Contents

1 ccnSim overview
  1.1 What is ccnSim? ........................................... 1
  1.2 ccnSim and Oment++ ...................................... 1
  1.3 Overall structure of ccnSim .............................. 2
  1.4 Downloading and installing ccnSim ..................... 3
  1.5 Organization of this manual ............................... 4

2 The ccnSim 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
  2.6 Summary ............................................... 11

3 Extending ccnSim ............................................ 13

4 Practical ccnSim ............................................ 15
  4.1 Run your first simulation ............................... 15
  4.2 Simulating ranges ...................................... 19
  4.3 Summary ................................................. 21
Chapter 1

ccnSim overview

1.1 What is ccnSim?

ccnSim is a scalable chunk-level simulator for Content Centric Networks (CCN)[5], 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. In particular, ccnSim is able to simulate content stores up to $10^6$ chunks and catalog sizes up to $10^8$ files in a reasonable time.
- It is distributed as free software, downloadable at \url{http://www.enst.fr/~drossi/ccnsim}

Roughly speaking, ccnSim extends Omnet++ as to provide a modular environment in order to simulate CCN networks. Mainly, ccnSim 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 ccnSim in which case we ask you to please cite our paper [4].

1.2 ccnSim and Omnet++

Omnet++ is a C++ based event-driven framework. It provides: i) a set of core C++ classes, to extend for designing the behaviour of a custom simulator; ii) a network description language (ned) to describe how the custom modules interact each other; iii) a msg language defining the messages node will exchange.

Our simulator comes as a set of custom modules and classes that extend the Omnet++ core in order to simulate a CCN network. A ccnSim simulation steps traverses three phases:

- Compiling ccnSim 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.

• Executing the simulation.

We report the aforementioned steps in Fig. 1.1. 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].

1.3 Overall structure of ccnSim

In order to better understand ccnSim structure, we provide a look at its internal directory structure:

As said within the introduction, ccnSim is a package built over the top of Omnet++. As such, we can divide its structure in three different subunits.

Module properties are defined in its ned file and simply specify the module appearance. For instance, for a cache we could specify its size, or its replacement algorithm. These files are all contained within the modules directory.
Module behaviours represent ccnSim core and are defined as C++ classes contained within the src and include directories (sources and headers, respectively). Roughly, for each module in the modules directory a C++ class is defined specifying its behaviour.

Packets Data and interest packets of the CCN architecture are defined within the packet directory. Messages are specified in the msg syntax.

Finally, the topologies directory contains a set of sample topology that the user can employ for starting with ccnSim (an example with Abilene network will be provided in Chap. 4.

1.4 Downloading and installing ccnSim

You can freely download ccnSim from the project site: http://ccnsim.googlecode.com or from the ccnSim home page http://www.enst.fr/~drossi/ccnsim.

We assume that you have downloaded and installed Omnet++ (version ≥ 4.1) on your machine. Indeed, the new version of ccnSim makes use of the boost libraries[2], 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:
In this snippet of code, and for the remainder of the document, we suppose that `CCNSIM_DIR` and `OMNET_DIR` contain the installation directory of ccnSim and Omnet++ respectively.

1.5 Organization of this manual

This document 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 ccnSim tutorial. We will show how to simulate and extending ccnSim.
Chapter 2

The ccnSim simulator

In the following we give an Omnet++ perspective of ccnSim. No C++ code will be shown. The user who wants 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[1]. 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 ccnSim simulation. In there, the user should define the connections between different CCN nodes 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 ccnSim 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 ≤ 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 ccnSim 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 functioning 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 parameter represents the average size of the file, in chunks. Moreover, if **file_size** is set to one, whole objects are considered (each one composed by a single chunk).
- **replicas**: this parameter indicates the degree of replication of each content, and has to be less \(< \text{num_repos}\). 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 ccnSim 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. NODES, CONTENT STORE, AND STRATEGY LAYERS

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 in 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 \( \langle F, D, R \rangle \). The forwarding strategy \( F \) determines which path exploiting, i.e., where the given Interest has to be sent. The decision policy \( D \) returns a boolean value saying if the current node on the path has to cache or not the given data. The replacement \( R \) drops an element from a full cache to make room for the incoming (cacheable) element. In the following, we explain how to set each element of this triple within ccnSim. We recall that these are pa

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 make the assumption that each node knows the location of the permanent copy of each content. There are different strategies actually implemented within ccnSim. One particular strategy 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. Setting TTL1 to \( \infty \) (i.e., greater than the network diameter) makes NRR degenerating in the ideal NRR (iNRR) strategy, which explore the entire network, looking for a copy of the given content.

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 neighborhood 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.

Note that the routing table is learned at run-time by the mean of the Dijkstra algorithm. It is possible passing the routing matrix as an ASCII file, specified by the file_routing parameter of the strategy_layer module.

Decision Strategies

By connecting more caches there arises the problem of caching coordination (who caches what?). Thus, a cache uses a decision strategy (or meta-caching algorithm) in order to decide if storing or not the incoming data. The parameter CD sets which decision algorithm employing for the given cache. In Chap. 3 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[6]** 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[3]** implements the Betweenness Centrality policy. On a given path, only the node with highest betweenness centrality stores the chunk.

• **DS = fix(p)[7]** implements the Fixed probability decision. The parameter p indicates the probability with which a given node stores the incoming chunk.

• **DS = prob_cache[8]** implements the ProbCache strategy. 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 ccnSim.

• **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 least 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 following parameters describe the client behaviour (implemented in the correspondent C++ class).

• $\lambda$ is the (total) arrival rate of the Poisson process. Of course, this parameter can be different for each client.

• check_time represents a timer, expressed in seconds. Client modules check the state of every download each check_time seconds.

• RTT Represents the Round Trip Time of the network (expressed in seconds). If the time needed for downloading a file exceeds this value, the client assumes that the download is expired and does something (usually resend the interest packet for the expired download).

2.5 Statistics

The way in which statistics are taken in ccnSim 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 ccnSim 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 an 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.
### Syntax | Meaning | Values
---|---|---
num_repos | Number of the repositories | Int > 0
repos | Nodes are connected to a repository | Comma-separated string
replicas | Degree of replication for each content | Int ≤ num_repos
node_repos | Number of clients | Int > 0
repos | Active clients | Int ≤ num_clients
lambda | Poisson arrival rate | Double > 0
RTT | Round Trip Time of the Network (in seconds) | Double > 0
check_time | Timer for checking the status of a download | Double > 0
file_size | Average file size (in chunks) | Int > 1 for geometric, = 1 for objects
alpha | MZipf shaping factor | Double ≥ 0
q | MZipf plateau | Double ≥ 0
objects | Catalog cardinality | Int > 0
FS | Forwarding strategy | String
TTL1 | TTL for the NRR1 strategy | Int > 0
TTL2 | TTL for the NRR strategy | Int > 0
routing_file | Routing matrix | String
DS | Decision strategy | String
RS | Replacement strategy | String
C | Cache size (chunks) | Int > 0
window | Stabilization window (in samples) | Int > 0
ts | Sampling time (in seconds) | Double > 0
partial_n | Stabilization node-set | Int > 0 or = -1(all nodes)
steady | Steady time to simulate (in seconds) | Double > 0

**Table 2.1:** Summary of the ccnSim parameters.

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 overviewed the parameters set that affect the behaviour of each ccnSim module. We recall that these parameters can be set either within the corresponding ned module, or within the initialization file ini. In Tab. 2.1 we briefly sum up what previously shown, indicating the meaning and the value got by the given parameter.
Chapter 3

Extending ccnSim

In this chapter we go deeper within the description of ccnSim. At the end of this section the user will be able to grasp the ccnSim source code, extending and customizing the simulator for her needs. This could be seen as a more programming perspective of ccnSim. Indeed, in Chap. 2 we just described the ned part of ccnSim. Recall that in Chap. 2 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 ccnSim.

[TODO]
CHAPTER 3. EXTENDING CCNSIM
Chapter 4

Practical ccnSim

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 $\lambda = 1 \text{req/s}$. 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 mentioned in Chap. 2 abilene_network has to extend base_network, and then specifying number of nodes and connections. If we omit extending base_network the basic modules do not get initialized, and the simulations would not be executed. Finally, the package network represents the actual directory where Omnet looks for the .ned file (in this case CCNSIM_DIR/networks).

The next step is to set the correct parameters within the initialization (ini) file. We set the network name to Abilene (the network defined in Snippet 1). The parameters are set following the indication in Tab. 2.1, and the final .ned file is shown in Snippet 2. If the reader has already read Chap. 2, understanding the ini file should not be hard.

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 alpha values, let’s
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 Abilene simulation.

say from 0.5 to 1 with a 0.1 steps. One choice could be running ccnSim different
times, each time with a different value of alpha. Moreover, for each execution
one has to specify different output files, in order to not override the results.

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. Being a a ini variable, it can be used for automatically
specifying (the directory name of) the output files(output-scalar-file and
output-vector-file). In this way each simulation run, will be hold on a dif-
f erent 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 does not forcedly correspond
to the ccnSim 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 ccnSim in console mode (without running
the graphic engine). In Snippet 3 we report the result of the execution, skip-
ping out uninteresting parts. As we can see, caches are being filled after 134 secs
the simulation is initialized. Then, ccnSim 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, flush-
ing on the console 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
4.1. RUN YOUR FIRST SIMULATION

```bash
# General parameters
[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: Initialization file for the Abilene 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
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
4.2 Simulating ranges

As the last part of this brief ccnSim tutorial, we deal with simulating different decision strategies $D$ 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, (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.

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 right directory $CCNSIM_DIR/results/abilene/F-spr/D-lce/R-lru/alpha-1/$. 

![Simulation result: p_hit per node.](image)

Figure 4.1: Simulation result: $p_{hit}$ per node.
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 ccnSim on a machine more than a single processor, we would like to run a bunch of parallel simulations. 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 $\alpha$, 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 $\alpha$ 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 --line-regexp "distance [...]/ccn-id0.sca | awk '{print $4}' > distance_lcd.data
```
4.3. SUMMARY

Note that we are considering the LCD (lcd) decision strategy and that the string replacing the "[...]" should be \texttt{abilene/F-spr/D-lcd/R-lru/alpha-\ast} in order to get the right results. 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 strategy, and plotting the result, we generate Fig. 4.2.

4.3 Summary

In this section we have shown how to practically simulate a CCN network by the means of ccnSim. We simulated a simple scenario considering the Abilene network. Then we showed how to simulate a range of parameters, and producing the correct results.
Bibliography


