Named Functions and Cached Computations

Christian Tschudin
Dept of Mathematics and Computer Science
University of Basel, Switzerland
Email: christian.tschudin@unibas.ch

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

***Abstract*—Current ICN research favors a key-value-store view of the network, where location agnostic *names* typically resolve to documents, data blocks or sensor values. We believe that *names* should not only refer to data but also to *functions and computation tasks*. In Named Function Networking (NFN) the network's role becomes to resolve *names* to computations, par example by reducing λ-expressions. In doing so, the network starts acting like a computing machine, capable of not only caching content but also computation results.**

We present basic concepts of NFN and report on our implementation that embeds the name resolution logic of CCNx in a generic resolver of λ-expressions. We demonstrate its resolution power beyond mere content-pull, to also leverage code-drag and computation-push as well as generalizing CCNx protocol functions.

***Index Terms*—Computer networks, information centric networking, named data networking, network architecture.**

I. Introduction

The main tenet of information centric networking (ICN) is that users should name data without reference to location, permitting the network to optimize from where the data is served. Still, the network is seen as an infrastructure to move data from one end (or from inside) of the network to another place. Overall, classic ICN looks like a distributed key-value-store that answers name-lookup requests.

However, *using the data*, rather than fetching it, is what users want. Moreover, there are cases where it is not feasible to transfer the data at all because of the sheer size, or security policy. In this case, "use" should happen in close locality of where the data is, implying that users must be able to express what function should be applied to the data of interest.

We suggest that an ICN should offer names for functions too, enabling users to say *what* result they need by writing expressions that refer to data and function names; The network substrate would then be in charge of finding out *how* these results can be obtained, either by computing them or by looking them up in case it was already computed by others. Like in the case of removing locality-of-storage aspects from data names, in our approach we remove locality-of-execution: Data caching is now extended to also cover caching of results.

Function Names for Information Access: Our work extends "named data networking" to the realm of functional programming: We propose 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. While turning every data object into a function might seem extreme, we can report that crafting a functional programming language onto existing ICN architectures is very straight forward with surprising "impedance match": *Name resolution* in ICN is a special case of *expression resolution*. We implemented a general λ-expression resolver, identified all memory accesses, and translated them directly to the pub/sub primitives of an ICN substrate.

This leads 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 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 well established, the network is in charge of moving data and code around and needs a way to orchestrate these moves.

Contribution: We show that the resolution strategies for λ-expressions solve exactly this orchestration problem. What we further explain in this paper is how to interlace name resolution and ICN forwarding strategy. We start in Section II with known scenarios where in-network code execution leads to better solutions than edge-only approaches, and describe prototypical orchestration problems that NFN must support. Section III describes our architecture and implementation and shows how we can let names interfere with ICN routing. Three application examples follow in section IV, as well as a discussion and related work section. Throughout the paper we assume the reader is familiar with CCNx' Interest (request) and Content (reply) interaction scheme [8].

II. Motivation and Approach

Designing an information centric network as a passive content delivery infrastructure still leaves operations on data to happen "at the edge" and precludes an active role of the network beyond forwarding and caching. However, let us (re)visit three cases where the network can easily capitalize on computation tasks taken over from edge systems.

Cloud-style data transformation: Content is generated in many forms (formats, volumes, quality, etc), and clients need various transcoders for different devices and different uses. Instead of straining the content source to match an unbounded set of client requirements, the network can simplify things:

Each client expresses its requirements by *composing names* in the form

```
/name/of/EnglishToMandarin( /name/of/document )
/name/of/FarenheitToCelcius( /name/of/sensor )
```

to let the network locate code, data, and an execution place. Moreover, transformed content could be cached, avoiding duplicated efforts upon repeated requests.

Conditional information retrieval: Filtering data is a known task which is more efficiently performed in the network, rather than letting clients download all possible data or having a source to service all possible filter requests. Clients can express filters in programs and let the network decide the best course of action. At an extreme filtering case, a client may wish to retrieve information local to only one node despite the ICN trying to shield locality:

```
ifelse (eq localContentStore.id 987)
  localContentStore.size
  nil
```

This program "names" a local piece of information, namely the number of entries in content store with id 987. Although the request spreads to several nodes, only one will match the condition and respond.

Data fusion: Occasionally, an information centric network may also take over the role of a database and might have to aggregate information. The following code would return the world's population by "special arrangement" of data and function names involving list traversal, using a recursive function definition:

```
(define count(acc, list) (
  ifelse list.empty
    acc
    count(acc + /yahoo/getpopulation(
        list.head), list.tail)
))
count(0, /uno/listOfCountries)
```

In this "composed name" the function definition is provided by the client, while the population lookup function stems from some third party and the list of countries from a trusted data source.

The last example also highlights the trade-off between caching and generating content. Once some information has been computed, other clients do not trigger the same computation but get the cached result. In the case of volatile data sources, results would need to expire rather quickly, while a transcoded media format can remain cached as long as there is space. This trade-off between memory and re-compute cost is a basis for network-internal optimization which none of the edge nodes, client or source, would be capable of doing.

A. Orchestrating (Function) Names in an ICN

We reduced the general problem of controlling data and code movements to three prototypical scenarios that an ICN will be faced with when attempting to work on programs like the ones presented above.

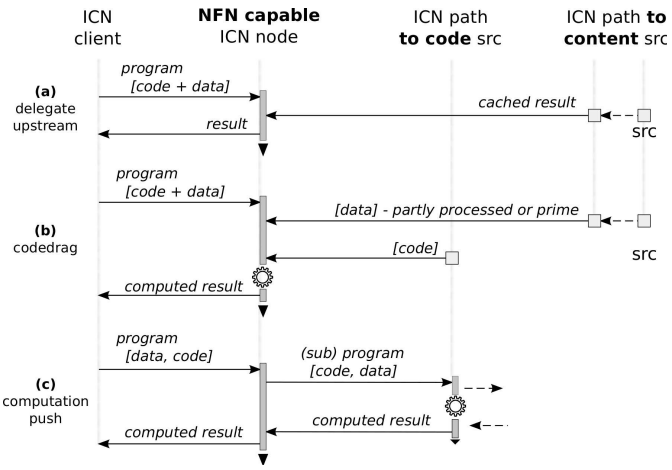

Fig. 1. Three scenarios that NFN must handle: upstream fetch, separate code and data fetch, computation push

Given a complex content processing request, case a) of Figure 1 avoids recomputing or refetching information that exists elsewhere in the net. Case b) applies when 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) task is delegated (pushed) to the pinning site.

These three cases cover only simple tasks and by themselves could not deal with the complex program examples given above. It remains to show how general name composition with λ-expressions can map to these three cases. Two major challenges are (i) to find a way to map arbitrary programs to ICN names and (ii) to develop some strategy for orchestrating the decomposition of complex names to computable and routable subterms. We assume two levels of programming languages: One is the untyped λ calculus used for "name shuffling" and computation coordination which is the focus of this paper. The other level (not discussed here) relates to bytecodes or similar mobile code technology that serves traditional code execution once an execution site is identified.

III. Architecture and Operations

We implement NFN by architecting the *NFN resolution engine* on top of an ICN system and by introducing network names for special constructs that give us access to intermediate computation results. The key element of the latter are nameable *computation configurations*.

The NFN system uses the ICN substrate as a storage service for its internal data structures, as well as for intercepting client requests and to return (computed) results. In NFN's untyped λ-calculus implementation, a computation configuration is a 4-tuple `<E,A,T,R>` where

- E name of an environment
- A name of an argument stack
- T term
- R name of a result stack

In that tuple, an environment E is a dictionary associating a term's *bound variables* to *closure names*. A closure is a 2-tuple `<E,T>` associating an environment E with a term T, and serving as a context pointer during *beta reductions*. The *argument stack* is a sequence of closure names and is used to hold intermediate state during the computation of a term. All operations of the NFN resolution engine are defined as op-codes over the argument stack. A *term* is a sequence of NFN-specific instructions, some of them parametrized with λ-expressions. Finally, the *result stack* is a normal operand-stack for interfacing with higher level languages and library functions (i.e. holding function *side-effects* during λ-expression evaluation).

Instances of these data types are kept in the ICN's content store by name. In the following we adopt an ASCII representation for these data structures[1] and use '`|`' to delimit positional fields. For example, the 4-tuple configuration `<E,A,T,R>` is encoded as a string `"CFG|e|a|t|r"`.

To illustrate how computations progress, assume that the following name bindings are known (stored) in an ICN:

```
name1 ~ ENV|true|name2
name2 ~ CLO||(\x\y x)
name3 ~ AST|name2|nameOfTail
name4 ~ RST|nil|0
```

and consider the configuration

```
CFG|name1||true 1 0|name4
```

which denotes that the λ-expression `(true 1 0)` should be evaluated in environment *name*1 using result stack *name*4. No argument stack is given (which would go at position 2). In the referenced environment *name*1, a mapping can be retrieved of expression `true` to closure *name*2, which gives the term $(\lambda x \lambda y.x)$[2]. Applying this term to the two initial parameters `1` and `0` results in selecting value "1" which will be stored in an updated copy of the result stack under *name*5. The new configuration of the reduce term is stored under *name*6:

```
name5 ~ RST|1|nil|0
name6 ~ CFG|||name5
```

A. The FOX instruction

Computation results are cached by storing configurations under a canonical name using a hash function. In the example above, this work as follows: Reacting to our request to resolve the start configuration `CFG|name1||true 1 0|name4`, the NFN engine will compute the canonical hash value h_1 of that configuration and consult the ICN substrate. If someone already computed the expression, there will be a trace of the form

```
h1 ~ LOOKUP(name6)
```

where `h1` is the hash value of the string `"RESULTOF(CFG|name1||true 1 0|name4)"`. The NFN engine will invoke an internal primitive called `FOX` (find-or-execute) which attempts to find the result of a given computation request before starting the computation. If a trace of the result is found, the NFN engine resolves the found indirection hash and returns the result. If a result is not found, the NFN engine launches the computation, stores the result under the indirection hash, and returns `CFG|||name5`, too. This caching process works for all intermediate results, permitting to restart an orphaned computation at any moment. In section III-C we explain how the `FOX` instruction links lambda expression reduction to an ICN's resolution of names.

[1]One might trade this ASCII representation for a less readable but more efficiently parsable encoding.

[2]We use the ASCII character '\' to encode the λ sign

B. The NFN Abstract Machine

Our NFN engine follows the approach of "Krivine's machine" [10] which resolves λ-expressions through a call-by-name strategy. Instead of directly implementing it, we implemented a resolution engine that is similar to Caml's ZINC abstract machine, called ZAM [11]. A resolution step consists in beta-reducing a given λ-term according to its syntactic structure, which can be either a variable lookup, an abstraction, or a function application. To this end, the NFN abstract machine features a resolve-by-name instruction `RBN` which is implemented through three different sequences of selected ZAM instructions and (recursive) RBN calls:

Call	Case	Rewrite as
`RBN(v)`	Var	`ACCESS(v);TAILAPPLY`
`RBN(\x body)`	Abstr	`GRAB(x);RBN(body)`
`RBN(f g)`	Appl	`CLOSURE(RBN(f));RBN(g)`

For completeness, we document the referenced ZAM instructions: Given a current configuration `<E,A,T,R>`, the semantics of the ZAM instructions are as follows:

`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.

Extending Krivine's machine we also added instructions to manipulate values on the result stack (which are not part of the standard Krivine's machine). For example, the `add` function is implemented by binding it in the outermost environment E_0 to

```
RBN((\op\a\b (a b op)));OP_ADD
```

Given an expression `(add 3 4)`, this function first evaluates the summands `a` and `b` –which end up on the result stack– before calling the built-in instruction `OP_ADD` that pops theses values and pushes their sum.

C. Combining Expression Resolution with Forwarding

So far our λ-expressions appear in NFN-internal data structures only. To spread the resolution of programs in an ICN network we have to map them to that ICN's specific naming scheme. We picked CCNx [8] as an instance of such a network and formulated a combined resolution and routing strategy that we explain in this section. In our scheme, the λ-expression

```
f(g(dat))
```

will be mapped in inverse order to a CCNx name's components like in

`[ccn:nfn|/name/of/dat|/name/of/g|/name/of/f]`

Note that the components of this NFN name are themselves names.[3] The term inversion for CCNx' wire format has to do with CCNx' longest prefix-match forwarding philosophy (that we had to slightly generalize). Assume we have a chain of function applications `f(g(h(data)))`: The inverted `[... |data|h|g|f]` encoding ensures that – when we inquire for the full expression but only intermediate results are available – partial results can be retrieved from the network along the path to the data, letting the network test first for `|data|h|g|f`, then `|data|h|g` etc. Figure 2 gives an overview of the resolution and routing strategy for the combination of NFN and CCNx; the three phases will be explained in the following paragraphs.

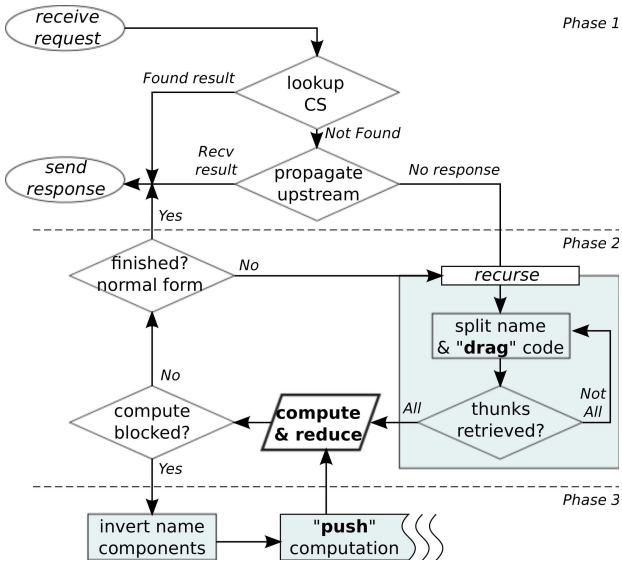

Fig. 2. NFN combined resolution and forwarding strategy for CCN.

Phase 1: Search by upstream delegation: The first phase, which covers case a) in Figure 1, is the currently default forwarding strategy in CCN: When an Interest is received, a search in the local content store (CS) may resolve the request and satisfy immediately the pending Interest.

If the search fails, the Interest is propagated further upstream according to the entries in the FIB. Since the leftmost component in the name that encodes a program refers to the innermost term of the expression, this is the target content for manipulation and due to prefix-based forwarding, the Interest program travels towards the content source. This means that by default preference, the responsibility for the computation of the result is *closer to the source* (by contrast to the preference that the response is cached *closer to the receiver*). This is useful because it increases the probability of caching (cooked) content as well as intermediate results in the middle of the network.

Phase 2: Search and Reduce "by name": If the search for the complete name times out, the strategy enters the second phase and seeks in turn intermediate results that would enable it to complete the computation locally.

Recursively, starting from the outermost level of the expression, it will search in the ICN for intermediate results of the inner terms (sub-expressions), and in parallel search for the code of the outer term. This is done by tail-trimming the last (rightmost) component of the name in the received Interest and searching for each of the two parts separately. If the search fails for an inner term, the strategy recurses deeper in the inner sub-expression, until all functions and the content are requested independently (implying that no intermediate results were found).

Requests for intermediate results of inner terms or function code in fact retrieves only *thunks* (names of configurations). One can think of a thunk as a contract that the respective term can be made available, if needed. When all thunks become available the reduction of a pending λ-expression can progress. Since in this process some terms might naturally disappear or canceled, the respective thunks will never be executed to retrieve the code/data from the network. At this point the strategy has implemented a *call by name* policy, which corresponds to case b) in Figure 1.

Phase 3: Inverse Semantics and Reduce "by value": It can be that either code or the required content cannot be obtained for some reason (no mobile code available or blocked content distribution), but that for example the code owner would be willing to execute the function for approved data. In this case phase 2 will not successfully complete. Phase 3 of the routing strategy will take care of this case which corresponds to scenario c) in Figure 1.

In λ-calculus the evaluation position of the terms affects the semantics in the order that reduction steps are performed. Given the 1:1 mapping of terms to name components in the Interest carrying a program, and also given the use of names for prefix-based routing/searching in CCN, this means that by changing the evaluation order of the λ-terms we can influence the forwarding of the expression in the network.

If all else in phase 2 fails we take advantage of this feature to swap the positions in the Interest name of the first components to effect the forwarding of the computation towards the code source. Computationally, the effect is that we have also switched the resolution strategy from *call by name* to *call by value*, which means that if at the remote end the component order is not recovered the reduction sequence will be different. But, due to the *confluence* theorem in λ-calculus, the final normal form of the reduced expression will be the same. Note that the strategy can resort to the 3rd phase either for the computation of the entire expression or any sub-term during the recursions of the 2nd phase.

[3]We use '[...]' brackets to enclose a CCNx name and '|' to separate its components. Moreover, this notation is used recursively, when a name component is itself a CCNx name.

IV. Proof of concept

To illustrate the concept of named function networking we give three examples of increasing sophistication.

Example 1: CCNx object selection

In CCNx, a client can request content that not only matches in the name but also in the data object's checksum. Using NFN, this functionality can be implemented at name level, leading to a simplification of the base CCNx protocol. The following program features a λ-expression with an anonymous function having two parameters and this function being applied to an object name and an integer value, which overall we will treat as being a single NFN name:

```
(\obj\hval
   (ifelse (eq hval (sha256 obj)) obj nil)
) objName 42
```

The purpose of this name is to only return content that matches the CCNx name `objName` and that has a matching sha256 value (`42` in this example). In our prototype this "program resolution" translates to 86 ZAM instruction steps. As mentioned above, future requests will access the cached (=computed) content without triggering a re-computation: The "program" has become the name of the content as if it were a passive name.

Generally, using such NFN names, a client can select content based on a broad range of criteria, which can also replace CCNx' (complex) filter specifications.

Example 2: Code drag

Consider an analogous but more complex program as those presented in Section 2, case 1 (data transformation):

```
/util/compress/zip(
    /codec/mpeg4( /name/of/media )
)
```

whose encoding as a CCNx name is

```
[ccn:nfn|/name/of/media|/codec/mpeg4|
                              /util/compress/zip]
```

After the NFN node fails in phase 1 to retrieve the full or partial result for the given expression name, it will fall back to phase 2 and try to compute the expression locally. To this end, it breaks up the original expression and independently (re-)searches for the partial result of the transcoded content as well as the compression code:

```
/codec/mpeg4(/mame/of/media) →
    [ccn:nfn|/name/of/media|/codec/mpeg4]

/util/compress/zip →
    [ccn:nfn|/util/compress/zip]
```

The two resulting Interest messages, as their CCNx name prefixes imply, will be forwarded in different directions in the namespace. In case of failing to retrieve the partial result for `/codec/mpeg4(/name/of/media)`,

the node will further start searching independently for the two inner terms

```
/name/of/media → [ccn:nfn|/name/of/media]
/codec/mpeg4 → [ccn:nfn|/codec/mpeg4]
```

Again, the two searches will follow different namespace forwarding paths, the second one aiming at fetching the function's mobile code. In the end, if this phase yields all necessary results, the NFN node will have managed to *drag and cache* locally all the terms needed to compute the result. It runs the code, caches the result and satisfies with it the original Interest.

Example 3: Computation push

The following example involves branching and exposes the challenge when a function is "pinned down" (is not-transferable). Assume a Digital Rights Management setting where a certification authority only allows the deployment of some transcoder code upon verification of the content author's signature. The authentication code itself is not mobile and so authentication has to be completed at the certifying authority' nodes (which may also be the actual transcoder code owner). A program that would tie together media, transcoder and verification routine could look like

```
(define playmedia(c) (
   ifelse (/ca/auth c)
     (/codec/mpeg c)
     nil
  )
  playmedia( /the/media )
)
```

Let us "drill down" to that subterm that creates a special problem, wherefore we show the NFN's internal rewriting steps (the terms that we do not show in this writeup are taken care of by phase 2 reductions):

```
(\x((ifelse (/ca/auth x) /codec/mpeg4 nil) x)) /the/media
(ifelse (/ca/auth /the/media) /codec/mpeg nil) /the/media
(/ca/auth /the/media) /codec/mpeg nil
(/ca/auth /the/media)
```

According to the explanations in section III-C, this last subterm is mapped to

```
→ [ccn:nfn|/the/media|/ca/auth]
```

which exposes the problem: The media owner site, to which this name is forwarded, will not be able and allowed to execute the pinned down authentication code. The solution is provided in phase 3 of the forwarding strategy. Reversing the call-semantics (call-by-value instead of call-by-name) by transposing the top level components in the CCN name, plus adding a "reversed-flag" ><, yields:

```
→ [ccn:nfn|/ca/auth|><|/the/media]
```

This Interest request will now be forwarded (pushed) towards the site of the authentication code, gets executed and it will return `True` (or `False`), which in the untyped λ calculus is represented by `(\x\y x)`. When received, the resolution proceeds as

```
(\x\y x) /codec/mpeg nil) /the/media
/codec/mpeg /the/media
```

and thus reaches the desired result.

At this point one could explore alterations to our resolution strategy which relate to call-by-need, the use of thunks and

continuation passing style. From such techniques we have so far only implemented tail recursion in order to execute recursively defined functions.

V. Discussion

Although our implementation focus has been limited to the CCN architecture, the essence of our contribution is scoped with information-centric networking in general. It remains (as part of future work) to conceptualize the embodiment of NFN in other ICN architectures, such that any ties which are specific to CCN are challenged.

One important such tie is the hierarchical namespace structure of CCN, that leverages prefix-based forwarding and which may come at odds with the flat naming schemes of other ICN architectures [9], [4], [14]. Particularly interesting is the case of NetInf [4], which manifests in many aspects as an information centric *meta*-architecture. In this respect, although it proclaims the use of flat naming, it seems possible to adopt various schemata that attach some structure to the flat identifiers (among others the one of CCN). Akin to such schemata, various name-based forwarding strategies can be introduced. In theory, the implication of such flexibility in the context of NFN would be that different expression-resolution strategies pertain through forwarding to different network-side optimization objectives (i.e. how the network orchestrates expression-resolution to match dynamics and path availability). However, before any such assumptions can be validated, a concrete mapping between named expressions and NetInf flat names needs to be developed.

Current progress in NFN has involved the development of the resolution machinery and the casting of interactions with the ICN substrate (interest/content interaction scheme), leaving still open a number of questions adhering to networked operation. One of them is how should timeouts be chosen such that "optimization efforts" do not eat up all caching benefits and unavailable results are computed without undue lag? Our work so far cannot answer such dynamics aspects. Similarly, the call-by-name resolution strategy generates a lot of intermediate state (updated binding environments, modified argument stacks and transient configurations) which is written into the ICN substrate and often retrieved only once, necessitating some form of garbage collection.

Another aspect (albeit common to all named-based ICN architectures) is the length of names in data packets, which is made even worse in our case. However, as the `FOX` primitive uses some canonical hash, popular content will be found with minimal space overhead, but could require routing on flat labels (which brings us square rooted back to the potential benefits of other ICN architectures). A mitigating factor is that lambda expressions are resolved as much as possible through call-by-name semantics, leading to a minimum number of ICN queries to be forwarded, each involving shorter names. Also in regard to the issue of long names, are the effects of different fragmentation schemes (cut-through versus hop-by-hop versus end-to-end) in the decision of "where" an expression gets resolved.

A security concern well known since active networking research, is the potentially uncontrolled use of computing and forwarding resources: In our case the obvious problem are divergent or looping programs. We consider the use of "resolution credits" (that a client has to obtain from the net) carried in the interest messages, similar to work done in PLAN [7] (where these TTL-tokens were assigned in a static way). The second critique analogous to active network research is dependability: Injection of malicious code, preventing insertion of or cleaning up caches from bogus results, trusting that hosts faithfully execute the provided code or do not leak private data, are but a few obvious security problems. We did not investigate yet these corners, but point out that they are heavily mitigated or even vanish e.g. in private clouds (which points to seeking policy-related solutions as by routing policies).

Last but not least are the harmonization of computations with routing policies, and the effects of different caching strategies in the trade-off between computing and searching. Such issues cannot be answered before larger scale experiments with NFN are possible, and even more before some maturity of CCN or other ICN architectures has been reached.

An important future work item for NFN is how its functional flavor can orchestrate network-wide computations based on the Map-Reduce (MR) paradigm. MR, or Map-Fold as it is often called in functional languages, is inherently supported by NFN: The key role of scheduling mappers and reducers in NFN is taken over by the network's forwarding strategy. As the combined expression resolution plus forwarding strategy is still on-going work, MR will help the assessment of its effectiveness and understanding of possible extensions that we will need to take into account.

VI. Related work

Our ideas in this work lay somewhere between the holly grail of an all optical Turing switch [3] where all nodes are universally programmable forwarding elements, and Borenstein's AtomicMail from 1992 [1], who incidentally used LISP for "programming emails" as questionnaires collecting answers from a pre-defined set of recipients (letting the email itself decide on the journey to take). And yet, more broadly speaking, our ambition comes closer to Sun Microsystems' slogan from the 1990s: "The network is the computer".

In the past, Active Networking (AN) research for more than a decade envisioned that users could load programs [18] into the net, that the net would provide richer programming primitives [5], [15], or complete programming (language) frameworks, eg. [7], [13], of which some of them were in fact functional. However, explicit end user handling of packet-level code and network locations, or restricted code deploying strategies like code-follow-capsule [17], are low-level restrictions that an information-centric approach like NFN overcomes by addressing programmability at a much higher level. At the same time, in NFN, the scope of programmability is also beyond configuring links or data paths and processing individual packets, by contrast to the modern incarnation of

programmable networks as in SDN [12] and thereby related programming environments [16], [6].

In the context of ICN, a similar idea of assigning names to services has appeared in Service Centric Networking [2], where a concatenation of CCN names corresponds to a processing workflow (at the input of which lies the data source) that satisfies a service request. The workflow may be instantiated inside the network as the pieces of the concatenation are valid CCN names, matched against functions running in execution environments, which sit next to selected content stores. This approach – despite the conceptual similarities to NFN – is taking advantage of ICN to improve and optimize over the workflow deployment paradigm of Service Oriented Architectures, but does not come to the level of coupling programming constructs to names in the strict sense of a Turing-complete language and engage the forwarding process in them. Although it maps names to functions (by analogy to our λ-variable lookup), names are not expressions of program logic (subject to manipulation as in beta reduction), which is the key to letting the network "orchestrate the game".

Another work that has looked in the aspect of composing services inside the network in the context of the pub/sub paradigm (and thereby relevant to ICN), is in the PURSUIT project [14]. Here the relevance to NFN is the dynamic/transparent decision of "where" to compute in the network. But the similarity also ends there: In NFN, this decision is the result of network conditions and forwarding state (allowing the network to optimize against them), while in PURSUIT it is the result of searching for rendezvous points where the components of a service have been prior-published. In regard to programmability, the PURSUIT approach is declarative (service specifications analogous to SoA or many service deployment frameworks in Active Networking) and therefore does not engage the network in any computation.

VII. Conclusions

We presented a Named Function Networking approach that re-interprets the request-content-by-name primitive of classic ICNs as a lookup of a variable. We embed this basic lookup operation into the more general context of λ-expression resolution: By adding support for function application and function abstraction, the network becomes a Turing-complete computing device that orchestrates the transfer and execution of code and data and that caches corresponding results in the most optimal way. The so called call-by-name resolution strategy for λ-expressions, for example, resolves names to their actual value only when actually needed in a program flow.

In this paper we have also shown how call-by-name resolution can be combined with a content centric routing strategy and how this enhanced network is capable of finding suitable execution places for named functions. We believe that letting clients request content by functions answers much better their need for *data use*, instead of merely shipping the data. At the same time, letting a name stand for either some data or a function, we have an indirection mechanism where a content producer as well as the network itself can both influence and optimize the way content processing and delivery is carried out.

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, A. Mauthe, and V. Siris. Service-Centric Networking Extensions. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 583–590, New York, NY, USA, 2013. ACM.

[3] J. Crowcroft. Turing Switches: Turing Machines for All-Optical Internet Routing. Technical Report UCAM-CL-TR-556, Cambridge University, January 2003.

[4] C. Dannewitz, D. Kutscher, B. Ohlman, S. Farrell, B. Ahlgren, and H. Karl. Network of Information (NetInf) - An Information-Centric Networking Architecture. *Comput. Commun.*, 36(7):721–735, Apr. 2013.

[5] D. Feldmeier, A. McAuley, J. Smith, D. Bakin, W. Marcus, and T. Raleigh. Protocol Boosters. *IEEE Journal on Selected Areas in Communications, Special Issue on Protocol Architectures for 21st Century Applications*, 16(3):437–444, April 1998.

[6] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. FRENETIC: A Network Programming Language,.

[7] 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.

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

[9] T. Koponen, M. Chawla, B. Chun, A. Ermolinskiy, K. Kim, S. Shenker, and I. Stoica. A Data-Oriented (and beyond) Network Architecture. *SIGCOMM Comput. Commun. Rev.*, 37(4):181–192, Aug. 2007.

[10] J.-L. Krivine. A call-by-name Lambda-Calculus Machine. *Higher Order Symbol. Comput.*, 20(3):199–207, Sept. 2007.

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

[12] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *Comput. Commun. Rev.*, 38(2), Mar. 2008.

[13] S. Merugu, S. Bhattacharjee, Y. Chae, M. Sanders, K. Calvert, and E. Zegura. Bowman and CANEs: Implementation of an Active Network. In *37th Allerton Conference on Communication, Control and Computing,*, Monticello, IL, September 1999.

[14] PURSUIT - Architecture Definition, Component Descriptions and Requirements. Deliverable D2.3 at http://www.fp7-pursuit.eu, 2011.

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

[16] 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.

[17] D. Wetherall, J. Guttag, and D. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *IEEE OPENARCH*, April 1998.

[18] J. Zander and R. Forchheimer. SoftNet - An Approach to High-Level Packet Communication. In *2nd ARRL Amateur Radio Computer Networking Conference*, 1983.