Contents

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, downloadable 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 [**?**]. ccn$\mathcal{S}$im is able to simulate content stores up to 106 chunks and catalog sizes up to 108 files in a reasonable time.

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. Moreover, it provides a network description language (`ned`) in order to describe how the custom modules interact each other.

A ccn$\mathcal{S}$im simulation comes across three steps:

- Compile ccn$\mathcal{S}$im source files with the Omnet++ core classes.
- Write the description of the topology (usually the user will need only to set up connections, and defining the cardinality of the network graph).
- Initialize parameters of each module. This can be done either directly from the `ned` files or from the verb—omnetpp.ini— file.

We report the aforementioned steps in Fig. **??**.

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, ccn$\mathcal{S}$im 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 modules used by ccn$\mathcal{S}$im, and included within the directory `modules` and `topologies`. The first directory, is basically the `.ned` description of the modules employed by ccn$\mathcal{S}$im, like clients, nodes, and so forth. Within the `topologies` directory there are some sample topologies (always descripted in `.ned` format) ready to be used.

The real implementation of these modules lie within the `src` and `include` directory, that contain sources and header files, respectively. Within the rest of this section we summarily describe the features of these components, together with a brief overview of their most important parameters.

1.4 Downloading and installing ccn$\mathcal{S}$im

You can freely download ccn$\mathcal{S}$im from the project site: `http://ccnsim.googlecode.com`.

We assume that you have downloaded and installed Omnet++ (version $\geq$ 4.1) on your machine. The new version of ccn$\mathcal{S}$im makes use of the boost libraries, thus you should have a minimal boost installation on your system.

In order to install ccn$\mathcal{S}$im, it is first necessary to patch Omnetpp. Then you can compile the ccn$\mathcal{S}$im 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 ccn$\mathcal{S}$im and Omnet++ respectively.

1.5 Organization of this manual

This manual is organized as follows:

- Organization.

2 ccn$\mathcal{S}$im in details

2.1 Packets and chunks

Data and interests packets are the only type of messages implemented within ccn$\mathcal{S}$im. Each of them carries a 64-bit identifier, named *chunk*. The name of the file carried by the packet is stored as a 32-bit flat identifier within the most significant 4-byte of the chunk. Instead, the chunk identifier is carried within the lowest significant 4-bytes as a 16-bit unsigned integer. Other fields within the packets depend by different factors, like caching, or the strategy layer.

2.2 The Content Distribution

The content distribution is a node which takes no part in the architecture itself, but accomplishes many different tasks crucial for the correct working of the simulation.

Clients initialization In ccn$\mathcal{S}$im clients represent an aggregate of users: thus at most one client is connected and active on a given node. The basic design is the following. 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 will see later). The basic parameters for clients initialization are two:

- `number_clients`: integer value that specifies how many clients are active over the network.
- `nodes_client`: ","-separated string that specifies which CCN node has an active client connected to itself. The number of clients specified should be at most `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 that it has a repository attached to itself. The distribution of repositories among the network is done within the Content Distribution module, and basically depends by two parameters:

- `number_repos`: integer value that specifies how many repositories should be distributed over the network.

- `nodes_repos`: ","-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.

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).

Catalog initialization The *catalog* is a (huge) array of 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:

- `F`: as the contents are distributed like a geometric distribution, this parameters represents the average size of the file, in chunks. 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. As extreme case, if `replicas` = `num_repos` each repository will store one copy of the given content.
- `cardF`: this parameters represents the cardinality of the catalog, expressed in number of contents.

Popularity distribution initialization Lastly the Content Distribution class generates the CDF of the content popularity distribution. Right by now, the only distribution implemented is a Zipf distribution. 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.

2.3 Nodes, caches, and strategy layers

The `node` module is the core part of the ccn$\mathcal{S}$im. `node`s form the core network and can be connected each other by the means of `faces`(see [1] for better under-

standing what a *face* is). However, nodes own not a real C++ implementation. Indeed, a node is a compound module, composed by three parts: the Core Layer, the Strategy Layer, and the Content Store, described below.

2.3.1 Core Layer

The core layer implements the basic tasks of a CCN node. Indeed, it handles the PIT, sending data back to the interested interfaces. In handles the incoming interest by replying to it (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 dispatches it to the *Strategy layer* in order to get a decision about where (i.e., on which interface) sending it. We could say that the core layer takes in charge the thin waist of CCN hourglass.

2.3.2 Strategy Layer

The strategy layer receives an interest for which no PIT entry exists yet. Then, it should *decide* on which 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. 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.

2.3.3 Content Store

The Content Store represents the caching part of a given node. The content replacement algorithm fully characterize a cache. In that sense, we can say that a cache "is" an LRU cache, an LFU cache and so on. Thus, within the

`node` module, one can choose which type of caching using within the given node setting the `CR` parameter.

- `CR = lru_cache` This is the classical algorithm used within
- `CR = lfu_cache` This is the classical algorithm used within
- `CR = two_cache` This is the classical algorithm used within
- `CR = fifo_cache` This is the classical algorithm used within

Furthermore, a cache "uses" a decision algorithm (or meta caching algorithm) in order to decide if storing or not the incoming data. The Content Store's parameter `CR` sets which decision algorithm employing for the given cache.

- `CD = lce`
- `CD = lcd`
- `CD = btw`
- `CD = fix(R)`
- `CD = prob`
- `CD = never`

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 characterize a client is `lambda`, the (total) arrival rate of the Poisson process. 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 quite complex. The first problem in taking statistics within a network of caches, is "when". We could start takings statistics at time $t = 0$, but we would take into account the period of time in which the caches are still empty (cold start). We could wait for caches fill up (hot start).

Besides, 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 phase within 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`
- `window`
- `partial_n`
- `steady`

Beside the hit probability, the other statistics are handled locally by each module. The `statistics` module at the end of simulations query the different nodes and collect the different statistics:

- `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.

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

- Per network: we output one single statistic averaged on every content and for every node (coarse grained statistics).
- Per node: we output n statistics (where n represents the number of nodes of the network) for each node averaged on every content.
- Per content: we output N statistics (where N represents the number of contents in the catalog) for each content averaged on all nodes of the network.
- Per content and per node: 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 `.vec` vector file. Instead, fine grained and per content statistics are collected within the `.sca` 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).

2.7 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.

- 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.

2.8 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.

3 Extending ccn$\mathcal{S}$im

References

[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.