ccn$\mathcal{S}$im user manual

September 5, 2013

Contents

1 ccn$\mathcal{S}$im overview **1**

1.1 Introduction . . . 1

1.1.1 What is ccn$\mathcal{S}$im? . . . 1

1.1.2 ccn$\mathcal{S}$im and Oment++ . . . 1

1.2 Overall structure of ccn$\mathcal{S}$im . . . 2

1.3 Downloading and installing ccn$\mathcal{S}$im . . . 3

1.4 Organization of this manual . . . 4

2 The ccn$\mathcal{S}$im simulator **5**

2.1 Topology definition . . . 5

2.2 Content handling . . . 6

2.3 Nodes, Content Store, and Strategy Layers . . . 6

2.3.1 Core Layer . . . 7

2.3.2 Caching strategies . . . 7

2.4 Clients . . . 9

2.5 Statistics . . . 9

3 Extending ccn$\mathcal{S}$im **11**

3.1 Extending ccn$\mathcal{S}$im . . . 11

4 Practical ccn$\mathcal{S}$im **13**

4.1 Run your first simulation . . . 13

4.2 Simulating ranges . . . 17

Chapter 1

ccn$\mathcal{S}$im overview

1.1 Introduction

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

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

- 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://www.enst.fr/~drossi/ccnsim`

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 extend. 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 10^6 chunks and catalog sizes up to 10^8 files in a reasonable time.

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

Omnet++ is a C++ based event-driven simulator engine. Omnet++ is a bare environment. 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.

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:

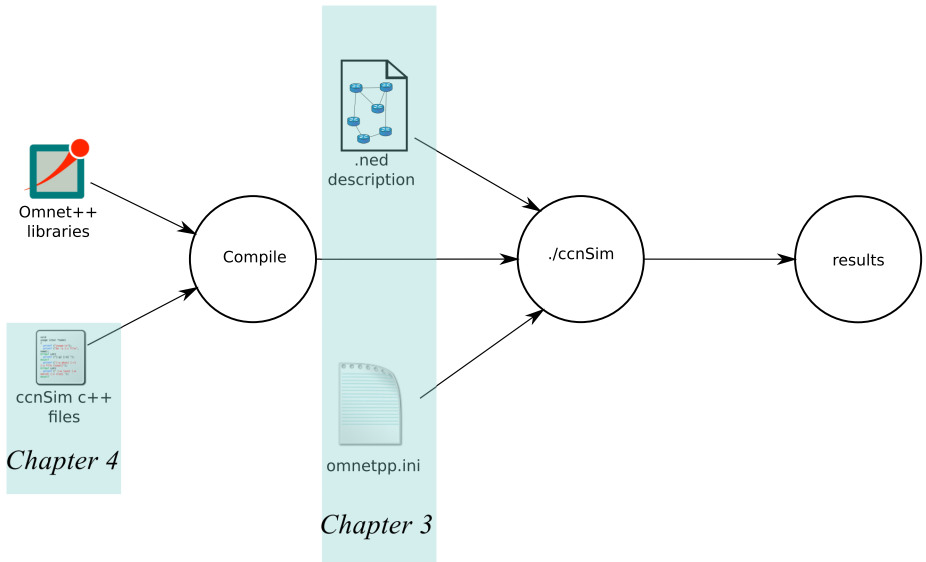

Figure 1.1: Essential class organization of ccn$\mathcal{S}$im.

- 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. **??**. In the remainder of this manual we assume the reader has a basic knowledge of the Omnet++ environment. Otherwise we invite the interested reader to give a look at [**?**].

1.2 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: 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 Omnet++ modules, and included within the directory `modules` and `topologies`. The first directory, is basically the description of the operational modules employed by ccn$\mathcal{S}$im, like clients, nodes, and so forth.

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

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

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

1.3 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. Indeed, 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 Omnet. 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.

1.4 Organization of this manual

This manual is organized as follows:

- In Chap. 2 we give a description of the module organization of ccn$\mathcal{S}$im together with a brief description of the most important parameters that describe the simulation.
- In Chap. 3 there is a more technical description of ccn$\mathcal{S}$im, 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. **??** shows some performance of the tool, its current issues, and further developments.

Chapter 2

The ccn$\mathcal{S}$im simulator

In the following we give an Omnet++ perspective of ccn$\mathcal{S}$im. No C++ code will be shown. The user who wishes just to learn how to run a simulation can read only this chapter. We remember that the following parameters can be set both from the `.ini` and from within the `.ned` description file of each module[?]. Most of the modules illustrated have a corresponding C++ class illustrated in Chap. 3.

2.1 Topology definition

The `network` represents the top level module of a ccn$\mathcal{S}$im simulation. In there, the user should define the connections between different CCN `node`s modules and the placement of clients and repositories as well as the number of nodes within the network used. Each network module, MUST extend the `base_network`, in which the other modules (i.e., clients, statistics, and so forth) are defined.

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* that he owns 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.

- `objects`: represents the cardinality of the catalog, expressed in number of contents.
- `file_size`: 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 For the time being, the only distribution implemented is a Mandelbrodt-Zipf. The `content_distribution` module (corresponding to the `content_distribution` class, see Chap. 3) takes, besides the number of objects (`objects` parameter), other two parameters: `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 compounded by three other submodules: the `core_layer`, the `strategy_layer`, and the `content_store` modules. Each of these submodules are described below jointly with the parameters they accept.

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).

2.3.2 Caching strategies

We can think to a caching algorithm on a CCN network as a triple $\langle \mathcal{F}, \mathcal{D}, \mathcal{R} \rangle$. The forwarding strategy $\mathcal{F}$ determines which path exploiting, i.e., where the given Interest has to be sent. The decision policy $\mathcal{D}$ returns a boolean value saying if the current node on the path has to cache or not the given data. The replacement $\mathcal{R}$ drops an element from a full cache to make room for the current element. In the following, we explain how to set each element of this triple within ccn$\mathcal{S}$im.

Forwarding strategies - FS

The forwarding strategy 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 `FS` parameter of a node module.

Shortest Path Routing - `FS = spr` The strategy layer choses the shortest path repository and sends packets on the corresponding interface.

Random Repository - `FS = 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).

Nearest Replica Routing (Two phases) - `FS = nrr` With this setting, the strategy layer first explores the neighboring nodes by flooding meta-interest (i.e., interests which do not change the content of the cache) with a given TTL. Then, the strategy sends the interest packet toward the nearest nodes having the content available. The TTL can be set by the means of the `TTL1` parameter.

Nearest Replica Routing (One phase) - `FS = nrr1` In this last case, a node which receives an interest sets up an exploration phase, in which the node floods the neighboring nodes with the request for the given object. When the copy is found the data comes back (it can be a permanent copy, or a *cached copy* as well). The scope of the flooding can be set by the means of `TTL2` parameter.

Decision Strategies

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.

- `DS = lce` implements the Leave Copy Everywhere policy. Store each incoming chunk within the cache.
- `DS = lcd` implements the Leave Copy Down policy. Store each incoming chunk only if is the downstream node of the (permanent or temporary) retrieved copy.
- `DS = btw` implements the Betweenness Centrality policy, proposed in [**?**]. On a given path, only the node with highest betweenness centrality stores the chunk.
- `DS = fix(p)` implements the Fixed probability decision. The parameter `p` indicates the probability with which a given node stores the incoming chunk.
- `DS = prob_cache` implements the ProbCache strategy proposed in [**?**]. As much a node is far from the (either temporary or permanent) copy, the less is the probability of caching the given element.
- `DS = never` disable caching within the network (useful only for debugging).

Replacement strategies

Finally, within the `node` module, the user can choose which type of caching using. This choice is fulfilled by the means of the `RS` parameter of the node compound module. The following is a brief description of the algorithms currently available within ccn$\mathcal{S}$im.

- `RS = 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.
- `RS = 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.

- `RS = 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.

- `RS = two_cache` implements an extension of LFU and random replacement. It takes two random elements, and then evicts the *least* popular one.

- `RS = 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.

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 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:

- Coarse grained statistics: we output one synthetic value averaged on every content and for every node (coarse grained statistics).
- Per node statistics: we output n curves (where n represents the number of nodes of the network), one for each node. Each curve is averaged on every content.
- Per content statistics: we output N curves (where N represents the number of contents in the catalog), one for each content. Each curve is averaged on all the nodes of the network.
- Fine grained statistics: we output Nn curves (fine grained statistics), one for each node and each different content of the system.

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.

Chapter 3

Extending ccn$\mathcal{S}$im

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 the 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 Extending ccn$\mathcal{S}$im

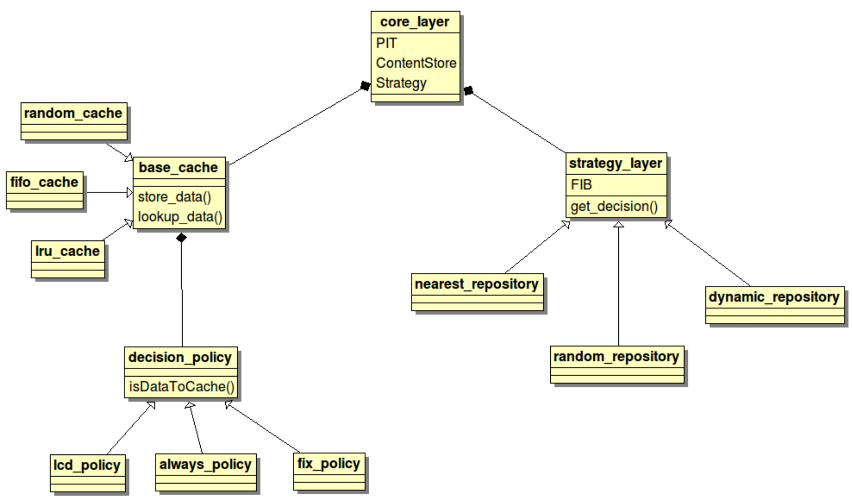

Figure 3.1: Essential class organization of ccn$\mathcal{S}$im.

Chapter 4

Practical ccn$\mathcal{S}$im

In this chapter, we will go through a complete simulation of a CCN. We simulate the following scenario:

- A general network of caches: Abilene.
- Clients are connected at each node.
- The client's arrival rate is $\lambda = 1req/s$. It is the same for every node in the network.
- The catalog is composed by 1000 objects, each one single sized.
- Object popularity is distributed like a Zipf with shaping factor $\alpha = 1$.

4.1 Run your first simulation

The first step for running a simulation is to define the network. As mentioned in Chap. 2, connections, and number of nodes are easily defined within a `.ned` file. In Snippet 1 we report the whole `ned` file which describes the network. As said it suffices extending the `base_network` network, and specifying number of nodes and connections. The package `network` represents the actual directory where Omnet looks for the `ned` file (in this case `CCNSIM_DIR/networks`). Note that `abilene_network` extends the `base_network`. Indeed, as mentioned within Chap. 2, `base_network` represents a simple single node network, and it's there where modules like nodes, clients, and statistics got initialized. It's mandatory that every new network topology extends this base network, otherwise such basic modules do not get initialized, and the simulations would not have the chance to be executed.

The next step is to set the `ini` file with the correct parameters. We set the network name to Abilene (the network defined in Snippet 1). The parameters are set following the indication at the beginning of the parameter. If the reader has already read Chap. 2, understanding the `ini` file in Snippet 2 should

```
package networks;
network abilene_network extends base_network{
parameters:
    n = 11;
connections allowunconnected:
    node[1].face++ <--> { delay = 5.48ms; } <--> node[0].face++;
    node[10].face++ <--> { delay = 3.80ms; } <--> node[0].face++;
    node[10].face++ <--> { delay = 5.02ms; } <--> node[1].face++;
    node[10].face++ <--> { delay = 1.68ms; } <--> node[9].face++;
    [...]
}
```

Snippet 1: Ned file for the simulation.

not be difficult. Notwithstanding, it's worth do spend some words for better understanding Snippet 2.

The `$` notation is used within the `ini` in order to define `ini` variables. `ini` variables are handy for dealing with ranges of values for the simulation parameters. Let's suppose that we want simulate different α values, let's say from 0.5 to 1. One choice could be running ccn$\mathcal{S}$im different time, each time with a different value of `alpha`. Note that in this case we should take care of specifying different files for each execution. Instead, by the means of the `$` notation, we can easily write:

```
    **.alpha = ${a = 0.1...1 step 0.1}
```

In this way Omnet++ will execute 10 different runs of the same scenario, each with a different α. The variable `a` is used for specifying (the directory of) the output file name (`output-scalar-file` and `output-vector-file`). In this way each simulation run, will be hold on a different file. This notation has been used in Snippet 2 for: random generator seed (`rep`), shaping factor(`a`), forwarding(`F`), decision (`D`) and replacement strategy (`R`). Just recall that the name of the `ini` variable should not forcedly correspond to the ccn$\mathcal{S}$im parameter.

Now, let's try to execute the simulation. The simpler method (the more complex one will be explained in a while) is to write the following.

```
    john:CCNSIM\_DIR$ ./ccnSim -u Cmdenv
```

The option `-u Cmdenv` is for executing ccn$\mathcal{S}$im in console mode (without graphic engine). In Snippet 3 we report the result of the execution, skipping out uninteresting parts. As we can see, caches are being filled after 134 secs the simulation is initialized. Then, ccn$\mathcal{S}$im waits for the variance of each node's `p_hit` to be under a given threshold. That happens after the second round, thus clearing out the old statistics. After about 1800 secs, the simulator stops, flushing on the console

```
#General paramteters
[General]
network = networks.${net=abilene}_network
seed-set = ${rep = 0}
########################  Repositories #######################
**.node_repos = ""
**.num_repos  = 1
**.replicas = 1
##########################  Clients  ##########################
**.node_clients = ""
**.num_clients = 11
**.lambda = 1
**.RTT = 2
**.check_time = 0.1
###################  Content Distribution #####################
**.file_size =  1
**.alpha = ${a = 1}
**.objects = 10^4
##########################  Forwarding #######################
**.FS = "${F = spr }"
**.TTL2 = 1000
**.TTL1= 1000
**.routing_file = ""
##########################  Caching  ##########################
**.DS = "${ D = lce }"
**.RS = "${ R = lru }_cache"
**.C =  10^2
########################  Statistics #########################
**.window = 60
**.ts = 0.1
**.partial_n = -1
**.steady = 3600/2
###############################################################
output-vector-file =
${resultdir}/${net}/F-${F}/D-${D}/R-${R}/a-${a}/ccn-id${rep}.vec

output-scalar-file =
${resultdir}/${net}/F-${F}/D-${D}/R-${R}/a-${a}/ccn-id${rep}.sca
```

Snippet 2: `omnnetpp.ini` for the simulation.

```
[. . . ]

Setting up network ‘networks.abilene_network’...
Initializing...
Initializing Zipf...
Initialization ends...
Start content initialization...
Content initialized

Running simulation...
** Event #1   T=0   Elapsed: 0.000s (0m 00s)
     Speed:     ev/sec=0   simsec/sec=0   ev/simsec=0
     Messages:  created: 25   present: 25   in FES: 23
Caches filled at time 134.6
0] variance = 0.113934
1] variance = 0.0499831
2] variance = 0.0846741
[. . . ]
0] variance = 0.0110749
1] variance = 0.0064429
2] variance = 0.00657124
[. . . ]
** Event #372608   T=2054.6   Elapsed: 1.040s (0m 01s)
Speed:ev/sec=358277   simsec/sec=1975.58   ev/simsec=181.353
Messages:  created: 121187   present: 22   in FES: 22
[ . . . ] Simulation stopped with endSimulation().
Calling finish() at end of Run #0...
p_hit/cache: 0.161627
Distance/client: 2.20726
Time/client: 0.011430159267
Downloads/client: 1821.64

End.
```

Snippet 3: Execution of the simulation

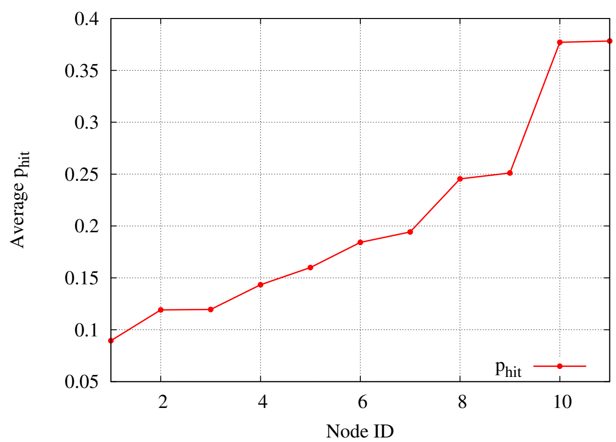

Figure 4.1: Simulation result: `p_hit` per node.

some of the synthetic values (saved in the `sca` file). If we go through the directory `CCNSIM_DIR/results/abilene/F-spr/D-lce/R-lru/alpha-1/` we would find three files with the same name `ccn-id0` and different extensions. As said in Chap. 2, the `sca` file contains the coarse and per node metrics. Let's say we would produce the `p_hit` for each node of the network. Moreover we want sort the output in ascending value. The following snippet of code, is apt to do that:

```
john:CCNSIM_DIR/results$ grep ``p_hit\['' [...]/ccn-id0.sca|
>  awk '{print $4}' |
>  sort -n >  p_hit.data
```

The plot of the file `p_hit.data` is shown in Fig. 4.1. In pasting and copying the previous snippet, take care of substituting the "[. . .]" with the actual directory `CCNSIM_DIR/results/abilene/F-spr/D-lce/R-lru/alpha-1/`.

4.2 Simulating ranges

As the last part of this brief ccn$\mathcal{S}$im tutorial, we deal with simulating different decision strategies $\mathcal{D}$for $\alpha \in 0, 1$. In particular, we are going to simulate `lce`, `lcd`, `prob_cache`, and `fix0.1`. In this last case, positive decision is taken one time over ten times, or, otherwise stated, on a path long 10 hops, only one node will cache the data. We will consider the average download distance `hdistance` as coarse grained metric. It's turn out that, by exploiting the `$` notation, we can simulate this scenario in one single shot. First of all, we have to modify the `ini` file in the following way.

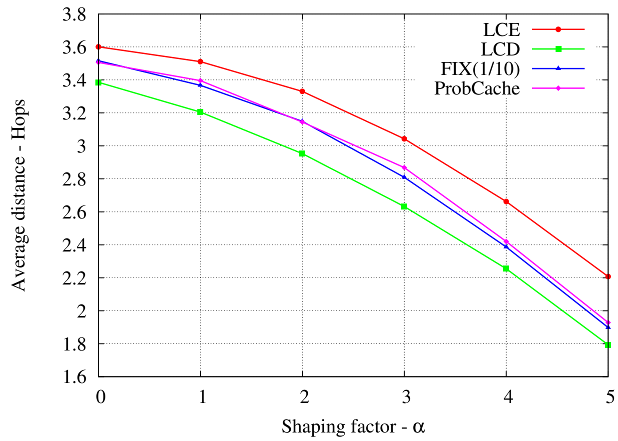

Figure 4.2: Comparison of different decision strategies for different values of α.

```
[ . . .]
**.alpha = ${a = 0.5...1 step 0.1}
[ . . . ]
**.DS = ${D = lce,lcd,prob_cache,fix0.1}
[ . . .]
```

As we see, we have a bunch of 24 simulations to run. Doing `./ccnSim -u Cmdenv` would execute the 24 runs in sequence. If we are running ccn$\mathcal{S}$im on a machine more than a single processor, we would like to run a bunch of runs in parallel. The `runall.sh` script in the directory `CCNSIM_DIR/scripts`, launches the simulations in parallel on different processor. For modifying the number of processor, it suffices to open the file with a suitable text editor and modify the option `-j` of the command. That said, by writing:

```
john:CCNSIM_DIR$ ./scripts/runall.sh
```

The whole set of simulations is executed in parallel.

Now, we have a bunch of 72 files. Indeed, for each different strategy we simulate 6 different values of α, generating a `vec`, a `sca` and a `vci`. Each file is in the right directory with the name of the corresponding strategy and the corresponding α value. Let's start evaluating the performance in terms of the (coarse grained) average download distance. As before some bash scripting does the magic:

```
john:CCNSIM_DIR/results$ grep ‘‘distance [...]/ccn-id0.sca|
> | awk ’{print $4}’  > distance_lcd.data
```

Note that we are considering the LCD (`lcd`) decision strategy and that the string replacing the "[...]" MUST be `abilene/F-spr/D-lcd/R-lru/alpha-*`. Indeed, for the given decision policy (in this case the LCD policy) we need all the different values for each different shaping factor. Repeating the above snippet for each decision strategy, and plotting the result, we have Fig. 4.2.