Icarus: a Caching Simulator for Information Centric Networking (ICN)*

Lorenzo Saino, Ioannis Psaras and George Pavlou
Department of Electrical and Electronics Engineering
University College London
London, UK
{l.saino, i.psaras, g.pavlou}@ucl.ac.uk

ABSTRACT

Information-Centric Networking (ICN) is a new networking paradigm proposing a shift of the main network abstraction from host identifiers to location-agnostic content identifiers. So far, several architectures have been proposed implementing this paradigm shift.

A key feature, common to all proposed architectures, is the in-network caching capability, enabled by the location-agnostic, explicit naming of contents. This aspect, in particular, has recently received considerable attention by the research community.

However, despite this wide interest, there is a shortage of publicly-available tools suitable for evaluating the performance of caching systems effectively. In fact, all available simulators or emulators are either bound to a specific architecture or cannot execute simulations at the scale required and within a reasonable time-frame.

To address these issues, we present *Icarus*, a Python-based caching simulator for ICN. *Icarus* allows users to evaluate caching strategies for any ICN implementation and also provides modelling tools useful for caching research.

Categories and Subject Descriptors

I.6.7 [**Simulation and Modeling**]: Simulation Support Systems—*Environments*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Software libraries*

General Terms

Design, Performance, Experimentation

Keywords

ICN; caching; simulator

*Source code and documentation of the *Icarus* simulator are available at `http://icarus-sim.github.io`. *Icarus* is released under the terms of the BSD license.

SIMUTools 2014 March 17–19, Lisbon, Portugal.

1. INTRODUCTION

The Internet is witnessing a shift from a host-to-host communication system to a content distribution platform. In fact, apart from traditional applications which require strict point-to-point (or machine-to-machine) connections (e.g., telnet, ssh, VoIP), the majority of current Internet traffic is inherently host- and/or location-independent. Examples of such traffic include Web, bulk data transfer, video distribution, but also more recent advances on social networking platforms. To accommodate the need for a content-oriented Internet, the research community has recently put a considerable effort into rethinking Internet architecture having in mind user-to-content traffic as the prevalent usage pattern.

This novel networking paradigm, known as Information Centric Networking (ICN), puts content at the forefront of any communication between any two users of the system, ignoring concepts of location- and host-identifiers. In ICN, contents are explicitly and uniquely named and users request contents based on those content identifiers (instead of traditional node identifiers, i.e. IP addresses). Several different architectural approaches have already been proposed to accommodate the required changes in the network architecture. Examples include CCN/NDN [19], COMET [10], PSIRP/PURSUIT [15] and NetInf [14]. Although different architectures have different objectives and, hence, exhibit different operational properties, they share the following common features.

- **Request-response model.** In an ICN environment a user sends requests for explicitly named contents to which the network replies. This is similar to `HTTP GET` requests, but happens at the network-layer, rather than at the application-layer. Furthermore, requests and responses in ICN take place *per chunk*, rather than per whole object or file.

- **Location independence.** Content names do not reflect the location (machine) where the content is stored. Rather, content naming can be based on any, possibly human-readable, sequence of bytes. Names, however, in most ICN architectures are used to route requests to contents and contents to users, hence, naming has to follow a specific pattern depending on the *route by name* approach adopted.

- **In-network caching.** Explicitly named chunks of content can be cached in arbitrary locations of the network and be retrieved from there once subsequently requested. This is in stark contrast to the buffering of

Simutools 2014, March 17-19, Lisbon, Portugal

DOI 10.4108/icst.simutools.2014.254630

IP packets, which once sent to the destination IP address cannot be reused (unless in case of retransmission to the same destination IP address), as their content is unknown. Instead, an explicitly named content chunk can be cached and reused without the need for proxies or redirections.

In particular, in-network caching has recently received considerable attention by the research community. Compared to traditional approaches to network caching, such as proxy-caching and hierarchical caching, which largely depend on overlay architectures, in-network caching touches upon more critical network operations. In-network caching vaguely resembles P2P operations pushed at the network-layer of the architecture. However, recent advances in this area have identified several challenges that warrant deeper investigation. For example, ICN caching takes place at network routers and is therefore happening at line-speed. Line-speed operation, in turn, raises scalability and efficiency concerns, but it also prohibits co-operative techniques between caches themselves, or between caches and the control plane.

Several techniques have been proposed already to tackle these peculiar characteristics of in-network caching operations (see for example, [6], [9], [24], [26], [28]), which are substantially different to past Web- and hierarchical-caching [7], [11], [16] requirements. This new field of research has also triggered efforts towards the development of new simulation software (e.g., [3], [4], [13]) to assess the performance and viability of the new networking paradigm.

These new simulation tools attempt to incorporate ICN functionalities, which are missing from traditional network simulators. However, all ICN simulators currently available are not suitable for efficiently evaluating the specific aspect of in-network caching. In fact, all of them present one or more of the following fundamental limitations.

- **Support for only a specific architecture.** Most ICN simulators available have been built to evaluate the performance of a single architectural approach and have not been designed to be extensible to support other architectures.

- **Poor scalability.** Caching nodes may converge very slowly to their steady state, depending on traffic patterns. For this reason, simulations may require from few hundred thousands to tens of millions of requests to yield reliable results. Most ICN simulators have been designed focusing on other aspects, such as protocol interoperability and congestion control, which can be generally evaluated with far smaller simulations. As a result, they been implemented with a considerably lower level of abstraction than what required for caching, which makes them unable to run large scale simulations within a reasonable time-frame.

- **Inability to run trace-driven simulations.** The use of trace-driven simulations is necessary to produce reliable caching results. In fact, synthetic stationary workloads fail to capture real-world phenomena impacting caching performance, such as flash-crowd effects, geographic diversity and more generally temporal and geographical locality patterns. No ICN simulators currently support this feature.

As a consequence, most research papers regarding ICN caching present results generated either by running small scale experiments using publicly available simulators or by using custom built simulators which are not made available to the community. Clearly, the lack of publicly available simulators suitable for this purpose can strongly affect the reliability of research results.

To address all these issues, we present *Icarus*, a simulator specifically designed for evaluating the performance of caching systems in the context of Information Centric Networking. *Icarus* implements all required features that are missing from other simulators.

In designing *Icarus*, we focused on satisfying two key non-functional requirements.

- **Scalability.** This requirement has been identified because, as mentioned earlier, reliable caching results may require experiments encompassing millions of simulated content requests. This is required to reach steady state in all caching nodes. As a result, *Icarus* has been implemented to run experiments of that size in a reasonable time-frame. As we will show more in detail in section 5, *Icarus* is capable of simulating several hundred thousand requests per minute per core on commodity hardware. As a result, it could easily complete simulations consisting of millions of requests in few minutes.

- **Extensibility.** This requirement is fundamental to ease the adoption of this simulator by the community. In fact, since caching research is currently very active, simulators' requirements are also changing very fast. As a result, by focusing on achieving extensibility, we expect this tool to serve the purpose of a larger user base. We also expect that making this tool publicly-available will improve reliability and reproducibility of research results. As a matter of fact, *Icarus* has already been used in ICN caching research to generate the results presented in [9], [24] and [28].

The remainder of this paper is organized as follows. Section 2 provides an overview of the related work in the area, as well as the motivation for this work. Section 3 describes the implementation of the *Icarus* simulator. Section 4 describes the modelling tools provided by *Icarus*. Section 5 evaluates the performance of the *Icarus* simulator in terms of CPU and memory utilization. Finally, conclusions are drawn in section 6.

2. RELATED WORK

The need for simulation software for the new Information-Centric Networking paradigm has already attracted research efforts. Different projects in the area have built their own simulation or emulation software to test the performance of their architectures (e.g., [1], [2]). Related to the Named Data Networking (NDN) [4] project, there already exist three different simulation/emulation software platforms (CCNx [2], ndnSIM [4] and ccnSim [13]).

CCNx [2] is an emulation testbed, which incorporates most of the operations included in the CCN proposal [19]. These operations focus heavily on security issues, as well as end-node and router design. As CCN proposes changes in the traditional OSI layer architecture of the Internet, CCNx is focusing on the interoperability of these new layers in a real

simulation testbed, while it is paying less attention to transport and caching operations. Due to this reason, alternative simulation software has been developed to focus on transport and caching operations of the architecture. In particular, ndnSIM [4] builds on top of ns-3 [18] and provides all the required hooks to evaluate mainly the transport behaviour of the new networking paradigm under CCN. However, the structure of ndnSIM is not suitable for easy extension and evaluation of in-network caching strategies, although it does support router caching.

ccnSim [13] has been built to focus mainly on the caching behaviour of the architecture proposed in [19]. ccnSim is a packet-level simulator based on Omnet++ [30] and is the one closer to our work presented in this paper. Although ccnSim [13] supports simulation of large catalogue and cache sizes and it is shown to scale to simulations of millions of requests, its implementation is bound to the on the design of the NDN/CCN architecture. As a consequence, it is not suitable for evaluating the performance of caching systems in architectures different from CCN. For example, ccnSim does not allow asymmetric routing of request and content packets because not supported by the CCN architecture. In addition, it does not support the execution of trace-driven simulations, which is a feature of primary importance for caching research.

We argue that *Icarus* comes to fill a gap in the simulation software for ICN research, as it provides features that apply to the more general context of ICN, without being tied to a specific architecture or approach only. Furthermore, the easy extensibility of its Python-based implementation (as discussed in the next section) allows for easy implementation of new features and strategies, as well as for scalable and fast large-scale simulation experiments.

3. IMPLEMENTATION

3.1 Architecture and design

As already mentioned earlier, *Icarus* has been designed to satisfy two main non-functional requirements: *extensibility* and *scalability*.

We addressed the extensibility requirement with a number of design decisions.

First of all, we selected Python as programming language. This provides several advantages. For example, its high-level syntax which can be learned with a little learning curve. In addition, the availability of scientific, simulation and network modelling libraries such as NumPy/SciPy [20], NetworkX [17] and FNSS [27] further simplifies the implementation of new functionalities by its users.

Extensibility has also been achieved by adopting appropriate design patterns providing good modularity.

For example, in implementing components most likely to be replaced and extended in the future (e.g. cache eviction policies, routing strategies and so on) we used a proxy design pattern together with a plug-in registration system.

We illustrate how this system works by showing how a new replacement policy can be added to *Icarus* (see snippet below). In this case, a user simply needs to implement the new policy in a class extending the `Cache` class and overriding relevant methods to implement the desired behavior. On top of the class implementation, the decorator `register_cache_policy` is invoked to register this implementation to the cache policy registry under the name `FOO_POLICY`. This registration makes the implementation discoverable and usable by the simulator.

```
@register_cache_policy('FOO_POLICY')
class FooCache(Cache):
    ...

    def get(self, k):
        ...

    def put(self, k):
        ...
```

At this point, to run experiments using this policy, the user only needs to specify it in the configuration file, as shown here below.

```
...
policies = ['LRU', 'LFU', 'FOO_POLICY']
...
```

The scalability requirement instead has been addressed mainly by selecting appropriate network abstractions that provide a good tradeoff between realism and simplicity. As a result, *Icarus* operates with flow-level granularity, whereby the lowest-level event is the download of a content object. The flow-level abstraction would make *Icarus* inaccurate for simulating other aspects of ICN architectures, such as congestion control and protocol interoperability. However, these functionalities are outside the scope of this work and, anyway, they are already satisfactorily addressed by existing simulators. In addition, scalability has also been addressed by using highly optimized implementations of complex routines provided by the Scipy package and by the optimization of the code through extensive profiling.

The *Icarus* simulator implements the workflow depicted in Fig. 1. Accordingly, the program execution is carried out following these four sequential steps:

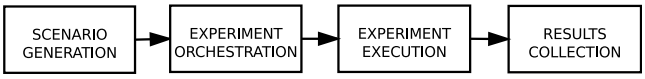

Figure 1: Icarus workflow

- **Scenario generation:** This stage comprises all the steps required to set up a fully configured network topology and a random event generator for the simulation. The scenario generation is based on the FNSS toolchain [27], as explained in more detail in section 3.2.
- **Experiment orchestration:** In this stage, the simulator reads from a configuration file the range of parameters that the user desires to simulate (e.g. cache sizes, cache policies, content popularity distribution) and sets up experiments with all combinations of parameters required. It then schedules the parallel execution of experiments over a pool of processes.
- **Experiment execution:** This stage consists in the actual execution of the experiment. An instance of a simulation engine is provided with a scenario description (i.e. a network topology and an event generator). The engine reads events from the generator and dispatches them to the relevant handler. One or more

data collectors measure various user-specified metrics and, at the end of the experiment, return results to the engine which will then pass them on to a results aggregator.

- **Results collection and analysis:** After each experiment terminates, results are collected by an object which aggregates them and allows users to calculate confidence intervals, plot results or serialise data in various formats for subsequent processing.

In line with our objective to make *Icarus* extensible, all these four subsystems have been designed to be very loosely-coupled among each others. This has been achieved by using a façade design pattern to hide the internal implementation of each subsystem to other components and to use a simplified interface for interactions among them.

The following sections will describe more in detail how the stages presented above are implemented.

3.2 Scenario generation

A simulation scenario is modelled as a tuple of two objects: a network topology and a request generator.

The network topology is an instance of an FNSS `Topology` class [27], which extends the NetworkX `Graph` class [17]. All network configuration (e.g. cache and source placement, link delays and capacities) are embedded in the `Topology` object as annotations of edges and nodes. The modelling of a network using an FNSS `Topology` object allows users to easily generate synthetic topologies or import them from various datasets using the functions provided by FNSS. In addition, since the FNSS `Topology` class extends the NetworkX `Graph` class, it is possible to use all functions provided by NetworkX, such as calculation of shortest path, centrality metrics and network diameters, all of which are required by various cache placement and request-to-cache routing algorithms.

The event generator is a Python generator (i.e. an iterator) whose items are a tuple containing:

- the timestamp at which the event occurs,
- a dictionary containing all the parameters describing the event (e.g. node issuing a request, content requested and so on).

This generator is simply an interface provided to the simulation engine. The internal implementation can be easily modified or new implementations can be added without modifying any other component. Currently, *Icarus* provides functions to create request schedules in the following ways.

- Synthentically, with content requests generated following a Poisson distribution with Zipf-distributed content popularity.
- By parsing requests from Squid proxy[1] log files (which is also the format in which traffic traces provided by the *IRCache* project[2] are available) or from the Wikibench dataset [29].
- By importing complex synthetic workloads generated by GlobeTraff [21].

[1] `http://www.squid-cache.org/`
[2] `http://www.ircache.net`

3.3 Experiment orchestration

The experiment orchestration subsystem is responsible for (i) parsing configuration parameters, (ii) obtaining relevant topologies and event generators from the scenario generation subsystem and (iii) dispatching them to the experiment execution subsystem.

The configuration is parsed from a Python-formatted file that is provided by the user as a command line parameter when *Icarus* is launched. The configuration file contains, among other attributes, the ranges of parameters for which experiments must be repeated, as well as the number of repetitions to run for each combination of parameters. Examples of customisable parameters include number of requests, cache size, content population, content popularity distribution, cache eviction policies and caching and routing strategies.

The decision of formatting the configuration file as a Python source file simplifies the implementation of the parser. In addition, it allows users to use powerful Python features, such as list comprehension, to easily specify complex ranges and combinations of parameters.

After parsing the configuration, the subsystem computes all combinations of parameters to be evaluated. For each combination, it first retrieves the relevant scenario from the scenario generation subsystem and then passes it to a process running an instance of the experiment execution subsystem.

Icarus supports the parallel execution of experiments to exploit the capabilities of multi-core machines. However, it should be noted that it does not run a single experiment on multiple cores as in Parallel Discrete Event Simulations (PDES). Differently, *Icarus* allocates a single experiment to one core and parallel execution is achieved by running different experiments simultaneously on different cores.

The rationale behind this decision is that in caching research it is extremely rare to carry out performance evaluations with a single experiment. Instead, several experiments are executed using different combinations of parameters to evaluate results sensitivity against various factors or repeated several times to obtain statistically significant results. Therefore, as we show in more detail in section 5, in a normal simulation campaign comprising several experiments, we can achieve a nearly linear speedup without PDES mechanisms.

3.4 Experiment execution

The experiment execution subsystem is responsible for running a single experiment and collecting results.

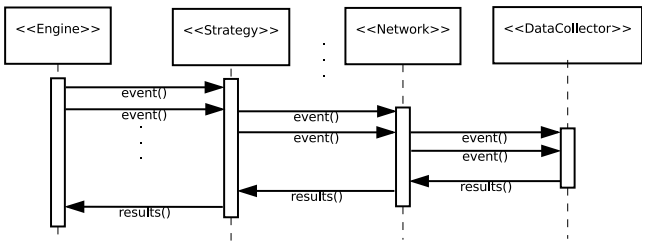

Figure 2: Experiment execution sequence diagram

As shown in Fig. 2, this subsystem comprises the following four components:

- `Engine`: it is in charge of instantiating a caching and routing strategy for the experiment to which all events will be dispatched as they are read from a request

schedule. Once an experiment is terminated, the engine is in charge of receiving results to pass to the results collection and analysis subsystem.

- `Strategy`: it implements the logic according to which requests and contents are routed and cached in the network. `Strategy` in reality is an abstract class, which can be extended by a concrete class implementing a caching and routing strategy. *Icarus* already provides implementations for the most common strategies (see sec. 3.6). However, it is possible to easily implement a new strategy by overriding methods of the `Strategy` class without any modification to the rest of the code.

- `Network`: this component models the simulated network on which the `Strategy` component operates.

  It is implemented using a Model-View-Controller (MVC) design pattern. Accordingly, a strategy executes operations on the network exclusively by calling methods of the `NetworkController` object. The controller updates the state of the `NetworkModel` object, which maintains the current state of the simulated network. In turn, the model updates the `NetworkView`, which provides the strategy with an up-to-date view of the network.

  The `Network` also reports events to a data collector for gathering results. By doing so, it abstracts the complexity of data collection from strategy implementations. This largely simplifies the implementation of new strategies, which we reckon will be a popular use case of our simulator, since research in request routing and content placement techniques is currently very active.

- `DataCollector`: it receives notifications about every event occurring in the network. The `DataCollector` component, however, does not collect data itself. On the contrary, it dispatches events to various data collectors, each in charge of measuring a specific metric. The list of data collectors to which data is dispatched is specified by the user in the configuration file.

  Icarus provides collectors for measuring cache hit ratio, latency, link utilization and path stretch. Similarly to the case of caching strategies, further data collectors can be implemented without any change to the rest of the code using a dedicated plug-in registration system.

  Once an experiment is terminated, the engine queries the data collector for a summary of results. The latter gathers results from the various collectors and aggregates them in a dictionary, where each key is the identifier of the collector which gathered the data. Results are then passed back to the engine which passes them to the results collection and analysis subsystem. Finally, the analysis subsystem aggregates results from various experiments.

It should be noticed that the experiment execution subsystem communicates with the experiment orchestration and results collection subsystem only via the `Engine` component. This is an implementation of the façade design pattern, where the `Engine` acts as an interface to other subsystems. This design decouples all other components of the experiment execution subsystem from the rest of the code.

3.5 Results collection and analysis

The main functionalities of the results collection and analysis subsystem are to collect data from the various experiments, aggregate them and then process them. The aggregated results are stored using a list of pairs. In each item of the list, the first element is a dictionary storing all the parameters of the experiment and the second element is another dictionary storing the results of the experiment.

The results set can be serialised and saved in a file for further processing. *Icarus* supports serialization into a Python pickle file. However, the format of the code handling serialization is loosely coupled to the rest of the simulator. Therefore, serialization in different formats, such as JSON or CSV files could be easily implemented with localized changes.

Finally, this subsystem is responsible for analysing and plotting data. This subsystem automatically calculates the confidence interval of data sets, where relevant, and plots graphs with results of various metrics and various combinations of parameters. The automatic calculation of confidence intervals is in our opinion a fundamental feature contributing to the production of reliable results. This, in particular, is one of the objectives the motivated this work.

3.6 Caching and routing strategies

The caching strategy defines the replication algorithm used to spread copies of the content in the network of caches. Replication algorithms have two dimensions: i) content-based replication, which is based on the characteristics of the content and makes caching decisions according to that, e.g., choose popular contents to cache and ignore unpopular, and ii) cache- or node-based replication, which is based on the characteristics of the topology, e.g., choose more "important" nodes to cache contents. Both content-based and node-based replication are *resource optimization approaches*, as they both try to allocate caching space in a way that will return the most benefit (cache hits in this case).

The replication algorithms investigated in the context of ICN research are all cache- or node-based, since content-based algorithms can hardly operate at Internet scale. The replication spectrum in this case is bounded by two extremes, namely, the ubiquitous caching approach, where contents are replicated at every node the content is traversing (denoted as "Leave Copy Everywhere" below) and the "no caching" approach, where no node is implementing caches. The latter approach is similar to an end-to-end IP path, where nodes are buffering incoming contents and ones the contents are inserted in the link they are discarded from the memory.

Icarus supports the following strategies:

- **Leave Copy Everywhere (LCE) [19]**: According to this strategy, a copy of every content requested and delivered to the user is replicated at every router the content is traversing on its way to the user. Clearly, this strategy is causing caching redundancy, as different caches along the path are consuming cache resources to hold identical items. On the other hand, LCE is a good choice in case of flash-crowd events.

- **Leave Copy Down (LCD) [23]**: LCD was proposed for hierarchical Web-caching systems and builds on the following concept: whenever there is a cache hit for a particular content at some point along the path, the content is replicated one level down the cache hierarchy or path towards the user. This way, the content is

gradually replicated towards the edge of the network, given that the content is popular enough to receive a cache hit, before it is evicted from the cache.

- **Bernoulli random caching**: This simple strategy randomly caches a content item at each node it is traversing with a fixed probability p.
- **Random choice caching**: It caches a content item at only one randomly selected node along the delivery path.
- **Probabilistic Caching (ProbCache) [24]**: ProbCache has been recently proposed as a *resource management* approach to in-network caching. ProbCache is trying to reduce redundancy between network caches, that is, maximise the number of distinct content items cached along a delivery path. This way, subsequent requests for contents have higher probability of finding contents along the path.
- **Centrality-based caching [9]**: According to this strategy, content objects are cached only once along the path, specifically in the node with the greatest betweenness centrality (i.e. the node with the greatest number of shortest paths traversing it). If more than one node has the maximum value of betweenness centrality, then the content is stored in the node closer to the user. If this strategy is deployed on very dynamic networks (e.g. ad-hoc mobile networks) where it is challenging for a node to learn its betweenness centrality value, then caching decisions can be made based on the betweenness centrality of its ego-network, i.e. the subnetwork composed of all nodes directly connected to the caching node with an edge.
- **Hash-routing [28]**: While the above six caching strategies are focusing on on-path caching, that is, content items are replicated and cached on the delivery path from the server to the user, *hash-routing schemes* have been recently proposed as off-path caching strategies. According to these schemes, edge nodes receiving a content request compute a hash function mapping the content identifier to a specific caching node and forward the request to that specific node. If the cache holds the requested content, it is returned to the user, otherwise it is forwarded to the original source. Similarly, when a content is delivered to the requesting user, it can be cached only by the caching node associated to the content identifier by the hash function.

  The advantage of the hash-routing schemes is that if the content is to be found in a cache, this cache is indicated by the hash function. Therefore, the available cache space within the domain is allocated optimally (i.e., there is no caching redundancy). In [28] we investigate five different hash-routing schemes to identify the ideal behaviour in terms of both cache hits and path stretch. The *Icarus* simulator implements all of them.

3.7 Cache eviction policies

Caching nodes can be configured to operate according to one of the following well-known cache eviction policies:

- **Least Recently Used (LRU)**: It is the most well-known and most widely used replacement policy. When a new item needs to inserted into the cache, it evicts the least recently requested one. This eviction policy is efficient for line speed operations because both search and replacement tasks can be performed in constant time ($O(1)$). This policy has been shown to perform well in the presence of temporal locality in the request pattern. However, its performance drops under the Independent Reference Model (IRM) assumption (i.e. the probability that an item is requested is not dependent on previous requests).
- **Least Frequently Used (LFU)**: The LFU replacement policy keeps a counter associated each item. Such counters are increased when the associated item is requested. Upon insertion of a new item, the cache evicts the one which was requested the least times in the past, i.e. the one whose associated value has the smallest value. In contrast to LRU, LFU has been shown to perform optimally under IRM demands. However, its implementation is computationally expensive since it cannot be implemented in such a way that both search and replacement tasks can be executed in constant time. This makes it particularly unfit for large caches and line speed operations. For this reason, in the context of ICN, the LFU replacement policy is mainly used as a theoretical benchmark.
- **First In First Out (FIFO)**: According to the FIFO policy, when a new item is inserted, the evicted item is the first one inserted in the cache. The behavior of this policy differs from LRU only when an item already present in the cache is requested. In fact, while in LRU this item would be pushed to the top of the cache, in FIFO no movement is performed. The FIFO policy has a slightly simpler implementation in comparison to the LRU policy but yields worse performance.
- **Random (RAND)**: This caching policy randomly selects the item to evicts when a new one is inserted. It generally yields poor performance in terms of cache hits but is sometimes used as baseline and for this reason it has been implemented here.

Table 1 reports the computational complexity of the search and replacement tasks of the policies listed above with respect to their *Icarus* implementation. The parameter c reported in the table refers to the size of the cache. As it can be noticed from the table, while LRU, FIFO and RAND replacement policies can perform both operations in constant time, the complexity of LFU replacement operation grows linearly with the cache size c. As shown in section 5, this greater complexity of the LFU cache results in slower execution of experiments in comparison to other replacement policies, especially when large caches are used.

Policy	Search	Replacement
LRU	$O(1)$	$O(1)$
LFU	$O(1)$	$O(c)$
FIFO	$O(1)$	$O(1)$
RAND	$O(1)$	$O(1)$

Table 1: Average computational cost of search and replacement operations for various cache replacement policies

4. MODELLING TOOLS

In addition to all the functionalities required for simulating networks of caches, *Icarus* also provides a set of modelling tools useful for caching research.

4.1 Modelling of caching performance

Apart from the practical considerations about in-network caching (e.g., line-speed operation), the research community has also focused on the theoretical modelling of the system behaviour as a means to drive its design. Several recent studies have tried to model the behaviour of cache networks taking into account the characteristics of an ICN environment [5], [8], [25]. Given the rich literature on caching, the research community has also tried to evaluate whether past analytical models apply also to the case of in-network caching. Examples of such past studies that have attracted attention are [12] and [23]. *Icarus* includes implementations of some of these models, in order to cater for easy evaluation of their properties under different network conditions. We provide a brief summary of these models below.

4.1.1 Che's approximation

In [12], the authors propose a model that approximates the hit ratio of an LRU cache. This model works under the Independent Reference Model (IRM) condition. Assuming that the probability that a specific item $i \in (1, N)$ is requested is equal to p_i, then the cache hit ratio for the item i is approximately equal to:

$$h_i \approx 1 - e^{-p_i r_i} \tag{1}$$

where r_i is known as the *characteristic time* of the cache for item i and is calculated by solving the following equation numerically:

$$\sum_{i=1}^{N}(1 - e^{-p_i r_i}) = C. \tag{2}$$

The overall cache hit ratio h can then be calculated as the mean of all per-item hit ratios weighted by the probability of the associated item being requested, i.e.:

$$h = \sum_{i=1}^{N} p_i h_i \approx \sum_{i=1}^{N} p_i \left(1 - e^{-p_i r_i}\right) \tag{3}$$

The following snippet shows how to use this method to estimate the cache hit ratio of an LRU cache with 100 slots under stationary Zipf demand with coefficient $\alpha = 0.8$ and content catalogue of 1000 items.

```
>>> from icarus import *
>>> che_cache_hit_ratio(
          TruncatedZipfDist(alpha=0.8, n=1000).pdf,
          100)
0.36482948293429832
```

4.1.2 Laoutaris approximation

Icarus also provides an implementation of a method proposed by Laoutaris [22] to approximate the cache hit ratio of LRU caches under generalized power-law demand.

This method provides a closed-form approximation of the cache hit ratio by modifying some steps of the Che's approximation. More specifically, Laoutaris approximates the characteristic times associated to each item with their mean r. It then approximates the cache hit ratio by replacing the exponential form $e^{p_i r}$ with its Taylor expansion in terms of the variable r around point C (i.e. the cache size).

This approximation yields good results if the content popularity has reduced skewness (i.e. low Zipf α coefficient) or if the ratio between cache and catalogue size is small.

The following snippet shows how to use this method to estimate the cache hit ratio of an LRU cache with 100 slots under stationary Zipf demand with coefficient $\alpha = 0.8$ and content catalogue of 1000 items.

```
>>> from icarus import *
>>> laoutaris_cache_hit_ratio(0.8, 1000, 100)
0.35934820920359255
```

4.1.3 Numeric steady-state cache hit ratio

Finally, *Icarus* provides a function for estimating the steady-state cache hit ratio of a cache numerically.

Differently from the methods presented above, which can be used only with LRU caches, this method can be used with any cache implementation. In fact, it could be used with the cache replacement policies already provided (LFU, FIFO and RAND) or with a newly designed cache replacement policy.

This method is useful for two purposes. First, to evaluate the cache hit ratio of a new cache replacement policy. Second, as a benchmark for a new cache hit ratio approximation of known cache replacement policies.

This function calculates the steady-state cache hit ratio by calculating a moving average of the cache hit ratio and monitoring its standard deviation among adjacent windows. When the standard deviation falls below a predefined threshold, this is perceived as a signal that the cache is reaching steady state. At this point, the function returns the cache hit ratio computed over the last window.

The following snippet shows how to use this method to calculate the cache hit ratio of an LRU cache with 100 slots under stationary Zipf demand with coefficient $\alpha = 0.8$ and content catalogue of 1000 items.

```
>>> from icarus import *
>>> numeric_cache_hit_ratio(
          TruncatedZipfDist(alpha=0.8, n=1000).pdf,
          LruCache(100))
0.37861264056574684
```

4.2 Analysis of content request traces

4.2.1 Estimation of content popularity skewness

Icarus provides a function to verify whether a given content request trace follows a Zipf distribution and to estimate what value of the Zipf coefficient (α) fits the trace best.

The estimation of the Zipf coefficient is carried out using the Maximum Likelihood Estimation (MLE) method. Once the α parameter is estimated, a χ^2 test is executed between the expected distribution with the estimated coefficient and the data.

The following snippet shows how to use this function to estimate the α parameter from a known Zipf distribution. The result is a tuple whose first item is the estimated α coefficient and second item is the p value that the given distribution actually follows a Zipf distribution with the estimated coefficient.

```
>>> from icarus import *
>>> zipf_fit(TruncatedZipfDist(0.8, 1000).pdf)
(0.79999999999571758, 1.0)
```

4.2.2 *Trace parsers*

Finally, *Icarus* provides a set of functions allowing users to parse content traces from the most common datasets. These traces can then be used to feed a request generator and be used in simulations or can be analysed to identify specific patterns.

Icarus supports parsing from two data formats.

First, it supports the parsing of logs generated by a Squid proxy server. This is one of the most common open-source HTTP proxies available. This is also the format in which the traces provided by the IRCache project are made available.

Second, it supports parsing requests from the Wikibench dataset [29]. This is a dataset of requests received by all Wikimedia websites over a period of time between 2007 and 2008.

5. PERFORMANCE EVALUATION

In order to assess the scalability of the *Icarus* simulator and to validate its fitness for running large scale caching simulations, we evaluate its performance in terms of both CPU and memory utilization under varying conditions.

In particular, our analysis focuses on measuring the memory and processing footprint against various catalogue sizes and the speedup achieved when experiments are run in parallel on multiple cores.

5.1 Performance vs catalogue size

The first set of experiments run, whose results are reported in Fig. 3 and 4, have been carried out to evaluate the sensitivity of execution time and memory utilization against variations in sizes of caches and content catalogue. We report that we calculated the 95% confidence interval of those results using the replications method, but did not plot the related error bars because they were too small to be easily distinguishable from point markers.

These results have been gathered by running a set of simulations each consisting of 500 thousand requests, generated following a Poisson distribution. Content popularity has been modelled as a Zipf distribution with coefficient $\alpha = 0.7$. The simulation scenario comprised a binary tree network topology with depth equal to 5 in which the root node was assigned as content source, the leaves as content receivers and all intermediate nodes as caches. In summary, such a topology comprises 63 nodes of which one is a source, 30 are caches and 32 are receivers. Routing and caching decisions have been taken according to the LCE strategy. Cache space has been assigned uniformly to all caching nodes. The cumulative size of caches in the network was equal to 10% of the content catalogue.

Experiments have been run with all cache evictions currently supported by *Icarus* (LRU, LFU, FIFO and RAND) as well as with no caches (NULL). We evaluate performance in a range of content catalogue sizes from 10^3 to 10^7. It should be noted that since the ratio between cumulative cache size and content catalogue is fixed, a scenario with greater content catalogue also has greater cache sizes.

Fig. 3 shows the wall clock time elapsed between the beginning and the end of the experiment. As it can be noticed from the graph, the execution time increases proportionally

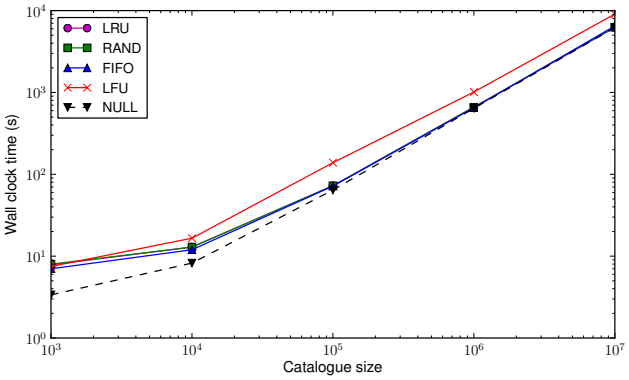

Figure 3: Wall clock time vs. content catalogue size

to the size of the content catalogue (and of the caches). This is caused mainly by the fact that in order to generate a random content identifier following a Zipf distribution, it is necessary to run a binary search in an array storing the cumulative density function of the content popularity distribution. Although this routine has been implemented using the highly-optimized NumPy's `searchsorted` routine, such operation has still a $O(log(n))$ complexity (with n being the size of the content catalogue). The computational cost of these search tasks becomes considerably greater than the computational cost of LRU, FIFO and RAND caches operations for increasing content catalogues. This is evidenced by the fact that the execution time without caches (NULL) approaches those of LRU, FIFO and RAND policies when the catalogue size increases. In addition, while LRU, RAND and FIFO replacement policies yield comparable performance for all sizes of content catalogues investigated, the performance of LFU policy degrades worse as the cache sizes increase. This is caused by the fact that while other replacement policies have a $O(1)$ cost for both search and replacement operations, LFU policy has a replacement cost of $O(n)$.

From the graph of Fig. 3, it is also interesting noticing the absolute time required to run these simulations, each consisting of 500 thousand requests. For example, in the case of a content catalogue of 10^5 items and LRU eviction policy the mean execution time is nearly equal to 60 seconds, i.e. one minute. These results, collected using dated hardware (CPU AMD Opteron 2347 with 1 GHz clock frequency), show that *Icarus* can actually scale very well and run realistic simulation scenarios with millions of requests in few minutes, even on modest hardware.

With respect to RAM utilization, whose results are depicted in Fig. 4, we also notice an increase proportionally to catalogue and cache size. This is caused by the memory occupied by the objects modelling caches and by the data structure which maps content identifiers to server locations. The latter is needed to route requests to a persistent content repository. In particular, in this case analysed, where overall cache space accounts for 10% of the content catalogue, this data structure is responsible for a much greater memory consumption than caches. In fact, the line plot of memory consumption in the case of no caches (NULL) almost overlaps those representing cases where caches are used. It should be noticed that the superlinear increase of memory utilization is caused by the widespread use of hashmaps (or dictionaries

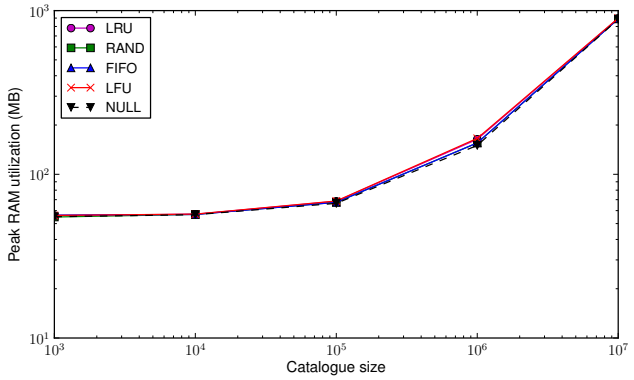

Figure 4: Peak RAM utilization vs. content catalogue size

in Python terminology) to implement caches and also the content-server map. In fact, Python dictionaries have been adopted because they yield $O(1)$ search cost but do so at the cost of a superlinear memory utilization.

Anyway, despite such superlinear increase in memory footprint, even in the largest scenario considered (i.e. a catalogue of 10 million contents and caches for 1 million contents), the total RAM utilization is still below 1 GB. As a result, on most commodity machines currently available, it would be possible to run parallel experiments on all CPU cores without exhausting all available RAM. This memory footprint could be reduced and traded off with CPU utilization by using a hash function to map content identifiers to source locations on-the-fly as opposed to using a precomputed hashmap. However, since in the current implementation performance is normally CPU-bound, such alternative implementation would further increase the load on CPU resulting in a performance drop. Therefore, it is reasonable to maintain the current implementation.

5.2 Parallel execution speedup

We mentioned in section 3.3 that in the design of *Icarus* we decided not to employ any PDES mechanism to parallelize the execution of experiments. Instead we assumed that, since in caching research experiments are generally repeated with different parameters, we speculated that the parallel execution of separate experiments would yield satisfactory speedup performance in normal usage conditions. We also showed in section 5.1 that even with very large caches and content catalogues, the RAM required to run a single experiment is small enough to effectively enable the execution of parallel simulations on the same machine.

In this section we validate these assumption by running a combination of heterogeneous experiments using different number of cores and analyse the performance achieved in terms of speedup. We calculate the speedup as the ratio between the average wall clock time elapsed between the start of the first experiment and the end of the last experiment using a single core and the time required by the same workload using multiple cores.

The set of simulations executed consisted in 1280 distinct experiments, which have been generated from a combination of four different real topologies with variable sizes (parsed from publicly available datasets), four cache sizes, ten content popularity distributions and eight caching and routing strategies. Experiments have been designed to be as heterogeneous as possible and by selecting variable topology sizes and cache sizes in order to ensure that they would have different execution times. All experiments have been configured to have a content population of 10^7 object in order to increase the load on the RAM.

These experiments have been executed on a server equipped with two Quad-Core AMD Opteron processors (i.e. 8 cores) and 32GB of RAM and repeated using a variable number of cores from 1 to 8.

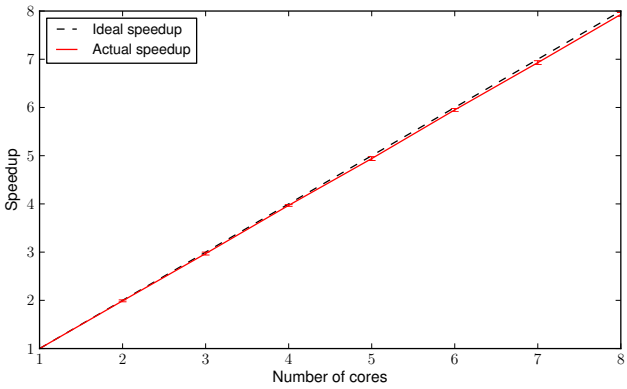

Figure 5: Parallel execution speedup

The results, depicted in figure 5 show an almost linear speedup for the entire range considered. Moreover, even in the case of eight simultaneous experiments running, performance has always been CPU-bound and never reached the level required to consume all the available RAM. In summary, this experiment validates the soundness of our design decision to implement parallel execution with a per-experiments granularity only.

6. SUMMARY AND CONCLUSIONS

In this paper we presented *Icarus*, a caching simulator specifically designed for Information-Centric Networking (ICN). *Icarus* has been designed to address all shortcomings of currently available simulators which made them unsuitable for simulating in-network caching systems.

With this objective in mind, *Icarus* has been designed to be *extensible* and *scalable*.

The extensibility objective is of fundamental importance to facilitate its fast adoption by the research community. We addressed this requirement by appropriately using design patterns to effectively decouple the various components of the system.

The scalability objective is also fundamental for ensuring that the simulator is capable of running large scale simulations (which are required for producing statistically significant caching evaluations) in a reasonable timeframe. We show in our evaluations that *Icarus* scales well and is able to handle large content catalogues and caches with modest RAM and CPU utilization.

In conclusion, *Icarus* provides a set of utilities for modelling the performance of cache replacement policies and to analyse content request traces from widely used datasets. To the best of our knowledge, such functionalities are not provided by any other publicly available software library.

Acknowledgments

The research leading to these results was funded by the EU-Japan initiative under European Commission FP7 grant agreement no. 608518 and NICT contract no. 167 (the GreenICN project) and by the UK Engineering and Physical Sciences Research Council (EPSRC) under grant no. EP/K019589/1 (COMIT).

7. REFERENCES

[1] Blackadder. `https://github.com/fp7-pursuit/blackadder`.

[2] CCNx. `http://www.ccnx.org/`.

[3] Content-Centric Networking Packet Level Simulator. `https://code.google.com/p/ccnpl-sim/`.

[4] A. Afanasyev, I. Moiseenko, and L. Zhang. ndnSIM: NDN simulator for NS-3. Technical Report NDN-0005, NDN, October 2012.

[5] J. Ardelius, B. Grönvall, L. Westberg, and A. Arvidsson. On the effects of caching in access aggregation networks. In *Proceedings of the second edition of the ICN workshop on Information-centric networking*, ICN '12, New York, NY, USA, 2012. ACM.

[6] S. Arianfar, P. Nikander, and J. Ott. On content-centric router design and implications. In *ReArch Workshop*, volume 9, page 5. ACM, 2010.

[7] L. Breslau and et al. Web caching and zipf-like distributions: Evidence and implications. In *In INFOCOM*, pages 126–134, 1999.

[8] G. Carofiglio, M. Gallo, L. Muscariello, and D. Perino. Modeling data transfer in content-centric networking. In *Proceedings of the 23rd International Teletraffic Congress*, ITC '11, pages 111–118. ITCP, 2011.

[9] W. K. Chai, D. He, I. Psaras, and G. Pavlou. Cache less for more in information-centric networks (extended version). *Computer Communications*, 36(7):758 – 770, 2013.

[10] W. K. Chai, N. Wang, I. Psaras, G. Pavlou, C. Wang, G. de Blas, F. Ramon-Salguero, L. Liang, S. Spirou, A. Beben, and E. Hadjioannou. Curling: Content-ubiquitous resolution and delivery infrastructure for next-generation services. *Communications Magazine, IEEE*, 49(3):112 –120, march 2011.

[11] A. Chankhunthod and et al. A hierarchical internet object cache. In *Proceedings of USENIX*, 1996.

[12] H. Che, Y. Tung, and Z. Wang. Hierarchical web caching systems: modeling, design and experimental results. *Selected Areas in Communications, IEEE Journal on*, 20(7):1305–1314, 2002.

[13] R. Chiocchetti, D. Rossi, and G. Rossini. ccnsim: an highly scalable ccn simulator. In *IEEE International Conference on Communications 2013 (ICC'13)'*, Budapest, Hungary, jun 2013.

[14] S. Farrell, E. Davies, and D. Kutscher. The netinf protocol. 2013.

[15] N. Fotiou, D. Trossen, and G. Polyzos. Illustrating a publish-subscribe internet architecture. *Telecommunication Systems*, 51(4):233–245, 2012.

[16] N. Fujita, Y. Ishikawa, A. Iwata, and R. Izmailov. Coarse-grain replica management strategies for dynamic replication of web contents. *Comput. Netw.*, 45(1), 2004.

[17] A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, Pasadena, CA USA, Aug. 2008.

[18] T. Henderson, S. Roy, S. Floyd, and G. Riley. ns-3 project goals. In *Proceeding from the 2006 workshop on ns-2: the IP network simulator*, page 13. ACM, 2006.

[19] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, CoNEXT '09, pages 1–12, New York, NY, USA, 2009. ACM.

[20] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001–.

[21] K. V. Katsaros, G. Xylomenos, and G. C. Polyzos. Globetraff: A traffic workload generator for the performance evaluation of future internet architectures. In *NTMS*, pages 1–5. IEEE, 2012.

[22] N. Laoutaris. A Closed-Form Method for LRU Replacement under Generalized Power-Law Demand. *ArXiv e-prints*, May 2007.

[23] N. Laoutaris, H. Che, and I. Stavrakakis. The lcd interconnection of lru caches and its analysis. *Performance Evaluation*, 63:609–634, 2006.

[24] I. Psaras, W. K. Chai, and G. Pavlou. Probabilistic in-network caching for information-centric networks. In *Proceedings of the second edition of the ICN workshop on Information-centric networking*, ICN '12, pages 55–60, New York, NY, USA, 2012. ACM.

[25] I. Psaras, R. G. Clegg, R. Landa, W. K. Chai, and G. Pavlou. Modelling and evaluation of CCN-caching trees. In *Proceedings of the 10th international IFIP TC 6 conference on Networking - Volume Part I*, NETWORKING'11, Berlin, Heidelberg, 2011. Springer-Verlag.

[26] D. Rossi and G. Rossini. On sizing CCN content stores by exploiting topological information. In *IEEE NOMEN Workshop*, 2012.

[27] L. Saino, C. Cocora, and G. Pavlou. A toolchain for simplifying network simulation setup. In *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques (SIMUTools '13)*, pages 82–91, 2013.

[28] L. Saino, I. Psaras, and G. Pavlou. Hash-routing schemes for information centric networking. In *3rd ACM SIGCOMM workshop on Information-Centric Networking (ICN'13)*, Hong Kong, China, Aug. 2013.

[29] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia workload analysis for decentralized hosting. *Elsevier Computer Networks*, 53(11):1830–1845, July 2009.

[30] A. Varga and R. Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems (SIMUTools'08)*, pages 1–10, 2008.