PSIRP
Publish-Subscribe Internet Routing Paradigm
FP7-INFSO-IST-216173

DELIVERABLE D3.3

Progress Report and Evaluation of Implemented Upper and Lower Layer

Title of Contract	Publish-Subscribe Internet Routing Paradigm
Acronym	PSIRP
Contract Number	FP7-INFSO-IST 216173
Start date of the project	1.1.2008
Duration	30 months, until 30.6.2010
Document Title:	Progress Report and Evaluation of Implemented Upper and Lower Layer Function
Date of preparation	30.06.2009
Author(s)	Petri Jokela (LMF) (editor), Janne Tuononen (NSNF) (editor), Jimmy Kjällman (LMF), Pekka Nikander (LMF), Jari Keinänen (LMF), András Zahemszky (LMF), Dirk Trossen (BT), Borislava Gajic (RWTH), George Xylomenos (AUEB-RC), Dmitrij Lagutin (HIIT), Walter Wong (UNICAMP),
Responsible of the deliverable	Petri Jokela (LMF) Phone: +358 9 299 2413 Email: petri.jokela@ericsson.com
Reviewed by	George Polyzos (AUEB), Dirk Trossen (BT)
Target Dissemination Level	PU
Status of the Document:	Completed
Version	1.0
Document location	http://www.psirp.org/publications/
Project web site	http://www.psirp.org/

Table of Contents

1 Introduction

From the design phase of the project, the prototyping work is one major component in the development of the architecture. Thus, the implementation work started already in the beginning of the project in a close co-operation with the architecture work.

The PSIRP project started to work towards a publish/subscribe solution, where even the IP-based forwarding was not considered to be a necessary part of the system. This created a need for a prototype with a different structure than existing systems. The prototype development was divided into two separate areas, namely the *Lower Layer implementation* (LoLI), the *Upper Layer implementation* (UpLI). The LoLI is motivated by the need for a new kind of data handling at the end-host as well as for forwarding data packets due to the chosen publish/subscribe approach of the architecture. This host internal publication management, APIs, as well as forwarding mechanism implementation was assigned to the LoLI development. The requirements on locating publications and managing the network topology were considered to be “upper layer” tasks, necessary for the system. The implementation of these components has been done in the Upper Layer implementation part. Last but not least, different applications are created on top of the upper layer implementation.

As can be seen from the division of the implementation work, there is a clear time-line for the prototype. The LoLI part is essential for the development of the rest of the system, and it has been the most active part in the beginning of the project. After the development of LoLI has now reached a point where the system can be used, the Rendezvous and Topology development parts will now take the major role in the implementation process. One important target in the prototype development has been publishing the code outside the project. The first release of the LoLI prototype, called Blackhawk, has been published in the beginning of June, 2009.

Developing applications was not in the scope of the PSIRP project. While applications are still a crucial part of a complete networking environment, an Application Innovation Process was introduced to activate external partners in using the project’s implementation for developing and testing new kinds of publish/subscribe based applications and concepts.

This deliverable will provide an overview of the current progress of the different parts of the implementation work, along the abovementioned areas of implementation. We will further outline the planned evolution of the implementation as well as first evaluation steps for the implementation.

2 Current implementation progress

In this section we describe the current status of the prototype. First, we go through the LoLI implementation, including the FreeBSD based blackboard implementation (Blackhawk), packet I/O and forwarding operations as well as the security and PLA related implementation. In addition, the zFilter based forwarding implementation on a NetFPGA device is described. In the next section, we describe the status of the UpLI implementation, including Rendezvous and Topology parts, and after that, we present the results from a questionnaire sent out to the users of the implementation. The status of the application development is given, and finally, we describe the current status of integrating the LoLI and UpLI Implementations into a single demonstrator.

2.1 Lower layer Implementation

2.1.1 FreeBSD implementation overview

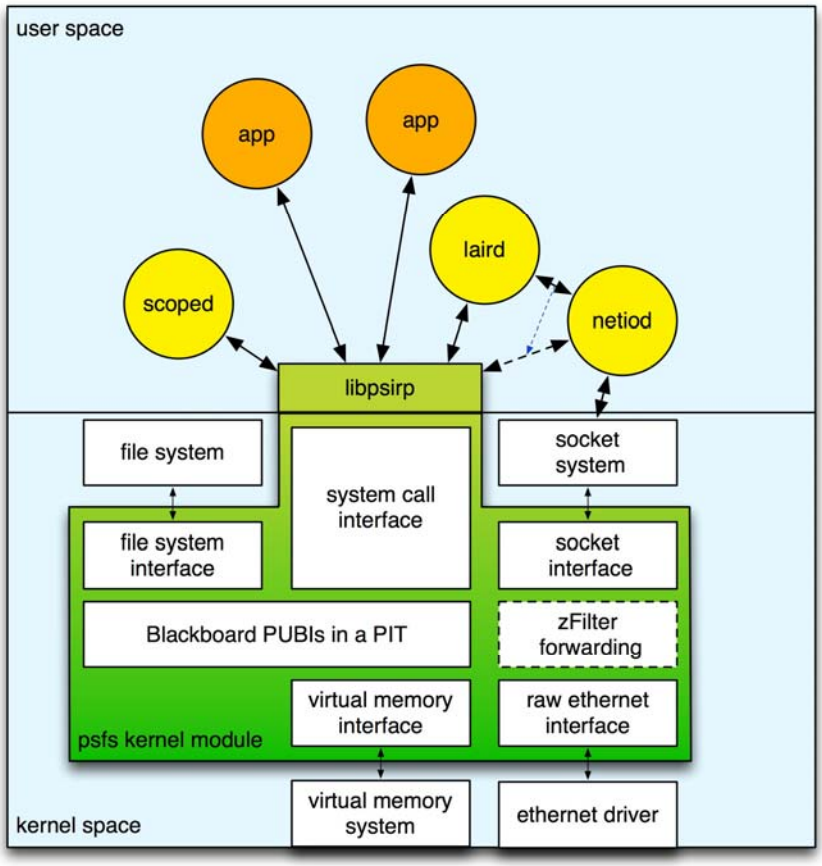

Figure 1 Blackhawk FreeBSD prototype architecture

Figure 1 shows the partially implemented architecture of the Blackhawk end-host implementation, including the blackboard based communication model, describing how helper applications can use the blackboard, and how the system communicates with the network.

From the figure we can see that the blackboard system is integrated with the operating system kernel's virtual memory system in order to efficiently implement the memory-object-based pub/sub model presented in D2.3. In addition, the implementation includes a set of helper applications in user-space for certain functions of the component wheel described in D2.3, such as node-local scope management (scoped), link-local rendezvous (laird), and network

I/O (netiod) including also packet forwarding. These helpers, as well as other applications, can use two alternative interfaces for accessing the blackboard: the libpsirp library (with versions for C, Python and Ruby), and the file system view. By default, helper applications are assumed to communicate with each other through the blackboard.

For efficiency reasons, some functions currently found in helpers can later be moved to the kernel. One example of this is the packet send/receive and forwarding functionalities.

2.1.2 FreeBSD blackboard implementation

The core of the Blackhawk prototype for FreeBSD is the in-kernel blackboard implementation. From a conceptual perspective, the blackboard is the place where publications are kept inside an end-node. Different entities can access the data in the blackboard by subscribing to it, and publishing is used to put the, either modified or new, data to the blackboard. Additionally, subscribers can register to events that are triggered when publications, including scopes, are updated.

The blackboard can be accessed through an API that is very similar to the memory object service of D2.3. This API offers functions for creating memory objects, publishing them, and subscribing to them. When the API functions are called, they further initiate system calls that are handled in the kernel.

Publications in the blackboard refer to the memory objects in the FreeBSD kernel and to the metadata associated with these objects. A memory object's metadata, in turn, points to a version object, whose metadata points to memory pages. All memory objects, versions, and pages have their own entries in a data structure called the Publication Index Table (PIT) shown in Figure 2. In fact, they also have their own RIds. This model allows memory objects with the same RId to have versions with different data content, and memory pages with the same content to be shared between different memory objects (something that, at this point, is not yet fully implemented).

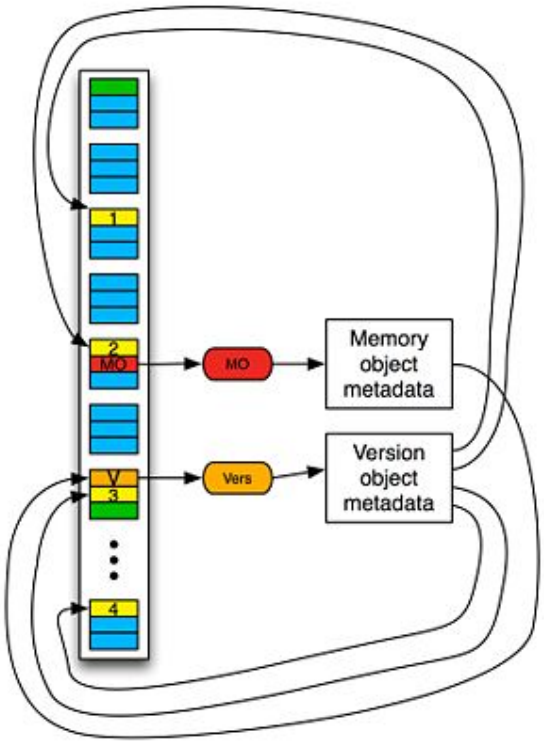

Figure 2 Publication Index Table

To programmers, publications are exposed as data structures with pointers to a data area (i.e., an array) and to the relevant metadata information, containing, e.g., the length of the data. In addition, these data structures provide a file descriptor that can be used for listening to FreeBSD's *kevents* that occur, when publications are re-published. An application can register to *kevents* using regular FreeBSD library functions. In the kernel, a *kevent* is triggered on the *vnode* that the publication's PIT entry also points to.

In this system, scopes are also stored as publications in the blackboard. The scopes are in essence structures listing RIds of other publications. As implied in the previous subsection, a special *scope daemon* handles instantiations and updates of such publications. In other words, it creates new scopes and adds RIds to existing scopes (starting from the root scope).

2.1.3 FreeBSD network I/O and forwarding implementation

In the current Blackhawk prototype, the network I/O is mainly handled by the *netiod* (network I/O daemon) helper application. It implements packet sending and receiving, the zFilter-based forwarding functionality, and publication creation out of the data chunks. In addition, support for FreeBSD link-layer sockets is implemented in the kernel module (psfs), and the local rendezvous function, supporting matching between publications for matching publications and subscriptions locally inside a node, in the corresponding helper (laird).

The Blackhawk implementation uses a binary packet format. From the high-level perspective, all packets contain a forwarding header, followed by a rendezvous header, and data (or alternatively, metadata). Constructed packets are sent to the network as broadcast Ethernet frames. Forwarding decisions are not made based on the link-layer addresses in the ethernet header, but using the FIds found in the forwarding header. The rendezvous header, on the other hand, defines a pub/sub operation (publish/subscribe data/metadata) for a specific publication identified by the (SId, RId) pair.

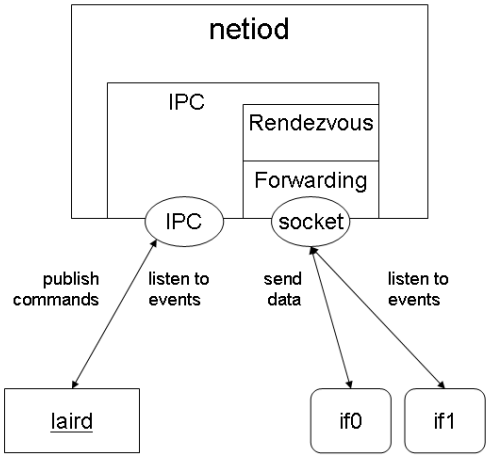

Figure 3 Network I/O Daemon

As seen in Figure 3, *netiod* uses sockets for sending and receiving packets over different network interfaces. When a packet is received, a *kevent* is triggered, and the packet is processed by the forwarding component. The forwarding component checks the FId in the packet header and if it matches the pre-configured link-Ids of any of the outgoing interfaces, the packet is sent out from the corresponding interface.

When a packet arrives to the actual destination node, we have to get it to the local processing module. In the current implementation, a node internal virtual interface is defined for this purpose. This interface has been assigned a separate link ID, indicating that the packet needs to be passed to node local processing. This is a temporary solution in the next phase; the publication matching will be done using SId/RId-based matching on the node. Thus, when constructing the Fid for the packet, the destination node's internal Link ID is included in the

Fid. If the FId matches the node's own virtual interface, the packet goes to netiod's internal rendezvous component for further processing. From there, subscriptions and metadata publication messages are published to laird through the blackboard (the current implementation also supports the use of node-local sockets for publication delivery) and packets containing the payload data of publications are handled within the *netiod*. Once a publication is completely received, it is published locally. The current implementation does not support subscribing only to partial publications.

For outgoing traffic, the *laird* informs the *netiod* to send out packets in case of node-local publish and subscribe events. First, the *netiod* creates the rendezvous and forwarding headers, and once the packet is ready, the forwarding component makes the decision which interfaces to use to deliver the packet. As a special case, if the packet is not given any Fid (it is all-zero) the packet is sent out using the default FId, leading to a local rendezvous node (see Section 2.2.1 for more information).

2.1.4 Security and PLA implementation

The PLA functionality has been refined in the prototype. An access control mechanism for scopes, as described in Section 5.2.3 of D2.3, has been implemented. In this scheme, it is assumed that the scope identifier (Sid) is derived from scope owner's public key. The scope owner acts like a trusted third party and authorizes potential publishers to publish to a rendezvous identifier (Rid). Authorization is done by issuing implicit certificates tied to the Rid. As a results, any verifying node in the network can independently verify that:

- A packet is authentic and the publisher has a right to use the Rid
- The SId matches the public key of the issuer of the certificate, preventing an attacker from issuing certificates to itself.

The mechanism described requires that a publisher has a separate certificate for each of the RIds. Therefore, a PLA library now supports creating, loading and saving certificates at runtime, simplifying the certificate management in the nodes.

The current PLA implementation has not been integrated with the Blackhawk implementation – this is planned for the next iteration. There are two potential ways to accomplish this. Either the PLA functionality will be included into the networking daemon, and it will be used to verify and sign incoming and outgoing traffic. Or alternatively, the PLA library will be implemented as a separate helper function that utilizes the blackboard architecture. In the latter case, the PLA function would read publications from the blackboard and republish them after performing the required verifications or the signature generation. The first approach would probably be more efficient and simpler to implement. However, the latter approach utilizes the blackboard principle and allows more flexible use of the PLA functionality. For example, PLA could also be easily used for enforcing access control in the local scope.

2.1.5 NetFPGA forwarding implementation

To be able to test the forwarding mechanism on a faster hardware, we implemented the zFilter based forwarding functionality using a NetFPGA platform [NetFPGA]. This platform was selected due to the flexible implementation environment. The current NetFPGA forwarding implementation has roughly 500 lines of Verilog code, and it implements the main ideas of the basic zFilter forwarding. In this section, we will go deeper in the implementation and describe the changes we have made to the original NetFPGA reference implementation. The zFilter based NetFPGA forwarding implementation is available at the project web pages as an open source release.

The implementation

For the zFilter based implementation, we identified all unnecessary parts from the NetFPGA reference switch implementation and removed most of the code that is not required (Figure 4). The removed parts were replaced with a simple zFilter switch.

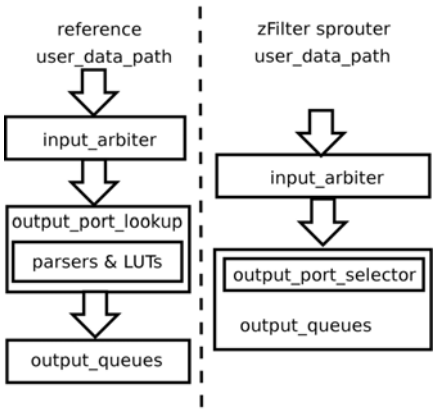

Figure 4 Reference vs. modified catapults

Our prototype has been implemented mainly in the new output_port_selector module. This module is responsible for the zFilter matching operations, including binary AND operation between the LIT and zFilter, and comparing the result with the LIT, as well as placing the packets to the correct output queues based on the matching result. The new module is added in output_queues. The detailed structure of the output_port_selector module is shown in Figure 5.

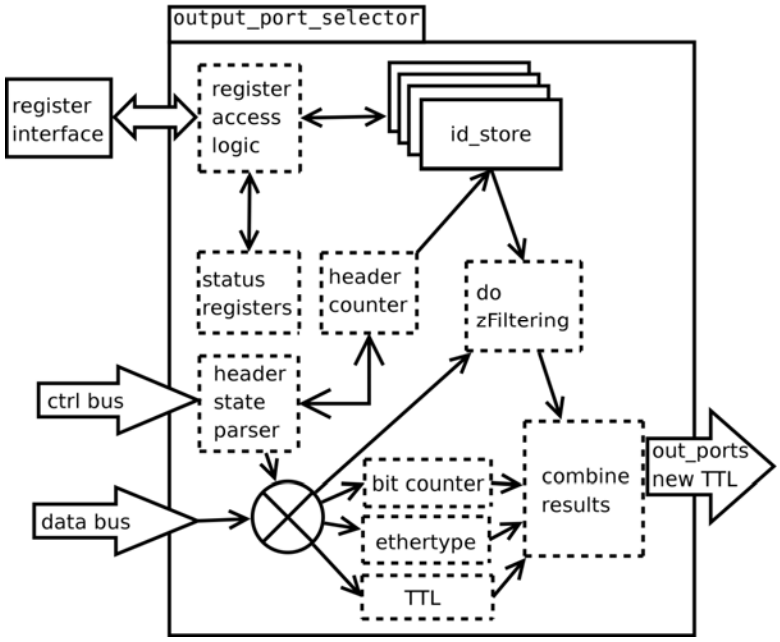

Figure 5 Structure of the output port selector module

The released version implements both the LIT and the virtual link extensions, and it has been tested with four real and four virtual LITs per each of the four interfaces. To identify zFilter based packets, the implementation uses a specific ethertype value. The implementation performs checks on incoming traffic and drops packets that fail the check.

Packet forwarding operations

All incoming data is forwarded from the input arbiter both to the store_packet module for storage into the SRAM, and to the output_port_selector module for processing. Packets arrive in 64-bit pieces one piece per clock cycle. Packet handling starts with initiating processing for different verifications on the packet as well as on the actual zFilter matching operation.

The incoming packet processing takes place in various functions, where different kinds of verifications are performed to the packet. The three parallelized verification operations, bit_counter, ethertype, and TTL, make sanity checks on the packet, while the do zFiltering function makes the actual forwarding decisions. In practice, for enabling parallelization, there exists separate instances of logic blocks that do zFiltering, one for each of the Link IDs (both for ordinary and virtual links). In the following, we go through the functions in Figure 5 function-by-function; in Figure 6, the operations are shown as a function of clock cycles.

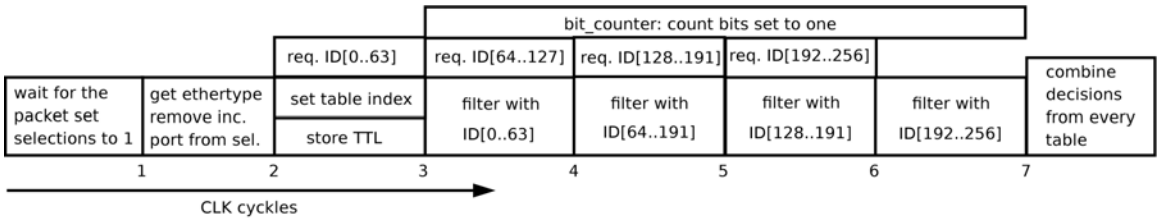

Figure 6 Data flow diagram

In do zFiltering, we make the actual zFilter matching for each 64-bit chunk. First, we select the correct LITs of each of the interfaces based on the d-value in the incoming zFilter. To track the forwarding decision status for each of the interfaces during the matching process, we maintain a bit-vector where each of the interfaces has a single bit assigned, indicating the final forwarding decision. Prior to the matching process, all bits are set to one. During zFilter matching, when the system notices that there is a mismatch in the comparison between the AND-operation result and the LIT, the corresponding interface's bit in the bit-vector is set to zero.

Finally, when the whole zFilter has been matched with the corresponding LITs, we can determine the interfaces where the packet should be forwarded which of the bits from the bit-vector are still ones. While the forwarding decision is also based on the other verifications on the packet, the combine results function collects the information from the three verification functions in addition to the zFilter matching results. If all the collected verification function results indicate a positive forwarding decision, the packet will be put to all outgoing queues indicated by the bit vector. The detailed operations of the verification functions are described in the following subsection.

Support blocks and operations

To avoid the obvious attack of setting all bits to one in the zFilter, and delivering the packet to all possible nodes in the network, we have implemented a very simple verification on the zFilter. We have limited the maximum number of bits set to one in a zFilter to a constant value, which is configurable from the user space; if there are more bits set to one than the set maximum value, the packet is dropped. The bit counting function calculates the number of ones in a single zFilter in the bit_counter module. This module takes 64 bits wide input and it returns the amount of ones on the given input. Only wires and logic elements are used to calculate the result and there are no registers inside, meaning that the block initiating the operation should take care of the needed synchronization.

The Ethertype of the packet is checked upon arrival. At the moment, we are using 0xacdc as the ethertype, identifying the zFilter-based packets. However, in a pure zFilter based network, an ethernet is not necessarily needed, thus this operation will be obsolete. The third packet checking operation is the verification of the TTL. This is used in the current implementation to avoid loops in the network. There are other solutions considered in the project for loop detection, but the TTL based solution was implemented as a good enough solution before the proper design has been completed.

The id_store module implements Dual-Port RAM functionality making it possible for two processes to access it simultaneously. This allows modifications to LIT:s without blocking the forwarding functionality. The id_store module is written so that it can be synthesized by using

either logic cells or BRAM (Block RAM). One of the ports is 64 bits wide with read only access and it is used exclusively to get IDs for zFiltering logic. There is one instance of the id_store module for each LIT and virtual LIT. This way the memory is distributed and each instance of the filtering logic has access to id_store at line rate. The other port of the id_store module is 32 bits wide with a read and write connection for user space access. This port is used by the management software [NetFPGA] to configure LIT:s on the interfaces. One new register block is reserved for this access.

One additional register block is added for module control and debug purposes. It is used to read information that is collected into the status registers during forwarding operations. Status registers contain constants, amount of links, maximum amount of LITs and virtual LITs per link and also the LIT length. In addition, information about the last forwarded packet is stored together with the result of the bit count operation, d, TTL, and incoming port information. This block is also used to set the maximum amount of ones allowed in a valid zFilter.

Management software

For configuration and testing purposes, we have developed a specialized management software. When the system is started, the management software is used to retrieve information from the card and, if needed, to configure new values on the card. The information that the software can handle includes the length of the LITs, the maximum d value describing the number of LITs used as well as both link and virtual link information from each of the interfaces.

Internally, the software works by creating chains of commands that it sends as a batch to the hardware, gets the result and processes the received information. The commands are parsed using a specific, special purpose, grammar. The parsing is done using the *byacc* and *flex* tools, and is therefore easily extendable.

For testing purposes, the software can be instructed to send customizable packets to the NetFPGA card, and to collect information about the forwarding decisions made. The software supports the following features:

- Selecting the outgoing interface
- Customizing the delay between transmitted packets
- Varying the sizes of packets
- Defining the Time-to-live (TTL) field in packet header
- Defining the *d* value in packet header
- Defining the zFilter in the packet header
- Defining the ethernet protocol field

2.2 Upper layer implementation

2.2.1 Rendezvous

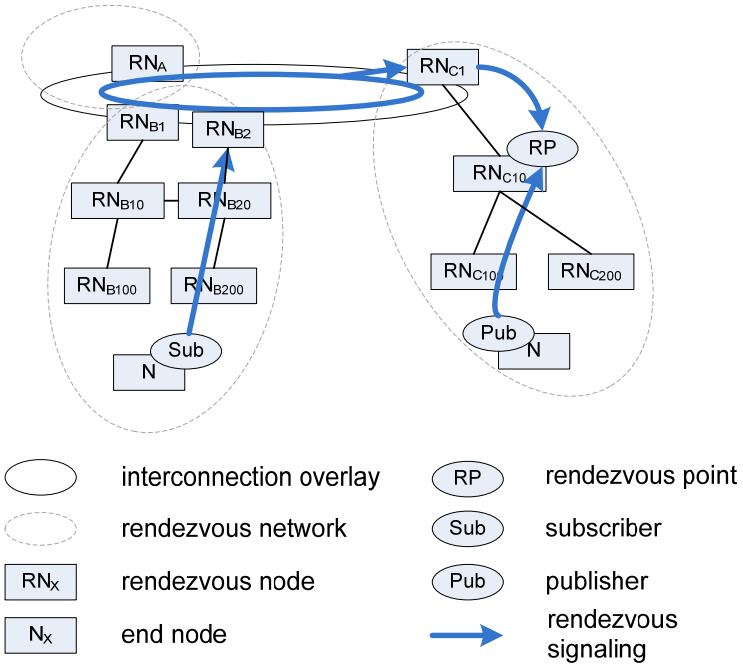

Figure 7 Rendezvous system

Since the writing of D3.2, the implementation of the PSIRP rendezvous concept, which is roughly illustrated in Figure 7, has advanced considerably both in architecture and prototyping. The latest version of the rendezvous architecture is described in D2.3, and has been evaluated, to some extent, in the evaluation report D4.2. The astute reader should notice that the figure7 excludes the local rendezvous helper part of the rendezvous concept, which is presumably located in every node. The figure also illustrates on a very high level how signals flows in the system in an example publish-subscribe use case.

Even though the concepts have evolved, one thing that has remained is the internal division of the rendezvous implementation. It still contains two basic entities; the local rendezvous client, also called local rendezvous helper, and the rendezvous node, which in D3.2 was called the rendezvous point. The term rendezvous point remains in the conceptual architecture as a logical rendezvous entity within a rendezvous node, handling rendezvous for a specific (Sid, Rid) pair. That is, one rendezvous node may contain zero or more rendezvous points.

In the following subsections we go through the current implementation status of both the local rendezvous helper and the rendezvous node.

Local rendezvous helper

The current Blackhawk prototype includes a helper application for rendezvous in small-scale local area networks. This helper, called laird (Local Area Intra-domain Rendezvous Daemon), is implemented in Python 2.6. It uses the Python version of the libpsirp API for node-internal publish/subscribe operations. Network communication is handled via the network I/O helper (netiod).

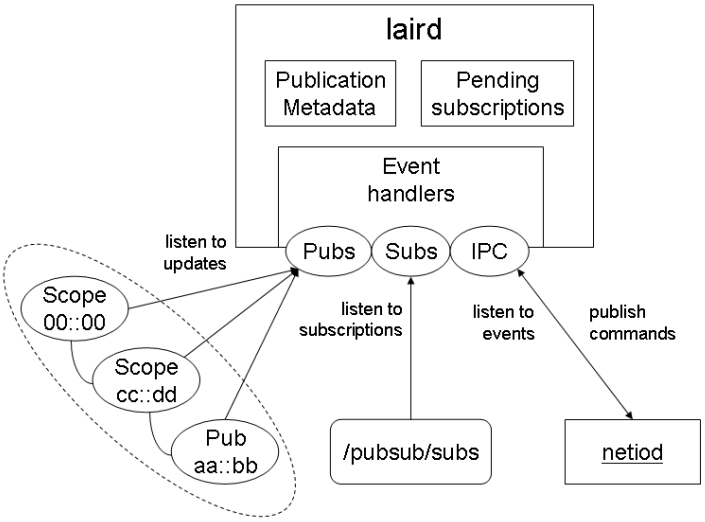

Figure 8 Local Area Intra-domain Rendezvous Daemon

An instance of the local rendezvous helper is installed in all end nodes. By default, this application listens to publish (by recursively listening to new scopes and publication appearing under the root scope) and subscribe (by listening to the '/pubsub/subs file) events in the node. If the SId of a publish or subscribe operation matches a set of predefined scopes, network pub/sub is triggered. In the case of a publish operation, this means sending metadata to the network (that is, to the local rendezvous node). A subscribe operation, on the other hand, implies sending out subscription messages: the first one for metadata, then another one for data. Additionally, laird keeps state for pending subscriptions.

At least one node in the local area network needs to act as a local rendezvous node (LRN), also running laird. The LRN listens to metadata publication and subscription messages coming from other nodes. Metadata publications are stored within laird (currently only internally, but in later versions they could be placed in the blackboard). When a subscription to metadata arrives, stored metadata is re-published as the response. Subscriptions to data, on the other hand, are dispatched to publishers using the FId found in the metadata.

The current version of laird does not support different versions of publications or subscriptions to single memory pages. It also lacks topology management functions, so data can be published using suboptimal delivery trees.

Rendezvous node

As already mentioned above, the rendezvous node implementation is as of today still a separate Ruby-1.8 based implementation. However, postponing the integration task did not halt the rendezvous node implementation, quite the opposite. After finishing the initial version of the intra-domain rendezvous node, the implementation effort extended the implementation to cover inter-domain rendezvous functionality, namely implementing the first version of the rendezvous network concept as described in D2.3.

The rendezvous network is a simple version of an inter-domain rendezvous system implementation, assuming each RN in a rendezvous network representing a different Autonomous System (AS). Due to scalability issues, a single rendezvous network cannot represent the entire AS topology. Hence, multiple rendezvous networks are needed, being inter-connected through a scalable interconnection overlay. In short, the current status of the rendezvous node implementation is that it supports basic features to establish a rendezvous network containing set of rendezvous nodes that compose a policy compliant rendezvous network topology and support basic rendezvous in it. From the architecture concept of rendezvous, the interconnection overlay that connects the rendezvous networks has not yet

started. However, the existing version inherently supports connecting two rendezvous networks via peering, in the same way as it support connecting to rendezvous node via peering.

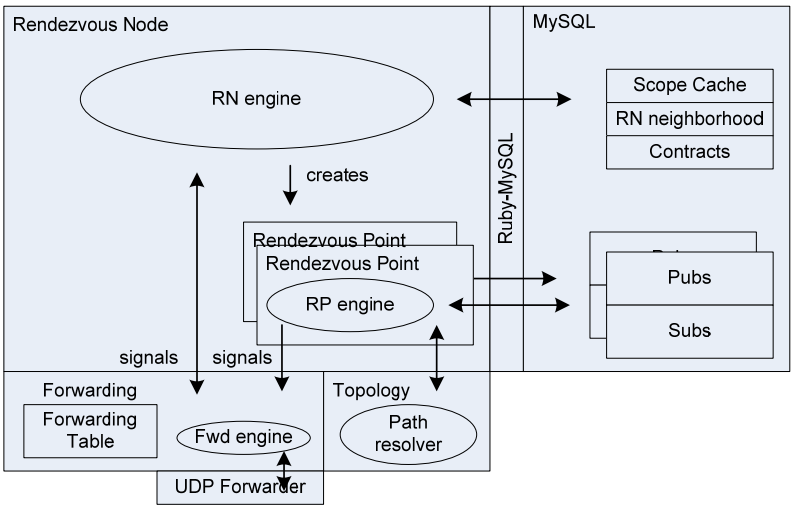

Figure 9 Rendezvous node implementation architecture

Figure 9 represents architecture of the rendezvous node implementation. As visualized in the figure, the rendezvous node implementation also includes stub versions of functions that cooperate with the rendezvous function, such as forwarding and topology. These were implemented to enable the testing of actual rendezvous node functionality. Another important fact relating to the implementation is that at this point the focus is in implementing the features of the rendezvous node for the rendezvous network and not in making rendezvous node perform optimally. This is due to ongoing work on the conceptual level. For example MySQL, even though it is a powerful database implementation, has been used for all rendezvous node and rendezvous point data structures, even though at least some of them would perform better, if implemented differently. Another example of a non-optimized module is the rendezvous signalling model, where the actual format is still under development. Therefore, the implementation uses a field tag based character string format for rendezvous signals.

Topology stub in the rendezvous node

The rendezvous node implementation contains a minimal stub version of topology function that is needed to enable inter-domain rendezvous. The functionality provided is:

- Upgraph probing, which all nodes use to find their valley-free policy compliant inter-RN upgraphs.
- Subscriber – Publisher inter-AS path generation, where upgraphs from the subscriber and publisher are evaluated and joined by topology function and as a result an initial inter-AS path gets generated between them.

Path generation is done based on two prior policies; promote peer links over transit links (minimal cost) and domain path length (minimal domain hop-count). The working assumption has been that the upgraph of the topology function is provided to the rendezvous point, which iacts as a man in the middle and provides upgraphs to the topology function. The topology function is implemented locally as part of each rendezvous node. The reader should notice that this version of the stub topology function draws analogy from the rendezvous node topology to the actual inter-domain network topology and each RN is assumed to represent a different AS. In real-life case, the rendezvous topology cannot be assumed to be the same as the network topology. Therefore, the topology function should be implemented also in those nodes that do not host a rendezvous node.

Forwarding stub in the rendezvous node

In the same manner to the topology function, the rendezvous node implementation includes a stub version for the forwarding function to enable inter-domain rendezvous. The forwarding has been implemented using a generic layer of forwarding between the rendezvous node. The actual forwarding engine is using UDP. The idea of the generic forwarding layer has been to hide all forwarding engine parameters from the rendezvous node in order to provide transparency. Due to the usage of UDP, each rendezvous node has a predefined forwarding table, where rendezvous node IDs are mapped to UDP specific transport information, namely hostname and port number. Even though the forwarding table is initialized in to each rendezvous node from a configuration file, the forwarding table is dynamic in the sense that messages coming from an unknown hostname and port are added to the table.

2.2.2 Topology

One of the essential pieces of the PSIRP architecture is the topology management functionality. We differentiate between intra-domain and inter-domain topology mechanisms as two separated entities with strong correlation between then. Intra-domain topology management is responsible for local network topology generation, i.e., within one administrative domain. Its main functionality is based on discovering topology information, using it as an input for computing necessary network states and updating forwarding the information to involved nodes. Inter-domain topology formation function defines topology formation on the domain level, i.e., between administrative domains. In other words it has a role of configuring and maintaining inter-domain topology states which is further used for creating forwarding paths based on policy compliance.

At the present stage, we focus on the implementation of the intra-domain topology management, integrating into other related components such as the forwarding and rendezvous functions. Extensions towards the inter-domain topology formation will be planned as the architectural understanding on the related design choices develops.

The topology management implementation follows the same design cycle as the rest of the prototyping efforts. An early version of the topology design was developed together with the first prototype, demonstrating the basic concepts and helping to gain implementation experience on the platform.

Our current Python based implementation of topology management is compliant with the first PSIRP prototype and implements a simple inter-domain topology formation. The overall functionality is divided into two main modules: client and server which can simultaneously coexist on each node. The client module runs on each forwarding node and is mainly responsible for discovering local connectivity information, whereas the server module is collecting this local information and piecing it together to form a picture of the overall network topology within the domain of operation. The server module is also responsible for computing the optimal forwarding paths and publishing that information towards forwarding nodes.

The basic mode of operation of the topology management client application implementation is similar to any link state routing protocol link state advertisements (LSAs). The client sends periodic announcements of its existence and collects the information coming from neighbouring nodes. Having knowledge of all available neighbours of each node is prerequisite for building general topology information. The topology management server application is responsible for gathering environmental information from all nodes and deriving the inter-domain topology states.

In order to facilitate the exchange of required data, two scopes are defined: “Hello” scope for exchange of existence information coming from each particular node and “LSA” scopes for distributing and collecting information representing the set of neighboring nodes. In our implementation, every node is periodically publishing “Hello message” containing node’s ID in

predefined “Hello” scope. At the same time, all nodes are subscribed to the same scope, “listening” to their neighbor’s advertisement messages and generating the list of neighbors. Upon receiving the “Hello message”, i.e., matching the subscription and publication in the “Hello” scope, nodes update the neighborhood list and publish it in “LSA” scope. Any new LSA message is published every time the current topology changes, e.g., a new node becomes available, or the old one, already existing in the list disappears (i.e., does not publish Hello message within predefined period).

The topology management server application is responsible for collecting LSA messages of all nodes, based on which it can compute the shortest paths in the network. Thus, subscribing to the LSA scope, the server is receiving the neighbors’ lists of nodes publishing LSA advertisement in the same scope. The topology manager implementation also does simple forwarding tree formation using simple shortest hop-count spanning tree formation. For calculating the shortest paths, we currently simply apply Dijkstra’s algorithm, using the implementation from the igraph library [Igr2009]. This part will be heavily extended in future work as outlined below.

2.3 Applications

The main target of the project has been to develop a clean slate internetworking architecture based on the publish/subscribe paradigm together with an implementation. While applications are a crucial part of a fully working environment, but still not the core of the project, an Application Innovation Process was developed to inspire external parties to join the work by allowing them to use the implementation provided by the project. However, to be able to do testing during the development process, a few smaller applications are being developed for this purpose in the project. In the following, we describe the status of the application development.

2.3.1 Bittorrent

As one of the main motivations for the publish/subscribe architecture developed by PSIRP is the popularity of BitTorrent like content distribution applications, we are developing a PSIRP based application that draws upon the BitTorrent concepts of breaking down the content exchange to fixed size pieces and exchanging pieces among peers on a tit-for-tat basis, as well as upon the PSIRP concepts of network supported rendezvous and native multicast communications [MBitTorrentb]. The goals of this application are to serve both as a demonstrator for the PSIRP architecture and as a benchmark of PSIRP performance compared to the current Internet.

The main idea in PSIRP based BitTorrent is that each piece of the content will use a separate RId. Each peer that already has the piece will be able to act as a publisher, and each peer that needs the piece will be able to act as a subscriber. PSIRP will provide the rendezvous between publishers and subscribers and the multicast distribution of each piece from a publisher to the subscribers. Despite its similarity with BitTorrent, the PSIRP based application is fundamentally different than BitTorrent in aspects such as the co-ordination between the multiple publishers of each piece, so as to avoid dUpLIcate transmissions, the incentives for participation in the data exchange, which cannot be based on simple tit-for-tat, and the relationship between the RIds of the pieces, which could be based on algorithmic identifiers.

In order to better understand the possible options and tradeoffs, we have implemented a standard BitTorrent simulator over a general purpose simulation platform (OmNET++) [BitTorrenta]. This simulator serves both as a benchmark against which the PSIRP version will be compared, and as the basis for a multicast BitTorrent version that operates over an overlay multicast routing scheme (in particular, Scribe over Pastry, both provided by the OverSim extension to OmNET++). The multicast BitTorrent simulator, which is currently a work in progress, will allow us to determine the appropriate mechanisms and policies for a generic multicast based content distribution application. This knowledge will then be used to

implement the actual PSIRP based BitTorrent application over the PSIRP prototype. In parallel, the BitTorrent development team participates in the architectural design items that are particularly relevant to this application, such as algorithmic IDs, multicast routing and forwarding and rendezvous design.

2.3.2 Firefox plugin

This section presents a plugin written for the Firefox web-browser to access publications provided by the PSIRP library. The goal is to get and show publications transparently by typing a PSIRP ”address” in the web-browser's address bar. The plugin development involves two separate functionalities: the interception of PSIRP URLs and the subscription to the (SId, RId) pair, provided by the PSIRP URL. The first operation will require the extension of an existing component. The second will require the implementation of a new XPCOM component to bridge the Javascript in the browser with the C language in the PSIRP library.

The introduction of the PSIRP protocol requires the registration of the protocol in the core network component of the browser. Before the registration procedure takes place, the protocol must have a unique identifier assigned to identify it. In this case, the core network component will be extended to hold the new protocol and a new identifier will be used to identify the protocol. Each XPCOM component has two identifiers: the component ID, which is a 128-bit number, and the Contract ID (CID), which is a unique string identifying the component and either can be used to identify it.

The registration procedure is performed by the Component Registrar and takes the component's ID, name and CID as parameters during the registration. The core network component can be accessed by its text identifier

("@mozilla.org/network/protocol;1?name=psirp")

and the new protocol is passed as an argument during the component request.

The second step required is the implementation of the PSIRP protocol handler, which will be called whenever a PSIRP address is used. The core network component will redirect the event to the newChannel method. In this method, a parser for the PSIRP protocol was implemented to extract the (Sid, RId) pair related to the target publication, and an instance of the PSIRP XPCOM component, responsible for connecting the Javascript and C languages, is created. If the target publication is available, it will be returned as a file to the local machine. Then, the newChannel method will create an instance of the XPCOM LOCAL ("@mozilla.org/file/local;1") component to access the local files and an instance of the IO-SERVICE XPCOM ("@mozilla.org/network/io-service;1") component to create a new channel from the URI resulted from the Sid/RId from the URL to create the path to the local file. Otherwise, if the file is not found, the browser will display the default “HTTP400 - File Not Found Error” in the navigator.

The PSIRP XPCOM is a new component implemented to connect the PSIRP protocol hook, written in Javascript and the PSIRP library, written in C. Since XPCOM is platform independent, it can connect the Javascript, used in the web-browser with the C language, used in the library. The first step to create a XPCOM component is to define its interfaces using the IDL. The IDL has a set of attributes that can be used to pass values through different platforms and languages. After the abstract definition, the interface is compiled in the local language, in this case, the C language used to access the PSIRP library, resulting in a set of stubs. These stubs must be implemented in the local language, directly using the API provided by the PSIRP library. The second step is to create a unique ID and CID for the component. Linux provides the uuidgen tool to generate random 128-bit numbers that can be used as Ids. Finally, the stubs must be compiled in the target platform, resulting in a shared library containing the implementation of the component and the Firefox plugin component file (xpt).

The last part involves compacting all the files into a XPI (Zippi) file to be distributed to the users. The XPI file contains an install file with installation instructions for the browser and a components folder with the extension and the new component implementation.

The current version of Firefox plugin is working and compatible with the Blackhawk release

2.3.3 Application innovations process

D3.1 [D3.1] described the application innovation process as a method to engage with external partners in bringing the required application-level innovation to the PSIRP platform. This innovation process is obviously driven by the open source licensing model of the main PSIRP code. But even more, the timing of this process is driven by the availability of this code release. Given the current first release of the PSIRP node implementation to the public, we expect these engagement to increase in the near future. Current discussions with external partners include:

- MIT (see below)
- UK TSB PAL project (see below)
- NORS open source project: an open source wireless sensing platform to be integrated with a PSIRP core infrastructure

From these engagements, we would also like to feed back (a) bugs and usability reports into future code releases (see Section 2.4) and (b) determine necessary support for future engagements with other partners.

In the following, we will highlight two initiatives that have been discussed in the ramp-up to the code release.

MIT project on knowledge plane

In collaboration with Karen Sollins, efforts have been formulated to pursue practical issues of the Knowledge Plane proposition [Cla2003], e.g., information exchange between different providers, incentives for such exchange, and others. Of interest in this investigation is the impact of a novel information-centric routing fabric on the range of potential solutions.

In first discussions with the Principal Investigator Karen Sollins at MIT, it was agreed to base the investigation on the available PSIRP open source code release as an example for such information-centric routing platform. Hence, the investigations will shed light on this particular network services area, an application from the PSIRP perspective. Hence, we expect direct contributions from these efforts in the important area of helper functions (see [D2.3]).

TSB PAL project

BT in collaboration with Cambridge and Essex University in the UK have established a Technology Strategy Board (TSB) funded national research project named PAL (Personal and Assisted Healthcare). This project is based on the premises to explore requirements posed by future lifestyle and health management services on future communication systems. Novel services will be developed in this growing field of societal interest and their impact on the underlying infrastructure will be explored. In addition, the availability of alternative new communication infrastructures and their impact on realizing new services in this space are a target of the project.

Given the information-centric character of many of the considered scenarios, the explorative thrust of this project will focus on a platform akin to PSIRP and its applicability for the scenarios at hand. To this end the project will exploit the available PSIRP code release to base its investigations on this platform. The project will furthermore explore ethernet and optical topology management functions (see [D2.3]) and resource allocation mechanisms within a single domain - therefore extending the platform on the link layer level rather than only on the application level!

BT and Ericsson (UK) are the main industrial partners in this project with Dirk Trossen (PSIRP technical manager) being the PAL technical manager, too. This ensures synergy in the work and transfer of knowledge between these two efforts.

2.4 Open source release and user feedback

The first public release of the implementation was made available in the beginning of June 2009. During the first weeks, we have collected early information from users, related to their experience on the code. Four users have responded to the questionnaire that was made available in the Internet.

The main idea of the questionnaire is to get an understanding of the usage of the released code. We ask about the system where the implementation is installed on, about the installation process and related documentation, as well as about the quality of the code that the users are experiencing.

So far, all users installed the implementation on some virtualization environment, e.g., on a FreeBSD host installed on a Vmware environment.

The instructions were, in general, considered to be good and provided enough information. However, some modifications were suggested related to the operation system installation, and related to the software that was required before the installation could begin. During the project plenary in Ipswich (June 2009), a presentation was given about the internal structure of the prototype. Those users who had followed the presentation, understood the operation of the underlying system, but those who were not joining the meeting, suggested that more documentation would help understanding the system internals.

The installation procedure was considered to be smooth and no major problems were discovered. Some suggestions were provided to make the installation documentation better, and one bug for the compilation was found. Instructions to avoid the bug have been published. The networking part was tested only by one user data, with success. The rest of the users have not yet had the possibility to test it. One bug has also been found when the modules are loaded and afterwards unloaded.

Running and testing the implementation was simple with the provided sample scripts, but more information was requested in the form of documentation. In the feedback forms, there was valuable feedback on how to improve the documentation.

From the feedback forms it seems that the development team has been able to respond to questions promptly with enough information to solve the user problems.

We will continue to provide these questionaires to future users, in particular in relation to the application innovation process (see Secion 2.3.3).

2.5 Integration status

In the original implementation plan, one of the targets was to aim for an integrated basic prototype platform right from the start, i.e., integrate the first versions of LoLi and UpLI prototypes and this was assumed to happen during the first half of 2009. The integration task for the LoLi version1 and the intra-domain version of the Rendezvous Node was actually started already in November 2008, but despite good progress, it quickly became evident that the LoLi implementation architecture required changes. Due to these changes, it was decided to continue with the integration after the new version of LoLi, Blackhawk, would be available.

The decision to put the integration on hold naturally meant that we needed to re-schedule the rendezvous node implementation work. The logical way to proceed with the rendezvous node implementation was to keep the prototype as a separate code base and start implementing features for the inter-domain rendezvous prototype. This delay also provided enough time for the topology module implementation to catch up the development gap that had been building up during the earlier phases.

Given the first official code release of the LoLI (Blackhawk) at the end of June, the inter-domain rendezvous node as well as the topology module implementation can now be continued. It is expected that the work will be finished during 2H09. We also plan to have a consequent code release before the end of the year.

3 Compliance with architecture definitions

The following section addresses the compliance of the current implementation with the architecture definition, as outlined in D2.2 and D2.3. This presentation also addresses the evolving compliance of the different stages of the implementation, therefore outlining the design methodology of PSIRP D2.2.

3.1 Lower Layer implementation

In this section, we compare the lower as well as upper layer implementation to the architecture deliverables D2.2 and D2.3. Apart from discussing potential differences, we also highlight where the implementation work affected the architecture work, aligned with our design methodology outlined in D2.2.

3.1.1 End-host implementation

In this section, we compare the current kernel-based FreeBSD end-host implementation with the architecture, as defined in the D2.2 Conceptual Architecture [D2.2], and D2.3 Architecture Definition [D2.3].

Component wheel architecture

The Component wheel concept was introduced in Section 7.2 of D2.2 and revised in Section 3.4 of D2.3. The first, FUSE-based, FreeBSD prototype did not really implement the component wheel concept, as all the components were in practice integrated within the user level psirpd daemon. However, the experience gained during the first implementation round, and especially the early difficulties in the second, kernel-based, implementation round allowed us to refine the concept from D2.2, resulting in a much stronger description in D2.3 than what would have otherwise been possible. However, as the first really usable version of the component wheel became available only in late May 2009, it remains to be seen how the implementation experience from it will affect the later versions of the architecture.

The early APIs, as imagined in Figure 7.1 of D2.2, where clearly too abstract to really guide implementation work, while the later API definitions in D2.3 provided the necessary details (see below).

As of today, we are developing the actual interaction patterns between the components in the component wheel. Figure 3.7 in D2.3 provides some guidance, but as the actual node-internal event mechanism was completed after the publication of D2.3, it is clear by now that the actual interaction patterns will be different from those described in D2.3.

Identifiers

The types of identifiers were initially defined in Section 6.3 of D2.2 and revised in Section 4.1 of D2.3.

As of today, there is some early support for Application Identifiers in the Firefox plugin. However, as the plugin has not yet been used in practise, it is too early to evaluate the implementation implications to the architecture.

The present FreeBSD in-kernel prototype implements Rendezvous Identifiers and Scope Identifiers as (almost) equal, both placed in the same name space. It has become apparent that there will be many different types of Rendezvous Identifiers (as outlined in D2.2 and D2.3), including at least Scope Identifiers, Memory-Object Identifiers, Version identifiers, Page identifiers, and perhaps Packet identifiers. This was already partly reflected in the API descriptions in Section 3.3 of D2.3, due to frequent interactions between the implementation and architecture groups while finalizing D2.3.

At the Forwarding Identifier side, the project has, at least for now, focused on using Bloom Filters based zFilters as the Forwarding Identifiers. This works well for the small networks we

are able to build right now, and apparently will continue to work up to metropolitan sizes. However, the choice may have to be evaluated again, once more work on intra-domain forwarding gets done. As of now, algorithmic identifiers have not been implemented.

Service model

The initial service model was defined in 7.4 of D2.2. In D2.3, the service model was made much more concrete, up to the API level (see next section). Much of the text in D2.3 Section 3.3 was affected by the first FUSE-based FreeBSD prototype and the frequent interactions between the architecture and implementation group. In this section we evaluate the prototypes from D2.2 service model point of view.

In all prototypes, the only supported communication entities have been publisher and subscriber, as described in Section 7.4 of D2.2. Of the publisher functions, metadata, multicast, and scoping have been implemented. The publishers remain relatively anonymous, but the finer details of anonymity and accountability are to be studied during fall 2009. Caching is likely to be implemented in late 2009 or early 2010. As of now, it looks likely that there will be no real implementations for data correlation or anycast. Considering the subscriber functions, publisher authentication and data integrity are supported through PLA and hashes. It seems unlikely that there would be real implementations of subscription state removal or subscriber accountability, as they do not seem to bring considerable value from an application prototyping point of view.

As of today, the implementation work has not had any substantial affect on the service modell at this level. However, the effects at the API level are very clear.

APIs

Section 3.3 of D2.3 defines a number of APIs, including a Low-level API (3.3.1), Memory-object service (3.3.2), Channel mode (3.3.3), and Higher-level Service Models (3.3.4).

The basic user-level API (libpsirp) of the in-kernel FreeBSD prototype is quite close to the Memory-Object Service, as defined in Section 3.3.2 of D2.3. The minor differences are mainly due to the event service, which is still seeking its final form. Once the API has stabilised at the implementation side, we expect the experiences to be reflected in future architecture deliverables.

Additionally, the basic libpsirp API contains functions roughly corresponding to some functions of the low-level pub/sub API presented in Section 3.3.1 of D2.3. The publish and subscribe operations related to memory objects also trigger corresponding operations in the network. As indicated in D2.3, the low-level forwarding and rendezvous APIs are mainly conceptual and are not exposed to applications. In the current implementation the functionality they provide exists internally in the network I/O and local rendezvous helper applications, although not as explicit interfaces.

Neither the first FUSE-based nor the present in-kernel FreeBSD prototypes have provided any real implementation of the Channel API, as described in Section 3.3.3 of D2.3. However, at the time of writing there is a clear need for this, and a number of implementation options are currently on the table. In contrary to the channel mode, as there is no apparent need right now for any of the higher-level service models (Section 3.3.4 of D2.3), it remains to be seen if they are implemented during the project or not.

Actual components

Both D2.2 and D2.3 discuss a number of components, sometimes also called helper applications. In this section, we briefly evaluate the status of local rendezvous, forwarding, and network attachment. For the other defined components, namely transport, caching, and

security, there either are no implementations or the existing code is too rudimentary to form a bases for evaluation.

Local rendezvous

Section 8.1.3 of D2.2 defines the rendezvous component in general terms. Section 3.5.1 of D2.3 further deepens the architecture by making a distinction between node-local blackboard, link-local rendezvous, and larger network-based rendezvous implemented by the rendezvous nodes. While network-based rendezvous is discussed in length elsewhere in this document, we briefly evaluate here node-local and link-local rendezvous.

In the in-kernel FreeBSD blackboard system, the node-local rendezvous consists of four components. First, an in-kernel collection function makes all publication events available through a special system-internal interface /pubsub/pubs. Second, a special small user level daemon, scoped, accesses this interface and creates a separate memory-object publication for each locally-existing scope. That is, whenever any other component in the system publishes anything in any scope, the scoped daemon gets notified, and if the publication is a new one, either creates or updates another publication that describes the contents of the scope. The resulting recursive process always terminates when the published scope already exists in an enclosing scope, in which case the event does not change the enclosing scope.

Third, whenever an application attempts to subscribe, the kernel checks that the requested RId exists in the defined scope. That is, the kernel subscribes internally to the scope document, as created by scoped, and sees that the RId has indeed been published in the scope. If so (and if the scope permissions allow), the subscription succeeds immediately and the subscriber gets access to the data.

Fourth, if the RId does not appear (yet) in the specified scope, the user level library is able to subscribe to the scope and wait for the RId to appear in the scope. However, at this writing there is only an early implementation of this, and the system is likely to evolve.

Overall, the local rendezvous system is a very nice example of the publish/subscribe principles working within the system itself. For example, the scoped daemon works solely through the memory-object API in a stateless manner, relying on the publish/subscribe system (i.e. blackboard) for storing persistent information. This also makes the system robust; the scoped daemon can be killed and restarted at any time without causing data loss. As architectural feedback, this can be seen as a re-enforcement for the architectural choices.

The same principles have been applied to the link-local rendezvous, which is currently implemented by a separate helper application, that is, a user-level daemon. This link-local rendezvous helper was created because a concrete need for a rendezvous mechanism compatible with the rest of the lower-layer prototype was identified. At this time it is too early to evaluate its implications to the overall architecture, due the local blackboard IPC failing due to some bugs.

Node-local network I/O

At the forwarding side, the node-local forwarding is currently implemented as a separate user-level daemon. However, this functionality is planned to be moved into the kernel due to performance reasons. As a major drawback, the current user-space implementation requires two memory copies for each incoming packet; a properly implemented in-kernel implementation should be able to receive packets without any copying at all. From an architectural point of view, the node-local rendezvous does not seem to have any larger implications.

Network attachment

The implemented network attachment functionality currently exists as a separate piece of code, a userspace daemon written in Python, which has not been properly integrated with the rest of the end-node implementation. Such integration is planned to take place during 2009.

From an architectural point of view, the existing network attachment implementation is close to what was described in Section 8.7 of D2.2. It is also aligned with the plan in Section 4.2.3 of D3.2. The current implementation uses generic, algorithmically generated message identifiers. It provides a simple framework for attachment point discovery, node-to-node communication bootstrapping, and performing attachment operations such as authorization negotiations and node configuration.

The upcoming integration work will make the implementation compatible with the rest of the system, which includes changing the generic identifiers to proper SIds, RIds, and FIds. How to achieve this is partially outlined in the updated architecture description found in Section 4.7 of D2.3. The implementation also has to interact with other components, such as the rendezvous system, enabling automatic node configuration in different network scenarios.

Node-internal architecture

Section 4.2 of D2.3 defines the blackboard approach to the component wheel. The current in-kernel FreeBSD implementation follows closely the approach at the general level. However, as the network I/O was for the first time properly implemented as a component only in early June 2009, and as the implementation is planned to be moved to the kernel as discussed above, it is too early to evaluate whether the considerations on input, output, and segmenting scopes, as explained in Sections 4.2.1 and 4.2.2 of D2.3., are really practical or not. Early experience seems to suggest that while the overall approach may also apply here, refinements are most probably needed.

3.1.2 Forwarding

NetFPGA forwarding node implementation

The NetFPGA implementation development has been quite straightforward during the project. This is mainly because the implementation contains only the forwarding part of the prototype, and the design has been relatively stable. The initial implementation was using Link IDs only for making forwarding decisions, and during the next phase, new features were added. These new features were the new LIT tables, Time to Live field for loop detection, bit counter for dropping packets that have too full zFilters, and management software for configuring the system and retrieving statistics.

Forwarding in the implementation for FreeBSD

Blackhawk's network I/O daemon implements the basic zFilter-based forwarding mechanism. Initially, "broadcast" zFilters were utilized, but the current version also provides the possibility of static pre-configuration of interfaces and their link-Ids. The forwarding component was inherited from the PSIRP daemon presented in D3.2.

Forwarding design team

The Forwarding Design Team has been developing inter-domain forwarding solution for the PSIRP project. The initial starting point was to use the zFilter based forwarding as the intra-domain forwarding solution. While the very basic zFilter solution does not scale into global networks, other mechanisms were introduced to solve the problem. Two alternative solutions have been proposed to enhance the inter-domain forwarding, one based on Merkle trees and the other a direct enhancement of the zFilter based forwarding. However, no implementation has yet started either of the solutions.

3.2 The rendezvous system

The decision to postpone the integration task between the two prototypes meant that the rendezvous node implementation could proceed as a standalone prototype directly to the inter-domain implementation phase. At the end of June 2009, the inter-domain rendezvous implementation has reached a state where it supports elementary rendezvous network concept for a rendezvous node. This support includes new features, such as rendezvous

neighbour discovery mechanism for rendezvous network topology establishment, policy compliant scope advertisements and rendezvous point search function within topology. It also upgrades some already existing features, such as to the rendezvous signal model, which now supports rendezvous signalling within policy compliant rendezvous network topology.

In general, it can be concluded that in 10 months of rendezvous implementation the implementation of the rendezvous node features has proceeded according the schedule - even bit ahead of it. This should provide us some breathing space with the schedule, even though the integration is only continuing now.

The current prototype follows the inter-domain rendezvous concepts for a single rendezvous network, which are described in D2.3. Thus, the prototype demonstrates basic inter-domain rendezvous functionality in single rendezvous network environment. As usual in prototyping there are some parts in the rendezvous implementation that are still looking for their final form, such as the signal model and bootstrap mechanisms, but the interconnection overlay concept which connects separate rendezvous networks, is the only unrealized major architectural concept. The schedule for the interconnection overlay implementation is left open for now, because all efforts will be put in to the integration. In general, we can conclude that the implementation of the rendezvous functions, although still on the parallel path with the LoLi implementation, has proceeded according to the ealier plans and the concepts provided by the architecture rendezvous team.

Additional effort has been placed on work that was in focus during the architecture work, such as the rendezvous signal format and the signal modeling. While we expect the integration still to inject changes and additions to the signalling model, we have identified and evaluated integral paramaters and those are included into the model. Therefore we believe that the current signalling model is "close enough" to be integrated.

3.3 The topolygy module

Our first topology management prototype was focused on simple intra-domain topology discovery and formation splitting the functionality into client and server module. In order to provide a more flexible and extensible topology management architecture, we aim to extend the currently implemented scenario into a more modular and extensible design. Therefore, major changes should be done in data structure and its management, also taking into account the requirement that topology management should provide more details about network conditions which together with user demands can lead to better routing decisions based on application requirements.

As a first step towards the revised topology management architecture, we envision extending the simple “Hello message” structure into a generic table containing information about outgoing node’s links as row entries. Each link is represented by its ID and Type, but it can additionally contain a set of link related properties, e.g., Throughput, Delay and ID(s) of the node(s) attached to the other side of the link. Information about relevant link properties can be provided by low level helper functions, which collect physical data of the link. The table needs to have an easily extensible format in order to support future requirements. Related node IDs and link IDs could be distributed, for example, in the network bootstrapping process, e.g., Network Attachment phase. Updating the table is done asynchronously according to the network changes, e.g., link failure, delay increase. Managing the policies related to publishing of updated information is under control of separate network management entity which defines the network conditions and thresholds for notification. Since different applications are interested in different topology parameters and sensitive to their different value changes setting the limitations in announcing network state altering avoids unnecessary traffic. Accordingly, in the next implementation version LSA messages will keep the same basic structure, containing additional link information except carrying just the list of available neighboring nodes.

4 Conclusion and next steps

The implementation work in close cooperation with the architecture work has proven to work and provide good results. The development of the prototype has been rapid and multiple versions of different parts have been done, quickly changing the implementation direction, when either the implementation or the architecture work has indicated the need for change.

The LoLI implementation work has provided now the end-host implementation and basic forwarding means to be used by the UpLI as well as applications. The work with the LoLI continues with mainly making the needed changes found during the integration work with UpLI and application development, both inside the project and by external partners.

The public releases of the FreeBSD and NetFPGA implementations as well as the PLA implementation, together with the active work on publishing results on different conferences, is expected to inspire also new partners to test and implement on top of the provided code. Based on the received feedback and discussions with various other projects, it is highly assumed that the implementation will be taken for further development also elsewhere.

The rendezvous implementation work will continue primarily with the LoLI integration. Depending on the progress in the integration task, further developing of the inter-domain rendezvous system, based on the design created in the architecture work package, may take place after the integration is finished.

The next version of the topology management implementation will go towards a more modular system, with better extensibility. The near future work includes the inter-domain topology management work, as well as issues that emerge from the integration work.

As mentioned, the integration work will be the major implementation effort during the next six months. When the rendezvous and topology are put into a single demonstrator, it is expected that multiple issues will be found that will affect mostly the implementations, but also the architecture work, at least on some details. As a result of the integration work, a new official code release, containing rendezvous and topology implementations as the new elements, is planned to be available before the end of the year 2009.

5 References

[D2.2] Ain et al. *Conceptual Architecture of PSIRP Including Subcomponent Descriptions*, FP7-INFOSO-ICT-216173-PSIRP-D2.2, August 2008

[D2.3] Ain et al. *Architecture Definition, Component Descriptions and Requirements*, FP7-INFSO-ICT-216173-PSIRP-D2.3, February 2009

[D3.1] Jokela et al. *Prototype Platform and Applications Plan and Definition*, FP7-INFSO-ICT-216173-PAIRP-D3.1, April 2008

[D3.2] Jokela et al. *Implementation Plan based on Conceptual Architecture*, FP7-INFSO-ICT-216173-PSIRP-D3.2, September 2008

[D4.2] Riihijärvi et al. *First report on quantitative and qualitative architecture validation*, FP7-INFSO-ICT-216173-PSIRP-D4.2, April 2009

[Cla2003] D. D. Clark, C. Partridge, J. C. Ramming, J. T. Wroclawski. *80A Knowledge Plane for the Internet*. Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications, 2003.

[Igr2009] The igraph library, available at: http://igraph.sourceforge.net

[BitTorrenta] K. Katsaros, V. Kemerlis, C. Stais, and G. Xylomenos. *A BitTorrent module for the OMNeT++ simulator*. In MASCOTS, 2009.

[NetFPGA] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. *NetFPGA – an open platform for gigabit-rate network switching and routing*. In MSE ’07, 2007.]

[BitTorrentb] G. Xylomenos, K. Katsaros, and V. P. Kemerlis. *Peer assisted content distribution over router assisted overlay multicast*. 1st Euro-NF workshop on Future Internet Architecture - New Trends in Service and Networking Architectures, Paris, France, Nov. 2008.