A Packet Rewriting Core for Information Centric Networking

Christopher Scherb
University of Basel
Switzerland
Email: christopher.scherb@unibas.ch

Manolis Sifalakis
University of Basel
Switzerland
Email: sifalakis.manos@unibas.ch

Christian Tschudin
University of Basel
Switzerland
Email: christian.tschudin@unibas.ch

***Abstract*—Although very similar, Information Centric Networking implementations like CCNx or NDN have not converged (yet). Moreover, high-level extensions like named functions and low-level considerations for IoT or logical link control will lead to even more variety in formats and experiments, also including slight (and not well-documented) changes of the ICN protocols' behavior.**

Based on our experience with the multi-protocol implementation CCN-lite, we have developed a domains-specific intermediate language ICCL (Information Centric Core Language). Initially, the purpose of ICCL was to document the common as well as diverging aspects of the various approaches *and* to provide a run-time which can directly execute an ICCL program. But as it turned out, ICCL became a tool beyond modeling the behavior of ICN forwarders. For example, by including appropriate primitives to access local data sources, one can now seamlessly include in an ICCL program the access to data repositories or serve on-demand data from sensors. This enables easy programmability of network functionality and contributes to "do-it-yourself" networking. We discuss the benefit of this fusion for the development of ICN functionality and report on performance measurements for our prototype system.

I. Introduction

CCN-lite [1] is an independent ICN implementation of the Named-Data-Networking (NDN) and of PARC's CCNx protocol. Over the years, while CCN-lite was tracking the changes and addendums and refactored its code, it became clear that a majority of routines could be reused for "competing" architectures and that a considerable body of functionality remained rather stable. One example is the debate of the Type-Length-Value encoding which in CCN-lite boiled down to a template-like code structure where new encoding variants could be implemented.

The second observation was that code written for the forwarder logic often reappeared inside applications i.e., outside the forwarding layer. An example is the pending interest table (PIT) data structure which a client also has to implement in order to track lost packets.

Starting from the first observation we have worked on an intermediate domain-specific language [2] that we called ICCL (Information Centric Core Language). The goals of ICCL were:

- to provide a "spec-like" abstract description language for CCN, NDN and other variations, documenting commonalities as well as differences at high level,
- to be executable in order to easily emulate or extend existing and to experiment with novel ICN architectures,
- to easily create in-network applications such as Web servers or workflow services (e.g. Named Function Networking (NFN) [3]).

With the observation that application programming needs seem to overlap considerably with those of programming a forwarder, we then started to extend ICCL to support applications, mostly by including I/O primitives. The result is a programming environment for both the application and the forwarding layer.

Described in a pictorial way, ICCL has become the base layer for **ICN packet rewriting**, ideally turning an incoming Interest request into Content reply. On top of this core, and placed side-by-side, forwarders (both CCNx and NDN) can be expressed and executed equally well as a repository serving static content or a compute server producing data on demand. Furthermore, ICCL provides primitives to manipulate list data structures which can be used for batch requests of data like collecting sensor data on the sensor itself and transferring all data at once after receiving a request from a controller. For example, this could be useful for implementing sensor networks which monitoring water supply and other critical infrastructure with less programming complexity.

In this paper we demonstrate a simple Web-server like app written in the ICCL language. While such a service for itself is not novel at all, we use it to demonstrate the tight integration of Interest/Content packet handling with issuing SQL code.

The structure of this paper is as follows: First we give an overview over existing ICNs, next we will show details about ICCL and two use cases, we already implemented.

II. Information Centric Networking Flavors

Information Centric Networking (ICN) is an alternative concept to the traditional connection oriented network system. Instead of connecting machines, an ICN network connects content. Each content (data) object can be "addressed" by using a *name* (instead of a server) in order to abstract away the data's location. When clients request some content, routers can serve it from any cached copy that might exist in the network. The binding between a name and its associated data is secured by a cryptographic signature added by the producer.

978-1-4673-9292-1/16/$31.00 ©2016 IEEE

This permits to validate that the delivered content actually belongs to the name used to retrieve it.

A. CCNx and NDN

Mandating that each content object has its own name might lead to giant Forwarding Information Bases (FIB) wherefore Jacobson [4] proposed that his CCNx architecture uses hierarchical names. Requests for content, called Interest messages, will be forwarded through these FIBs towards the data source. Each interest message contains the name (identifier) of the requested content object: It is the clients responsibility to request each data item independently in a receiver-driven fashion (while in TCP, it is the server sending the data according to its own pace).

There are three important data structures in an ICN node: The Forwarding Information Base (FIB), the Pending Interest Table (PIT) and Content Store (CS). The FIB stores forwarding information, the PIT is used for interest deduplication and the CS is the cache on a router. If an interest message arrives, a router first checks if it is possible to serve the request by using the local CS. If not, it checks if there already is a PIT entry available, which means that the interest was already forwarded. In this case the request is added to the existing PIT entry. Otherwise, the node will create a new PIT entry and forwards the interest by using the FIB.

In the past years, Xerox PARC's CCNx project forked which has led to two incompatible ICN architectures. Although at a very coarse level they offer the same data lookup service, they now have major differences. The follow-up NDN project [5] has more or less sticked to the original CCNx profile. But the "new CCNx" (subsequently labeled CCNx1.0) has opted for a radical streamlining: Most prominently, the network will do *exact match* on hierarchical content names instead of longest-prefix match; moreover, selectors (for filtering and namespace exploration) are completely eliminated except for two "restrictions" called ObjectHash and ProducerID. The argument of the CCNx1.0 team is that this minimal base level, which is just a pure named-based forwarding service, is a sufficiently rich base-line *on top* of which all previous features can be implemented. Other incompatibilities relate to the encoding of the message types. To support domains with special (packet size) constraints, other ICN flavors like CCN-IoT were derived from CCNx and NDN, introducing even more packet formats.

III. ICCL - A Domain Specific Language for ICNs

A Domain Specific Language (DSL) is a programming language designed for a specific usage [2]. The most popular example for a DSL is HTML and is used to describe how to render text elements. The domain for which we designed a language for is the programming of ICN network elements.

Our DSL called "Information Centric Core Language" (ICCL) is a functional language designed for general programmability and extendability and has *no* network specific functions in its grammar: It is just an extended form of the λ-calculus. We chose the λ-calculus as basis since there already exist expendable execution environments (abstract machines[6]). Moreover, it is free of side effects which matches well the retransmission logic in case of lost request *and* result packets. ICCL permits to invoke "built-in" functions that are more less the same as the functions defined in a standard library of a programming language like "C". Of course, users can also define their own functions in ICCL. The main goal of ICCL is to emulate specific ICN architectures, for example to express different forwarding strategies or name handling logics, or as a basis for programming network applications.

The ICCL Execution Model: A specification language that should capture content-based packet processing logic must come with a suitable abstract execution model. With ICCL we chose to describe a node's behavior through a concurrency model over a finite-state description. In this section we describe this choice and the implementation philosophy that we applied when we cast existing architectures into our framework.

The basic operation model is that a node has two handlers: One for upstream (request) and another for downstream (reply) packets. Should other packet types be envisaged (e.g. ACK-/NACK instead of CONTENT) we would provide additional handlers, but otherwise our scripts all have the same structure based on a `onRequest`and a `onReply` handler, as visualized in Figure 1.

Each incoming packet induces an independent thread that executes that handler's code. The code has access to the node's shared data structure, for which we provide a basic (and sometimes architecture-specific) API. Typically, these shared data structures would be PIT, FIB, as well as routines to propagate requests upstream or send back replies downstream.

In all the code examples given below we adopt a request/reply pattern and assign responsibility over a request to the onRequest thread only. A reply thread will have a short chance to process the packet but we typically restrict it to signal the waiting request thread that will continue with the request processing. The synchronization concept is the one of UNIX where threads can block on a condition with `wait(cond, TIMEOUT)`. The opposit primitive `signal(cond)` will unblock **all** waiting threads (and is a null action should no thread be waiting). If a requesting thread times out, or is woken up, it checks for results and if existing then it continues with replying otherwise it simply ends (or generates a NACK etc).

In our ICCL model, the FIB and PIT tables rely on special indexes: The entries in the PIT and the FIB, but also the conditions, will be "named" by attribute sets. These sets are trivial for CCN and NDN, for example the set of {`contentName="/a/b/c.txt"`}, but can be as general as in [7] where sets of predicates are used.

As a toy example we sketch the code structure for a CCN/NDN forwarder. Their (one-time) Interest/Content pattern naturally maps to the onRequest/onReply mechanics. Note that "self" refers to the incoming packet:

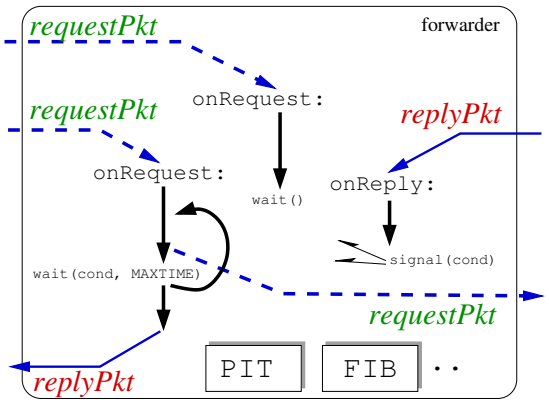

Fig. 1. Request aggregation example: The first request thread is in charge of retransmissions; the followup thread is reactivated, too, when the reply thread signals that condition.

```
1  handler onRequest(name) (  // interest pkt
2    checkContentStoreAndReturnIfFound(name,
         self) else
3    checkPITandPropageAndWaitThenServeDownstream
         (name,self)
4  )
5
6  handler onReply(name) (    // content pkt
7    checkValidityAndStoreAndWakeup(name, self
         )
8  )
```

It is not surprising that this is also the structure chosen in the original CCNx document which dedicates one paragraph each about these two handlers.

A. Capturing ICN Functionality in ICCL

In this section we "de-construct" the *specific* CCN and the NDN pipeline in terms of ICCL. In a first step, we have to add some built-in functions for forwarding purpose. The most important operation is the sending of interest and content messages to some existing "face" (which is a generalized interface). Faces are also used in PIT and FIB entry manipulations (see Section II-A) like `existsInPIT()` and the like. Similarily, the content store (CS) has its own primitives. Table I provides a list of these essential (and some more) ICN-related primitives.

For clarity purposes we differentiate between checking whether an entry in a data structure is available vs the grabbing of an actual entry. Finally, in order to forward the entire packet, we add the command `self` to refer the packet itself, while `inFace` refers to the face where a packet was received. Having presented these constructs, we can now program a first, basic forwarding pipeline – and will discuss more complex tasks like selectors in Section III-B.

In NDN, the forwarding of interests relies on longest prefix matching. The program in Figure 2 shows this NDN forwarding behavior. Note that `ifelse` expects a boolean expression and two code blocks.

TABLE I
BASIC ICN-RELATED BUILT-IN INSTRUCTIONS OF ICCL

Function	*Usage*
`sendInterest name face`	send an interest message
`sendContent contentObject face`	send a content message
`existsFIBexact name`	check if there is a FIB entry available using exact prefix matching
`existsFIBlongest name`	check if there is a FIB entry available using longest prefix matching
`grabFIBexact name`	take an entry from the FIB using exact prefix matching
`grabFIBlongest name`	take an entry from the FIB using longest prefix matching
`addToFIT name`	add a new entry into the FIB
`existsPIT name`	check if there is a PIT entry available
`removePIT name`	remove an entry from the PIT
`addToPIT name`	add a new entry into the PIT appends if the entry is already available
`refreshPITTimeout name`	update timeout of a PIT entry
`existsCSexact name`	check if there is a CS entry available using exact prefix matching
`existsCSlongest name`	check if there is a CS entry available using longest prefix matching
`grabCSexact name`	take an entry from the CS using exact prefix matching
`grabCSlongest name`	take an entry from the CS using longest prefix matching
`addToCS name contentObject`	add a content object into the CS
`wait name timeout`	put computation in the waiting state wait for a content objects with a specific name and for a specific time (in ms)
`signal name`	signal all interests waiting for a specific name
`seq`	sequence of commands
`exit`	quit a computation

```
1  ifelse boolExpr
2  block1
3  block2
```

There are to different entry points to ICCL programs: The `onInterest` and the `onContent` handler.

a) Interest Message Handling: If a interest message is received, the `onInterest` handler is called. It first checks if there already exists a content object in the content store CS (line 2). If the content object is cached, the node can directly reply to the data requester (line 3). Otherwise the program consults the FIB, hopeing to find an upstream node for this request (line 4). If a FIB entry exists, the handler checks the PIT (line 6) to see if the same interest message was already propagated. In this case the PIT entry's timeout is updated (line 8) and the thread waits for the content to arrive (line 9). Line 14 is executed if there was no PIT entry for the given name. In this case one is created (line 15) and the interest is propagated using the function `interestPropagate`.

The function `interestPropagate` transmit the interest several times (RETRYCOUNT). First it checks whether it exhausted all permitted attempts. In this case all other threads waiting for the same name are notified with a `signal` (line 23) and because no content is found they all terminate. Otherwise, the function searches for a matching FIB entry (line 27) and forward the interest message (line 27). Next the

```
1  handler onInterest(name, inFace) (
2    ifelse existsCSlongest(name)
3      sendContent(grabCSlongest(name), inFace
           )
4      ifelse not(existsFIBLongest(name))
5        exit
6        ifelse existsPIT(name)
7          seq(  // another thread leads
8            refreshPITtimeout(name)
9            wait(name, -1)
10           ifelse existsCSlongest(name)
11             sendContent(grabCSlongest(name)
                   , inFace)
12             exit
13         )
14         seq( // we are the first thread
15           addToPIT(name)
16           interestPropagate(name,
                 RETRYCOUNT)
17         )
18 )
19
20 function interestPropagate(name, count)(
21   ifelse eq(count, 0)
22     seq( // we give up
23       signal(name)
24       removePIT(name)
25     )
26     seq(
27       sendInterest(self, grabFIBLongest(
             name))
28       wait(name, grabPITtimeoutval(name))
29       ifelse not(existsCSlongest(name))
30         interestPropagate(name, dec(count))
31         seq( // finally the reply arrived
32           sendContent(grabCSlongest(name),
                 inFace)
33           removePIT(name)
34         )
35     )
36 )
37
38 handler onContent(name)(
39   ifelse existsCSlongest(name)
40     exit    // dup
41     ifelse not(existsPIT(name))
42       exit  // unsolicited
43       seq(
44         addToCS(name, data)
45         signal(name) // good news, content!
46       )
47 )
```

Fig. 2. ICCL implementation of NDN

function waits for a reply. If no reply message was received during the timeout interval, the function calls itself recursively with a decremented retransmission credit (line 30). If after waking up we find a reply in the content store, the thread will forward it to the requester, and so will the other threads which were reactived, too.

b) Content Object Handling: If a content object is received by a node, the `onContent` handler is called. It first checks if the content object is already cached (line 39, duplicate detection). If not, the node will check whether there is a PIT entry for this name: an unsolicited content object for which no PIT entry is available will be dropped, too. If, however, the PIT entry is available, the node will add the content object to the CS (line 44) and notify all interest threads which waits on this name (line 45).

Beside NDN it is also possible to express other forwarders in ICCL. For instance the forwarding pipeline of CCNx v1.0 is very similar to the NDN forwarding pipeline. In fact the main difference between both forwarders is that the Interest/Content matching in NDN is performed with longest prefix matching and in CCNx exact prefix matching is used. Therefore, we use the functions `checkCSexact` and `grabCSexact` instead of `checkCSlongest` and `grabCSlongest` in Figure 2. In this scenario we use predefined matching functions. A reason for this is to increase the performance of the forwarder. Nevertheless, it would be possible to define the matching function by using ICCL. Since ICCL is a Turing-complete complete language, it is possible to define own matching function. Thus, it is possible to match not only by using the name, it is also possible to match e.g. on the hash value of a desired data object.

B. Selectors

NDN as well as CCNx contain features to specify the requested content more precise than only by using names. Usually, a selector is an additional condition –beside the name matching– which either has to be fulfilled (includes) or must not be fulfilled (excludes). This condition is often defined on the metadata of the actual content object.

NDN support selectors to specify the request in more details. A client can *impose* a certain KeyID value (called PublisherPublicKey), and the digest of the requested data object (via an implicit, last name component). Thus, a client can ensure that it receives only data from a specific publisher. The other part relates to the *exclusions* which are useful in the context of NDN's longest prefix matching. The exclude selector specifies name components which a name must *not* contain. In ICCL we therefore add a `contain(name)` function to invoke such tests. A third kind of selectors in NDN are the "MinSuffixComponents" and "MaxSuffixComponents" fields which specify which of the name components should be matched. With this functionality it is possible to extract single name components out of a name and compare it with given component values.

In the following example we show how some of the NDN selectors could be applied (as we have not yet implemented all of the selector semantics).

```
1 function matchContent(name, excludeComp)(
2   ifelse checkCSLongest(name)
3     ifelse contains(name, exludeComp)
4       false
5       true
6     false
7 )
```

In this example we test for a name and whether a component is available or not. An interesting aspect of ICCL and selectors is that it writes down the *order* in which selection is done. We expect interesting semantic subtleties when it comes to the implementation of the `leftmost` and `rightmost` selectors.

CCNx restrictions are a very similar feature to NDN Selector. In a CCNx interest message the client can specify the KeyID of the publisher and/or the hash of the content object which is requested.

C. A Service Implementation in ICCL

Beside the forwarder inside the network, there are always data and/or service provider at the edge of the network. By reusing existing code from the forwarder and providing a common programming interface, ICCL enables users to simply create customized network application, without deep knowledge of the network.

As example we use a data repository with provides data from a SQL database, but ICCL could provide an arbitrary service.

In our use case, we assume an ICN service which provides a document with customized information. These information are stored in a database. To access the database, ICCL is extended with a SQL binding and a functions to execute SQL statements. The important functionalities are listed in Table II.

TABLE II
BASIC SQL RELATED BUILT-IN INSTRUCTIONS OF ICCL

Function	*Usage*
`head list`	first element of a list or name
`tail list contentObject face`	list or name without the first element
`last list`	last element of a list
`sqlexecute database sqlqurey args`	execute a SQL query on a database returns a list with all results
`getField dbentry field`	extract a specific field out of a database entry
`createContent name data`	create a new content object invokes onContent

The function `sqlexecute` returns a list of all matching database entries. By using the `head`, `tail` and `last` functions it is possible to select a specific database entry and with `getField` a specific field can be extracted out of a database entry. To send the requested result over the network, the function `createContent` can be used.

A very simple example how to use this functionality is to create a simple ICN web server. We choose a very simple example to give an easy introduction and because of the very early state of our runtime environment. First we assume that all data on the web server are available under a given name. In our case this name is `/web/server`. All data on the web server have an individual name. This name is used as primary key in the database. If we want to request the data object with the name `index`, we express the interest message `/web/server/index`. This interest will be forwarded to the web server. The web server cuts the first components and searches in the database for a matching entry. The result will be stored in a content object and transfers it back to the requester. The web server will also maintain a CS to avoid duplicated queries.

In the following (Figure 3) we show how such a web server could be implemented in ICCL.

```
1  handler onInterest(name, inFace) (
2    ifelse existsCSexact(name)
3      sendContent(grabCSexact(name), inFace)
4      ifelse eq( head(name), 'web') )
5        ifelse eq( head(tail(name)), 'server'
               )
6          sendContent(
7                createContent(
8                  name,
9                  head(
10                   sqlexecute(
11                     'localhost|user1|pw1|
                           db1' //db connect
12                     'SELECT * FROM DATA1
                           WHERE NAME = $1'
13                     last(name)
14                   )
15                 )
16               )
17               inFace
18             )
19     exit
20   exit
21
22 )
23 handler onContent(name)(
24   ifelse existsCSlongest(name)
25     exit    // dup
26     addToCS(name, data)
27 )
```

Fig. 3. ICCL service implementation

First the application checks if the content was already requested. Next it verifies that the name matches. If it matches, a database query is invoked. The query contains the information to connect to the database and a SQL query. Since we search for a unique primary key, we know that only one dataset will be returned. We put the result in a content file and return it.

The SQL database could be replaced with a MongoDB database or just with a file system, depending to the application requirements. This gives the possibility to implement a complete webserver in ICCL. Since ICCL is Turing complete, the language can also be used to develop any kind of "network-services".

IV. RESULTS

We have implemented an ICCL runtime environment (in SCALA) that is able to execute programs as those presented in this paper. Conceptually, a specific ICN architecture and forwarding engine are replaced by the generic ICCL layer on top of which the ICN-specific programs are executed.

In the following we will compare the performance of an ICCL forwarder with the performance of a CCN-lite forwarder by using the NDN-repo and the ICCL server (Section III-C). We measure the number of packets which can travel through a forwarder when requesting chunked data from the network by using a very simple topology. The test setup consists of a

data source, a client and a forwarder. The different nodes are connected with 1 GBit/s Ethernet. The test setup is visualized in Figure 4.

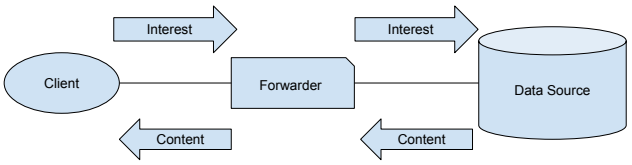

Fig. 4. Evaluation Setup: we measured the chunk rate per second inside of the forwarder.

As data we use a 45MB file chunked in the NDN format. We request chunk per chunk and do not use any flow optimization like a sliding window protocol (in receiver driven networks like current ICNs). One chunk has the size of 4KB. By measuring the chunk rate inside the forwarder, we can calculate the bandwidth.

Figure 5 shows the average bandwidth of the file download. The bandwidth of the native CCN-lite forwarder expected to be higher than the bandwidth of the ICCL forwarder. In our scenario the difference is about 40% when using the NDN repository. The bandwidth of a CCN-lite forwarder was arround 550kb/s while the bandwidth of the ICCL was arround 340kb/s. The performance of both forwarders becomes very similar (when we use the ICCL server) since in this case not only the forwarder but also the data source is limiting the bandwidth. Note, that none of the implementations is optimized for speed. Nevertheless, as a proof of concept we can say that ICCL add considerable flexibility, extensibility and compute capabilities with a reasonable loss of performance. For real world scenarios the performance loss could be critical, but for big networks it is usually required to use high optimized implementations or even FPGA or ASIC implementation of the forwarder, anyway. Especially for testing new ICN concepts, ICCL provides a way to deploy feature and at the same time documents their semantics. Since ICCL gives the possibility to describe forwarding strategies of ICN in a high level, it can be used to define the behavior of an ICN in a formal way and to discover subtle interpretation differences.

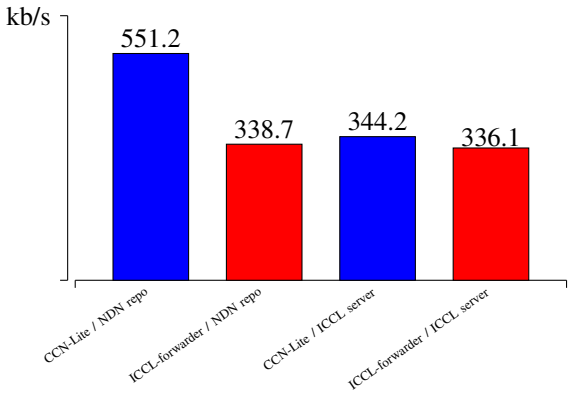

Fig. 5. Evaluation Results of our measurement. The graphic shows the average bandwidth of a CCN-lite and an ICCL forwarder when downloading a 45MB video file from the NDN repo and the ICCL server.

V. Conclusions & Future Work

In this paper we introduced the ICCL programming language for the domain of Information Centric Networking. It aims at extracting from the various ICN proposals a common core of functionality that all implementers have to provide, and to identify those additions that each architecture needs specifically. As it turned out, network-level as well as application-level programming requirements are quite similar wherefore we suggest that ICCL is used for both tasks.

Our goal with ICCL was to simplify the development of complex network application. We started with a simple Web server to prove ICCL's functionality for network applications and could create user-specific networks and network applications on a high level without requiring deep knowledge about network programming. Our measurements showed almost native performance which makes us believe that compiled ICCL programs will be able to match the performance of a CCN-Lite forwarder.

The next step will be to create a fully functional Web server that provides customized websites and "WEB 2.0" functionality like user-created content (requires functionality similar to HTTP-POST and GET). Furthermore, we plan to explore how to implement end-to-end security features in ICCL network application.

A very interesting exercise (at network level) will be to integrate support for Named Function Networking (NFN) into the ICCL environment: This would permit to treat NFN as an independent ICN architecture instead of layering it over NDN or CCNx.

References

[1] "`CCN-lite`," 2011-2015. [Online]. Available: http://ccn-lite.net/
[2] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, Dec. 2005.
[3] M. Sifalakis, B. Kohler, C. Scherb, and C. Tschudin, "An information centric network for computing the distribution of computations," in *1st Proc. Int. Conf. on Information-centric Networking*, ser. ACM ICN, 2014, pp. 137–146.
[4] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *5th Proc. Int. Conf. on Emerging Networking Experiments and Technologies*, ser. CoNEXT, 2009, pp. 1–12.
[5] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, K. Claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, "Named data networking," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 66–73, Jul. 2014.
[6] J.-L. Krivine, "A call-by-name lambda-calculus machine," *Higher-Order and Symbolic Computation*, vol. 20, no. 3, pp. 199–207, 2007.
[7] A. Carzaniga, M. J. Rutherford, and A. L. Wolf, "A routing scheme for content-based networking," in *INFOCOM (32nd Annual Joint Conference of the IEEE Computer and Communications Societies)*, vol. 2. IEEE, 2004, pp. 918–928.