ccn$\mathcal{S}$im: an Highly Scalable CCN Simulator

Raffaele Chiocchetti, Dario Rossi, Giuseppe Rossini
Telecom ParisTech, Paris, France – `first.last@telecom-paristech.fr`

***Abstract*—Research interest about Information Centric Networking (ICN) has grown at a very fast pace over the last few years, especially after the 2009 seminal paper of Van Jacobson et al. describing a Content Centric Network (CCN) architecture. While significant research effort has been produced in terms of architectures, algorithms, and models, the scientific community currently lacks common tools and scenarios to allow a fair cross-comparison among the different proposals.**

The situation is particularly complex as the commonly used general-purpose simulators cannot cope with the expected system scale: thus, many proposals are currently evaluated over small and unrealistic scale, especially in terms of dominant factors like catalog and cache sizes. As such, there is need of a scalable tool under which different algorithms can be tested and compared.

Over the last years, we have developed and optimized ccn$\mathcal{S}$im, an highly scalable chunk-level simulator especially suitable for the analysis of caching performance of CCN network. In this paper, we briefly describe the tool, and present an extensive benchmark of its performance. To give an idea of ccn$\mathcal{S}$im scalability, a common off-the-shelf PC equipped with 8GB of RAM memory is able to simulate 2-hours of a 50-nodes CCN network, where each nodes is equipped with 10 GB caches, serving a 1 PB catalog in about 20 min CPU time.

I. Introduction

Information Centric Networking (ICN) is a networking paradigm almost 10 years old. Shortly, observing that the Internet is now mostly serving popular content (e.g., and especially video, like in the YouTube portal), ICN architectures propose that (i) content is split in *chunks* (ii) each router becomes a cache for those chunks, so that popular content can be served more efficiently. Among the many ICN architectures sprouted in the last decade, the one which has received, by far, the largest attention is the Content Centric Networking (CCN) paradigm proposed by Van Jacobson et al. [18]

While numerous proposals[1] have seen the day in the ICN and CCN fields, they have typically been evaluated with custom simulators, that are not available to the scientific community. Since this clearly hampers their comparison, and slows down the ICN/CCN research path, there is need for a scalable, efficient and common platform on which the different proposals can be tested and compared.

The problem is however not easy to solve. Indeed, due to the scale of the problem, the usual simulators such as `ns2` [5], cannot be directly used. Still, even that the majority of the work to date deals with rather simple and small-scale scenarios, especially in terms of dominant factors, like catalog, network topologies, and cache size. For instance, the YouTube catalog is estimated to be on the order of 1 PB i.e., 10^{15} Bytes (details in Sec. IV-A): yet, largest catalogs considered in the literature are off by some orders of magnitude (i.e., 20K object or 138GB [11]). Similarly, while [9] sizes to about 10 GB the amount of memory that can be addressed at line speed in ICN architectures, current simulations operate with much smaller caches (e.g., 6.4MB [20]-50MB [11]).

Of course, the design of a scalable simulator, capable of efficiently simulating system dynamics at such large scale is not a trivial task at all. Over the last years, we have developed and optimized ccn$\mathcal{S}$im, an highly scalable chunk-level simulator especially suitable for the analysis of CCN caching performance, that we have released to the scientific community as open source software [3]. As we will briefly overview in the following, ccn$\mathcal{S}$im is a modular, flexible, and fast ICN simulator, capable to deal with different caching algorithms/policies, forwarding strategies, topologies, and popularity laws.

While our previous research has focused on the performance of CCN gathered through ccn$\mathcal{S}$im [14]–[16], [21], this paper focuses on the performance of the ccn$\mathcal{S}$im tool itself. Specifically, two crucial aspects have been taken into account in the design of ccn$\mathcal{S}$im, namely a) *Memory occupancy*: The simulation of very large catalogs and very large cache size is essential to gather CCN performance under realistic settings. Yet, the CDF of commonly used (i.e., Zipf-based) popularity distributions cannot be expressed in a closed form: hence, huge catalogs swell up the memory demand of the simulator; b) *CPU time*: system dynamics have to be represented at *chunk level*, i.e., a minimal data unit, which is typically packet-size or slightly larger (e.g., 10KB). Clearly, efficient operation require a careful engineering and optimization of the most computationally intensive tasks CCN has to deal with.

In order to reduce CPU time we perform the usual optimization steps: i.e., we profile the ccn$\mathcal{S}$im code, identify the bottleneck functions and refactory the code responsible of the largest portion of the execution time. Instead, the memory occupancy is driven by the scenario requirement: hence, we expect that in order to simulate the largest possible scenarios, all the available RAM memory will be used. Since current off-the-shelf server offer multi-core CPU, we also investigate the possibility of parallelizing the execution of ccn$\mathcal{S}$im using Message Passing Interface (MPI).

Overall, our careful engineering of ccn$\mathcal{S}$im allows to simulate very large scale scenarios in a reasonable time: to give an idea of ccn$\mathcal{S}$im performance, a common off-the-shelf PC equipped with 8GB of RAM memory is able to simulate

[1]As our focus is on the open-source tools available for ICN performance evaluation, it would be out of the scope of this paper to provide a full literature review (for a representative sample of recent research on ICN and CCN, the reader may refer to IEEE INFOCOM NOMEN Workshop, ACM SIGCOMM ICN Workshop, and the CCNxCON community meeting).

2-hours of a 50-nodes CCN network, where each node is equipped with 10 GB caches, serving an Internet-like 1 PB catalog in about 20 min CPU time.

II. THE CCN SYSTEM MODEL

At high level, a CCN architectures is a *receiver oriented network of caches*. When the receiver (client) wishes to get a content, he sends an *interest* for a *chunk* of that content. The interest travels over all the network, until it hits either a cache with a *temporary copy* of the chunk, or the repository that stores the *permanent copy* of the content. Then, a data chunk is sent back toward the client, consuming the interest.

While we refer the reader to [18] for a detailed description of the CCN architecture, we need to introduce the main system elements that ccn$\mathcal{S}$im has to implement:

- *FIB*. Nodes forward interests by looking up their Forwarding Information Base (FIB), that points, for each content, to the right output interface.
- *CS*. Every router is equipped with a Content Store (CS), to temporarily cache received data chunks. When an interest hits a cache, the corresponding data is sent back in reply, if cached in the CS, or the interest is forwarded to another node according to the FIB.
- *PIT*. Data is forwarded back to the client by using a Pending Interest Table (PIT) data structure. When a node receives an interest for a content that is not in its CS, it forwards the interest according to the FIB, and adds the incoming interface to the set of interfaces interested by that chunk in a PIT. When the data travels back, it is forwarded according to PIT information, and the PIT information is deleted.

Hence, in CCN, CS acts as a cache, while interests are forwarded according to FIB information and data is forwarded according to PIT information. As interest packets and data chunks travel across CCN nodes, multiple lookups in several data structures have to be performed. Efficient implementations of these data structures is thus a key point in ccn$\mathcal{S}$im.

The CCN architecture involves other aspects, like naming and security, that in our opinion tradeoff with the scalability requirement. Indeed, CCN needs to perform, for each chunk, cryptographic computation on the chunk name for security and correctness of operation. Yet, this computation would definitively constitute a major CPU bottleneck, which tradeoffs with our scalability goal, and that we thus prefer to avoid altogether in ccn$\mathcal{S}$im.

III. SIMULATOR ARCHITECTURE

Basically, ccn$\mathcal{S}$im is a C++ package built on the top of the Omnet++ [7] framework. In the following, we give an overview of the set of ccn$\mathcal{S}$im classes, that are illustrated in Fig. 1, highlighting the key design choices.

A. Catalog and popularity model

Clients represent an aggregate of users who request contents with a given popularity distribution. Requests for new content follow a Poisson process, with a customizable rate. In ccn$\mathcal{S}$im,

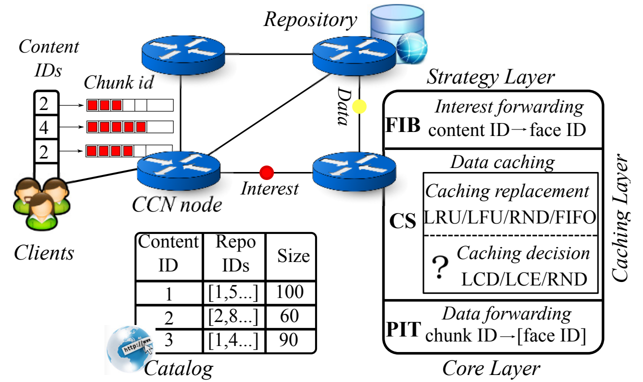


Fig. 1. ccn$\mathcal{S}$im main components at a glance.

users are not CCN nodes, thus they do not implement the whole CCN stack. Each aggregate is implemented as an STL multi-associative map[2] that keeps track of different contents downloaded by individual users.

Popularity model, and hence catalog and content size, represents a crucial aspect of every ICN architecture. We have designed and accurately tuned the catalog and popularity model according to recent literature in Internet metrology (details in [15]). Size of the files in the catalog is a configurable parameter, and follows a geometric law by default [17]. For matters of efficiency, the size of each file is computed during the simulation bootstrap, and is stored within a large static array (named *catalog* in Fig. 1).

We model popularity distribution with a Mandelbrot-Zipf (M-Zipf), shaped by parameters (q, α), that we statically initialized during the startup of the simulator, and we employ a $\log(N)$ binary search for each M-Zipf random number (with N size of the catalog). Notice that while this may seem a minor implementation detail, a binary search rather becomes a requirement when the catalog size reaches $N = 10^8$ objects as for YouTube [12]. The popularity model possibly includes a spatial heterogeneity to account for skew in the popularity law due to geographical/cultural barriers [16].

B. Messages and chunks

At low level, interests and data messages carry a 64-bit unsigned integer, namely the *chunk identifier*. This is a simplification with respect to the original naming structure of CCN, that however gives ccn$\mathcal{S}$im an advantage of several orders of magnitude in terms of memory scalability (e.g., PIT and CS entries consume about 64bits in ccn$\mathcal{S}$im with respect to about 1KB in [6], see Sec. V).

The chunk ID stores information about the content name (most significant 32 bits) and the chunk sequence number (least significant 32 bits). This design represents a tradeoff between space and flexibility. On the one hand, we minimize the space needed for moving content all over the network.

[2]A multimap is like a simple map, but it considers the possibility of having more entries with the same key.

On the other hand, we have 32-bit content identifiers (up to 4 billions of individual contents).

C. *Node architecture*

This simulator is not targeted for a particular topology, and the user can freely arrange the nodes following her needs. However, we provide 8 built-in topologies: five of them are realistic ISP networks (Geant, Abilene, Level3, Qwest, Sprint), and three are synthetic (random, torus and tree). For each network, ccn$\mathcal{S}$im is able to build different routing FIBs (either precomputing the shortest-path and multiple-path based on standard graph algorithms [15] or possibly with a dynamic flooding-based exploration [14]). As we can observe from Fig. 1, a CCN node comprises three different submodules: core, cache, and strategy layer.

Core layer. Is the responsible for PIT management, and communicates with caching and forwarding layers. Any new interest raises a CS lookup. If the data is not in the CS, but a repository for the given content is attached to the node, the CCN node returns the data packet back. Otherwise, it deflects the interest to the forwarding layer. Any data chunk raises a PIT lookup, that eventually sends the content back on all the PIT interfaces for that chunk. The PIT is implemented as an associative map of arrays, indexed by the 64bit chunk identifier.

Caching layer. Caching is one of the crucial aspects of any ICN architecture. CS acts according to a caching *decision policy* (i.e., whether to store the data in CS or not) and a *replacement policy* (i.e., what to drop from CS in case it is full). While we already provide a fairly large number of decision (LCD [19], Random [13], LCE) and replacement policies (LRU, Random, FIFO) we have designed the CS in a modular fashion. Specifically, new replacement algorithms are implemented as modules overwriting the caching polymorphic methods `store()` and `lookup()`; a similar trick is done for new decisions policies with the polymorphic method `isToCache()`.

At low level, CS is an associative map. Since CS lookups are very frequent operations in CCN, we optimized their implementation. In fact, while naive LRU implementations can represent a drastic bottleneck in large-scale simulation, we resort to an efficient LRU implementation through a map of pointers [9] (the map speeds-up the access to the elements in cache, while pointers are used to take the elements order).

Strategy layer. The strategy layer basically takes decisions about interest forwarding, through the `getDecision()` method. This polymorphic function returns a bit mask with the same cardinality of the output interfaces set. A 1 in the i-th position of the mask will forward an interest toward the i-th interface.

The default strategy layer in ccn$\mathcal{S}$im sends messages toward the nearest repository over the shortest path (to speed-up simulation, FIB of each node are pre-filled at the simulation startup and `getDecision()` spoofs from the catalog the repositories who store permanent copies). Other multi-path strategies, both static [15] and dynamic [14] are already available, while further strategies can be implemented by overwriting the `getDecision()` method.

D. *Simulation statistics*

Statistic collection starts only when the cache hit metric has reached a stationary state. In more details, we start with empty caches and, as soon as caches fill up, every node samples its hit rate every t_s simulated time (usually t_s is about hundreds of milliseconds), and the variance of the collected samples is computed every t_w (usually t_w is about tens of seconds).

Only when the cache hit variance falls under a given threshold, the node declares itself stabilized. Statistic collection starts only after *all nodes are stable*. The stationary state is then simulated for a customizable duration (typically one or two hours of simulation time) after which statistics of interest are collected (e.g., hit rate, distance, cache diversity, download time, and so forth).

IV. Simulator benchmark

We now extensively benchmark ccn$\mathcal{S}$im, considering a very challenging scenario, that we describe in Sec. IV-A. Our evaluation is structured along two main axis. The first axis goes along a *profiling of the simulator*, in order to pinpoint the function call representing the major CPU bottleneck. Based on the profiling results, we refactory part of the code to reduce the execution time as much as possible (Sec. IV-B).

The second axis investigates *parallel execution* using Message Passing Interface (MPI) capabilities. The main driver here is the fact that, even though the simulator has been designed to have a small RAM memory footprint, the execution of large-scale simulation (in terms of the catalog, network and cache size) will nevertheless sooner or later pose a RAM bottleneck. Since the typically available off-the-shelf servers have a large number of cores (typically 4-8, if not more), it is worth investigating whether the simulation run can be speed-up by parallelizing the execution of multiple cores (Sec. IV-C).

Finally, we investigate the scalability of the simulation in terms of the network size, always under the challenging YouTube scenario, developing a simple model of the requirements in terms of memory and execution time that ccn$\mathcal{S}$im user can expect (Sec. IV-D).

A. *Benchmark scenario*

As target application for our benchmark, and in reason of the growing importance of video application in the future Internet, we consider one of the most popular VoD application nowadays, namely YouTube. The YouTube catalog is sized at about 1 PB, as it consists of about 10^8 files [12] having geometrically distributed size with average 10MB [17]. Contents are partitioned in 10KB chunks, thus the average file size in chunks turns to be C=1000 chunks.

As we are not interested on the CCN performance, we select a synthetic torus topology, and let the network size

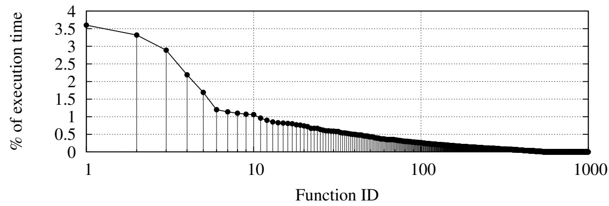


Fig. 2. ccn$\mathcal{S}$im profiling. Functions ordered by decreasing execution time.

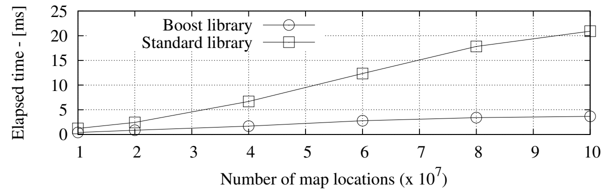


Fig. 3. Access time comparison of STL and Boost libraries.

grow from 10-50 nodes (specifically, from 3×3 to 7×7 torus). Nodes are equipped with 10GB [9] (or 10^6 chunks). The decision/replacement pair considered here, is LCE/LRU (the most used within the ICN proposals). As for the strategy layer, interest packets are forwarded toward the nearest repository along the shortest path.

Clients aggregate are attached to each node, and users request have average arrival rate of $\rho = 20\,\text{Hz}$. We instrument ccn$\mathcal{S}$im to simulate $T = 2\,\text{hour}$ of simulated time after the cache hit rate stabilizes. Overall, the number of per-chunk operation in steady state can be evaluated as ρCTD, with D average path length, and is thus on the order of $[10^8, 10^9]$ depending on the specific scenario.

Results reported in this paper are gathered on a off-the-shelf server equipped with Intel Xeon E5620 8-cores CPU (running at 2.4GHz), and with 12 MB L3 cache and 24 GB RAM memory.

B. Profiling ccn$\mathcal{S}$im

For profiling, we used the standard GNU profiler *gprof*, whose results are shown in Fig. 2 – where for the sake of the illustration we show only the first thousand functions. In the picture, functions are ranked by decreasing execution time percentage. For instance, from Fig. 2 is easy to see that if the total execution time is 100 seconds, the CPU has been busy with function having rank 1 for about 3.5 seconds. We see that the curve is slowly decreasing for most of the ccn$\mathcal{S}$im functions: this means that each function is individually accounting for only a small fraction of the total execution time. This is a bad scenario for optimization, as after profiling, one would reimplement only the few functions that are representative of the bulk of the CPU time. However, Fig. 2 shows that such “quick win” approach does not apply to ccn$\mathcal{S}$im.

Still, the very first handful of functions are responsible for about 15% of the CPU time. A more in-depth inspection, reveals these functions to be responsible for providing access to CCN data structures (CS, PIT, FIB, catalog, etc.) that are implemented as associative maps, array, and so forth in C++. As these structures are accessed very often in CCN (basically, most chunk-based operations will require multiple accesses into the structures, on multiple nodes), we can reduce execution time by adopting their most efficient C++ implementation.

We thus consider the two most common libraries to implement such structures, namely the Standard Template Library (STL) [22] and the Boost library [1]. The comparison is shown in Fig. 3, that plots the elapsed time of a stress-test program filling the associative map with different integer sequences. Even for very simple operations, the Boost library outperforms the STL of a factor of 4-5. This performance gap is due to the fact that while STL maps are ordered and implemented through a red-black tree, Boost maps are fully unordered and implemented through more efficient hash functions. Apparently, the Red-Black tree maintenance introduces a significant overhead especially for large structures.

The comparative experiments let us conclude that Boost is more efficient with respect to the STL implementation we were using so far. We therefore refactor ccn$\mathcal{S}$im code using unordered Boost hash-map for any associative map. Notice that while there is a factor of 4-5 speedup in using Boost, this will affect only the first handful of functions related to data structure access. Since these were accounting for about the 15% of the CPU time, we can expect the overall CPU time after refactoring with Boost to be about 85% of the previous execution time under STL.

C. Parallelizing ccn$\mathcal{S}$im

A Parallel Discrete Event Simulation (PDES) has basically two meanings: distributing the model over *different computers* as a meaning to reduce the memory occupancy of the simulator; or distributing the model over *multiple processors* for optimizing its execution time. In our case, we’re interested in pursuing the second goal only, since otherwise RAM may become a bottleneck, leaving many core possibly unused.

Omnet++ has native support of PDES through Message Passing Interface (MPI). As MPI is a built-in feature, changes in the underlying ccn$\mathcal{S}$im code are minimal. Among different parallel algorithms supported, especially worth of interest is the Ideal Simulation Protocol (ISP) introduced by [10], as it helps to determine the maximum speed-up achievable by any PDES algorithm for a particular model and simulation environment.

We start our experiment plotting the simulation duration as function of the number of parallel processors over which the simulations are split up. We set ccn$\mathcal{S}$im for simulating half an hour of a simpler CCN network (w.r.t. the one described in Sec. IV-A) after the transient period ended. Results are shown in Fig. 4, and are rather counterintuitive. In fact, performance *significantly worsens* when we increase the number of parallel processors.

To explain these results, we have to briefly introduce PDES concepts of *lookahead* and *laziness*. The *lookahead* is asso-

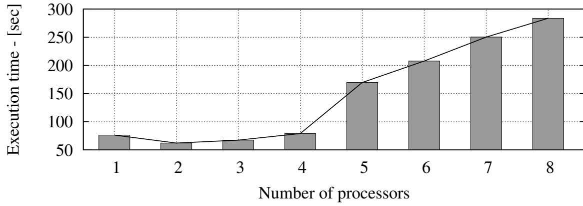


Fig. 4. Simulation duration vs parallelism degree.

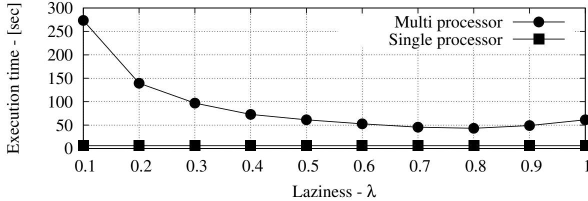
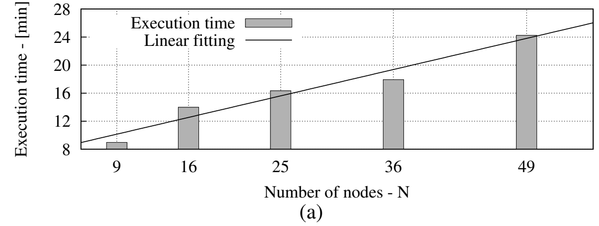


Fig. 5. Simulation duration vs laziness.



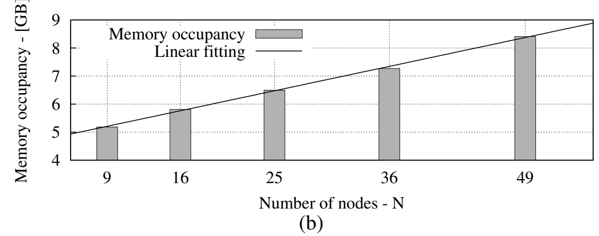


Fig. 6. ccn$\mathcal{S}$im (a) Execution time and (b) Memory occupancy.

ciated with the ability of a logical process (LP) to predict its future behavior: at any simulation time t, if an LP can predict that the earliest event it will cause to occur in another LP is no sooner than $t + \ell$, its lookahead turns to be ℓ. Intuitively, we can say that a small lookahead value badly affects the performance of the overall PDES system, as LPs have to very often synchronize among themselves. The system *laziness* is then correlated with the frequency at which synchronization messages are exchanged between different LPs. In more detail, the laziness is indicated with $\lambda \in [0, 1]$ and represents the synchronization rate, with maximum (minimum) synchronization rate is achieved for λ equal to 1 (0). Generally, a rule of thumb in PDES is to roughly approximate the synchronization period with the system lookahead.

In Fig. 5 we plot the simulation duration as a function of the laziness λ. The plot shows that increasing the synchronization rate λ, ccn$\mathcal{S}$im performance tends to ameliorate. At the same time, the number of synchronization messages, and so the percentage of time that each individual CPU devotes to the synchronization task, grow with λ: therefore, increasing λ can also turn into an excessive overhead (see that execution time increases for $\lambda > 0.8$).

The most important takeaway from Fig. 5 is however that, generally, single-processor ccn$\mathcal{S}$im execution is more efficient than its multi-processor counterpart[3] We conclude that the bulk of the CCN operations requires high synchronization frequencies, which entails that CCN simulation has an inherently small lookahead ℓ and is thus inherently non parallelizable.

D. Overall performance

Finally, we report the expected simulation time and memory occupancy for large-scale CCN simulation under the scenario described in Sec. IV-A. Fig. 6-(a) plots the running time of a simulation, as function of the network size. We can breakdown the total simulation duration t_{tot} (i.e., the CPU time) as the sum of the *bootstrap* time t_B (e.g., fill the catalog, build the network, allocate data structures, build reverse index to speed-up lookup, etc.), plus the cache *fill* time t_F (especially important in case of cold startup with empty caches), plus a *transient* period until the cache hit rate reaches a steady state t_T, plus the CPU time t_S needed to run 1 hour of simulated time (in this case of figure). Simplifying, we can write $t_{tot} = t_B + N t_{sim}$ where t_B accounts for the one-time startup cost and $t_{sim} = t_F + t_T + t_S$ aggregates the time spent in running the CCN dynamics per-node. A linear regression yields to a bootstrap time of $t_B = 7\,\text{min}$ and $t_{sim} = 0.3\,\text{min/node}$ as for the CPU time needed to simulate 2 hour of YouTube catalog. This simple model tells us that (letting aside RAM bottlenecks) about 2-days of CPU time are sufficient to simulate 2 hours of YouTube catalog over a 10,000 nodes network with ccn$\mathcal{S}$im.

Fig. 6-(b) depicts the memory occupancy for each individual simulation. As we can see, the largest simulated 50-nodes network requires about 8GB of RAM: thus, despite we cannot parallelize a single simulation with PDES, we can in principle run several single-core simulations in parallel[4]. To estimate RAM requirements, consider that the catalog stores several useful information (e.g., the repositories who own the permanent copies of each content), and that furthermore every CCN node implements a CS and PIT data structures[5]. Fig. 6-(b) plots the memory consumption as a function of the network size. A linear regression over Fig. 6-(b) data yields $m_{tot} = m_C + N m_N$ where $m_C = 4.5\,\text{GB}$ is the baseline

[3]The only exception (not shown for lack of space) is represented by the case of hot-startup, i.e., when caches are pre-filled during a warm-up phase at time $t = 0$ with random content, proportionally to the catalog popularity. Since cache warm-up does not need synchronization (as each cache is independent), it may result in shorter execution time in PDES.

[4]Consider however that this kind of parallelism may tradeoff with the scale of the scenario. Indeed, the catalog already represents a significant memory footprint (a 10^8 array of 64bit integers requires about 1GB of RAM), and clearly memory requirement grows with the catalog size, the number of chunks per file, the size of individual CCN caches and the size of the CCN network.

[5]The FIB does not directly impact RAM requirement, as it could be responsible for significant RAM footprints only for huge networks.

memory occupancy for handling a YouTube-like catalog (and all its associated fields), and $m_N = 80$ MB/node accounts for CCN nodes memory requirement (to store FIB, PIT and 10^6 chunks-long CS data structures).

V. Related work

While providing a literature review of CCN and ICN is out of the scope of this work, it is useful to point the reader to [8] for a general survey of ICN, and to [18] for details of CCN architecture. Closer work to ours is represented by [2], [4], [6], that are, to the best of our knowledge, the only CCN-related software readily available to the scientific community.

In more detail, CCNx [4] is a fully operational prototype of CCN. However, CCNx does not provide per se any testing, validation or experimentation capabilities. As such, users have to instrument ad hoc testbeds or PlanetLab, which is a rather complex task: as such, this generally requires a preliminary simulation step in order to understand system dynamics and define and select the best performing algorithms to implement in the prototype.

An intermediate step toward the above solution is represented by ns3 simulation with Direct Code Execution (DCE) [2] of the (opportunely recompiled) CCNx prototype. This approach has the advantage of using the same codebase of CCNx, and additionally simplifies the experimentation. Yet, as the *whole* CCNx stack (e.g., including crypto due to security) is run, this hinders scalability of the simulation.

Finally, yet another option is ns3 simulation with the `ndnSIM` [6] simulator very recently developed at UCLA, which is the closest work to our. Yet, since the primary goal of `ndnSIM` is completeness, it is far less scalable with respect to ccn$\mathcal{S}$im – as the memory footprint of individual PIT and CS entries is about 2 orders of magnitude larger than in ccn$\mathcal{S}$im.

VI. Conclusion and Future work

This work presents and benchmarks an highly scalable and open-source CCN simulator, named ccn$\mathcal{S}$im. The scalability of ccn$\mathcal{S}$im is the joint result of key design choice, as well as a careful engineering (i.e., profiling and code refactoring).

Given the availability of a relatively large number of cores in current off-the-shelf servers, we explored the parallel (PDES) execution of ccn$\mathcal{S}$im. Indeed, since for very large scenarios memory may become a bottleneck, it would have been useful to exploit all the available cores, that would otherwise remain idle in a single-core execution. Unfortunately, PDES does not generally yield to shorter execution time, as due to the small system lookahead, the major part of the additional CPU power is wasted in the synchronizing process among processors.

The main message of the benchmarking section is that scalability issues, either in terms of memory and latency, are not due to the size of the network. Rather, the main factor affecting ccn$\mathcal{S}$im scalability is the size of the catalog, especially for what concerns RAM memory occupancy (that we have seen to represent the crudest bottleneck). In future work, we aim at reducing these catalog memory requirements by smarter representation of each catalog entry (see Sec. III-A). Indeed, limiting the size of each content to 65536 chunks (16 bit per entry), and the maximum number of repositories in the network to 16 (using bitmaps of 16 bit), each catalog entry occupies 4 bytes – a significant reduction compared to the current implementation requiring 16-76 bytes for 1-16 repositories respectively (32 bits for the file size, plus a 64 bits pointer to an array of 32 bits per repository). To make a conservative example in the case of single repository, for a 10^8 items catalog we could reduce memory occupancy from current $10^8 \cdot 16 = 1.6$GB to about $10^8 \cdot 4 = 400$MB.

Acknowledgements

This work has been carried out at LINCS http://www.lincs.fr. The research leading to these results has received funding from the European Union under the KIC EIT ICT Labs Project Smart Ubiquitous Contents (SmartUC).

References

[1] Boost homepage. http://www.boost.org/.
[2] Ccn ns3 quick start documentation. http://www-sop.inria.fr/members/Frederic.Urbani/ns3dceccnx/getting-started.html.
[3] ccnSim homepage. http://www.infres.enst.fr/~drossi/ccnSim.
[4] Ccnx homepage. http://www.ccnx.org/.
[5] The network simulator - ns-2. http://www.isi.edu/nsnam/ns/.
[6] Ns-3 based named data networking (ndn) simulator. http://ndnsim.net.
[7] Omnet++ homepage. http://www.omnetpp.org/.
[8] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman. A survey of information-centric networking. *Communications Magazine, IEEE*, 50(7):26 –36, july 2012.
[9] S. Arianfar and P. Nikander. Packet-level Caching for Information-centric Networking. In *ACM SIGCOMM, ReArch Workshop*, 2010.
[10] R L Bagrodia and M Takai. Performance evaluation of conservative algorithms in parallel simulation languages. *IEEE Transactions on Parallel and Distributed Systems*, 11:395–411, 2000.
[11] Giovanna Carofiglio, Massimo Gallo, Luca Muscariello, and Diego Perino. Modeling Data Transfer in Content-Centric Networking. In *ITC*, 2011.
[12] M. Cha, H. Kwak, P. Rodriguez, Y.Y. Ahn, and S. Moon. I tube, you tube, everybody tubes: analyzing the world's largest user generated content video system. In *ACM IMC*, 2007.
[13] W. Chai, D. He, I. Psaras, and G. Pavlou. Cache less for more in information-centric networks. pages 27–40, 2012.
[14] R. Chiocchetti, D. Rossi, G. Rossini, G. Carofiglio, and D. Perino. Exploit the known or explore the unknown?: hamlet-like doubts in icn. In *ACM ICN*, pages 7–12, 2012.
[15] D. Rossi G. Rossini. Caching performance of content centric networksunder multi-path routing (and more). Technical report, Telecom ParisTech, 2011.
[16] D. Rossi G. Rossini. A dive into the caching performance of content centric networking. In *IEEE 17th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD'12)*, 2012.
[17] P. Gill, M. Arlitt, Z. Li, and A. Mahanti. Youtube traffic characterization: a view from the edge. In *ACM IMC*, pages 15–28, 2007.
[18] V. Jacobson, D. K. Smetters, N. H. Briggs, J. D. Thornton, M. F. Plass, and R. L. Braynard. Networking Named Content. In *ACM CoNEXT*, 2009.
[19] N Laoutaris, H Che, and I Stavrakakis. The LCD interconnection of LRU caches and its analysis. *Performance Evaluation*, 63(7), 2006.
[20] I. Psaras, R. G Clegg, R. Landa, W. K. Chai, and G. Pavlou. Modelling and Evaluation of CCN-Caching Trees. *IFIP Networking*, 2011.
[21] D. Rossi and G. Rossini. On sizing ccn content stores by exploiting topological information. In *IEEE INFOCOM, NOMEN Worshop,*, Orlando, FL, March 25-30 2012.
[22] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.