all rights therein are retained by authors. This work may not be modified or reposted without the explicit permission of the copyright holder"
## Contents

1. What is CONET? .............................................................................................................. 3

2. Problem statement ......................................................................................................... 4

3. Related work .................................................................................................................. 7
   3.1 Related Works ............................................................................................................ 7
   3.2 Comparison between our work and other solutions ................................................. 7
      3.2.1 Specific comparison of CONET against CCN/CCNx implementation ................. 7

4. CONET Architecture ..................................................................................................... 8
   4.1 Naming ...................................................................................................................... 9
   4.2 Content routing and forward-by-name operations ................................................. 10

5. CONET IPv4 and IPv6 Options .................................................................................... 11

6. CONET fragmentation and transport .......................................................................... 12

7. Measurement and simulation results .......................................................................... 14
   7.1 Support of unrecognized IP Options across Internet ............................................. 14
   7.2 Lookup and cache effectiveness ............................................................................. 15
   7.3 Throughput measurements of CCNx (also with packet loss) .................................. 16
      7.3.1 Measurement of the (TCP) throughput using Proftpd on Linux-Ubuntu machines .. 16
      7.3.2 Measure of the CCNx throughput ................................................................... 17
   7.4 Transport issues: fairness among ICN flows and TCP flows ......................... 24
      7.4.1 Analysis of goodput versus packet loss probability ...................................... 25
      7.4.2 Coexistence of content-centric flows and plain TCP flows ......................... 26
      7.4.3 Timescale of congestion control .................................................................. 27
   7.5 Transport level efficiency ....................................................................................... 27

Acknowledgements .......................................................................................................... 31

References ......................................................................................................................... 32
1. What is CONET?

Information Centric Networking (ICN) has been presented as a revolutionary approach to networking. The proposers of ICN aim to replace the current forwarding mechanism of TCP/IP networks based on host addresses (i.e. IP addresses) with a mechanism based on the “name” of the information to be transported (i.e. the “content name”). The work on ICN has mostly focused on "clean-slate" solutions, in which the IP networking layer is fully replaced by a new information centric networking layer. In this work we propose a different approach towards ICN, based on extending the current IP networking architecture rather than considering its replacement.

The proposed framework is called CONET (COntent NETwork). We argue that it is possible to add ICN functionality to IP in a backward compatible way, so that most advantages of ICN can be exploited without dropping IP. Note that considering an evolutionary approach does not preclude the clean-slate one, as the CONET framework and most protocol solutions that we describe can also work in a clean slate environment that does not foresee the IP layer.

The CONET framework is very modular and open, it lends itself to support different solutions for specific issues like naming, name based routing, forwarding and transport mechanisms.

In addition to this general framework we pursued the definition of a specific ICN solution denoted as coCONET\(^1\). We have started an open source implementation of coCONET, which allows us to build a testbed and to perform measurements. The implementation work (still ongoing) is mostly based from the CCNx (http://www.ccnx.org/) implementation of an ICN.

The home page for retrieving more information about CONET, downloading coCONET source code and installing it is:

http://netgroup.uniroma2.it/CONET/

---

\(^1\) coCONET : convergence CONET, named after the EU project CONVERGENCE
2. Problem statement

In this section we analyze the problem space, i.e. the communication patterns that needs to be supported by the ICN (with no claim of being exhaustive and rigorous), then we discuss the overall approaches for the design of an ICN and decompose the ICN solution space in a set of logical components (see Figure 1).

<table>
<thead>
<tr>
<th>Problem Space</th>
<th>Solution Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supporting diverse communication patterns</td>
<td>Overall Approach</td>
</tr>
<tr>
<td>content based</td>
<td>clean slate</td>
</tr>
<tr>
<td>file transfer</td>
<td>overlay</td>
</tr>
<tr>
<td>file streaming</td>
<td>evolutionary</td>
</tr>
<tr>
<td>live streaming</td>
<td>Components</td>
</tr>
<tr>
<td>multicast</td>
<td>naming</td>
</tr>
<tr>
<td>point-to-point multimedia</td>
<td>routing</td>
</tr>
<tr>
<td>transactions (e.g. interactive web)</td>
<td>forward-by-name</td>
</tr>
<tr>
<td>conversational based</td>
<td>data forwarding</td>
</tr>
</tbody>
</table>

Figure 1 - Problem space and solution space for ICN

A “content-oriented” communication pattern takes place when a user (or a user application) needs a specific content irrespective from where the content is stored. The content is associated with an identifier (the “name”) unique over a certain namespace. This communication pattern is best served by the networking layer with a couple of primitive operations as shown in left column of Table 1, that allow the user to put content in the network and to get content from the network. Note that the content can be already available and stored somewhere in a content server (like a document, a picture, a movie, a static web page) or it can be a live stream (audio, video or multimedia). Moreover a stored content like a movie can also be “streamed” i.e. the requester can start playing out the content without waiting for the full file to be transferred. Taking the above into account, a different couple of primitive operations (publish/subscribe) may prove useful in some cases, and the content-oriented communication pattern can be sub-classified in “file transfer” / “file streaming” / “live streaming”. A final dimension to be added is the unicast/multicast attribute, as a the same content can be requested by more than one user at the same time (and this is relevant in particular to live streaming).
Table 1 – Content oriented and socket oriented communication primitives

<table>
<thead>
<tr>
<th>Content oriented primitives (basic version)</th>
<th>Socket oriented primitives (unconnected)</th>
</tr>
</thead>
<tbody>
<tr>
<td>data = get(contentName)</td>
<td>sendData(remoteEntityAddress, data)</td>
</tr>
<tr>
<td>put(contentName, content)</td>
<td>data=receiveFrom(localAddress)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Content oriented primitives (pub/sub version)</th>
<th>Socket oriented primitives connected</th>
</tr>
</thead>
<tbody>
<tr>
<td>data = subscribe(contentName)</td>
<td>channel = openChannel(remoteEntityAddress)</td>
</tr>
<tr>
<td>publish(contentName, content)</td>
<td>write (channel, data)</td>
</tr>
<tr>
<td></td>
<td>data=receiveFrom(channel)</td>
</tr>
</tbody>
</table>

On the other hand we have a “conversational” communication pattern when a user / user application wants to exchange information in a bidirectional way with a remote entity. In this case there is a natural fit with the set of primitive operations shown in the right column of Table 1. The reader can recognize the socket abstractions which are currently used in TCP/IP network (for unconnected and connected services). This pattern clearly applies to voice and audio/video calls, but also to most of the current typical applications and services on the web. If for example we look at most social networking applications, they require a continuous exchange of information from the client (the browser) to the server. These applications are not only about downloading and playing/showing a content, but they are about sending lots of information from the client side to the server side, therefore requiring a conversational communication pattern. More in general, most web applications can be seen as composed by content-oriented communication patterns mixed up with conversational patterns.

If we look at the solution space for ICN, the “clean slate” approach aims at fully replacing the existing IP layer, i.e. assuming that the ICN layer will sit over layer 2 networks like Ethernet. On the other hand, existing prototype implementation of “clean slate” ICN solutions, as for example CCNx [8], mostly rely on an “overlay” approach to run over the existing IP based networks. The overlay approach means that ICN information units are tunneled within TCP or UDP flows running over IP. This is explained by the proposers as a short-term approach, i.e. a temporary “hack” needed to deploy ICN implementations over the current networks. Indeed, we think that this demonstrates the necessity and the usefulness for an ICN solution to operate on and to interwork with existing IP networks.

Considering the above, we see two main drivers for an evolutionary approach towards ICN.

i) **Content-oriented communication pattern is overestimated, conversational communication pattern still remains very important.** Content-oriented and conversational communication patterns needs both to be well addressed in the future internet evolution. Proponents of clean slate approaches for ICN argue that also conversational pattern would be supported by content centric networking mechanisms (as in [7]). We are not sure that this is the case, or at least we think that there are no benefits in changing the way this communication pattern is supported by current IP based networks. Looking at Table 1, turning the primitives in the right column into those in the left column is doable, but it does not always bring benefits compared to the required effort.
ii) **Having an evolutionary path from existing IP networks could be the only way for a real deployment of ICN.** The development of “overlay” solutions demonstrate the need of evolving from existing networks. Another obvious argument is that existing IP networks will simply not be dismantled at once to migrate to ICN.

Looking at the solution space for the ICN we identify seven fundamental components, i.e. naming, forward-by-name, content routing, data forwarding, caching, segmentation & transport, and security. The **naming** scheme specifies the identifiers for the contents addressed by a ICN. The choice of the naming scheme strongly impacts some aspects of ICN like usability, scalability and security. For instance, human-readable flat-names improve usability; hierarchical names foster aggregation of names in name-based forwarding tables; names containing the public key of the owner of the content simplify security. The **forward-by-name** is the mechanism used by ICN nodes to relay an incoming content request to an output interface. The output interface is chosen by looking up a “name-based” forwarding table. The size of the forwarding tables is critical when high speed packet forwarding is needed. The **content routing** is the mechanism used to disseminate information about location of contents, so as to properly setup the name-based forwarding tables. For instance, content routing could use plain IP distance vector mechanisms, where name prefixes are distributed instead of IP prefixes. Content routing is one of the assets of ICN, as a provider could use content routing to improve the efficiency and reliability of content access in its network. The **data forwarding** mechanism allows the content to be sent back to the device that issued a content request. Data forwarding cannot use the forward-by-name mechanisms, because the devices are not addressed by the content routing plane of an ICN. Therefore, an ICN requires two different forwarding strategies to forward content requests and to deliver the data. In-network **caching** concerns the ability of ICN nodes to cache data and to directly reply to incoming content requests. In-network caching is another asset of ICN. Caching can be performed in a simpler with respect to other approaches like HTTP caching, that require the handling of more complex state information (e.g. TCP and HTTP sessions). Thanks to this simplification, a provider could exploit caching in critical network elements, like gigabit POP nodes connecting thousands of DSL users or high speed border gateways. The **segmentation & transport** mechanisms are needed: 1) to split a content in different chunks (each chunk is an autonomous data unit with embedded security and addressable by the routing plane); ii) to ensure a reliable transfer of chunks from the origin node (or from a cache node) towards the requesting node; iii) to counteract congestion that may happen in the network. **Security** mechanism needs to prevent the requesting nodes from receiving fake content and the ICN nodes from storing fake content in their cache. Without these security mechanisms, it would be very easy to inject fake content or to corrupt the genuine content achieving a denial of service attack.
3. Related work

3.1 Related Works
<Work in progress>

3.2 Comparison between our work and other solutions
- <Work in progress>
- The possibility to record the path info in content requests, and use this path info as a kind of source routing for forwarding the content back towards the requester. This can be used as a replacement for the Pending Interest Tables that needs to be maintained by the nodes otherwise.

3.2.1 Specific comparison of CONET against CCN/CCNx implementation
- lookup-and-cache for forwarding of content requests
- different approach for the naming of content
- segmentation of the chunks in carrier packets
- transport algorithms in a transport layer rather than in application layer
- full receiver based implementation of TCP like mechanism (slow start congestion avoidance fast retransmit fast recovery) rather than variable size windows with selective retransmission request
- kernel based implementation
4. CONET Architecture

Figure 2 shows the basic elements of the proposed CONET framework. We have been inspired by CCN [6] and share some design principles with it. In CONET we have Serving Nodes, End-Nodes, Border Nodes, Internal Nodes and plain IP Routers. Serving Nodes provide content. End-Nodes require content using CONET protocols, providing the name of the content to be retrieved. Under CCN [6] terminology, the content requests are called “interests” or “interest packets”. These content requests are forwarded over the ICN network taking into account the requested content-name, this process is denoted as “forward-by-name”. The Border Nodes: i) forward content-requests from End Nodes to Serving Nodes, performing the “forward-by-name” operation; ii) deliver content from Serving Nodes to End Nodes; iii) may cache content and therefore provide it to End Nodes without forwarding the requests to Serving Nodes, in this case the Border Nodes perform security checks in order not to store and redistribute fake content.

In order to discuss Internal Nodes and plain IP Routers, we first introduce the concept of CONET Sub System (CSS). A CSS contains a set of nodes and exploits an under-CONET technology to transfer requests and data among the nodes. CONET is an inter-network that interconnects CSSs. A given CCS can be for example:

- two nodes connected by a point-to-point link
- a layer 2 network like Ethernet
- a layer 3 network e.g. a private IPv4 or IPv6 network or a whole Autonomous System

The nodes within a CSS use an autonomous and homogeneous under-CONET addressing space and, if necessary, an interior under-CONET routing protocol. As shown
in Figure 2, Border Nodes interconnect different CSSs, therefore the end-to-end forward-by-name process can be seen as the process of finding a sequence of Border Nodes from the End-Node up to the Serving Node. When a CSS is an IP network, the downstream Border Node performs a forward-by-name procedure to resolve a content name and gets as a result the IP address of the upstream Border Node. It sends the content-request (interest) packet to the upstream Border Node using IP. The IP CSS between downstream and upstream Border Node can be composed by an arbitrary number of plain IP Routers and Internal Nodes. The latter can be seen as “enhanced” IP routers in the sense that they can perform content caching but are not able to perform “forward-by-name” (only plain IP routing).

At a glance, the operation in a CONET internetwork can be described as follows. A Border Node checks if the content requested by the content request is available in its cache, if not it performs forward-by-name. If the CSS is an IP network, the result of the forward-by-name operation is the IP address of the upstream Border Node, therefore the content request can be sent using this destination IP address. An Internal Node in the path between the two border nodes “intercepts” the content request, it checks if the requested content is available in its cache, if not it forwards the packet using the IP destination address. A plain IP Router in the path between the two Border Nodes will simply forward the packet looking at the IP destination address. When data packets providing the requested content are generated by the Serving Node towards the End-node (or by any Border or Intermediate Node that had cached the content, the crossed downstream Border Nodes and Internal Node can in turn cache the content while forwarding it. In this way, further content requests for the same content will not need to travel up to the Server Node.

4.1 Naming

The issue of naming in an ICN, i.e. which identifiers should be used to identify and then route content is a very hot topic and different proposals are still the table. The proposals belong to two different “families”, one foresees that the names should be “human readable”, another foresees that the name needs not to be human readable and therefore they may have other interesting security properties like for example being “self-certified”. The CONET options proposed in [4] can support any choice even in parallel, i.e. different types of naming can be supported in an ICN. The content name is transported in a field called ICN-ID. This is tuple `<namespace ID, name>`. The namespace ID determines the format of the name field. Thus, the name field is a namespace-specific string. Each namespace follows its own rules to release unique names with its own format. Going into details, the ICN-ID starts with a two bytes field called ICN-ID namespace ID that determines the structure of the rest of the ICN-ID. ICN-ID namespace values to be used in the future public Internet needs to be assigned by a registration authority like IANA.

Within coCONET, we made a specific proposal for a naming format. The content name is the composition of two hash values, i.e. content name=<hash (Principal), hash (Label)>. Principal and label [9] are flat-names that can be human readable and a hash function transforms them to a fixed number of bytes. A principal is the owner of her address space identified by the Principal identifier. The hash of the Principal identifier must be unique in the namespace (therefore an entity should assign the Principal identifiers). Label is an identifier chosen by the principal to uniquely differentiate each content. The hashing transforms variable length Principal identifier and Label into a fixed size structure. For instance, to support the WEB resources we could define the namespace “www”, which follows the actual domain name assignment rules and uses the domain name (e.g. www.cnn.com) as principal identifier and the URL path (e.g. /foo/index.html) as label.
4.2 Content routing and forward-by-name operations

The content routing is the set of mechanism used to distribute information about location of contents and network topology. Thanks to this mechanism, it is possible to update the name-based forwarding tables, used by End-nodes and Border Nodes to perform forward-by-name on the content requests.

In [6] the authors suggest to use traditional routing protocols, e.g. BGP or OSPF, to disseminate name-prefixes and the related topological information. This approach could produce very large name-based routing tables, because the aggregation of names (i.e., network-identifiers) is not effective, when names do not include information about “where” is the serving node.

The solution we have designed and implemented in coCONET is based on the concept of lookup-and-cache. The routing process is handled at a layer called NRS (Name Routing System), without directly involving the Border Nodes and End-Nodes.

An End-Node or Border Node uses a fixed number of rows in its name-based forwarding table as a cache. If the info required to forward a content request is missed, the node asks for the forwarding entry to a NRS Node and inserts the entry in the forwarding table. Therefore we have a full routing table in the NRS and a limited and “opportunistic” forwarding table within End-Nodes and Border Nodes. Only the nodes in the NRS participate to the content routing process, using the approach proposed in DONA [9] to distribute name-prefixes among them.
5. CONET IPv4 and IPv6 Options

The full details on the definition and usage of the proposed IPv4 and IPv6 CONET options can be found in [4]. The CONET IP Options (see Figure 3) have the same content for IPv4 and IPv6, the initial byte is different according to the different rules for coding of options. In IPv6, the CONET Option will be transported in the Hop-by-Hop Options header, that is meant to be analyzed by all routers in the path. The CONET IP Options include the content name, called ICN-ID, which can be of variable size or fixed size according to the selected naming approach. A namespace ID allows to support different types of naming schemas. A Chunk Sequence Number (CSN) of variable length can be optionally present. This allows to offer the chunk number information to the ICN layer in a “standard” way (the alternate approach is to define a naming schema that include the Chunk number, so that this information is transported in the ICN-ID). The CONET IP Option include a “Diffserv and Type” field. This one byte length field is used to differentiate quality of services that can be provided by the network to the delivered content and to identify the content type.

We have verified (with practical experiments on PlanetLab, see section 7.1) how an unrecognized option is handled by current routers in the Internet. The result was that, barring few exceptions, it was possible to add unrecognized option and achieve end-to-end connectivity.
6. CONET fragmentation and transport

Content to be transported over an ICN can be very variable in size, from few bytes to hundreds of Gigabytes (and it is not easy to set an upper bound to content size). Therefore it needs to be segmented in smaller size data units, typically called chunks, in order to be handled by the ICN nodes. A chunk is the basic data unit to which caching and security is applied. This means that: i) a single chunk out of a larger content can be requested by an End-Node; ii) single chunks will be signed by the origin Server-Node for security reasons; iii) Border and Intermediate Nodes will authenticate the chunks and store them in their caches as needed.

Each chunk contains an amount of overhead, mostly due to the security information (e.g. the signature needed to authenticate the chunk) which is more or less independent from the chunk size. Taking the CCNx [8] implementation as reference, this overhead is in the order of 400 bytes per chunk. Each signature and authentication operation is computationally very expensive and again this cost is mostly independent from the chunk size (this is because the computationally expensive operations, based on asymmetric cryptography, are applied to a fixed size hash of the chunk, while the cost of producing the hash that depends on the chunk size can be neglected). These considerations on the overhead and on the number of cryptographic verification operations to be performed by caching nodes suggest the chunk size to be rather large. We argue that reasonable chunk sizes can be in the order of hundreds of KBs (e.g. 128, 256, 512) up to few MBs (e.g. 1, 2, 4). When chunks of such relatively large size are exchanged among ICN nodes they need to be in turn fragmented to be handled by level 2 technologies that prescribe a Maximum Transfer Unit of much smaller size. Moreover, a proper transport protocol is needed to handle reliability and congestion control. These issues have not been adequately covered in the design and early implementation of Information Centric Networks. Taking for example CCN [6], it uses the chunk as the fundamental transport unit to be exchanged among nodes and relies on the underlying technologies and protocols to actually forward them (e.g. UDP tunneling in the overlay approach used by the CCNx [8] implementation). Congestion control and reliability aspects are handled by the CCN applications on a chunk by chunk basis. We observe that in this approach the transport protocol can work at maximum efficiency if the chunk size is smaller than the IP Maximum Transfer Unit over the crossed link layers. If the chunk size is bigger than the IP MTU, the chunks will be subject to IP fragmentation. Let us denote Nc the number of IP fragments per each chunk (i.e. Nc=Chunk Size / MTU). In case of IP packet loss (for example due to congestion) a larger Nc has a negative effect on performances (and efficiency) as the loss of a single IP packet will lead to the loss of the whole chunk to which the IP packet belong. It can easily be shown that for Nc greater than 3-4 the efficiency decreases and it is not possible to achieve fairness with respect to competing TCP flows. In fact, the TCP flows will cause the starvation of the ICN flows in case of congestion. For this reason the default value of chunk size in CCNx [8] implementation is set to the relatively small default value of 4 KB, i.e. roughly 3 IP packets of 1500 bytes. This implies a 10% overhead on the bit rate and the need to perform signature checks at a very high rate, compared to what would have been possible with a chunk size in the order of hundreds of KBs or few MBs as argued above.
Taking the above into account we propose to handle the segmentation of content with two levels: at the first level the content is segmented in chunks, at the second level the chunks are segmented into smaller data units that are handled by an ICN specific transport protocol performing reliability and congestion control, much like TCP does for the transfer of a byte stream in current TCP/IP networks. This approach is represented in Figure 4. CONET nodes exchange **CONET Information Units (CIUs):** *interest CIUs* convey content requests; *data CIUs* transport chunks of content, e.g., parts of a file. In order to handle the fragmentation in an efficient way, fitting the transfer units of the under-CONET technologies, all CIUs are carried in smaller data units named *carrier-packets.*

![Figure 4- CONET Information Units (CIUs) and carrier-packets](image)

The proposed transport protocol is based on the principles described in [6]. It is a receiver-driven approach implementing the same algorithms of TCP (slow-start, congestion avoidance, fast retransmit, fast recovery). The transport algorithm issues a sequence of interest CIUs and each of them requests only a fraction of a data CIU. By controlling the sending of these interest CIUs, we have achieved a transport protocol that is TCP friendly and can achieve fairness both among multiple competing ICN content downloaded and among ICN content download and regular TCP flows. In the CCNx implementation the transport protocol (reliability and congestion control) is implemented at the application level, above the API toward the ICN layer. We believe that this is not efficient and preferred to embed the transport protocol below the API towards the application, just above the networking layer (it is the same approach used in TCP/IP networks, where the TCP is implemented in the kernel and the applications are provided with an API that hides the details of the transport protocol). The details of the transport protocol in terms of bit and bytes can be found in [5].
7. Measurement and simulation results

7.1 Support of unrecognized IP Options across Internet

In this section, we verify the feasibility of conveying the header of carrier-packets in an IPv4 option, i.e. the CONET option. The rationale of this test lies in the fact that IP routers tend to process packets with IP options in the slow forwarding path; therefore, current IP routers could become a critical performance bottleneck for our solution, as plain IP routes and CONET nodes would need to co-exist in a hypothetical real deployment scenario.

![Graph of Throughput and round-trip-delay of IP packets with and without CONET options on different Internet paths](image)

To check the behavior of current IP routers, we sent IP packets with and without our CONET option (simultaneously) on the Internet and we measured the difference in terms of round-trip-delay and throughput (i.e. the available capacity between a sender and a receiver). We used eleven PlanetLab nodes, spread over the Internet (Asia, Europe, North America, Australia). Each measurement was performed between a PlanetLab node and a node in our premises (Rome, Italy). Each measurement has been repeated ten times and Figure 5 reports the average values.

As regards the throughput, we observe that we have almost the same performance, with and without the CONET option, for the first nine PlanetLab end-nodes. On the other hand, we observed considerable differences in the case of the last two end-nodes. Further analysis revealed that: i) on the Beijing-Rome path there is a router that statistically drops half of the packets with IP options; ii) on the Colgate-Rome path there is a router (in Australia) that drops all packets with IP options. The problem regards a minority of the examined routers, depends on a software configuration and we conjecture that these policies are enforced to prevent DoS attacks [10]; such policies could be modified, so as to
accept CONET carrier packets without restrictions. As regards the round-trip-delay, we observe a small increase of the latency for packets with the CONET option. Overall, our measurements show that IP routers, properly configured, would not be a critical performance bottleneck, and therefore the use of the IP CONET option seems feasible (see also [11] for a similar analysis).

7.2 Lookup and cache effectiveness

We remind that the routing-by-name process involves only interest-CIUs, since data-CIUs are routed back to the end-node by means of source-routing. The CONET nodes involved in routing-by-name are either end-nodes or border-nodes. In case of end-nodes, the lookup-and-cache approach resembles the interaction between an Internet host and a DNS server, where the host implements a local DNS cache service. Therefore, we argue that lookup-and-cache is feasible on end-nodes and we focus on its feasibility in border-nodes.

We assume to replace a standard TCP session between a client and a WEB server with a CONET session (exchange of CONET CIUs) between an end-node and a serving-node, or an intermediate cache. Specifically, we assume that:
- an URL <http://IP address:80 (or domain-name)/path> is replaced by the network-identifier: namespace="www", principal="IP address:80", label="path";
- TCP segments are replaced by carrier-packets that convey segments of named-data CIUs;
- TCP ACKs are replaced by carrier-packets that convey interest CIUs.

With these assumptions, we can map a real Internet trace, formed by TCP segments and ACKs, to a “CONET trace”, formed by carrier-packets. We applied this re-mapping to two Internet traces: the first one captured on an interface at 10 Gbit/s of a tier-1 router [12]; the second one captured on an interface at 10 Mbit/s of a tier-3 router [6].

![Figure 6- Lookup frequency of a tier-3 and a tier-1 border-node](image)

The two re-mapped traces have been fed to a CONET border-node, which we emulated in SW, to analyze the effectiveness of the lookup-and-cache routing for a tier-1 and a tier-3 border-node. Following the approach suggested in [9], we assumed that routing-by-name is performed only on the base of the principal identifier. This means that a name-based routing entry has the form <namespace, hash(principal),*> and that all the named-data of a given principal are stored in a serving-node (and in its replicas, if any). We also assume
that the route cache adopts a Least Recently Used (LRU) caching policy, discarding the least recently used item first.

Figure 6 shows the obtained results in terms of name-lookups per second issued by the border-node to the name-system, versus the size of the route cache. The route caching performance improves (i.e. lower lookup frequency) in a log-like fashion versus the cache size. In the case of the tier-3 node, we have about 2 lookups per second and a cache-miss probability of about $10^{-3}$, by using a route cache of 2k entries. In the case of the tier-1 node, we have about 10 lookups per second and a cache-miss probability of about $10^{-4}$, by using a route cache of 8k entries. Considering that nowadays BGP routers handle about 350k entries and 2 or 10 lookups per seconds are reasonable values, we can conclude that lookup-and-cache seems feasible with the current technology.

## 7.3 Throughput measurements of CCNx (also with packet loss)

### 7.3.1 Measurement of the (TCP) throughput using Proftpd on Linux-Ubuntu machines

The first test we performed was measure the throughput achieved with the TCP (using the open source software Proftpd) and compare the results with those achieved when using CCNx to transfer data or contents in the same conditions.

The scenario used is formed by two Linux (ubuntu) machines directly connected with a crossover Ethernet cable.

The main software to install and use is Proftpd; it is a FTP server developed for use on *nix OSs.

#### 7.3.1.a. Equipments used

- Two Linux machines called PC1 and PC2 with characteristics shown in the table below (tab.4)
- One Gigabit Ethernet Crossover cable (1000Mbps) with RJ45 connectors

<table>
<thead>
<tr>
<th>Construct</th>
<th>Processor (CPU)</th>
<th>RAM</th>
<th>Net Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC1</td>
<td>SONY VAIO</td>
<td>4.00 GB</td>
<td>100Mbps</td>
</tr>
<tr>
<td>PC2</td>
<td>DELL</td>
<td>4.00GB</td>
<td>100Mbps</td>
</tr>
</tbody>
</table>
7.3.1.b. Tests

For this test, we used ten different files to transfer. For each file sent, we measured the transfer time $\Delta t$ (seconds) and calculated the throughput (Mbit/s). The results are recorded in Table 3.

- Table 3 – TCP transfer time and throughput

<table>
<thead>
<tr>
<th>File N°</th>
<th>File Size (MB)</th>
<th>Transfer Time $\Delta t$ (sec)</th>
<th>Throughput (Mbit/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19.5</td>
<td>1.76</td>
<td>93.16485</td>
</tr>
<tr>
<td>2</td>
<td>50.5</td>
<td>4.52</td>
<td>93.686031</td>
</tr>
<tr>
<td>3</td>
<td>0.001585</td>
<td>0.03</td>
<td>43.27893</td>
</tr>
<tr>
<td>4</td>
<td>24.1</td>
<td>2.15</td>
<td>93.99358</td>
</tr>
<tr>
<td>5</td>
<td>211</td>
<td>18.85</td>
<td>93.89173</td>
</tr>
<tr>
<td>6</td>
<td>6.3</td>
<td>0.57</td>
<td>92.048715</td>
</tr>
<tr>
<td>7</td>
<td>19.9</td>
<td>1.79</td>
<td>93.169006</td>
</tr>
<tr>
<td>8</td>
<td>3.3</td>
<td>0.30</td>
<td>91.0944</td>
</tr>
<tr>
<td>9</td>
<td>284.3</td>
<td>25.38</td>
<td>93.97611</td>
</tr>
<tr>
<td>10</td>
<td>236.4</td>
<td>21.11</td>
<td>93.94994</td>
</tr>
<tr>
<td>11</td>
<td>141.9</td>
<td>12.66</td>
<td>93.99423</td>
</tr>
</tbody>
</table>

7.3.2 Measure of the CCNx throughput

The main applications used are ccnsendchunks cccatchunks, cccatchunks2 and ccnputfile.

ccnsendchunks is a CCNx C API. It is a program used to chop (segment) contents in small data units called chunks and inject them from stdin to ccn. These chunks are produced as the program receives interests and are sent as consecutively numbered “ContentsObjects” under the given CCNx URI (the CCNx Name). Data produced for the moment can be read by a correspondent CCNx API, cccatchunks and cccatchunks2. This program is a CCNx command-line utility. The default blocksize is 1 k (1024 bytes). If
you give a different blocksize (e.g. 2048) as option using ccnsendchunks, the receiving program to use is ccncatchunks2, otherwise the ccncatchunks program generate an error message inviting to use ccncatchunk2 and saying that it only works with segments of 1024 blocksize; ccnsendchunks which **signs every block** and, by default, uses 1024 byte blocks you can use `-b 4096` to increase it. Unlike Java library used, there is no option yet for generating MHT (Merkle Hash Trees) signed streams from the C library; signing every block increases overhead and as result decreases throughput. The pipeline option `[-p pipe_value]` can be used to increase performances. In CCNx pipeline is a “transmission window” like in TCP; it is used by CCNx applications for flow control scope.

![Figure 8- Segmentation of CCNx content in chunks](image)

**ccncatchunks2** is CCNx C API, a command-line utility used to read stuff written by ccnsendchunks and write them to stdout. As I said in the uppers lines, ccncatchunks2 is used rather than the simple ccncatchunks when you give at the command-line a blocksize value (option) different to the default one 1k (1024 bytes) to speed up the data transfer. Example:

```
$ccnsendchunks ccnx:/pippo.ccn -b 2048
```

ccncachunks2 implements a flow control mechanism. It is a “window based” flow control, using a maximum window size as a parameter like TCP does. The windows size is dynamically controlled. With the RTT (Round Trip Time or Round Trip Delay) estimation (RTTe), ccncatchunk2 makes the automatic variation of the window size. The ccncatchunks2 tool was written as a proof of concept that pipelining could work with the ccn protocols. It is very simple compared to what Transport Protocol TCP does. The maximum windows size is settable up to the value of 127. What is important in CCN is the flow balance (one interest sent-one content back), this basic mechanism provide indirectly a flow control. In CCN the sender must transmit at most one Content Object message in response to a single received Interest message, even if the party has many Content Objects that match. This one-for-one mapping between Interest and Data messages maintains a flow balance that allows the receiver to control the rate at which data is transmitted from a sender. In the source code, the actual window size is the current pipeline or current window called “curwin”: It is actively managed with a value between 1 and the maximum pipeline size given (default 31). The pipeline starts at 1 (slow-start: it issues a single interest) no matter what the pipeline size is, and then on receipt of the first packet opens up the pipeline and is increased by 1 up to the maximum each time data is successfully received, it is reduced by 1 when out-of-order data is received. It is reset to 1
when the hole-filling code (fill_holes() function) initially determines that no progress is being made and there is out-of-order data present.

In the test below, we varied the block size and used it with the default flow control mechanism implemented. The command to be used is:

```bash
$ccncatchunks2 [-b bloc-size] [-p pipeline_value] URI > ~/[path_to_save_content]
```

The same equipment described in Table 2 was used for the test.

![Figure 9- CCNx performance evaluation using Catchunks2 & ccnsendchunks](image)

After configuring machines, from the shell, we started the **ccnd** (CCNx Deamond, the software forwarder/router) with the following command without options: **ccndstart**

We configured the FIB (Forwarding Information Base) using the simple routing utility/deamon ccndc; It is the only tool (in this current release) to use to configure the routing table. It used to manipulate CCNx forwarding table.

The CCNx web interface can be used to display the status of the running ccnd (cache statistics and forwarding table entries) and is accessed at the following URL: http://localhost:9695. An example result is shown in the following table:
At this point we are ready to start the test. We can launch the wireshark protocol analyzer on the machine PC1 to see what was going across the link and interface. We ran `ccndsendchunks` program from PC2; after few minutes we ran `ccncatchunks2` program from PC1. The command used was:

```
$ccndsendchunks ccnx:/Name –b ChunkSize < ~/Folder1/FileName
```

The symbol "< " is used to specify were the content is stored in the machine; Name is arbitrary, we used test1, test2 and so on; Folder1 is the the real name of the folder (in the home directory of PC1) containing files to use for the test (e.g. Test_ccnx); it must be created and be populated with files before running this command. You can choose another
path for that folder; ccnx:/Name is the URI (ccnx name chosen following the name convention for the software ccnx-0.4.1); ChunkSize is the desired size of chunks; I passed this as option. For this test we used five different files with different sizes (3.3MB, 6.3MB, 19.5MB, 50.5MB, 141.9MB) and for each file we ran the command 8 times using different values of ChunkSize (1024 bytes, 2048 bytes, 3072 bytes, 4092 bytes, 5120 bytes, 6140 bytes, 7168 bytes, 8192 bytes)

$ccncatchunks2 ccnx:/Name > ~/Folder2/FileName

The symbol " >" is used to save contents in Folder2 (created before starting the test on PC1) with FileName arbitrarily chosen.

For each file sent we recorded the transfer time Δt (seconds) and I calculated the throughput (Mbit/s). All data used and recorded are shown in the following table.
### Table 4 – CCNx throughput

<table>
<thead>
<tr>
<th>File N°</th>
<th>File_Size (MB)</th>
<th>Chunk_size (bytes)</th>
<th>∆t (sec)</th>
<th>Throughput (Mbit/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3,3</td>
<td>1024</td>
<td>2.31821</td>
<td>11,8275</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2048</td>
<td>1.203045</td>
<td>22,8763</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3072</td>
<td>0.832582</td>
<td>32,8236</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4092</td>
<td>0.656998</td>
<td>41,5957</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5120</td>
<td>0.510131</td>
<td>53,5712</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6140</td>
<td>0.444845</td>
<td>61,4334</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7168</td>
<td>0.397538</td>
<td>68,7439</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8192</td>
<td>0.368897</td>
<td>74,0812</td>
</tr>
<tr>
<td>2</td>
<td>6,3</td>
<td>1024</td>
<td>4.3938</td>
<td>11,9413</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2048</td>
<td>2.236366</td>
<td>23,461176</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3072</td>
<td>1.575316</td>
<td>33,30618</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4092</td>
<td>1.191417</td>
<td>44,0381</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5120</td>
<td>0.938329</td>
<td>55,9162</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6140</td>
<td>0.824158</td>
<td>63,6599</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7168</td>
<td>0.751818</td>
<td>69,7879</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8192</td>
<td>0.661471</td>
<td>79,3198</td>
</tr>
<tr>
<td>3</td>
<td>19,5</td>
<td>1024</td>
<td>13,56055</td>
<td>12,09169</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2048</td>
<td>6,95934</td>
<td>23,4479</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3072</td>
<td>4,770471</td>
<td>34,3718</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4092</td>
<td>3,67321</td>
<td>44,6395</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5120</td>
<td>2,97726</td>
<td>55,0742</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6140</td>
<td>2,1445871</td>
<td>61,720944</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7168</td>
<td>2,28482</td>
<td>71,7821</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8192</td>
<td>2,00534</td>
<td>81,7668</td>
</tr>
<tr>
<td>4</td>
<td>50,5</td>
<td>1024</td>
<td>34,9455</td>
<td>12,1178</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2048</td>
<td>18,500</td>
<td>22,8898</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3072</td>
<td>12,30425</td>
<td>34,4158</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4092</td>
<td>9,53245</td>
<td>44,4231</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5120</td>
<td>7,13392</td>
<td>59,3589</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6140</td>
<td>6,91143</td>
<td>61,2691</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7168</td>
<td>6,22845</td>
<td>67,9877</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8192</td>
<td>5,65667</td>
<td>74,8605</td>
</tr>
<tr>
<td>5</td>
<td>141,9</td>
<td>1024</td>
<td>100,24332</td>
<td>12,201</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2048</td>
<td>47,32126</td>
<td>25,1465</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3072</td>
<td>36,0391</td>
<td>33,0188</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4092</td>
<td>26,86763</td>
<td>44,2899</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5120</td>
<td>25,63867</td>
<td>46,4129</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6140</td>
<td>20,95773</td>
<td>56,7793</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7168</td>
<td>19,22791</td>
<td>61,8874</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8192</td>
<td>17,453314</td>
<td>66,18</td>
</tr>
</tbody>
</table>

We plotted the results in two graphs. The first one (Figure 10) represents the throughput obtained varying the FileSize (x-axis) for CCNx and Proftpd (with TCP protocol). The second one (Figure 11) represents the throughput varying the ChunkSize for CCNx.

In Figure 10 we can note the great difference between CCNx and TCP when transferring contents. The TCP throughput ranges from 90 Mbit/s to 93 Mbit/s. With CCNx, the throughput obtained is very low mostly when we used the default ChunkSize 1024 bytes. The throughput value is around 12 Mbit/s. This is due to many chunks produced with small size, and we know that in CCN networks, every segment is self-contained: CCNx Content Object messages are self-identifying and self-authenticating, i.e. every segment is named,
encrypted or signed and contains all information necessary to treat it during its transfer. All these information with a high number of segments increase the overhead. Since CCN is encapsulated in IP/UDP for this test, it has all the overhead of the TCP test plus an additional one for its own headers. Note that ccnsendchunks implementation singularly signs each chunk before sending when the interest is received, which implies high processing load. To speed up the transfer, we used a Blocksize greater than 1024 (e.g. 2048 3072...); this operation has decreased the number of segments produced to be handle and as consequence the total overhead decreased. we achieved a throughput around 33 Mbit/s with BlockSize of 3072 bytes (3K), and 70 Mbit/s with BlockSize of 4092 bytes (4K). Note that for chunk size above the Maximum Transfer Unit of the underlying level 2 network (e.g. 1500 bytes for Ethernet) the chunks will undergo IP fragmentation. Figure 11 shows how CCNx throughput increases with the ChunkSize.

![Figure 10- CCNx and TCP throughput for different file size](image-url)

![Figure 11- CCNx throughput for different Chunk size](image-url)
7.4 Transport issues: fairness among ICN flows and TCP flows

In this section we evaluate useful bit rate that we “expect” from the CONET architecture and compare this performance with the one of the CCN architecture Errore. L’origine riferimento non è stata trovata. and with plain TCP/IP.

CONET congestion control, as well as the CCN one, is receiver-driven TCP (R-TCP). Nevertheless, a main difference between the CCN architecture and the CONET one is that the data unit of the CONET congestion control are the segments (i.e. carrier-packets), while the data unit of CCN congestion control are the chunks; indeed CCN not have the carrier-packets and, if necessary, uses IP fragmentation.

Currently both CONET and CCN [15] implementations are at an early stage and they provide a simple Go-Back-N ARQ rather than a R-TCP. Consequently, in order to evaluate what would be the expected performance of these two architectures in the case of a proper implementation of a R-TCP, we resort to the following approach: since a R-TCP behaves as a source-driven TCP in case they use same algorithms to control the congestion window [14], than we measure the CONET and CCN expected performance by using a plain source-driven TCP Reno, which data-units have a length equal to: the chunk in case of CCN, and the segment in case of CONET. Furthermore, the obtained results are properly scaled to account for the CONET and CCN overheads.

The experiment setup consists of two Linux PC connected by a 100 Mbit/sec Ethernet. A PC is the source of data and other PC is the sink. We assumed that the size of named-data-CIU header and the header of the CCN data-packet is 650 bytes², the length of a segment is 1428 bytes, the length of the IP CONET option is 24 bytes, the length of path-info field is 28 bytes, i.e. 7 IPv4 addresses³, the size of maximum congestion window is equal to the default 64 kB.

In what follow we describes our experiments. The expected performance of CCN and CONET evaluated as previously described will be indicated respectively with the term eCCN and eCONET. We use the acronym “e” to stress that these are not the performance of current CCN and CONET implementations but they are what we expect to be obtainable in case of a receiver-driven TCP Reno congestion control.

² We observe a CONET named-data CIU is quite equivalent to a CCN data-packet and current CCN data-packet has about 650 bytes of overhead.

³ We consider the long-term scenario where a CSS coincides with an Autonomous System. In [16] the author show that the average number of hops in the current Internet AS graph is about 3. Therefore, the number average of border nodes crossed by a Interest CIU is 6 and we have on average 7 IPv4 addresses in the path-info, since this field also include the IP address of the end-node.
7.4.1 Analysis of goodput versus packet loss probability

Figure 12 reports the application goodput versus the IP packet loss probability in case of different size of the data chunk, i.e. 8kbyte, 2 kbyte, 512 byte. Packet loss is a Bernoulli random process. As a benchmark, we also report the performance of a plain TCP/IP Reno session.

Considering the lossless case, we observe that we have a performance quite close to the current TCP Reno when using chunk sizes greater or equal to 8 kbytes; for smaller chunk sizes, the impact of protocol overhead (650 byte per chunk) is as high as to reduce valuably the application level goodput. We also note CONET provides a bit lower performance in the lossless case because the present IP CONET option is anyway an additional overhead respect to the CCN.

At the increase of packet loss probability the CCN goodput dramatically suffers for packet loss (e.g. see the 8 kbyte case). This critical behavior is motivated by two facts:

- the CCN congestion control operates with large data-unit (e.g. 8 kbytes + 650 bytes) that are fragmented by IP in several IP packets; so the loss probability at the congestion control level is greater than one at IP level;

- the maximum congestion window is quite small measured in terms of data-units, so fast retransmit and fast recovery mechanisms are less effective.

Those two issues are solved by CONET carrier-packets; indeed in this case CONET congestion control data-units are similar with the plain TCP Reno segments and we have a performance close to the TCP Reno one. Albeit not reported we observe that in case of chunk size of 64 kbytes and greater the CONET performance with carrier packet are very close with the TCP Reno one.
7.4.2 Coexistence of content-centric flows and plain TCP flows

We expect that during the roll out of a information-centric technology there will be a very small number of information-centric data flows that coexist with a huge number of plain TCP/IP flows. For this reason in this section we investigate the performance of a CCN and a CONET flow when they coexist with several TCP/IP streams.

In Figure 13 we report the expected IP goodput (i.e. the bit-rate on top of the IP layer), in case of a CCN flow that coexists with 10 TCP/IP flows. We use the IP goodput rather than the application goodput because it better highlights how TCP/IP and CCN share the capacity of a network link. In all the experiments we measure a packet loss probability close to $10^{-2}$. We observe that when we use small chunks, the IP goodputs are comparable and TCP/IP and CCN flow fairly share the link capacity. Conversely, when we use large chunks, CCN critically suffer the packet loss (see Figure 12) and TCP/IP flows tends to starve the CCN one.

Albeit not reported, we also repeated the test by using a CONET flow and the CONET flow obtained the same performance of a TCP/IP flow, e.g. about 9 Mbit/sec, independently of the chunk size. Indeed, the chunk size in CONET has only an impact on the protocol overhead, but not changes the congestion control performance, since CONET congestion control operates on segments rather than chunks.

We conclude the section remarking that both in CCN and CONET we have to use large chunks to be competitive with TCP/IP. In this case, a congestion control based on chunks like the CCN one critically suffers the packet loss. Consequently, in a scenario where a lot of TCP/IP flows coexist with a CCN flow, the TCP/IP flow may starve CCN flows.
Figure 14- Expected raw IP goodput in case of coexistence of a CONET session without carrier-packets and 10 TCP/IP sessions

7.4.3 Timescale of congestion control

The previous analysis has shown that that in an evolutionary deployment of CONET technology the use of carrier-packets is necessary to coexist with TCP/IP flows. Finally we investigated if we could avoid to use carrier-packets in at least in cases where there isn’t TCP/IP flows at all. Nevertheless also in this cases we reveal that, without carrier-packet the large size of congestion control data units stretches the timescale of congestion control and produces valuable oscillation of the goodput. We observed this phenomenon by starting 10 CONET flows at the same time. Figure 14 reports the raw IP goodput in case of using and not using carrier-packets. We see that, albeit on the average the performances are the same, in absence of carrier-packets we have a very higher variability of the goodput while in case of carrier-packets the goodput is definitively more stable. In general, it is better to have a stable goodput rather than a variable one. For instance, in case of not-real time streaming video services like the one currently offered by youtube via TCP/IP, this variability requires very large dejitter buffer so penalizing the quality of experience.

7.5 Transport level efficiency

7.5.1 Static overhead evaluation

Let us first evaluate the maximum achievable goodput for CCNx transport and coCONET transport protocol.
coCONET and CCNx overhead are function of the name and the chunk number, we take an average sample considering the same content used for next measures.

The overheads up to UDP transport layer is composed by:

- \( OH_{eth} = 12B[IFS] + 26B[OH_{eth,frags}] \)
- \( OH_{ip} = 20B \)
- \( OH_{udp} = 8B \)

In CCNx using the standard ccn_repo size of 4096B for each chunk and with the ccnb encoding overhead the amount of byte to be transmitted increases to:

\[ \text{ChunkLen}_{\text{CCNx}} = 4096B + OH_{ccnb} = 4720B \]

For coCONET we add an overhead of \( OH_{ccnct} = 61B \) for each segment of the chunk.

In both cases a 4096 Bytes chunk is split into 3 IP packets: in CCNx we have the IP fragmentation of an UDP packet, in case of CONET we have 3 UDP packets, each one sent into an IP packet.

We can evaluate the total number of bytes that are transmitted for each chunk:

- CCNx: \( 3 \times (OH_{eth} + OH_{ip}) + OH_{udp} + \text{ChunkLen}_{\text{CCNx}} = 4902B \)
- coCONET: \( 3 \times (OH_{eth} + OH_{ip}) + 3 \times (OH_{udp} + OH_{ccnct}) + \text{ChunkLen}_{\text{CCNx}} = 5101B \)

Therefore, we can evaluate the maximum frame rate as:

- CCNx: \( 100 \text{Mbps}/(4902 \times 8) = 2550 \text{ frame/sec} \)
- coCONET: \( 100 \text{Mbps}/(5101 \times 8) = 2450 \text{ frame/sec} \)

Hence we can calculate the goodput of each implementation in Mbps:

- CCNx: \( 2550 \text{ frame/s} \times (4096B \times 8) = 83,550 \text{ Mbps} \)
- coCONET: \( 2450 \text{ frame/s} \times (4096B \times 8) = 99,200 \text{ Mbps} \)

This result is expected, as in coCONET the overhead is slightly higher due to segmentation overhead incurred in each fragment. The advantage of coCONET is that the different segments are sent in different IP packets rather than in IP fragments. This is a big advantage in case of IP packet loss.
7.5.2 Performance measurements.

We conducted performance tests with the following configuration:

- A pc running Linux acts as serving node executing ccnd and hosting the content loaded via the ccn repository (using java tools ccn_repo and ccnputfile).
- A pc running Linux acts as a client node which run ccnd and ccncatchunks2 to retrieve the files.

Both PCs are connected via an ethernet 100Mb/s switch.

Both CCNx and coCONET configurations were fine-tuned to achieve best performance. Main reasons of tuning is due to the reaching of the limit of UDP receive buffer size that causes a huge burst of loss and therefore does not let the recovery mechanisms of transport protocols to run smoothly. In this conditions the efficiency is drastically limited.

- For CCNx we manipulate the pipeline option (-p option in command line) of ccncatchunks2. The pipeline parameter is set to 20.
- For CONET we manipulate the threshold used to switch in the congestion avoidance mechanism. The threshold is set to 150 segments.

We also increase the udp receive buffer sizes to be sure to avoid unexpected UDP receive buffer overflows.

The measures below shows the average goodput of 10 runs in a network without loss for various file dimensions.

The configuration reported are CCNx, CONET and CONET with pre-fetch option, the goodput is in Mbps.

<table>
<thead>
<tr>
<th>File size</th>
<th>CCNx</th>
<th>coCONET</th>
<th>coCONET+PREFETCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 Kbyte</td>
<td>23.45 Mbps</td>
<td>31.69 Mbps</td>
<td>41.35 Mbps</td>
</tr>
<tr>
<td>1Mbyte</td>
<td>56.71 Mbps</td>
<td>55.00 Mbps</td>
<td>57.26 Mbps</td>
</tr>
<tr>
<td>10Mbyte</td>
<td>79.59 Mbps</td>
<td>70.38 Mbps</td>
<td>72.31 Mbps</td>
</tr>
</tbody>
</table>

From these preliminary results we can see that our implementation is slightly slower than CCNx in a steady state (10 Mbyte file) in a context without loss, but half of the difference is due to the greater overhead of coCONET transport protocol, highlighted in the goodput evaluation before.

A great improvement of performance can be seen with little file for coCONET with prefetch option, this is due to the prefetch option that allow the requester to pre-download the entire contents disregarding application requests. In fact without the option the behavior of CONET transport protocol is similar to CCNx.
Finally, we tested the efficiency of both transport protocols in a context of loss. For this test we used a 10MB file and we setup a loss in the network with linux traffic control (tc) on the receiving node.

<table>
<thead>
<tr>
<th>Loss %</th>
<th>CCNX</th>
<th>CONET</th>
<th>CONET+PREFETCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1%</td>
<td>31.28 Mbps</td>
<td>63.65 Mbps</td>
<td>66.50 Mbps</td>
</tr>
<tr>
<td>0.5%</td>
<td>14.93 Mbps</td>
<td>21.67 Mbps</td>
<td>29.45 Mbps</td>
</tr>
<tr>
<td>1%</td>
<td>12.63 Mbps</td>
<td>17.67 Mbps</td>
<td>26.97 Mbps</td>
</tr>
</tbody>
</table>

These results show clearly what we expected, in a context with loss, CCNx performance are much worse than coCONET ones.

These results can be explained with some considerations:

– The implementation of a receiver-driven TCP-like congestion control with the use of mechanisms like fast-recovery and fast-retransmit allow coCONET to be faster than CCNx which uses in this test a simple receiver driven selective retransmission.

– The small size of segment data unit used in coCONET limits each loss to only one segment of a chunk, the chunk size of CCNx causes the loss of a complete chunk with a loss of only an IP packet.

Further measures will regard efficiency in a context of loss for various size of chunk and various loss probability and how the two protocols react to an increase or decrease of the chunk size.

Note also that at this time, coCONET and CCNx solution are both in early stages of development, so some results can be unexpected or unpredictable due to bugs or performance issues.
Acknowledgements

The authors would like to thank:
- Cedric Teufack for his contribution on performance measurements
References


[15] CCNx project web site: www.ccnx.org
