An Information Centric Network for Computing the Distribution of Computations

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

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

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

Christian 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), such that in addition to resolving data access by name, it also supports the concept of function definition and application to data (or other functions) in the same resolution-by-name process. This empowers the network to select internally (optimal) places for fulfilling a potentially complex user expression. Forwarding optimization and routing policies become thereafter a basis of dynamic decisions for (re)-distributing computations, and retrieving results.

In this paper we describe the intrinsic operations and mechanisms of an instantiation of NFN based on untyped Lambda expressions and Scala procedures. Then, we demonstrate through a series of proof-of-concept experiments how they extend the capabilities of an information centric network (CCN), for orchestrating and distributing data computations, and re-using cached results from previous computations. In the end we report and discuss the main observations stemming from these experiments and highlight important insights that can impact the architecting of ICN protocols that focus on named-data.

Keywords

Network architectures; information centric networking; named data networking; named-function networking

1. INTRODUCTION

The architectural foundations and design "principles" of the early Internet 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. Today ICN research focuses on architecting away the shortcomings of host-centricity in the original Internet, addressing aspects of node mobility, security, dynamics of content dissemination, and most important, factoring out location dependence from the interaction of the user with information. Two common design foundations in many ICN architectures [22] are the adoption of indirection semantics (by varying similarity to a publish-subscribe system [4]), and the use of names to address content without involving host references; as implied in the characterisation "Named Data Networking" [10]. 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.

It is worth pondering to what extent these key design foundations of current ICN architectures simplify all possible aspects of *interconnecting information*, and what potential is thereby created for application/service innovation. On first sight, and amidst the cloud computing era, only connecting users to information seems a "halfway vision" for an information-centric Internet.

A broader vision called "Named Function Networking" (NFN) was introduced in [21] where information *access* (ala ICN) is complemented by information *processing* (as in Cloud computing). NFN essentially generalises the semantics of access 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 (and involving multiple names).

By composing expressions involving named data as well as named functions the user can describe information transformations, and the network gets in-charge of finding if and *how* the result can be obtained or synthesised, by interlacing expression-resolution with name-based forwarding. In this process in-network caching is now extended to also involve caching of computation results.

Like in the case of removing locality-of-storage aspects from data names, NFN removes locality-of-execution: instead of inferring from a user request the location for the computation and expecting from the routing substrate to reinforce its reachability, the NFN network discovers or appoints alternative places for hosting computations.

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

ICN'14, September 24–26, 2014, Paris, France.

ACM 978-1-4503-3206-4/14/09 ...$15.00.
http://dx.doi.org/10.1145/2660129.2660150 .

The remaining of this paper is organised as follows. In Sec. 2 we provide an overview of the main concepts, design tenets and mechanisms of NFN, and we present the node architecture and the unified expression resolution/forwarding strategy. In Sec. 3 we present a number of concept-proofing experiments, and report the results. In Sec. 4 we discuss the main observations, the issues we encountered, and possible solutions alongside their implications on ICN architecting. Finally Sec. 6 concludes the paper.

2. NAMED FUNCTION NETWORKING (IN A NUTSHELL)

NFN blends the interpretation of a program's control-flow with network forwarding, and thereby dynamically distributes computation tasks across an ICN network; orchestrating in this way the interaction of code with data on user's behalf (and outside his explicit control). This orchestration is effected in one of three different ways, depicted in Fig. 1.

The first case is 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 information needs be generated, either because it never was computed before or is not timely available. Case (c) covers the situation when some function or data required to evaluate an expression, is "pinned down" (non retrievable) by policy or other reasons. In this case the name resolution (evaluation and possibly execution) is delegated (pushed) towards the pinning site.

To achieve these objectives, names in NFN *represent* functional programs in their simplest, most compact and archaic form: λ-calculus expressions. Their manipulation and evaluation (progressively) "interferes" with name-based forwarding in ICN, and thereby is subject to network conditions, load, and routing policies.

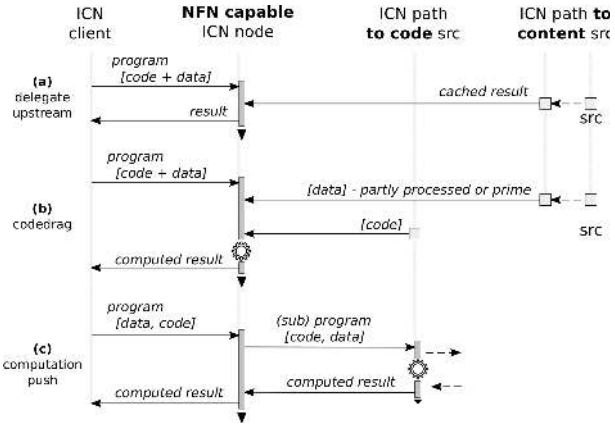

Figure 1: Three scenarios that NFN must handle: upstream fetch, separate code and data fetch, computation push

2.1 Lambda calculus & Expression evaluation

Church's Lambda (λ) Calculus, which is the basis of functional programming, defines recursively the form of terms that compose a valid expression, in one of three cases: variable lookup, function application and function abstraction

`expr` ::= `v` | `expr-l expr-r` | λ`x.expr`

The most basic form of a λ-term is just a variable name `v` (that may be resolved). The second valid form `expr-l expr-r`, the so called function application, is nothing else than a simple function call `expr-l(expr-r)` with one argument (`expr-r`). Notice that both the function and its argument are in turn λ-expressions and the only distinction of their roles is their relative left-vs-right placement. The third term form is called abstraction: λ`x.expr` is a definition of a function with one argument. It consists of a λ-expression `expr`, in which all occurrences of the formal parameter `x` are the places where the actual parameter value (function argument) has to be substituted.

Invocations (applications) and definitions of function with more than one arguments are possible by a succession of single argument function invocations and definitions respectively. Parentheses may be used to make expressions more readable but strictly speaking are not needed.

Complex expressions are transformed (evaluated) and often reduced to simpler ones, by iteratively applying *beta-reduction* operations whenever a function application term is encountered (beta-reduction specifies the rules of transformation). For example, the following line succession of λ-expressions shows such a sequence of (simplifying) transformations between equivalent forms, through beta-reductions.

(λp.λq.(p q p)) (λx.λy.x) ((λz.λs.z) (λk.k)) →
(λq.((λx.λy.x) q (λx.λy.x))) ((λz.λs.z) (λk.k)) →
(λx.λy.x) ((λz.λs.z) (λk.k)) (λx.λy.x) →
(λy.((λz.λs.z) (λk.k))) (λx.λy.x) →
(λz.λs.z) (λk.k) →
(λs.λk.k)

When starting the evaluation of an expression from a different term and following a different convention for selecting the next term to reduce (e.g. innermost/rightmost –λz.λs.z– as opposed to leftmost –λp.λq.(p q p)), one may realise that different reduction paths emerge, all of which however lead to the same result (confluence theorem). This is a basis for alternative resolution strategies such as call-by-name/need/value/etc. *Call-by-name* resolution is particularly useful in the context of network computations because sub-expression resolution is delayed until their result "becomes essential"; implying in this way a potential saving of processing resources and reduction of the entailed amount of network traffic.

Although on first sight the untyped λ-calculus, presented above, seems nothing more than an elegant name-reshuffling machinery, it nevertheless allows us to express program logic of arbitrary complexity, very compactly encoded in ICN names, and limited only by the maximum allowed length of a name (i.e. packet size). On the other hand, actual binary data processing operations, cannot be efficiently handled at the name-manipulation level (although theoretically possible in reality it is impractical). For this reason NFN assumes two levels of program execution: One regards the name-manipulation and the orchestration of computation distribution; handled by the functional untyped λ-calculus. The other regards native code execution for actual data processing tasks at the identified execution site(s), which in our prototype is done in Java byte code for procedures written in the Scala language [18].

Overall the use of λ-calculus in NFN serves a role similar to a simple IDL[1] language, involving only two operations:

[1]Interactive Data Language, but also Intermediate Definition Language

(i) variable look-up –the name resolution ala ICN–, and (ii) term reduction, where a function is applied to its arguments and composes new terms from them.

2.2 NFN node architecture

In its current instantiation NFN extends the CCN/NDN [10] architecture[2], hereafter referred to as CCN, (i) by integrating a λ-*expression resolution engine* in a CCN relay, and (ii) by optionally hosting an application processing/execution environment. These two extensions correspond to the two levels of program execution discussed earlier.

The λ-expression resolution engine is situated within the CCN relay, and processes all Interests that have the *implicit* postfix name component `/NFN` (this is by analogy to the way the current CCN protocol implementation handles name checksum hashes). It embodies a Krivine *Abstract Machine* [12] (AM) that follows a call-by-name reduction strategy for "lazy evaluation" of λ-expressions. Call-by-name evaluation guarantees that Interests for the recursive evaluation of sub-expressions will be sent out in the network only if/when the result of the sub-expression is needed. To implement the required primitives (Table 1) we used the ZAM [13] instruction set of Caml, on a *Stack-Machine* with two stacks: One, for holding intermediate reduction state, and the other for resolving external invocations to native code data processing functions. This implementation is compact and lightweight in a typical CCNx relay, enabling controlled resource allocation for NFN-extended processing.

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: Krivine Abstract Machine primitives using the ZAM instruction set.

The application processing environment is currently a Scala language[18] ComputeServer (practically a JVM), layered over CCN by being attached to a node-local Face. The CCN relay demuxes to it requests for data processing computations by usual longest prefix match against prefix `/COMPUTE`. The application processing environment is optional in the sense that there is no requirement for all NFN nodes to be capable of "number crunching" data processing operations (NFN nodes that have a pure router/forwarder role, only need the AM for the distribution of computation tasks and caching of results).

On NFN nodes hosting a ComputeServer, a native code named function is registered in the NFN realm with a *publish* primitive. This, aside from populating the CCN node's FIB with a corresponding namespace entry, also inserts in the AM's dictionary a mapping to a `'call <num> <func>'` statement (where `<num>` refers to the number of arguments required to invoke `<func>`). When processing a CCN name as a λ-expression, and `<func>` is encountered in the next term, it is replaced with the respective `call` mapping. This prepares the AM to interface with the ComputeServer by "mangling" enough additional terms for the external native function invocation. Furthermore, the `call` operation is a "game-changer" during the expression resolution, because it forces a switch in the evaluation strategy from call-by-name to call-by-value momentarily, for the completion of the external native code operation. This means that before the external call to `<func>` is made, each mangled term that will be used as a parameter must be a fully evaluated and resolved expression.

2.3 Distributing computations: Program translation & 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 be encoded equivalently in any of the λ-expressions that follow[3].

1. `func data`
2. `(`λ`zy.z y) func data`
3. `(`λ`y.func y) data`
4. `(`λ`z.z data) func`

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

As code and data are treated interchangeably by virtue of their names in the ICN network, both `func` and `data` can be independently addressable CCN names for content. For example if

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

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

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

NFN packages this named expression inside a CCN Interest as follows (in this symbolism of a CCN Interest, '|' delimits individual name components[4])

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

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. A more convenient representation that we will use hereafter to refer to the same Interest is

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

[2]Architecture compatibility with ver $\leq$ 0.8.1 of CCNx, and ver $\leq$ 0.2 of NDN.

[3]The possible forms are not limited to these four only of course.

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

The underlined name component appears in the first (position 0) in the Interest packet encoding, influencing by rule of longest prefix match, the Interest forwarding.

To distribute computations in the network, NFN currently implements the following strategy (Fig. 2). Initially (phase 1) using Interest i_1, a cached copy of a transcoded version of media is sought for, en-path to `/name/of/media`. Having component `/name/of/media` in first position 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 a perfectly plausible search also at the data source, even in a CCN-only node if such a naming convention has been adopted.

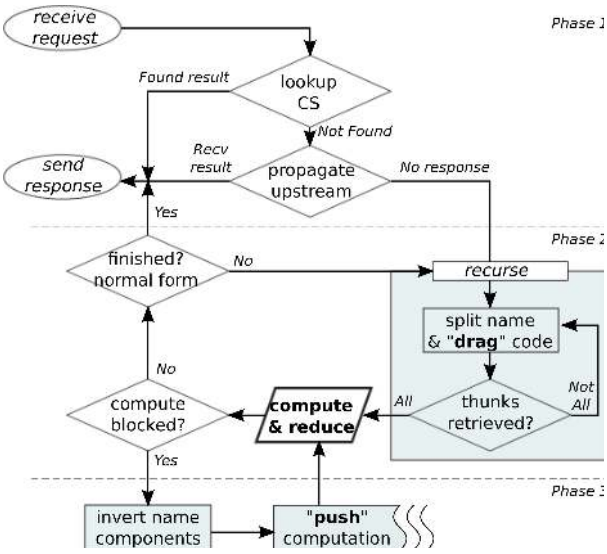

Figure 2: NFN forwarding strategy for CCN.

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`, may attempt to compute the result (subject to policies, and processing resource availability). This is made feasible at any NFN node, by extracting the CCN names from the λ-expression and forking separate individual Interests for `/name/of/media` and `/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 will be cached still en-path and possibly close to the data source, increasing its re-use potential in other requests.

If any of the two Interests does not yield the content back for some reason, and before giving up, there is still the possibility to take the computation off-path (phase 3). The NFN node may become a computation-proxy directing the computation towards the code source 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 equivalent form of the expression refers to the same computation, but has the `/name/of/transcoder` name component in the first position, which results in forwarding the Interest in the direction of the function ("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 if the code data does not exist, due to a "name pinning" policy for not distributing the code, etc. What is important, however is that: (a) Distribution of computation tasks in the network does not entail 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 or may not take place, leaving the computation placement and resource allocation decision entirely to the network (avoiding single point of failure, compensating for routing failures, and partly protecting against DoS attacks targeting a specific host or service).

Finally, in the whole process of evaluating a named expression, requests for intermediate results in nested terms can in fact retrieve only *thunks*. One can think of a thunk in NFN as a reference or contract for the feasibility of a computation, whose results can be retrieved later in time. Thunks allow the evaluation of a named expression to progress even when results are not available yet (enabling asynchronous and parallel computations as we will demonstrate in one of the experiments later on)[5].

3. EXPERIENCES WITH NFN

In this section we report and discuss our experiences with a proof-of-concept evaluation of NFN in a small testbed topology. The goals set out for this evaluation have been to

- Showcase the ability of NFN for dynamic distribution of computation tasks and interactions between static data and functions inside an ICN network (and involving the benefits of caching computation results).
- Test, and identify occasions where NFN empowers network side decisions and optimisations. Understand the nature of these optimising decisions, and develop insights of how to improve the effectiveness of NFN.
- Have a first indication of the comparative overhead of running NFN, in a CCN network.

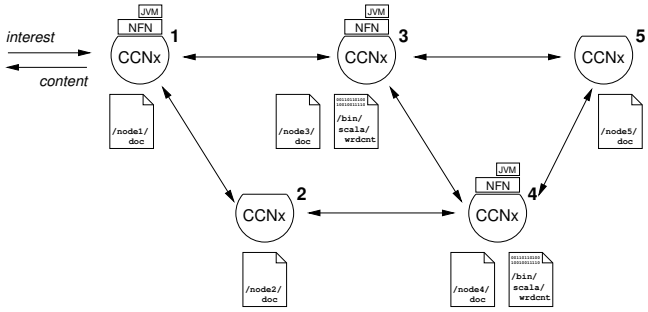

Figure 3: Testbed topology – two CCN & three NFN nodes.

[5]A thunk-ed name appearing in a CCN packet has 2 postfix components i_n`[..|/NFN|/TH]` for delivering to, and appropriately interpreting by, the NFN resolver

3.1 Experimental set-up

Our testbed is a hybrid topology including CCN-NFN as well as CCN-only nodes (Fig. 3). The topology and size of the testbed were kept simple enough for alpha-testing and ease of tracking the node interactions, and at the same time complex enough to serve our demonstration purposes of NFN's features. As shown in Fig. 3, it consists of five nodes, where two are pure CCN nodes and the other three are NFN nodes (AM extension to the CCN relay) hosting additionally the Scala [18] ComputeServer (JVM) execution environment; connections between nodes are bidirectional; client requests always arrive at `Node1` first. Any deviations from this set-up is reported in the individual sections of the experiments.

The FIBs of the nodes are initialised manually (in absence of dynamic routing currently for CCN) and such that each node can reach every other node over the shortest path. When more than one paths are available, 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, published with name `/bin/scala/wrdcnt`, and corresponding FIB entries are placed in all other nodes. This procedure takes as a single argument a document and computes the number of words in it. In our simulation 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

Experiment 1 (code+content pull): The first experiment is a vanilla check of case (b) of Fig. 1 for carrying out locally computations by first retrieving code and data. It starts by a user requesting the word-counting of `/node1/doc`. Following our default mapping of λ-expressions to CCN messages, the argument `/node1/doc` becomes the first name component, which characterises `Node1` as the recipient of the expression. When `Node1` starts resolving the expression and the component `/bin/scala/wrdcnt` is encountered, it issues Interest i_2 to retrieve it, which is satisfied by `Node3`. When `Node1` receives the bytecode of the procedure it applies it to the locally available document and returns the word count result in a content object to the client.

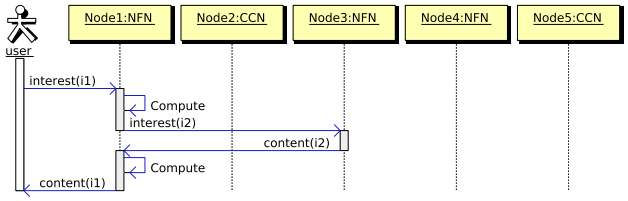

Figure 4: Experiment 1 – the network pulls code, applies it to locally available data
i_1`{/bin/scala/wrdcnt (/node1/doc)}`
i_2`{/bin/scala/wrdcnt}`

Experiment 2 (computation push): The second experiment demonstrates case (c) of Fig. 1 for delegating a computation, as a result of name manipulations in NFN that influence decision of the CCN 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 named objects is available locally on `Node1`, according to the strategy discussed in Sec. 2.3, `Node1` places the name component `/node5/doc` at the first position and propagates the Interest i_2 towards `Node5`. However, at `Node5` the Interest times out because it is a CCN-only node: the remaining name components cannot be matched exactly or the Interest cannot be propagated further based on longest prefix match.

`Node1` then reverts to the next phase of the strategy from Sec. 2.3 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 `wrdcnt`, which means it can start evaluating the expression and then separately request the content of `/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.

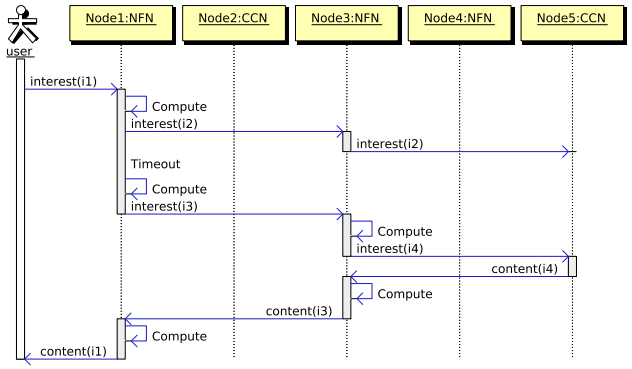

Figure 5: Experiment 2 – computation push (the network works around a CCN-only node)
i_1`{/bin/scala/wrdcnt (/node5/doc)}`
i_2`{/bin/scala/wrdcnt (/node5/doc)}`
i_3`{(λx.(x /node5/doc)) (/bin/scala/wrdcnt)}`
i_4`{/node5/doc}`

Experiment 3 (failover conditions): Assume the same query was issued as in experiment 2 and additionally that the connection between `Node1` and `Node3` failed: How should the network use the alternate path that exists between `Node1` to `Node5`? Fig. 6 shows that 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! Note that although there is an alternative (albeit longer) path to `Node3`, who may compute the expression, re-routing does not try to deliver the computation there. Instead, another (additional and closer) location is found for the computation, on `Node4`. Since the locality of computation is not part of the user request, if alternative computation-capable places are available, NFN might try to exploit them rather than forcing traffic to one only specific place!

Experiment 4 (recursive distribution): For this experiment another native (bytecode) procedure that accepts a

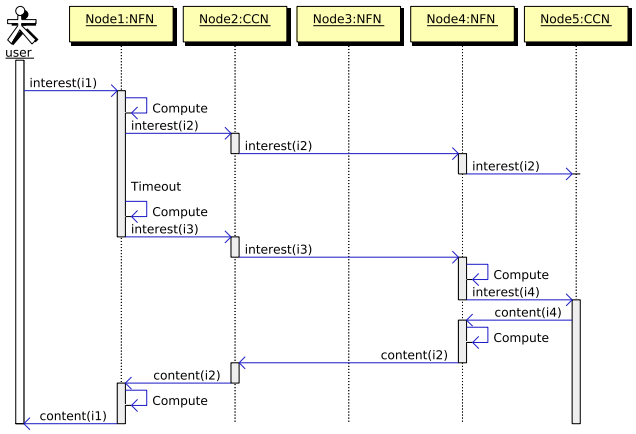

Figure 6: Experiment 3 – failover (the network discovers another suitable computing place)
i_1`{/bin/scala/wrdcnt (/node5/doc)}`
i_2`{/bin/scala/wrdcnt (/node5/doc)}`
i_3`{(λx.(x /node5/doc)) (/bin/scala/wrdcnt)}`
i_4`{/node5/doc}`

variable list of integer arguments and sums their values, is registered with name `/bin/scala/sum` at `Node1`.

In Fig. 7, a client sends a request for the sum of the word-counts of two documents in i_1. The word-counting of each document can take place independently and at different places. As shown this is orchestrated at `Node1`, where `/bin/scala/sum` (first name component of i_1) is found. The expression evaluation progresses by two reduction steps until the point that results from the word-counting sub-expressions are necessary for the sum to be computed. Each sub-expression is resolved *in turn* through a separate Interest: The first subexpression leads to Interest i_2, forwarded to `Node3` and computed there. The second subexpression yields Interest i_3 which is computed at `Node4`. The partial results are collected at `Node1` which can then use them to evaluate the sum and return the answer to the client.

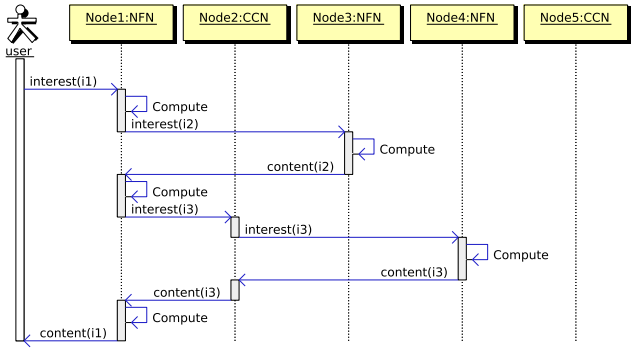

Figure 7: Experiment 4 – sequential evaluation of sub-expressions
i_1`{((λf.λg.(f (g /node3/doc) (g /node4/doc)))`
`(/bin/scala/wrdcnt)) (/bin/scala/sum)}`
i_2`{/bin/scala/wrdcnt (/node3/doc)}`
i_3`{/bin/scala/wrdcnt (/node4/doc)}`

This experiment 4 shows how more complex computations are recursively decomposed by NFN into a workflow of tasks with rather opportunistic coupling among them (compare this to the tight signalling coupling expected by components of most SoA architectures today). Later on, in Sec. 3.3, we show how through the use of *thunks* the same request is worked out in parallel, effectively providing an opportunistic Map-Reduce transport across an ICN network.

Experiment 5 (cached results prevent repeated computations): 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.

A word-count request for document (`/node3/doc`) is sent in Interest i_1, before the request for the sum of the two document word-counts as in experiment 4. `Node1` passes the request in i_2 to `Node3` (where both the data and code are available), which computes and returns the result; `Node1` will cache this result. When the next request for summing up the two word-counts of `/node3/doc3` and `/node4/doc4` is received in i_3, `Node1` already has the result of the first computation for `/node3/doc`. Hence, it issues only one Interest (i_4) for the sub-expression involving the word-count of `/node4/doc`. This time the request for i_3 completes much faster than in the previous example, as seen in Fig. 12 (comparing the times of $Exp4$ and $Exp5_2$).

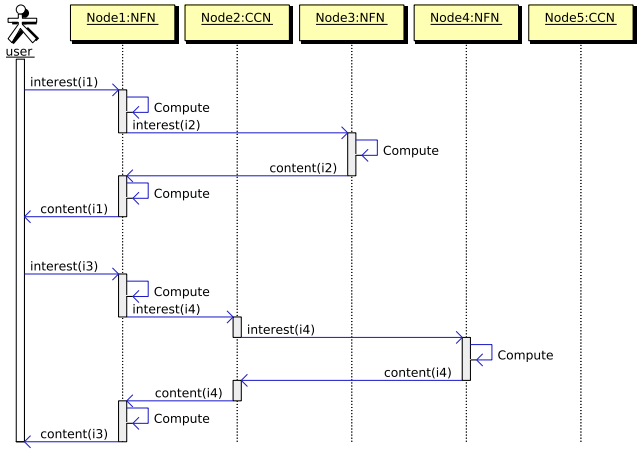

Figure 8: Experiment 5 – accessing cached results
i_1`{/bin/scala/wrdcnt (/node3/doc)}`
i_2`{/bin/scala/wrdcnt (/node3/doc)}`
i_3`{((λf.λg.(f (g /node3/doc) (g /node4/doc)))`
`(/bin/scala/wrdcnt)) (/bin/scala/sum)}`
i_4`{/bin/scala/wrdcnt (/node4/doc)}`

Experiment 6 (Node loaded, pass it to the next): It can happen that some NFN node capable of data computations is overloaded, e.g. only because it happens to be closer to popular content or because it receives voluminous requests. The opportunistic location-decoupled nature of computations in NFN can enable implicit load-balancing in the ICN network.

In this experiment, `Node3` is designated to be in *overloaded* state and 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 be found on the CCN-only `Node5`.

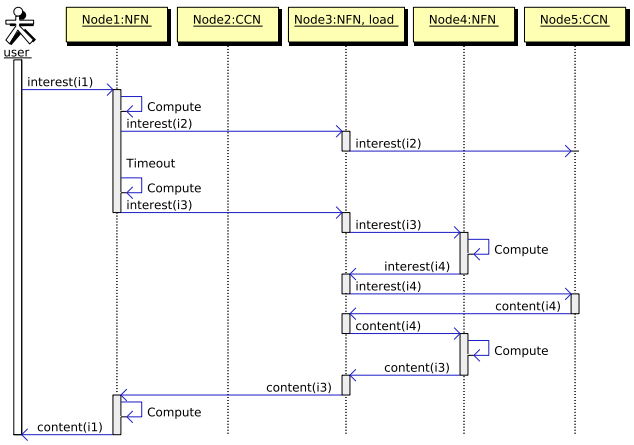

Figure 9: Experiment 6 – implicit load-balancing
i_1`{/bin/scala/wrdcnt (/node5/doc)}`
i_2`{(λx./bin/scala/wrdcnt x) /node5/doc5}`
i_3`{(λx.(x /node5/doc)) (/bin/scala/wrdcnt)}`
i_4`{/node5/doc}`

At first, the network reacts similarly to experiment 2: when the propagated whole request in i_2 arrives to `Node5` it times out. `Node1` re-admits the program modified (according to the strategy in Sec. 2.3) as Interest i_3. This time however, `Node3` cannot assume the computation because it is marked overloaded, and instead 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 a load-balancing action was taken implicitly by the network rather that being administratively configured.

3.3 Gratuitous parallelism with thunks

Experiment 7 is a modified version of Experiment 4 where two word-counting results are summed up. We have now enabled the use of thunks to allow asynchronous non-blocking dispatching of the two sub-expressions.

When `Node3` receives the request i_2 for word-counting `/node3/doc`, it will immediately return a thunk (and start the actual work). A thunk response contains a temporary name that is routable back to the node that started the computation, along-side an optional completion time estimate. It is a "contract" that allows the requester (`Node1`) to continue work (e.g. proceed in the reduction of a blocked expression, and evaluation of other sub-terms) and ask for the thunk-ed result later. At the same time `Node1` may adjust the PIT timer for the pending Interest from the client not to expire for as long time as the longest sub-computation will need (time estimate returned with the thunk) – so as to refresh the Interests on the thunks later on – and pass an equivalent thunk name response to the client. In case the PIT timer on `Node1` expires without having successfully acquired the thunk-ed result, the client can either re-issue the request, or try and re-animate the partially completed computation by using the thunk name as a "reminder" to refer `Node1` to possibly cached state of the sub-computation.

Meanwhile, `Node3` and `Node4` compute in parallel the results for the i_2 and i_3 and when the thunk-ed requests arrive anew from `Node1`, they respond with the actual results. This second round of Interests sent from `Node1`, contain the thunk names and *not* the original names of the sub-expressions. This guarantees that the computation results will be retrieved from the correct (contracted) places.

The effect of thunks and the parallelism that they enable can be seen in the measurements of 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.

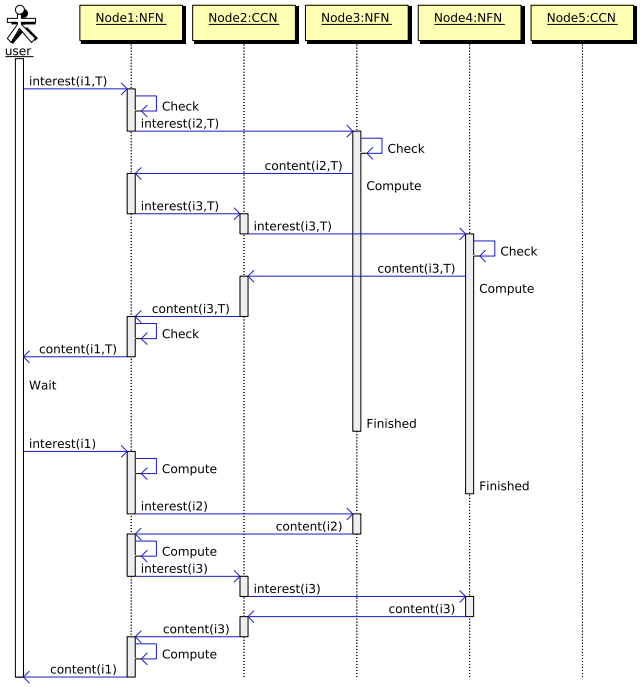

Figure 10: Experiment 7 – gratuitous parallelism
i_1`{((λf.λg.(f (g /node3/doc) (g /node4/doc)))`
`(/bin/scala/wrdcnt)) (/bin/scala/sum)}`
i_2`{(λx./bin/scala/wrdcnt x) (/node3/doc)}`
i_3`{(λx./bin/scala/wrdcnt x) (/node4/doc)}`

3.4 Preferential opportunism in the distribution of computations

The previous experiments demonstrated the NFN network's ability to discover places where computations can take place. In this process it is also possible for clients to "give hints" to NFN for preferential placement of computations, through simple user-prepared abstraction transformations of the λ-expression or selected sub-terms, in the initial request. For example, as seen in **experiment 8** (Fig. 11), the following two equivalent expressions even though they produce the same result, they follow different resolution paths in the network:

i_1`{/bin/scala/wrdcnt (/node3/doc)}`
i_3`{(λf.(f /node3/doc)) (/bin/scala/wrdcnt)}`

Transforming i_1 before issuing the initial request, with a λ-abstraction to i_3, does nothing else than simply changing the argument name, which will comprise the first prefix component in the CCN name (shown underlined). In the first case the computation will be attempted first near `/node3/doc`, and in the latter near `/bin/scala/wrdcnt`. This trick can be

exploited *by the client* to format programs, such that preferences for the distribution of computations are expressed –but importantly, they cannot be enforced– for individual sub-expressions (in the program).

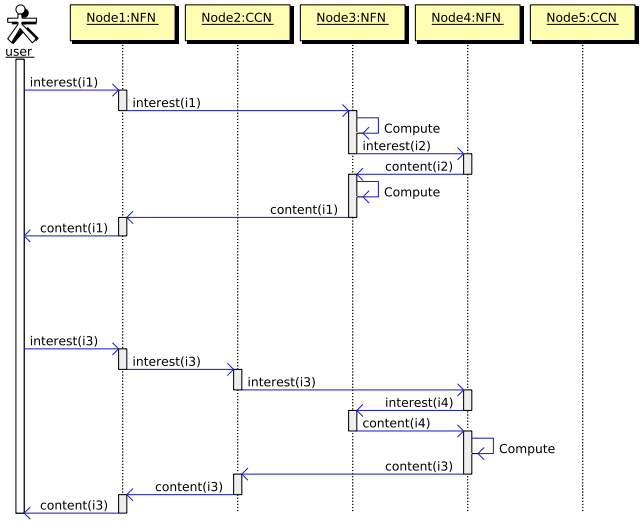

Figure 11: Experiment 8 – same result, but different preferences hinted at by the client
i_1`{(λd./bin/scala/wrdcnt d) /node3/doc3}`,
i_2`{/bin/scala/wrdcnt}`,
i_3`{(λf.f /node3/doc3) /bin/scala/wrdcnt}`,
i_4`{/node3/doc3}`

3.5 Performance overhead of NFN-over-CCN

Fig. 12 provides an indication of the completion times for the experiments we conducted, and for each of them the distribution of delay components across (*i*) native code execution at the ComputeServer/JVM, in blue, (*ii*) waiting time on Interest timeouts in the CCN network, in red, and (*iii*) NFN processing, in white. NFN processing includes expression evaluation at the abstract machine and communication with the ComputeServer.

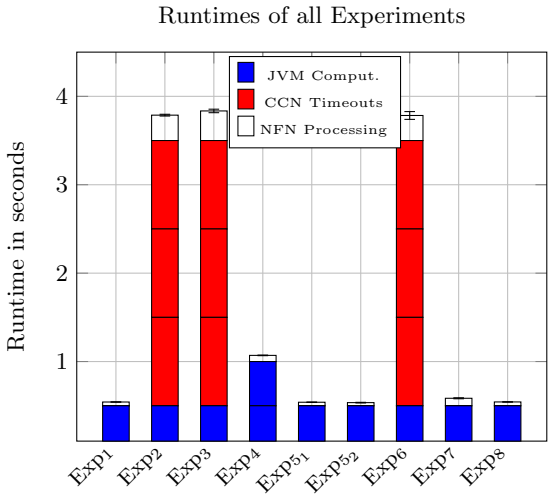

Figure 12: Per experiment runtimes

Although these experiments are only indicative (given the non-quantitative character of our experiments), nevertheless two standing out observations may be made. The first is that NFN processing is definitely affordable in the time scales of operation and decisions in CCN. Even with a very crude prototype implementation all NFN AM-related operations across up to 3-4 nodes take only a fraction of the typical CCN Interest timeout; and affirms the lightweight type of operation it adds to the ICN forwarding plane.

The second observation regards the overhead that the current CCN architecture inflicts on NFN, witnessed in the effects of timeouts (as an implicit feedback mechanism) on the total completion time of NFN client requests. The agility of NFN in making decisions is seriously limited by the absence of fast feedback; after probing for computations that are infeasible or for content/code that cannot be retrieved. The effect is more pronounced in NFN than in CCN only, since a typical NFN request entails several CCN level transactions.

4. DISCUSSION & KEY INSIGHTS

4.1 NACKs, explicit vs. implicit feedback

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. 2.3, 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 and react to them. If explicit notifications were available by the CCN protocol (such as ACKs/NACKs), to help detect faster infeasible computations or unavailable content, the role of selecting alternative actions would be assumed by NFN nodes "deeper" in the network, and lending to much wider distribution of tasks and computations. The time scale of decisions would also be much shorter (currently waiting for 1-3 timeouts), thus improving NFN's agility.

In current absence of explicit notifications in the CCN architecture, a number of alternative solutions deserve exploration. One of them is to regard thunks as a means of explicit notifications e.g. as in Map-Reduce type of operations. A thunk currently includes a time-to-compute estimate, which in practical reality it is challenging to qualify at different nodes in absence of a global clock. This time estimate is mostly useful therefore to differentiate unavailability from statistical plausibility (of acquiring a result), which is otherwise not discernible from timeouts. A NACK in this context is nothing more than a thunk with a timer to infinity. The problem with this "NFN-level" approach is that in a hybrid environment the semantics would only be understood by NFN nodes, making its effects partial or occasional.

Another partial solution[6], which would likely improve the problem, without architecting explicit notifications inside CCN, would be the creation of a decreasing timeout gradient as Interests travel away from the receiver. This would require that all CCN nodes have the same default PIT timer, and that an Interest travelling from hop to hop has a means to request its registration in the PIT with an increasing decrement from the default timer. The Interest will then have the possibility of expiring earlier the further away it has travelled from the source, and the respective NFN node

[6]Contributed by one of the reviewers of this paper

would be the one to seek alternative courses of action. This approach is quite fragile however and requires hard-wiring common defaults to all CCN nodes.

4.2 The right semantics for thunks

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, would be to convey 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 parallelized execution for later. Although this would have the cost of some additional communication work 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 CCN in several directions. This exploration is the topic of future quantitative evaluation.

4.3 Security

Intuitively one may ponder on the security implications of having a possibility to spawn arbitrary computations from the middle of the network through "hand-grenade" programs masqueraded as normal requests (even by accident). Although we have not architected NFN or engineered its key functions with security as a prime concern, we have been however security-implications aware.

The tenets of caching and re-using computation results, and the removal of locality-of-execution, are two top-listed features of NFN, which can minimize the effectiveness of DoS attacks towards specific targets in the network (more than what is actually possible in today's Internet).

Call-by-name expression resolution warrants that a request for evaluating a sub-expression will be dispatched in the network only if the result of that sub-expression will actually be used, making the plausibility of an attack not deterministically discernible.

Thunks with the alternative semantics describe before have also the potential to protect against waste of computation resources: the orchestrating NFN node can ensure that thunked computations are spawned only if their enclosing expression is feasible and allowed.

All these features to start with, although not securing the network, they nevertheless, intend to limit or localise the effects of an attack, and make it difficult to plan against specific targets.

Additionally, access control to functions can be protected by similar means that it is today in SoA. Data cannot be altered in NFN, only new data can be generated from other data and while older fade-away from caches. Every entity that generates new data must sign them according to data authorship rules in CCN, and when source data are transformed by some function, chained signing can be used to assert and verify the function owner, its inputs and its outputs (independently always of location).

4.4 NFN in ICN architectures

Clearly (to us), NFN-type of functionality deserves to be a part of an ICN enabled Internet. The question is where should this functionality be fleshed out? Should it be *part of* the ICN forwarding plane or engineered *on top of* it?

The arguments that speak against its embedding in the ICN plane are those of execution performance overhead, and architectural simplicity (the internetworking layer "needs" to be super fast and simple). On the aspect of execution performance, our first indications (albeit admittedly very preliminary to support evidence) are that the overhead is easily sustainable considering the CCN protocol time scales of operation (decisions driven by timeouts). Moreover, as the typical NFN topology would sufficienly be a hybrid one, involving NFN as well as CCN-only nodes, scalability should not be a concern (although this also requires a quantitative validation to confirm).

The counter arguments that speak for its placement in the ICN plane are twofold: (*i*) Name manipulation by NFN's Abstract Machines influence the forwarding/routing semantics, which naturally belongs to the network's forwarding and routing layer, by analogy that NAT's IP address manipulation functions today are part of the IP and transport layers. On the other hand, the actual NFN's data processing execution environment and host of application logic, resides at CCN's application layer, mostly to be found near the network's edge. (*ii*) It is critical that routing and compute-placement decisions are made as close as possible where things (timeouts, unavailability of resources or capabilities) can be discovered, i.e. close to the root cause. Otherwise, if nodes somewhere at the edge have to discover what happened, one needs either to setup rich diagnostic feedback protocols and pay a price for that complexity and additional delay, or let the edge nodes timeout, which is catastrophic for performance.

Finally, if one accepts a de-facto role of NFN in ICN, the question arises about the feasibility of NFN with other architectures apart from NDN/CCN. 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 not directly routable but manipulated by the AM in NFN. Interestingly, the latter also appears in most ICN architectures, often specialised as a "routing context" but which is nevertheless identified with simple manipulations: E.g. AS routing vector in DONA [11], *scope* in PSIRP/PURSUIT [8], *routing hints* in NetInf [6], *namespaces* in Convergence [16], and so on. On the other hand, a more challenging requirement in other ICN architectures is NFN's current reliance on symmetric paths for the implementation of the combined expression-resolution strategy (Sec. 2.3). This is a topic of follow up exploration.

5. RELATED WORK

Mostly related to the work presented in this paper is the research on Service Centric Networking (SCN) [2]. 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. Expressibility of distributed computations and workflow creation in SCN is much more basic when compared to NFN. On the other hand SCN creates workflows by means of interface specifications and service descriptions. This feature of SCN can provide an elegant approach of introducing SLAs in NFN, which can be the basis for an information centric Service-oriented-Architecture.

An inspiring work for our ideas is Borenstein's Atomic-Mail [1] from 1992, who proposed and developed a "programmable email system". Similarly inspiring has been the

idea of a Turing Switch [5], which is a universal network interconnecting element on which all link-layer and network-layer functions are expressed through λ-calculus programs.

In seeing the network as a distributed database, work on declarative networking [14] pioneered interesting ideas, including a network query language (NDlog), that enabled access to distributed information and coordination of computations at disparate locations, without explicit reference of communication primitives. Although the core philosophy and methodology between declarative networking and NFN are different (e.g. declarative query language, versus imperative functional programs), there are important similarities and analogies that deserve further exploration and possible cross fertilisation of ideas.

In the general topic of in-network programmability, in the past, Active Networking (AN) research [3] envisioned users able to load programs [23, 19] in a network data path that supports programming primitives [7, 20], or programming language frameworks, eg. [9, 17]. A modern re-incarnation of parts of the AN vision is found today in the objectives of Software Define Networking [15]. Unlike AN and SDN, however, NFN's primary focus is not on explicitly programming a data path, but rather letting the network compose a data path that satisfies the needs of a high level user program, thus the main decision making lies with the network.

6. CONCLUSIONS AND FUTURE WORK

NFN captures an essential aspect of modern use of the Internet: multi-modality of information and multi-purpose use. With this goal it extends the ICN name semantics and proposes that a name generally stands for a function: a constant mapping (as in ICN today), but also a complex recipe involving many sub-operations. *Name resolution* in the current ICN-way is then only a special case of *expression resolution*.

We presented first experiences with NFN in action, and demonstrated through a series of experiments several scenarios where this functional extension leads easily to a generalization of "information access" in the network: static information look up complemented by dynamic computation on the fly. We have gained interesting insights to fuel follow up work and orient more extensive evaluations, but also which have the possibility to influence architectural work in CCN and ICN in general.

7. REFERENCES

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

[2] T. Braun et al. Service-Centric Networking. In *Int'l IEEE Conference on Communications (ICC)*, pages 1–6, June 2011.

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

[4] M. Caporuscio et al. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. *IEEE Trans. Softw. Eng.*, 29(12):1059–1071, December 2003.

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

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

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

[8] N. Fotiou et al. Developing Information Networking Further: From PSIRP to PURSUIT. In *International Conference on Broadband Communications, Networks, and Systems*, pages 1–13. Springer, 2010.

[9] M. Hicks et al. PLAN: A Packet Language for Active Networks. In *3rd ACM SIGPLAN Int'l conference on Functional Programming*, 1998.

[10] V. Jacobson et al. Networking Named Content. In *Int'l ACM conference on Emerging networking experiments and technologies (CoNEXT)*, 2009.

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

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

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

[14] B. T. Loo et al. Declarative networking: Language, execution and optimization. In *Int'l ACM SIGMOD Conference on Management of Data*. ACM, 2006.

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

[16] 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.

[17] 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.

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

[19] S. Schmid et al. A highly flexible service composition framework for real-life networks. *Computer Networks, Special Issue on Active Networks*, 50:2488–2505, October 2006.

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

[21] C. Tschudin and M. Sifalakis. Named functions and cached computations. In *Annual IEEE conference on Consumer Communications and Networking*, Jan. 2014.

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

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