ccn$\mathcal{S}$im-v0.4
User Manual

April, 2017

Contents

Chapter 1

ccn$\mathcal{S}$im Overview

ccn$\mathcal{S}$im is a scalable chunk-level simulator for Content Centric Networks (CCN) [14], whose development started in the context of the Connect ANR Project. Written in C++, it is developed upon the Omnet++ framework, which provides all the APIs used to simulate key features of CCN networks, i.e., forwarding and caching strategies, cache decision policies, content request model, and so on. Thanks to its modular design and optimizations, ccn$\mathcal{S}$im allows to perform classic event-driven (ED) simulations of large-scale CCN networks, i.e., up to $M = 10^9$ contents, with moderate CPU and memory budgets. The last v0.4 version (distributed as free and open source software at `http://www.enst.fr/~drossi/ccnsim`) provides, also, a new downscaling technique based on TTL caches [22, 23], which dramatically reduces both CPU and memory footprint (more than 100× w.r.t. ED simulations), thus enabling the simulation of web-scale CCN networks (i.e., up to $M = 10^{12}$ contents).

For the list of all the new features introduced in ccn$\mathcal{S}$im-v0.4, please refer to Sec. 3 of this manual. If you enjoy working with ccn$\mathcal{S}$im-v0.4 and you need to cite it, please refer to [22, 23]; in case you use former versions of ccn$\mathcal{S}$im, instead, please refer to [10].

This manual is organized as follows:

- Chap. 2 describes how ccn$\mathcal{S}$im integrates with Omnet++, depicting, also, its general *structure*. Info about *download* and *installation* are also provided.
- Chap. 3 introduces the fundamental changes which characterize the latest v0.4 version of ccn$\mathcal{S}$im.
- Chap. 4 deeply describes the organization of ccn$\mathcal{S}$im modules, together with a brief description of the most important parameters.
- Chap. 5 reports a brief ccn$\mathcal{S}$im tutorial useful to simulate your first scenarios. Info on how to extend ccn$\mathcal{S}$im are also provided.

Chapter 2

ccn$\mathcal{S}$im: a modular CCN simulator

2.1 Integration with Omnet++

Omnet++ is a C++ based event-driven framework used in networking simulation. It is characterized by: i) a set of core C++ classes, which can be extended in order to customize the simulated environment; ii) a simple network description language (`ned`) used to describe the interactions between modules; iii) a `msg` language defining messages exchanged between network nodes.

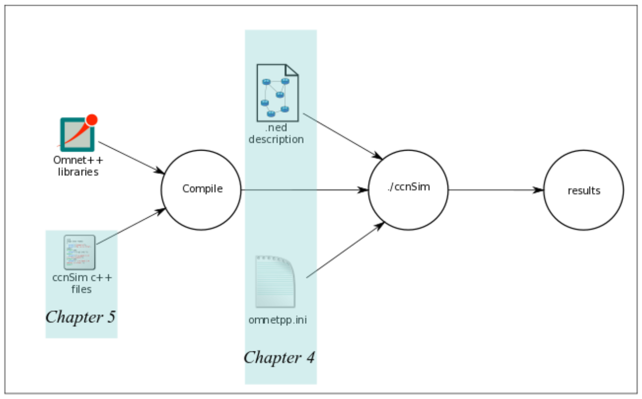
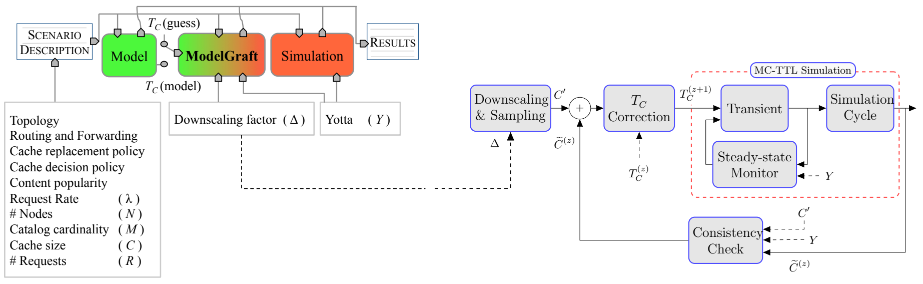


Figure 2.1: ccn$\mathcal{S}$im workflow.

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 classic workflow for ccn$\mathcal{S}$im consists in:

- Compiling ccn$\mathcal{S}$im source files, and linking them with the Omnet++ core.



(a) Seamless Integration (b) Synoptic of ModelGraft Workflow

Figure 2.2: ccn$\mathcal{S}$im-v0.4 overview.

- Writing `.ned` files which describe network topologies (only connections between the CCN nodes are usually required).
- Initializing the parameters of each module. This can be done either directly from the `.ned` files, or from the `.omnetpp.ini` initialization file.
- Launching the simulation.

We report the aforementioned steps in Fig. 2.1. 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].

2.2 Overall structure

ccn$\mathcal{S}$im-v0.4 provides users with the possibility of seamlessly selecting the most suitable simulation technique, as reported in Fig. 2.2-(a). Starting from a unique scenario description, users can analyze the performance of cache networks via either an analytical model [17] (when available, left), a classic event-driven simulation engine (right), or via the ModelGraft [22, 23] engine (middle), where MonteCarlo simulations of opportunely downscaled systems are performed by replacing LRU caches by their Che's approximated version, implemented in practice as TTL caches (for more details on ModelGraft simulations please refer to Sec. 3.1.

In order to help the reader familiarizing with the modularity of ccn$\mathcal{S}$im, its internal directory structure is provided in Snippet 1.

Each module in ccn$\mathcal{S}$im can be identified by three different subunits:

Module properties The correspondent `.ned` file specifies its appearance. For instance, the *cache* module can be described by specifying its size, replacement algorithm, and decision policy. `.ned` files are all contained within the respective `modules` directory.

Snippet 1 ccn$\mathcal{S}$im directory structure.

```
    |-- networks
    |-- modules
    |   |-- clients
    |   |-- content
    |   |-- node
    |   |   |-- cache
    |   |   |-- strategy
    |   |-- statistics
    |-- packets
    |-- Tc_Values
    |-- include
    |-- src
    |   |-- clients
    |   |-- content
    |   |-- node
    |   |   |-- cache
    |   |   |-- strategy
    |   |-- statistics
    |-- logs
    |-- results
    |-- infoSim
    |-- doc
```

Module behaviors The represent the core of ccn$\mathcal{S}$im, and they are defined as C++ classes contained within the respective `src/` and `include/` directories (sources and headers, respectively). A C++ class is defined for each module.

Packets Data and Interest packets are defined within the `packets/` directory. Messages are specified using the `msg` syntax. In order to modify packets' structure, only `msg` files should be edited. New `.cc` and `.h` files will be automatically generated in the `packets/` directory ccn$\mathcal{S}$im is compiled.

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

ccn$\mathcal{S}$im-v0.4 can be freely downloaded from `http://www.enst.fr/~drossi/ccnsim`.

Prerequisites

- `Omnet++` (version $\geq$ 4.1). It can be downloaded from `http://www.omnetpp.org`, and installed following related instructions.
- `Boost` libraries [2]. They can be installed either by using the standard packet manager of your system (e.g., apt-get install, yum install, port install, etc.), or by downloading them from `http://www.boost.org/users/download/`, and following instructions therein.

Installation In order to install ccn$\mathcal{S}$im-v0.4, it is necessary to patch Omnet++ before. ccn$\mathcal{S}$im-v4.0 comes with two different set of patches, each correspondent to the installed version of Omnet++ (e.g., v4.x or v5.x).

Please follow the instructions below, provided that Omnetpp is correctly installed under the $OMNET_DIR, and that the .tgz file of ccn$\mathcal{S}$im-v0.4 has been correctly downloaded and decompressed in the $CCNSIM_DIR folder.

According to Omnetpp version, the respective ccnSim installation commands are:

Snippet 2 Installation commands for Omnetpp-v4.x

```
pint:~$ cd $CCNSIM_DIR
pint:CCNSIM_DIR$ cp ./patch/omnet-4x/ctopology.h $OMNET_DIR/include/
pint:CCNSIM_DIR$ cp ./patch/omnet-4x/ctopology.cc $OMNET_DIR/src/sim
pint:CCNSIM_DIR$ cd $OMNET_DIR && make && cd $CCNSIM_DIR
pint:CCNSIM_DIR$ ./scripts/makemake.sh
pint:CCNSIM_DIR$ make
```

Snippet 3 Installation commands for Omnetpp-v5.x

```
pint:~$ cd $CCNSIM_DIR
pint:CCNSIM_DIR$ cp ./patch/omnet-5x/ctopology.h $OMNET_DIR/include/omnetpp
pint:CCNSIM_DIR$ cp ./patch/omnet-5x/ctopology.cc $OMNET_DIR/src/sim
pint:CCNSIM_DIR$ cd $OMNET_DIR && make && cd $CCNSIM_DIR
pint:CCNSIM_DIR$ ./scripts/makemake.sh
pint:CCNSIM_DIR$ make
```

For details about launching and performing simulations, please refer to Chap. 5.

Chapter 3

ccn$\mathcal{S}$im-0.4 Main Updates

ccn$\mathcal{S}$im-0.4 considerably improves flexibility and scalability with respect to its previous version (i.e., ccn$\mathcal{S}$im-0.3). Indeed, ccn$\mathcal{S}$im-0.4 now offers users a simulation framework through which they can select the simulation technique that mostly fits their needs. In particular, users can select between:

- Event-driven (ED) simulations: the scenario under study is simulated by means of a classic event-driven simulator, as it was for ccn$\mathcal{S}$im-0.3. Optimizations have been introduced in the v0.4 version.

- *ModelGraft* simulations: this new technique [22, 23] remarkably boosts scalability by combining Time-to-Live (TTL) caches with a MonteCarlo simulative approach. For a thorough description and comparison of *ModelGraft* against the ED version of ccn$\mathcal{S}$im-0.4, as well as for a list of motivations that pushed us develop this technique, the interested reader is referred to [23]. More details are also provided in the following.

- Analytical models: more than simple hierarchical LRU cache networks might be analyzed through a generalized analytical model [17].

Optimizations and new features included in ccn$\mathcal{S}$im-v0.4 are enumerated in the following.

3.1 ModelGraft simulations

ccnSim-0.4 pushes forward the limits of simulation scalability by implementing ModelGraft [22, 23], whose aim is to simulate large-scale networks by means of their TTL-based *downscaled* version. The original scenario is accurately downscaled by combining a MonteCarlo approach with an advanced sampling technique, and with TTL caches. As a whole, ModelGraft preserves the statistical properties of the original scenario, while remarkably cutting down on memory occupancy and CPU time.

As reported in Fig. 2.2-(a), ModelGraft formally depends on a single additional parameter, namely the *downscaling factor* Δ; in particular, if M is the original catalog cardinality, C the cache size, and R the number of simulated requests, then $M' = M/\Delta$, $C' = C/\Delta$, and $R' = R/Delta$ will be the equivalent values in the downscaled ModelGraft scenario. For guidelines on how to set Δ, please refer to [23]. Relying on the use of TTL caches as a more flexible and efficient alternative to LRU ones, each cache in the network should receive, as input, its "characteristic" of "eviction" time T_C. One option could be to bootstrap ModelGraft with informed guesses of T_C gathered via, e.g., analytical models (notice the T_C(model) switch in Fig. 2.2-(a)); nevertheless, the choice of routing/forwarding strategies, catalog dynamics, and so on, would be limited, since only cases covered by the model could be considered, thus restricting the generality of the methodology itself. A more interesting approach, used by default in ModelGraft, is instead to start from uninformed guesses of T_C (notice the default wiring to the T_C(guess) switch in Fig. 2.2-(a)), and let the system iteratively correct input values. In this case, no algorithmic restrictions are made on the system, which is simulated with a TTL-based MonteCarlo approach: this is possible since ModelGraft is intrinsically conceived as an *auto-regulating system*, so that, by design, it converges to accurate results even when the input T_C values (that users do not even need to be aware of) differ by orders of magnitude from the correct ones.

Fig. 2.2-(b) reports all the individual building blocks of ModelGraft; in a nutshell, ModelGraft starts with the configuration of the downscaling and sampling process, before entering the MonteCarlo TTL-based (MC-TTL) simulation. During the MC-TTL phase, statistics are computed after a transient period, where an adaptive steady-state monitor tracks and follows the dynamics of the simulated network in order to ensure that a steady-state regime is reached without imposing a fixed threshold (e.g., number of requests, simulation time, etc.) a priori. Once at steady-state, a downscaled number of requests is simulated within a MC-TTL cycle, at the end of which the monitored metrics are provided as input to the self-stabilization block: a consistency check decides whether to end the simulation, or to go through a T_C correction phase and start a new simulation cycle.

As for the applicability of ModelGraft, we summarize in Tab. 1 those features/algorithms implemented in ccnSim, which are either supported by ModelGraft or available only with the event-driven engine. Intuitively, as long as the original system admits a mean-field approximation 'a la Che, the dynamics of its contents can be decoupled using the characteristic time T_C, which can be in turn used as input for TTL caches. While the original Che's approximation holds only for LRU caches, the class of cache networks that can be modeled by its extensions is fairly large. For a thorough description please refer to [23].

Table 3.1: Scope of *ModelGraft* applicability

	Supported (tested)	*Supported (untested)*	Unsupported
Workload	IRM, Dynamic	-	-
Forwarding	SP, NRR[21], LoadBalance[20]	-	-
Cache decision	LCE, LCP[6], 2-LRU[17]	CoA[5], LAC[7]	LCD/MCD[15]
Cache replacement	LRU[9], FIFO[17], RND[11]	-	LFU

3.2 Rejection Inversion Sampling

This method, originally proposed in [13], and extended in [3], allows to produce random samples following a Zipf's distribution, without the need to allocate memory to store the Cumulative Distribution Function (CDF) vector. As a consequence, ccn$\mathcal{S}$im-0.4 dramatically reduces its memory requirements, not only when using the ModelGraft approach, but also for classic ED simulations. For a description of the technique, readers are referred to [23].

3.3 Optimized Request Generation for ModelGraft Simulations

One of the distinctive features of ModelGraft is that it preserves the statistical properties of the original non-downscaled scenario. In order to achieve this, an ad-hoc request generation technique has been designed. In particular, ccn$\mathcal{S}$im-v0.4 introduces a new request generation process for ModelGraft simulations, referred as *request sharding* which steps out of the previous one, based on a preemptive uniform catalog binning [22, 23], and develops a technique which is entirely based on the Inversion Rejection Sampling. In particular, at each request, a content is extracted from the original non-downscaled catalog, and the correspondent meta-content of the downscaled catalog (which will be effectively requested) is computed a posteriori.

In order to visually compare the performance of ModelGraft in the simulation of a large-scale scenario, both with the previous and the new request generation technique, against the classic ED approach of ccn$\mathcal{S}$im-v0.4, a sample table is reported in Tab. 3.2. In this case, ModelGraft is tested for three different cache decision polices, while using exact T_C values as input. Both accuracy loss, CPU and memory gains are reported.

As it can be seen from Tab. 3.2, the new request generation technique allows to further reduce the CPU execution time of the entire simulation, and the accuracy loss, at the same time, with respect to the former version of ModelGraft. At the present state, the accuracy loss is kept under 0.6%, while both CPU and memory gains are well above 100×.

Table 3.2: ED simulation vs Old ModelGraft vs New ModelGraft with request sharding, (4-level binary tree, $M = R = 10^9, C = 10^6, \Delta = 10^5, Y = 0.75$)

Cache Decision Policy	Technique	p_{hit}	Loss	CPU time	Gain	Mem [MB]	Gain
LCE	ModelGraft	**31.4%**	**1.8%**	**211s**	**194x**	**38**	**168x**
	Event-driven	33.2%		11.4h		6371	
	ModelGraft w **ReqSha**	**33.1%**	**0.1%**	**143s**	**286x**	**23**	**277x**
LCP(1/10)	ModelGraft	**34.0%**	**1.4%**	**291**	**90x**	**38**	**168x**
	Event-driven	35.4%		7.3h		6404	
	ModelGraft w **ReqSha**	**35.3%**	**0.1%**	**171s**	**153x**	**23**	**277x**
2-LRU	ModelGraft	**36.1%**	**0.9%**	**402s**	**97x**	**38**	**234x**
	Event-driven	37.0%		10.8h		8894	
	ModelGraft w **ReqSha**	**37.6%**	**0.6%**	**180s**	**216x**	**24**	**370x**

3.4 Optimized Model Solver

ccnSim-v0.4 comes with an out-of-the-box model solver based on [17]. Complex networks can be efficiently analyzed through this analytical model. So far, supported replacement strategies (RS), cache decision policies (or meta-caching - MC), and forwarding strategies (FS) are:

- RS: LRU;
- MC: LCE, FIXp;
- FS: SP, NRR.

3.5 Shot Noise Request Model

ccn$\mathcal{S}$im-v0.4 provides users with the possibility of choosing between two different kinds of content requests models:

- Independent Request Model (IRM): it is the classic and most used approach in simulation studies. i.i.d Content requests are generated for a catalog of M contents with fixed popularities following a Poisson process of mean rate equal to Λ.

- Shot Noise Model (SNM): it reproduces the model proposed in [18], where the temporal locality of requests is taken into account through an ON-OFF process that models the request patterns of different classes of contents. The SNM allows to simulate scenarios where content popularities changes and evolves over time.

In the SNM, dynamically evolving content popularity is modeled by means of an ON-OFF process: contents are divided into classes characterized by a series of parameters. A sample scenario with characteristic parameters is reported in Tab. 3.3, which compares both original parameters taken from [18], and the downscaled ones used for the experiments reported in the following. First of all, a lifetime (T_{on}) is associated to each class, indicating the time during which the respective contents are active and can be fetched. At the same time, each class has its own period of inactivity (T_{off}), during which the respective contents cannot be retrieved. Each class is, then, characterized by an average total request rate ($E[V]$), which expresses the total number of requests per seconds generated for that class during the ON period. The group of contents inside each class is also characterized by a popularity distribution that influences the generation of content requests.

Using the sample scenario reported in Tab. 3.3, we show in the following how the Shot Noise scenario compares with the stationary Independent Request Model. We use a grid network as a reference topology, and for the IRM case we consider a catalog whose cardinality is equal to the sum of the six classes of the SNM, in order to guarantee the maximum fairness in the comparison. Furthermore, we simulate 10^9 content requests, and we use a Zipf's probability distribution with exponent $\alpha = 1$ to generate content requests. Results are reported in Fig. 3.1, which clearly confirms that IRM models which do not consider the temporal locality of content requests are more pessimistic (i.e., smaller hit ratio, higher load and hit distance, etc.). Fig. 3.2 reports, also, per each class, the percentage of "scheduled" against "validated" requests, where "validated" refers to those requests that have been scheduled and forwarded during the ON periods of the respective class (scheduled requests during the OFF periods are not forwarded).

It is worth noticing that the SNM implementation available in ccn$\mathcal{S}$im-0.4 is not at its final stage of development yet (i.e., it has not been thoroughly tested being not the main focus of the simulator so far). However, starting from a provided simple scenario, users are encouraged to experiment and improve the implementation.

3.6 2-LRU cache decision policy

ccn$\mathcal{S}$im-v0.4 comes with an implementation of a new cache decision policy proposed in [17], namely 2-LRU. Practically speaking, an additional LRU cache

Class	Trace	T_{on} ([s]-[days])	$E[V]$	Alpha	# Contents	Brk Down [%]
1	Our	43200-0.5	41.66	1	$6.25 \cdot 10^4$	71.31
1	[12]	172800-2	83.33	0.7	$2.5 \cdot 10^6$	68.89
2	Our	151200-1.75	37.5	1	$7.5 \cdot 10^4$	22.01
2	[12]	604800-7	75.00	0.7	$3 \cdot 10^6$	21.26
3	Our	648000-7.5	33.33	1	$7.5 \cdot 10^4$	4.56
3	[12]	2592000-30	66.66	0.7	$3 \cdot 10^6$	4.41
4	Our	2160000-25	25	1	$8.75 \cdot 10^5$	1.20
4	[12]	8640000-100	50	0.7	$3.5 \cdot 10^6$	1.16
5	Our	21600000-250	25	1	$3.75 \cdot 10^5$	0.51
5	[12]	86400000-1000	50	0.7	$15 \cdot 10^6$	0.04
6	Our	21600000-250	0.8	1	$93.25 \cdot 10^5$	0.41
0	[12]	86400000-1000	1.6	0.7	$4 \cdot 10^9$	4.23

Table 3.3: Settings for the Shot Noise Model scenario.

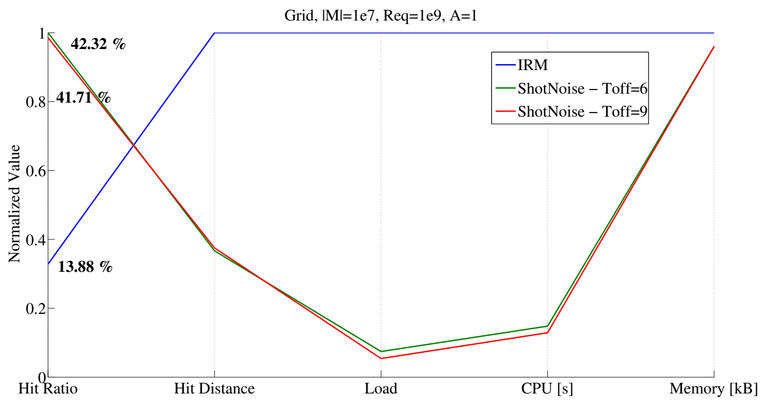


Figure 3.1: SNM vs IRM.

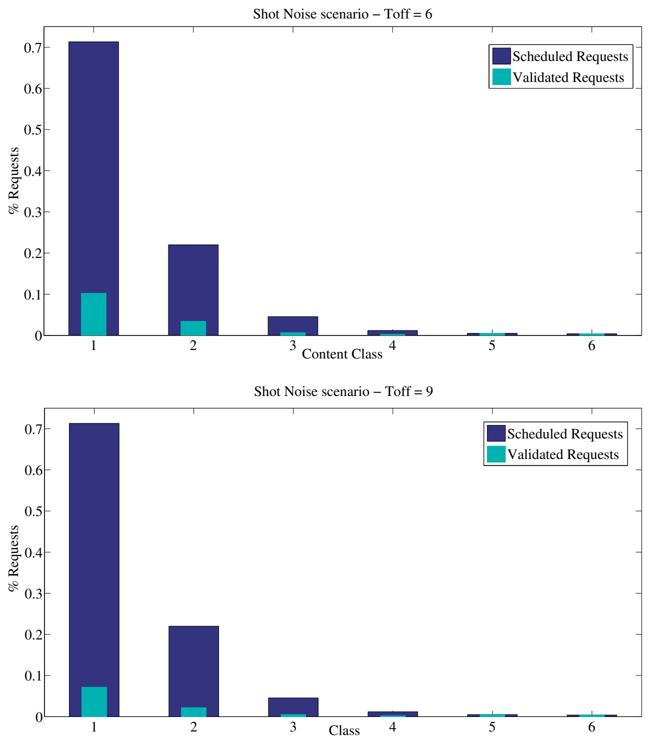


Figure 3.2: Percentage of *scheduled* and *validated* requests per each class - (a) $T_{off} = 6\ T_{on}$, (b) $T_{off} = 9\ T_{on}$.

(known as Name Cache) where names of requested contents are stored is put in front of the main one. When a request is received, a first lookup is performed on the Name Cache: in case of a positive outcome, the correspondent fetched content will be later stored inside the main cache, otherwise only the name of the requested content will be inserted inside the Name Cache (without effectively caching the fetched content).

The aim of the 2-LRU cache decision policy is to reduce cache pollution generated by unpopular contents (i.e., contents that are requested very rarely) in order to increase the hit ratio. Owing to its design, the 2-LRU policy provides, also, good performance in the presence of temporal locality, meaning that it can locally adapt to short-term popularity variations of content popularity caused by temporal locality.

Chapter 4

The ccn$\mathcal{S}$im simulator

In the following we provide a more technical overview of the modules that constitute ccn$\mathcal{S}$im-v0.4, describing their main parameters. Note that each parameter can be set either via the `.ini` file associated to the simulation (see Chapter 5), or by directly modifying the `.ned` description file associated to each module[1]. All the presented modules have a corresponding C++ class which defines their behavior.

4.1 Topology definition

The highest level of ccn$\mathcal{S}$im simulations is the `network` module, where users can define useful information about the simulated topology, like (i) total number of CCN `node`s modules within the network, (ii) their connectivity, as well as (iii) the placement of clients and repositories. Each newly created network module MUST extend the `base_network` one, where other modules related to other aspects of the simulation (i.e., clients, statistics, and so forth) are also defined.

Placement of Clients: in ccn$\mathcal{S}$im-v0.4 *clients* represent aggregate of users, attached to each node, which can be active or not. Indeed, since clients are connected to each node of the network (see the `networks/base.ned` file where port 0 of each node is connected to a client), the placement consists in specifying how many nodes will present *active* clients. The process of activating a client takes place in the `client::initialize()` method. The basic parameters for their placement are defined in `networks/base.ned`:

- `num_clients`: this integer value specifies how many clients are active over the whole network.
- `node_clients`: comma separated string which specifies CCN nodes with an active connected client. The number of nodes enumerated in `node_clients` should be $\leq$ `num_clients`. If it is strictly smaller than `num_clients` (this includes also the case of an empty string `‘ ’`), the remaining clients will be randomly distributed across the network.

It is possible to understand which node the i-th active client is attached to by searching for `client-`i in the output `.sca` file (see Sec. 4.5.1).

Placement of Repositories: in ccn$\mathcal{S}$im-v0.4 there is no real node acting as a repository, while selected node are just aware that they are directly connected to a repository. The distribution of content repositories in the whole network takes place in `content_distribution::init_repos()` method. It basically depends on two parameters, defined in `networks/base.ned`:

- `num_repos`: integer value that specifies how many repositories should be distributed over the network.
- `nodes_repos`: comma separated string which specifies CCN nodes directly connected to a repository. The number of enumerated repositories in `nodes_repos` should be at most `num_repos`. If it is $\leq$ `num_repos` (this includes the case of an empty string `‘ ’`), the remaining repositories will be randomly distributed across the network.

It is possible to understand which node the i-th repository is attached to by searching for `repo-`i in the output `.sca` file (see Sec. 4.5.1).

4.2 Content handling

The `content_distribution` module takes no part in defining the network topology, but it accomplishes many crucial tasks for the correct functioning of the simulation.

Catalog initialization: The *catalog* is defined as a collection of *contents* (or *objects*), and it is characterized by the following parameters:

- `objects`: it represents the *cardinality* of the catalog, expressed in number of contents.
- `file_size`: it represents the average number of *chunks* a content is made of (content size in terms of number of chunks is geometrically distributed). If $file\_size = 1$, there is a 1:1 correspondence between contents and chunks.
- `replicas`: it indicates the degree of replication of each content (i.e., how many seed copies of the same content are available in the whole network). Since each repository cannot store twice the same content, `replicas` has to be $<$ `num_repos`. According to this parameter, each content will be (randomly) replicated over exactly `replicas` repositories. If $replicas = 1$, seed copies of all the contents will be equally split among the `num_repos` repositories.
- `downsize`: it represents the downscaling factor Δ used in ModelGraft simulations [22, 23]. It is defined in the `statistics.ned` file, and its optimal and maximum value is equal to $\Delta = C/10$, where C is the cache size (for a thorough explanation and evaluation of the limits of Δ please refer to [23]).

In ccn$\mathcal{S}$im-v0.4 the content catalog, which is initialized in the `initialize()` method of the `content_distribution` class, is specifically represented by a static vector of bit strings, namely `content_distribution::catalog`. Each element of the vector encodes all the information related to the respective object, like *object id*, the number of chunks composing it, the repositories where it is stored, etc. These information are accessed through bit masks coded as macros (defined in `ccnsim.h`): `__info,__size,__repo`.

Content popularity: for the time being, the only popularity distribution implemented in ccn$\mathcal{S}$im-v0.4 is the Mandelbrodt-Zipf. As previously said, ccn$\mathcal{S}$im-v0.4 introduces a remarkable improvement: the generation of requests following a Zipf's distribution is implemented by means of a sampling technique known as Inversion Rejection Sampling [13]. In particular, Zipf's distributed random numbers are generated without allocating memory space to store the Cumulative Distribution Function (CDF) vector, which for large scenarios, represents the main cause for memory demand. This technique is currently implemented for pure Zipf's distributions (i.e., $q = 0$) with $\alpha > 0$. Key parameters are passed to the `content_distribution` module (corresponding to `content_distribution` class), like number of contents (`objects`), and Zipf's exponent α.

4.3 Node Module

The `node` module represents the fundamental unit of ccn$\mathcal{S}$im. It is compounded by three main submodules: `core_layer`, `strategy_layer`, and `content_store`. Together they define the general behavior of a CCN node in terms of *forwarding* and *caching*; this behavior can be shortly identified by the triple $\langle \mathcal{F}, \mathcal{D}, \mathcal{R} \rangle$, where $\mathcal{F}$ indicates the *forwarding strategy*, $\mathcal{D}$ the cache *decision* policy, and $\mathcal{R}$ the cache *replacement* policy. Each of the aforementioned submodules are described in the following, jointly with their correspondent parameters.

4.3.1 Core Layer

The `core_layer` module implements the basic tasks of a CCN node, along with the communication with the other submodules. In particular, it is in charge of *Interest* and *Data* handling: in case of a cache hit (within the node's `content_store`), it generates the reply message and it sends back the requested Data towards the interfaces associated with the incoming Interest found in the PIT; otherwise, it adds/creates an entry into the PIT reporting the incoming interface, and, in case of a newly created entry, it queries the `strategy_layer` in order to fetch the correct output interface(s) to forward the Interest to.

4.3.2 Strategy Layer

The `strategy_layer` is responsible for the forwarding of Interest packets. When an Interest for which no PIT entry exists is received, an output face needs to be determined. In ccn$\mathcal{S}$im-v0.4, each node can retrieve the information on where the respective seed copy is permanently stored by using the macro `__repo`; the routing table of each node is populated at run-time by using the Dijkstra algorithm. Routing matrices can also by passed as ASCII files, specified by the `file_routing` parameter of the `strategy_layer` module.

By setting the `FS` parameter inside the `node` module, Interest packets can be forwarded by using different forwarding strategies $\mathcal{F}$:

Shortest Path Routing [`FS = spr`] Interest packets are sent towards the nearest repository in terms of hops.

Random Repository [`FS = random_repository`] The strategy layer choses one repository at random out of the given set. Note that this strategy requires core nodes following the path chosen by the client node.

Nearest Replica Routing (Two phases) [`FS = nrr`] With this setting, the strategy layer first explores the neighboring nodes by flooding meta-Interest (i.e., Interest packets which do not change the content of the cache) with a given TTL, in order to locate the closest copy of the requested content (both permanent and temporarily cached ones). Once terminated this step, the actual Interest is sent out towards the nearest node having the content available. The TTL can be set by means of the `TTL1` parameter. Setting `TTL1` to ∞ (i.e., greater than the network diameter) makes NRR degenerating in the ideal NRR (iNRR) strategy, which explores the entire network, looking for a copy of the given content.

Nearest Replica Routing (One phase) [`FS = nrr1`] In this case, a preemptive exploration phase, in which the neighborhood is flooded with the request for the given object, is needed. When the copy is found, the actual Data is fetched back (it can be a permanent copy, or a *cached copy* as well). The scope of the flooding can be set by means of the `TTL2` parameter.

4.3.3 Content Store

The `content_store` module reproduces all the dynamics of a caching system, from its replacement to its decision strategy. In particular, the decision policy $\mathcal{D}$ returns a boolean value indicating if the current node (on the path between the client and the target node having the content) has to cache or not the fetched Data, while the replacement strategy $\mathcal{R}$ is in charge of selecting an element to be dropped from a full cache, in order to make room for the incoming content.

Decision Strategies - $\mathcal{D}$

The decision strategy (or meta-caching algorithm) is used by each node in order to decide if the incoming Data should be stored or not in the local cache. The decision strategy can be selected by setting the `DS` parameter in the `node` module.

- `DS = lce` selects the Leave Copy Everywhere policy, meaning that each incoming contents is always stored within the cache. The correspondent definition can be found in the `include/always_policy.h` file.
- `DS = lcd`[15] selects the Leave Copy Down policy, meaning that the incoming content is stored in the local cache only if it has been originally fetched from a node one hop away from the current node. It is defined in the `include/lcd_policy.h` file.
- `DS = fixP`[16] selects the fixed-probabilistic decision policy. In particular, the parameter `P` indicates the *probability* with which a given node stores incoming chunks (e.g., `DS = fix0.1`).
  It is defined in the `include/fix_policy.h` file.
- `DS = btw`[8] selects the Betweenness Centrality policy. On a given path, only the node with highest betweenness centrality stores the chunk. It is defined in the `include/betweeness_centrality.h` file.
- `DS = two_lru`[17] selects the 2-LRU strategy described in Sec. 3.6. It is defined in the `include/two_lru_policy.h` file.
- `DS = two_ttl`[17] selects the 2-LRU strategy described in Sec. 3.6, but for TTL simulations with TTL caches.
- `DS = prob_cache`[19] selects the ProbCache strategy, where the probability of caching a given content is inversely proportional to the distance of the current node to the original node the content has been fetched from. It is defined in the `include/prob_cache_policy.h` file.
- `DS = never` disables caching within the network.
  It is defined in `include/never_policy.h`.
- `DS = costawareP`[4] selects the Cost-Aware (CoA) strategy. An incoming object is accepted with a probability proportional to its price. `P` is the average cache admission probability weighted on the different prices. It is defined in the `include/cost_related_decision_policies/costaware_policy.h` file. See `cost_scenario.tar.gz` to start experimenting with it.
- `DS = ideal_blind`[4] selects the Ideal-Blind strategy. An incoming object is accepted only if its rank is greater than or equal to the rank of the eviction candidate, i.e., the least recently used object stored in the cache. It is defined in the
  `include/cost_related_decision_policies/ideal_blind_policy.h` file.
  See `cost_scenario.tar.gz` to start experimenting with it.
- `DS = ideal_costaware`[4] selects the Ideal-CoA strategy. An incoming object is accepted only if its price is greater than or equal to the one of the eviction candidate, i.e., the least recently used object stored in the cache. It is defined in the

`include/cost_related_decision_policies/ideal_costaware_policy.h` file. See `cost_scenario.tar.gz` to start experimenting with it.

To have an idea on how the decision policy is set up, the reader is referred to the `base_cache::initialize()` method in `src/node/cache/base_cache.cc`.

Replacement strategies - $\mathcal{R}$

In the end, a caching system is characterized by its replacement strategy. In ccn$\mathcal{S}$im-v0.4 it can be selected by setting the `RS` parameter of the `node` compound module.

- `RS = lru_cache` selects a LRU replacement strategy, where the least recently used item stored within the cache is replaced by the fetched content.
- `RS = ttl_cache` selects a TTL replacement strategy, where contents are evicted at the expiration of a timer, unless new requests have arrived in the meantime (in that case the timer is reset).
- `RS = lfu_cache` selects a LFU replacement strategy by means of counters (the less requested content is deleted when the cache is full).
- `RS = random_cache` selects a random replacement policy.
- `RS = fifo_cache` selects a basic First In First Out replacement logic.

4.4 Clients

As previously said, a `client` represents an aggregate of users. In ccn$\mathcal{S}$im-v0.4, users can select three different types of clients, according to the simulation they intend to carry out: (i) `client_IRM` is that standard client implementation which requests contents one by one (i.e., supposing a request window $W = 1$), following an IRM model; (ii) `client_ShotNoise`, which is selected in case the SNM is used for the simulation; (iii) `client_Window`, which is able to generate batches of Interest packets, i.e., $W > 1$ (not fully tested).

In general, the behavior of a client is described by the following parameters:

- `lambda`, which is the (total) arrival rate of the Poisson process per each client.
- `check_time`, which is a timer, expressed in simulation seconds, after which Clients check the state of every download.
- `RTT`, which 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 retransmits the correspondent Interest.

4.5 Collection of Statistics

The key point of a simulation campaign is that of gathering scientifically sound statistics and results. The dimensioning of the *warm-up* phase plays a paramount role when our aim is to gather statistics which are actually computed at steady-state; in real cases, the length of the transient can be affected by several parameters, like forwarding strategy (e.g., shorter paths under ideal NRR can reduce the transient with respect to shortest path [21]), or cache decision policy (e.g., Leave Copy Probabilistically (LCP) [6], with a reduced content acceptance ratio with respect to Leave Copy Everywhere (LCE), is expected to yield longer transient durations).

ccn$\mathcal{S}$imv0.4 is characterized by an intelligent transient monitoring process which automatically adapts the duration of the warm-up phase by monitoring relevant metrics. In particular, the convergence of a single node i is effectively monitored using the Coefficient of Variation (CV) of the *measured hit ratio*, $\bar{p}_{hit}(i)$, computed via a *batch means* approach.

The i-th node is considered to enter a steady-state regime when:

$$CV_i = \frac{\sqrt{\frac{1}{W-1}\sum_{j=1}^{W}\left(p_{hit}(j,i) - \bar{p}_{hit}(i)\right)^2}}{\frac{1}{W}\sum_{j=1}^{W}\bar{p}_{hit}(j,i)} \leq \varepsilon_{CV}, \tag{4.1}$$

where,

- W is the size of the sample `window`
- ε_{CV} is a user-defined convergence threshold
- $p_{hit}(j,i)$ is the j-th sample of the measured hit ratio

The sampling is governed by the joint combination of W and ts parameters, where ts is the sampling time expressed in seconds. As an example, suppose that $W = 60$s and $ts = 0.1$s; this means that each 100ms, a sample *should* be collected, and when 60s of samples are collected (i.e., 600 samples), the CV is computed and compared against the ε_{CV} threshold. To avoid biases, new samples are effectively collected only if (i) the node has received a non-null number of requests since the last sample, and (ii) its state has changed, i.e., at least a new content has been admitted in the cache since the last sample. To exemplify why this is important, consider that with a LCP(p) cache decision policy, where new contents are probabilistically admitted in the cache, the reception of a request is correlated with the subsequent caching of the fetched content only in 1 out of $1/p$ cases.

The convergence of the whole network is also governed by a tunable parameter $Y \in (0, 1]$; if we denote with N the total number of nodes, we consider the whole system entering a steady-state when:

$$CV_i \leq \varepsilon_{CV}, \qquad \forall i \in \mathcal{Y}, \tag{4.2}$$

where $|\mathcal{Y}| = \lceil YN \rceil$ is the *set of the first* YN *nodes satisfying condition* (4.1). The rationale behind this choice is to avoid to unnecessarily slow down the convergence of the whole network by requiring condition (4.1) to be satisfied by all nodes: indeed, in cases where particular routing protocols and/or topologies are simulated, same nodes might be associated with very low traffic loads (hence, long convergence time), and, at the same time, with a marginal weight in the computation of network KPIs. The Y parameter can be configured by setting the `partial_n` variable in the main `.ini` file. Once at steady-state, the simulation will run for `steady` seconds, where `steady` (defined and configurable in the `statistics.ned` file) expresses the simulation time after stabilization.

In general, KPIs which are gather by default are:

- `p_hit`: it represents the average *hit probability*, computed as $p\_hit = \frac{\#hit}{\#miss+\#hit}$.
- `hdistance`: it reports the average number of hops needed to retrieve a Data packet.
- `elapsed`: it is 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 packets, respectively, handled during the simulation.

4.5.1 Output files

The usual way to collect results is that of resorting to standard Omnet++ files, like `.sca` files for coarse-grain and per node statistics, or `.vec` file for fine-grain and per content statistics. Usually, `.vec` files may grow up to GBytes for long simulations; as a consequence, they are disabled by default in ccn$\mathcal{S}$im-v0.4. If it is necessary to use them, the following line should be added to the main `.ini` file:

```
    **.vector-recording = true
```

For more information about the use of `.sca` and `.vec` files, please refer to the Omnet++ manual.

4.5.2 Output in ccn$\mathcal{S}$im-v0.4

ccn$\mathcal{S}$im-v0.4 offers the possibility to run simulations both in the classic way, and by means of a sample script which is freely provided. In the former case, simulations should be launched with the following command:

```
    pint:CCNSIM_DIR$ ./ccnSim -u Cmdenv -f ED_TTL-omnetpp.ini r 0
```

where `Cmdenv` indicates that the simulation will be run without the graphical interface, `r 0` tells that only a single run will be executed, and `ED_TTL-omnetpp.ini` indicates the default .ini file, which allows to simulate a simple scenario for the sake of illustration. Since all the parameters inside the `ED_TTL-omnetpp.ini` file are documented, it can be used as a starting point to simulate more complex scenarios. Outputs are produced inside the `results/` folder, in the form of `.sca` files.

In the latter case, instead, simulations can be launched by means of a sample script through the following command:

```
    pint:CCNSIM_DIR$ ./runsim_script_ED_TTL.sh {parameters}
```

This script automatically creates a new .ini file, starting from the default one, where {`parameters`} values passed from the command line are associated to the correspondent variables. For a complete list of all the possible {`parameters`} that can be passed from the command line please refer to Chap. 5 of the manual.

This script allows also to automatically collect the aforementioned KPIs; indeed, at the end of the simulation, outputs are produced under `logs/`, `results/`, and `infoSim/` folders. In particular, a file containing a summary of all the collected metrics is produced inside the `infoSim/` folder with the name `ALL_MEASURES_`{`scenario_description`}, where `scenario_description` reflects the set of key/value substrings that identify the simulated scenario. The temporal evolution of the simulated scenario, along with some aggregated metrics, is logged inside the `scenario_description` file in the `logs/` folder.

4.6 Debug

ccn$\mathcal{S}$im source code contains some blocks of code as:

```
    #ifdef SEVERE_DEBUG
    ... code ...
    #endif
```

The nested lines of code, disabled by default, have only debug purposes. It is highly advisable to enable the `SEVERE_DEBUG` mode when you modify ccn$\mathcal{S}$im source code, in order to check if you are introducing inconsistencies or if you are experiencing strange errors which are easier to debug in the `SEVERE_DEBUG` mode. For more details, search for `SEVERE_DEBUG` in `include/ccnsim.h`. You can also add your own debug lines.

4.7 Summary

In this chapter we overviewed the set of parameters that affect the behavior of the main ccn$\mathcal{S}$im modules. In Tab. 4.1, we briefly sum up what previously

Syntax	Meaning	Values
`num_repos`	Number of repositories	Int > 0
`node_repos`	Nodes a repository is attached to	Comma-separated string
`replicas`	Number of seed copies for each content	Int $<=$ `num_repos`
`num_clients`	Number of active clients	Int > 0
`node_clients`	Nodes which have an active attached	Int $<=$ `num_clients`
`lambda`	Poisson arrival rate	Double > 0
`RTT`	Timeout to indicate a download failure (in seconds)	Double > 0
`check_time`	Timer indicating how often a download should be checked	Double > 0
`client_type`	Type of simulated client	String(e.g., IRM)
`file_size`	Average file size (in chunks)	Int ≥ 1
`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 when fixed routing is used	String
`DS`	Cache Decision Strategy	String
`RS`	Cache Replacement Strategy	String
`C`	Cache size (chunks)	Int > 0
`NC`	Name Cache size for `two_lru` sim	Int > 0
`tc_file`	File containing T_C values for each node (only for TTL sim)	String
`tc_name_file`	File containing T_C values for name cache (only for TTL sim)	String
`window`	Sample window (in seconds)	Int > 0
`ts`	Sampling time (in seconds)	Double > 0
`partial_n`	Percentage of nodes considered for the convergence check	Double $\in]0, 1]$
`steady`	Simulation time after stabilization (in seconds)	Double > 0
`start_mode`	Cold vs Hot start	String
`fill_mode`	Used in case a Hot start mode is selected	String
`onlyModel`	Flag indicating what happens after Model Solver	Boolean
`downsize`	Downscaling factor used with TTL simulations	Int $\in [1, C/10]$

Table 4.1: Summary of the main parameters.

shown, indicating the meaning and the value associated to the given parameter. We recall that these parameters can be set either within the corresponding `.ned` module, or within the initialization file `.ini`, or by using the provided `.sh` scripts as command line parameters.

Chapter 5

Practical ccn$\mathcal{S}$im

5.1 Run your first simulation

As previously stated in Sec. 4.5.2, ccn$\mathcal{S}$im-v0.4 offers the possibility to run simulations both in the classical way, and by means of a sample script which is freely provided.

Classic method

ccn$\mathcal{S}$im-v0.4 still keeps the classic way simulations are executed:

```
    pint:CCNSIM_DIR$ ./ccnSim -u Cmdenv -f ED_TTL-omnetpp.ini r 0
```

The option `-u Cmdenv` is used to execute ccn$\mathcal{S}$im in console mode, i.e., without running the graphic engine. Sometimes, it might be useful to launch ccn$\mathcal{S}$im using the Omnet++ graphical interface (e.g., for preliminary checks). To do so, the following command should be used:

```
    pint:CCNSIM_DIR$ ./ccnSim -f ED_TTL-omnetpp.ini r 0
```

Please refer to Omnet++ manual for more information about the graphical interface.

With the default `ED_TTL-omnetpp.ini` file, a very simple scenario is simulated, and results are produced inside `logs/` and `results/` folders.

By directly modifying the `.ini` file, users can execute different simulations at once by specifying the correspondent value `ranges` for the specific parameters. For example, suppose that we want to simulate different cache decision policies, $\mathcal{D}$, for different values of α. In order to execute this set of simulation at once, we can exploit the `$` notation in the `.ini` file:

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

We will end up with a total of 24 simulations to be executed.

If we are running ccn$\mathcal{S}$im on a machine with more than a single processor, we might want to run 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.

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

New method

With ccn$\mathcal{S}$im-v0.4 simulations can be launched also by means of a sample script through the following command:

```
pint:CCNSIM_DIR$ ./runsim_script_ED_TTL.sh {parameters}
```

This script automatically creates a new .ini file, starting from the default one, where {`parameters`} values passed from the command line are associated to the correspondent variables. For a complete list of all the possible {`parameters`} that can be passed from the command line please refer to Tab. 5.1 of the manual.

This script allows also to automatically collect the main KPIs; indeed, at the end of the simulation, outputs will be produced under the `logs/`, `results/`, and `infoSim/` folders. In particular, a file containing a summary of all the collected metrics is produced inside the `infoSim/` folder with the name ALL_MEASURES_{`scenario_description`}, where `scenario_description` reflects the set of key/value substrings that identify the simulated scenario. The temporal evolution of the simulated scenario, along with some aggregated metrics, is logged inside the `scenario_description` file in the `logs/` folder.

In the following section we will go through a practical tutorial on how to use the aforementioned script in order to run a complete simulation.

5.2 Small scenario

In this chapter, we will go through all the steps needed to simulate a simple scenario:

- 4-level binary tree topology (i.e., 15 nodes)

Parameter #	**Parameter**	**Description**	**Values**
1	T	Simulated Network	String
2	#Clients	Number of Clients	$[1, N]$
3	#Repos	Number of Repositories	$[1, N]$
4	FS	Forwarding Strategy	String
5	MC	Cache decision policy (or Meta-caching)	String
6	RS	Replacement strategy	String
7	Alpha	Zipf's Exponent	$]0, \infty]$
8	C	Cache size	$[0, M]$
9	NC	Name Cache size (valid only for 2-LRU)	$[0, M]$
10	M	Catalog cardinality	$[1, 1e12]$
11	R	Total number of simulated requests	$[1, \infty]$
12	Lambda	Aggregated request rate for each client	$[0.1, \infty]$
13	ClientType	Client Type (i.e., IRM)	String
14	CDType	Content Distribution Type (i.e., IRM)	String
15	Toff	Off Time (only for Shot Noise Model)	String
16	n	#runs to be simulated (#runs=n+1)	$[0, \infty]$
17	StartType	Initialization Type (i.e., Cold vs Hot start)	String
18	FillType	How caches are filled in case of Hot Start	String
19	Y	`partial_n` parameter	$[0.1, 1]$
20	Δ	Downscaling factor (TTL simulations)	$[1, C/10]$
21	TcFile	File of T_C values for all nodes	String
22	TcNameFile	File of T_C values for NC of all nodes	String

Table 5.1: Parameters passed to the ./runsim_script_ED_TTL.sh script.

- Clients are connected at each leaf node
- Aggregate request rate for each client is $\lambda = 20 req/s$
- Catalog cardinality $M = 10^7$. All the seed copies are supposed to be stored in a single repository connected to the root node
- Catalog popularity following a Zipf-like distribution with $\alpha = 1$
- Cache size $C = 10^4$, equal for each node
- Total number of simulated requests $R = 10^7$
- LRU cache replacement policy
- LCE cache decision policy

Once the scenario is defined, the first step is that of passing the right `parameters` to the `./runsim_script_ED_TTL.sh` script. In particular, the meaning of each parameter, following the order of insertion, is reported in Tab. 5.1.

For an extended description of each parameter, please refer to the `ED_TTL-omnetpp.ini` file. It is worth noticing that:

- with ModelGraft (i.e., with TTL downscaled simulations described in Sec. 3.1, and available in ccn$\mathcal{S}$im v-0.4), scenarios with up to $M = 1e12$ contents can be simulated (provided the availability of an adequate CPU time budget since $R = 1e12$ requests should be considered in order to obtain results which are statistically relevant), while for classic ED simulation the natural limit is set to $M = 1e9$.

- $\Delta > 1$ makes sense only for ModelGraft simulations (i.e., RS=*ttl*). It is worth noticing that Δ cannot be indefinitely large; indeed, there is a rule of thumb to be followed in order to avoid *instability* of the downscaled TTL-based simulation. In particular, the maximum downscaling factor is related to the cache size C: the system is stable for a large plateau of Δ values up to $\Delta^* = C/10$, where largest gains are expected; for $\Delta < \Delta^*$ the system is still stable but gain reduces, whereas for $\Delta > \Delta^*$ ModelGraft becomes unstable.
- For ModelGraft, if no TcFile is provided, the script tries to automatically search for the correspondent one inside the `Tc_Values` folder. If also this trial fails, ccn$\mathcal{S}$im-v0.4 tries to randomly generate Tc values at runtime per each node. This allows ModelGraft to perform any simulation thanks to its feedback loop which iteratively corrects input Tc values. It is, however, a very general heuristic which does not guarantee optimal performance.

In order to simulate the scenario considered herein with the classic *event-driven* approach (i.e., RS=*lru*), the correspondent command would be:

```
pint:CCNSIM_DIR$ ./runsim_script_ED_TTL.sh tree 8 1 spr lce
 lru 1 1e4 1e4 1e7 1e7 20 IRM IRM 1 0 cold naive 0.75 1
```

Notice that for the classic ED-sim, RS=*lru* and $\Delta = 1$.

To simulate, instead, the same scenario with the ModelGraft approach, the correspondent command would be:

```
pint:CCNSIM_DIR$ ./runsim_script_ED_TTL.sh tree 8 1 spr lce
 ttl 1 1e4 1e4 1e7 1e7 20 IRM IRM 1 0 cold naive 0.75 1e3
```

Notice that to launch ModelGraft simulations we need to set RS=*ttl*, and then we need to choose our downscaling factor Δ; in this case, we follow the rule of thumb presented above, that is $\Delta = C/10 = 1e3$.

5.3 Very Large scenario: ED-sim vs. ModelGraft

In order to quantify the benefits of using ModelGraft w.r.t. ED-sim, in this section we compare the two approaches when simulating the biggest scenario that is currently feasible with ED-sim in ccn$\mathcal{S}$im-v0.4 in terms of content catalog cardinality: $M = 1e9$.

- Catalog cardinality $M = 10^9$. All the seed copies are supposed to be stored in a single repository connected to the root node
- 4-level binary tree topology (i.e., 15 nodes)
- Clients are connected at each leaf node
- Aggregate request rate for each client is $\lambda = 20req/s$
- Catalog popularity following a Zipf-like distribution with $\alpha = 1$
- Cache size $C = 10^6$, equal for each node
- Total number of simulated requests $R = 10^9$
- LRU cache replacement policy for ED-sim, TTL one for ModelGraft
- LCE cache decision policy
- Downscaling factor $\Delta = 10^5$ for ModelGraft

ED-sim command:

```
pint:CCNSIM_DIR$ ./runsim_script_ED_TTL.sh tree 8 1 spr lce
 lru 1 1e6 1e6 1e9 1e9 20 IRM IRM 1 0 cold naive 0.75 1
```

ModelGraft command:

```
pint:CCNSIM_DIR$ ./runsim_script_ED_TTL.sh tree 8 1 spr lce
 ttl 1 1e6 1e6 1e9 1e9 20 IRM IRM 1 0 cold naive 0.75 1e5
```

The 4-level binary tree topology is described in `tree.ned` file, located inside the `network/` folder. In Snippet 4 we report the whole `tree.ned` file as an example. As mentioned in Chap. 4, the main steps are those of extending the `base_network`, specifying number of nodes, clients, repositories, and describing connections. `networks` represents the actual directory where Omnet++ will search for the `.ned` file (in this case `CCNSIM_DIR/networks`).

As it can be noticed, ModelGraft simulation is launched without specifying any T_C file from where T_C should be taken; in this case, ccnSim will search for the default file:

`tc_tree_NumCl_8_NumRep_1_FS_spr_MC_lce_M_1e9_R_1e9_C_1e6_Lam_20.0.txt`

which is provided in the `Tc_Values` folder. In case it would not have been present, ccn$\mathcal{S}$im-v0.4 would have tried to randomly generate Tc values at run-time per each node. This allows ModelGraft to perform any simulation thanks to its feedback loop which iteratively corrects input Tc values. It is, however, a very general heuristic which does not guarantee optimal performance Snippet 5 reports an extract of the `.ini` file which would be created be executing the script for the ModelGraft simulation.

Snippet 4 Ned file for the 4-level binary tree topology.

```
package networks;

network tree_network extends base_network{

parameters:
    n = 15;    //Number of ccn nodes

    // Number of repositories and points of attachment
    num_repos = 1;
    node_repos = "0";
    replicas = 1;

    //Number of clients and points of attachment
    num_clients = 8;
    node_clients = "7,8,9,10,11,12,13,14";

connections allowunconnected:

node[0].face++ <--> { delay = 1ms; } <-->node[1].face++;
node[0].face++ <--> { delay = 1ms; } <-->node[2].face++;
node[1].face++ <--> { delay = 1ms; } <-->node[3].face++;
node[1].face++ <--> { delay = 1ms; } <-->node[4].face++;
node[2].face++ <--> { delay = 1ms; } <-->node[5].face++;
node[2].face++ <--> { delay = 1ms; } <-->node[6].face++;
node[3].face++ <--> { delay = 1ms; } <-->node[7].face++;
node[3].face++ <--> { delay = 1ms; } <-->node[8].face++;
node[4].face++ <--> { delay = 1ms; } <-->node[9].face++;
node[4].face++ <--> { delay = 1ms; } <-->node[10].face++;
node[5].face++ <--> { delay = 1ms; } <-->node[11].face++;
node[5].face++ <--> { delay = 1ms; } <-->node[12].face++;
node[6].face++ <--> { delay = 1ms; } <-->node[13].face++;
node[6].face++ <--> { delay = 1ms; } <-->node[14].face++;
}
```

Snippet 5 `.ini` file for the considered scenario.

```
#General parameters
[General]
network = networks.${net = tree }_network
seed-set = ${repetition}

#############  Repositories #############
**.node_repos = "0"
**.num_repos  = ${numRepos = 1 }
**.replicas = 1

#############  Clients  #############
**.node_clients = "7,8,9,10,11,12,13,14"
**.num_clients = ${numClients = 8 }
**.lambda = ${lam = 20.0 }
**.client_type = "client_${clientType = IRM }"

#############  Content Distribution #############
**.file_size =  1
**.alpha = ${alp = 1 }
**.q = 0
**.objects = ${totCont = 1e9 }
####*.content_distribution_type = "${contDistrType = IRM }"

#############  Forwarding #############
## Strategy layer (interest forwarding):
**.FS = "${ fs = spr }"
**.TTL2 = ${ttl = 1000}
**.TTL1= ${ttl}
**.routing_file = ""

#############  Caching  #############
**.DS = "${ mc = lce }"
**.RS = "${ rs = lru }_cache"
**.C = ${cDim = 1e6 }
**.NC = ${ncDim = 0 }
**.tc_file = "${ tcf = ./Tc_Values/tc_single_cache_NumCl
_8_NumRep_1_FS_spr_MC_lce_M_1e9_R_1e9_C_1e6_Lam_20.0.txt }"
**.tc_name_file = ""

#############  Statistics #############
**.partial_n = ${checkedNodes = 0.75 }
**.start_mode = "${startMode = cold }"
**.fill_mode = "${fill = naive }"
**.downsize = ${down = 1e5 }
**.cvThr = ${cv = 0.005 }
**.consThr = ${cons = 0.1 }
**.num_tot_req= ${ totReq = 1e9 }

## OUTPUT FILES
output-scalar-file = ${resultdir}/ED_T_tree_[key/value].sca
```

In the first place we compare the two approaches when ModelGraft is provided with exact T_C values as input. Results are reported in Tab. 3.2.

If we execute the two simulations, the respective temporal evolutions will be logged in `ED_T_parameters.out` and `TTL_T_parameters.out` files inside the `logs/` directory. From these files it is possible to gather very useful information, such as when caches are being filled (only for ED-sim), when the simulation reaches a steady-state, as well as some aggregated statistics and estimates of the "characteristic time" (T_C) for each node in ED-sim; these values could be used as starting point for TTL simulations. Moreover, all the aggregated statistics can be found in the `ALL_MEASURES_ED_parameters` (for ED-sim) or `ALL_MEASURES_TTL_parameters` (for ModelGraft) file inside the `infoSim/` folder. The `.sca` files are still produced inside the `results/` folder, from which, for example, we can extract the hit probability for each node by using:

```
    pint:CCNSIM_DIR/results$ grep p_hit results/ED{}.sca >
    > | awk '{print $4}' >
    > | sort -n >  p_hit.data
```

Results for the aforementioned comparison are reported in Tab. 3.2, which shows that ModelGraft gains can raise up to more than two orders of magnitude in terms of both CPU time and memory occupancy.

Despite the highest gains are obtained with the former conditions (i.e., exact T_C values as input), ModelGraft is also able to drastically reduce CPU time and memory occupancy w.r.t. ED-sim even when random T_C guesses are provided as input, thanks to the presence of a feedback loop which lets ModelGraft converge to a consistent state according to the simulated scenario. An example of a sensitivity analysis on T_C input values is provided in Fig. 5.1 (for more details please refer [23]), which demonstrates that: (i) ModelGraft accuracy is not affected by the magnitude of the bias of the initial guess; (ii) even though the number of cycles needed to reach such accuracy increases for increasing overestimation/underestimation bias, it, however remains bounded by a small number in practice; (iii) CPU gain is, however, still significant (above 50x-100x) even for very large T_C biases (100x overestimation - 1/100x underestimation, respectively); (iv) memory gain is, as expected, independent of the initial T_C bias.

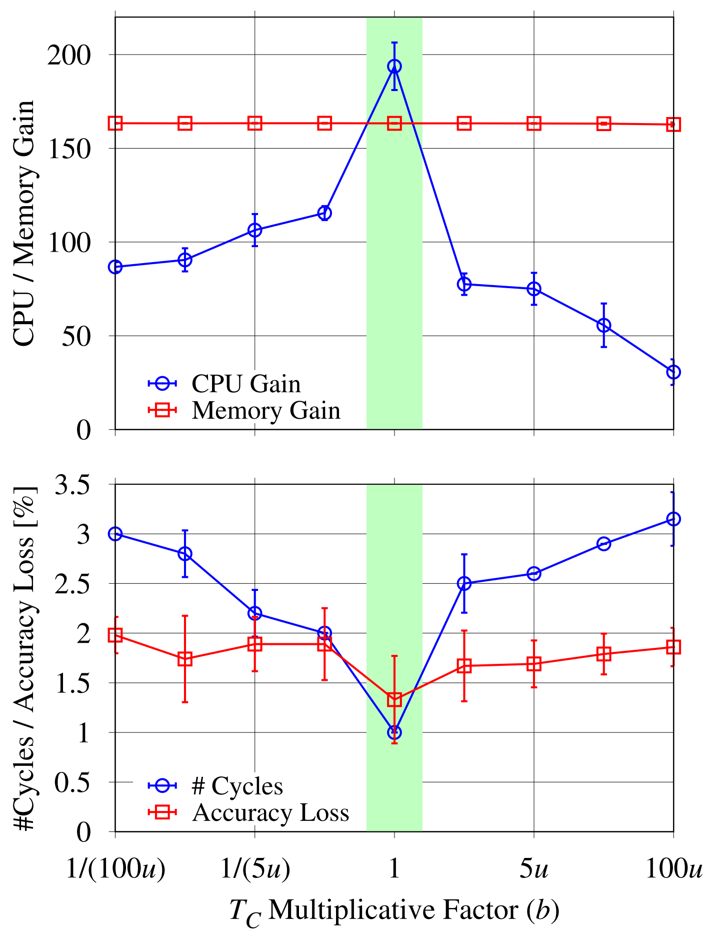


Figure 5.1: T_C sensitivity - ModelGraft performance with variable input T_C values.

Bibliography

[1] http://www.omnetpp.org.

[2] Boost homepage. `http://www.boost.org/`.

[3] Zipf distributed random number generator . `https://github.com/apache/commons-math/blob/master/src/main/java/org/apache/commons/math4/distribution/ZipfDistribution.java`.

[4] A. Araldo, D. Rossi, and F. Martignon. Design and evaluation of cost-aware information centric routers. In *ACM ICN*, 2014.

[5] A. Araldo, D. Rossi, and F. Martignon. Cost-aware caching: Caching more (costly items) for less (isps operational expenditures). *IEEE Transactions on Parallel and Distributed Systems,*, 2015.

[6] Somaya Arianfar and Pekka Nikander. Packet-level Caching for Information-centric Networking. In *ACM SIGCOMM, ReArch Workshop*, 2010.

[7] Giovanna Carofiglio, Leonce Mekinda, and Luca Muscariello. Analysis of latency-aware caching strategies in information-centric networking. In *ACM CoNEXT Workshop on Content Caching and Delivery in Wireless Networks (CCDWN '16)*, 2016.

[8] W. Chai, D. He, I. Psaras, and G. Pavlou. Cache "less for more" in information-centric networks. In *IFIP NETWORKING*, volume 7289, pages 27–40. 2012.

[9] H. Che, Y. Tung, and Z. Wang. Hierarchical web caching systems: Modeling, design and experimental results. *IEEE JSAC*, 20(7):1305–1314, 2002.

[10] R. Chiocchetti, D. Rossi, and G. Rossini. ccnSim: an highly scalable ccn simulator. In *IEEE ICC*, 2013.

[11] C. Fricker, P. Robert, and J. Roberts. A versatile and accurate approximation for LRU cache performance. In *Proc. of ITC24*, Sep. 2012.

[12] M. Garetto, E. Leonardi, and S. Traverso. Efficient analysis of caching strategies under dynamic content popularity. In *Proc. of IEEE INFOCOM*, Apr. 2015.

[13] W. Hörmann and G. Derflinger. Rejection-inversion to generate variates from monotone discrete distributions. *ACM Trans. Model. Comput. Simul.*, 6(3):169–184, Jul. 1996.

[14] 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 1–12, 2009.

[15] N Laoutaris, H Che, and I Stavrakakis. The LCD interconnection of LRU caches and its analysis. *Performance Evaluation*, 63(7), 2006.

[16] Nikolaos Laoutaris, Sofia Syntila, and Ioannis Stavrakakis. Meta Algorithms for Hierarchical Web Caches. In *IEEE ICPCC*, 2004.

[17] V. Martina, M. Garetto, and E. Leonardi. A unified approach to the performance analysis of caching systems. In *Proc. of IEEE INFOCOM*, Apr. 2014.

[18] S. Traverso et al. Unraveling the Impact of Temporal and Geographical Locality in Content Caching Systems. *IEEE Transactions on Multimedia*, 17:1839–1854, 2015.

[19] I. Psaras, W. K. Chai, and G. Pavlou. Probabilistic in-network caching for information-centric networks. In *ACM ICN*, 2012.

[20] G. Rossini and D. Rossi. Evaluating CCN Multi-path Interest Forwarding Strategies. *Comput. Commun.*, 36(7):771–778, Apr. 2013.

[21] G. Rossini and D. Rossi. Coupling caching and forwarding: Benefits, analysis, and implementation. In *Proc. of ACM ICN*, Sep. 2014.

[22] M. Tortelli, D. Rossi, and E. Leonardi. ModelGraft: Accurate, Scalable, and Flexible Performance Evaluation of General Cache Networks. In *Proc. of International Teletraffic Congress (ITC)*, Würzburg, Germany, Sep. 2016.

[23] M. Tortelli, D. Rossi, and E. Leonardi. A hybrid methodology for the performance evaluation of internet-scale cache networks. *Elsevier Computer Networks, Special Issue on Softwarization and Caching in NGN*, 2017.