IEICE TRANS. COMMUN., VOL.E96–B, NO.7 JULY 2013

INVITED PAPER *Special Section on Internet Architectures, Protocols, and Management Methods that Enable Sustainable Development*

A Node Design and a Framework for Development and Experimentation for an Information-Centric Network

George PARISIS[†*], **Dirk TROSSEN**[†*], *Nonmembers*, *and* **Hitoshi ASAEDA**[††a)], *Senior Member*

SUMMARY Information-centric networking has been touted as an alternative to the current Internet architecture. Our work addresses a crucial part of such a proposal, namely the design of a network node within an information-centric networking architecture. Special attention is given in providing a platform for development and experimentation in an emerging network research area; an area that questions many starting points of the current Internet. In this paper, we describe the service model exposed to applications and provide background on the operation of the platform. For illustration, we present current efforts in deployment and experimentation with demo applications presented, too.

key words: *information-centric networking, click router, publish-subscribe, future Internet*

1. Introduction

Information-Centric Networking (ICN) is increasingly attracting attention in the networking community. Several technological solutions within a range of architectures have been proposed, such as in [1]–[4], with differences but also commonalities that stretch across the approaches.

One important aspect is the design of a network node within such architecture(s). One strategy is to evolve the current IP-centric design with its socket abstraction through information-centric extensions. This may result in overlaying on top of IP, a potentially desirable outcome since it allows for 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 network nodes 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 today. Following this strategy, this paper presents the design and implementation of an ICN stack. Our architectural starting point is the one presented in [1] and elaborated in [2]. This work argues and lays the ground for a clean-slate information-centric architecture in which information is the first principle. Individual information items represent anything useful within a computational context, and they are identified through *labels*, effectively replacing the role of IP addresses in today's Internet. Information items are organized through *scopes*, each again identified through its own label. This allows for building directed acyclic graphs of information, manipulated through a *publish-subscribe service model*.

This service model is realized through three core functions. The first one, *rendezvous*, matches supply of information to demand for it. This process results in some form of (location) information that is used for binding the information delivery to a network location by the second function, *topology management and formation*, to determine a suitable delivery relationship for the information transfer. This transfer is finally executed by the *forwarding* function. With this control and data plane separation, routing and forwarding are decoupled, enabling to trade off options in state management between various network components. For instance, in our prototype, we remove flow-dependent state from forwarding nodes in favour of route computation during topology formation, inserting the state into the packet header. The architecture in [2] also allows for different realizations of the core functions through *dissemination strategies*. Hence, the core functions can be optimized within a given strategy, e.g., for catering to varying requirements of the underlying networks.

Any attempt to design a network node for such an environment needs to align with these architectural starting points. In addition, the network node design needs to accommodate the requirement to serve as a platform for an emerging and likely evolving area of research. Hence, any platform design needs to enable experimentation and development across a growing research community. Specifically, it is crucial to support experiments in real high-speed networks as well as in emulated environments in order to enable experimenting with solutions, such as inter-domain or global rendezvous solutions, for which real deployments at scale will be missing. Furthermore, with ICN penetrating a variety of environments that range from high-speed core networking over mobile to delay-tolerant environments, supporting various device platforms is crucial, too.

To present our contribution, we structure the paper as follows: We outline related research work (Sect. 2) before we delve into the network design aspects (Sect. 3). We then present an overview of our implementation (Sect. 4). Finally, we present several deployments in real-world testbeds and an experimental evaluation in an international testbed

Manuscript received December 10, 2012.
Manuscript revised February 21, 2013.

[†]The authors are with the Computer Laboratory, University of Cambridge, UK.
[††]The author is with the Network Architecture Laboratory, NICT, Koganei-shi, 184-8795 Japan.

[*]Their work is supported by the EU FP7 project PURSUIT.

a) E-mail: asaeda@nict.go.jp

DOI: 10.1587/transcom.E96.B.1650

(Sect. 5), before concluding the paper.

2. Related Work

Since the inception of the IP node architecture, there have been many attempts to rethink the design of a network node. In the area of content-centric architectures, Haggle [5] and CCN [3] stand out. Haggle provides manipulations of a linked information graph based on publish/subscribe operations. The Haggle component wheel logically separates core functions for information dissemination in a plug-and-play manner. However, Haggle does not provide a layering structure but resides between the application and the network (in the model of a network that can include many functionalities of today's protocol stacks, such as at TCP/IP level). CCN extends the IP node design with a forwarding information database for a hierarchical naming system (based on the DNS). It also introduces a forwarding function that can be configured based on some strategy for selecting particular interfaces for given named data. However, the function of routing, i.e., the population of the forwarding table based on availability of named data in different domains, is currently undefined. Only a broadcast strategy has been presented so far which is not suitable for most communication scenarios. Any progress of this aspect is likely to have an impact on the overall node design. Furthermore, CCN aims to provide a shim layer between a DNS-based naming at application level and the IP substrate.

Another area of development is that of data-center networking (DCN). An example here is RipCord [6]. Based on the IP abstractions provided to applications, the node design separates core functions for topology management and forwarding for separate optimization. As a shim layer in existing nodes, it does not provide any particular layering structure that is different to that of IP.

Finally, we note the design of IP end nodes today. Endpoint-based abstractions are provided instead of information flow ones. The authors in [7], however, argue for HTTP and the DNS as the effective waist of the Internet. This lifts the IP node design onto the level of efforts like CCN or RipCord, providing manipulations of hierarchically named data. IP node designs separate functions like name resolution (DNS lookup), routing and fast-forwarding. However, layering within an IP-based node design is rigid with its transport functionality being directly exposed to applications. Overlaying enables introducting shim layers but often with significant performance penalties.

3. Network Node Design

The principles of our ICN architecture are now mapped to a node design. Along with the architectural principles, we take into account the following design considerations that must be met by our node design. Firstly, we aim to support dissemination strategies, enabling to change such strategies during system runtime. For example, an application could adapt the strategy for an information graph after its creation in order to extend the information from being node-local towards an entire network domain. Secondly, we want to preserve the modularity of the core functions as introduced in [1]. For example, as we design new forwarding mechanisms, the network node must be able to incorporate such new mechanisms without breaking existing deployments and affecting any previous functionality. This is achieved by assigning a different dissemination strategy to a part of the information structure that utilizes the new functionality. Thirdly, managing the information about available information, e.g., its suppliers and consumers, is a crucial task of our network node; therefore, our design should provide an approach to managing this information that is independent from the particular choices for its realization. Hence, our design needs to afford extensibility and flexibility.

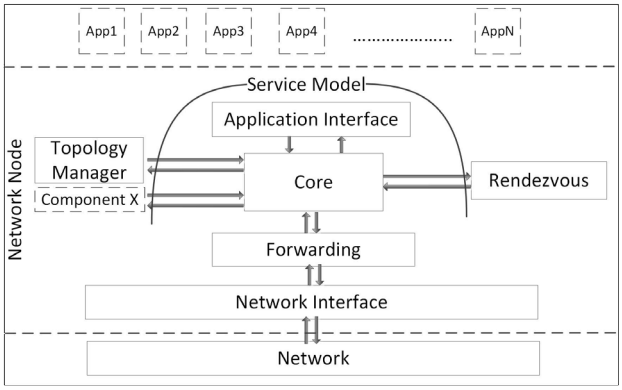

Fig. 1 Network Node Design: Software components implementing the architectural core functions and enabling the interaction of applications with the network through the service model, as explained in Sect. 3.3.

These considerations lead us to a modular design for our network node, illustrated in Fig. 1. Here, various software components implement the core functions of the underlying architecture, i.e., rendezvous for matching the demand and supply of 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 design aspect is the ability for a node to assume any role in the network. For instance, the same network node that realizes all three core network functions can take the role of a rendezvous node that, in cooperation with other nodes in a network, provides information resolution among publishers and subscribers. Other nodes could assume mere forwarding functionality. Forwarding could be offloaded in hardware, while missing all other node components such as topology formation. This enables considerable flexibility in experimental deployments, particularly in an early phase of development.

The component-based node design also drives our implementation approach with the goal to develop a unified framework for deployment and experimentation that spans from experimentation in dedicated networks over overlays to emulated as well as large-scale networks.

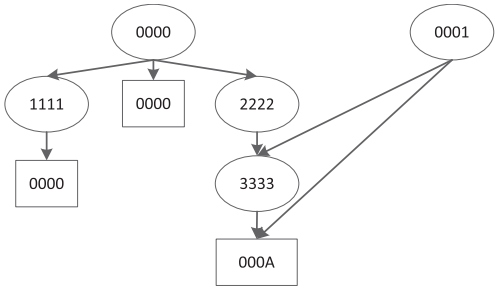

Fig. 2 An example information structure.

3.1 Information Structure and Semantics

The fundamental change compared to an IP-based architecture is that information is at the core of our network architecture. Therefore, the management of information is a very important task that is undertaken by the rendezvous network function. Depending on the assigned dissemination strategy, the rendezvous function can be implemented in different ways: (1) locally to a network node to manage information across applications running within the same node, (2) in a centralized network node for small-scale networks, (3) in a distributed, potentially hierarchical, fashion for managing large information spaces across one or more network domains (e.g., as the one presented in [8]).

Aligned with [1], [2], *information items* are the central principle in our architecture. These represent any information pertaining to the computational task that is executed via the network (e.g., delivering a video from a sender to many receivers). Each information item is identified through its own *statistically unique fixed size label*. Furthermore, any number of information items can be nested under a *scope*, which is again identified through its own label (see Fig. 2 for an example of a resulting information structure). Labels carry no semantics and are meaningless to most network components and applications. Their meaning is restricted to the entities that produce and consume the information items, while the network entities merely transfer the information. Global uniqueness of labels within the overall structure is not a prerequisite although it may be enforced in specific sub-graphs by the rendezvous function. Information can be identified in the context of a scope using a fixed size label, and the absolute path from a root of the graph (with possibly more than one path that may exist) must be used when accessing the service model (see Sect. 3.3).

Figure 2 depicts an information graph that is managed by the rendezvous function. We observe that an information item can be published under multiple scopes. Scopes and information items are identified using one or more full identifiers starting from a root scope. With that, the item with label *000A* is identified using the following identifiers: */0000/2222/3333/000A*, */0001/3333/000A* and */0001/000A*. A publisher or subscriber can use any of these identifiers, depending on its potentially partial knowledge of the information structure, to advertise or subscribe to this information item using the service model exported by our network node. As an example, the information item with identifier */0000/1111/0000* has the same label as the root scope */0000* and scope */0000/0000*. As long as all full identifiers of a scope or information item are unique, they are legitimate.

Our node design provides support for a wide range of possible information semantics. Let us consider the case in which individual information items are *immutable*. Here, an application-specific item is identified with a (statistically) unique label under a given scope, for instance by hashing its content or a human readable and memorable name. As an example, we could assume each version of a document being labelled individually. For the application, there needs to be an additional information exchange that disseminates the version identifiers. We also consider *mutable* information, with any item carrying the same identifier. Hence, the application needs to take care of any issues arising from this mutability, without having the capability to rely on (statistically) unique labels. Mutable items are important when realizing, e.g., live video delivery, in which video chunks are published using the same identifier. Finally, determining the identifiers through an *algorithmic function* represents a hybrid of the previous approaches. For instance, a sequence number scheme could be included into the hashing function that creates an individual identifier. Assuming that publishers and subscribers are aware of this relation, the seemingly random identifiers can be associated with each other, e.g., when re-assembling the received fragments.

3.2 Network Functions

The Rendezvous Component receives and processes all requests that are published by applications running locally or in other nodes. Upon receiving such requests, it matches potential publishers and subscribers 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 enforcing 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.

The Topology Management and Formation (TM) Component realizes the overall delivery topology management and the formation of delivery graphs for pub/sub relations. For that, the TM updates the topology information when network nodes join or leave the network and creates the necessary forwarding information (and the necessary state in the network) when requested. In a typical scenario, the rendezvous component may request such forwarding information after successfully matching 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 shortest-path approach, while topology formation is reduced to finding the interested applications

in this node when disseminating information purely within the boundaries of a single network node.

The Forwarding Component receives publications and forwards them to the network and/or to node-local components. For this, it maintains necessary state and may coordinate with the TM component, e.g., when new nodes attach. Optimized network nodes may only realize the forwarding function using, e.g., hardware-optimized mechanisms.

3.3 Service Model

Applications as well as node components interact with each other and with the network through the service model, exposed as an internal API. This service model enables the manipulation of information in a publish/subscribe fashion. While the manipulation through the service model is similar to many existing event-based pub/sub systems, our description outlines these operations in the context of utilizing the core functions of the architecture.

3.3.1 Publish/Subscribe Semantics

When scopes are published or un-published, subscribers of their parent scopes are notified about new or deleted scopes by the rendezvous network function. Such notifications are published to a well-known scope to which all nodes in the network are subscribed. When a publication reaches a destination, according to the forwarding function, the interested application running in that node is notified by an up-call notification as described later in this section.

When new information items are advertised, the rendezvous function matches all publishers (including pre-existing ones) with any subscribers and publishes a topology formation request to the topology management function. Respectively, when information items are un-published, rendezvous takes place and if there are any publishers left, a topology formation request is published to the topology management function. The topology management function, upon receiving such requests, calculates forwarding information that is published to one more publishers. For instance, our implementation of the intra-domain forwarding assumes LIPSIN [9] identifiers. Therefore, the topology management function calculates an identifier that defines a source-based multicast tree from a publisher to a number of subscribers and publishes it to the respective publisher.

A subscription to a scope triggers the rendezvous function to look for any scopes or information items that are published under that scope and acts as follows: for any scope, it publishes a *notification* about its existence to the subscriber. For all information items, one or more publishers that previously advertised the item are notified to publish the respective data after the topology management function creates the forwarding information.

Un-subscribing removes the subscriber from the scope's subscriber list. If there are no other publishers and subscribers as well any items or sub-scopes, the scope is deleted from the information graph. After receiving a subscription to an information item, the rendezvous component matches all publishers with this subscriber and any other previously subscribed one and publishes a topology formation request. Then, the necessary forwarding information is calculated and published to one or more publishers. When un-subscribing, 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 takes place again and one or more publishers are notified to publish data for the specific piece of information. A publisher publishes data for a specific information item using as a reference one of the potentially many identifiers of the published item. A publisher issues such request if rendezvous has already taken place, i.e., it is already known by the network node how to forward data for the identified information item. There can also be cases where a publisher requests to immediately forward data. In such cases, it is assumed that the publisher itself is able to construct the forwarding information. An example of such operation is when the TM publishes data (i.e., responses to previous requests) to the rendezvous component.

3.3.2 Asynchronous Upcalls

In most cases, requests sent by an application to the network stack will result in further actions taken by the rendezvous and topology management functions. These actions may result to one or more notifications towards the network node that issued the request, which in turn may notify one or more applications. These notifications are asynchronous. A *new* and *deleted scope notification*, along with the scope identifier, is sent to an application when a scope is created or deleted under a scope to which the application is subscribed. The *start publish notification* is sent to an application, which 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. A start publish notification contains the identifier of the item to be published. 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 was previously sent. Finally, a *published data notification* is sent to an application (subscriber) when data has arrived for a specific publication. This notification contains the information identifier as well as the received data.

Let us now present a simple example where information is disseminated within a single network domain running a single rendezvous node (RV) and topology manager (TM), as shown in Fig. 3. Let us also assume that the forwarding function is realized by a number of forwarding nodes (FW) using LIPSIN identifiers. Subscribers *S1*, *S2* and *S3* have already subscribed to an information item that is later advertised by publisher 1 (*message 1*). The advertisement is

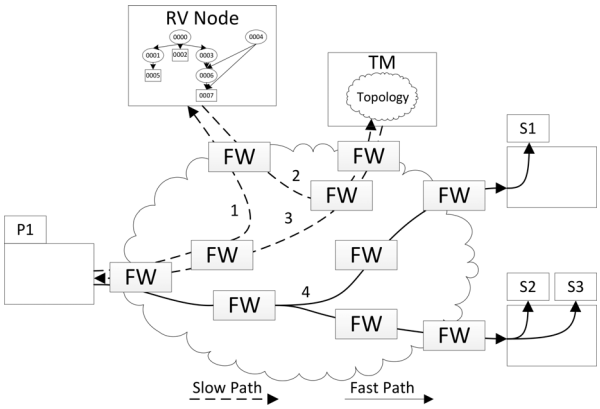

Fig. 3 Intra-domain information dissemination.

forwarded to the RV node. Note that all nodes in the domain are assigned with at least one LIPSIN identifier that is used to forward pub/sub requests to the RV node during each node's bootstrapping. The RV node then matches the availability of information with the interest for it and publishes a topology formation request to the TM (*message 2*), using a pre-configured LIPSIN identifier. The TM creates a LIPSIN identifier from *P1* to *S1*, *S2* and *S3* (the last two running in the same host) and publishes it to *P1* (*message 3*). *P1's* network stack maps the received forwarding identifier to the advertised information item and notifies the application about the existence of subscribers. Finally, it is up to the application to publish data using this information identifier (*message 4*). For instance, in a live TV scenario, *P1* can constantly publish video chunks until no subscribers exist for the information identifier (at this point a similar message sequence causes *P1's* network stack to notify the application that no more subscribers exist).

3.4 Core Component

This component has three 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 notifications from the network and notifies all interested applications. Note that these notifications are sent as publications using a predefined information identifier.

3.5 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. We foresee interfaces for Ethernet or Bluetooth as well as overlays on top of IP, TCP or even HTTP. The Application Interface Component provides the means for an application to interact with our network stack.

4. Node Implementation

We now describe the current implementation of our network node, which is aligned with the component-based node design, presented in Fig. 1.

4.1 Platform Choice

Our node implementation is based on the Click modular router [10] platform and is publicly available under GNU GPL2 license [11]. All node components, presented in Sect. 3, are implemented as Click components. Click complements our efforts because (a) its communication components support a variety of transport mediums, ideal for experimenting with IP replacements, (b) it allows for different application-facing interface techniques, and (c) the notion of Click components enables the development of our main node components in a way that eases portability between kernel and user space as well as across operating systems.

Currently, we run our prototype in Linux (user/kernel space), FreeBSD (user/kernel space), Mac OS X (user space), Android (user space) and integrated into ns-3 [12], enabling emulation environments. A user space deployment supports quick prototyping as well as experimenting in environments where throughput is not the primary metric. On the other hand, a kernel space deployment is more efficient in terms of performance. Note that most of the core network node functionality is independent of the mode in which our node is running. Only some operating system dependent functionality varies across different operating systems. The integration with ns-3 allows for moving between real deployments and emulations, using one or more powerful machines or even simulations of the same functionality with virtually no programming overhead. Only minor changes to publish/subscribe applications are required to run the same deployment in emulated or simulated mode.

4.2 Interfacing Applications

In this subsection, we elaborate on the implementation details of our network node. We present how communication among different network entities is achieved and how network publications look like. The Application Interface Component (see Fig. 1) interfaces our network stack towards applications. Usually, applications interact with the networking software of an operating system via system calls. 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 need to open such a socket and then interact with the network stack using existing system calls. This comes closest to introducing a new set of system calls. Netlink sockets provide two further advantages. First, applications use (almost) the same API for accessing the network stack regardless of the mode in which the implementation

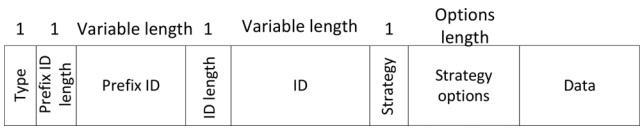

Fig. 4 Publish/subscribe request format.

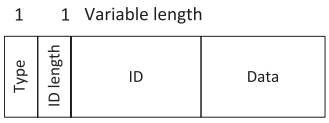

Fig. 5 Upcall format.

runs. Second, in kernel space, Netlink sockets are Click-friendly since the network stack receives socket buffers that are wrapped into Click packets with no extra memory cost.

The network stack receives application requests in the process context, which immediately unblocks each process by placing the received data buffer in a FIFO queue. The buffer process is deferred to a separate Click task. After wrapping each received buffer, the Application Interface component annotates the packet using the Netlink port at which the sender application expects all events.

4.3 Core Component

The Core Component is at the heart of the node design. All Click packets received by the Application Interface Component are annotated with an application identifier. Click packets received by other Click components are annotated with the Click port with which the core component 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 components in the network. Instead, a statistically unique node label that identifies the network node from which a request is sent is (self-) assigned by the Core Component. The memory buffer format (wrapped in a Click packet) expected by our network node from applications is shown in Fig. 4. Note that this Click packet never traverses the network. It is only pushed from applications or Click components like the rendezvous and topology manager ones to the Core component.

All different types of publish/subscribe requests are described in the next subsections. The *Prefix ID* is the identifier of the scope under which the identified (with *ID*) scope or information item resides. Its length, which is not constant, is denoted in the byte before the *Prefix ID*. Moreover, *ID* is a variable length identifier that can be a single scope or information item label or a full identifier when a scope or item is published under multiple scopes. The *strategy* is a single byte identifier of the dissemination strategy that the scope or information item is assigned with. *Strategy options* are arbitrary data required by a dissemination strategy. This data can be interpreted by one or more network core functions according to the specified dissemination strategy. For instance, we implement an *Implicit Rendezvous* strategy with which a publisher can publish data directly to a subscriber without separate rendezvous. The strategy option, in this case, is a LIPSIN [9] identifier that will be used to forward the data in the network. Finally, the data field is included only in publish data requests.

Figure 5 presents the format of a Click packet that is sent from the Core component to interested applications or other Click components as described in Sect. 3.3. The type of the notification (e.g., *start publish*) is the first byte. The variable length identifier of the information item (for start and stop publish notifications) or scope (for new or deleted scope) notifications follows. Finally, when the notification is about published data by a publisher, the actual data reside at the end of the buffer.

Processing Publish & Unpublish Requests. Whenever the Core component receives such request, it checks if any other application or Click component has previously advertised the same scope or item. In the former case, it creates a publication with the initial request as the data. Then, according to the dissemination strategy, it publishes the data to the appropriate rendezvous component. In case that another application or Click component 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. When requests are published to the rendezvous node via the network, the Core component uses the */RVSCOPE/nodeID* as the information identifier, where *RVSCOPE* 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 it as the publisher of the requested scope or information item.

Processing Subscribe & Unsubscribe Requests. 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. Un-subscribe requests are published to the rendezvous node when no further applications are subscribed to a scope or information item.

Processing Publish Data Requests. The Core component publishes data on behalf of an application or a Click component only when the information item has been previously advertised and rendezvous has taken place, i.e. if the Core component holds a forwarding identifier to one or more subscribers. Applications are notified by the Core component when such a forwarding identifier is assigned to an information item (see Sect. 3.3).

Handling Network Publications. The Core component receives all publications destined to this node from the forwarding component. It then looks up its local index to find potential subscribers. A Published Data notification (along with the data) is sent to all interested applications and Click components. The Core component does not pro-

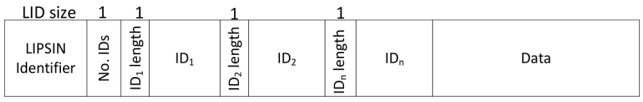

Fig. 6 Network publication format.

cess the data of received publications — this is left to the receiver. All packets that cross the network are publications that have the format presented in Fig. 6. Note that a network publication may have more than one information or scope identifiers, since, as described in Sect. 3.1, a scope or an information item can be identified by multiple identifiers, starting from a root of the information graph. Anything that can be published in the network must, therefore, have one or more identifiers and is always included in the data. All publications include forwarding information as a trailer. For instance, our intra-domain implementation utilizes LIPSIN identifiers for efficient forwarding in a single domain. Let us assume that a network node needs to forward a subscription to a rendezvous network node. For this, the subscribe message (Fig. 4) will be included in the data of a network publication (Fig. 6). The LIPSIN identifier that will be used is one (of potentially many) acquired by the publishing network node during its attachment to the network. This identifier will be used to forward the publication to the rendezvous node.

Processing Rendezvous Notifications. 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 notify a publisher. Initially, the core component 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 item. 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 LIPSIN mechanism [9] for forwarding within single domains. For this, it maintains a forwarding table that maps link identifiers (LIDs) to Click ports that point to a Click component that can access the network (e.g., a *ToDevice* component for Ethernet networks). Another LID is used to "connect" the Forwarding with the Core component. If such a LID is included in a LIPSIN identifier, the forwarding component will push the packet to its Core component. In this way a network node is instructed to process a network publication rather than merely forwarding it.

4.5 Interfacing the Network

We utilize Click components for communicating with other network nodes, supporting Ethernet via the *FromDevice* and *ToDevice* Click components as well as communication over raw IP sockets. The former can be used when experimenting in a LAN or VPN, while the latter is appropriate when overlaying on top of IP. Nodes (especially forwarders) may have multiple instantiations of the aforementioned components. The Forwarding component also allows for running our network stack in a mixed mode where a node may bridge LANs over an IP network, with individual LANs running the network stack over Ethernet. This allows for complex deployments as well as experimentation.

4.6 Example Applications

Let us briefly present examples that show how diverse computational tasks can use the exported service model and satisfy their networking requirements.

Managed Caching. We have implemented an approach for content placement, which can directly be realized on top of our node implementation [13]. Caches in the network receive content by subscribing to the respective information items and make themselves available as replica holders by republishing them. The topology manager uses a shortest path algorithm in order to select the best publishers for a specific set of subscribers.

Video Transmission. We have also 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 un-subscribing from the information ID, the publisher's node receives a new forwarding identifier that defines the revised multicast tree. The intra-domain multicast support results in a video delivery to multiple receivers at a reasonable speed with full SD video resolution that is realized within the international testbed described in Sect. 5.1.

Audio and Video Conference in Android. For demonstrating our Android platform realization, we have implemented an audio and/or video conference between multiple mobile users. Each conference session is represented as a scope, published under the application's root scope. The video and audio streams are published as separate scopes under the session scope. A user can join a session by publishing an audio and/or video information item under the respective scopes. To receive streams from other users in the session, a user simply needs to subscribe to the respective scopes.

Fig. 7 International testbed.

5. Deployment, Feasibility and Experimentation

5.1 Deployed Testbeds

We have deployed our network node prototype in several testbeds natively on top of Ethernet or as an overlay running on top of IP. Our major testbed is an international one that interconnects 10 major universities and institutions across the world. Eight European sites are connected with one in Japan (NICT) and another one in the US (MIT). All sites are connected via OpenVPN, which exports a virtual Ethernet device to all machines in the testbed. In total, we interconnect 40 machines in a graph topology containing one Topology Manager and one Rendezvous node that handle all publish/subscribe and topology formation requests, respectively. We have also performed tests in a high-performance Ethernet network consisting of 15 machines. There, the deployment runs in kernel space where we achieve line speed performance when forwarding publications across the network. Finally, we have deployed our prototype in overlay topologies of more than 100 PlanetLab nodes.

5.2 Feasibility and Experimental Evaluation

Let us now present a preliminary evaluation of our node design in our international testbed, depicted in Fig. 7, showing the feasibility of our network node design as well as its performance characteristics. The physical topology of our testbed is shown in the world map in Fig. 7. Each organization (listed on the right side of Fig. 7) hosts a number of physical hosts that run one or more virtual machines. Note that the numbering of the organizations corresponds to the physical sites depicted in the world map, and is independent of the numbering used in the ICN overlaid topology. On top of this physical topology we have deployed an ICN topology, depicted on the top left part of Fig. 7. Table 1 presents the ICN nodes in the overlaid topology for each organization. This overlaid topology forms a full mesh of 8 machines, one from each organization. The rest of the machines from each site are attached to a node that is connected to the core of the network. For overlaying purposes, all nodes participate in the same OpenVPN, whose server runs in Essex University, UK. Therefore, a direct ICN link, e.g., between node *6* (Cambridge University) and node *8* (NICT), incurs propagation delays to and from the OpenVPN server, with the OpenVPN server's packet processing capacity being the bandwidth bottleneck as observed by ICN applications.

Table 1 ICN topology nodes.

Organization	Nodes in ICN Topology
University of Essex	1, 9
Aachen University	2, 10, 18
Centre of Research and Technology	3, 11, 19, 20
AUEB	4, 12, 21, 22
Ericsson Finland	5
University of Cambridge	6, 13
CTVC	7, 14, 15, 23, 24, 25
NICT	8, 16, 17, 26, 27, 28-32

5.2.1 Fast Path Evaluation

Figure 8 presents the results regarding the fast path performance of our network node, namely the forwarding performance, as well as the capability of network nodes to locally process incoming network publications and dispatch them to the interested applications. To do so, a number of subscribers, running in one or more nodes, subscribe to a known information item under a root scope. Then, a publisher advertises this item and, therefore, rendezvous takes place. After the matching takes place, the topology manager is notified to form a LIPSIN identifier that is published to the publisher. After that, the publisher starts publishing thousands of publications using the same information identifier, effectively creating a potentially multicast channel between itself and any interested subscribers. In our topology, the Rendezvous and Topology Management and Formation network functions run in node *6* (University of Cambridge).

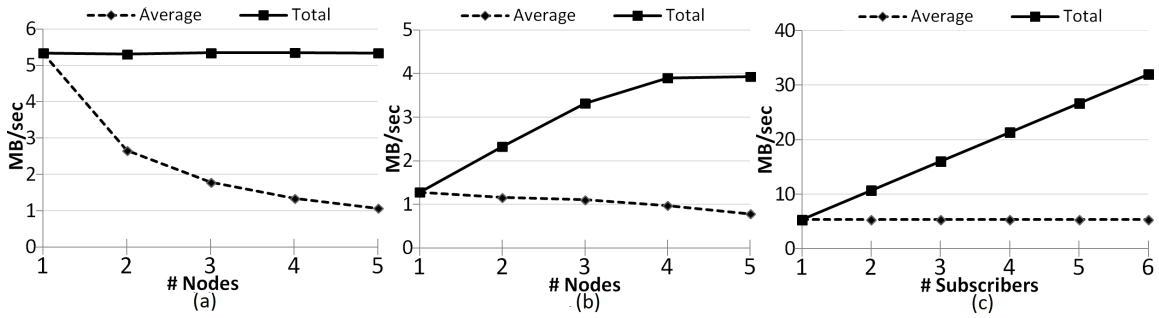

Fig. 8 Fast path evaluation ((a) 1 hop, (b) 5 hops, (c) 1 hop multiple subscribers).

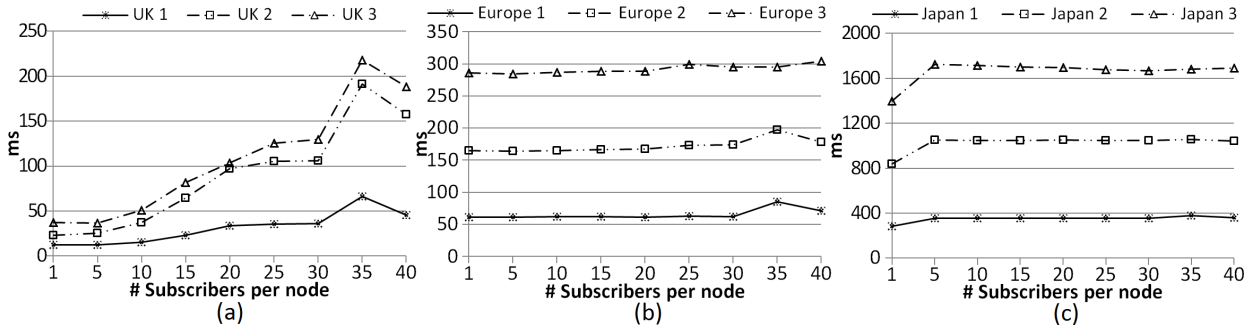

Fig. 9 Slow Path Evaluation ((a) UK, (b) Europe and (c) Japan Response Times).

Figure 8(a) presents the average application throughput of the subscriber application as the number of network nodes that express interest in the same information item increases with one subscriber per node. All subscribers run in machines residing at NICT. The publisher node also runs at NICT (node *17*). Although the publisher is only 1 hop away from the subscribers, this logical hop traverses through the OpenVPN server in the UK, which is the main reason for the performance reduction, as mentioned later. As we observe in Fig. 8(a), the measured throughput for a single subscriber is about 5.3 MB/sec, the maximum packet processing speed of the OpenVPN server. As the number of subscribers increases, the observed throughput decreases since all publications run via the OpenVPN server, although the ICN supports native multicasting. However, the total measured application throughput always stays at the maximum level limited by the OpenVPN server.

We repeat the same experiment, but place the publisher 5 hops away (at node *22*). The measured performance, shown in Fig. 8(b), is much lower than in the previous experiment, because of the shared nature of the OpenVPN virtual network. Here, the maximum throughput for a single subscriber is about 1.2 MB/sec. This happens because for each ICN topology hop, all packets are sent via the OpenVPN server, dividing the available bandwidth by 5. As more subscribers join, the performance does not drop significantly. This is because for all subscribers each network publication is sent only once from node *22* to node *17* (always via the OpenVPN server) and then it is multicast to the interested nodes. For 5 subscribers, the total measured application throughput approaches the maximum supported by OpenVPN.

Finally, we repeat the experiment by having multiple applications in a single node subscribing to the same information item while placing the publisher 1 hop away. Therefore, publications are sent only once to the subscriber node, which in turn clones and dispatches them to multiple applications. In Fig. 8(c), we observe that the average performance of each application is not affected by the number of local applications, effectively multiplying the total application throughput by the number of applications. As we mentioned before, preliminary experiments show that in high-speed testbeds our network node can cope with multiple subscribers or multiple forwarding links at Gigabit rates.

5.2.2 Slow Path Evaluation

Next, we present performance results regarding the slow path operations as realized by the two core network functions, rendezvous and topology management. In these experiments, the involvement of the rendezvous and topology management components is stress-tested, matching information in a structure such as the one depicted in Fig. 2 and calculating appropriate forwarding information. A publisher running in node *6* advertises 100,000 information items under a root scope. Then, a number of subscribers from each node randomly subscribe to a number of the advertised items. Each subscriber issues a subscription request every 100 ms. For each subscription, rendezvous takes place and, subsequently, the topology manager is notified to form a LIPSIN identifier from the publisher to the subscriber. Both core network functions also run in node *6* in

order to minimize the propagation delay until the publisher is notified to publish an information item. Upon such notification, the publisher publishes the minimum transfer unit of data. Finally, the subscriber receives this data, calculates and records the time difference since it issued the subscription. We call this the response time.

The testbed is spread across the world, therefore we separate response times in three graphs in order to group machines with similar propagation delays from the publisher (as well as the OpenVPN server). In Fig. 9(a), we observe that as the number of subscribers per node increases, so does the response time. Note that for 40 subscribers per node, 1280 subscribers issue subscribe requests, stressing both slow path network functions and the packet processing capacity of the OpenVPN server. For a single subscriber in each node the average response time for subscribers in the UK, running one hop away from the publisher, is about 12 ms (and 60 ms for 40 subscribers per node). For nodes *2* or *3* hops away from the publisher the response time increases only because every network publication is transferred via the OpenVPN server for each ICN hop. The values for the testbed nodes running in the rest of Europe are higher because of the higher propagation delays involved (61 ms for nodes *1* hop away from the publisher and for a single subscriber per node and 304 ms for nodes *3* hops away from the publisher and for 40 subscribers per node, see Fig. 9(b)). Finally, the response time for the testbed nodes running in Japan are even higher (see Fig. 9(c)) since the propagation delay is much higher. For instance, for nodes *3* hops away from the publisher, a subscription and the published data must travel multiple times, one per each ICN hop, from a node in Japan to the OpenVPN server and vice versa (leading to response times of more than 1 second).

6. Conclusion

Increasing interest in ICN creates the need for a flexible and extensible development platform to allow research in the area to progress. We address this need through the following contributions in this paper. Firstly, we presented a node design that allows for experimentation through its modular design. 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. These insights can prove useful for other clean slate designs. Thirdly, we presented results within a growing international testbed, outlining the potential of our platform for experimentation and demonstration of key ICN developments.

References

[1] D. Trossen, M. Sarela, and K. Sollins, "Arguments for an information-centric internetworking architecture," SIGCOMM Comput. Commun. Rev., vol.40, no.2, pp.26–33, April 2010.
[2] D. Trossen and G. Parisis, "Designing and realizing an information-centric Internet," IEEE Commun. Mag., vol.50, no.7, pp.60–67, July 2012.
[3] V. Jacobson, D.K. Smetters, J.D. Thornton, M. Plass, N. Briggs, and R. Braynard, "Networking named content," Commun. ACM, vol.55, no.1, pp.117–124, Jan. 2012.
[4] T. Koponen, M. Chawla, B.G. Chun, A. Ermolinskiy, K.H. Kim, S. Shenker, and I. Stoica, "A data-oriented (and beyond) network architecture," Proc. SIGCOMM'07, pp.181–192, 2007.
[5] J. Scott, P. Hui, J. Crowcroft, and C. Diot, "Haggle: A networking architecture designed around mobile users," Proc. IFIP WONS 2006, 2006.
[6] 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," SIGCOMM Comput. Commun. Rev., vol.40, no.4, pp.457–458, Oct. 2010.
[7] L. Popa, A. Ghodsi, and I. Stoica, "HTTP as the narrow waist of the future Internet," Proc. Hotnets-IX, pp.6:1–6:6, 2010.
[8] K.V. Katsaros, N. Fotiou, X. Vasilakos, C.N. Ververidis, C. Tsilopoulos, G. Xylomenos, and G.C. Polyzos, "On inter-domain name resolution for information-centric networks," Proc. IFIP Networking'12, pp.13–26, 2012.
[9] P. Jokela, A. Zahemszky, C. Esteve Rothenberg, S. Arianfar, and P. Nikander, "LIPSIN: Line speed publish/subscribe inter-networking," Proc. ACM SIGCOMM 2009, pp.195–206, 2009.
[10] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M.F. Kaashoek, "The click modular router," ACM Trans. Comput. Syst., vol.18, no.3, pp.263–297, Aug. 2000.
[11] PURSUIT EU Project, "Blackadder node implementation, available: https://github.com/fp7-pursuit/blackadder, last accessed: 17/02/2013."
[12] "Ns3 network simulator, available: http://www.nsnam.org, last accessed: 17/02/2013."
[13] P. Flegkas, V. Sourlas, G. Parisis, and D. Trossen, "Storage replication in information-centric networking," Proc. ICNC'13, 2013.

George Parisis received a Ph.D. in Computer Science from Athens University of Economics and Business in 2009. He is now a Research Associate in the Computer Laboratory at Cambridge University. His research interests include information-centric networking, publish/subscribe systems and high-performance storage systems.

Dirk Trossen is a Senior Researcher in the Computer Laboratory at Cambridge University. He is the technical lead for the EU project PURSUIT. He held prior positions as a Chief Researcher at BT Research and as a Senior Principal Scientist at Nokia Research. He is a Research Affiliate with the Advanced Network Architecture group at MIT CSAIL. He holds a Ph.D. from the Aachen University in Germany and published more than 65 papers and holds 27 international patents.

Hitoshi Asaeda is a Research Manager of Network Architecture Laboratory, National Institute of Information and Communications Technology (NICT). From 1991 to 2001, he was with IBM Japan, Ltd. From 2001 to 2004, he was a Research Engineer Specialist at INRIA Sophia Antipolis research unit, France. He was Project Associate Professor of Graduate School of Media and Governance, Keio University, where he was during 2005-2012. He holds a Ph.D. from Keio University.