An Information Centric Network for Computing the Distribution of Computations

M. Sifalakis
Dept of Mathematics and Computer Science
University of Basel
sifalakis.manos@unibas.ch

B. Kohler
Dept of Mathematics and Computer Science
University of Basel
basil.kohler@unibas.ch

C. Scherb
Dept of Mathematics and Computer Science
University of Basel
christopher.scherb@unibas.ch

C. Tschudin
Dept of Mathematics and Computer Science
University of Basel
christian.tschudin@unibas.ch

ABSTRACT

Named Function Networking (NFN) extends classic Information Centric Networking (ICN): Beyond requests for named data, it supports the concept of function application as well as function definition. This empowers an NFN to choose internally the best places for fulfilling a potentially complex user expression: Forwarding optimization and routing policies all become a cause of dynamic decisions for (re)-distributing computations, and retrieving results.

In this paper we describe a systematic way to let such an extended information centric network compute where the actual computation on the data, if not avoidable through lookup of previously cached computation results, should be carried out. We present simulation experiments with our prototype (which is based on untyped Lambda expressions and Scala procedures) for demonstrating both the feasibility and the added benefits. We also highlight important insights, e.g. the need for explicit feedback, which might impact the design of classic ICN protocols that focus on named-data only.

Keywords

Computer networks, information centric networking, named data networking, named-function networking.

1. INTRODUCTION

In the recent years, and more than a decade after the seminal work in the TRIAD project [5][1], ICN research is at the forefront of attention of future Internet architecting. Surveys such as [1, 24] document and review numerous projects, some of which propose information-centric ramifications and extensions for the current Internet, while others are developing more *clean-slate* architectures and claim the central role of IP layer in the future Internet.

[1]possibly the precursor of most ICN architectures today

The focus of ICN research is on architecting away the shortcomings of the host-centricity in today's Internet, addressing aspects of node mobility, security, dynamics of content dissemination, but most important, factoring out location dependence from the interaction of the user with information. A common design foundations in many ICN architectures is the adoption of publish-subcribe semantics and the use of names to address content without involving host references. This contributes to a perception and use of the network as a data repository, a global database of some sort, or in its simplest form as a (semantic) memory, as is well characterized by the label "Named Data Networking" [11].

The architectural foundations and design "principles" of the early Internet aimed at and made very simple to link networks and interconnect resources. The success of these foundations enabled unprecedentent growth and innovations for services and applications on either side of the IP layer.

It is therefore worth pondering to what extend the key design foundations of current ICN architecting encompass analogous effectiveness when it comes to *interconnecting information*, and potential for application/service innovation. In fact, amidst the cloud computing era, only connecting users to information, seems a "halfway vision" for an information-centric Internet.

In [22] the authors have envisioned the other half of the way towards an information-centric Internet to close the gap between information access (ICN) and information processing (Cloud computing), enabling the capacity for the user to request from the network information interactions and manipulations alike. We presented the Named Function Networking (NFN) foundation/extension to ICN, which essentially generalises the semantics of names in ICN, such that they are treated as expressions. A name can thus interchangeably represent a mapping to an information object, a function capable of processing information objects, or an expression that combines the two (as well as other names). By composing names, users can express *what* result or transformed content they need, and the network gets in charge of finding *how* these results can be obtained, by interlacing expression-resolution with name-based forwarding.

Like in the case of removing locality-of-storage aspects from data names, NFN removes locality-of-execution, which allows the network to dynamically distribute computations and collect results: Data caching is now extended to also cover caching of results. In this way NFN addresses the more practical modern need for using (transforming) information as opposed to solely retrieving information.

In this paper we present the NFN concept in action. We report our first experiences of using NFN on a small testbed, demonstrate though a series of experiments the added value, and discuss our observations and challenges encountered.

The remaining of this paper is organised as follows. In the Sect.2 we provide an overview of the main characteristics and design tenets of NFN, present the node architecture and the combined expression resolution/forwarding strategy. In Sec.3 we directly start presenting a number of experiments, and dicussion of results and observations. In Sec.4 we discuss the observations, the issues we encountered and the differences from [22]. Finally in Sec.6 we conclude the paper and describe the plans for future work.

2. NFN (IN A NUTSHELL)

In its essence NFN blends program interpretation with network routing/forwarding, and thereby dynamically spreads computation tasks across an ICN network (and collects-synthesizes the results); orchestrating in this way the interaction of code with data on behalf of the user. This is effected in one of three ways of controlling code and data movement in the network as depicted in Fig. 1.

The first case depicts an attempt to locate results of computations that may have already taken place before, and so in case (a) a node handling a request avoids recomputing information which exists elsewhere in the net. Case (b) applies when some information needs be produced, either because it never was computed before or is not timely available. Case (c) covers a situation when some code or data is "pinned down" by either the owner or due to technical constraints. In this case the name resolution (execution) is delegated (pushed) to the pinning site.

Such programs are represented in their most simplest and most compact form as functional programs encoded in the λ-calculus. The λ-calculus is recursively defined

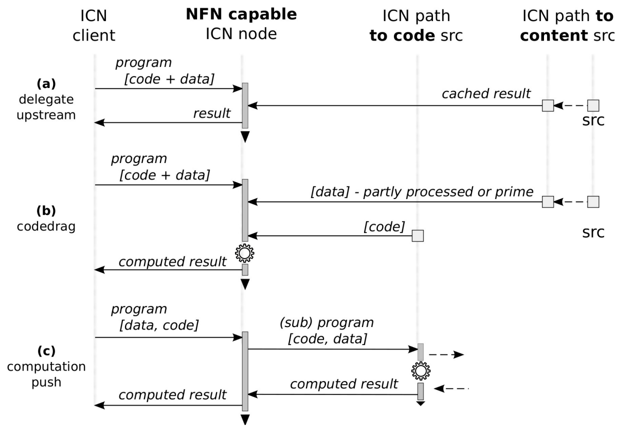

Figure 1: Three scenarios that NFN must handle: upstream fetch, separate code and data fetch, computation push

as three expressions: variable v, application fg and abstraction $\lambda x.e$.

The simplest λ-expression is a variable v, which in principal is just a name. A application `f g` means that a function `f` should be applied to some parameter `g`, where `f` and `g` are λ-expressions. An abstraction λ`x.e` stands for a function that consists of some λ-expression `e` where all occurrences of formal paremeter `x` in `e` are the places where the actual parameter value has to be substituted. Parentheses may be used to make expressions more readable but strictly speaking are not needed.

The λ-calculus can represent compactly program logic of arbitrary complexity (loops, conditions, etc) limited only by the maximum length of a name. In Sec.2.2 we describe a strategy of how the composition and decomposition of terms in these names exploits CCN/NDN forwarding to orchestrate the aforementioned cases of distributing computations. Then at Sec.4 we briefly describe how NFN could use other ICN naming schemes as well, and what are possible implications.

NFN assumes two levels of program execution: One is the functional untyped λ-calculus used for "name shuffling" and orchestrating the computation distribution. The other level relates to bytecodes or similar mobile code technology (JVM and a Scala [20] interpreter in our prototype) that serves data processing; once an execution site has been identified.

One way of regarding the use of λ-calculus is that of a simple IDL (interactive data language), involving only two operations: variable look-up, known as name resolution, and term reduction, where a function is applied to its arguments and composes new terms from them. Name resolution is the essential functionality of ICN. Term reduction is what the foundation of information interactions.

2.1 NFN Node Architecture

NFN "sits on top" of the CCN/NDN [12] architecture (hereafter refered to as CCN) by adding (*i*) a λ-*expression resolution engine* to a CCN relay, and (*ii*) optionally an application processing/execution environment. These two extensions correspond to the two levels of program execution mentioned earlier. The resolution engine is currently situated within the CCN relay, and all Interests that have the *implicit* postfix name component `/NFN` as processed by it. (This is by analogy to the way CCNx protocol currently handles name checksum hashes). The application processing environment is optional in the sence that there is no requirement for all NFN nodes to be capable of "number crunching" data processing operations. It is currently a Scala language[20] ComputeServer (practically a JVM), bound to a node-local Face. The CCN relay demuxes to it requests for data processing computations by the usual longest prefix matching against prefix `/COMPUTE`.

The resolution engine (see [22] for a detailed description) implements an *Abstract Machine* (Krivine's Lazy Machine [14]) that embodies call-by-name evaluation for λ-calculus expression. We have used the ZAM [15] instruction set (see Caml) to implement the required primitives, on a *Stack-Machine* with two stacks: One, for holding intermediate reduction state, and the other for resolving external calls to external data processing functions on content. *Call-by-name* resolution guarantees that this happens only if/when needed. Overall this leads to a very compact and lightweight implementation (see table below), and execution under controlled resource allocation of an NFN-extended CCN relay.

Primitive	λ-op	AM Instructions
`RBN(v)`	VAR	`ACCESS(v);TAILAPPLY`
`RBN(\x body)`	ABSTR	`GRAB(x);RBN(body)`
`RBN(f g)`	APPLY	`CLOSURE(RBN(f));RBN(g)`

`ACCESS(var)`	Lookup name *var* in environment E and push the corresponding closure to the argument stack A
`CLOSURE(code)`	Create a new closure using E and term *code*, push it to the argument stack A.
`GRAB(x)`	Replace E with a new environment which extends E with a binding between x and the closure found at the top of A.
`TAILAPPLY`	Pop a closure from the argument stack A and replace the current configuration's E and T with those found in the closure.

Table 1: Abstract Machine primitives

2.2 Distributing computations: Program translation and network forwarding

In NFN a name can hold an expression of the sort `func(data)`, which imperates the application of `func` on `data`. Such an expression can is encoded equivalently in any of the following λ-caclulus expressions[2].

```
1. func data
2. (λzy.z y) func data
3. (λy.func y) data
4. (λz.z data) func
```

The equivalence of these expressions and the manipulative capacity to convert from any of them to any other is the essence of NFN.

Lookup operations for functions or data are treated interchangeably by virtue of their names in the ICN network. As a result both `func` and `data` can be independently addressable CCN names-to-content. For example if

```
func:  /name/of/transcoder
data:  /name/of/media
```

then the application of transcoding on the media content can be represented, according to the 3rd form above, in the following *named expression.*

```
(λy./name/of/transcoder y) /name/of/media
```

NFN packages this named expression inside a CCN Interest as follows

i_n`[/name/of/media | (λy./name/of/transcoder y)]`

In this symbolism of a CCN Interest '`|`' delimits individual name components[3]). The term inversion in CCN's wire format has to do with its longest prefix-match forwarding as we will see shortly. In general the term placement in a name composition relates to how the ICN architecture implements its resolution process based on the name's components.

An alternative way that we will use hereafter to symbolise the same interest is

i_1`{ (λy./name/of/transcoder y) /name/of/media }`

The name component in boldface is the one that comes first (position 0) in the Interest packet encoding influencing by longest prefix match the forwarding.

To distribute computations in the network NFN currently implements the following strategy. Initially (phase 1) using Interest i_1 it searches for a cached copy of a transcoded version of media, en-path to `/name/of/media`. Having the component `/name/of/media` first, warrants that the Interest will travel in the direction of the data source. Representing "a transcoded version of media" as `/name/of/media/name/of/transcoder` is perfectly plausible also at the data source, even if that is a CCN-only node that has adopted this convention.

If this search does not yield results, then ideally one of the NFN nodes that has received the Interest en-path to the source of `/name/of/media`, should try to compute the result. This is feasible at any NFN node, by extracting the CCN names from the λ-expression and forking separate individual Interests for `/name/of/media` and

[2]the possible forms are not limited to these four only of course.

[3]We use this convenience notation recursively, when a name component is itself a CCNx name.

`/name/of/transcoder`. Each of those will hopefully retrieve the video data and the transcoder code respectively, enabling the node to compute and then cache the result ("code drag" case in Fig.1). At the end of this 2nd phase the result has been cached still en-path and possibly close to the data source, which would increase the re-use potential by others.

If any of the two Interests, does not yield the content back for some reason, and before giving up, there may still be hope by taking the computation off-path (phase 3). The NFN node can become a computation-proxy and push the computation towards `/name/of/transcoder` by simply transforming the named expression in i_1 to an equivalent form as in a new Interest i_2

i_2`{ (λz.z /name/of/media) /name/of/transcoder }`

This expression carries out the same computation, but has the component `/name/of/transcoder` in the first position which results in forwarding the Interest in the direction of the code. ("computation push" in Fig.1).

Due to the symmetric routing in CCN, if the computation succeeds on-the-way to `/name/of/transcoder`, the result will travel the same way back to the proxy point and satisfy the original Interest. The Interest for `/name/of/transcoder` may yield no results in phase 2 e.g. because the code does not exist, due to a "name pinning" policy for not distributing the code, etc.

Fig 2 summarizes in a flow diagram the 3 phases of the combined resolution-forwarding logic. What is important, is that: (a) Distribution of computation tasks in the network does not require forwarding state or cache state alterations – ephemeral content may appear in a cache as a by-product, which in absence of popularity will be eventually erased. (b) Computations may happen anywhere or nowhere, leaving the resource allocation decision entirely to the network.

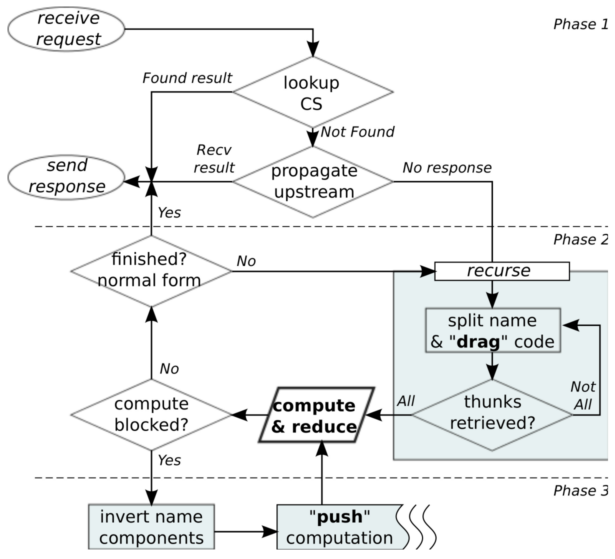

Figure 2: NFN forwarding strategy for CCN.

Requests for intermediate results in nested terms of the named expression can in fact retrieve only *thunks*. One can think of a thunk as a reference to computations, whose results can be retrieved later in time. Thunks allow the evaluation of a named expression to progress even when no results are available yet (enabling asynchronous and parallel computations). We demonstrate their use later on.

An Interest carrying a named expression has an additional implicit (not shown) postfix component i_n`[..|/NFN]` and when it represents a thunk it has 2 postfix components i_n`[..|/NFN|/TH]`. This allows their dispatching to the NFN resolution engine embedded in the CCN relay.

3. EXPERIENCES WITH NFN

In this section we report and discuss our first evaluation experiences with NFN. The goals of this evaluation has been to

- Demonstrate the ability of NFN for dynamic distribution of computation tasks and information-code interactions inside an ICN network (and show benefits of caching computations).
- Test, verify and understand different cases where NFN empowers network side decisions and optimisations, in face of network conditions.
- Have a first assessment of the performance overhead of NFN.
- Gain insights of how to improve the effectiveness of NFN.

3.1 Experimental set-up

We conducted our experiments on a small testbed with CCN and CCN-NFN nodes. The topology we have used in most of the experiments is shown in Fig.3. It consists of five nodes, where two are pure CCN nodes and the other three nodes contain an instance of our NFN extension of CCN as well as a Scala [20] Compute-Server (JVM); connections between nodes are bidirectional; client requests always arrive at `Node1` first. Any deviations from this set-up is reported in the individual sections that document the experiments.
The FIBs of the nodes are initialized manually (in absence of dynamic routing for CCN) and such that each node can reach every other node over the shortest path; when there are more than one path, both are included.

In regard to content distribution, we have placed a different document at every node's content store with name `/nodeX/doc`. Additionally, the content stores of `Node3` and `Node4` contain bytecode of a word counting procedure which is named `/bin/scala/wrdcnt`, corresponging FIB entries are in place at all other nodes. This procedure takes a single argument and computes the number of words of the argument. In our simulation

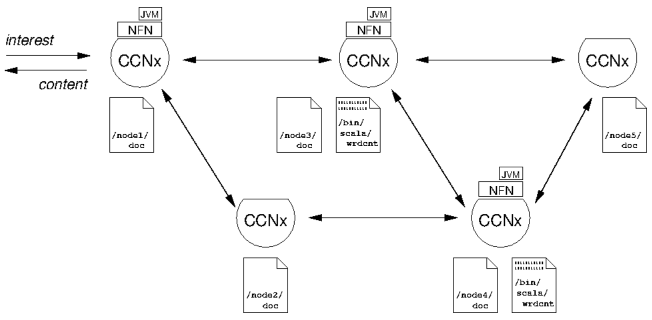

Figure 3: Testbed topology: 2 CCN & 3 NFN nodes.

it waits for 500 milliseconds to model a more compute-intensive function, before returning its result.

3.2 Six cases where the network is in charge of placing computations

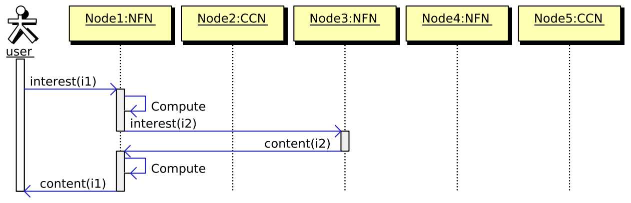

Figure 4: Experiment 1 – the network pulls code, applies it to locally available data

i_1{/bin/scala/wrdcnt (/node1/doc)}

i_2{/bin/scala/wrdcnt}

Experiment 1 (code+content pull): The first experiment is a vanilla check of the code fetching feature of NFN. It starts by a user requesting the word-counting of /node1/doc. In our default mapping of λ-expressions to CCNx messages, the innermost argument becomes the first name component, thus determines routings towards Node1. (/node1/doc1). Node1 starts resolving the expression: When the component /bin/scala/wrdcnt is encountered, it issues a normal interest (i_2) for it, retrieves from Node3 the bytecode of the procedure and applies it (in the JVM) to the locally available document. Finally, Node1 returns the word count result in a content object to the client.

Experiment 2 (computation push): The second experiment demonstrates computation push through the interaction of NFN with the CCNx forwarding fabric. The client issues a word-counting request as in experiment 1, but this time for /node5/doc, which is located on Node5, a CCN-only node. As neither of the missing names is available locally on Node1, and according to the strategy discussed in Sec.2.2, Node1 sets the name component /node5/doc5 at the first position and propagates the interest i_2 towards Node5. However, at Node5 the interest times out because it is a CCNx-only node,

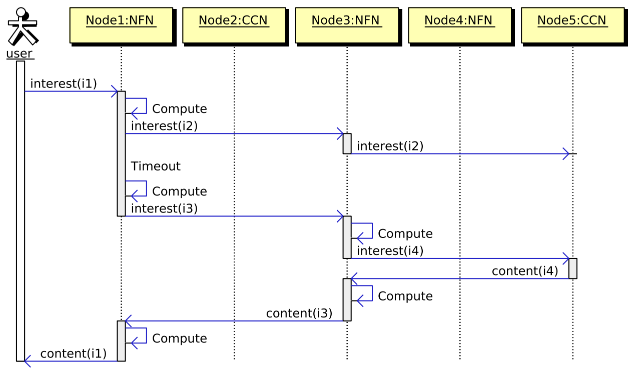

Figure 5: Experiment 2 – computation push (the network works around a CCNx-only node)

i_1{/bin/scala/wrdcnt (/node5/doc)}

i_2{/bin/scala/wrdcnt (/node5/doc)}

i_3{(λx.(x /node5/doc)) (/bin/scala/wrdcnt)}

i_4{/node5/doc}

the remaining name components cannot be matched exactly or based on longest prefix match.

Node1 then reverts to the next phase of the strategy from Sec.2.2 and transforms the expression to an equivalent one that has the component /bin/scala/wrdcnt at the first position in the name. This new interest i_3 is now forwarded by CCN to Node3. This node has a local copy of /bin/scala/wrdcnt, which means it can partially evaluate the expression and then separately request the content for /node5/doc in i_4 . This time, the name can be matched exactly: Node3 receives the content, computes the result and returns it to Node1, who in turn satisfies the client request.

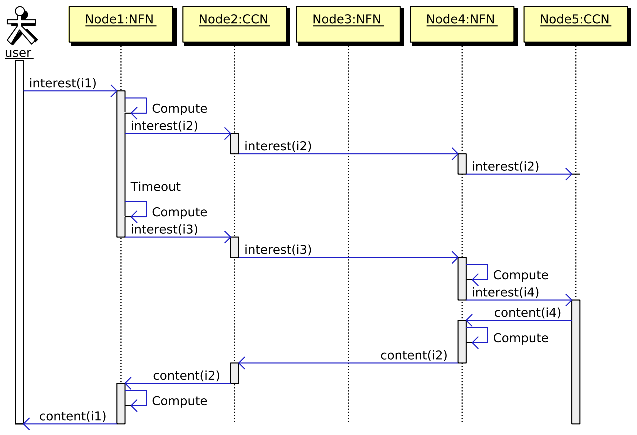

Figure 6: Experiment 3 – failover (the network discovers another suitable computing place)

i_1{/bin/scala/wrdcnt(/node5/doc)}

i_2{/bin/scala/wrdcnt(/node5/doc)}

i_3{(λx.(x /node5/doc)) (/bin/scala/wrdcnt)}

i_4{/node5/doc}

Experiment 3 (failover): Assume the same query as in experiment 2 and that the connection between `Node1` and `Node3` failed: How will the network use the alternate path that exists between `Node1` to `Node5`? Interest i_2 (which carries the complete name expression) now travels to `Node5` via `Node2` and `Node4`. As before, this request times out. Then the transformed interest i_3 is generated as before by `Node1`; it is sent to `Node2` which is a CCN-only node, and upon reaching `Node4`, the computation completes there! The important aspect here is that although there is an alternative path to node `Node3` who can compute the expression, rerouting does not deliver the computation there. Instead it takes place on node `Node4`, which was found opportunistically and happens to be better placed (in the new topology)!

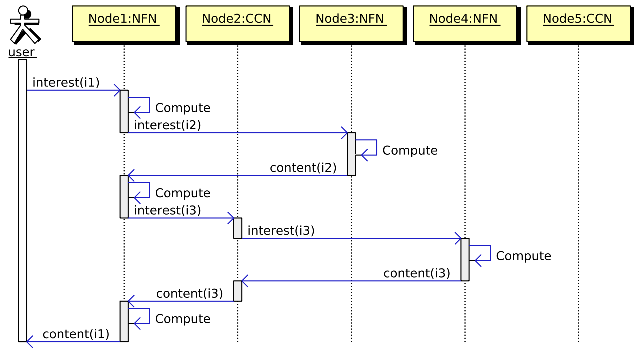

Figure 7: Experiment 4 – sequential evaluation of subexpressions

i_1`{(λf.(add(f(/node3/doc))(f(/node4/doc))) (/bin/scala/wrdcnt)}`

i_2`{/bin/scala/wrdcnt (/node3/doc)}`

i_3`{/bin/scala/wrdcnt (/node4/doc)}`

Experiment 4 (recursive distribution): In this experiment, a client sends a request for the sum of two different word-counts. As we will see, the word-counting of each document can take place independently and at different places. This is orchestrated by `Node1`, which performs two sequential reduction steps. Each resulting subexpression is separately dispatched: The first subexpression leads to interest i_2 and forwarded to `Node3` and computed there. The second subexpression yields interest i_3 which is computed on `Node4`. Both results are collected at `Node1` and the final result is returned to the client.

This experiment shows complex computations are recursively broken down to a workflow type of task with rather opportunistic coupling among them (compare this to the tight coupling expected by most SoA architectures today). Later on, in Sec.3.3, we show how through the use of thunks the same request is worked on in parallel, effectively providing an opportunistic Map-Reduce transport across an ICN network.

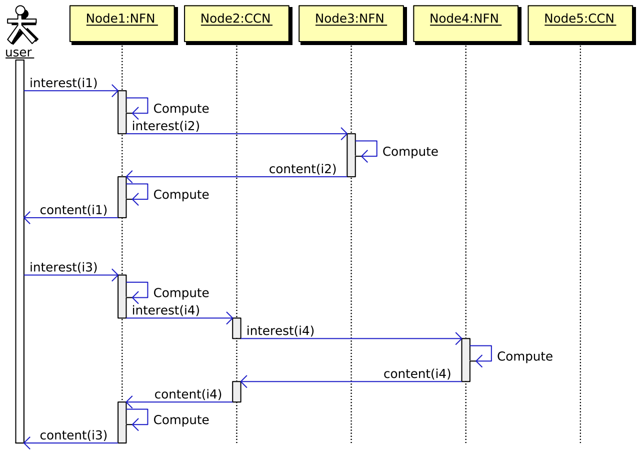

Figure 8: Experiment 5

i_1`{/bin/scala/wrdcnt(/node3/doc)}`

i_2`{/bin/scala/wrdcnt(/node3/doc)}`

i_3`{(λf.(add(f(/node3/doc))(f(/node4/doc))) (/bin/scala/wrdcnt)}`

i_4`{/bin/scala/wrdcnt (/node4/doc)}`

Experiment 5 (cached results prevent recomputation): This experiment extends the previous one to show that if the result of a (presumably popular) computation can be cached, this will reduce the computational cost or delivery time of subsequent similar computations.

We assume that before the task of summing up two word-counts hits the network, a single word-count request for the documents (`/node3/doc`) is received in interest i_1. `Node1` passes it to `Node3` (where both the data and code are available), which computes the word-count and returns its result; `Node1` will cache the resuls. The next request for summing up the two word-counts for `/node3/doc3` and `/node4/doc4` is received in i_3. This time, `Node1` already knows the result of the first computation regarding `/node3/doc`: hence it issues only one interest i_4 for fetching `/bin/scala/wrdcnt`. Then, the overall response is computed and returned much faster than in the previous example, as can be seen in Fig.12 (comparing the times of $Exp4$ and $Exp5_2$).

Experiment 6 (Node loaded, pass it to the next): It can happen that some node is overloaded with computations e.g. only because it happens to be closer to popular or voluminous requests (or because it becomes the target of some DoS attack). The opportunistic location-decoupled nature of computations in NFN note only makes it difficult to target (in the case of DoS) a specific node, but also enables implicit load-balancing in the ICN network.

In this experiment `Node3` is designated to be in *overloaded* state and additionally the link between `Node4` and `Node5` was cut. The client sends a request i_1 for word-counting content object `/node5/doc5` which is to

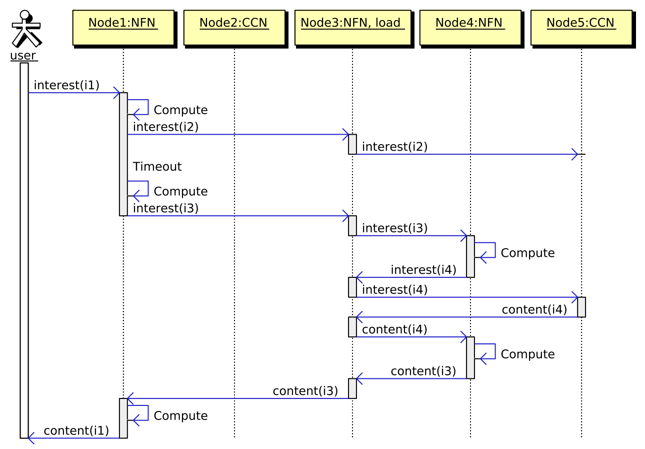

Figure 9: Experiment 6

i_1{/bin/scala/wrdcnt(/node5/doc)}

i_2{(λx./bin/scala/wrdcnt x) /node5/doc5}

i_3{(λx.x /node5/doc5) /bin/scala/wrdcnt}

i_4{/node5/doc5}

be found on the CCN-only Node5. At first the network reacts similar to Experiment 2: when the propagated Interest i_2 with the whole expression is propagated to Node5 it cannot be satisfied and times out. Node1 re-admits the program modified (according to the strategy in Sec.2.2) in Interest i_3. This time however Node3 does not take over the computation because it is marked overloaded, but forwards it towards Node4. Node4 computes the result by retrieving /node5/doc5 via Node3 and using the local copy of /bin/scala/wrdcnt. In the end, not only the client request was not rejected but the load-balancing action was taken implicitly by the network rather that being administrativelly congigured.

3.3 Gratuitous parallelism with thunks

Experiment 7 is a modified version of Experiment 4 where two word-count results have to be added up. We now use thunks as a means of continue working on the next sub-expression without waiting for the completion of the first sub-expression's result.

When Node3 receives the request to work on its sub-expression, it will immediately return a thunk (and start the actual work). A thunk is a temporary name that is routable to the node which started the computation, along side an optional completion time estimate. It can be seen a kind of "contract" that allows the requestor (Node1) to proceed immediatelly with the reduction of other terms of the expression and ask for the thunk's result later.

When Node1 cannot proceed any more with its reduction work, it has two options: (*i*) it can either adjust the PIT timer for the pending Interest of the client request not to expire for as long as the longest computation will last (time estimate returned with the thunk) so as

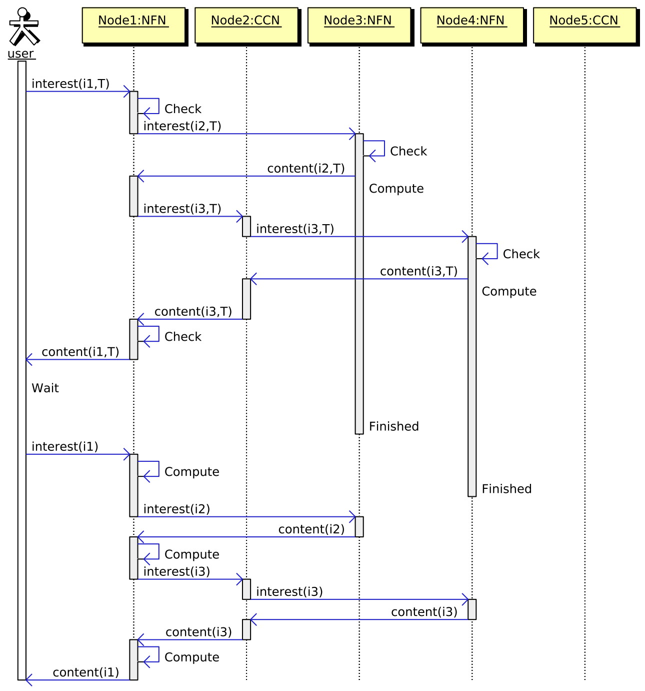

Figure 10: Experiment 7

i_1{(λf.add (f /node3/doc3) (f /node4/doc4))
/bin/scala/wrdcnt}

i_2{(λx./bin/scala/wrdcnt x) /node3/doc3}

i_3{(λx./bin/scala/wrdcnt x) /node4/doc4}

to issue the Interests on the thunks later on, or (*ii*) it can cache the thunks and return an equivalent thunk response to the client. In our case we have implemented the latter case policy, and so it is the client that later on re-issues the same request (based on the reported estimated execution time) for collecting the result.

Meanwhile, Node3 and Node4 compute in parallel the results for the initial i_2 and i_3. When the new requests arrive, they respond with the actual results which Node1 combines immediatelly. Note that when that second interest wave arrives, the interest messages do not ask for the same expression but for the thunk names. These names guarantee that the computation results will be retrieved from the right places.

The effect of thunks (and the parallelism that they enable) can be seen in Fig.12, where experiment 7 terminates in roughly half the time of experiment 4 although the client asks for the same computation result in both settings.

3.4 Regaining partial control of opportunism

The experiments 1 to 7 demonstrate an NFN network's capability to compute adequate placements for computations. In this section we show how clients can give placement hints to the network by a simple transformation of the given λ-expressions. This is based on

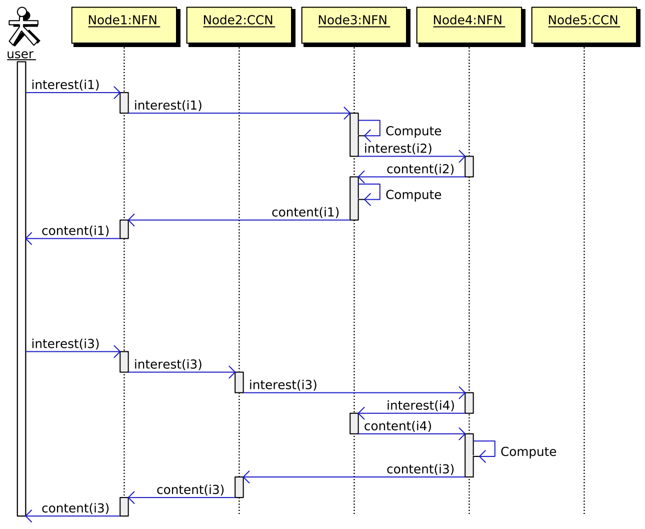

Figure 11: Experiment 8
i_1{(λd./bin/scala/wrdcnt d) /node3/doc3},
i_2{/bin/scala/wrdcnt},
i_3{(λf.f /node3/doc3) /bin/scala/wrdcnt},
i_4{/node3/doc3}

the observation that the following two expressions lead to different execution patters even though they both produce the same result:
i_1{/bin/scala/wrdcnt (/node3/doc)}
i_3{(λf.(f /node3/doc)) (/bin/scala/wrdcnt)}

Prepending a function abstraction (in interest i_3) that does nothing else than applying the function passed as a parameter to a fixed value, changes the routing target (shown underlined). That is, a client can influence the order in which the network should start resolving (and routing) names. The first case guides the network to a node which knows about the content - and this node then has to pull the `wrdcnt` code if needed. In the other case, the request will travel towards a node which has a copy of the code, and from there it is quite probably that the data must be pulled. The time-sequence chart of Experiment 8 shows both cases.

3.5 Performance overhead of NFN

Fig.12 shows for the experiments we conducted the completion times and for each of them the distribution of delay across (*i*) binary computations at the ComputeServer/JVM (blue), (*ii*) waiting time on Interest timeouts in the CCN network (red), and (*iii*) NFN processing (white). NFN processing includes expression evaluation at the abstract machine, network messaging and communication with the ComputeServer. Although these experiments are only indicative rather than conclusive so far, two standing out observations may be made. On one hand, NFN processing overhead is truly minimal compared to the actual data computations as well as the waiting times in the network. This affirms the lightweight type of operations it adds to the ICN forwarding plane. On the other hand the extreme effect of CCN timeouts on the total completion time of requests, confirms an important observation that stems from all our experiments so far regards the implications and potential benefits of adopting explicit response notifications (as by example NACKs) in CCN.

Specifically, in most of the experiments presented as well as others that we have conducted, it is easy to realise that the role of deciding alternative courses of action as entailed in the *resolution-forwarding* strategy of Sec.2.2, is almost always assumed by the first NFN node upstream from the client (`Node1` here). This is because of the fact that in the interaction with the CCN substrate this is the first one to detect timeouts. If explicit notications were available by the CCN protocol, which would be used to report infeasible computations or unavailable content, the role of selecting alternative actions would be taken much "deeper" in the network, lending to much wider distribution of tasks and computations.

A second implication of explicit notifications in relation to the *resolution-forwarding* strategy, is the time-granulatiry of for decisions for alternative actions (not having to wait first for 1-3 timeouts), which speeds up the overall completion time of the client request. We are currently working on tests with NACKs to validate and quantify both of these observations.

A third advantage of explicit notifications relates to the use of thunks for Map-Reduce type of operations. A thunk they way we have presented its use, has semantics of a contract that a computation will take place, along side a time-to-compute estimate. In practical reality estimating this "how-long" and qualifying it at different nodes in absence of a global clock is a very challenging task. This time estimate is mostly needed however to differentiate unvailability from statistical plausibility (of delivering a result) which is not discernible from timeouts. An explicit notification (which is to be interpreted as a timer to infinity in this context), largely obsoletes the hard need of the time-to-compute estimate.

4. FURTHER DISCUSSION

Following up the topic of thunks, so far we have used them to convey a notification of the sort: *"I start computing now, contact me later for the result"*. An alternative type of semantics, which requires further testing is for conveying the notification: *"I can compute, contact me when to start"*. The latter semantics have the benefit of allowing the client side of the computation to "collect offers" and choose among several candidate places where a computation can be completed, or delay a parallel execution for later. Although this would have the cost of some additional communication work

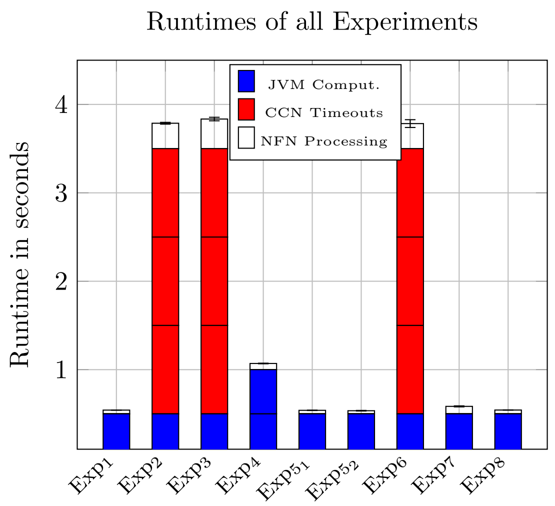

Figure 12: Per Experiment Runtimes

on the client side of the computation (e.g. `Node1` in Experiment 4 and 7), it could however avoid triggering redundant computations in different places when the request is multicast by the ICN in several directions.

A last topic of discussion we would like to touch regards the feasibility of NFN with other ICN architectures apart from NDN/CCN. Currently most ICN architectures have two key ingredients, one being the use of names for addressing information and a pub-sub like resolution capability to resolve names to data. These are the main two ingredients atop of which the NFN extends ICN functionality conceptually. In practise however we have seen that NFN names contain two types of components, one which is routable/resolvable in the ICN, and another which is glue for expressions, and which is non-routable but reducible by the abstract machine in NFN. Luckily, the latter also exists in one or another way in most ICN architecures usually refer to as “routing context” and which is often also subject to simple manipulations. E.g. in AS routing vector in DONA [13], *scope* in PSIRP/PURSUIT [19], *routing hints* in NetInf [7], *namespaces* in Convergence [17], and so on. On the other hand the more challenging requirement to fullfil is the current reliance of NFN on symmetric paths for the implementation of the combined expression-resolution strategy (Sec.2.2). This is a topic of follow up exploration.

5. RELATED WORK

Contextually and conceptually close to the work presented in this paper is the research on Service Centric Networking (SCN) [3]. SCN envisions to create processing workflows inside an ICN network through a manipulated concatenation of names that identify network services in the network to be interfaced. An SCN workflow is established using a much more basic methodology than NFN (as names do not encode expressions of program logic) which limits the level of network involvement (opportunism) in the “orchestration of the game”.

Also conceptually a point of inspiration for our ideas lay close to Borenstein’s AtomicMail [2] from 1992, who used a “programmable email system” of questionnaires collecting answers from a pre-defined set of recipients.

On the methodology, namely the use of λ-calculus for in-network programmability, our work resembles what has been in the past proposed as the Turing Switch [6]. The contextual and pragmatic difference is that while NFN applies this methodology for information handling at the application/information level, [6] envisions through it an infrastructure of universally programmable communication elements.

In the general topic of in-network programmability, in the past, Active Networking (AN) research [4] for more than a decade envisioned that users could load programs [25] into the net, that the net would provide richer programming primitives [8, 21], or complete programming (language) frameworks, eg. [10, 18], of which some of them were in fact functional. A modern reincarnation of the AN vision is seen in the objective of Software Define Networking [16] and thereby related programming environments [23, 9]. NFN’s focus however is not on programming the network, but rather using programs to involve the network more effectivelly (less statically) in information dissemination tasks. In this process the user does not configure the data path, change the forwarding plane, or explicitly load programs somewhere. These remain decisions of the network.

6. CONCLUSIONS AND FUTURE WORK

NFN attempts to capture an essential aspect of modern use of the Internet: the multimodality of information and its multi-purpose use. Towards this goal it extends the ICN name semantics and proposes that a name generally stands for a function. This function can be a constant mapping (variable lookup, as in ICN today), or a complex recipe involving many sub-operations which the network computes. *Name resolution* in the current ICN-way is then only a special case of *expression resolution.*

We present our first experiences with NFN in action, and demonstrate through a series of experiment scenarios that this functional extension leads easily to a generalization of “information access” so that it does not matter whether information is looked up or computed on the fly. The network is then put into a position where it can make corresponding trade-offs: Should it keep results in network memory or is it more economical to recompute it on demand and does it need to first fetch the code to do so? With mobile code technology in place, the network gets in charge of moving data and code around and orchestrates these moves.

7. REFERENCES

[1] Ahlgren et al. A survey of information-centric networking. *Communications Magazine, IEEE*, 50(7):26–36, July 2012.

[2] N. Borenstein. Computational mail as network infrastructure for computer-supported cooperative work. In *Int'l conference on Computer-Supported Cooperative Work*, 1992.

[3] T. Braun et al. Service-centric networking. In *Communications Workshops (ICC), 2011 IEEE International Conference on*, pages 1–6, June 2011.

[4] A. Campbell et al. A survey of programmable networks. *SIGCOMM Comput. Commun. Rev.*, 29(2):7–23, Apr. 1999.

[5] D. R. Cheriton and M. Gritter. Triad: A scalable deployable nat-based internet architecture. Technical report, Stanford University, 2000.

[6] J. Crowcroft. Turing Switches: Turing machines for all-optical Internet routing. Technical Report UCAM-CL-TR-556, Cambridge University, January 2003.

[7] C. Dannewitz et al. Network of information (netinf) - an information-centric networking architecture. *Comput. Commun.*, 36(7):721–735, Apr. 2013.

[8] D. Feldmeier et al. Protocol Boosters. *IEEE JSAC, Special Issue on Protocol Architectures for 21st Century Applications*, 16(3), April 1998.

[9] N. Foster et al. Frenetic: a network programming language. *SIGPLAN Not.*, 46(9), Sept. 2011.

[10] M. Hicks, P. Kakkar, J. Moore, C. Gunter, and S. Nettles. PLAN: A Packet Language for Active Networks. In *3rd ACM SIGPLAN International Conference on Functional Programming*, 1998.

[11] V. Jacobson et al. Ntworking Named Content. In *5th ACM CoNEXT*, 2009.

[12] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Ntworking Named Content. In *5th international conference on Emerging networking experiments and technologies*, ACM CoNEXT, pages 1–12, 2009.

[13] T. Koponen et al. A data-oriented (and beyond) network architecture. *SIGCOMM Comput. Commun. Rev.*, 37(4):181–192, Aug. 2007.

[14] J.-L. Krivine. A call-by-name lambda-calculus machine. *Higher Order Symbol. Comput.*, 20(3):199–207, Sept. 2007.

[15] X. Leroy. The Zinc Experiment: An Economical Implementation Of The ML Language. Technical Report TR 117, INRIA, 1990.

[16] N. McKeown et al. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2), Mar. 2008.

[17] N. B. Melazzi. Convergence: extending the media concept to include representations of real world objects. In *The Internet of Things*, pages 129–140. Springer, 2010.

[18] S. Merugu et al. Bowman and CANEs: Implementation of an Active Network. In *37th Allerton Conference on Communication, Control and Computing,*, Monticello, IL, September 1999.

[19] F. N. et al. Developing Information Networking Further: From PSIRP to PURSUIT. In *BROADNETS*, pages 1–13. Springer, 2010.

[20] M. Odersky et al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL Lausanne, 2004.

[21] C. Tschudin and R. Gold. Network pointers. *SIGCOMM Comput. Commun. Rev.*, 33(1):23–28, Jan. 2003.

[22] C. Tschudin and M. Sifalakis. Named functions and cached computations. In *11th Annual IEEE Consumer Communications and Networking Conference (to appear)*, January 2014.

[23] A. Voellmy, H. Kim, and N. Feamster. Procera: a language for high-level reactive network control. In *1st workshop on Hot topics in software defined networks*, HotSDN '12, pages 43–48, 2012.

[24] G. Xylomenos et al. A survey of information-centric networking research. *Communications Surveys Tutorials, IEEE*, 16(2):1024–1049, Second 2014.

[25] J. Zander and R. Forchheimer. Softnet - An approach to high level packet communication. In *2nd ARRL Amateur Radio Computer Networking Conference*, 1983.