CCNx for Contiki: implementation details

Bilel Saadallah, Abdelkader Lahmadi, Olivier Festor

CCNx for Contiki: implementation details

Bilel Saadallah, Abdelkader Lahmadi, Olivier Festor

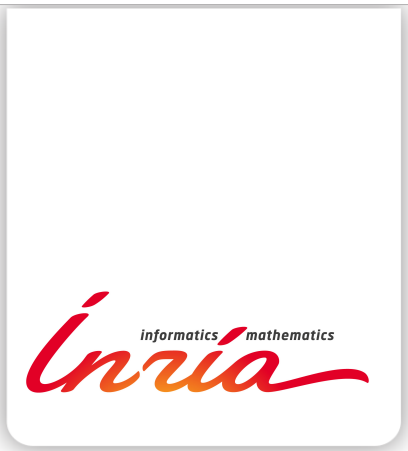
TECHNICAL REPORT

N° 432

November 2012

Project-Team MADYNES

ISSN 0249-0803 ISRN INRIA/RT--432--FR+ENG



CCNx for Contiki: implementation details

Bilel Saadallah*, Abdelkader Lahmadi†, Olivier Festor‡

Project-Team MADYNES

Technical Report n° 432 — November 2012 — 49 pages

Abstract: In this work we adapted the Content Centric Networking (CCN) approach which is a promising communication architecture based on named data to be employed in wireless sensor networks. We implemented and integrated a CCN communication stack into Contiki which is an operating system for resources-constrained embedded systems and wireless sensor networks. We evaluated our implementation using a synthetic monitoring application under varying network sizes through simulation and a real deployment on the SensLAB testbed.

Key-words: Wireless Sensor Networks, Content Centric Networking, CCNx, Contiki

* bilel.saadallah@inria.fr
† abdelkader.lahmadi@loria.fr
‡ olivier.festor@inria.fr

RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

CCNx pour Contiki: détails d'implantation

Résumé : Dans ce travail nous avons adapté l'approche basée sur les réseaux orientés contenus (CCN) pour être utilisée dans les réseau de capteurs sans fil. Nous avons conçu et implémenté une nouvelle couche de communication CCN qui se base sur l'utilisation de données nommées. Cette couche est intégrée dans Contiki qui est un système d'exploitation dédié aux réseaux de capteurs sans fil et aux systèmes embarqués. Nous avons évalué les performances de notre implantation par simulation et sur la plateforme SensLAB.

Mots-clés : Réseaux de capteurs sans fil, Content Centric Networking, CCNx, Contiki

Contents

List of Figures

List of Tables

Introduction

The Internet of Things (IoT), a part of the Future Internet, conceives the world as a global network of interconnected objects where objects are seen as proactive participants in information, business and social processes. These objects have the ability to generate and exchange data among themselves and to interact with their surrounding environment.

Wireless sensors networks (WSNs) are considered as a promising element of the IoT. Their capacities of monitoring the environments where they are deployed and collecting information is completely compliant with the IoT expectations. WSNs have found their way into a variety of applications in different domains such as healthcare, environmental monitoring, production control and smart building monitoring. In addition to their applicative boom, they are witnessing an increasing interest from the research community. WSNs are at center of many research activities aiming at enhancing their performances, autonomy and communication efficiency.

This work comes to participate in the efforts to improve and evaluate the communication models of WSNs. Since Named Data Networking (NDN), or more generally Information-Centric Networking (ICT) became one of the significant directions of current networking research, it is challenging to see if such information-based communication models can fit WSNs characteristics and application requirements. Content Centric Networking (CCN) [16] is an emerging communication architecture that is based on named data. Information is exchanged only in response to a request specifying the name of the data to retrieve. CCN forms a data-polling communication model which seems adapted for the functioning of a WSN considering the fact that its main use is to gather data. In a data-driven network, a query instructs each node to sense its environment at a certain rate, for a period of time, and deliver matching data back to the sink.

In this work we provided a content centric communication layer adapted to the specificities of wireless sensor networks. We focus on one particular challenge: the compliance between the Content Centric Networking approach and wireless sensor networks. In order to achieve this, we implemented a CCN communication stack which was later integrated into Contiki, an operating system for memory-constrained embedded systems and wireless sensor networks.

This report is organized as follows. In section 1, we briefly present the state of the art related to the CCN communication approach and gives an overview of the CCNx protocol in particular. We presented as well the Contiki operating system and we provide details about its programming style using protothreads. In section 2, the design, the implementation and the integration of the CCN communication layer are described. This layer is then tested and validated using simulations and experiments on physical sensor nodes. Section 3 presents the performance evaluation of our implementation and the analysis of the obtained results. Finally, we present the limits of the current version of our CCN layer and a set of insights and recommendations for its enhancement.

1 Background

In this section, we describe the CCN communication architecture and the CCNx protocol. Then we present the Contiki operating system and its programming style.

1.1 CCN approach

Networks were mainly created to solve the problem of resource sharing. The resultant communication model developed into a conversation between two machines, with one machine requesting data and another one giving access to data. In the IP architecture, communication is made between hosts identified by addresses. It is built on a permanent mapping of content to host locations.

But the use and the properties of present networks are different from those of the first networks. They are witnessing a big expansion in term of size and data-contents. A study, conducted in 2007, announced that between 2006 and 2010, the information added annually to the digital universe would increase more than six fold from 161 exabytes to 988 exabytes (1 exabyte is 1 billion gigabytes)[15]. For such data-growing networks, it is more convenient to focus on data itself rather than data location.

Content Centric Networking (CCN) is a communication architecture built on named data[16] where the identification and the transport of contents rely on their names and not on their location. The CCN approach addresses the issues limiting the current use of networks by increasing the availability of data. It provides caching to reduce congestion and improve delivery speed. In term of security, CCN suggests that trust in content is easily misplaced when relying on data locations. Instead, it builds security into the network at the level of data. In addition, because the communication relies only on data names, no mapping between contents and locations is done. Thus, the configuration of network devices is much simpler.

1.2 CCNx

The CCNx protocol is a transport protocol for the Content-Centric Networking communication architecture. According to the CCN specifications [3], it is built on named data where the content name replaces the location address. The CCNx protocol provides location-independent delivery services for named data packets. The services include multihop forwarding for end-to-end delivery, flow control, transparent and automatic multicast delivery using buffer storage available in the network, loop-free multipath forwarding, verification of content's integrity regardless of delivery path, and carriage of arbitrary application data[1].

Applications use the CCNx protocol on top of a lower-layer communication service that can handle packet transmitting. No restrictions are imposed on the nature of the lower-layer service. It may be a physical transport, another network or a traditional transport protocol. Although the CCNx protocol is designed to deliver contents based on their names, applications can run it on top of UDP or TCP to take advantage of existing IP connectivity. Since content is named independently of location in the CCNx protocol, it may also be preserved indefinitely in the network. Every packet of data may be cached at any CCNx router. Providing support for multicast or broadcast delivery, the network's use is more efficient when many people are interested in the same content.

The CCNx protocol supports a wide range of network applications by leaving the choice of naming conventions to the application. It may be natural to think of stored content applications such as distribution of video or document files, but the CCNx model also supports real-time communication and discovery protocols and it is general enough to carry conversations between hosts such as TCP connections.

The flexibility of the CCNx protocol to be deployed in different environments, mainly where providing data content is of primary focus, we take the initiative to apply the Content-Centric Networking architecture in embedded systems, and sensor networks particularly. In this work, we adapted the existing CCNx implementation (version 0.3.0) to provide a communication layer based entirely on named data, where the notion of host identification-by-location is discarded in wireless sensor networks. Some parts of CCNx code sources are reused, others are simplified or adjusted to meet the specificities of sensor networks.

1.3 Contiki

Contiki [5] is an open source operating system for memory-constrained embedded systems and wireless sensor networks. It is highly portable and ported to more than twelve different microprocessor and microcontroller architectures. Contiki is designed for microcontrollers with a very limited memory size. A typical Contiki configuration requires 2 kilobytes of RAM and 40 kilobytes of ROM.

Both the Contiki system and applications for the system are implemented in the C programming language. Contiki consists of an event-driven kernel, on top of which application programs can be dynamically loaded and unloaded at run time. Contiki processes use lightweight protothreads that provide a linear, thread-like programming style on top of the event-driven kernel. In addition to protothreads, Contiki also supports per-process optional preemptive multi-threading [12] and interprocess communication using message passing.

Contiki provides IP communication, both for IPv4 and IPv6, through uIP and uIPv6 stacks [10]. Its IP stack allows it to communicate directly with other IP-based applications and web services, including Internet-based services. Contiki supports many other protocols and mechanisms like 6LoWPAN header compression, RPL routing and CoAP application layer protocol. It has also a low-power radio networking stack called Rime that can be used for sensor network communication[13]. The Rime protocol stack provides a set of communication primitives, ranging from best-effort local neighbour broadcast and reliable local neighbour unicast, to best-effort network flooding and hop-by-hop reliable multi-hop unicast. Applications or protocols running on top of the Rime stack may use one or more of the communication primitives provided by this stack.

One of the crucial tasks in sensor networks development is to control and reduce the power consumption of each sensor node to ensure a longer sensor network lifetime. Contiki provides a software-based power profiling mechanism that keeps track of the energy expenditure of each sensor node. Being software-based, the mechanism allows power profiling at the network scale without any additional hardware. Contiki's power profiling mechanism can be used for research purposes like the evaluation of sensor network protocols or as a way to estimate the lifetime of a network of sensors.

Different ways of interaction with a network of Contiki sensors are available using a web browser,

a text-based shell interface or dedicated software that stores and displays collected sensor data. The text-based shell interface is inspired by the Unix command shell but provides special commands for sensor network interaction and sensing.

Contiki can store data inside a sensor node using Coffee, a flash-based file system. The file system allows multiple files to coexist on the same physical on-board flash memory and has a performance that is close to the raw data throughput of the flash chip.

Contiki provides three simulation environments: the MSPsim emulator, the Cooja cross-layer network simulator, and the Netsim process-level simulator. They can help in the development and debugging of a software before testing it on the hardware.

Contiki's programming style and protothreads : The programming model influences the structure and performance of the software developed based on that model. Two models are particularly used : multi-threaded programming model and event-driven programming model.

Multi-threading is a programming technique that allows multiple programs to run at the same time on a single processor [18]. In multi-threaded programming, each program has its own thread. Different threads can run alongside each others in the system and thus share the same microprocessor. The operating system coordinates the execution of the threads to give each thread time to run on the microprocessor.

The benefit of multi-threading is that it prevents a thread from taking over all the computing resources of the CPU and blocking the other waiting threads. It permits equally to take advantage of the unused computing resources by managing threads in a way to not leave the CPU idle.

The drawback of the multi-threaded programming is that each thread needs its own piece of memory to hold the state of the thread, the so-called stack of the thread. The thread keeps in its stack the set of local variables and return values of the functions it called. The problem is that the amount of stack memory a thread needs is unknown in advance, so the allocated stack memory is typically over sized with a comparatively large amount of unused memory.

For memory-constrained embedded systems, including sensor networks, the event-driven programming model is considered a memory-efficient model and thus is often privileged to other models. It requires less memory than multi-threaded programming because there are no threads that require stack memory. The entire system can run as a single thread, which requires only one single stack.

With the event-driven programming, the program is composed of event handlers. An event handler is a short section of code that describes how the system responds to events. When an event occurs, the system executes the corresponding event handler.

The Contiki operating system uses a programming model based on Protothreads [14]. Protothreads are a programming abstraction that provides a conditional blocking wait operation, that is intended to simplify event-driven programming for memory-constrained embedded systems.

Protothreads can be seen as a combination of events and threads. From threads, protothreads

have inherited the blocking wait semantics. From events, protothreads have inherited the stacklessness and the low memory overhead. The blocking wait semantics allow linear sequencing of statements in event-driven programs [14]. The main advantage of protothreads over traditional threads is that protothreads are very lightweight. A protothread is stackless: it does not keep track of function invocations. Instead, all protothreads in a system run on the same stack, which is rewound every time a protothread blocks. This is advantageous in memory constrained systems, where a thread's stack might use a large part of the available memory.

In Contiki, processes are implemented as protothreads running on top of the event-driven Contiki kernel. When a process receives an event, the corresponding protothread is invoked. The event may be a message from another process, a timer event, a notification of sensor input, or any other type of event in the system. Processes may wait for incoming events using the protothread conditional blocking statements [14].

Protothreads can be efficiently implemented in the C programming language without any assembly language or changes to the compiler. The drawback is that programmers must explicitly store variables when protothreads block. As protothreads are implemented in C, they are very portable across different platforms [18].

Ultimately, the choice of programming model is up to the software designer. For this reason, Contiki uses a hybrid model: the system is based on an event-driven kernel where preemptive multi-threading is implemented as an application library that is optionally linked with programs that explicitly require it [12]. In this way, Contiki combine the benefits of the two programming models.

2 Implementation details

This section presents the details of the CCN communication layer design, implementation and integration in Contiki. In the first part, we describe the designed CCN communication model and the adaptations made to meet with the WSN specifications. The second part presents the technical details of the implementation. The third part shows how the implemented CCN communication module is integrated into the Contiki OS.

2.1 CCN adaptation and extension

2.1.1 Architecture overview

The main purpose of this work is to design and implement a CCN communication layer based on named data. This layer is integrated in Contiki, an operating system for networking embedded systems and wireless sensor networks, as presented in Fig. 1.
To meet Contiki's implementation style, the CCN communication layer is composed of a Stack and a Driver. The Stack implements the CCN processing, forwarding and caching functions and manage event posting to processes. The Driver handles messages exchange with the lower layer.

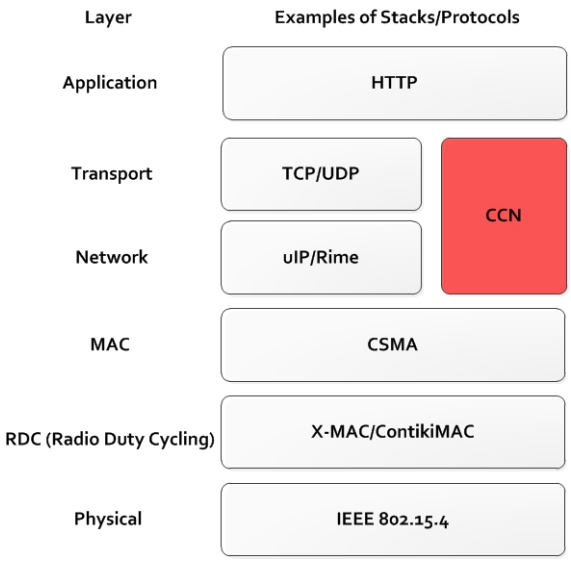


Figure 1: The integration of the CCN layer into Contiki system architecture.

The CCN communication layer handles packets transmission and does not rely on other transport protocols to deliver messages. It uses directly the MAC layer available in the Contiki to transmit its messages. It is built according to a full CCN-compliant communication model that relies entirely and uniquely on named data.

2.1.2 CCN messages

Message types CCN communication is driven by the consumers of data. There are two CCN message types: Interest message and Content Object message.
The Interest message is a request of named data. It contain the full name that identifies a piece

of content. This content will be specifically retrieved if its available in a node of the network. It can also contain simply a prefix of the content name. Then any content whose name matches the Interest name prefix can be a potential response to this Interest.
The Content Object message is used to supply data. A Content Object message contains a data payload preceded by the identifying name.

Message formats The CCN communication layer is integrated in Contiki which relies on the IEEE 802.15.4 standard specifications to implement its physical and MAC layers. The size of a 802.15.4 frame is limited to 127 bytes, with 72-116 bytes of payload available after link-layer framing, depending on a number of addressing and security options. In this work, we fix the maximal size of a CCN packet or message to 102 bytes (127 bytes - 25 bytes of a MAC header). In order to provide a simple CCN communication model without a fragmentation mechanism, the message formats are simplified to leave more payload bytes for names and data carrying. Based on the messages scheme defined by the CCNx protocol [3], all the optional fields are discarding from the CCN messages provided by our implementation. An Interest message contains only the prefix of the content to be retrieve, while a Content Object message is composed of two elements. As depicted in Figure 2 The first element contains the content name and the second element is reserved for data .

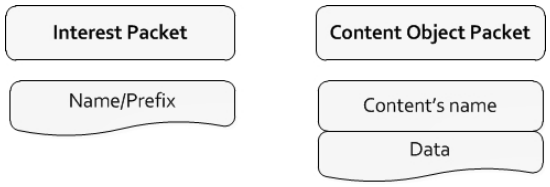


Figure 2: CCN messages format.

Messages exchange A consumer asks for content by broadcasting its Interest message over all available connectivity. Any node receiving the message and having data that matches the request may transmit a matching Content Object message. Data must only be transmitted in response to an Interest that matches the Data.
A node has to transmit at most one Content Object message in response to a single received Interest message, even if the node has many matching Content Objects. This one-for-one mapping between Interest and Content messages maintains a flow balance that allows the receiver to control the rate at which data is transmitted from a sender. It also helps to avoid consuming the bandwidth by sending data where it is not requested.
To provide a reliable delivery, Interest messages that are not satisfied in some reasonable period of time must be retransmitted. A receiver has to maintain a timer on unsatisfied Interests and retransmit them when the timer expires.

2.1.3 CCN names

In the Content Centric Networking approach, the Name element plays a pivotal role in the communication. CCN relies on the use of names to deliver messages and exchange contents between the network nodes. A CCN name can identify a specific chunk of data. It can also identify a collection of data in the case where this name is a prefix of the name of every piece of content in the collection. This would explain why a CCN name may be refereed to as a name

prefix or simply a prefix. In the example given in 3, the prefix /Temperature/Inria identifies the set of contents indicating the temperatures in the offices 132, 214 and 125 in the Inria building.

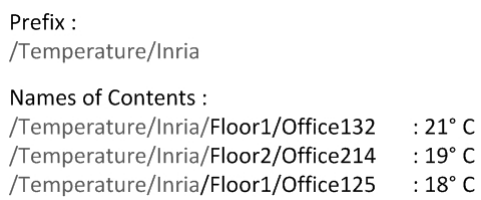


Figure 3: An example of CCN naming.

A CCN name is a hierarchical name attributed to a content. It simply contains a sequence of components of arbitrary lengths. There are no restrictions on what byte sequences may be used. The implemented communication layer specifies only the name structure and does not assign any meanings to names. It is up to applications or global naming conventions to set and interpret meanings given to names. Application developers are free to design their own custom naming conventions.

To represent names, we refer to URI scheme (RFC 3986 : URI Generic Syntax)[9]. A Uniform Resource Identifier is a sequence of characters identifying a physical or abstract resource. The URI syntax is organized hierarchically, with components listed in order of decreasing significance from left to right. A URI is composed from a limited set of characters consisting of digits, letters, and a few graphic symbols. A reserved subset of those characters may be used to delimit syntax components within a URI while the remaining characters, including both the unreserved set and those reserved characters not acting as delimiters, define each component's identifying data. When producing a URI from a CCN name, only the generic URI unreserved characters are left unescaped. These are the US-ASCII upper and lower case letters (A - Z, a - z), digits (0 - 9), and the four specials period (.), underscore (_), tilde (~) and hyphen (-). All other characters are escaped using the percent-encoding method of the URI Generic Syntax.

A percent-encoding mechanism is used to represent a data octet in a component when that octet's corresponding character is outside the allowed set or is being used as a delimiter of, or within, the component. A percent-encoded octet is encoded as a character triplet, consisting of the percent character "%" followed by the two hexadecimal digits representing that octet's numeric value. For example, "%20" is the percent-encoding for the binary octet "00100000", which in US-ASCII corresponds to the space character (SP).

URIs include components and subcomponents that are delimited by characters in the reserved set. These characters are called reserved because they may (or may not) be defined as delimiters by the generic syntax, by each scheme-specific syntax, or by the implementation-specific syntax of a URI's dereferencing algorithm. In the CCN implemented syntax, a limited set of delimiters is held including the characters "/", "?" and "#". The other reserved characters are not supported and cause parse errors. These characters are the following: ":" / "[" / "]" / "@" / "!" / "$" / "&" / "'" / "(" / ")" / "*" / "+" / "," / ";" / "=". In our implementation, we use the slash ("/") character to delimit name components, allowing easier name parsing and comparison.
To unambiguously represent name components that would collide with the use of "." and ".." for relative URIs, any component that consists solely of zero or more periods is encoded using three additional periods. Note that this means the zero-length component is encoded as three

periods "...".

2.1.4 Node model

According to the specifications of the Content Centric Networking approach, a node requires the following data structures to provide buffering/caching of data, manage content requests and forward messages to other nodes in the network.

Face : A face is a generalization of the concept of interface. In the CCNx specifications, a face may be a connection to a network or directly to an application party. It may be configured to send and receive broadcast or multicast packets on a particular network interface, or to send and receive packets using point-to-point addressing in the underlying transport, or using a tunnel.
In sensor networks, a node is usually containing a single network interface controller. At first, we made the choice to create at each node a single face reserved for broadcast to communicate with other nodes. This would be convenient when a consumer, which wants to retrieve a piece of content, asks for it by broadcasting its Interest message to all nodes within its communication range. Then, we added a second face to which we attributed permanently the identifier 0 in order to differentiate the Interest messages initiated by a node (the node is the first source of the request) from the other Interests messages relayed by this node.
It could be also interesting to create a face for each node in the neighbourhood. In this way, a node can address its message to a specific neighbour like in a unicast communication. However, we needed to find how to identify a neighbour node. Linking a face to the link layer address of a node may seem as turning back to the IP model. A solution of this can be done through naming routines by adding a node identifier and injecting it into name prefixes. In this work, we only implemented a broadcast face.

Content Store (CS) : The Content Store is a cache where data is stored. It holds Content Objects created locally using the data collected by sensor devices and Content Objects received from other nodes. Contents are indexed to facilitate their retrieval and suppression.
The Content Store may retain Content Object messages indefinitely but is not required to take any special measures to preserve them; the CS is a cache not a persistent store. Data in CS is updated when receiving an existing content in the cache but with different value of data.

Forwarding Information Base (FIB) : The FIB is a table of destinations for Interests, organized for retrieval by longest prefix match lookup on names. An entry in FIB can be a prefix that points to a set of destinations rather than a specific one.
FIB plays a role equivalent to the IP routing table as it indicates on which faces a node should send its Interest message to retrieve a matching content. An entry in a FIB table contains a prefix and a list of face identifiers. To retrieve data that matches the prefix, a node has to send an Interest through at least one of the faces whose identifier is included in the list.
In this work, no mechanism or protocol is implemented to fill FIB tables dynamically. It's done statically during the initialization of the application running on top of a node before starting the communication with the other nodes in the network. So the entries in the FIB would not change or expire during the execution of the application.

Pending Interest Table (PIT) : The PIT is a table of sources for unsatisfied Interests. It is organized for retrieval by complete prefix match lookup on names (a match occurs when the interest prefix to compare and the prefix in the PIT entry match completely). Each entry in the PIT may point to a list of identifiers of the faces which are sources of the unsatisfied Interest.

Interests messages propagate in the network toward potential sources of data leaving, in the intermediate nodes, PIT entries as marks of the way a matching data packet has to take back to reach the original requester of this content. In this way, only Interest messages are routed in CCN. When a node disposes of data or receives a content data that matches a pending Interest, the corresponding PIT entry is used to forward a matching Data packet and then erased from the table.
Entries in the PIT are not held indefinitely; they must time out. When pending Interests fail to find a matching content, they become expired and are subsequently deleted from PIT. It's up to the application, or more generally the consumer, to re-express the Interest message if it is still interested in getting the data. Note that the original requester of data which created the Interest message keeps track of its message in its PIT table and uses the face with the identifier 0 to send it.

2.1.5 Communication algorithms

The communication algorithms implemented in this work are quite the same as CCNx algorithms, except some minor changes.

Processing Interest Messages : A received Interest message is processed according to the following steps:

1. Lookup is performed on the Content Store. If the CS has a Content Object whose name matches the prefix in the Interest message, it will transmit the matching Content Object out to the face where the Interest arrived. If the CS holds multiple Content Objects that simultaneously match the received Interest message and as Interests implemented in this work do not offer any specifications more than a prefix, the first matching Content Object found in the lookup on CS is sent in response to the Interest message. When a match is found in the CS, processing stops and the Interest message is discarded since it has been satisfied.

2. Lookup is performed on PIT. If a matching Interest message is found in the PIT it means that an equivalent Interest message has already been forwarded and is pending. If there is an exact-match PIT entry, the Interest's arrival face will be added to the PIT entry's list of requesting faces (list of sources of this unsatisfied Interest) and the Interest will be discarded.

3. Lookup is performed on FIB. If a matching prefix is found in the FIB, an entry is created in the PIT from the Interest message and its arrival face and the message is transmitted to the destination faces registered for the prefix in the FIB, except on the arrival face if it is registered to the matching FIB entry too.

4. If no match is found in the previous steps then the node has no way to satisfy the Interest message and thus discard it.

Therefore, when an Interest packet arrives, a prefix match lookup is done on names. As shown is Figure 4, the lookup is ordered so that a Content Store match will be preferred over a PIT match which will be preferred over a FIB match.
In order to minimize the cost of lookup operations when processing and forwarding CCN messages, FIB and PIT tables are coupled so that PIT entries will be attached to the FIB entry whose prefix offers the longest match. This would result in some modifications to the previous algorithm as any lookup performed on PIT will result in a lookup on FIB prefixes. So to optimize

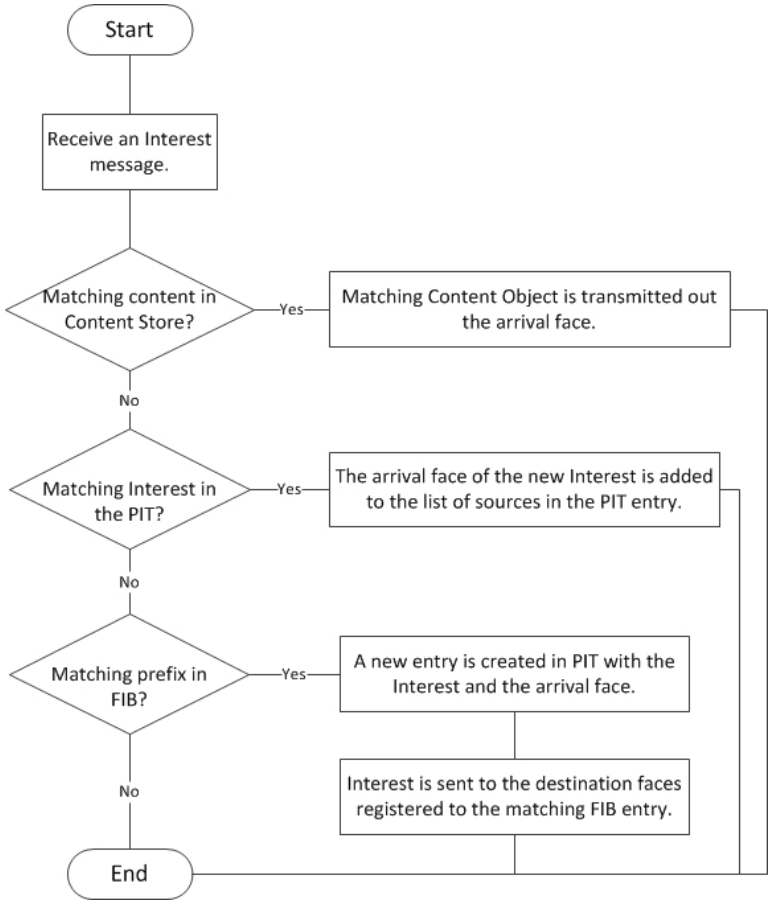


Figure 4: Incoming Interest processing.

the lookup operations, we reversed step two and three as shown in Figure 5. Step one remains unchangeable so the first lookup will be performed on CS. If no matching Content Object is found, a lookup is performed on FIB. If there is no matching entry, we can deduce automatically with the coupling of the two tables that there are no matching PIT entries. The Interest message will be then discarded. If a FIB match is found, we check the PIT entries attached to the matching FIB entry. If a PIT entry match is found, we simply add the arrival face to the list of face identifiers of the matching pending Interest in the PIT and processing is over. Otherwise, if no PIT entry matches the received Interest message, a new entry is added to PIT and attached to the matching FIB entry. The Interest will be then forwarded out the faces registered to the FIB matching entry.

Processing Content Messages : A received Content Object is processed according to the following steps as depicted in Figure 6:

1. Lookup is performed on CS. If a matching Content Object is found it means that the newly arrived Content Object is a duplicate which can safely be discarded, because all previous Interest messages have already been satisfied and new ones will be satisfied out of the CS. In this work, we check the data value of a duplicate Content Object and we use it to update

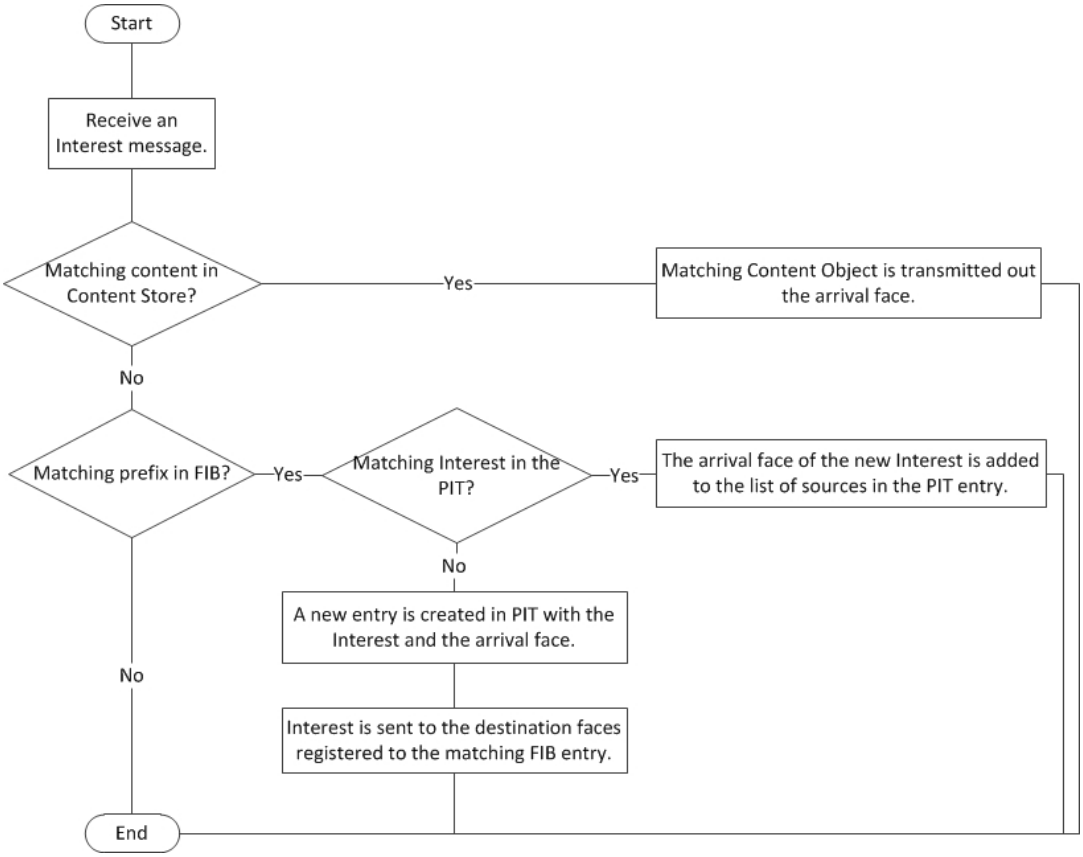


Figure 5: Modified algorithm for incoming Interest processing.

the data stored in Content Store.

2. Lookup is performed on PIT. A PIT match means the data was solicited by one ore more Interests sent by this node. If there is a match in the PIT, the Content Object is transmitted on all of the source faces for the Interests represented in the PIT, except the face with the identifier 0. If a matching PIT entry has the face identifier 0 among its list of source faces, it means that this node is the original requester of the received Content Object. In this case, it delete the reference of the face with the identifier 0 from the list of faces in the matching PIT entry before sending the Content Object on the rest of source faces (if existing). Next, the matching PIT entry is deleted from the table.

3. If no match is found in the previous steps, then the content is unsolicited. A node must not forward unsolicited data and may discard unsolicited data. Here we store unsolicited data it in the Content Store in case it is subsequently requested.

2.1.6 Memory allocation

Contiki supports three different types of memory management:

1. The Malloc() function to dynamically allocate memory. Its use is deprecated.

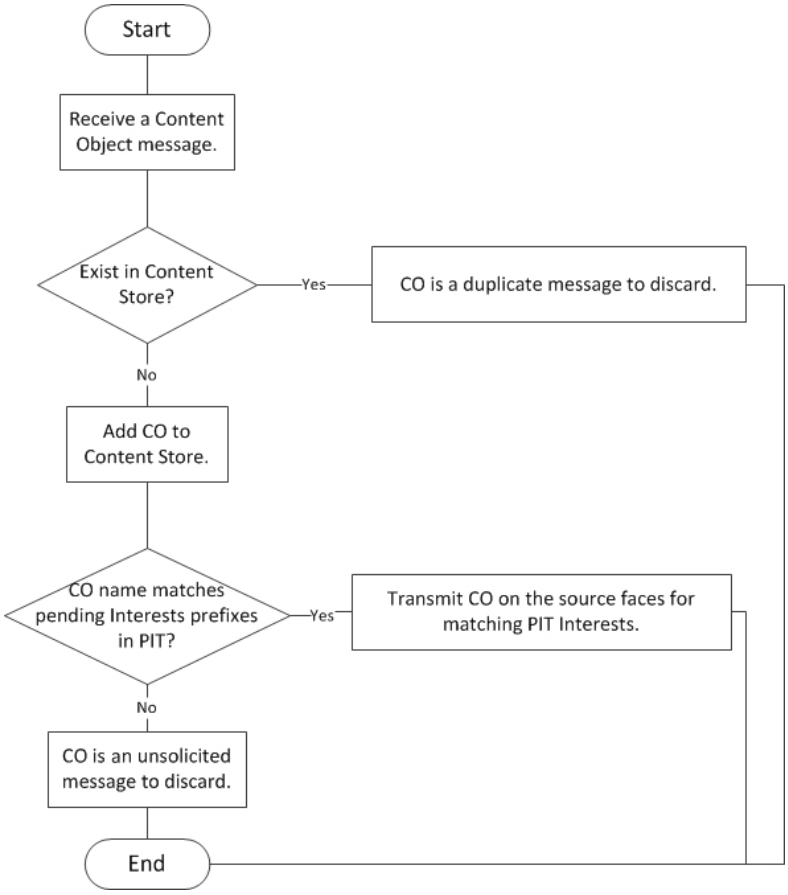


Figure 6: Incoming Content Object processing.

2. The memory block function to allocate a set of memory blocks of fixed size.

3. Managed memory function to allocate a fragmentation-free set of memory blocks.

However, on the Tmote Sky platform, dynamic memory allocation is not allowed, so malloc() cannot be used to allocate memory. Instead, all memory blocks have to be allocated statically. The memory block allocation routines provide a collection of functions for managing a set of memory blocks of fixed size. A set of memory blocks is statically declared with the MEMB() macro. Memory blocks are allocated from the declared memory by the memb_alloc() function, and are deallocated with the memb_free() function.

Our CCN stack uses MEMB() to allocate the different buffers, caches and other data structures to handle the content centric communication and manage names and data.

The CCN stack has only few kilobytes of RAM to use. This impacts the size of caches and name buffers, limiting the number of entries in CS, FIB and PIT tables and the number of buffers to hold names/prefixes. Furthermore, the stack uses one single global buffer to hold packets. The global buffer is large enough to contain a packet of maximum size and it is used to hold both incoming and outgoing packets.
When a packet arrives on the network, the CCN driver places it in the global buffer and calls

the CCN stack to handle it. The stack will apply the necessary processing according to the CCN approach and will notify the corresponding application. Data in the buffer is overwritten every time a packet is received. The application has to either act immediately on data or to place it in another buffer for later processing. In the current CCN stack, no queues are implemented to hold incoming packets or packets to be retransmitted.

2.1.7 Binary encoding/decoding

To encode and decode messages, we relied on existing CCNx encoding mechanism [2]. The message's fields can have values of arbitrary length and boundaries between fields are explicitly identified. The transmission unit is a byte (8 bits) with units of high order transmitted first. The encoding leads to a sequence of encoding units where each unit is a block followed eventually by a value (data). The block has a header composed of a number value, which is a non-negative integer in big-endian representation, and a type TT coded in the 3 low-order bits that allows to interpret the block's type. The block's type TT gives information about the value to follow and its length. An ending delimiter, called also closer, is inserted at the end of the block. The closer is a single byte with the value zero (0x00).
In the CCNx documentation, CCN binary encoding (ccnb) is defined as binary encoding of XML structures. CCNx data formats are defined by XML schemas. This design permits field values of arbitrary length, optional fields that consume no packet space when omitted, and nested structures. As a result, in an encoding unit, a block represents the start of a XML element or attribute, or data character while the ending delimiter represents the close of a XML element.
Different types of blocks are available in CCNx:

- BLOB(TT=5): Arbitrary binary value, arbitrary length (including 0), no alignment requirements. Header number value is length in bytes. Value will beencoded as base 64 when converted to text XML.
- UDATA(TT=6): Same as BLOB but value is restricted to valid UTF-8 sequences.
- TAG(TT=1): XML element tag name in UTF-8 encoding. Header number value is length in octets of UTF-8 encoding of tag name - 1 (minimum tag name length is 1). It may be followed by anything due to XML nesting.
- ATTR(TT=3): XML attribute name in UTF-8 encoding. Header number value is length in octets of UTF-8 encoding of attribute name - 1 (minimum attribute name length is 1). UDATA must follow immediately representing attribute value (though 0 length value may be represented).
- DTAG(TT=2): Dictionary Tag. Same as TAG except header number value is an index into a dictionary of tag names (for compressed encoding of tag names).
- DATTR(TT=4): Same as DTag but for attribute names. Distinct dictionary for tags and attributes.
- EXT(TT=0): Anything other than the XML structures covered by explicit types above, e.g. processing instructions. Number value is extension subtype identifier (identifiers to be allocated as extensions are defined).

In this work, types DTAG and BLOB are the most used among implemented types. The following enumeration lists the different DTAGs used when encoding CCN messages. The same tags are located when decoding a message and used to identify the message's composition.

```
enum ccn_dtag {
    CCN_DTAG_Any                = 13,
    CCN_DTAG_Name               = 14,
    CCN_DTAG_Component          = 15,
    CCN_DTAG_Content            = 19,
    CCN_DTAG_Interest           = 26,
    CCN_DTAG_Signature          = 37,
    CCN_DTAG_Nonce              = 41,
    CCN_DTAG_ContentObject      = 64,
};
```

Here is an example of a CCN encoding mechanism's usage. An Interest message with the prefix "/BATTERY/NODE1" will be encoded according to the following scheme :

```
CCN_DTAG_Interest
 CCN_DTAG_Name
     CCN_DTAG_Component
         CCN_BLOB
         BATTERY
     CLOSER
     CCN_DTAG_Component
         CCN_BLOB
         NODE1
     CLOSER
 CLOSER
CLOSER
```

2.1.8 Parsing

Incoming CCN messages have to be parsed to detect the message's type and call the corresponding processing functions. Outgoing messages are parsed before their transmission to check that they are well formed. Parsing allows to delimit the message components (like name and data in a Content Object) and the prefix components. Three routines of parsing are implemented: name parsing, Interest message parsing and Content Object message parsing.
Name parsing consists in looking for name components boundary offsets. These offsets are often stored in an array to avoid every time re-parsing of name components which are used in matching routines or for other processing.
The parse of an Interest message results in an array of offsets into the wire representation, with the start and end of each major element and a few of the important sub-elements. The following enum allows those array items to be referred to symbolically. The _B_ indices correspond to the beginning offsets and the _E_ indices correspond to the ending offsets. An omitted element has its beginning and ending offset equal to each other. Normally these offsets will end up in non-decreasing order. Some aliasing tricks may be played here, e.g. since offset[CCN_PI_E_ComponentLast] is always equal to offset[CCN_PI_E_LastPrefixComponent], we may define CCN_PI_E_ComponentLast = CCN_PI_E_LastPrefixComponent. However, code should not rely on that, since it may change from time to time as the interest schema evolves.

```
enum ccn_parsed_interest_offsetid {
    CCN_PI_B_Name,
```

```
    CCN_PI_B_Component0,
    CCN_PI_B_LastPrefixComponent,
    CCN_PI_E_LastPrefixComponent,
    CCN_PI_E_ComponentLast = CCN_PI_E_LastPrefixComponent,
    CCN_PI_E_Name,
    CCN_PI_B_Nonce,
    CCN_PI_E_Nonce,
    CCN_PI_B_OTHER,
    CCN_PI_E_OTHER,
    CCN_PI_E
};
```

The parse of a Content Object message is done with the same mechanism but using the following enumeration. Using the stored offsets it is possible to retrieve the value of a specific field or its length. For example, calculating $[CCN\_PCO\_E\_Name - CCN\_PCO\_B\_Name + 1]$ gives the length of the Content Object's name.

```
enum ccn_parsed_content_object_offsetid {
    CCN_PCO_B_Signature,
    CCN_PCO_E_Signature,
    CCN_PCO_B_Name,
    CCN_PCO_B_Component0,
    CCN_PCO_E_ComponentN,
    CCN_PCO_E_ComponentLast = CCN_PCO_E_ComponentN,
    CCN_PCO_E_Name,
    CCN_PCO_B_Content,
    CCN_PCO_E_Content,
    CCN_PCO_E
};
```

2.1.9 Applications input/output

When the CCN driver receives a packet from the lower layer, it is placed in a packet buffer CCN_BUF and then it calls the ccn_input() function which posts a PACKET_INPUT event to the CCN stack. The global variable ccn_len holds the length of the packet in the buffer. Then the stack processes the received message and notifies the application. The CCN stack notifies the application whenever an event occurs by calling the CCN_APPCALL() function. It uses ccn_event to signal a CCN stack event to the application. The sequence diagram presented in Figure 7 summarizes the interactions between the different Contiki layers in order to deliver and process an incoming packet.

The ccn_appdata pointer points to application data. The size of the data is obtained through the CCN function ccn_datalen(). The data will be overwritten after the application function returns, and the application will therefore have to either act directly on the incoming data or copy the incoming data into a buffer for later processing.

To send data, the application used the CCN function ccn_send(). The ccn_send() function takes two arguments: a pointer to the data to be sent and the length of the data. If the application needs a buffer where to produce the data to send, the CCN packet buffer (pointed to by the CCN_BUF pointer) can be used for this purpose. Note that it is not possible to call ccn_send() more than once per application invocation; only the data from the last call will be sent. The

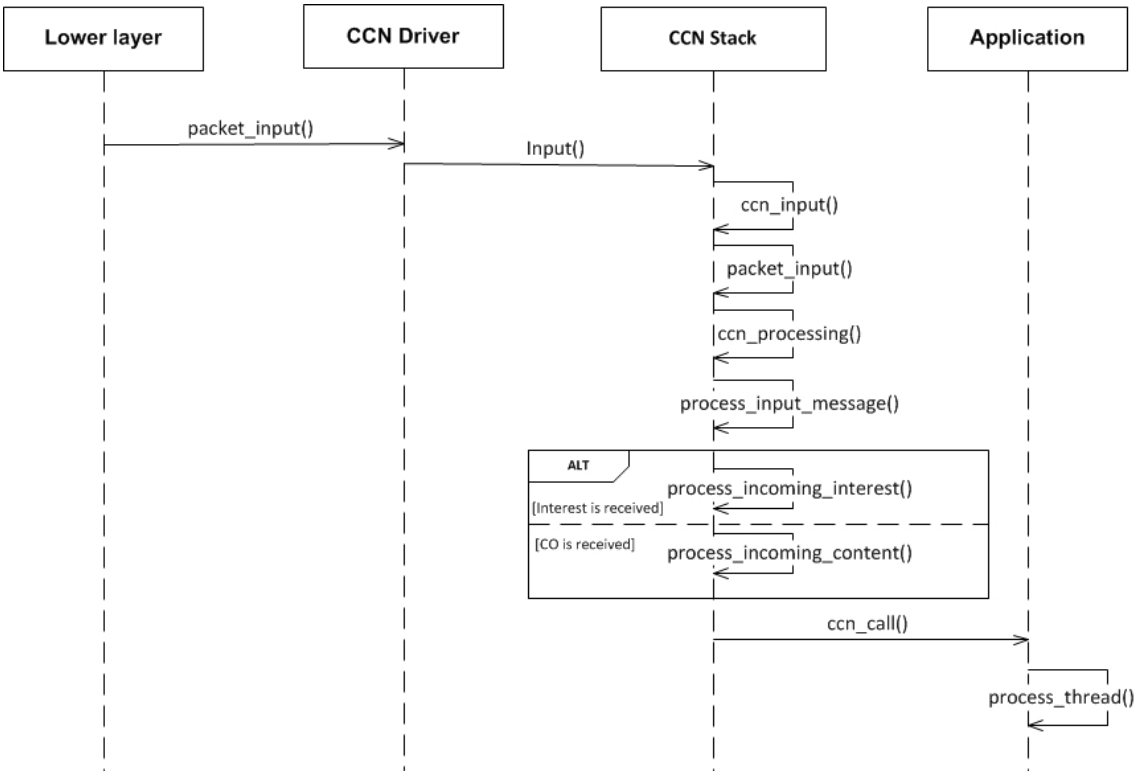


Figure 7: Sequence diagram for incoming messages processing.

sequence diagram in Figure 8 shows the different function calls to create a valid Interest message and to send it using the implemented CCN communication module.

2.2 CCN implementation

2.2.1 Modules

Handle management functions : The functions of this module are used for handle management. They allow to initialize the CCN handle created by the CCN stack when application starts running. Note that only one handle is used by the CCN stack.

Face management functions : This module holds the CCN functions used for faces management. They allow to create faces, to add them to the Face table, to delete them and to search for a specific face based on a face identifier.

Content Store management functions : The functions of the CS management module are used to create a piece of Content, to add it to the Content Store, to pack CS entries and eliminate blanks, to delete a content from CS and to look for a Content Object based on its accession number.

Nameprefixes management functions : The CCN nameprefix functions are used to manage the FIB and PIT tables. They allow to register prefixes to the FIB table and to delete them.

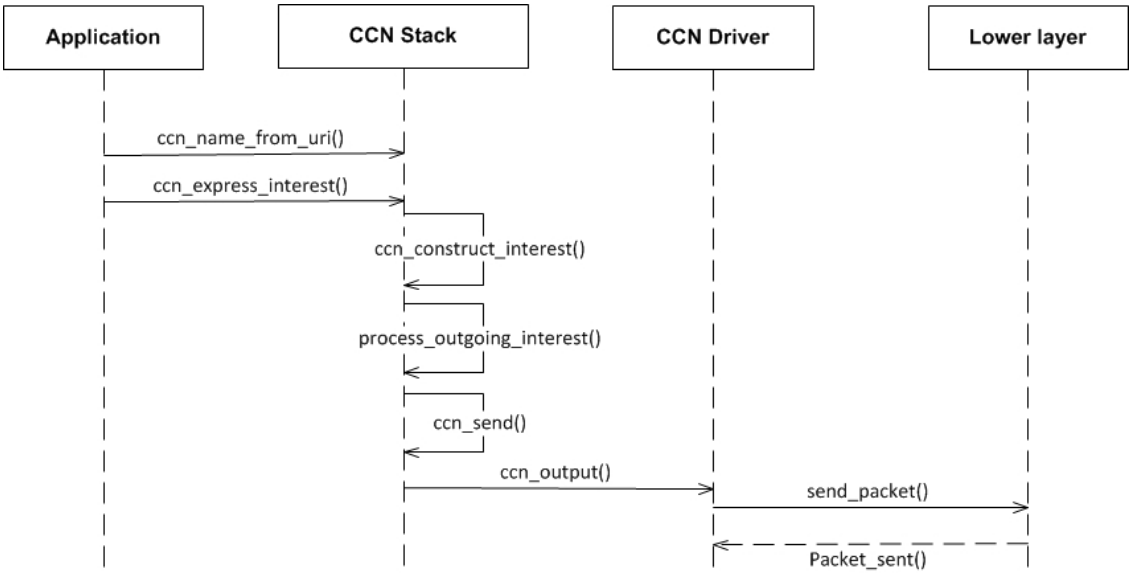


Figure 8: Sequence diagram for Interest sending.

They attach to a FIB entry the face identifiers leading to sources of name matching contents. They also link unsatisfied Interest messages to the corresponding FIB prefixes (if they do exist) and manage PIT entries by consuming pending Interests when they timeout or when a matching content arrives. More functions are implemented to enable lookup for a specific entry in FIB or PIT based on a given prefix.

CCN name management : The CCN name functions allow to create and manage CCN names and to convert between URI form and CCN binary form.

CCN binary encoding : This module implements necessary routines to create CCN binary-encoded data. These routines serve to create well-formed CCN messages.

CCN binary decoding : Functions of this module are used to decode a CCN binary-encoded message.

CCN message parsing : The CCN message parsing functions parse received messages to check their types and apply the right processing treatment. They detect the limits between the message's fields of arbitrary length. They also parse names to locate delimiters between the name components. The offsets of name delimiters are kept in an array and used to retrieve name components without re-parsing the name or the message again.

CCN matching functions : The CCN approach relies on name matching operations. This module gathers different functions used to look for matches. They allow to simply compare two names, to test for a match between a Content Object message and an Interest message, to look for a matching content in Content Store to a given Interest, to find and consume pending Interests that match a given content.

CCN messages processing : The CCN processing functions are used to process incoming Interest messages and incoming Content Object messages. They use most of the other modules' functions to detect the type of the received message and process the corresponding algorithm.

CCN Interest construction and processing : These functions serve to construct a valid Interest message and to process it. An Interest message is processed by checking if FIB allows to send it and whether a similar Interest is pending in the PIT table and then it is pushed to the CCN Driver which will take care of sending the message to the lower layer.

Transmission & reception module : This module handles the interaction between the CCN stack and the CCN driver by filling and retrieving data from the CCN input/output buffer.

CCN Driver : CCN Driver is charged of communication with the link layer. It sends the packets created by the CCN stack to the link layer and calls the CCN stack when a new packet is received.

The most important functions of these CCN modules are represented in the Figure 9

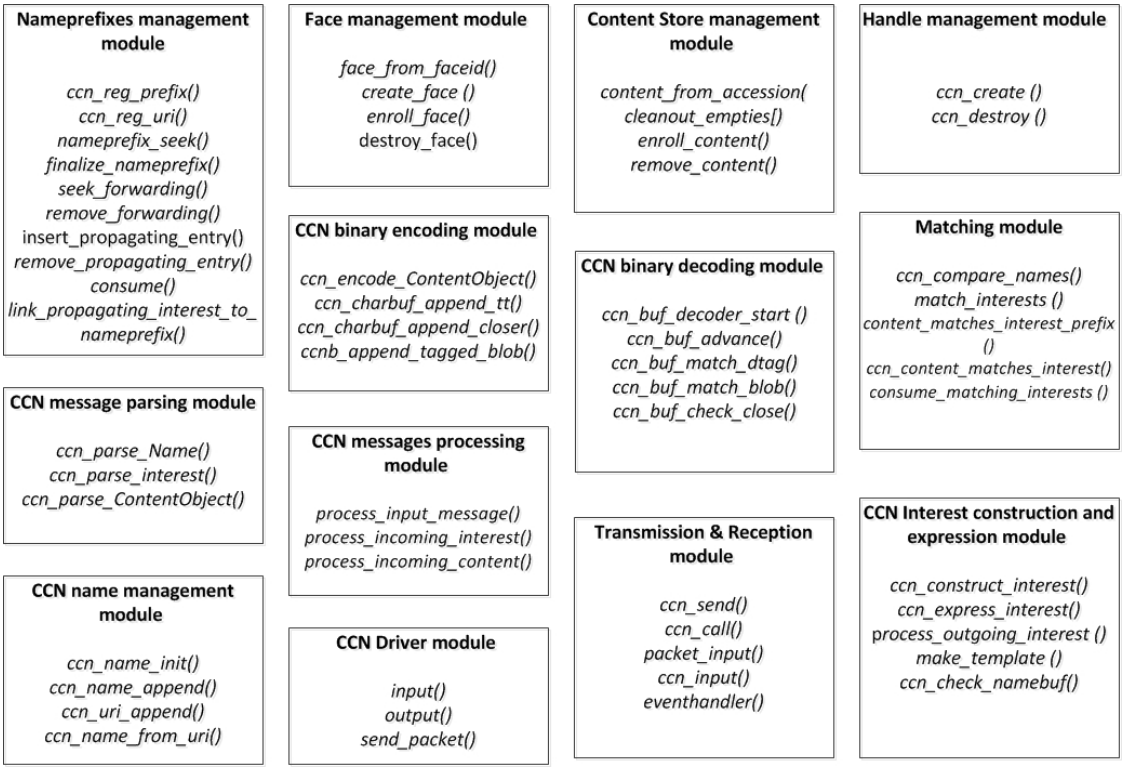


Figure 9: Most important functions of CCN modules.

2.2.2 Data structures

Face: struct face (defined in ccn.h file) is used to represent a face.

```
struct face {
    uint8_t faceid;
    struct ccn_skeleton_decoder decoder;
    uint16_t pending_interests;
};
```

Every instance of the *struct face* has a unique face identifier stored in the field *faceid*. The struct *Decoder* is used to decode CCN binary messages received on this face. *Pending_interests* is a variable holding the number of Interest messages sent out from this face.

content_entry: struct content_entry (defined in ccn.h file) contains a piece of content.

```
struct content_entry {
    ccn_accession_t accession;
    struct ccn_indexbuf *comps;
    uint8_t ncomps;
    unsigned char key[MAX_CONTENT_SIZE];
    uint8_t key_size;
    uint8_t size;
};
```

Accession is a unique index given to each content object stored in CS. *Key* is a pointer to an array of MAX_CONTENT_SIZE characters where the whole ccn binary-encoded Content Object is held. *Size* is the size of the Content Object. *Key_size* is the size of the content's name. Data's size is deduced from subtracting *key_size* from *size*. Content_entry keeps also the parsed name results by filling *comps*, an array of name components offsets, and *ncomps*, the number of name components.

Nameprefix_entry: struct nameprefix_entry (defined in ccn.h file) is an entry of the table of names where FIB and PIT tables are mixed.

```
struct nameprefix_entry {
    struct propagating_entry *pe_head;
    struct ccn_charbuf *name;
    struct ccn_forwarding *forwarding;
};
```

Nameprefix_entry is a FIB entry to which matching pending Interests are attached. The coupling of the two tables allows to minimize the cost of lookup operations. When processing an incoming Interest message, we start by looking for a match in the FIB table. If no matching prefix is found, this means that the node is not allowed to route interests with that prefix or that it simply ignores where to send it to retrieve matching data. In this case, we are quite sure that the node will not keep any unsatisfied Interests with that Interest in its PIT table. If lookup on FIB leads to a name match, then searching for other matching pending Interest will be limited to a lookup on the set of unsatisfied Interests attached to the matching FIB entry. Then, the node do not have to check the full PIT table for matches.
The nameprefix_entry structure is composed of pointers to three other data structures. *name* contains the name or the name prefix of the FIB entry. *Forwarding* is pointer to a linked-list where are kept the identifiers of faces sources of matching contents. *Pe_head* is pointer to a double linked-list holding unsatisfied matching Interests.

ccn_forwarding : struct ccn_forwarding (defined in ccn.h) is used by nameprefix_entry to identify a source face of data matching the prefix.

```
struct ccn_forwarding {
    uint8_t faceid;
    struct ccn_forwarding *next;
};
```

A ccn_forwarding entry keeps track of a face to which Interests, matching a given name prefix, are allowed to be forwarded. This face should lead to matching data and is thus considered as a source of matching content. The structure allows to have a linked-list of face identifiers, each one is held in *faceid*.

propagating_entry : struct propagating_entry (defined in ccn.h) represents a pending Interest in the PIT.

```
struct propagating_entry {
    struct propagating_entry *next;
    struct propagating_entry *prev;
    struct ccn_indexbuf *faceids;
    unsigned char interest_msg[MAX_INTEREST_SIZE];
    uint8_t size;
    struct ctimer ct;
    struct ccn_handle *h;
    struct nameprefix_entry* npe;
};
```

propagating_entry holds in *interest_msg* the buffer containing the pending Interest message. *Size* contains the message's size. *Faceids* is an array where identifiers of faces, sources of this unsatisfied Interest, are kept. If a node receives the same Interest stored in PIT but on a different face, it simply adds the face identifier to *faceids* instead of creating a new PIT entry for the newly received Interest message.
While the Interest is pending, the propagating_entry is held in a doubly-linked list belonging to the matching nameprefix_entry *npe*. When the Interest is consumed, the propagating_entry is removed from the doubly-linked list and is cleaned up by freeing unnecessary bits (including the interest message itself). When the timer *ct* expires, the Interest is erased as PIT entries must time out.

ccn_handle : struct ccn_handle (defined in ccn.h) gathers the different CCN tables and several monitoring variables.

```
struct ccn_handle {
    struct nameprefix_entry *nameprefix_tab[NAME_MAX_ENTRIES];
    uint8_t name_limit;
    struct face *faces_by_faceid[FACE_MAX_ENTRIES];
    uint16_t face_gen;
    uint8_t face_rover;
    uint8_t face_limit;
    struct content_entry *content_by_accession[CS_MAX_ENTRIES];
    ccn_accession_t accession_base;
    uint8_t content_by_accession_window;
    ccn_accession_t accession;
```

```
    unsigned long content_dups_recvd;
    unsigned long content_items_sent;
    unsigned long interests_accepted;
    unsigned long interests_dropped;
    uint8_t running;
    ccn_appstate_t appstate;
};
```

The structure ccn_handle is the main data structure. Only one handle is created in the CCN stack and applications have to use it. *Running* is set to 1 when the handle is used and *appstate* indicates the application's state to the attached process.
The structure ccn_handle contains the FIB, PIT, CS and Face tables. *Nameprefix_tab* is an array holding nameprefix entries. *Name_limit* holds the array's maximal size. *Faces_by_faceid* is the face table. *Face_limit* contains the maximum allowed number of face elements. *Face_rover* is the number of faces enrolled to the face table. *Content_by_accession* represents the Content Store of a maximal size of *content_by_accession_window*. *Accession_base* is the number of Contents Objects stored in CS.
The ccn_handle structure keeps also some monitoring data. *Accession* gives the number of Content Objects stored in CS since the node has started to work. *Face_gen* contains the lifetime number of generated faces. *Interests_accepted* and *interests_dropped* hold the numbers of accepted and dropped Interests. *Interests_dropped* counts also Interests a node created locally but failed to send due to errors or FIB/PIT restrictions. *Content_dups_recvd* contains the number of duplicated received Content Objects. *Content_items_sent* counts the Content Objects sent correctly by the node.

ccn_charbuf: struct ccn_charbuf (defined in charbuf.h) is a buffer for counted sequences of arbitrary bytes.

```
struct ccn_charbuf {
    uint8_t length;
    uint8_t limit;
    unsigned char buf[MAX_CHARBUF_SIZE];
};
```

The structure ccn_charbuf is mostly used to hold names, Interests and Content Objects messages. *Length* gives the size of the message stored in *buf*.

ccn_indexbuf: struct ccn_indexbuf (defined in indexbuf.h) is a buffer of non-negative values.

```
struct ccn_indexbuf {
    uint8_t n;
    uint8_t limit;
    size_t buf[MAX_INDEXBUF_SIZE];
};
```

The structure ccn_indexbuf is an array used often to hold the delimiting offsets for a name components after parsing. Keeping these offsets allows to avoid re-parsing the name or the message every time a name matching operations have to be done. *N* gives the number of elements stored in *buf*.

2.3 CCN integration in Contiki

This section describes how we have integrated the CCN communication layer into Contiki. In this work, we used the version 2.6 of Contiki and our code relies on the version 0.3.0 of CCNx.

2.3.1 Contiki build system

The Contiki build system is composed of a number of Makefiles:

- Makefile: the project's makefile, located in each project directory.
- Makefile.include: the system-wide Contiki makefile, located in the root of the Contiki source tree.
- Makefile.$(TARGET): where $(TARGET) is the name of the platform that is currently being built. It contains compiling rules for the specific platform, located in the platform's subdirectory in the $platform/$ directory.
- Makefile.$(CPU): where $(CPU) is the name of the CPU or microcontroller architecture used on the platform for which Contiki is built. The compiling rules for the CPU architecture are located in the CPU architecture's subdirectory in the cpu/ directory.
- Makefile.$(APP): where $(APP) is the name of an application in the apps/ directory. The compiling rules for applications in the apps/ directories. Each application has its own makefile.

The Makefile in the project's directory is intentionally simple. It specifies where the Contiki source code resides in the system and includes the system-wide Makefile, Makefile.include. The project's makefile can also define in the APPS variable a list of applications from the apps/ directory that should be included in the Contiki system.
Firstly, the location of the Contiki source code tree is given by defining the CONTIKI variable. Secondly, the name of the application is defined. Finally, the system-wide Makefile.include is included.
The Makefile.include contains definitions of the C files of the core Contiki system. Makefile.include always resides in the root of the Contiki source tree. When make is run, the Makefile.include includes the Makefile.$(TARGET) as well as all the makefiles for the applications in the APPS list (which is specified by the project's Makefile). Makefile.$(TARGET), which is located in the platform/$(TARGET)/ directory, contains the list of C files required by the platform to be added in the Contiki system. This list is defined by the CONTIKI_TARGET_SOURCEFILES variable. The Makefile.$(TARGET) also includes the Makefile.$(CPU) from the cpu/$(CPU)/ directory. The Makefile.$(CPU) typically contains definitions for the C compiler used for the particular CPU.

2.3.2 Code integration

The newly created code sources are placed in a new folder named ccn in the directory *contiki/core/net/ccn*. To make it possible to use the CCN stack, few modifications to some headers and makefiles are needed. The file *contiki-net.h* in the directory *contiki/core* contains the different include files for the available implemented network stacks. We have added thus ccn.h et ccn_stack.h header files to the *contiki-net.h* file as follows:

```
#include "net/ccn/ccn.h"
#include "net/ccn/ccn_stack.h"
```

The next file to modify is *netstack.h* at the directory *contiki/core/net*. It holds the configuration of the network, mac and rdc drivers to be used by default in case of the platform or application configuration file is missing or the drivers are not specified. We have replaced the default rime_driver by our ccn_driver in the file *netstack.h*:

```
#ifndef NETSTACK_NETWORK
#ifdef NETSTACK_CONF_NETWORK
#define NETSTACK_NETWORK NETSTACK_CONF_NETWORK
#else /* NETSTACK_CONF_NETWORK */
#define NETSTACK_NETWORK ccn_driver
#endif /* NETSTACK_CONF_NETWORK */
#endif /* NETSTACK_NETWORK */
```

A new makefile named *Makefile.ccn* is created in the new ccn folder at the directory *contiki/core/net/ccn*. It lists the C source files to be compiled for the project.

in Makefile.ccn :

```
CONTIKI_SOURCEFILES += ccn.c ccn_buf_decoder.c ccn_buf_encoder.c ccn_charbuf.c \
ccn_coding.c ccn_driver.c ccn_indexbuf.c ccn_name_util.c ccn_stack.c ccn_uri.c
```

The created *Makefile.ccn* has to be added to the *Makefile.include* located at the directory *contiki* and regrouping all Contiki's makefiles. A new entry for Makefile.ccn is created and "net/ccn" is added to the list of directories. In order to reduce the code's size after compilation, the files needed for the CCN functioning are separated from the IP files in the makefile. If the CFLAG WITH_CCN is set to one then only the ccn stack and driver files will be included. Otherwise, the usual ipv4 or ipv6 files will be selected when compiling. The RPL makefile reference is moved inside the IP block.

The modifications of the Makefile.include are as follows:

```
ifdef WITH_CCN
  CFLAGS += -DWITH_CCN=1
include $(CONTIKI)/core/net/ccn/Makefile.ccn
else # WITH_CCN
  NET   += uip-debug.c
ifdef UIP_CONF_IPV6
  CFLAGS += -DUIP_CONF_IPV6=1
  UIP   = uip6.c tcpip.c psock.c uip-udp-packet.c uip-split.c \
          resolv.c tcpdump.c uiplib.c simple-udp.c
  NET   += $(UIP) uip-icmp6.c uip-nd6.c uip-packetqueue.c \
          sicslowpan.c neighbor-attr.c neighbor-info.c uip-ds6.c
  ifneq ($(UIP_CONF_RPL),0)
    CFLAGS += -DUIP_CONF_IPV6_RPL=1
    include $(CONTIKI)/core/net/rpl/Makefile.rpl
  endif # UIP_CONF_RPL
else # UIP_CONF_IPV6
  UIP   = uip.c uiplib.c resolv.c tcpip.c psock.c hc.c uip-split.c uip-fw.c \
```

```
          uip-fw-drv.c uip_arp.c tcpdump.c uip-neighbor.c uip-udp-packet.c \
          uip-over-mesh.c dhcpc.c simple-udp.c
  NET   += $(UIP) uaodv.c uaodv-rt.c
endif # UIP_CONF_IPV6
endif # WITH_CCN
...
CONTIKIDIRS += ${addprefix $(CONTIKI)/core/,dev lib net net/ccn net/mac net/rime \
                net/rpl sys cfs ctk lib/ctk loader. }
```

Contiki's platforms have their own configuration files where many values are set. For the Sky platform, the ccn_driver has to be set as the network driver instead of the rime_driver. The contiki-conf.h file is modifies as follows:

```
#define NETSTACK_CONF_NETWORK ccn_driver
```

3 Performance evaluation

The testing and the evaluation of the implemented CCN communication layer in Contiki were conducted in three parts: using simulation with Cooja Simulator, testing on a small number of physical nodes available in our lab and finally testing on SensLab, a large WSN platform. This section presents the details of those different tests. Firstly, we present a synthetic test application to be used for data collection using the CCN polling mechanism. Secondly, we describe the simulations done within the Cooja Simulator to validate the implemented code. We also present the simulations results and their analysis. The third part is about the tests we made on a set of 10 TelosB sensor nodes. Finally, we present the experiments done using the SensLab testbed.

3.1 Test application

To validate and evaluate the performances of the implemented CCN communication layer, we developed an application that runs on top of the Contiki OS. The application provides a data collection scenario where a sink asks periodically for the data collected by a certain node. Each node creates a content with the gathered information and stores it locally in the Content Store. The content contains the length of the collection frame, the timestamp, the time synchronization status, the cpu time, the low power mode time, the transmit time, the listen time and the sensors values (battery voltage, battery indicator, light1, light2, temperature, humidity, radio intensity, etx1, etx2, etx3 and etx4). Nodes can update their contents by locally refreshing sensing data. The content stored at a Node N_i is identified by the name "$/COLLECT/Ni$".

The sink node periodically sends an Interest message to each node requesting its collected data. The Interest message holds the name "/COLLECT/Ni". If there are more than one collecting node in the network, each one has to have a unique identifier number i. Because the transmission is limited to broadcast, this is the only available method that guarantees data collection from a specific node and subsequently from all nodes. If two collecting nodes share the same identifier number, then their two contents will be referenced with the same name, which implies that only one content will be delivered to the sink in response to its request.

In this application, the sink or the nodes receiving a propagating Content Object process the message without keeping a copy in their local Content Store. Giving the very limited memory resources on a sensor device, it would not support keeping more than few Content Objects in its CS. In addition, both sink and nodes have the prefix "/COLLECT" in their FIB tables so that they can forward the Interests generated by the sink and provide a hop-by-hop communication.

The application code sources are available at the directory *contiki/examples/ccn*.

3.2 Simulation

3.2.1 Cooja Simulator

Cooja is a simulator designed for simulating sensor networks running the Contiki operating system [11]. The simulator is implemented in Java but the code running on a sensor node is written in C.

Unlike most of wireless sensors networks simulators that target one fixed level of simulation,

Cooja enables cross-level simulation. It allows to simulate a system at three levels simultaneously, namely the networking level (or the application level), the operating system level and the machine code instruction level.

Cooja can execute Contiki programs in two different ways, either as compiled native code for the platform on which the simulator is run, or in a sensor node emulator that emulates an actual sensor node at the hardware level. When running the program code as a compiled native code, the Contiki system is compiled for a specific indicated hardware platform. Each platform has a set of drivers and particular communication methods with the hardware.

Cooja enables simulation of networks of sensor nodes with different types, differing in the loaded software or in the simulated hardware. Nodes sharing the same type have common properties. They are initialized with the same data memory. They run the same software on the same simulated hardware peripherals. Their behaviour, however, will differ at runtime due to different external inputs.

Nodes creation and configuration can be easily done using the graphical interface of the Cooja simulator. The user has a set of plugins and interfaces that enable him to control and monitor the simulation progress. Plugins are used to interact with simulations. Interfaces represent the hardware peripherals of simulated nodes. They allow the simulator to detect and trigger events and show the properties of simulated nodes like radio or position. A user can customize the simulation environment by adding interfaces and plugins that provide simulation-specific functionalities.

3.2.2 Simulation setup

The Contiki system running on top of simulated sensor nodes was compiled for the sky platform. Node's configuration was done using the following drivers:

- Network driver : ccn_driver
- Mac driver : csma_driver
- Radio duty cycling driver : sicslowmac_driver
- Radio driver : cc2420_driver

The simulation tests run on Cooja consist in 4 scenarios of data collection with 10, 20, 30 and 40 deployed nodes. In each scenario, there is only one sink node. A collecting node can exchange messages with the sink node and the other nodes using the broadcast mode. Sensor nodes are placed randomly in a 100m x 100m area. The sink transmits an Interest message every 20 seconds to a node N_i. Thus, a data collection round requires *N x 20 seconds* to interrogate the N deployed nodes.

3.3 Physical nodes experimental setup

In order to test the functioning of the CCN communication layer, tests were made on a set of physical sensor devices. The tests were performed using TelosB sensor nodes.

3.3.1 TelosB nodes

TelosB is an open source mote platform developed by UC Berkeley for wireless sensor network experimentation. It is designed for extremely low power, high data-rate sensor network applications. A TelosB mote (Fig. 10) platform disposes of an IEEE 802.15.4 radio with integrated antenna, a low-power MCU with extended memory, optional sensor suite and offers USB programming capability. It is also equipped with a TI MSP430 F1611 microcontroller which assures the low power operations. The 16-bit RISC processor reduces power consumption by switching to the sleep mode most of the time but it grants fast wakeup from sleep state. The mote is powered either by two AA batteries or by the host computer when it is plugged into the USB port for programming or communication.

Figure 10: The TelosB mote.

The TelosB mote is provided in two models: (TPR2400) and (TPR2420). The difference between them is that the latter holds a sensor suite including integrated light, temperature and humidity sensors. Below are listed some features offered by both models [8]:

- Program Flash Memory of 48K bytes
- RAM of 10K bytes
- 1MB external flash for data logging
- IEEE 802.15.4/ZigBee compliant RF transceiver
- 2.4 to 2.4835 GHz, a globally compatible ISM band
- 250 kbps data rate
- Integrated onboard antenna
- Outdoor Range 75 m to 100 m
- Indoor Range 20 m to 30 m
- Programming and data collection via USB

To run Contiki OS on TelosB motes, the system must be compiled using the Sky platform. We have then to run the compilation of the test application directory as follows:

```
$ make TARGET=sky
```

Note that there is also the Tmote Sky mote which shares the same design with the TelosB mote. The former is sold by Sentilla (formerly Moteiv); the latter is sold by Crossbow. The Tmote Sky/TelosB port was integrated into the Contiki system build in march 2007 and has since become one of the main platforms for Contiki [7].
The platform-specific source code for the Tmote Sky/TelosB port can be found in the directories platform/sky and cpu/msp430 in the Contiki source tree. Code for writing to the on-chip flash ROM is in the cpu/msp430/flash.c and code for reading and writing to the external flash is the file platform/sky/dev/xmem.c. The serial/USB port is read from and written to with either the code in cpu/msp430/dev/uart1.c if the mote is running TCP/IP or cpu/msp430/slip_uart1.c if Rime is used. The CC2420 radio chip drivers in the Contiki source code can be found in core/dev/cc2420.c.

3.3.2 Test scenarios

Using the set of TelosB sensor nodes, we developed two test scenarios based on the same data collection application used for simulation. The sink sends an Interest message each 20 seconds to a node N_i. A data collection round takes then 200 seconds to be done over 10 nodes. A collecting node updates its locally collected information each 70 seconds by sensing its environment. In the first test scenario, data-collecting nodes do not store the Content Objects they may receive in their Content Stores. They simply forward them when they are allowed to. At the contrary, in the second test scenario, content storage in Content Store is enabled for a data-collecting node. Nodes update the stored Content Objects when receiving new contents from other nodes with the same names. We are interested in analyzing how content caching in the network's nodes may affect the time delays and the network load.

Experiments were done on set of 10 TelosB nodes: 3 TPR2420-model nodes and 8 TPR2400-model nodes including the sink node. All nodes were placed in the same room. Each experiment lasted at least 1h30. The Contiki system running on top of these TelosB nodes was compiled for the sky platform. Node's configuration was done using the following drivers:

- Network driver : ccn_driver
- Mac driver : csma_driver
- Radio duty cycling driver : sicslowmac_driver
- Radio driver : cc2420_driver

The sicslowmac RDC driver is used to ensure high packet reception rates while radio is kept active during the execution time.

3.4 Testbed experiments setup

3.4.1 SensLAB platform

SensLAB [6] is a very large scale open wireless sensor network testbed. It has been developed and deployed in order to allow the evaluation through experimentations of scalable wireless sensor network protocols and applications. SensLAB is aiming at providing an accurate open access multi-users scientific tool to support the design, the development and the experimentation of real

large-scale sensor network applications [17].

The SensLAB testbed counts 1024 nodes deployed at 4 sites. Each site hosts 256 sensor nodes with different deployment topologies, environment and hardware specificities. Nodes have different characteristics in order to offer a wide spectrum of possibilities and heterogeneity. Nodes on the same site can communicate with their neighbours using their radio interfaces. Each one can be configured to perform as a sink node and may exchange data with any other sink node of the whole SensLAB testbed or any computer on the Internet [17]. The sites are located in the cities of Lille, Strasbourg, Rennes and Grenoble in France. They offer fixed and mobile sensor nodes. Mobile nodes are only available on 2 sites (Strasbourg and Lille).

SensLAB provides each user with a web portal access, from which experiments can be set, and a virtual machine holding a set of development tools. Through the SensLAB's web portal, the user can set and configure an experiment, reserve resources, load firmwares to sensor nodes and collect experimental data. The configuration of an experiment allows to set a number of parameters such as the number of nodes, sensor and radio characteristics, topology considerations and experimentation time.

3.4.2 Test details

In this work, we ran tests on the SebsLAB platform in the Strasbourg's site. It is composed of fixed and mobile nodes equipped with a CC1101 radio chip. The Contiki system running on top of sensor nodes was compiled using the wsn430 platform. Each node is configured to use the following drivers:

- Network driver : ccn_driver
- Mac driver : csma_driver
- Radio duty cycling driver : sicslowmac_driver
- Radio driver : cc1100_driver

We ran 2 test scenarios with 10 and 20 collecting nodes. Each test lasted between one and two hours.

3.5 Results

3.5.1 Simulation results

Figures 11 and 12 present the obtained collection delay of a single interest and its data response sent by the sink under varying network sizes 10, 20, 30 and 40 nodes running on the cooja simulator. Figure 11 shows the box plots of delays. We observe that the mean delay is close to the 25th percentile and remains stable around 219 ms while increasing the number of nodes. However the maximum delay reaches 512 ms and remains stable for 20, 30 and 40 nodes. Thus, we observe that on average a data requires 219 ms to be collected by the sink. The Figure 12 depicts the CDF (Cumulative Distribution Funciton) of the obtained delays. We observe that for a small size networks, the collection delay is at minimum 211 ms for 10 nodes and at maximum 512 ms for 40 nodes.

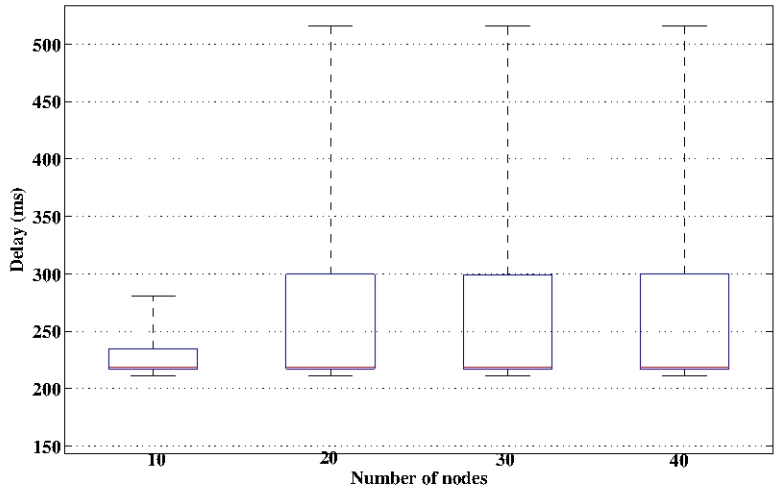


Figure 11: Box plot of collection delay measured on the sink node under varying network sizes.

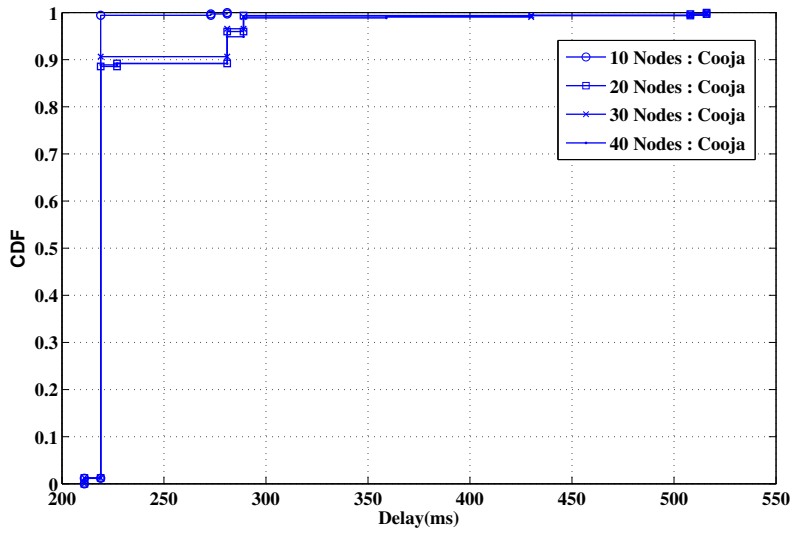


Figure 12: CDF of collection delay measured at the sink node under varying network sizes.

3.5.2 TelosB nodes results

The results of the experiments over the TelosB sensor nodes lead to a set of observations about the storage of forwarded contents in CS. Table 1 shows the results of the data-collection application at the sink level for the two scenarios with a caching function on the CS respectively disabled and enabled. With an execution time of approximately 1h32 and an Interest generation rate of 1 message each 20 seconds, both applications in the two scenarios generated 276 Interest

messages for data collection from nodes. This test validates the Interest generation and sending mechanism in the CCN stack. Theoretically, generating one Interest every 20 seconds in a period of 93 minutes leads to a total of 276 Interests (93 * 60 / 20 = 276). However, the CS-disabled application sent an additional 374 Interest messages, rising the total number of Interests sent over the CCN driver to 650 messages. This is due to the CCN forwarding mechanism which implies to forward a received Interest message if no matching content is found locally in the Content Store cache. The Interest forwarding led to more Content Object forwarding as well through the network. When a node receives an Interest message to which he has no matching data, it keeps a copy of it in the PIT table and forwards the message. If it receives a matching Content Object, it deletes the corresponding pending entry in the PIT and forwards the Content Object message. But nothing prevents the node from receiving the same Interest message again if the message is still transiting in the network. In this case, the node does not observes a corresponding pending entry in its PIT as it was cleared up and will then re-forward the Interest and eventually the Content Object if it will receive later a matching piece of content. When the application does not allow the CCN stack to store the forwarded contents locally, more data forwarding will be done. At the contrary, in the case where forwarded contents are stored locally, if a node receives the same Interest message for a second time, it will directly reply with the matching content kept in the CS cache and the processing of the incoming Interest is finished. There will be in consequence no extra Interest forwarding and eventually less Content forwarding which will make the network load less important. From the results in Table 1, we observe that the network load is more important in the CS-disabled scenario with 151568 bytes received and 29118 bytes sent in comparison to the CS-enabled scenario results where only 60222 bytes are received and 7571 bytes are sent.

-	CS-disabled scenario stats	CS-enabled scenario stats
Execution time (seconds)	5539	5520
Generated Interests	276	276
Satisfied Interests	272	266
Total sent Interests	650	276
Bytes received	151568 (Interest: 16794 / Content Object: 134774)	29118 (Interest: 683 / Content Object: 28435)
Bytes sent	60222 (Interest: 13053 / Content Object: 47169)	7571 (Interest: 5567 / Content Object: 2004)
Average delay (ms)	315	272

Table 1: The results of tests on the TelosB nodes.

Thus, storing forwarded Content Objects locally in the Content Store cache has an effect on delay. As the content will spread over the network's nodes, the availability of the content will increase and the sink may receive the data it requests with better delays. The CS-enabled application has a better average delay of 272 ms per packet while the CS-disabled application has an average delay of 315 ms per packet. Table 2 shows the measured delay in the first 7 data collection rounds for the two scenarios. After few rounds, the round average delay of the CS-enabled application became lower than the delay of the CS-disabled application as illustrated in Figure 13.

Delays of the CS-enabled scenario are regrouped in Table 3, we can see that the 1st round average delay is the highest among the delays of the different rounds with a value of 440.11 ms. The average delay tends to decrease in the next data collection rounds as shown in Figure 14. Before the first collection round, the Content Store of each node holds only its proper content.

Data Collection Round	1st Round	2nd Round	3rd Round	4th Round	5th Round	6th Round	7th Round
Average Delay with CS enabled (ms)	440,11	310,9	317,8	258,67	262,2	260,8	260
Average Delay with CS disabled (ms)	296	327,9	355,3	371,8	320,9	313,9	300,5

Table 2: Summary of average delays with enabled and disabled caching strategy using 10 TelosB nodes.

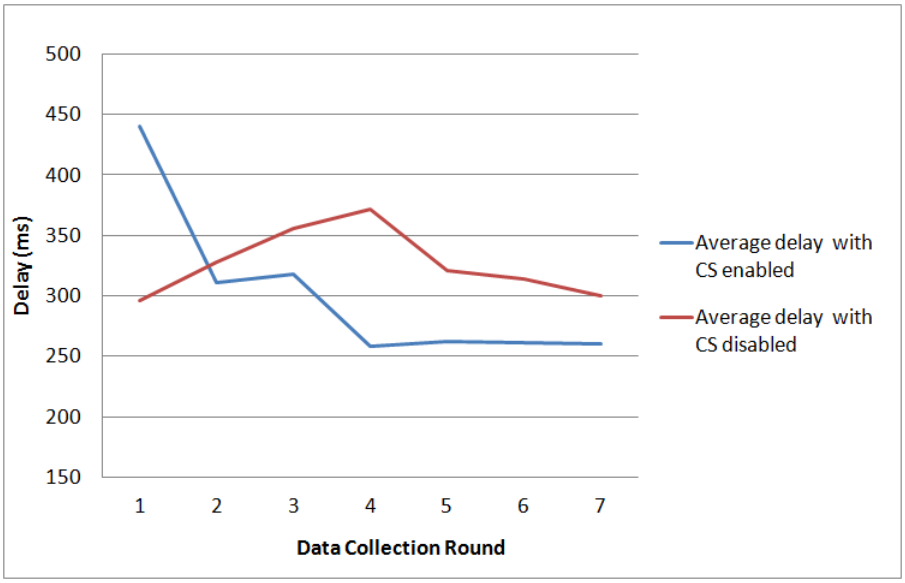


Figure 13: Average collection delays at the sink over several collection rounds with enabled and disabled caching strategy using 10 TelosB nodes.

When the data collection starts, for each Interest message, only one node has a matching content that it propagates in response to the received request. As long as a Content Object is spreading over the network to reach the sink, it is replicated at the intermediate nodes and kept in CS. From the second data collection round, sink can get the requested content from many nodes (the content's origin or the replicating nodes). By keeping the first Content Object received and discarding the duplicates, the sink may ensure to satisfy its Interest with better delays.

3.5.3 SensLab results

In a next step, we measured collection delays using the sensLab platform using 10 and 20 nodes. Figure 15 depicts the CDF of the collection delays measured at the sink node. We observe that minimum delay is around 240 ms and the maximum delay is around 445 ms.

Data Collection Round	1st Round	2nd Round	3rd Round	4th Round	5th Round	6th Round	7th Round
Average Delay (ms)	440,11	310,9	317,8	258,67	262,2	260,8	260
Max Delay (ms)	547	461	453	265	265	273	265
Min delay (ms)	336	250	257	250	258	257	257

Table 3: Summary of collection delays on different rounds using 10 TelosB nodes and an enabled caching strategy.

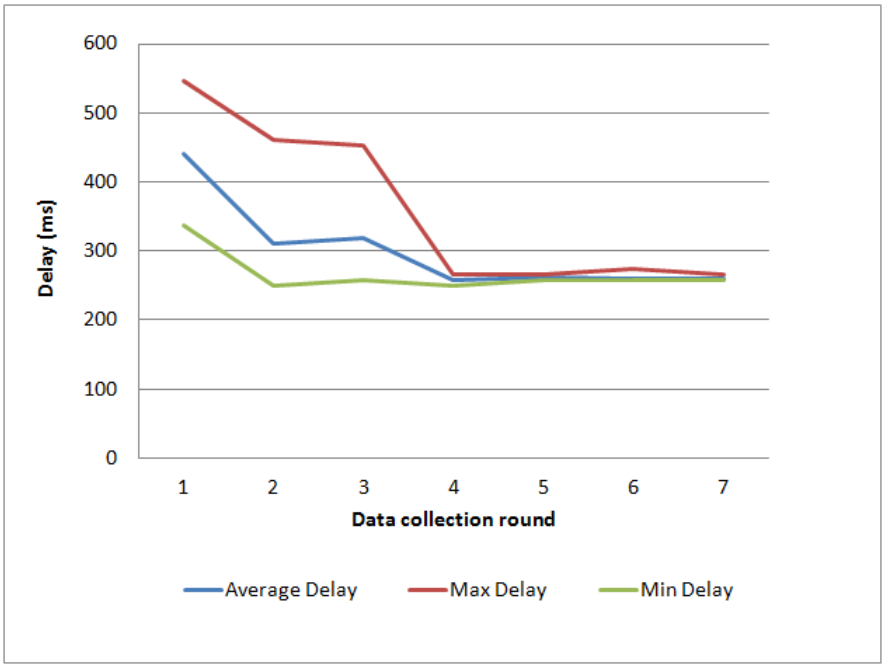


Figure 14: Measured collection delays at the sink with 10 TelosB nodes and an enabled caching strategy.

4 Limits and discussion

Simulations and testing on TelosB sensor nodes allowed us to validate and check the functioning of our implementation of CCN communication layer in Contiki. The CCN layer was tested successfully on top of the different available MAC and RDC drivers (contikimac, xmac, cxmac and sicslowmac). The current version of the implementation does not provide fragmentation and queuing mechanisms. Queues can be integrated on the level of Faces in the way CCNx does.
The CCN layer can transmit messages only in the broadcast mode at the MAC layer. Transmitting Interests and Content Objects in broadcast can lead to a loop problem. A node that generates or forwards an Interest message places it in the PIT table and waits until it receives a matching content or the pending Interest expires. Next, the pending Interest is removed from the PIT, allowing by that the forwarding of this Interest if it is received again. Figure 16 shows an example of a message exchange loop with only three nodes. In step 1, the sink sends an Interest message with prefix /Collect/N1 in broadcast. Node 2 receives the Interest and keeps it in PIT.

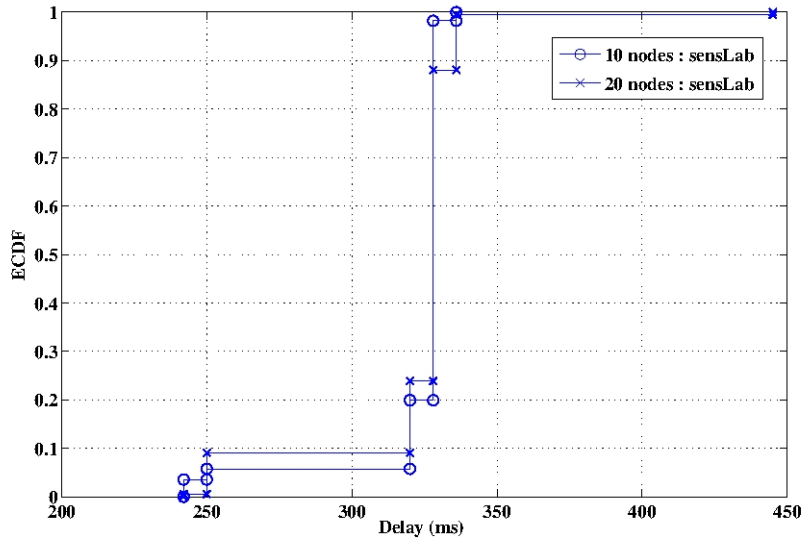


Figure 15: CDF of collection delay measured at the sink node under varying network sizes over the SensLAB platform.

Node 1 has a matching Content Object that is then forwarded. In step 2, the sink receives the Content Object and removes the pending Interest from its PIT. However, the Content Object sent from Node 1 to Node 2 is differed as the latter has scheduled to forward the Interest received in step 1. So node 2 forwards the same Interest message (step 3) and handles the Content Object received from Node 1 (step 4). After that, the sink, having received the Interest from Node 2 and cleared up its PIT table, forwards again the message (step 1) resulting in a communication loop. This problem was mainly observed with an important numbers of nodes.

We had some difficulties in running experiments using the SensLab platform. The WSN430 platform lacks many sensor drivers like light driver. In addition, a bug occurred repetitively in tests with more than 10 deployed nodes. Some collecting nodes start resetting or stop running for an unknown reason. The problem becomes more frequent when the network density increases. We observed also the same bug on some TelosB motes.

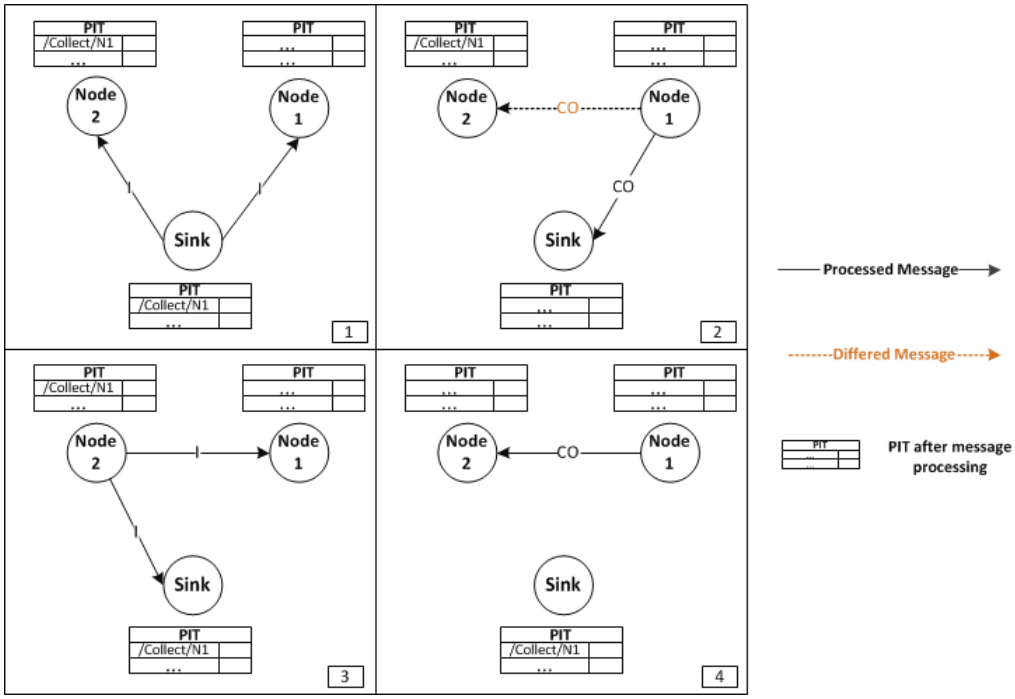


Figure 16: An example of the loop problem under CCN.

Conclusion

Wireless sensor networks are among the pioneer networks used for data collection in a variety of applications. Their capacities to efficiently cover different environments and provide monitoring information make them an essential element in the IoT vision of a global network of interconnected objects. By interconnecting wide-range devices around the world, the generation and exchange of information will become easier, rising in consequence the data availability and control for users and applications regardless of their location. In a such data-driven network, it may seem spontaneous to turn to information centric networking communication architecture. In this work, we focused on the Content Centric Networking, a communication architecture based on named data where data is exchanged independently of location. We succeeded in getting a resource limited wireless sensor network to run with a CCN communication layer. This layer, which was designed and implemented based on the CCNx protocol. It was integrated into the Contiki operation system and tested on TelosB and Sky motes. We evaluated the performance of our implementation using a data collection application with different scenarios and node configurations for the simulations and the experiments on physical sensor nodes. We also presented the limits of our current implementation.

The implemented CCN communication layer can be extended to offer more functionalities as provided by the reference CCNx implementation. By keeping the same general message formats, encoding/decoding and parsing routines, the integration of more CCNx message fields in the Interest message and the Content Object message is possible. The CCN layer may then add more specifications in Interests to target more specific contents and provide security and content authentication functions.

We recommend in particular to integrate some functions based on the CCNx basic name conventions for content's versioning and sequence numbering. Attributing versions to contents' names will help to efficiently manage and update the stored data. We also recommend to take advantage of the Functional/Command name components feature in CCNx to construct application-level protocols [4] such as data aggregation and processing that can be useful in data collection used for wireless sensor networks. In a larger scope, the CCN communication layer in Contiki can be extended to become interoperable with fixed networks running the CCNx protocol.

References

[1] http://www.ccnx.org. The CCNx project website.

[2] http://www.ccnx.org/releases/latest/doc/technical/binaryencoding.html. ccnb, the CCNx binairy encoding.

[3] http://www.ccnx.org/releases/latest/doc/technical/ccnxprotocol.html. The CCNx protocol presentation.

[4] http://www.ccnx.org/releases/latest/doc/technical/nameconventions.html. The CCNx name conventions.

[5] http://www.contiki-os.org. The Contiki OS website.

[6] http://www.senslab.info. The SensLab platform.

[7] http://www.sics.se/contiki/wiki/index.php/tmote_sky. The Sky platform in Contiki.

[8] http://www.willow.co.uk/telosb_datasheet.pdf. The TelosB mote platform.

[9] T. BERNERS-LEE, R. FIELDING, and L. MASINTER. Uniform resource identifier (uri): Generic syntax. *Internet Engineering Task Force RFC 3986*, pages 10 – 13, 2005. http://www.ietf.org/rfc/rfc3986.txt.

[10] A. DUNKELS. Full tcp/ip for 8-bit architectures. In *The 1st International Conference on Mobile Systems, Applications, and Services (MOBISYS 03)*, San Francisco, California, USA, 2003.

[11] A. DUNKELS, J. ERIKSSON, N. FINNE, and T. VOIGT. Cross-level sensor network simulation with cooja. In *First IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp 2006)*, Tampa, Florida, USA, November 2006.

[12] A. DUNKELS, B. GRÖNVALL, and T. VOIGT. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *The 1st IEEE Workshop on Embedded Networked Sensors (IEEE Emnets 2004)*, Tampa, Florida, USA, 2004.

[13] A. DUNKELS, F. ÖSTERLIND, and Z. HE. An adaptive communication architecture for wireless sensor networks. In *The 5th ACM Conference on Embedded Networked Sensor Systems (SenSys 2007)*, Sydney, Autralia, 2007.

[14] A. DUNKELS, O. SCHMIDT, T. VOIGT, and M. ALI. Protothreads : Simplifying event-driven programming of memory-constrained embedded systems. In *The 4th ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, 2006.

[15] J.F. GANTZ. The expanding digital universe: A forecast of worldwide information growth through 2010, 2007. An IDC White Paper.

[16] V. JACOBSON, D.K. SMETTERS, N.H. BRIGGS, J.D. THORNTON, M.F. PLASS, and R. L. BRAYNARD. Networking named content. In *The 5th ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT 2009)*, Rome, Italy, 2009.

[17] C. BURIN DES ROZIERS, G. CHELIUS, T. DUCROCQ, E. FLEURY, A. FRABOULET, A. GALLAIS, N. MITTON, T. NOEL, and J. VANDAELE. Using senslab as a first class scientific tool for large scale wireless sensor network experiments. In *Networking 2011*, pages 241–253, Valencia, Espagne, April 2011.

[18] J.P. VASSEUR and A. DUNKELS. *Interconnecting Smart Objects with IP*. Elsevier Inc., 2010.

Glossary

CCN	Content Centric Networking
CPU	Central Processing Unit
CS	Content Store
FIB	Forwarding Information Base
ICT	Information-Centric Networking
IoT	Internet of Things
IP	Internet Protocol
MAC	Media Access Control
MCU	Multipoint Control Unit
NDN	Named Data Networking
PIT	Pending Interest Table
RAM	Random Access Memory
RDC	Radio Duty Cycling
RF	Radio Frequency
ROM	Read Only Memory
TCP	Transmission Control Protocol
URI	Uniform Resource Identifier
USB	Universal Serial Bus
WSN	Wireless Sensor Network

A Appendix: source code files

- File ccn_stack.h

  Header file for the CCN stack.

- File ccn_stack.c

  CCN stack implementation.

- File ccn.h

  Header file for the CCN caches management and message treatment.

- File ccn.c

  Support for CCN caches management and message treatment.

- File uri.h

  CCN-scheme uri conversions.

- File ccn_uri.c

  Support for URI representation.

- File coding.h

  Details of the ccn binary wire encoding.

- File ccn_coding.c

  Support for scanning and parsing ccnb-encoded data.

- File ccn_buf_encoder.c

  Support for constructing various ccnb-encoded objects.

- File ccn_buf_decoder.c

  Support for Interest and ContentObject decoding.

- File ccn_name_util.c

  Support for manipulating ccnb-encoded Names.

- File charbuf.h

  Expandable character buffer for counted sequences of arbitrary bytes.

- File ccn_charbuf.c

  Support expandable buffer for counted sequences of arbitrary bytes.

- File indexbuf.h

  Expandable buffer of non-negative values.

- File ccn_indexbuf.c

  Support for expandable buffer of non-negative values.

- File definitions.h

- File ccn_driver.h

  Header for CCN Network driver.

- File ccn_driver.c

  CCN Network driver's implementation.

B Appendix: defined macros

- #define FACE_MAX_ENTRIES

  The maximum number of Faces.

- #define CS_MAX_ENTRIES

  The maximum number of Content Store entries.

- #define NAME_MAX_ENTRIES

  The maximum size of FIB table.

- #define PE_MAX_ENTRIES

  The maximum number of pending Interests in PIT.

- #define FORWARDS_MAX_ENTRIES

  The maximum number of forwarding entries.

- #define MAX_INTEREST_SIZE

  The maximum size of an Interest message.

- #define MAX_CONTENT_SIZE

  The maximum size of a Content Object message.

- #define MAX_BUF_DECODER_SIZE

  The maximum size of decoding buffer.

- #define PROPAGATION_TIMER

  The Interest lifetime.

  This defines how long a pending Interest is kept in PIT table. When timer reaches the Interest lifetime value, the Interest message is deleted.

- #define INDEXBUF_MAX_ENTRIES

  The maximum number of indexbuf buffers.

- #define MAX_INDEXBUF_SIZE

  The maximum size of an indexbuf buffer.

- #define CHARBUF_MAX_ENTRIES

  The maximum number of charbuf buffers.

- #define MAX_CHARBUF_SIZE

  The maximum size of a charbuf buffer.

- #define CCN_LINK_MTU

  The link level maximum transmission unit.

- #define CCN_BUF

  The CCN packet buffer.

  This buffer is used to hold incoming and outgoing packets. The device driver should place incoming data into this buffer. When sending data, the device driver should read the link level headers from this buffer.

- #define CCN_LLH_LEN

  The size of the link level headers.

- #define CCN_APPDATA_SIZE

  The buffer size available for user data in the CCN packet buffer.

- #define CCN_APPDATA_PTR

  The buffer available for user data in the CCN packet buffer.

C Appendix: used variables

- struct ccn_handle *h

  A pointer to the current handle.

- uint8_t ccn_buf[CCN_BUFSIZE]

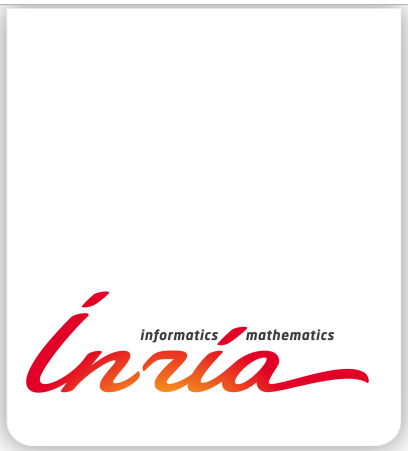
  The CCN packet buffer.

- void *ccn_appdata

  A pointer to the application data.

- u16_t ccn_len

  The size of the CCN packet buffer.



RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-0803