ccn$\mathcal{S}$im user manual

May 8, 2013

Contents

Chapter 1

Introduction

1.1 What is ccn$\mathcal{S}$im?

ccn$\mathcal{S}$im is a scalable chunk-level simulator of Content Centric Networks (CCN)[1], that we developed in the context of the ANR Project Connect.

- It is written in C++ under the Omnet++ framework.
- It allows to simulate CCN networks in scenarios with large orders of magnitude.
- It is distributed as free software, available at `http://site`.

ccn$\mathcal{S}$im extends Omnet++ as to provide a modular environment in order to simulate CCN networks. Mainly, ccn$\mathcal{S}$im models the forwarding aspects of a CCN network, namely the caching strategies, and the forwarding layer of a CCN node. However, it is fairly modular, and simple to . We hope that you enjoy ccn$\mathcal{S}$im in which case we ask you to please cite our paper [**?**].

1.2 ccn$\mathcal{S}$im and Oment++

Omnet++ is a C++ based event-driven simulator engine. Omnet++ is only an engine. It provides a set of core C++ classes to extend, in order to design the behaviour of a custom simulator. Besides, it provides `ned`, a network description language used to describe the simulation modules and how these modules interact each other (e.g., connections, module composition, end hence forth). During the rest of this manual, we refer to *modules* to indicate Omnet++ `ned` modules, while we talk of *class* to indicate the C++ data structures which implements the behaviour of these modules.

Our simulator, ccn$\mathcal{S}$im comes as a set of custom modules and classes that extend the Omnet++ core in order to simulate a CCN network. A ccn$\mathcal{S}$im simulation steps across three phases:

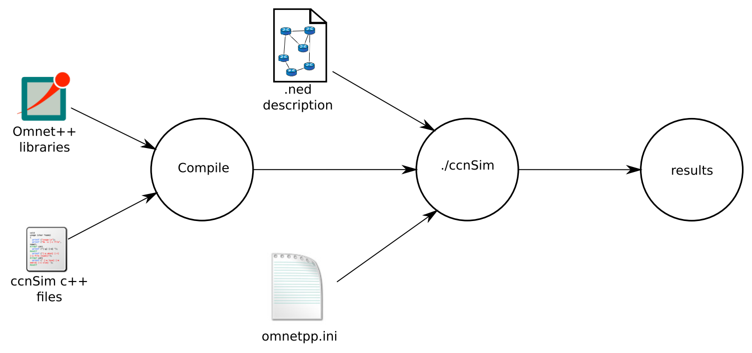

Figure 1.1: Overview of the whole compilation/execution process of ccn$\mathcal{S}$im.

- Compiling ccn$\mathcal{S}$im source files and linking with the Omnet++ core.
- Writing the description of the topology (usually the user will need only to set up connections between the CCN nodes).
- Initializing the parameters of each module. This can be done either directly from the `ned` files or from the `omnetpp.ini` initialization file.

We report the aforementioned steps in Fig. 1.1.

1.3 Overall structure of ccn$\mathcal{S}$im

In order to better understand the organization of ccn$\mathcal{S}$im, the best is to look at its internal organization. In the following we reproduce the basic directory organization of ccn$\mathcal{S}$im:

```
|-- topologies
|-- modules
|    |-- clients
|    |-- content
|    |-- node
|    |    |-- cache
|    |    |-- strategy
|    |-- statistics
|-- packets
|-- include
|-- src
|    |-- clients
|    |-- content
|    |-- node
|    |    |-- cache
|   |   |-- strategy
|   |-- statistics
```

As said within the introduction, ccn𝒮im is a package built over the top of Omnet++. As such, we can divide its implementation in two different subunits. One subunit is represented by the `ned` description of the Omnet++ modules, and included within the directory `modules` and `topologies`. The first directory, is basically the description of the operational modules employed by ccn𝒮im, like clients, nodes, and so forth.

Instead, within the `topologies` directory there are some sample topologies (in `.ned` format) ready to be used.

The real implementation of the Omnet++ modules lie into the `src` and `include` directory, which contain sources and header files, respectively.

1.4 Downloading and installing ccn𝒮im

You can freely download ccn𝒮im from the project site: `http://ccnsim.googlecode.com`.

We assume that you have downloaded and installed Omnet++ (version $\geq$ 4.1) on your machine. Indeed, the new version of ccn𝒮im makes use of the boost libraries, thus you should have a minimal boost installation on your system.

In order to install ccn𝒮im, it is first necessary to patch Omnet. Then, you can compile the ccn𝒮im sources. These steps are as follows:

```
    john:~$ cd CCNSIM_DIR
    john:CCNSIM_DIR$ cp ./patch/ctopology.h OMNET_DIR/include/
    john:CCNSIM_DIR$ cp ./patch/ctopology.cc OMNET_DIR/src/sim
    john:CCNSIM_DIR$ cd  OMNET_DIR && make && cd CCNSIM
    john:CCNSIM_DIR$ ./scripts/makemake
    john:CCNSIM_DIR$ make
```

In this snippet of code we suppose that `CCNSIM_DIR` and `OMNET_DIR` contain the installation directory of ccn𝒮im and Omnet++ respectively.

1.5 Organization of this manual

This manual is organized as follows:

- In Chap. 2 we give a description of the module organization of ccn𝒮im together with a brief description of the most important parameters that describe the simulation.

- In Chap. 3 there is a more technical description of ccn𝒮im, in terms of class implementation and design choices.

- Chap. 4 reports a brief ccn$\mathcal{S}$im tutorial. We will show how to simulate and extending ccn$\mathcal{S}$im.
- Finally, Chap. 5 shows some performance of the tool, its current issues, and further developments.

Chapter 2

The ccn$\mathcal{S}$im simulator

2.1 Topology definition

The `network` represents the top level module of a ccn$\mathcal{S}$im simulation. There, the user can define the connections between different CCN `nodes` modules. *Each network, has to extend the `base_network`*, in which the other modules (i.e., clients, statistics, and so forth) are defined. Besides, a topology specifies the placement of clients and repositories.

Clients placement Clients represent an aggregate of users: thus, at most one client is connected and active on a given node. Indeed, in ccn$\mathcal{S}$im clients are connected to each node of the network. The placement consists in specifying how many (and which) of them are active. The basic parameters for client placement are the following:

- `number_clients`: this integer value specifies how many clients are active over the network.
- `nodes_client`: comma separated string that specifies which CCN node has an active client connected to itself. The number of clients specified should be $\leq$ `number_clients`. If the number of clients specified is $<$ `number_clients` (this includes the case of an empty string) the remaining clients are distributed randomly across the network.

Repositories initialization In ccn$\mathcal{S}$im there is no real node representing a repository. A CCN node just *knows* owning a repository connected to itself. The distribution of repositories basically depends by two parameters:

- `number_repos`: integer value that specifies how many repositories should be distributed over the network.
- `nodes_repos`: comma separated string that specifies which CCN node has a repository connected to itself. The number of repositories specified should be at most `number_repos`. If the number of

repositories specified is $\leq$ `number_repos` (this includes the case of an empty string) the remaining repositories are distributed randomly across the network.

2.2 Content handling

The `content_distribution` module takes no part in the architecture itself, but accomplishes many crucial tasks for the correct working of the simulation.

Catalog initialization The *catalog* is a table of contents. Each content is described by these parameters.

- `cardF`: represents the cardinality of the catalog, expressed in number of contents.
- `F`: as the contents are distributed like a geometric distribution, this parameters represents the average size of the file, in chunks. Moreover, if `F` is set to one, whole objects are considered (each one composed by a single chunk).
- `replicas`: this parameters indicates the degree of replication of each content, and has to be less $<$ `num_repos`. In other words, the i-th content will be (randomly) replicated over exactly `replicas` repositories.

CDF initialization Lastly, the `content_distribution` has the task of generating the CDF of the content popularity distribution. For the time being, the only distribution implemented is a Mandelbrodt-Zipf. Besides the `cardF`, there are other two parameters which affect this CDF: `alpha` is the shaping factor of the MZipf, and `q` represents the MZipf plateau.

2.3 Nodes, Content Store, and Strategy Layers

The `node` module represents the ccn$\mathcal{S}$im core. It is a compound module, composed by three parts: the Core Layer, the Strategy Layer, and the Content Store. Each of these submodules are described below.

2.3.1 Core Layer

The `core_layer` module implements the basic tasks of a CCN node, and the communication with the other node's submodules. It handles the PIT, sending data toward the interested interfaces. It handles the incoming interests by sending back data (in the case of a *cache hit* within the Content Store), or by appending the interest to the existent PIT entry. In the case no entry exists yet it queries the *Strategy layer* in order to get the correct output interface(s). We could say that the `core_layer` takes the charge of the CCN hourglass thin waist.

2.3.2 Content Store

The Content Store represents the caching part of a CCN node. A Content Store is implemented by the means of a cache and its replacing algorithm. Thus, within the `node` module, the user can choose which type of caching using. This choice is fulfilled by the means of the `CS` parameter of the node compound module. The following is a brief description of the algorithms currently available within ccn$\mathcal{S}$im.

- `CS = lru_cache` implements an LRU replacement cache. It is the most used algorithm within the literature, and simply replaces the least recently used item stored within its cache.
- `CS = lfu_cache` implements an LFU replacement cache. By the means of counters an LFU cache may establish which is the leas popular content, and deleting it when the cache is full.
- `CS = random_cache` implements a random replacement cache. When the cache is full the canonical behaviour of a random replacement is to choice at random an element to evict.
- `CS = two_cache` implements an extension of LFU and random replacement. It takes two random elements, and then evicts the *least* popular one.
- `CS = fifo_cache` implement a basic First In First Out replacement. The first element entered in the cache is the first to be evicted, once the cache is full.

In the case of path of caches, the mere replacement algorithm is not enough. Indeed, connecting more caches arises the problem of caching coordination (who caches what?). Thus, a cache "uses" a decision algorithm (or meta caching algorithm) in order to decide if storing or not the incoming data. The `cache` parameter `CD` sets which decision algorithm employing for the given cache. In Sec. **??** we will show how to implement new kind of decision policies.

- `CD = lce` implements the Leave Copy Everywhere policy. Store each incoming chunk within the cache.
- `CD = lcd` implements the Leave Copy Down policy. Store each incoming chunk only if is the downstream node of the (permanent or temporary) retrieved copy.
- `CD = btw` implements the Betweenness Centrality policy. On a given path, only the node with highest betweenness centrality stores the chunk.
- `CD = fix(p)` implements the Fixed probability decision. The parameter `p` indicates the probability with which a given node stores the incoming chunk.

- `CD = never` disable caching within the network (useful only for debugging).

While decision policies basically decide *on a given path*, who caches what, the choice of deciding which path exploiting is left to the forwarding policy. In CCN a forwarding policy is named Srategy Layer. The module that implements this part of the architecture is described within the next section.

2.3.3 Strategy Layer

The strategy layer receives an interest for which no PIT entry exists yet. Then, it *decides* on which output face the interest should be sent. We suppose that each node knows both the network topology and the repositories which store permanent copies of the content (see also Sec. **??**). Recall that the repositories for a given content are stored within the catalog. There are different strategies actually implemented within ccn$\mathcal{S}$im. One particular repository can be chosen by setting the `SL` parameter of a `node` module.

- `SL = nearest_repository` The strategy layer choses the best out of the given set of repositories and then forwards among the right interface.

- `SL = random_repository` The strategy layer choses one repository at random out of the given set of repository. Note that this strategy requires that the core nodes follow the path chosen by the edge node (the node to which the client is attached to). Thus, we need a field within the interest message for setting the given *target* repository. Thus, it's only the edge node who chooses at random. The subsequent one follow the path by looking at the target within the message (i.e., we are just emulating a sort of source routing).

- `SL = dynamic_learning` In this last case, a node which receives an interest sets up an exploration phase, in which the node learns (by flooding the network) where the content is stored (it needs not to be a permanent copy, but even a *cached copy*). Then, when the content is found, the content is downloaded in unicast by the target node. Indeed, we use the same packet field as before, this time set by the target node.

2.4 Clients

As said above, a `client` represents an aggregate of users modeled as a Poisson process. In the current implementation, a client asks for files chunk by chunk (i.e., the *chunk window* W is fixed to one). The only parameter that characterizes a client is `lambda`, i.e., the (total) arrival rate of the Poisson process. Of course, this parameter can be set different for each client.

2.5 Statistics

The way in which statistics are taken in ccn$\mathcal{S}$im is rather complex. More in general, one of issue in taking statistics within a network of caches, is "when" starting to collect samples. One could start at time $t = 0$, taking into account the period of time in which the caches are still empty (*cold start*). Otherwise, we could wait for caches that fill up (*hot start*).

Moreover, when comparing simulations with models, often the system is supposed to be *stable*. In other words, statistics should be considered only after that the transient phase of the system is vanished. Identifying the transient of the system is not a simple task. In ccn$\mathcal{S}$im things work in the following way.

First, we wait that the nodes (or a subset of them) is completely full. After that, we wait for the system to be stable. Stabilization happens when the variance of the *hit probability* of each node goes below a threshold. This is implemented by sampling the hit probability of each node, and then calculating the variance of the samples collected. The parameters that affects the statistics calculations are:

- `ts`: the sampling time of the stabilization metric (i.e., the hit probability).
- `window`: the window of samples for which the variance is calculated.
- `partial_n`: the set of nodes for which waiting for filling and stabilization.
- `steady`: real duration of the simulation.

All the time variables here are expressed in seconds. For the sake of the example, let's suppose `window`=60s and `ts`=0.1s. That means: each 100ms a sample is collected. When 60s of samples are collected (i.e., 600 samples) the variance is calculated and tested against the threshold. The `partial_n` parameter is useful in the case which there are few clients and shortest path is used. In this case, some node could remain empty, and waiting for it would mean a infinite simulation. Besides the hit probability, the other statistics are handled per single module (e.g., per client or per CCN node).

- `p_hit` ($p_{hit} = \frac{n_{hit}}{n_{miss}+n_{hit}}$): it defines the probability of finding a content within the node.
- `hdistance`:represents the number of hops that an interest travels before hitting a copy of the requested chunk.
- `elapsed`: the total time for terminating a download of a file.
- `downloads`: the average number of downloads terminated by a given client.
- `interest` and `data`: the average number of Interest and Data messages (respectively) handled by a node of the network.

For most of these statistics it's possible to average on the different contents and/or on the different nodes within the network:

- Corse grained statistics: we output one single statistic averaged on every content and for every node (coarse grained statistics).
- Per node statistics: we output n statistics (where n represents the number of nodes of the network) for each node averaged on every content.
- Per content statistics: we output N statistics (where N represents the number of contents in the catalog) for each content averaged on all nodes of the network.
- Fine grained statistics: we output Nn statistics (fine grained statistics).

The output is collected in standard Omnet++ files. In particular, coarse grained, and per node statistics, are collected within the corresponding `.sca` vector file. Instead, fine grained and per content statistics are collected within the `.vec` file.

2.6 Summary

In this section we report the whole `.ini` file used to run simulation. In the next section we will see how to implement a simple simulation on a bus network, with one client and one repository placed at the extremity of the bus. Request arrives at a rate of We recall that the symbol `**.parameters` is syntactic sugar for indicating that the parameter is applied for each node who supports it (e.g., doing `**.C=1000` is the same of writing `node[i].cache.C = 1000` for `i` representing each node of the network).

```
[General]
network = networks.geant_network
seed-set = ${rep = 0}
output-vector-file =  ${resultdir}/foo/bar/ccn-id${rep}.vec
output-scalar-file =  ${resultdir}/foo/bar/ccn-id${rep}.sca

########################  Repositories #########################
**.node_repos = ""   #Specify repositories
**.num_repos  = 1    #Number of repositories
**.replicas = 1      #Content degree replication

##########################  Clients  ###########################
**.node_clients = ""   #Specify clients
**.num_clients = 1     #Number of clients
**.lambda = 1          #Poisson arrival rate
**.check_time = 0.1    #Checking time for completion
**.RTT = 1             #Max delay within the network

###################  Content Distribution ######################
**.F = 1          #Average size of  a content (in chunks)
**.alpha = 1      #Shaping factor of the Zipf distribution
**.cardF = 100    #Cardinality of the catalog

##########################  Caching  ###########################
**.CS = "lru_cache"      #Content Store implementation
**.CD = "always"         #Caching decision
**.C =  100              #Cache size (in chunks)

##########################  Strategy ###########################
**.SL = "random_repository" #Strategy layer (interest forwarding)

########################  Statistics ###########################
**.window = 60       #Time window for the stability checking
**.ts = 0.1          #Sampling hit_rate time
**.partial_n = 2     #Partial node filling
**.steady = 1000     #Simulation time
```

Chapter 3

ccn$\mathcal{S}$im under the hood

In this chapter we go deeper within the description of ccn$\mathcal{S}$im. At the end of this section the user will be able to grasp the ccn$\mathcal{S}$im source code, extending and customizing the simulator for her needs. This could be seen as a more programming perspective of ccn$\mathcal{S}$im. Indeed, in Chap. 2 we just described the `ned` part of ccn$\mathcal{S}$im. Recall that in Chap. **??** we mentioned as every Omnet `ned` module has a C++ class counterpart. Of course, we don't dive into about 10.000 lines of codes. We just give to the user what she needs in order to understand how things work. This knowledge will be sufficient for extending the basic ccn$\mathcal{S}$im.

3.1 Packets and chunks

Let's start the journey by the packets implemented in ccn$\mathcal{S}$im. Although Omnet follows the rule that *each event is a message*, data and interests represent the only packets defined in ccn$\mathcal{S}$im. Each of them carries a 64-bit identifier, which represents the *chunk*. The chunk is split in two 32bit fields. The most significant 4 bytes represent *the name* of the content. Instead, the *chunk number* is represented within the lowest significant part. Other fields within the packets may depend by different factors, like caching, or the strategy layer.

3.2 The client class

The `client` class represents an *aggregate of users*. The basic design is the following: users are modeled by the means of a C++ STL multimap, i.e., a map in which there could be multiple keys. The keys represent the name of different contents. The values are instead ccn$\mathcal{S}$im clients are connected by default to each node of the network. ccn$\mathcal{S}$im users may specify which client is active and which is not, and the rate of the incoming requests (distributed like a Poisson process, as we saw in Chap. 2).

3.3 The `content_distribution` class

The repositories are represented as a static array allocated by the Content Distribution class. Thus, at each moment each node knows where the repositories are located within the network (this will be relevant for the forwarding process, as we will see later on). 32-bits elements. Each elements is divided in two 16-bit entries. The most significant 16-bits of the i-th entry represents the size of the i-th file in chunks. The least significant 16-bits indicate the repositories which store a permanent copy of the content. This information is represented as a string of `num_repos` bits, where a 1 at the j-th positions means that the j-th repositories actually store the i-th content. In this sense the array of repositories acts like a look-up table which associates the value within the bit string to the right repository. Three parameters affects the catalog initialization: We implement the aforementioned CDF as another static array, and use binary search in order to retrieve elements with a given probability. In addition to `cardF` the other parameters that affects the behaviour of the Content Distribution class is `alpha` which represents the shaping factor of the Zipf distribution.

Chapter 4

Practical ccn$\mathcal{S}$im

4.1 Run your first simulation

It's time for practice. In this section you will learn how to run a simple simulation. We will focus on a simple bus network of fours nodes as shown in Fig. **??**, in which the client is placed on node 0, and the repository is placed within the node 3.

4.2 First step: defining the network

As the bus is a standard topology, we define the parameters that specify clients and repositories directly within the file `bus.ned`.

```
package networks;
network bus_network extends base_network{
    parameters:
        //Number of ccn nodes
            n = 4;

        //Repository on node 3
        node_repos = "3";
        num_repos = 1;
        replicas = 1;

        //Client on node 0
        num_clients = 1;
        node_clients = "0";
connections allowunconnected:
    //Bus connections
    for  i = 0..n-2{
        node[i].face++ <--> {delay = 1ms;} <--> node[i+1].face++;
    }
}
```

Note that the `bus_network` simply extends the `base_network`. Indeed, `base_network` represents a simple single node network, and it's there where modules like nodes, clients, and statistics get initialized. It's mandatory that every new network topology extends this base network, otherwise basic modules do not get initialized, and the simulations would probably die.

4.3 Extending ccn$\mathcal{S}$im

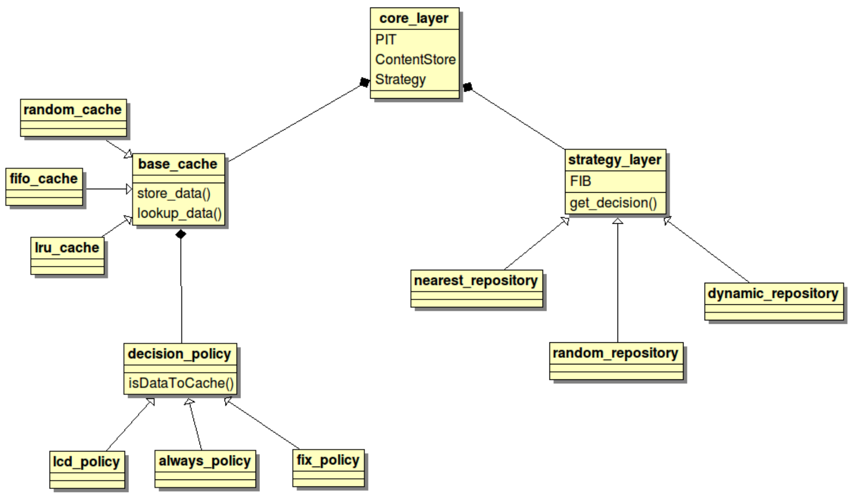

Figure 4.1: Essential class organization of ccn$\mathcal{S}$im.

- The total arrival rate of the client is of $\lambda = 1req/s$. The catalog is composed by 1000 items, each one single sized, their popularity distributed like a Zipf with shaping factor $\alpha = 1$.
- Caches are of 10 to 100 chunks or objects (in this case these two quantities are the same), which represent the 1% and th 10% of the catalog, respectively. We will use LRU caches, which store every data they receive (LCE caching decision).
- We want to see how the (coarse grained) distance varies as function of the cache size.

Chapter 5

ccn$\mathcal{S}$im performance

Bibliography

[1] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. Networking named content. In *ACM CoNEXT*, page 112, 2009.