Node Design for an Information-Centric Network Architecture

Paper #174, 14 pages

ABSTRACT

Information-centric networking has been touted as an alternative to the current Internet architecture by several research groups. Our work addresses a crucial part of such a proposal, namely the design of a network node within an information-centric networking architecture. We describe the service model exposed to applications and other network nodes, and present performance results for our current prototype. Our evaluation shows the overall feasibility of the design, the current performance in a test bed setup as well as qualitative advantages we see in our approach. In addition, we qualitatively contrast our design against CCNx and include a quantitative comparison with the existing prototype.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network architecture and design – *network communications.*

General Terms

Design, Experimentation.

Keywords

Information-centric networking, Click, network stack.

1. INTRODUCTION

The area of information-centric networking is increasingly attracting attention within the networking community. Several technological solutions within a range of architectures have been proposed, such as in [6][9][1], with subtle differences but also commonalities that stretch across the approaches.

One important aspect is the design of the network node within such architecture(s). One strategy is to evolve the current IP-centric design with its socket abstraction provided to applications through specific information-centric extensions. This may result in overlaying to a large extent on top of IP at the network level. This is desirable since it allows for a gradual evolution of the node design, in the face of a network-wide paradigm shift towards information dissemination.

Another strategy is a clean-slate design of a network node without the historical baggage of the IP stack. This strategy could, for instance, play a role in deployments in which access as well as fixed and mobile end nodes are replaced first. The questions arise as to what advantages such re-thinking could bring as well as how feasible such a design would be with today's computational means?

Following the latter strategy, this paper presents insights into a node design for information-centric networks that leaves nothing untouched with respect to the IP legacy. Our architectural starting point is the one presented in [1], outlining a network architecture that exposes a publish-subscribe service model, with functions for rendezvous, topology management and formation as well as forwarding. Based on this thinking, we provide significant implementation details as well as a qualitative and quantitative evaluation of advantages and performance.

For this, we organize the remainder of the paper as follows. Section 2 briefly elaborates on our architectural starting point, before delving into the various aspect of our node design in Section 3. We present an implementation of our design in Section 4 with example applications given in Section 5. Our evaluation in Section 6 focuses on quantitative performance as well as qualitative issues arising from our design choices. We contrast our design against other designs in Section 7, in particular taking the current CCNx node design as an alternative approach. We conclude this paper in Section 8.

2. ARCHITECTURAL STARTING POINT

Before delving into the node design, we first review the starting point for the internetworking architecture laid out in [1], deriving six major properties that the design for a network node within such an architecture must address

The first property is the means for *identifying individual information items*. While identification can take place through hierarchical naming schemes, such as proposed in [6], we advocate the usage of statistically unique fixed sized labels as a mean of identification. These labels carry no semantics and are meaningless to most network components and applications. However, relations to *real-world entities* can be established through algorithmically linking information item and identifier.

The second property places information items into a context, called *scope*. Hence, a scope represents a set of information and is therefore an information item itself, being identified as such with an individual identifier. Being information items, scopes can be nested under other scope(s), allowing for building complex directed acyclic graphs of information that can be utilized by distributed computational tasks for their purposes.

The third property is what kind of *service model* operates on the information graph. Distributed computational tasks are realized through an information flow between entities that utilize the exposed service model. The authors in [1] propose a publish-subscribe service model, which also

supports other models, such as pull or request/response, being realized on top of such basic model.

Our fourth property addresses the main functions of the architecture, as presented in [1]. These main functions realize the dissemination of information within a given scope of the overall information structure. The first one, *rendezvous*, matches availability of information to interest registered therein. This process results in some form of (location) information that is being used for binding the provisioning of information to a network location. This information is used by the second function, *topology management and formation*, to determine a suitable delivery relationship for the transfer of the information, this transfer being executed by the third function, *forwarding*. Although the functions are explicitly defined, they can be jointly implemented for optimization reasons.

The fifth property addresses the methods used for implementing the aforementioned functions but also the particular issues regarding information space governance and management within said part of the information space. For this, we define *dissemination strategies* associated to (parts of) the information structure, these strategies capturing the implementation details. Together with the scoping of information sub-spaces, these strategies establish a *functional scoping* through which the distinct functions can be optimized towards the particular computational task that uses this type of communication system. This optimized implementation of single strategies is extended towards an end-to-end realization of distributed applications by resolving any potential conflicts between strategies that are used within the system. Such conflict resolution might take place at design time of the system through methods of requirement engineering, functional specification, and standardization. Runtime assembly of the system can be envisioned through formalizing the strategies to enable reasoning over potential conflicts and mediating in case any such conflicts occur. We leave the details of such runtime assembly for future work.

3. NETWORK NODE DESIGN

We now map our system properties onto the specific component-level node design choices in our prototype. We base this design on an architectural starting point for an information-centric network architecture that was laid out in [1]. In order to accommodate the ongoing efforts in this space, our design needs to afford extensibility, which we will take into account in our work. We illustrate the internal components of the node design in Figure 1, showing how component boundaries correspond to main functions outlined in our fourth system property. These components provide rendezvous for information among publishers and subscribers, formation of a topology for information dissemination, and forwarding information in the network.

In a manner reminiscent of early IP nodes, an important aspect in our node design is the ability that a node can assume any role in the network with respect to these main functions. Such flexible assumption of roles is of utmost importance since it enables considerable flexibility in test and experimental deployments, particularly in an early phase of development and deployment. Apart from this deployment advantage, there is a fundamental issue that makes such flexibility important. Given that dissemination strategies can be manipulated throughout the lifetime of the information structure, a flexible node design allows for potentially utilizing *any* node in the network (rather than only the ones being optimized for certain strategies). However, it does not prohibit strategy-specific optimizations, e.g., high speed forwarding elements in the core network that utilize specialized hardware or hardware-offloading solutions (for which one could assume very little role change throughout deployment) or memory management optimized rendezvous nodes.

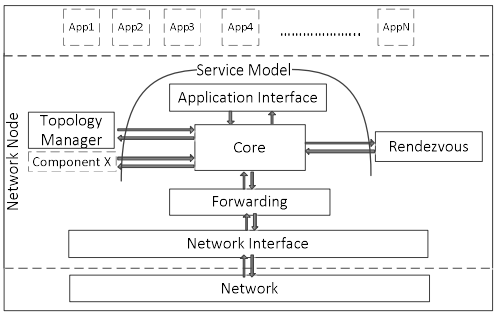

Figure 1. Network Node Design

3.1 Structuring Information

A significant aspect of an information-centric network node is the management of the information structure that is manipulated through the service model. This management is realized through the rendezvous component.

To address our first and second system property of Section 2, we choose to identify information items as well as scopes through statistically unique fixed size labels. These labels carry no semantics and are meaningless to most network components and applications. Any meaning to these labels can be assigned by the entities that produce them as well as by other network entities that utilize the particular information structure for their purposes. Global uniqueness of labels within the overall structure is not a prerequisite although it may be enforced in specific sub-graphs by the rendezvous module. Although information can be identified in the context of a scope using a fixed size label, the absolute path from a root of the graph (more than one path may exist) must be used when accessing the service model exported by our network node.

3.2 Immutability vs. Mutability

We now discuss the question as to what information semantics are supported in our design. First, consider the case in which individual information items are *immutable*. In this case, an application-specific item is labelled with a

(statistically) unique label under a given scope, for instance by performing a hash function over the information item. In a document example, we could assume each version of the document being labelled individually. In this case, each version is individually identifiable within the communications system. For any application, however, there needs to be an additional information exchange that disseminates the version identifiers. This is left to the application of our protocol stack.

We also consider *mutable* information, with each item carrying the same identifier as any previous one. Hence, the application needs to take care of any issues arising from this mutability, without having the capability to rely on (statistically) unique labels to do so, as it is the case for immutable items. Mutable items are very important when realizing, e.g., video delivery, in which the sequence of video images is published using the same information identifier, as also discussed in [18].

A hybrid of the previous two approaches is that of determining the identifiers through an algorithmic relation. This relation forms a *channel* of communication between the publisher and subscriber, while information items are still individually identified through the algorithmic relation. For instance, a numbering scheme could be included into the hashing function that creates an individual item identifier. Assuming that the communicating parties are aware of this algorithmic relation, the seemingly random identifiers can be associated with each other, e.g., in a re-assembly function within a network node that receives the fragments. Such algorithmic relations could span from simple numbering over sibling relations to encoded graphs. This powerful form of identifying an entire class of information could be utilized for areas such as caching or error control, in a very efficient way rather than specifically grouping the information through dedicated scopes.

Given that the semantics outlined above are advantageous to certain applications in one way or another, our node design supports all of them.

3.3 Realizing the Main Functions

Our fourth system property introduces three main functions for disseminating (part of) the information structure. Below, we discuss how these functions are realized.

3.3.1 Rendezvous

The Rendezvous component receives and processes all requests that are published by applications running locally or by other nodes. Such publications are created within the semantics of the exported service model. Upon receiving such requests, the rendezvous component matches potential publishers and subscribers of information items in order to facilitate the exchange of information between them. Realizations of the rendezvous component are defined through the dissemination strategy that underlies (parts of) the information structure. Such realizations may entail an enforcement of global identifier uniqueness, node-local visibility of information or information accessibility across one or more network domains. Moreover, a dissemination strategy could minimize the realization of the rendezvous module. As an example, a link-broadcast strategy may completely omit information management since publishers can directly utilize the broadcast nature of the local link(s).

3.3.2 Topology Management

The TM realizes the management of the overall delivery topology and the formation of specific delivery graphs for pub/sub relations. For that, the TM updates the topology information accordingly when nodes join or leave the network and creates the necessary forwarding information (and the potentially necessary state in the network) when requested. In a typical scenario, the rendezvous component may request such forwarding information in response to a successful match of a publish/subscribe request. Specific dissemination strategies may be reflected in the way forwarding information is created. For instance, multicast trees from a publisher to a set of subscribers may be created using a centralized shortest-path approach. Alternatively, the TM may create diffusion trees exploiting a socially aware network view. But strategy-dependent TM realizations might also be very minimal. For instance, when information is disseminated within the boundaries of a single node, creating topology formation is reduced to simply finding the interested applications in this node.

3.3.3 Forwarding

The forwarding module receives publications and forwards them to the network and/or to the local node. In order to do so, it maintains all necessary state and may coordinate with the topology management function. Dedicated forwarding nodes in the network may only realize the forwarding function using, e.g., hardware-optimized mechanisms.

3.4 Service Model

As depicted in Figure 1, applications as well as internal node components interact with each other and with the network through the provided *service model,* exposed as an internal API. This service model enables the manipulation of an information flow with a simple publish/subscribe semantic. In the following, we outline these manipulation operations and the utilization of the main functions. Although the text assumes a centralized realization of these functions for simplicity reasons, any realization within a given dissemination strategy can be utilized.

3.4.1 Publishing & Unpublishing Scopes

A publisher can create a scope in the information structure (maintained by the rendezvous component) by issuing a *publishScope(string id,string prefix,strategy s)* request.

Upon publication of a scope, the rendezvous component notifies all subscribers that have subscribed to the scope (if such a scope exists) under which the new scope has been published. Assuming that scope and information IDs are 2

bytes, represented as hexadecimal digits, the following requests (by publisher 1) would result in the scope structure (scopes are depicted as ellipses) shown in Figure 2, with the strategy being domain-local (DM).

a) publishScope(0000, NULL, DM)
b) publishScope(0001, NULL, DM)
c) publishScope(1111, 0000, DM)
d) publishScope(2222, 0000, DM)
e) publishScope(3333, 00002222, DM)
f) publishScope(000022223333, 0001, DM)

The prefix identifier is NULL when publishing root scopes. *Id* can be a single fragment in the information graph when publishing a new scope. Alternatively, it may contain multiple fragments that identify an existing scope when republishing a scope under another scope (identified by *prefix*). If the scope identified by *prefix* did not exist in the information structure, the request would be rejected.

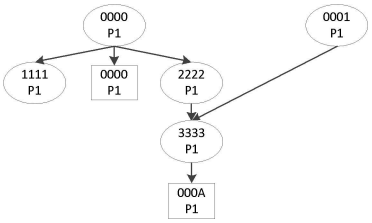

Figure 2. Publishing Scopes and Information Items

Unpublishing a scope is requested through *unpublishScope(string id,string prefix,strategy s).*

The rendezvous component will unpublish all information items that are children of the scope. If this succeeds and no other subscopes exist under the scope identified in the request, the scope is unpublished. If the scope was published under multiple scopes, only the specific branch in the graph is deleted. All subscribers that have subscribed to the father scope(s) of the deleted scope are notified.

3.4.2 Publishing & Unpublishing Information Items

A publisher advertises information items through *publishInfo(string id,string prefix,strategy s)*

Information items reside under one or more scopes. Following the previous example, the following requests would result in the information structure shown in Figure 2, with information items depicted as rectangles:

a) publishInfo(0000, 0000, DM)
b) publishInfo(000A, 000022223333, DM)

Depending on the existence of subscribers in the information graph, the rendezvous component may initiate the process of matching publishers and subscribers for the item. Depending on the dissemination strategy for this sub-graph, the rendezvous component may further request topology formation from a TM (e.g., in a domain-local strategy) or not (e.g., node-local strategy).

Unpublishing an information item is done through *unpublishInfo(string id,string prefix,strategy s).*

As a result, the publisher that issued the request will be removed from that information item in the information graph. If there are no other publishers or subscribers, the item is deleted from the graph. If there are remaining publishers and subscribers for this item, rendezvous will take place again since one or more publishers must be notified to publish data for this information item. If an information item has been advertised under multiple scopes, the item is only deleted from the scope identified by *prefix* (assuming that there are no other publishers or subscribers for the identified item).

3.4.3 Subscribing to & Unsubscribing from Scopes

A subscription to a scope is done through *subscribeScope(string id,string prefix,strategy s).*

Upon receiving such request, the rendezvous component creates a new scope if the scope defined in the request does not exist. By subscribing to a scope, a subscriber declares its interest in all scopes and information items that directly reside under that scope (not under the entire sub-graph). Therefore, the rendezvous component looks for any scopes or information items that are published under that scope and acts as follows: for any scope, it will publish a notification about the existence to the subscriber. For all information items, one or more publishers that have previously advertised the item must be notified.

Unsubscribing is done accordingly. The subscriber is removed from the subscribers' list of the scope. If there are no other publishers and subscribers as well any items or subscopes, the scope is deleted from the information graph.

3.4.4 Subscribing to & Unsubscribing from Information Items

A subscription to an information item is done through *subscribeInfo(string id,string prefix,strategy s).*

If publishers have previously advertised this item, the rendezvous component matches them with this subscriber and any previously subscribed one. Let us assume the information structure shown in Figure 3.

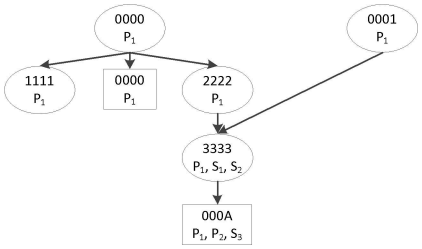

Figure 3. Subscribing to Information Items

Here, publishers and subscribers that previously issued requests are shown for clarification. Subscriber 3 has just subscribed to the information item with ID *000022223333000A*. The rendezvous component now

matches publishers *P1* and *P2* with subscribers *S1*, *S2* and *S3* and publishes a topology resolution request to the TM. Assuming that in our example *P1* is physically the closest to all subscribers (and that a shortest path algorithm is used), *P1* will be notified to publish data for that item. For unsubscribing, the subscriber is removed from the information item and if there are no other publishers and subscribers, the rendezvous component deletes the item from the information graph. Otherwise, rendezvous will take place again and one or more publishers will be notified to publish data for the specific piece of information.

3.4.5 Publishing Data

A publisher sends data for a specific information item by issuing *publishData(string id,strategy s,char *data)*

Here, *id* is the absolute identifier (i.e., starting from a root scope in the information graph) of an information item. A publisher usually issues such request if rendezvous has already taken place, i.e., it is already known how to forward data for the identified information item. There can also be cases where a publisher requests to immediately forward data for the identified item. In such cases, it is assumed that the publisher itself is able to construct the forwarding information, which is then included in the requested dissemination strategy. An example of such operation is when the TM publishes data (i.e., responses to previous requests) to the rendezvous component.

3.4.6 Asynchronous Upcalls

In most cases, requests sent by an application to the network stack will result in further actions taken by the rendezvous system and the TM. These actions may finally require a notification towards the network node that issued a request, which in turn may notify one or more applications. These notifications are asynchronous.

New scope **notification.** Sent to a subscriber when a new scope is created under a scope to which the subscriber has subscribed. The notification includes the scope identifier.

Deleted scope **notification.** When a scope is unpublished and removed from the information graph, subscribers of the parent scope(s) are notified. The scope identifier included in the notification is the one that is relevant to the previous subscription of the application.

Start publish **notification.** Sent to an application, which has previously advertised an information item, the first time the network stack receives a respective notification from the rendezvous component. A publisher does not receive notifications when the set of subscribers changes, although the network stack at the publisher's node internally updates the forwarding information. *Start publish* notifications contain the identifier of the item to be published.

Stop publish **notification.** A *stop publish* notification is sent to an application whenever there are no more subscribers to an information item, for which a *start publish* notification has been previously sent.

Published data **notification.** A *published data* notification is sent to an application (subscriber) when data has arrived for a specific publication. This notification contains the information ID as well as the received data.

3.5 Node Core Component

This component has two major roles. First, it receives all publish/subscribe requests sent by applications and other node components and, according to the dissemination strategy, stores them locally, forwards them to the local rendezvous component or publishes them to the network. Moreover, it receives publications from the network and dispatches them to subscribed software entities. Finally, it receives publish/subscribe notifications (see Section 3.4.6) from the network or from the local rendezvous component and notifies all interested applications. Note that these notifications are sent as publications using a predefined information identifier. A detailed description with strategy-specific implementation details can be found in Section 4.3.

3.6 Interfacing Network and Applications

All publications are sent to the network via the network interface component. This component abstracts the physical network through a unified API for sending and receiving publications. One could foresee interfaces for Ethernet or Bluetooth, overlays on top of IP, TCP or even HTTP. The actual realization of these interfaces will depend on the strategy that is chosen for a particular deployment. Section 4.5 describes the currently implemented mechanisms.

As is the case for any other network node, our network stack is required to provide the supported service model to applications. Section 4.2 outlines our current implementation choice for this particular aspect.

4. IMPLEMENTATION

We implement our node design as an open source prototype. We base the resulting prototype on the Click modular router [8] platform, where all node components are implemented as *Click elements*. Click has been used in many efforts that experiment with control plane extensions to the current IP forwarding. The scope of our node design, however, is wider than that. We also design forwarding components (replacing current IP) as well as application-facing interfaces that allow for accessing our network stack. We see Click complementing our efforts because (a) its Communication elements (*FromDevice* and *ToDevice*, exposed to other Click elements) support a variety of transport mediums, ideal for experimenting with IP replacements, (b) Click allows for integrating different application-facing interface techniques, and (c) the notion of Click elements enables the development of our main functions in the control plane (rendezvous and topology management) in a way that eases portability between kernel and user space as well as across operation systems.

The latter aspect is reflected by our implementation running both in kernel (as a multi-threaded kernel module) and user

space (as a multi-threaded process). The user space deployment supports quick prototyping of functionality as well as experimenting in environments where throughput is not the primary metric. The kernel space deployment is more efficient in terms of performance since the elements for interacting with the applications and the network use mechanisms specific to the kernel. The remaining node elements share the same implementation when running in user or kernel space. The cross-OS support is demonstrated by our node implementation being available in Linux as well as FreeBSD (with ongoing efforts to port to Android). Another advantage of Click is its possible integration into ns-3, an effort left for our immediate future work.

Although we envision a node to run all components, we foresee dedicated forwarding nodes that may only run the communication as well as the forwarder elements, possibly implemented (at least partially) through hardware offloading. On the other hand, if augmented with in-network functionality such as caching or mobility support, such forwarders should also support the rest of the elements as required for the additional functionality (bringing them again closer to the role of a full end node). It is the Click modular approach that eases these developments by adding dedicated Click elements to our node design.

4.1 Implemented Dissemination Strategies

Before delving into implementation details, we briefly describe the currently realized dissemination strategies. Across all strategies, we realize our information identifiers as 64 bit long statistically unique flat labels. Clearly, there is a trade-off regarding the identifier size that reflects the depth of the information graph contrasted to the width of the identifier space under each scope.

***Node-local* strategy**. Here, information is disseminated in a single node, effectively providing an inter-process communication mechanism. The rendezvous component maintains the information graph, which is visible and accessible only to applications and other Click elements running locally as well as the kernel itself. The topology management and formation component is minimal since it only needs to find the interested (local) processes. Finally, the forwarding component pushes publications to the right Click elements and applications.

***Link-local* and *broadcast* strategy**. This strategy allows a node to disseminate information to its immediate physical neighbors, possibly over a broadcast medium like Ethernet. The rendezvous component is minimal since there is no information graph maintenance. Instead, subscribers implicitly subscribe to specific information items. A publisher can then publish information to a specific link (link-local) or to all links (broadcast). If a subscriber exists on this link, the information is forwarded to the interested application(s). The topology management and formation component is again minimal, since it only needs to find the appropriate (active) network link. The forwarding component stores this information internally and simply forwards data to one or more of its links.

***Domain-local* strategy**. In the domain-local strategy, all components are fully realized. One or more nodes act as rendezvous nodes of the domain-local network (e.g. by sharing the information identifier space). One or more dedicated nodes run a Topology Management and Formation component with a centralized view of the network, creating multicast forwarding trees from a set of publishers to a set of subscribers using a centralized shortest-path algorithm (this creation of trees is requested by a rendezvous component after a successful match of publications and subscriptions). Such multicast tree is formed by computing source-based LIPSIN bloom filters [7] of the individual link identifiers in each forwarding node along the path. Based on this constant size identifier, the Forwarding component efficiently forwards each packet through a simple AND/CMP operation on the Bloom filter identifier, resulting in efficient multicast support.

***Implicit rendezvous* strategy**. This strategy is a mix of the previous two. Information is disseminated across a network domain but no explicit rendezvous takes place. Instead, subscribers declare their interests for information to their local node and publishers publish data by also providing a LIPSIN identifier to the destined node(s). In that way, information is directly disseminated in the network using the provided identifier. This strategy is used by the rendezvous components that publish their requests for topology formation to the TM (using a preconfigured LIPSIN identifier that "points" to the TM) as well as by the TM for publishing responses to publishers and subscribers. Other possible use cases are networks in which strong management of the link identifiers can be assumed (and therefore pre-configuration is possible).

4.2 Interfacing Applications

The *SysCall* component interfaces our network stack towards applications. Traditionally, all applications interact with the networking software of an operating system via system calls. In TCP/IP stacks, read/write and select/poll like system calls are used. In modern Linux kernels, adding system calls requires kernel recompilation, hindering quick experimentation. For that reason, we choose *Netlink* sockets to interact with applications. All applications are required to first open such a socket and then interact with the network stack using existing system calls provided by the operating system. It is this approach that comes closest to introducing a new set of system calls into our design. But Netlink sockets provide two further advantages over the implementation of system calls. First, applications use (almost) the same API for accessing the network stack regardless of the mode in which the implementation runs. Second, in kernel space, Netlink sockets are very Click-friendly since the network stack receives socket buffers that are wrapped into Click packets with no extra memory cost.

Within the user space implementation, we use the *selected()* callback method that is provided by the Click element implementation. Netlink sockets are datagram sockets. Hence, the component reads a data packet from the socket (i.e., an application pub/sub request) whenever the socket is readable and writes a data packet (i.e., a pub/sub event) whenever the socket is writeable.

Since Linux kernels provide a special interface for handling Netlink sockets, our kernel space implementation of this mechanism is very efficient. More specifically, the network stack directly receives data packets as socket buffers and wraps them in Click packets, which are the internal structures for element communication, without *any* extra buffer copies. The network stack receives application requests in the context of the process context and, therefore, it immediately unblocks each process by placing the received data buffer in a FIFO queue. The actual buffer process is deferred to a separate Click task. Efficiency can be further boosted in multi-core CPUs environments by running Click elements, along with their tasks, in separate kernel threads. After wrapping each received buffer, the SysCall component annotates the packet using the Netlink port at which the sender application expects all publish/subscribe events.

4.3 The Core Component

As depicted in Figure 1, the *Core* component is at the heart of the node design. All Click packets received by the Core component are annotated with an application identifier. Click packets received by other Click elements are annotated with the Click port with which the core element is connected. A further role is that of providing a proxy function to all publishers and subscribers. Rendezvous nodes (even the one running in the same node) do not know about individual application identifiers or click elements. Instead, a statistically unique node label that identifies the network node (e.g., the hash value of a MAC address which can be self-assigned) from which a request was sent is stored in the Core component.

4.3.1 Processing Publish & Unpublish Requests

Whenever the Core component receives a request for advertising a scope or an information item, it checks if any other application or click element has previously advertised the same scope or item. In the former case, it creates a publication with the initial request as the payload. Then, according to the dissemination strategy, it publishes the data to the appropriate rendezvous component. For example, if the strategy is domain-local, it publishes the request using a pre-configured LIPSIN identifier to a domain's rendezvous node. If the strategy is node-local, the core element will forward a Click packet to the local rendezvous component as a publication notification. This accommodates the support for varying dissemination strategies within our fifth system property. In case that another application or click element does advertise the same information, the Core component simply adds the publisher to its local index. The rendezvous component already stores the node label as a publisher, since another application running in the same node has previously published the scope or the information item.

The Core component publishes an unpublish request to the rendezvous node only when no more local applications or click elements are publishers of the scope or information item. Anything that crosses the network should be a network publication. When requests are published to the rendezvous node via the network, the Core component uses the */RV_SCOPE/nodeID* as the information identifier, where *RV_SCOPE* is a well-known scope to which the rendezvous component of all nodes subscribes. The rendezvous node receives this publication, extracts the node label from the information identifier and stores this label as the publisher of the requested scope or information item.

4.3.2 Processing Subscribe & Unsubscribe Requests

Subscribe and unsubscribe requests are processed similarly. When a subscribe request is first received, the Core component publishes a subscription request on behalf of the application to the rendezvous node according to the dissemination strategy. Further requests for the same scope or information item are stored locally. When the implicit rendezvous strategy is used, the core element stores the request only locally without notifying the rendezvous component. Unsubscribe requests are published to the rendezvous node when no further applications are subscribed to a scope or information item.

4.3.3 Processing Publish Data Requests

The Core component publishes data on behalf of an application or a click element only when the requested information item has been previously advertised and rendezvous has taken place, i.e., if the Core component already holds a forwarding identifier to one or more subscribers. The same identifiers are also used for the node-local (an internal forwarding identifier) and the link-local (the link identifier to the physical neighbor) strategies. Applications are notified by the Core component when such a forwarding identifier is assigned to an information item. The setting of an implicit rendezvous strategy by an application represents an exception to the above functionality. In this case, the Core component publishes the data using the provided forwarding identifier.

4.3.4 Handling Network Publications

The Core component receives all publications destined to this node from the forwarding element. Then, it looks up its local index to find subscribers for the published information item or for the scope that is the parent of this item. If no subscribers exist for the publication, it is rejected. In the opposite case, a *Published Data* notification (along with the data) is sent to all interested applications and click elements. For example, as we described above, the Core component publishes all publish/subscribe

requests to the rendezvous node. The Core component running in the rendezvous node receives these publications and publishes a *Published Data* notification to the Rendezvous component, which has already subscribed to the special rendezvous scope when the node starts. The Core component does not process the payload of the received publications - this is left to the receiver of the data.

The exception to the previous rule is when notifications from the TM arrive in response of a previous publish/subscribe request. These notifications are published under a special scope to which the Core component itself is implicitly subscribed. In this case, the publications' payload (i.e. the notification sent by the TM) is processed as described in the following section.

4.3.5 Processing Rendezvous Notifications

Depending on the dissemination strategy, a rendezvous notification may be published by the domain's TM (domain-local) or by the rendezvous component that runs locally (node-local). Such notifications are published in response to publish/subscribe requests. Notifications about the creation or deletion of scopes are matched with possible subscribers, forwarded to them as the respective events.

When a forwarding identifier is received for a previously advertised information item, the Core component may or may not notify a publisher application. Initially, the core element assigns a NULL identifier when an information item is advertised. Upon the reception of a non-NULL identifier, a *Start Publish* notification is forwarded to one of the potentially many publishers of the same item. We currently implement a random selection, although other solutions for, e.g., load balancing, can be integrated later. Whenever new forwarding identifiers are received, the Core component does not notify the publisher as long as the received identifier is not NULL. In the case of a NULL identifier being received, a *Stop Publish* notification is forwarded to the application that has previously received the *Start Publish* notification.

4.4 Forwarding Component

The forwarding component currently implements the basic LIPSIN forwarding mechanism [7] for domain-local dissemination, as outlined in Section 4.1. For this, it maintains a forwarding table that maps link identifiers (LIDs) to Click ports that point to an element that uses our node design to access the network. Another LID is used to "connect" the forwarding with the Core component. A publication may have multiple identifiers since it may be reached by multiple paths, starting from one or more roots of the information graph. Generally, the number of identifiers in a publication will be kept small. That is because a single node only needs to know about a small subset of these IDs (in most cases a single ID). This is true even if an information item is identified with multiple IDs.

4.5 Interfacing the Network

Our node design utilizes Click elements for communicating with other network nodes. We support Ethernet communication using the *FromDevice* and *ToDevice* Click elements as well as communication over raw IP sockets. The former can be used when experimenting in a LAN or VPN (using the *tap* virtual interface), while the latter is appropriate when overlaying on top of IP. Note that nodes (especially forwarders) may have multiple instantiations of the aforementioned elements. The Forwarding component also allows for running our network stack in a mixed mode where a node may run as a bridge interconnecting two or more LANs over an IP network, with the individual LANs running the network stack over Ethernet. This allows for complex deployments as well as experimentation.

4.6 Software-Engineering the Strategies

Dissemination strategies are a crucial part in our node design. They influence the information flow, by altering the way the core, rendezvous, TM and forwarding elements process publish/subscribe request, within a single node or across the network. Currently, this is implemented using switch/case code fragments in which this behavior is coded. However, we are working on formalizing the way strategies are expressed as well as modularizing the code using a base dissemination strategy class that is extended by classes representing each strategy.

5. EXAMPLE APPLICATIONS

In the following, we present two examples for utilizing our node design in applications. The first example introduces (managed) caches into a network segment, allowing for experimenting with the ability to improve overall network efficiency. The second example is that of a simple video delivery at the application layer.

5.1 Managed Caching

In [4], the authors implement an initial approach for content placement, which can directly be realized on top of our node implementation. Caches in the network receive the content by subscribing to the respective information items and make themselves available as replica holders by republishing all information items in the domain's rendezvous node. The TM uses a shortest path algorithm in order to select the best publishers for a specific set of subscribers. This solution demonstrates the flexibility to expose various caching solutions over the same basic substrate, as opposed to integrating caching as an inherent part of the forwarding solution (as done in CCN [6]).

5.2 Video Transmission

As a simple application, we have implemented a multicast-enabled video streaming application. Subscribers subscribe to information items that represent channels in which all video frames are sequentially published, i.e., information here is mutable. The identifiers for each video (channel) are advertised by the publisher. When the first subscriber joins

a stream, the application is notified and starts publishing the video data. When another subscriber joins (or leaves) a channel (by subscribing to or unsubscribing from the respective information ID), the publisher's node receives a new forwarding identifier that defines the revised multicast tree. The underlying domain-local multicast, described in Section 4.1, results in a video delivery to multiple receivers at a reasonable speed with full SD video resolution. This application currently runs in a test bed consisting of 26 nodes (across Europe and the US) connected via a VPN.

6. EVALUATION

Let us now present evaluation results for our node design. We begin with qualitatively assessing the general feasibility and advantages of our design before presenting quantitative performance results for our prototype.

6.1 Qualitative Evaluation

Qualitatively, we can differentiate aspects that derive from the particular design (guided by the overall architectural context in which the design is embedded) from others that stem from our particular choices made at the implementation level. We address the former first before turning to the latter at the end of this sub-section.

Transparent distribution: Information graphs are independent from the location of the information items at any point in time; it is the dissemination strategy that defines the distribution of the information within (regions of) the overall graph. Hence, while distribution is generally transparent, it can be controlled based on, e.g., application input but also as a system strategy for, say, redundancy. For instance, an application could adapt the dissemination strategy of an established graph after its creation in order to extend the information from being originally node-local toward an entire domain. With this, the notion of an IPC (inter-process communication) can transparently extend to an entire (even global) network through a simple change in dissemination strategy. The role-agnostic design of our node facilitates such flexible strategy changes.

Modularity of specific main function realizations: Our node design upholds the modularity of main functions that is introduced by our fourth system property in Section 2. This not only enables the transparent distribution outlined before but it also allows for separately optimizing functions throughout the lifetime of the prototype. Such separate optimization is not only likely due to the specificity of each function but also due to the potential different 'ownership' of each function in certain deployment situations. For instance, topology management in a domain-local scenario could be implemented by local ISPs, either through centralized or distributed solutions. The rendezvous function, however, could be entirely driven as well as realized by large content providers through a fully distributed, DHT-based solution, similar to the one proposed in [15]. Hence, the functions can be separately implemented along clearly defined modular boundaries. We can argue that these modular boundaries represent *tussle spaces* within the larger network architecture, with [13] emphasizing the importance of clearly defined boundaries between these spaces. The authors in [1] provide a first argumentation for the particular separation suggested here. We do not extend on this argumentation in this paper and refer to [1] and any possible future arguments that will elaborate on this aspect.

Independence from information management approach: Currently, our network stack uses a bespoke information space management within the various, e.g., rendezvous components. Other approaches can easily be integrated. These could utilize, e.g., database management or file system approaches, leading to rendezvous functions that are optimized for vast information spaces, e.g., for organizations or entire application domains.

The following aspects arise from our implementation choices:

Flexibility through Click basis: The choice of Click as a platform brings several advantages. Firstly, our node design as well as network-level extensions (e.g., for congestion and error control) can be easily implemented at user space level with reasonable effort to compile a kernel space version. This allows for faster development cycles. Secondly, Click abstracts the network and inter-process communication specifics of each host OS, allowing for easily porting our node design onto supported host platforms. Such cross-OS support is important for an early prototyping in a new research area such as information-centric networking. This is highlighted by our currently ongoing work to port our node design onto Android-based mobile devices, an effort greatly simplified by the chosen Click basis. And thirdly, resulting implementations can be further optimized through hardware offloading (as any other node design) with efforts such as [10] providing a first direct route for such hardware assistance.

Supporting various deployment models: The Click basis also leads to supporting various transport mediums. While much of the information-centric networking research aims at replacing the current IP layer, our prototype not only supports Ethernet and other link mediums but can also base its communication directly on IP or TCP/IP. With that, overlay evaluation scenarios are easily realized in platforms such as PlanetLab, an issue that is of utmost importance in a research prototype. This eventually also provides a larger variety of possible deployment scenarios for the technology beyond a research prototype.

6.2 Experimental Evaluation

We now present the results of the experimental evaluation, conducted in a Gigabit LAN consisting of 15 identical hosts and in PlanetLab. The focus of the evaluation is to assess performance of various aspects in our design.

Memory management performance: For stressing the local memory management, we test the implementation of our node-local dissemination strategy in dealing with heavy load of (local) publications, emulating an IPC-like mechanism. We use a single publisher that advertises a single information item under a root scope. We then measure the application throughput for a set of subscribers, ranging from 1 to 10, subscribing to the advertised information. Upon notification, the publisher publishes 100,000 items using the same information identifier. Therefore, rendezvous takes place only once. We repeat the same experiment for different payload sizes. Note that the upper limit of the payload size in this experiment is set by the size of the Netlink socket buffer.

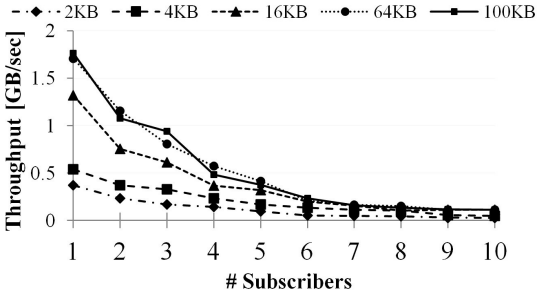

Figure 4. Memory Management Performance

In Figure 4, we observe that for large payloads and few subscribers the average achieved application throughput is more than 1 GB/sec. The performance is degraded when more subscribers exist due to multiple publication copies. For 10 subscribers, the measured throughput is between 40-100 MB/sec. We point out that for the current node-local dissemination strategy, the core element copies and forwards all publications to the interested applications. However, this local memory management could be replaced with different strategies, e.g., a local blackboard could be utilized in order to eliminate unnecessary copies.

Fast path performance: In our second experiment, we extend towards a domain-local strategy with a simple star topology consisting of 4 nodes, testing both memory and forwarding performance on the fast path of delivery. A satellite node is the publisher, which acts as in the previous experiment, and the other satellite nodes are the subscribers. All published items, including the Ethernet header, have an MTU size of 1500 bytes. The rendezvous and TM nodes run in the center of the topology. Although the experiments are performed in a shared medium test bed, the domain-local dissemination strategy does not directly utilize this shared medium nature since the LIPSIN-based forwarding of this strategy is based on the notion of dedicated *links* between nodes in a network. As a consequence, forwarding to e.g., two different nodes on two different LIPSIN links halves the effective throughput that can be observed. As observed in Figure 5, the measured throughput for a single subscriber per node reaches the maximum transfer rate since each publication is transferred using the same Ethernet interface.

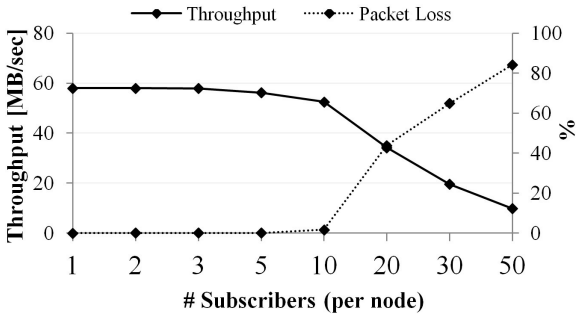

Figure 5. Fast Path Performance

The measured throughput starts degrading only for 10 subscribers per node and more. In the same figure, we also depict the measured packet loss, which dramatically increases when more than 20 subscribers per node exist; an expected consequence given the processing stress in each node. In order to measure the efficiency of the forwarding function, we create a topology of 15 nodes connected in a chain. The first node in the chain runs the domain's rendezvous node and TM. The second node is the publisher that behaves similar to previous experiments. The rest of the nodes run 1, 3, and 6 subscribers (depicted as (1), (3) and (6), respectively).

In Figure 6, we observe that when a single subscriber runs in each node, all subscribers receive data at line speed even when 13 subscribers exist.

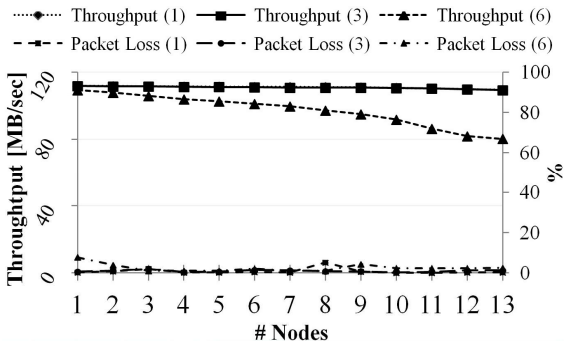

Figure 6. Forwarding Efficiency

In this extreme case, each forwarding node forwards all publications to its next hop and pushes the data to the local subscriber. Similar results are achieved when running 3 subscribers per node. Only in the case of 6 subscribers per node, the performance degrades for a chain larger than 3 nodes. For all cases, the packet loss is less than 5%.

Slow path performance: We now turn our attention to the slow path, namely the sequence of rendezvous as well as topology management and formation that needs to take place before executing the fast path forwarding. We first focus on the rendezvous process by utilizing the node-local strategy, which requires no explicit topology formation. The publisher creates a scope and then advertises 100,000

information items under that scope. Then, a number of subscribers are synchronized to start subscribing to items in the advertised range. Each subscriber iterates 500 times and for each one it subscribes to an item using a randomly generated identifier, waits until it receives the data and then subscribes to another one. For each subscription, rendezvous takes place and the publisher is notified to publish the payload (the information here is a minimum Ethernet frame in order to reduce forwarding operations) that corresponds to this advertisement. We then measure the time between each subscription and the receipt of the data at the subscriber. We call this the *response time*[1].

Figure 7 depicts the measured response time when 1 to 500 subscribers subscribe to randomly generated information identifiers. In the node-local case, the average response time for 200 subscribers is 20 ms, with a linear increase to 54 ms for 500 subscribers. Note that in application scenarios with mutable information semantics explicit rendezvous does not take place for each information item. Hence, the penalty is diminished as more data is sent (e.g., using the same identifier or algorithmic identifiers).

We repeat the same experiment using a domain-local strategy in a star topology of 15 nodes in our Gigabit LAN testbed. The star topology provides a constant forwarding delay for all subscribers since the rendezvous node and TM run in the central node and the publisher runs in one of the satellite nodes. In this strategy, the entire slow path, ranging from rendezvous to the topology management and formation, is involved. In Figure 7, we observe that the response time linearly increases with the number of subscribers (up to 500 per node).

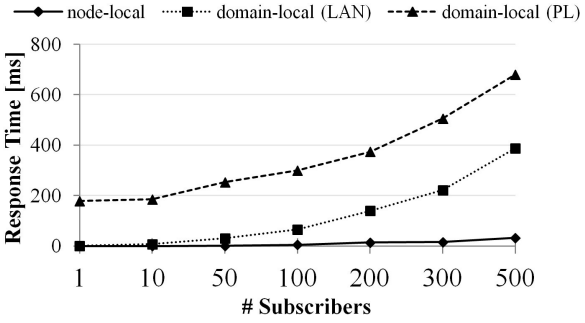

Figure 7. Slow Path Performance: Node-Local, Domain-Local in LAN, Domain-Local in PlanetLab

Compared to the node-local case, the response time is higher since network delays become a factor. Another factor is that each of the 14 nodes runs the number of subscribers depicted in the x-axis, resulting in response time of 388 ms for a total number of 7000 subscribers.

[1] This response time constitutes a worst case for the rendezvous overhead. Realistically, however, a solution would minimize slow-path operations by, e.g., caching rendezvous results or forwarding identifiers, improving the overall performance.

PlanetLab overlay performance: As an example for an overlay deployment of our prototype, we present the slow-path performance results in a Planetlab scenario. For this, we create a slice that consists of 106 slivers forming a (randomly generated) graph with 73 edge nodes. The graph is generated using the iGraph library [2]. The rendezvous node and TM of this overlay network run in a dedicated machine in order to eliminate the performance variations usually present in PlanetLab nodes. The publisher also runs in this machine. The results for this experiment are included in Figure 7. For 200 subscribers per node (totaling 14,600 subscribers), the average response time is 373 ms, which increases to 680 ms for 500 subscribers per node (i.e., 36,500 subscribers). The response time variation is higher than for the LAN case, since a different number of hops (overlay in the generated graph and physical) are involved until each subscriber receives the published data.

7. CONTRASTING AGAINST CCNx

Since the inception of the IP node architecture, there have been many attempts to re-think the design of a network node. For instance, Haggle [12] provides manipulations of a linked information graph based on publish-subscribe operations. Haggle's component wheel separates functions for information dissemination in a plug-and-play manner, similar to our design. In the area of data center networking (DCN), RipCord [5] is an example effort. Based on the traditional IP abstractions provided to applications, the node design separates functions for topology management as well as forwarding for separate optimization. Going beyond the IP node design of today, the authors in [11] argue for HTTP and the DNS as the effective waist of the Internet that widely resembles information-centric features. As a contribution to the larger architectural space, Day [3] abstracts the functions in each layer as a collection of fast path *forwarding*, slow-path *control* and longer-term *management*. The underlying service model is that of a remote procedure call between dedicated endpoints. Day also introduces *recursive layering* to properly expose overlaying as a property of the overall design (rather than leaving it to ad-hoc attempts of shim-layering).

More recently, there has been one notable effort in re-designing network nodes, an effort that is placed in a similar information-centric context as ours. In the following, we contrast our node design against this effort on content-centric networking (CCN) [6] at the qualitative as well as quantitative level, utilizing the open source *CCNx* prototype, as it is currently publicly available.

7.1 Qualitative Comparison

Similar to our presentation in Section 6, we can divide a qualitative comparison with CCNx into issues that are affected by design and those affected by implementation.

Modularity of main functions: Our node design realizes the separation of main functions outlined in [1], supporting

a wide range of realizations for these functions. For instance, solutions similar to IP multicast (i.e., table lookups performed on longest prefix matches) are as much envisioned as our current domain-local implementation that uses a fixed size source routing address [7].

In contrast, CCN separates its main functions into (a) *routing* (building *forwarding information tables*, FIBs, in each forwarding element) (b) *forwarding* (sending interest and data packets to producers and consumers, respectively, utilizing a *pending interest table*, PIT) and (c) *caching* (storing both interest and data packets for improving the overall delivery). While routing is currently neither fully specified nor implemented, forwarding and caching are realized in the current prototype. These two functions consult the local FIB, making forwarding decisions for data as well as interest packets for a given (data) name, while caching is performed for any packet that traverses a node, with the *content store* holding previously transferred packets. The work in [14] studies the applicability of this design with respect to memory requirements by estimating a conservative evolution of content names along a progression of host names similar to the current Internet. Any deviation from this estimation upwards leads to the conclusion that the FIB/PIT-based CCNx node design is not feasible according to available memory technology roadmaps. While our design can face similar bottlenecks, it also allows for design choices that are free of them, such as the stateless forwarding on fixed size source addresses.

Content security: According to [6], CCN requires the producer to sign every data packet injected into the network[2]. This ties the feasibility of the CCN node design to cryptographic advances that would make such signing feasible at high data rates. While mandatory signing is reasonable for videos, news or other media, it can pose a significant burden on end systems in scenarios such as sensor networks or mobile device based content production. Also in certain deployments, secure communication is already provided at the level of the host network, leading to redundant efforts at the CCN level[3]. These reasons have led to the decision in our node design to make signing of information items an optional part which can be easily added afterwards, including a coupling of content and information identifier through self-certification [16] or adding packet-level authentication with variable verification at network level [19].

Role of caching: One of the key promises of CCN is to improve network utilization through ubiquitously caching content. Within CCNx, the *Content Store* is consulted before passing an interest to the next hop. Among others, the authors in [17] have expressed their doubts regarding the overall performance of such caching scheme, based on existing evidence from past operations of, e.g., web caches. Furthermore, caching is costly albeit futile in scenarios such as conversational services. While the final jury is still out on this issue, the approach taken by our node design allows for incorporating a wide range of caching solutions into a final deployment, either independent from the main functions (as implemented in [4]) or as a solution that is jointly optimized with one or more of the main functions.

Supported information semantics: In CCN, all content is immutable. Hence, any new content carries a new name. While this assumption is reasonable for, e.g., content delivery, it poses a significant burden on, e.g., conversational or sensor applications. As an example, the voice application in [6] is realized by pulling chunks of voice data from the producer, these pull requests being frequently sent by the consumer. While the authors in [18] argue for supporting mutable content for such scenarios, their proposed CCN extension would lead to timer-based entries in each PIT along the path, which is unsustainable in terms of overhead. In contrast, our node design supports applications with mutable as well as immutable data, as discussed in Section 3.2. This has a significant impact on the performance of, e.g., conversational services.

Let us now turn towards issues that are affected by the particular realization of both node designs.

Platform modularity: In Section 6.1, we highlight the importance of choosing Click for realizing our node design. This choice enables (a) easier portability to other platforms, (b) support for different transport mediums, and (c) easier support of kernel and user level protocol stacks. In contrast, CCNx is implemented as a monolithic user-level implementation, which not necessarily results in superior node performance, as we will see in Section 7.2, while not providing the advantages of a Click-based design.

Supporting various deployment models: Part of the monolithic design of the CCNx prototype is the realization over a single transport medium, namely TCP/UDP (over IP). With that, deployment over other mediums, such as Ethernet, need dedicated support through specific extensions to the CCNx code base. In contrast, using Click in our implementation limits deployments to the supported communication elements in Click. This support in Click currently includes UDP (e.g., for PlanetLab deployments) as well as Ethernet (e.g., for LANs or VPNs).

---

[2] Signing is only necessary at the producer with an optional verification by the consumer. Delivering data from a cache removes any performance degradation that stems from signing. This assumption is made in [6] when determining the CCN throughput, i.e., data is played out by a cache. A true end-to-end performance would be significantly lower than portrayed in [6].

[3] The example in http://www.ccnx.org/pipermail/ccnx-dev/2011-May/000414.html envisions a VPN overlay scenario.

7.2 Quantitative Comparison

Since CCNx only implements forwarding and caching functions (with FIB entries being manually configured), we focus our comparison on the memory management and fast-path performance. All applications are written using the C library provided in the latest CCNx distribution.

Memory management performance: Similar to We repeat our experiment in Section 6.2 by emulating an IPC-like pipeline between two applications. This case is of interest for IPC scenarios in which applications only require opaque 'pipelines' rather than versioning support at network level. Information here is mutable, i.e., the application takes care of any versioning. In order to emulate this with CCNx, we implement a node-local CCNx application that expresses between 5,000 and 10,000 interests in content, each piece being individually labeled with its own name */content/segment_no*. Furthermore, we place the individual content pieces into the local CCNx content store, avoiding the signing overhead when played out by the producing application directly. In our prototype, we run the same application as for the experiment in Section 6.2.

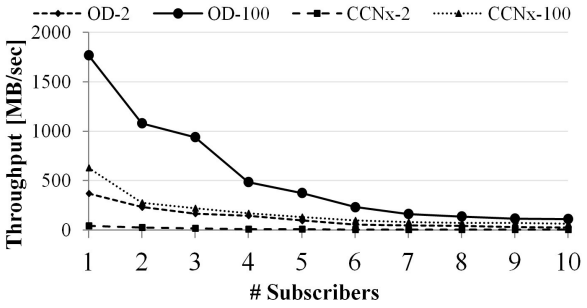

Figure 8. Memory Management Comparison

Figure 8 shows the results of our experiments for memory block sizes of 2kB and 100kB. Even though CCNx plays out data from the content store, the performance is significantly lower compared to our node design (labeled OD-2 and OD-100).

Fast path performance: For our second comparison, we create a chain of up to 14 nodes with the first one being the content producer and the last node being the consumer. At each node in the chain, additional content consumers are located (1 or 6, respectively). The nodes are connected in a 1Gbit/s dedicated Ethernet LAN environment. In order to eliminate timeout handling in CCNx, we implement a simple window-based mechanism for sending interests (of up to the window size) to the network stack. Also in this comparison, we differentiate a cached and non-cached case in order to single out the signing overhead at the producer. For our node design, we utilize both, the immutable as well as the mutable semantics, i.e., the former will incur rendezvous overhead while the latter case is relevant for cases where mutability is handled by the application. We use the same window-based mechanism in the immutable case, allowing for a fairer comparison with the window mechanism used in the CCNx case[4]. Figure 9 shows our results comparing our own prototype in the mutable mode (OD-Mutable-x) with the CCNx performance, each with 1 and 6 subscribers[5]. At one subscriber per node in the chain, our prototype sustains line speed throughout the entire chain, while degrading down to 80MB/s for the case of 6 subscribers. For validation, we can compare the CCNx performance with only one consumer being directly connected (CCNx-1 point for 1 node) with the number published in [6] for a 100Mbit/s test bed. The throughput reported is similar to the one we observe in our experiments, namely 10.4MB/s. However, the CCNx-1 performance continues to decrease along the chain of nodes, down to 0.38MB/s for 13 nodes.

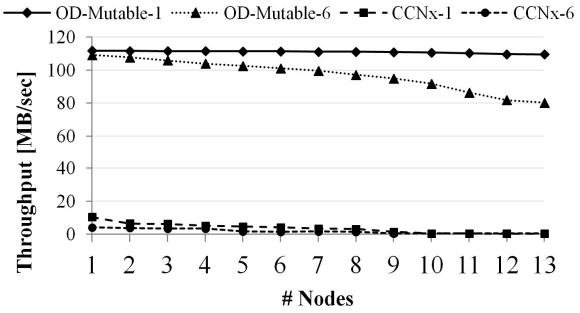

Figure 9. Throughput Comparison

Figure 10 compares the performance of CCNx (in cached mode) to our prototype with immutable semantics. The performance is more balanced, with similar performance degradations. The results, however, leave out the possibility to optimize the slow path in our prototype through, e.g., caching of rendezvous results or forwarding identifiers.

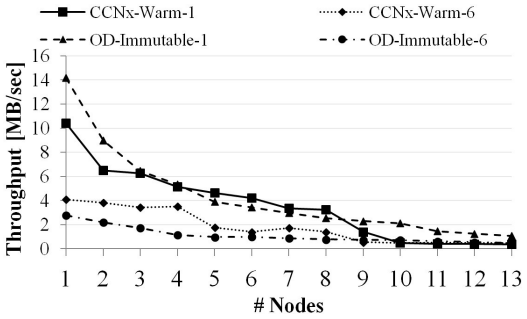

Figure 10. Throughput Comparison

[4] A thorough comparison depends on many factors related to the rendezvous function, such as location within the overall topology, the particular realization (centralized vs. distributed) and many others. Furthermore, the comparison here is somewhat unequal since there is nothing in CCNx to compare against when it comes to the overhead that would occur from distributing FIB entries to all forwarding nodes.

[5] We omit the non-cached case since the obtained performance numbers never exceeded 170kB/s.

From these measurements, we can infer that the used domain-local forwarding mechanism, based on [7], clearly outperforms the PIB/FIT-based lookup mechanism in CCNx. Given the simplicity of the mechanism to match link-local link identifiers to the Bloom filter received in each packet, this result is hardly surprising. The performance, however, is more balanced when adding slow-path operations to the scenarios.

8. CONCLUSIONS

Increasing interest in information-centric networking creates the need for a flexible and extensible development platform to allow research in the area to progress. We address this need, and make the following contributions in this paper.

Firstly, we presented a node design that allows for continuous development and experimentation in the area of information-centric networking. Its design is modular, utilizing the Click router platform. It can therefore easily accommodate future developments in the various areas defined by the main functions of the underlying network architecture. We also showed that this flexibility does not come at the price of performance since initial results are very promising. Secondly, our work provided an insight on how to design and build a network node that breaks with any historical baggage of the IP world. Here again, the Click platform proves essential in fulfilling our goal. The results from our design can prove useful for other clean slate designs that break more fundamentally with IP than merely improving its control plane. Thirdly, contrasting our work against the CCNx prototype provided useful insight into qualitative as well as quantitative differences that stem from these efforts. We concentrated in our work not only on quantitative differences that arise from implementation choices but also on qualitative differences that lie deeper within architectural foundations. Hence, we see our work potentially stimulating an architectural discussion beyond our initial points made in this paper.

REFERENCES

[1] D. Trossen, M. Sarela, K. Sollins, “Arguments for an Information-Centric Internetworking Architecture”, ACM Computer Communication Review, April 2010.

[2] G. Csardi and T. Nepusz. The iGraph software package for complex network research. *InterJournal Complex Systems (2006)*.

[3] J. Day. Patterns in Network Architecture - A Return to Fundamentals. *Prentice Hall, 2008*.

[4] P. Flegkas, V. Sourlas, G. Parisis and D. Trossen, “Storage replication in information-centric networking”, *In Proc. of the 17th IEEE ICON 2011*.

[5] B. Heller, D. Erickson, N. Mckeown, R. Griffith, I. Ganichev, S. Whyte, K. Zarifis, D. Moon, S. Shenker and S. Stuart. Ripcord: a modular platform for data center networking. *In Proc. of ACM SIGCOMM 2010*.

[6] V. Jacobson, D. K. Smetters, J. D. Thornton, M. Plass, N. Briggs, R. Braynard, “Networking named content”, Communications of the ACM, VoL. 55, No. 1, 2012

[7] P. Jokela, A. Zahemszky, C. E. Rothenberg, S. Arianfar, and P. Nikander. LIPSIN: line speed publish/subscribe inter-networking. *In Proceedings of ACM SIGCOMM 2009*.

[8] E. Kohler, R. Morris, B. Chen, J. Jannotti, and F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst*. 18, 3 (August 2000), 263-297.

[9] T. Koponen, M. Chawla, B. Chun, A. Ermolinskiy, K. Kim, S. Shenker, and Ion Stoica. A data-oriented (and beyond) network architecture. *In Proceedings of SIGCOMM 2007.*

[10] P. Nikander, B. Nyman, T. Rinta-Aho, S. Sahasrabuddhe, and J. Kempf. Towards software-defined silicon: Experiences in compiling click to netfpga. *In 1st NetFPGA developers workshop (2010)*.

[11] L. Popa, A. Ghodsi, and I. Stoica. HTTP as the narrow waist of the future Internet. *In Proc. of Hotnets 2010.*

[12] J. Scott, J. Crowcroft, P. Hui, and C. Diot. Haggle: a networking architecture designed around mobile users. *In Proceedings of IFIP WONS 2006*.

[13] D. Clark, J. Wroclawski, K. R. Sollins, R. Braden, “Tussle in Cyberspace: Defining Tomorrow’s Internet,” in IEEE/ACM Transactions on Networking, Vol. 13, No. 3, 2005

[14] D. Perino, M. Varvello, “A Reality Check for Content Centric Networking”, *Proc. of SIGCOMM workshop on Information-centric Networking (ICN)*, 2011

[15] J. Rajahalme, M. Särelä, K. Visala, J. Riihijarvi, “On name-based inter-domain routing”, Computer Networks 55:975–986, 2011

[16] A. Ghodsi. T. Koponen, J. Rajahalme, P. Sarolahti, S. Shenker, “Naming in content-oriented architectures”, *Proceedings of SIGCOMM workshop on Information-centric Networking*, 2011

[17] A. Ghodsi, T. Koponen, B. Raghavan, S. Shenker, A. Singla, J. Wilcox, “Information-Centric Networking: Seeing the Forest for the Trees”, ACM Workshop on Hot Topics in Networks (HotNets-X), 2011

[18] C. Tsilopoulos, G. Xylomenos, “Supporting Diverse Traffic Types in Information Centric Networks”, *Proceedings of SIGCOMM workshop on Information-centric Networking*, 2011

[19] Dmitrij Lagutin, "Securing the Internet with digital signatures," Doctoral dissertation, Aalto University, Espoo, Finland, December 2010