DiPIT: a Distributed Bloom-Filter based PIT Table for CCN Nodes

Wei You, Bertrand Mathieu, Patrick Truong, Jean-François Peltier
Orange Labs
Lannion, France
{wei.you, bertrand2.mathieu, patrick.truong, jeanfrancois.peltier}@orange.com

Gwendal Simon
Telecom Bretagne
Brest, France
gwendal.simon@telecom-bretagne.eu

***Abstract*—Content-Centric Network is a novel Internet design, which is motivated by the shifting of the Internet usage from browsing to content dissemination. This new Internet architecture proposal has attracted many research works, but one of the most critical components of a CCN node, the Pending Interest Table (PIT), did not get much attention. The PIT is involved in the forwarding processes in both upstream (reception of Interest messages) and downstream (reception of Data messages) ways. On the one hand, the PIT should be large enough to store a high volume of information. On the other hand, the PIT should be quick enough to not become a bottleneck in message processing. In this paper, we propose a novel implementation of PIT, named DiPIT. The idea is to deploy a small-size fast memory on every interface. Our approach relies on Bloom Filters in order to reduce the necessary memory space for implementing the PIT, completed with a central Bloom Filter for limiting the false positives generated by the individual Bloom Filters. Our evaluations highlight that our DiPIT approach can significantly reduce the memory space (up to 63%) in the CCN node and support a higher incoming packet throughput, than the hash table technology, which is largely implemented in current routers.**

I. Introduction

The Content-Centric Networking (CCN) proposal implies a major shift in the design and the implementation of network routers (a.k.a. CCN *nodes* in the CCN *jargon*). Some researchers have recently argued that such a shift is currently impossible because today's router technologies do not meet the requirement of CCN nodes [6, 16]. This analysis however assumed the same router architecture as the one described in the seminal CCN paper [10]. Our research aims at developing a new router implementation, which preserves the original features of CCN but can realistically be built with current hardware and software technologies. In this paper in particular, we propose a new implementation for one of the main components of CCN nodes, the *Pending Interest Table* (PIT).

The PIT is a central component in CCN node because it is involved in every message processing. The role of a PIT is to store the request (a.k.a. *Interest*) packets that passed through a CCN node until these Interest are "fulfilled" by the reception of matching response (a.k.a. *Data*) packets. For every reception of Interest packets, the PIT should create or update an entry, which contains the Interest names and the incoming router interface (a.k.a. *face*). For every Data packet, the router uses the PIT to find out the outgoing faces, and then the entries are deleted.

The implementation of PIT cannot accommodate to current memory technologies. On the one hand, a PIT needs a large memory space to store the pending Interests. Indeed, PIT cannot use aggregation because Data and Interests should exactly match. Consider an average Interest arriving rate of 125 millions packets per second (hereafter noted *Mpcks*) and a packet round-trip time of 80 ms [16]. A PIT should store a number of Interets in the order of 10^7. In addition, the content name structure is more complex than the IP address [7], therefore large memories should be used, with their known limitations, especially long access time. On the other hand, contrary to traditional IP FIB tables, a PIT table in a CCN node is highly dynamic. For every incoming Interest packet (unless the requested content has been already cached locally) and every matching Data packet, an operation should be carried out in the PIT table. Consequently operations should perform fast, which calls for fast memories that are unfortunately only available for small storage size.

To tackle this inextricable problem, we propose to cut the PIT table into several sub-tables, which we call *PITi*. We implement a PITi on each CCN node face so that a PITi should only store the list of pending Interest packets coming from the associated face. It thus becomes possible to use *Bloom filters* to store the Interests in a PITi. Benefits include fast lookups and small storage requirements. Our distributed PIT proposal (namely DiPIT for Distributed PIT) is independent from the naming structure, whatever the naming is hierarchical or flat, long or short.

In this paper we introduce some background about CCN network and current IP lookup methods in Section II and III. We detail our DiPIT proposal in Section IV. Finally in Section V, we validate our approach through several evaluations.

II. Content Centric Networking

We start with a brief overview of the CCN proposal.

The works about CCN (as well as other proposals related to information-centric approaches) are motivated by the observation that the Internet users now care more about *which* content or information they are interested in than about *where* the information are. However today's IP Network architecture relies on a host-to-host conversation model. The CCN proposal [10] is an attempt to replace today's host-to-host design by a new client-to-content model.

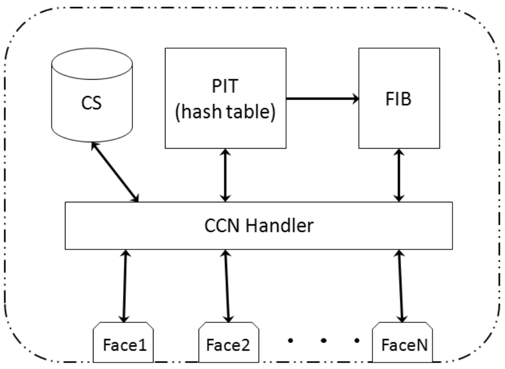


Fig. 1. Original CCN node

A. Named Data Networking

In CCN, each networking element is no longer identified by its location, but by a content name. CCN uses an hierarchical, structural, human-readable, unbounded name (for example */orange.fr/news/video.swf/v1/s0/*). Each networking process (routing, forwarding, content discovery and data retrieving) is also based on content name. CCN contains two types of packet: *Interest* for sending the requests and *Data* for replying the matching content. Each Interest packet carries a *ContentName* (CCN name), which expresses what the client wants. Each Data packet carries also a *ContentName*, which is used to describe the content in this Data and to match the Interest.

B. Architecture of a CCN node

Fig 1 illustrates the architecture of a CCN node, and also the processing when Interest and Data come. One CCN node is composed by three elements:

- *FIB* – Forwarding Information Base is used for forwarding the Interests to the sources that are known to potentially hold the matching Data. It is almost identical as the IP forwarding table.
- *PIT* – Pending Interest Table is for keeping the tracks of propagated Interest so that the returned Data can follow these tracks downstream to the consumers. The second role of the PIT is to prevent that multiple incoming Interest packets generate multiple packet forwarding. When Interests for the same content are received, only the first one is forwarded, the others will be only pushed in PIT and waiting for the Data back.
- *CS* – Content Store is a cache or a buffer set in CCN nodes. The in-network caching system of CCN leverages on the next-generation of routers, which incorporate some caching memory for the goal of minimizing network bandwidth and latency demands, as well as the server occupancy.

C. Message Processing in CCN

When an Interest packet arrives at a CCN face, the ContentName carried by this Interest is firstly checked in the Content Store. If there is a matching Data, it will be directly returned through the same face where the Interest arrived at. Otherwise the ContentName is further checked in PIT. If there is already a matching entry, the arrival face information is updated in the matching entry. If not, the FIB table is consulted for the forwarding information, and a new entry associated with the new arrival Interest is create in PIT.

The Data packet follows the "*bread crumb*" left by the Interest in the CCN nodes. When a Data packet arrives at a CCN node, the ContentName of the Data is checked in the Content Store. If there is already a matching content cached in the CS, this Data packet should be discarded. If there is no existing match in CS, the Data is checked in PIT. A PIT entry match means this Data has been required, it is sent out through the list of faces associated with the matching PIT entry, and the Data can be (optionally) stored in the Content Store.

III. Background and Related Works

There are mainly four IP lookup techniques: TCAM [18], radix tree, hash table [20] and Bloom-filter/hash-table [5]. As discussed in [16], none of them is acceptable for a CCN node. TCAM is a hardware solution with bounded lookup performance at one single memory access, however TCAM is expensive and the memory space capacity is limited. Radix tree is not suitable for the CCN naming structure. Hash table can perform fast lookup operations but, since CCN PIT is based on exact matching, every entry should be memorized, which leads to large memory requirements. Finally, Bloom-filter/hash-table is based on the implementation of several Bloom filters on fast memory chips and several hash tables for prefix storage on a slow memory. This approach suffers from the same drawback as the previous approach: the aggregation is still needed and the memory requirement for the hash table is too big for fast memory chip.

Up to now, previous work related to content-centric routers have mainly focused on the performances of Content Store [2, 4, 15] and on memory architecture for efficient FIB multicast forwarding processes [9]. Indeed, researchers are familiar with those domains of activities, since the Content Store component is functionally close to the existing caches or the CDN nodes, and the CCN FIB is similar to the IP FIB table. But the design and implementation of the PIT table have not get much attention so far since the PIT is a completely new component, not present in IP or in others information-centric networking proposals [14, 22]. However, this component is at least as important as the Content Store and more demanding than the FIB table because of the frequent updates in the PIT, contrarily to the FIB. This is the reason why we precisely focus our research work on this neglected component that plays a crucial role in the performance of the CCN node.

The implementation of Bloom filters for IP lookup in routers has been first proposed in [5]. The fast Bloom filters aim at reducing the number of accesses on the slow hash table. However this method currently is not employed in IP FIB because the Bloom filter can destroy the hierarchical structure of IP address. It is not an efficient lookup solution for aggregated table. But CCN PIT table performs exact match, thus even

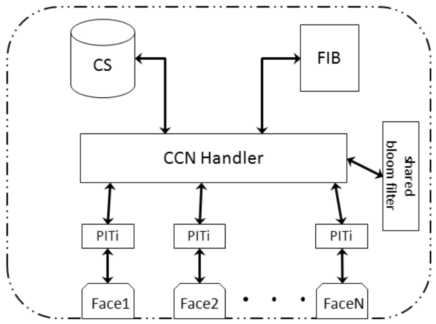


Fig. 2. DiPIT CCN node

if a CCN name is also structured, it does not matter that we break the hierarchical structure. The BUFFALO system [21] distributes IP forwarding tables on each router interface. Since this system does not tolerate too many entries, it targets especially small-scale networks (typically enterprises) and data-center networks. More recent papers dealing with Bloom filters for IP lookup or routing (for example [11, 12, 19]) assume a relatively stable set of elements, which is different form the dynamic behaviours of the PIT.

IV. DiPIT – Distributed Bloom filter based PIT architecture

We aim to implement a fast, space-efficient and cost-friendly PIT tables, with regard to the role of PIT defined in the CCN proposal. The two most popular table implementations that enable lookup in a time that does not depend on the size of the table (in other words, $\mathcal{O}(1)$-time lookup) are hash tables and Bloom filters. For DiPIT, we chose Bloom filters because Bloom filters are faster than hash tables for lookup and update operations. Bloom filters can also be more efficient in memory space, but in this case the ratio of "*false positive*" increases. In this paper, we explore the opportunity to leverage the appealing characteristics of Bloom filters for the implementation of PIT table.

The traditional design of Bloom filters does not match the requirements of PIT table. A Bloom filter only remembers the *footprint* of each element. Once an element has been inserted in the filter, it is impossible to retrieve any other information again. On the contrary, a PIT, as it has been conceived in CCN, needs to memorize not only the Interest names but also the incoming faces information. We tackle this mismatch by implementing a distributed PIT table, which is named DiPIT.

The original CCN node architecture (Fig. 1) implements a centralized PIT table. Our DiPIT proposal constructs one PITi (a small PIT table) on each CCN face (Fig. 2). Each PITi is constructed by Bloom filters. Our system also includes one additional Bloom filter which is shared by all faces. We present the principles of this proposal in the following.

A. PITi: One Bloom filter per CCN face

Each PITi works independently and records in a *counting* Bloom filter the footprints of the Interests packets that come from the associated face. Incoming Data packets are checked in parallel on all PITis and forwarded on the faces when the associated PITi has a matching footprint. Thus our DiPIT keeps the advantages of Bloom filters in terms of memory space and process time, and our design of one PITi per face tackles the aforementioned face information issue.

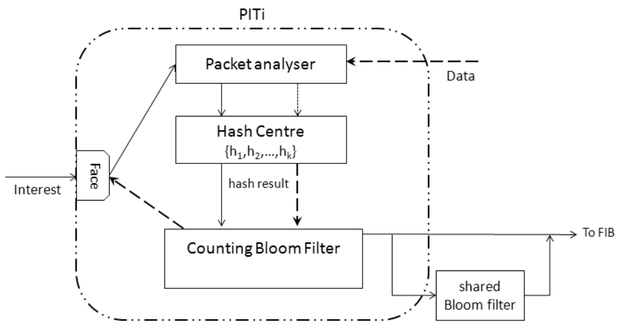


Fig. 3. Internal PITi architecture

We use a counting Bloom filter to deal with information removal. Indeed, a binary Bloom filter does not support the removal of an element because a bit position in the vector can be set to one by more than one element. In CCN, when a Data packet is forwarded, the related Interest entry should be deleted from the PIT table. Therefore, entries are frequently added *and removed* in a PIT. The counting Bloom filter addresses information removal by replacing the binary filter of Bloom Filters by a set of counters. Each position of the filter (each hashed position) is an integer that should be increased by one when another Interest comes. When a Data packet is forwarded through a face, the counters associated with the according Interest should be decreased by one. The implementation of this counting Bloom filter requires some extra memory, however it is still less than the memory space required by a hash table. We prove this by a series of evaluations in Section V.

The expired Interests should be removed from the PIT table. The hash-based PIT applies one timer for each PIT entry to delete the timed-out Interests. Similarly, in DiPIT we introduce one timer that manages the removal of expired Interests. Every PITi counters will be decreased by one (if not already at 0) at periodic interval. This interval is configurable and named DiPIT_TTL. This value should be large enough, *i.e.* at least twice the average response time, representing the time to wait for the reception of the matched Data. If the Data is received before DiPIT_TTL, the counter will be "normally" decreased because the Interest has received the corresponding Data. If no Data is received after the DiPIT_TTL period, the counter will be decreased meaning the removal of the expired Interest.

B. A shared Bloom filter to deal with false positive

A Bloom filter can introduce false positive since each filter position can be incremented by different element insertion. The false positive in our case is that the ContentName verification gives a matching result but in reality no Interest for this ContentName has been received on this face. We identify two side effects here.

- If an incoming Data packet mistakenly generates a match on the PITi of one face, this Data packet is forwarded through this face, although no Interest matches this Data packet. This false positive produce some useless networking bandwidth waste. It will also delete the "fake" matching entry when the Data goes out. Thus when the actual Data comes, it might be dropped due to the missing Interest. However these two problems are not critical. First of all, the extra emission will affect nothing but some networking resource consumptions. Secondly the probability that a series of consecutive linked routers generate a false positive on the same Data packet is low, so the extra networking load can be limited on a few hops. For the mistaken deletion of a matching Interest, the CCN designs incorporates regular Interest re-emission from the client side, so this event can only add some extra-latencies to the request.
- If an incoming Interest packet mistakenly generates a match on the PITi of one face, this Interest packet is considered as a duplicate thus it is not forwarded. The impact of false positives is critical because this Interest packet is lost.

We address the issue related to this latter side effect (an Interest is not forwarded because it mistakenly generates a match) by implementing a *shared binary Bloom filter* in the control panel. If two Bloom filters work independently, the total false positive ratio is the product of the two individual false positive ratios of the two filters. By adding another filter after the first Interest verification process, we can significantly reduce the ratio of false positive. Every matching for an Interest packet raises another verification on the shared Bloom filter. If the second verification result is negative, this Interest will be forwarded downstream and added into the shared filter for a further check. If the result is positive, we can consider that this Interest is indeed a duplicate.

It is important to understand that our system based on two serialized Bloom filters differs from the implementation of a bigger Bloom Filter on each face. We emphasize three reasons:

- Memory space. The shared Bloom filter is used by all PITis, although extending each PITi in order to reduce the ratio of false positive would make the size of this "augmented PITi" become too big for some memory technologies.
- Performance. The Interest packets, which only need one verification at the PITi, will take a longer lookup time because a bigger Bloom filter has more hash functions than PITi.
- Relevance with regard to the internal process. A ContentName needs a second verification only if it generates a match in one PITi. The number of potential ContentNames to be tested for the second filter is smaller than the number of different ContentNames that arrive at the node. Only a fraction of all the received Interests requires two verifications.

For the same reason that we implemented a counting Bloom filter for each PITi in order to support the deleting, we also need a mechanism to limit the false positive of the shared Bloom filter. Since the returned Data packets do not remove entries in the shared Bloom filter, we suggest to refresh the shared Bloom filter sometimes. We call it a RST mechanism. A RST notification is emitted from the control center, typically when the number of inserted elements reaches a threshold or on a regular basis. This RST mechanism does not damage the overall system behaviour, since the Data packet is forwarded only according to the "*bread crumbs*" left by the pending Interests in each PITi. The shared filter is not involved in Data delivering. In our system, the repeated Interest is dropped only when both of the two level filters give a positive verification, thus the only side effect of the RST mechanism is to forward one more time the duplicated Interest and waste some networking bandwidth.

C. Main DiPIT algorithms

We present the pseudo-code of DiPIT algorithms at the reception of both Interest and Data packets in Algorithm 1 and Algorithm 2, respectively.

a) Incoming Interest: An Interest that arrives on a given face is firstly checked in Content Store. If the Content Store does not have any matching Data, the Interest is then checked in the PITi associated with this face. We have several cases: (1) a negative result means that the Interest never came in. It is forwarded to the FIB and its footprint is added in the filter; (2) a positive result means that either the Interest has already come, or it is a false positive. It is then checked in the secondary shared Bloom filter; $(2a)$ if the second filter gives a negative answer, the Interest is forwarded to the FIB and its footprint is added in the second filter; $(2b)$ on the contrary, a positive result means that this Interest is a duplicated emission, the CCN node blocks it.

b) Incoming Data: The Data forwarding process is relatively simple. An incoming Data packet is verified in all the PITis, except the one where the Data comes. If a PITi contains a matching Interest footprint, the Data packet is forwarded through this face and the footprint should be deleted from the PITi.

We do not describe a pseudocode of the RST mechanism because there is no unique way to implement RST. We have designed several implementations of RST, which can be roughly distinguished into two families: (i) the RST is triggered by an accumulated number of bit set to one, and (ii) the RST runs on a regular basis. Due to lack of space, we do not report in this paper the evaluations we made, according to the CCN nodes, the size of the Bloom filters and the number of packets to treat.

V. Evaluation

We performed several evaluations in order to validate that our DiPIT system can significantly reduce the required memory space for CCN PIT table.

Algorithm 1 Treatment of Interest packet in DiPIT

```
Input:
    interest: incoming Interest packet on face $i$
    $CBF_i$: the counting Bloom filter associated with face $i$
    $BF_S$: secondary shared Bloom filter
Main program:
 1: if interest matched in Content Store then
 2:     return  matched Data
 3: else
 4:     if $CBF_i$ matching test on (interest) == false then
 5:         transfer interest to forwarding module
 6:         increase the counters of $CBF_i$ at the footprint positions
 7:         exit
 8:     else if $BF_S$ matching test on (interest) == false then
 9:         transfer interest to forwarding module
10:         add the footprint of interest in $BF_S$
11:         exit
12:     else
13:         block interest
14:     end if
15: end if
```

Algorithm 2 Treatment of Data packet in DiPIT

```
Input:
    data: incoming Data packet
    $i$: face id
    $Face[\,]$: set of faces
    $CBF_i$: the counting Bloom filter associated with face $i$
Main program:
 1: if (data) matched in Content Store then
 2:     discard data
 3:     return
 4: else
 5:     cache data in CS (optional)
 6:     for all $i \in Face[\,]$ do
 7:         if data matches $CBF_i$ then
 8:             send data through face $i$
 9:             decrease the counters of $CBF_i$ at the footprint positions
10:         end if
11:     end for
12: end if
```

A. Settings

We assume one networking line card holding 16 interfaces. The average Interest arrival rate ranges from 20 Mpcks to 200 Mpcks. Such a range is representative to several classes of routers. We set the Data packet RTT time as 80 ms [16]. We used five hash functions for the counting Bloom filters, Each counter has 3bits. The shared Bloom filter has a size of 1 Mbits. In most previous works, the acceptable false positive probability is comprised between 0.01% and 10% [3, 8, 17, 21]. The applications of today's IP network tend to tolerate a packet loss rate of 1% at least, we thus set our acceptable false positive in the range from 0.1% to 1%. Since the exact match is required in PIT table, the content identifier size is not important in our case. In our evaluations, the largest Interest arrival rate is 200 Mpcks and the RTT time of Data packet is 80 ms. According to *Little's Law*, the biggest number of elements in the PIT table is $(200*10^6)*(80*10^{-3}) = 16*10^6$, which can be represented by 2^{28}. Thus for the centralized hash table we used H-bit of 28 bits. We also add 32 bits H-bit in the evaluations, because 32 bits is a common value in hash functions. As in [16], we assume 40 Bytes Interest packet. The CCN name lengths are variable, so we take the middle value and give 128 bits for each ContentName matching in hash table. For each hash table entry we also had to add 2 Bytes to memorize the incoming interface identifiers.

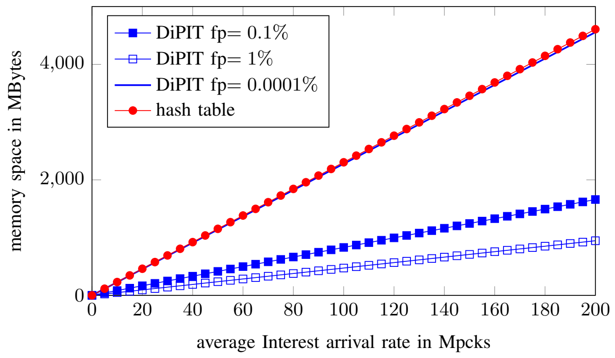


Fig. 4. Required memory size vs. λ_{in}

B. Required memory size

First we evaluated the required memory space according to the average Interest arrival rate (λ_{in}). In Fig. 4 (as well as in Fig. 5, which is a zoom on the lowest packet arrival rates), we represent the entire required memory space in the condition of different acceptable false positive ratios of the DiPIT system. In this figure the number of hash functions is fixed as 5. Lines with squared marks represent DiPIT for two false positives values (0.1 and 1%). We also represent the size of the hash table for both in lines with circled marks. Our system based on Bloom filter clearly outperforms an equivalent system based on a hash table. Typically for $\lambda_{in} = 100$ Mpcks, the required memory space for DiPIT with 0.1% of false positive is only 36% of the space required by the hash table.

Please note the represented sizes for DiPIT are for the *whole* DiPIT system. The required size for an individual PITi is only 51 MBytes for $\lambda_{in} = 100$ Mpcks. Therefore a PITi can be built on a fast memory. On the contrary, hash tables require an implementation on a large capacity memory, which has a longer access time. On the same space as the hash table, and for the same packet arrival rate, DiPIT would have a false positive ratio of nearly 0.0001% (plain line covered behind the hash table line).

In Fig 6, we studied more precisely the required memory space according to various false positive ratios. We fixed the

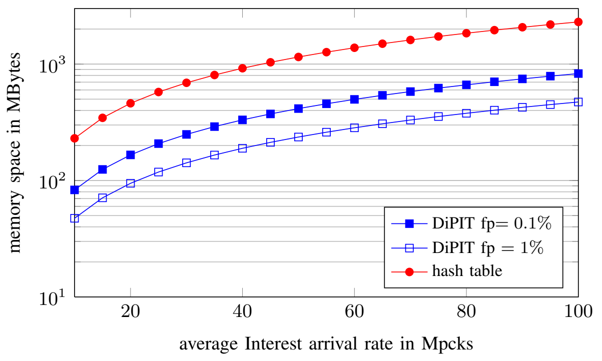


Fig. 5. Required memory size vs. λ_{in}

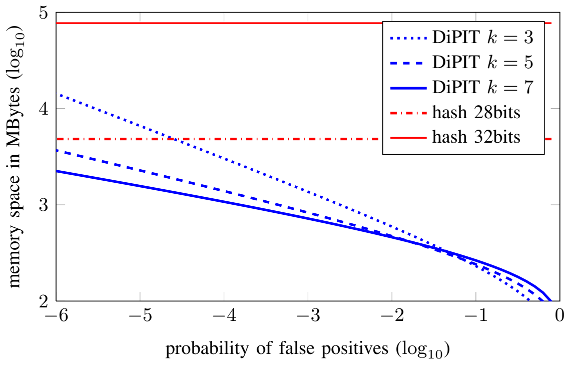
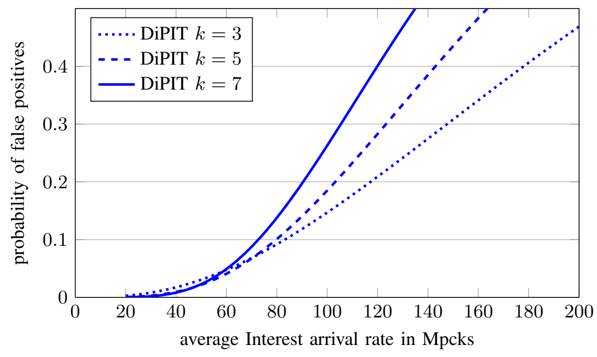
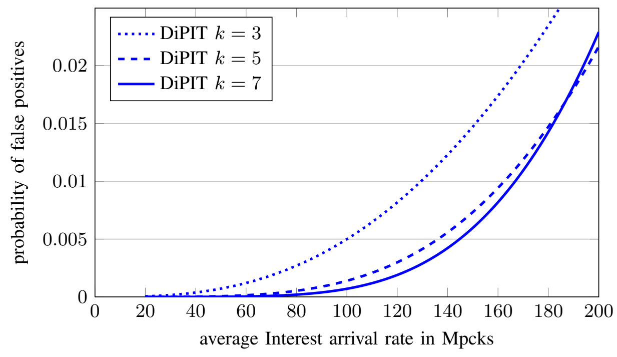


Fig. 6. Required memory size vs. false positive



(a) memory size: 32 Mbytes



(b) memory size: 128 Mbytes

Fig. 7. Probability of false positive vs. λ_{in}

Interest arrival rate at 100 Mpcks (Data response time is still 80 ms). The 28-bits hash table needs 4.8 GBytes. A 32-bits hash table can reduce the collision ratio, but it requires 77 GBytes. In order to limit the false positive probability at 0.1% (corresponding to −3 in the x-axis), DiPIT needs around 1 GBytes if the Bloom filters use three hash functions, but only 786 MBytes and 690 MBytes are necessary for five and seven hash functions, respectively. Thus we can also summarize that a router which does not deal with a high packet arrival rate can be implemented with more hash functions for saving the memory space. On the contrary, high level routers (*e.g.*, the edge routers at the peering point or the core routers) with a higher packet arrival rates would consider using less hash functions and relatively more memory to keep the acceptable performance speed.

Implementing relatively more hash functions can reduce the false positive ratio. However things are not that simple. The proper number of hash functions should be carefully chosen with regard of the number of packets which the CCN node face should treat and the implemented memory space. We represent in Fig. 7 the probability to obtain false positives for two typical classes of memory, according to the average packet arrival rate. As can be shown, seven hash functions paradoxically lead to bad results on 32 MBytes memory size at high packet arrival rate (see Fig.7(a)).

C. Bursty and multi-path traffic

We now deal with more complex (and realistic) traffic. Firstly we consider that a given Interest is received by several CCN faces of the same node. Then, we deal with fluctuant traffic.

If an Interest comes several times from several different CCN faces (within a Data response time), DiPIT is not able to filter the duplications because the same Interests coming to different faces are memorized independently at different PITis. A hash table is not impacted since it has only one centralized table. To be fair, we take into account this redundancy and we compute the actual space requirement for hash tables and for DiPIT according to the different probabilities that the received Interests from different faces are actually the same (Fig. 8). For example, the probability of 1 means that the whole traffic carries only one same ConetentName. We show that only the 28 bits hash table can require less memory size than DiPIT and only when more than 80% of traffic is redundant. In reality, it is easy to imagine that the chance we have 80% of traffic about the same ContentName within one Data response time (which is in order of millisecond) is quite small.

We now assume that the incoming traffic follows a *Poisson distribution* [13]. The more bursty the traffic is, the more false positives could be generated. In the meantime, it also means more packet losses for the centralized hash table. In Fig 9, DiPIT and the hash table are both designed to handle a traffic of $\lambda_{in} = 100$ Mpcks. Each PITi has five hash functions. The

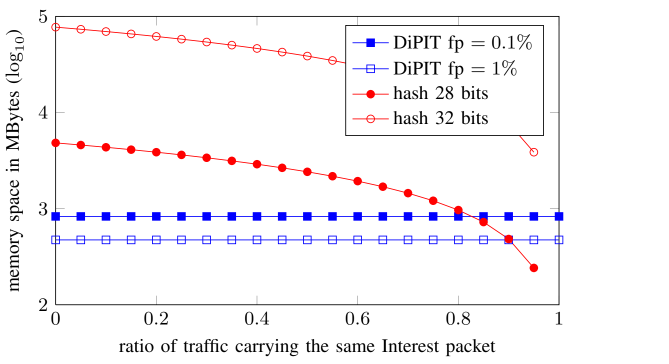


Fig. 8. Memory size vs. ratio of traffic related to only one Interest packet

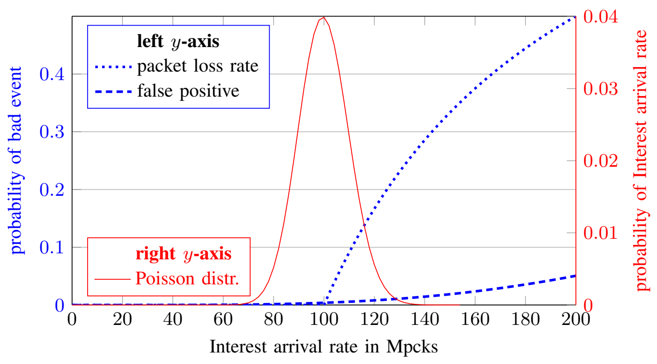


Fig. 9. The burst vs. packet arrival rate following Poisson law

left y-axis presents the probability of a bad event, which means either a false positive (in DiPIT) or a packet loss rate (in hash table). The right y-axis presents the probability of different packet arrival rates. We represent the Poisson law in red.

A well-dimensioned hash table experienced no packet loss for traffic smaller than 100 Mpcks, but the packet losses explode when the traffic is more intense. On the contrary, if we take a look at the curve slope, the false positive of DiPIT increases slowly. Typically, for 100 Mpcks $< \lambda_{in} <$ 120 Mpcks, which is a reasonable traffic burst, the probability of a bad event is still below 2% in DiPIT, although it reaches up to 15% for the hash table. Definitely, DiPIT is less sensitive to traffic burst.

VI. Conclusion and Future Work

Content-Centric Networking is a novel networking paradigm, which can bring many benefits for content delivery. If the basic principles are defined, improvements are necessary to reach a better performance and make CCN nodes more efficient. In this paper, we have proposed DiPIT, a distributed solution for the PIT component aiming at reducing the memory requirement in the CCN node. DiPIT is based on a PITi element, implemented on every CCN face with a counting Bloom filter, and a shared central element. Each PITi stores only the Interests coming on the associated face. The shared element enables to limit the false positive, inherent to the use of Bloom filters in the PITi. The evaluations show that with a small acceptable false positive ratio, DiPIT can significantly reduce the memory space for implementing the CCN PIT table. We are now implementing this solution on the latest CCNx [1] release in order to make real tests with a large set of nodes and be able to quantify the performance of our approach. Such large-scale tests will include the evaluations of the required table size, the processing time, the optimal false positive and the network load. With promising results, this approach might pave the way for building operational CCN routers, having different capabilities depending on their type and take into account the reality of Internet traffic.

Acknowledgment

This work is partially funded by the French National Research Agency (ANR), CONNECT project.

References

[1] Project CCNx. http://www.ccnx.org/.

[2] S. Arianfar, P. Nikander, and J. Ott. On content-centric router design and implications. In *ACM CoNext Workshop ReARCH*, 2010.

[3] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2004.

[4] G. Carofiglio, V. Gehlen, and D. Perino. Experimental evaluation of memory management in content-centric networking. In *Proc. of IEEE ICC*, 2011.

[5] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest prefix matching using bloom filters. In *ACM Sigcomm*, 2003.

[6] A. Ghodsi, T. Koponen, B. Raghavan, S. Shenker, A. Singla, and J. Wilcox. Information-centric networking: Seeing the forest for the trees. In *Proc. of HotNet'X*, 2011.

[7] A. Ghodsi, T. Koponen, J. Rajahalme, P. Sarolahti, and S. Shenker. Naming in content-oriented architectures. In *ACM Sigcomm Workshop ICN*, 2011.

[8] D. Guo, J. Wu, H. Chen, and X. Luo. Theory and network applications of dynamic bloom filters. In *IEEE INFOCOM*, 2006.

[9] H. Hwang, S. Ata, and M. Murata. Realization of name lookup table in routers towards content-centric networks. In *Proc. of CNSM*, 2011.

[10] V. Jacobson, D. Smetters, J. Thornton, M. Plass, N. Briggs, and R. Braynard. Networking named content. In *ACM CoNEXT*, 2009.

[11] Z. Jerzak and C. Fetzer. Bloom filter based routing for content-based publish/subscribe. In *ACM DEBS*, 2008.

[12] P. Jokela, A. Zahemszky, C. Esteve Rothenberg, S. Arianfar, and P. Nikander. Lipsin: line speed publish/subscribe inter-networking. In *ACM Sigcomm*, 2009.

[13] T. Karagiannis, M. Molle, M. Faloutsos, and A. Broido. A nonstationary poisson view of internet traffic. In *IEEE INFOCOM*, 2004.

[14] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A data-oriented (and beyond) network architecture. In *ACM Sigcomm*, 2007.

[15] L. Muscariello, G. Carofiglio, and M. Gallo. Bandwidth and storage sharing performance in information centric networking. In *Proc. of the ACM SIGCOMM workshop ICN*, 2011.

[16] D. Perino and M. Varvello. A reality check for content centric networking. In *ACM Sigcomm Workshop ICN*, 2011.

[17] Y. Qiao, T. Li, and S. Chen. One memory access bloom filters and their generalization. In *IEEE INFOCOM*, 2011.

[18] V. Ravikumar and R. Mahapatra. TCAM architecture for IP lookup using prefix properties. *IEEE Micro*, 24(2):60–69, 2004.

[19] H. Song, F. Hao, M. Kodialam, and T. Lakshman. Ipv6 lookups using distributed and load balanced bloom filters for 100gbps core router line cards. In *IEEE INFOCOM*, 2009.

[20] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed ip routing lookups. In *ACM Sigcomm*, 1997.

[21] M. Yu, A. Fabrikant, and J. Rexford. Buffalo: bloom filter forwarding architecture for large organizations. In *ACM CoNEXT*, 2009.

[22] A. Zahemszky, B. Gajic, C. E. Rothenberg, C. Reason, D. Trossen, D. Lagutin, J. Tuononen, and K. Katsaros. Experimentally-driven research in publish/subscribe information-centric inter-networking. In *Proc. of TridentCom*, 2010.