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

DELIVERABLE D3.2

Implementation Plan based on Conceptual Architecture

Title of Contract	Publish-Subscribe Internet Routing Paradigm
Acronym	PSIRP
Contract Number	FP7-INFSO-ICT 216173
Start date of the project	1.1.2008
Duration	30 months, until 30.6.2010
Document Title	Implementation Plan based on Conceptual Architecture
Date of preparation	30.9.2008
Authors	Petri Jokela (LMF), Janne Tuononen (NSN), Teemu Rinta-aho (LMF), Jukka Ylitalo (LMF), Dirk Trossen (BT), Dmitrij Lagutin (HIIT), Janne Riihijärvi (RWTH), George Xylomenos (AUEB-RC), Jimmy Kjällman (LMF), András Zahemszky (LMF), András Császár (ETH), Jari Keinänen (LMF),
Responsible of the deliverable	Petri Jokela Phone: +358442992413 Email: petri.jokela@ericsson.com
Reviewed by	Dirk Trossen, Janne Riihijärvi, Chris Reason, George Xylomenos
Target Dissemination Level	Public
Status of the Document	Completed
Version	1.0
Document location	http:// www.psirp.org/publications/
Project web site	http://www.psirp.org/

This document has been produced in the context of the PSIRP Project. The PSIRP Project is part of the European Community's Seventh Framework Program for research and is as such funded by the European Commission. All information in this document is provided "as is" and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability. For the avoidance of all doubts, the European Commission has no liability in respect of this document, which is merely representing the authors view.

Table of Contents

1 Introduction 3
2 Testing Implementations 3
2.1 Existing test networks/plans for new ones 3
2.2 Emulation/Simulation 4
3 Functional Overview 5
3.1 Applications 8
3.1.1 Current applications and application innovation process 8
3.1.2 Firefox Plugin 8
3.1.3 Multicast Bit Torrent 8
3.1.4 Socket's API Emulator 9
3.2 Rendezvous 10
3.2.1 Global Rendezvous Framework 10
3.2.2 Scope Rendezvous Subsystem 11
3.2.3 Host Local Rendezvous Subsystem 11
3.2.4 Rendezvous Identifier 11
3.2.5 Global Rendezvous Scope and Framework 11
3.3 Topology 11
3.4 Forwarding 12
3.5 Packet Level Authentication (PLA) 12
3.5.1 Cryptographic solutions and hardware acceleration 13
3.5.2 PLA Protocol 13
3.6 Network Attachment 14
3.6.1 Operations 15
3.6.2 Participants 15
3.6.3 Relation to other functions 15
3.7 Interfaces 15
4 Work Plan 16
4.1 Implementation Overview 16
4.1.1 Architecture 16
4.1.2 Network Nodes 17
4.1.3 Signaling Model 17
4.2 Helper Functions 20
4.2.1 Rendezvous 20
4.2.2 Topology 24
4.2.3 Network Attachment 24
4.3 PSIRP Daemon and I/O 26
4.3.1 PSIRP I/O module 26
4.3.2 PSIRP Daemon 26
4.3.3 PSIRP Daemon and PSIRP I/O Interaction 28
4.3.4 Network I/O 32
4.3.5 NetFPGA 32
4.4 PSIRP API and Module Interfaces 32
4.4.1 Libpsirp 33
4.4.2 API 33
4.4.3 Advanced API 34
4.4.4 Module interfaces 34
5 Abbreviations 35
6 References 36

1 Introduction

In PSIRP, implementation and architecture are intertwined in an approach of continuous development, establishing a methodology that combines top-down architecture with bottom-up implementation work. This leads to a macro-cycle of development, defining distinct *phases* (or rounds) for the architecture as well as for the implementation. Given this methodology of working, it must be understood that it is very difficult to make detailed implementation plans that can span all these phases. Instead, the intention of this document is to provide a more detailed description for the work in the first round with less detail about the work that will be performed towards the end of the project. The first iteration/round is concrete in the sense that it is based on our current understanding of the architecture as captured in D2.2 [1]. Refinements of the plan throughout the project will follow the increased understanding gained by the partners during the architecture and implementation work. In addition, it is expected that the evaluation work, on the simulative as well as the experimental level, will influence and therefore refine the plans for the next implementation rounds beyond the first one.

With that in mind, this document starts by presenting the implementation system and provides guidelines for evaluating and testing the system. After that, the implementation is presented, starting from the current understanding of the architecture. The required functionality is described and the system operation, as it is seen now, is presented. For this purpose, the document specifies the detailed implementation plan for the so called *Lower layer implementation* (LoLI) and *Upper layer implementation* (UpLI). These layers implement the operations required for rendezvous, topology management, and packet forwarding, thus covering the functions required to support internetworking in publish/subscribe networks. In addition, this document specifies the required interfaces, both the *Application Programming Interface* (API) that provides the functions necessary for application development, and the internal interfaces needed for inter-process communication within and between the layers. For this reason, the document also describes the resulting node architecture in which the lower and upper layer implementations are combined.

2 Testing Implementations

In this section we discuss the plans for testing the prototype implementation during development activities. More detailed evaluation also dealing with quantifying the scalability of the implementation and the overall PSIRP architecture is carried out together with WP4. Initial plans of these activities are documented in deliverable D4.1 [22], and will be further refined as the next cycles of prototyping activities are planned in detail.

2.1 Existing test networks/plans for new ones

Testing the PSIRP prototype in real test networks in crucial for a component-level testing of the implementation concepts. Such testing can take place in dedicated test networks within a PSIRP partner or within larger test networks. The former is likely to happen for a limited test of the feature interactions of the prototype. Stress testing entire components, such as rendezvous or forwarding, but also testing PSIRP under application load, is likely to take place in larger test networks.

One of these larger test networks is Planetlab [2], a contributory research facility that originated as an effort to create a large-scale facility that allows for testing new Internet-based technologies and services. Some partners in PSIRP are partners of the Planetlab consortium, enabling access to this platform. Running so-called slices (virtualized test networks) in this environment will enable testing certain aspects of the PSIRP prototype at a scale that is impossible to achieve within the limited partner test networks.

The EU FP7 project Onelab2 [3] maintains the European partition of Planetlab, called Planetlab Europe. Within the current structure of Onelab2, the work package 7 on data-centric networking focuses on extending Planetlab towards the specific requirements of projects like PSIRP (more concretely, the work items in this work package foresee extensions towards publish/subscribe as well as content delivery networks). BT is leading this work package in Onelab2. PSIRP has been identified as the main customer project for this task in Onelab2. This opens the possibility to shape new test networks, optimized to the proposed designs that PSIRP will develop throughout this project. It is therefore expected that platform-related requirements from the PSIRP architecture work will find entry in the work of Onelab2. Early prototypes of the Planetlab extensions, developed in Onelab2, can be exploited to ease testing of the PSIRP components. The special relation of PSIRP to Onelab2 and the current engagement of some partners in Planetlab in general, provide the possibility to perform prototype tests in relatively large and global test networks.

In addition to Planetlab, the *National Research Networks* (NRENs) provide the ability to test PSIRP prototypes, e.g., directly over dark fibre, where this is available. Relations exist at several partner sites to connect to the local NRENs. This, however, requires the availability of the PSIRP prototype over optical fibre technologies.

2.2 Emulation/Simulation

Network emulation is a work item where the Implementation and Evaluation work packages meet as it entails interconnecting a packet-level network simulator with the prototype. The planned emulation architecture is illustrated in Figure 1. Besides building an actual test bed with a few physical nodes, we plan to connect the prototype nodes to a network emulator that is capable of receiving packets from the physical test bed, thus simulating packet transmissions towards the simulated network, and sending out packets through a physical interface from the simulator to the real test bed. There are two reasons why this approach is beneficial. First, it allows large-scale testing of the actual implementation by investigating whether the implementation code works correctly even when there are multiple rendezvous points and numerous forwarding nodes in the network domain. Secondly, it allows the large-scale testing of the proposed architecture and protocols without requiring a separate implementation, i.e., it allows re-use of the implementation code for validation purposes.

We plan to use the new ns-3 [21] network simulator's emulation functionality. Although emulation support was not yet included into the first release that came out in June 2008, according to the current tentative roadmap, it will be the part of the third stable release due in November 2008. At the time of the writing, the development code for network emulation was available in the source repository of ns-3. Early experiments with the development version of the emulator using the included example scripts have shown that it is possible to send real packets from the ns-3 simulator through a physical attached interface to the real network. We have to note that if the planned network emulator is not available in the stable revision of ns-3, it is possible that we will not able to use it and testing plans will have to be updated accordingly.

For the actual code in ns-3 there are two options:

1. Use the existing prototype code and port as much as possible of it to ns-3. This would allow larger scale validation of the implementation code itself.
2. Define a protocol interface and implement a simulator specific simplified model of the architecture in ns-3. In this scenario, the real prototype will communicate with the simplified simulator using a well-defined interface allowing a larger topology to be simulated. In other words, the real test bed would behave as if it was part of a big network. This could also reveal interesting characteristics of the inter-domain network behavior.

Network emulation could also be important when the prototype code is still under heavy development as it allows quick testing of some features of the prototype even when other parts of the prototype are not ready. For this purpose, simplified code can be used in the simulator, which is usually quicker to implement than the actual prototype.

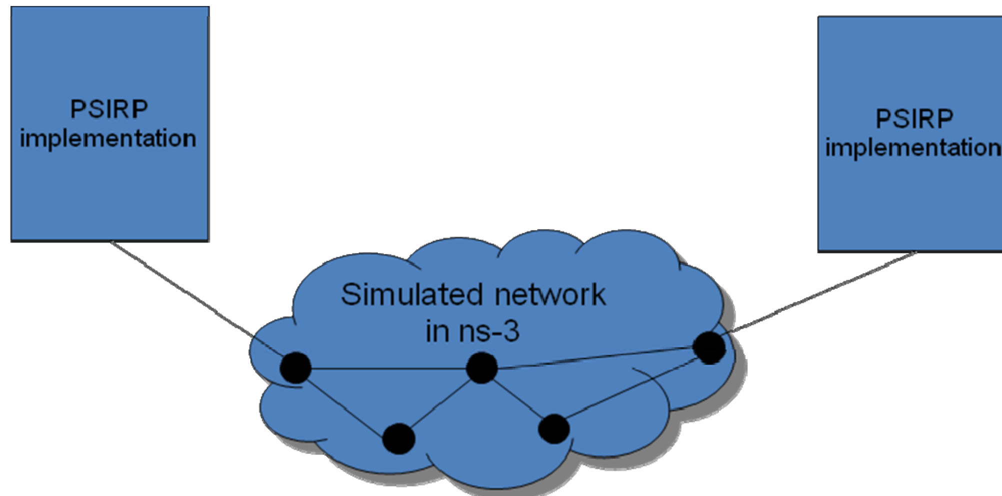

Figure 1 - The ns-3 emulation architecture

3 Functional Overview

This section provides an overview of the current network architecture and its components, based on the latest conceptual architectural deliverable [1]. At a very high level, the PSIRP architecture consists of three parts: the component wheel, which defines how the components are organized within a singe node, the networking architecture, which defines how a collection of nodes co-operate in order to create a publish/subscribe network, and the service model, which defines the interfaces provided to publishers, subscribers and network services.

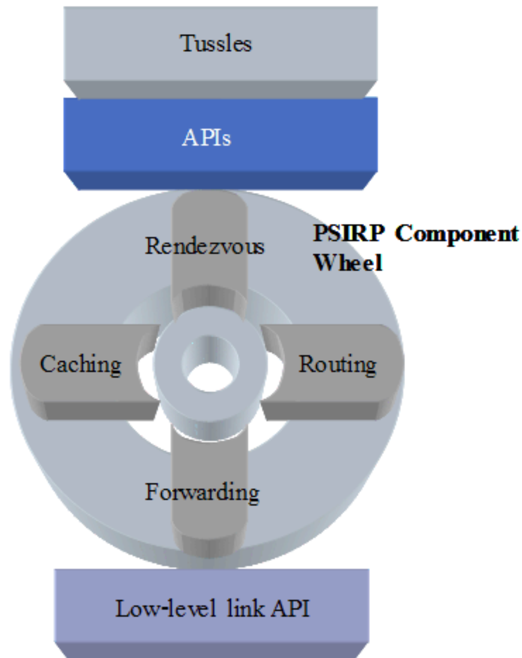

Figure 2 - The PSIRP component wheel

The component wheel, shown in Figure 2, replaces a traditional layered network protocol stack: it is a layerless collection of components implementing the various PSIRP protocols; these components also communicate between themselves in publish/subscribe style. The rendezvous component resolves the rendezvous identifiers provided by publishers and subscribers of data to forwarding identifiers used for data delivery within a publication's scope. This scope may reflect either a topological usage, i.e., a publication is available to link-local, intra-domain or inter-domain subscribers, or a semantic usage, where a publication is available to friends, family or colleagues. The routing component, also referred as *topology component*, establishes at (considerably) lower than wire speeds the intra-domain or inter-domain paths that will be needed for publication delivery, while the forwarding component employs these paths to deliver published data at wire speeds. The caching component stores copies of publications either in the local system or in any system on the communication graph formed by the network so as to speed up their delivery to future subscribers. The network attachment component is responsible for discovering points of attachment to the network and configuring an attaching node to make communication possible.

The nodes implementing the PSIRP component wheel co-operate to support the network architecture as outlined in Figure 3. In this architecture publishers and subscribers exchange data packets associated with metadata; metadata may include scoping information and other information useful for either receivers or intermediate network elements. Each piece of data is originally associated with one or more application identifiers and one or more scopes. Application identifiers are only used by applications for their own purposes and are transparent to the PSIRP network. Publication scopes are defined either via metadata attached to the data or via explicit scope identifiers; they determine the elements of the rendezvous system that should act on the data they accompany. The applications resolve the application identifiers to rendezvous identifiers that are supplied to the network along with the scope information. The network in turn maps these to sets of forwarding identifiers that are used to eventually deliver published data to subscribers; these forwarding identifiers essentially denote the per publication or shared multicast trees that a packet should follow through the network.

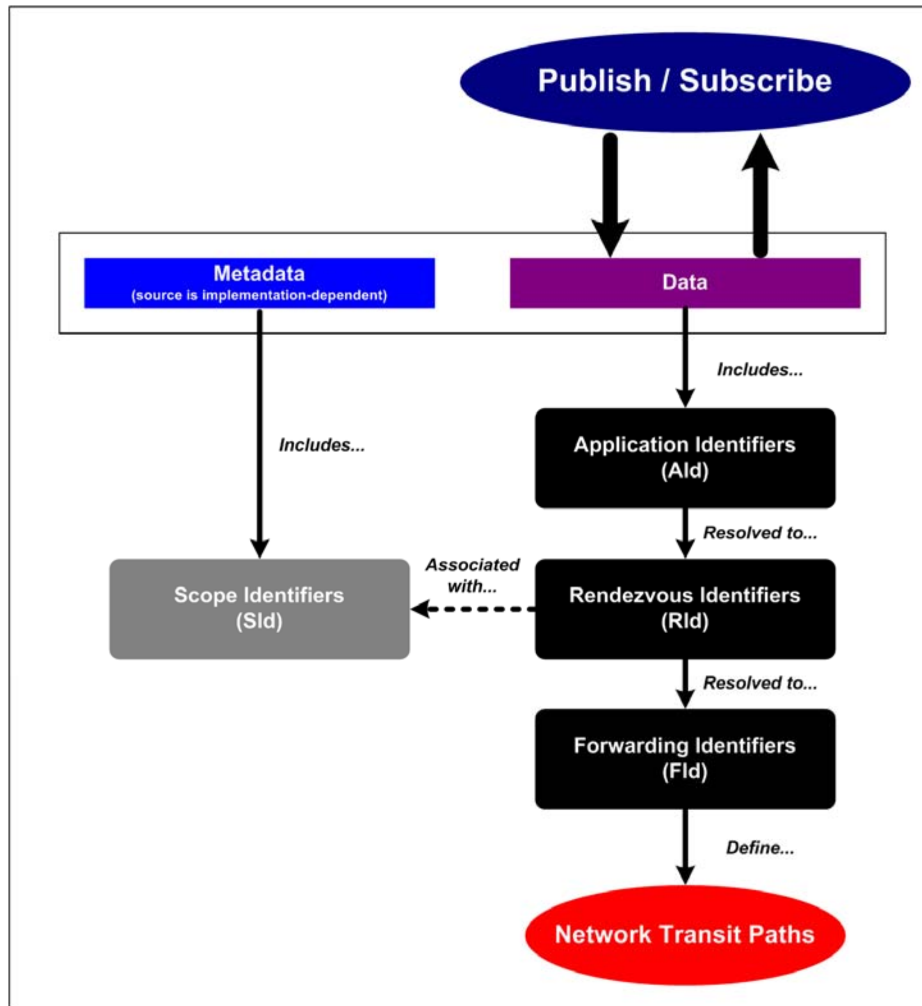

Figure 3 - The PSIRP network architecture

Finally, the PSIRP service model determines the interface between the network and publishers (for announcing and sending data), subscribers (for expressing interest to and receiving data) and network services (for management purposes). From the publisher (or sender) viewpoint, publications are associated with a rendezvous identifier and, optionally, some metadata. Features supported by the publisher service model may include on-demand publisher anonymity, transparent multicast distribution, indication that multiple publications are correlated, support for data caching directives, publications anycasted to a set of subscribers, limited publication scoping and publisher accountability (or authentication) by the system. From the subscriber (or receiver) viewpoint, publications are requested via rendezvous identifiers, also possibly associated with metadata. Features supported by the subscriber service model may include implicitly or explicitly limited subscription lifetimes, assurance of publisher authentication, protection of data integrity and subscriber accountability (or authentication). From the network service (management) viewpoint, management and measurement tools must be supported. Features supported by the network service model may include the provision of directory services and support for policies.

3.1 Applications

As outlined in D3.1 [4], PSIRP does not focus on the implementation of dedicated applications for PSIRP. The following section outlines the plan for enabling a set of applications through external partners and also the implementation of a small set of applications for demonstration purposes.

3.1.1 Current applications and application innovation process

D3.1 outlined the so-called application innovation process as the main driver for application development in PSIRP. Within that process, we engage with external partners with an interest in the wider publish/subscribe space to enable development or porting of applications to our platform. First contacts have been made in this regard although a complete list of applications has not yet been selected. However, efforts in the areas of Delay Tolerant Networks (DTN), sensor applications and event processing are expected to emerge through our collaborations.

3.1.2 Firefox Plugin

Web Browsing is a classic example of a publish/subscribe based data exchange: a web site is a, possibly evolving, complex publication consisting of web pages and their embedded objects, to which clients subscribe whenever they wish to read them. In the current Internet architecture, a client needs to periodically poll the web server to see if there are any changes to its content, since the rendezvous between the client/subscriber and server/publisher is instantaneous. The emergence of schemes like RSS that inform a client of changes to a server indicates that many Web sites would benefit from a more permanent association with their clients, such as the one provided by a publish/subscribe network.

In order to demonstrate possible user-lever applications of PSIRP, we are developing a PSIRP protocol handler extension for the Firefox web browser. The extension adds support for a "psirp:" URL scheme into the browser, allowing for complete web sites to be treated as PSIRP publications. The PSIRP URL scheme is of the following format: "psirp://[psirp_id]".

Once the protocol extension is installed into the browser, PSIRP URLs can be used in the same manner as "http:" URLs; they can, for example, be typed into the address bar of the browser and they can be given as the "src" attribute for image tags. When the extension encounters a "psirp://[psirp_id]" URL, it subscribes to the publication with "[psirp_id]" from the underlying publish/subscribe network and delivers the resulting data to the Firefox web browser. The data can be, for example, regular web pages or media files.

3.1.3 Multicast Bit Torrent

From its inception the PSIRP project has strongly argued that the Internet has evolved from a means of connecting endpoints to a means of connecting information with its users. Therefore, implementing a massive content distribution application to test the abilities and limitations of our design and implementation makes sense. We have thus decided to develop Multicast BitTorrent, a content distribution application that combines the best ideas of BitTorrent with the capabilities provided by a publish/subscribe network, such as native support for rendezvous and multicast delivery. In addition to providing the project with a very popular demonstration application, Multicast BitTorrent will help guide the design and implementation work, as it will provide us with an application that has a direct counterpart on the existing Internet.

In order to understand the opportunities offered by PSIRP for content distribution, we have analyzed the design and implementation of BitTorrent, with the aim of determining which features should be reused in PSIRP and which should be replaced. The core concept of BitTorrent is that the content to be distributed is split in individually verifiable pieces that can be independently exchanged between the participating users, or peers; as a result, each peer that has downloaded some pieces may then exchange them with other peers, thus spreading the network load around the network, instead of concentrating it close to its original distributor.

In addition, peers can come and go, something very important in large data exchanges, as each data piece can be separately exchanged and verified. The exchange proceeds on a tit for tat basis between the peers, therefore each peer has an incentive to behave well: if it wants to download new pieces, it needs to upload pieces to other peers. On the other hand, BitTorrent suffers from a costly (in terms of bandwidth and time) peer selection process, since good peers can only be distinguished after communication with them for some time.

At first glance, it may seem that BitTorrent is basically a crude substitute for multicast in the sense that it allows a content provider to reach large numbers of users without simultaneously unicasting the content to each one, an approach limited by the bandwidth available close to the content provider. However, the idea of breaking the data exchange into a set of piece exchanges has multiple uses: in addition to allowing each peer to start uploading pieces as long as it has downloading them, thus feeding the system, it also turns each peer into a cache for the pieces that it has exchanged, thus allowing the data exchange to become very robust even if it is spread over wide time intervals.

We have decided to retain the concept of exchanging data pieces in Multicast BitTorrent, also taking advantage of the PSIRP ability to provide a rendezvous facility between senders and receivers, as well as the PSIRP support for multicast delivery. In Multicast BitTorrent each piece becomes a separate publication, so that peers that do not have this piece may subscribe to it and peers that already have the piece may publish it. It is an item for study to determine whether the application (peers) or the publish/subscribe network should handle issues such as preventing publisher conflicts, i.e. not allowing multiple publishers to send the same piece at the same time, and delaying publishers from serving subscription requests until sufficient subscribers exist, to allow resource savings.

Another area where Multicast BitTorrent differs from standard BitTorrent is in incentives: unlike in a unicast setting where tit for tat is used to ensure data flow between peers, with multicast the publisher is decoupled from the subscribers: there is no direct piece exchange between the endpoints. Among the incentive mechanisms that we expect to test are the use of the rendezvous points as issuers and validators of receipts for data exchanged, exploiting the routing scheme to check at intermediate nodes whether peers are behaving well, and directly exchanging decryption keys between individual pairs of endpoints, i.e. using tit for tat between the publisher and each subscriber in the key rather than in the data exchange.

3.1.4 Socket’s API Emulator

While some major application areas, such as Web Browsing and file distribution, will be explored within the project, the scale and duration of the PSIRP project mean that there will be no resources and time to investigate other types of applications. In practice, even if the publish/subscribe model becomes prevalent in the future, it is unrealistic to expect that all network applications that people may want to execute will be rewritten in a publish/subscribe style. The network architecture and implementation developed by the project will therefore need to provide some type of backward compatibility for existing client/server applications, so as to offer a low cost migration path to publish/subscribe networking.

In D3.1 [4] we have indicated that a Socket’s API Emulator will be developed that will allow legacy applications to operate on top of the PSIRP upper layer(s), by replacing the client/server implementation of the sockets library with a publish/subscribe one. The emulator will need to support both stream and datagram sockets, that is, it will need to replicate the functionality offered by TCP and UDP to applications. In contrast to the Web Browsing application plugin, the Sockets API Emulator will not be able to take advantage of application semantics, treating each connection or datagram as a separate entity. As a result, application performance over the Sockets API Emulator is expected to be much worse than that offered by an application specific plugin, let alone a rewritten application.

Datagram sockets exchange individual datagrams, which can be modelled by individual publications if we view a publication as a single piece of data. Stream sockets on the other hand exchange sequences of datagrams that need to be reassembled exactly as transmitted; therefore they can be modelled by channel like publications. The design of the Multicast BitTorrent application will be exploited for the design of the datagram socket part of the Sockets API emulator, since multicast data transmission is a generalized case of datagram based communication, while the design of the Web Browsing plugin will serve as the basis of the stream socket part of the Sockets API Emulator, since interactions with a Web Server are at their most elementary level a set of connections over stream sockets.

3.2 Rendezvous

Perhaps the most important function in the PSIRP architecture is that of the rendezvous system. Its role is to match the interests of subscribers and publishers as well as to construct the initial data forwarding path between them in cooperation with the topology function.

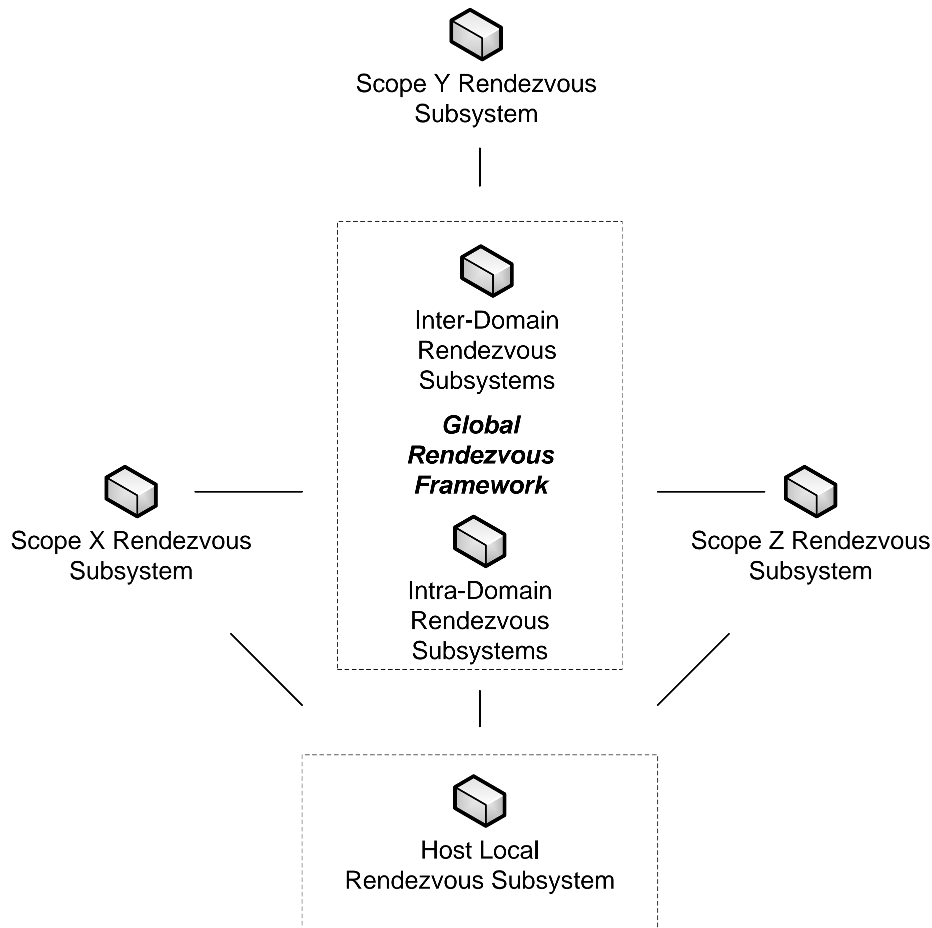

Figure 4 - The logical rendezvous architecture

In Figure 4 the logical rendezvous architecture is illustrated within a complete system consisting of three subsystems.

3.2.1 Global Rendezvous Framework

The global rendezvous framework's main task is to provide the means to globally publish and subscribe to the scope information n of the scope rendezvous subsystems. In a global network

this is not likely to be a single system; instead there will be several global rendezvous subsystems and each of them may contain different intra-domain and inter-domain rendezvous subsystems.

3.2.2 Scope Rendezvous Subsystem

The scope rendezvous subsystem is responsible for rendezvous within the specific scope it represents.

3.2.3 Host Local Rendezvous Subsystem

The host local rendezvous subsystem is responsible for node local rendezvous as well as for connecting the host rendezvous subsystem to other rendezvous subsystems and to the global rendezvous framework.

3.2.4 Rendezvous Identifier

In PSIRP architecture all data (fragments) are publications and are identified by a *Rendezvous Identifier* (RId). Each publication is published under one or more scopes, which are identified by *Scope Identifiers* (SId). (SId, RId) pairs constitute a flat identifier namespace. Semantically scope identifiers are a subclass of rendezvous identifiers typically being generated through some cryptographic mechanism like hash functions.

3.2.5 Global Rendezvous Scope and Framework

Scope Identifiers that are administrated by any of the subsystems in the global rendezvous framework are called global rendezvous scopes. All publications published under any global rendezvous scope are related to other scopes, i.e. a RId under any global rendezvous scope is pointing to metadata containing more scope related data. Thus, no standalone data items can be published under a global rendezvous scope. Scope rendezvous subsystems use this system to utilize global visibility for their scopes by publishing them under one or more global rendezvous scopes and end nodes use this system to acquire knowledge of scopes by subscribing to the publications available under the global rendezvous scopes.

3.3 Topology

Since the first complete architectural design of the topology and tree management function in PSIRP networks will only be ready at a later time (specifically for D2.3) our present design and implementation plans are focused on building up a basic framework that can serve as a basis for implementing different alternatives in the final design. At first the focus shall be on the level of intra-domain topology management, and this will be extended later on to the inter-domain case as the relevant parts of the architecture become more concrete.

The key elements in this framework are mechanisms for discovering the local topology information, exchanging that information with the necessary network components, carrying out the required computations, and then communicating any required changes in the forwarding configuration to the involved nodes. The discovery of local topology information will be primarily performed by using the techniques the network attachment mechanisms are based on. Local publish/subscribe messaging can be used to create the equivalent of router advertisements of today's networks. We plan to specify tentative formats for both the advertisement publications as well as for the messages which nodes can publish to announce their local topology information.

The network entities responsible for topology management would then subscribe to this topology information and carry out tasks related to tree construction and management. Any configuration changes related to forwarding, such as generation, modification, aggregation or destruction of forwarding trees will again be communicated to the forwarding nodes by means of publications with well-known or configurable RIds.

We plan to implement the software components to carry out the above tasks in a modular fashion, knowing that their final architectural location has not yet been defined. For example, it is not yet firmly decided whether topology management is kept as a separate entity from the Rendezvous system, with which it would interface via well-defined control messages, or whether a tighter integration is called for. The flexibility aimed at the early topology-related implementation activities is meant to cater for both possibilities, requiring minimal effort later on to apply the developed code in different architectural contexts while allowing for rapid prototyping of algorithms and designs.

3.4 Forwarding

Forwarding is used to actually deliver data from one location to another. In the first prototype it will be based on label switching, i.e. each packet will have a label (or a stack of labels) and each node will have a forwarding table as shown in Table 1.

A label is a bit string in the packet that is used by the nodes to make forwarding decisions. A label is a combination of Forwarding IDs, Scope IDs and Rendezvous IDs. Although the forwarding module should be only concerned about the Forwarding IDs, the first prototype will also examine the Scope ID and Rendezvous ID of each message, to determine whether there is a local subscription or a forwarding rule for those. Later, when forwarding is performed in kernel or in hardware, this may need to be optimized, but it is currently an open architectural issue.

A port is a local (internal to the node) numbering of the different next hops that the packet can be forwarded to, including the wired and wireless network interfaces towards the next hop forwarding nodes, as well as other software modules (such as applications or helper applications) internal to the node in question. Forwarding should be very simple so that it does not take too much time per packet.

Labels can be stacked as shown in Table 1. The first row shows that packets labeled X will be forwarded on port 1 and a label A is prefixed to the original label. The second row shows the branching of a multicast tree Y. The last row shows a packet from which the label Z is popped from the stack and the packet forwarded to port 3. The wildcard "*" denotes any label. Port 3 can even be a loopback interface, i.e. returning the packet to be processed by the forwarding table after Z has been popped from the label stack.

Incoming label	Outgoing port(s)	Outgoing label(s).
X	1	AX
Y	1 2	Y Y
Z*	3	*

Table 1 - Forwarding table

The forwarding table is a central element of node operation. Configuring its contents is the task of helper applications, such as the Rendezvous Point and the Topology Manager. The forwarding tables of the nodes in a network implicitly define and store the delivery trees and active subscriptions.

3.5 Packet Level Authentication (PLA)

Packet Level Authentication (PLA) [15],[19] is a novel way to enhance network security on the network layer by providing availability and protecting the network infrastructure from several

kinds of attacks, like *Denial-of-Service* (DoS) attacks. While the architecture work has not yet fully defined the authentication schemes, we use PLA as an example method to authenticate traffic. PLA is based on the assumption that per packet cryptographic operations are possible at wire speed in high speed networks with new cryptographic algorithms and advances in semiconductor technology.

The main principle of PLA is to detect and stop malicious traffic as quickly as possible while benevolent traffic should be allowed go through the network. The major difference between traditional network layer security solutions and PLA is that PLA gives nodes in the network the ability to detect attacks immediately by checking the authenticity and integrity of every packet, while with traditional end-to-end security solutions, like IPSec, only the end point of the connection can verify the authenticity of the packet. PLA allows every node to verify the packet independently without having to trust nodes that have previously handled the packet. It is important to note that PLA aims to complement existing security solutions instead of completely replacing them.

A good analogy to the principle of PLA is paper currency. Anyone can independently verify whether a bill is authentic simply by checking security measures inside the bill like watermark and hologram. There is no need to contact the bank which has issued the bill. Using the same principle PLA gives every node the ability to check whether the packet has been modified, duplicated or delayed without having any kind of contact with the sender of the packet.

PLA is based on cryptographic signature techniques and it adds a separate PLA header into a packet which contains all necessary information for verifying the packet's authenticity and integrity. Although PLA has been developed for IP networks, it does not depend on the network layer protocol used and therefore it can also work without the IP protocol.

3.5.1 Cryptographic solutions and hardware acceleration

To reduce the bandwidth overhead, PLA uses *elliptic curve cryptography* (ECC) [18],[20], since ECC keys and signatures are very compact. A 163-bit key used with PLA has the same cryptographic strength as 1024-bit RSA key.

Since public key cryptography is very computationally intensive, dedicated hardware for cryptographic operations is necessary for good performance. A proof-of-concept FPGA based hardware accelerator has been developed for PLA [17],[16]. The accelerator supports ECC signature verifications and can also be extended to support signature generations. Altera offers a Hardcopy technology [13] which allows an existing FPGA design to be converted into *Application Specific Integrated Circuits* (ASICs). According to simulation results, a 90 nm Hardcopy ASIC would achieve 850,000 verifications per second. Optimizing the design and converting it to a more modern manufacturing process would yield much higher performance. These results show that PLA is scalable for high speed core networks as long as a dedicated hardware is used for cryptographic operations.

3.5.2 PLA Protocol

Figure 5 below describes the structure of the PLA header and shows an example of how the PLA header can be combined with PSIRP-related information. The PLA header is marked as bold box in Figure 5. The fields of the PLA header are explained below.

The trusted third party certificate corroborates the binding between the sender's identity and its public key. It also guarantees that the sender is a valid entity within the network and is authorized by some trusted third party. To reduce computational and bandwidth overhead, PLA utilizes identity-based implicitly-certified cryptographic keys [14]. Therefore, the sender's public key can be calculated from the trusted third party certificate. This sender's public key, together with a signature, protects the integrity of a packet and guarantees that any modifications of the packet will be detected, while it also guarantees that the sender cannot deny sending the packet.

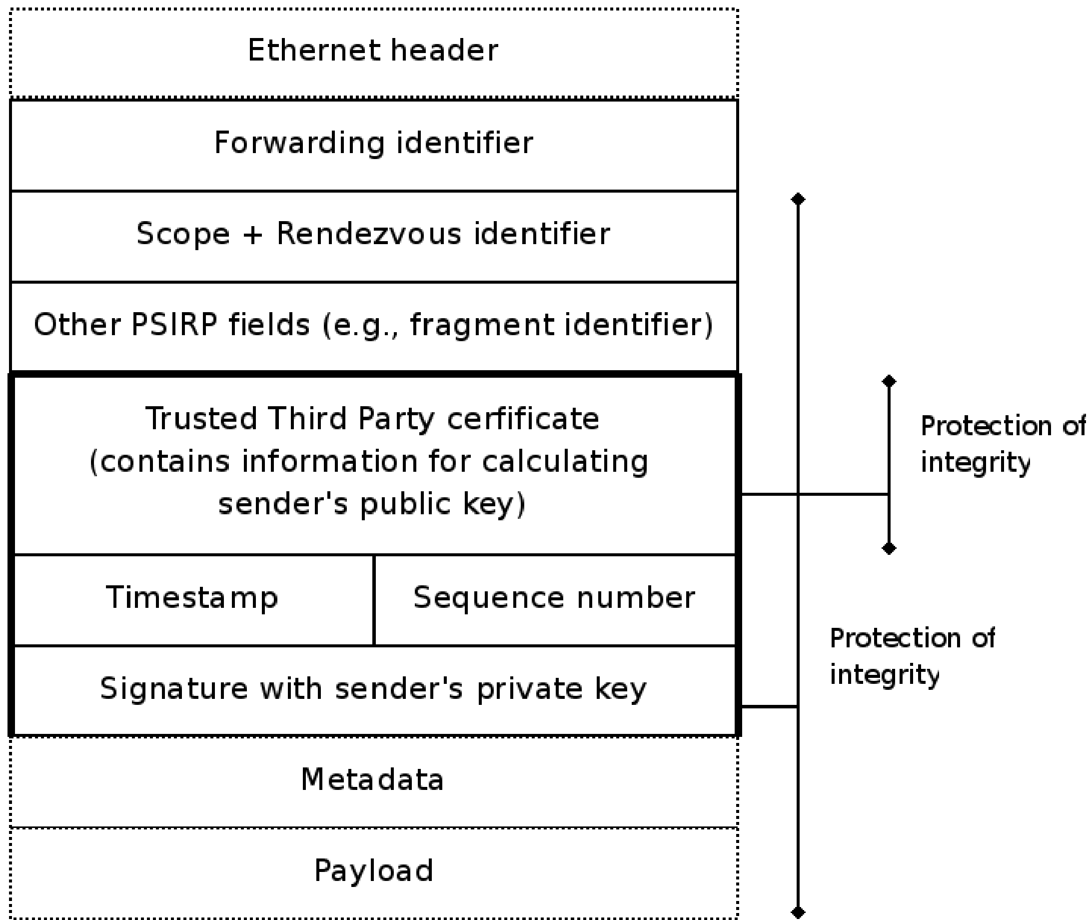

Figure 5 - The structure of the PLA header

The timestamp field makes possible the detection of delayed packets. Such packets can be a sign of a replay attack. The sequence number field contains a monotonically increasing number. It is used to detect duplicated packets. Finally, there is a cryptographic signature over the whole packet ignoring Ethernet header and forwarding identifier fields which can change during the lifetime of the packet. The total length of PLA header is roughly 1000 bits

It is important to note that this header design is preliminary and it will probably change based on our experiences with it; some header fields from the PLA header may not be necessary for PSIRP while other fields may need to be added. It is also optional for nodes to decide how strict PLA related checks will be performed. They may opt to check only some of the PLA header's fields or ignore PLA header information altogether.

3.6 Network Attachment

Network attachment is the component responsible for discovering attachment points and configuring components in such a way that communication becomes possible. An attachment process may include the following operations:

1. Attachment point discovery and network selection
2. Communication establishment with the attachment point
3. Authentication of entities, and/or authorization to use the network

4. Exchange of configuration information and setting up components for communication

3.6.1 Operations

From the perspective of a joining node, the first step in attachment is usually to find another node which can be used for accessing a network. Information available in this phase may include the attachment point's (or network's) identity and capabilities, as well as some basic configuration parameters. The second step is to initiate communication with a selected node, first using basic forwarding and link layer mechanisms. In our model, the result of this operation is a "control channel", where communication between network attachment daemons in two nodes can occur using publish/subscribe functions. For example, this implies that each participant has to know what identifiers its counterpart has subscribed to. If needed, this channel may be integrity protected, and also confidential. Thirdly, entities might need to be authenticated in a secure, but possibly opportunistic manner. In relation to this, nodes can be authorized to use certain services or perform certain operations in the network. Authorization can be based on authentication, but also on a contract defining services and compensation for them. The fourth issue we have identified is, of course, exchange of configuration information. This data is used for setting up other components, such as rendezvous, at both participants in order to enable communication. In addition, security associations can be set up during an attachment procedure. Notably, some of these operations can, at least partly, be run in parallel, within the same messages, that is, publications.

3.6.2 Participants

In addition to the two main participants that become attached to each other, network attachment can involve other entities, such as authentication and rendezvous systems. Furthermore, attachment can be performed between a single node and an existing network (asymmetric scenario), but also between nodes that both are connected to networks (symmetric scenario).

3.6.3 Relation to other functions

Network attachment is used for bootstrapping rendezvous in a node that joins a network. It involves setting up required forwarding state between the node local rendezvous system and the network’s rendezvous subsystem. In addition, we note that attachment usually implies a change in network topology. This is also closely related to the problem of mobility.

3.7 Interfaces

The current conceptual architecture defines only one generic interface, which is called pubsub-api. All communication between the applications and the networking functions is enabled through this API. The API will define required functions to support needed operations in the PSIRP system. In the purest form the only needed primitives are publish and subscribe. However, due to the requirements of the helper applications (e.g. Rendezvous), there may be a need for extended functionality besides the pubsub-api. This extended functionality for helper applications is provided by the Advanced API in the current implementation plan and described in more detail in section 4.4.3. The Advanced API includes functions that are not part of the publish-subscribe architecture, but are needed to implement the node architecture. An example function is the configuration of the forwarding table that is needed by both the Rendezvous Point and the Network Attachment.

4 Work Plan

4.1 Implementation Overview

This chapter outlines the implementation for each functional entity that will be included in the first iteration round prototype. It is important to keep in mind that the level of detail is kept reasonable to avoid overwhelming the reader while still providing sufficient detail to understand the current implementation of the conceptual architecture as described in D2.2 [1]. As outlined in the introduction, it is likely, and even desirable, that details of our implementation will change due to the continuous learning process and the interaction with the architecture team. Examples of such open topics are the packet header and payload formats.

4.1.1 Architecture

This section describes the architecture of a network node in the prototype network (see Figure 6). All nodes have a similar architecture, although they may not implement all the functions shown in the figure. For example, a forwarding node may implement more or less functionality than a dedicated end-host.

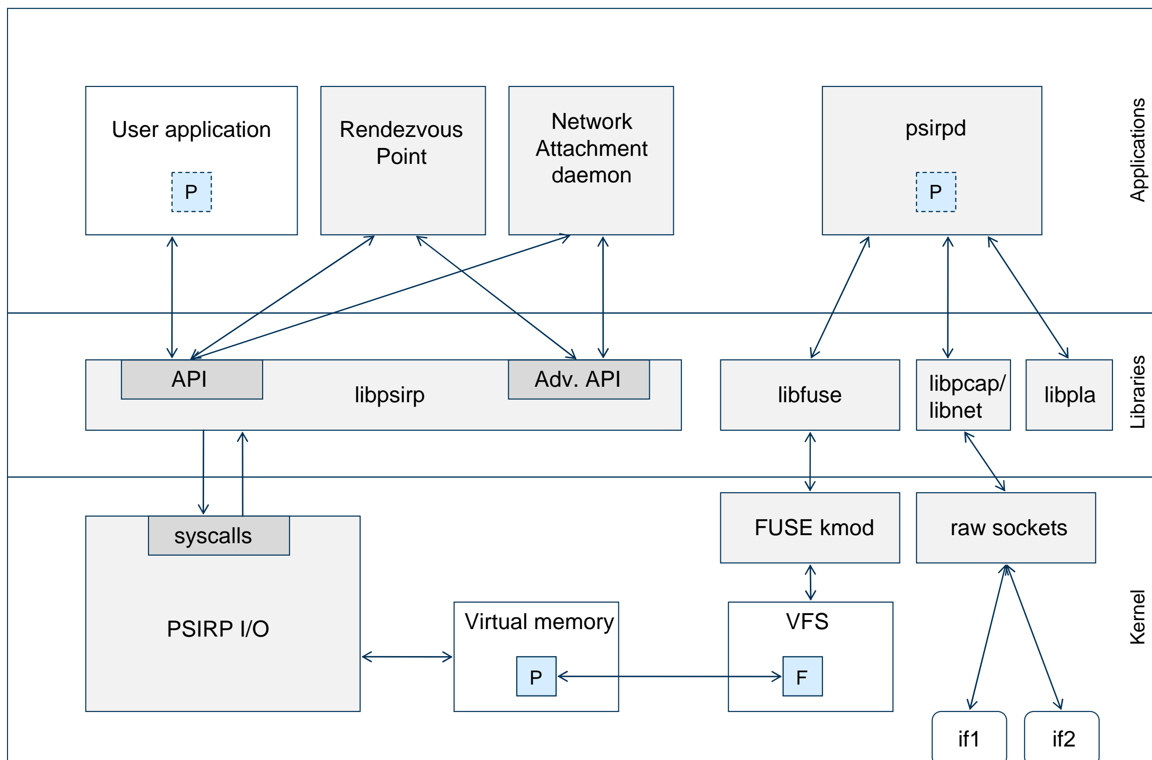

Figure 6 - The Overall Node Architecture

If we assume that the user level publications can practically be of any size, from one byte to several terabytes, and that users can read and write to them in random positions, the traditional socket API does not seem to be the most efficient solution any more. From the application developer point of view, instead of thinking about sockets and end-points, it is easier to just read and write to a memory location. As all general purpose operating systems of today provide virtual memory, which has become abundant with 64-bit address spaces, even the largest publications can be mapped to the process address space. To keep the implementation efficient, unnecessary copying between memory areas should be avoided.

To implement local representation of publications as virtual memory areas, some parts of the implementation need to be in kernel space. It is possible to implement the needed extensions in a kernel module which can be loaded and unloaded without the need to recompile other kernel parts. The kernel module needs to implement a new system call. This system call can be used to create publications, publish them and subscribe to publications. The kernel module handles the allocation of memory used for publications. As publications are just regular virtual memory areas, normal FreeBSD virtual memory management systems will handle them as any other application memory in the system, e.g. swap them out if the system is running low on memory.

While this core functionality of memory allocation is performed in a kernel module, it is better to place non-performance-critical components in user space. This allows easier development of prototype software both in terms of easier debugging but also for dividing work between developers. In the beginning, most of the prototype code will be in user space. We can put, for instance network I/O and PLA in a daemon process, which is linked to publications via the pager interface in the kernel, as it seems to be possible to take advantage of the *file system in user space* (FUSE) software [5]. With FUSE it is possible to create a file system in user space as backing storage for publications in virtual memory: when allocating memory for a publication, the pager for the memory area can be set to be a vnode pager, and the backing file to be in FUSE. In this manner we can trap fetching of missing memory pages (= subscribing) and paging out memory pages (= publishing).

Rendezvous, topology etc. could be implemented as user space daemons just like the network I/O daemon and the network attachment daemon, if we assume that they can be implemented via the same system call interface. Compared to user applications however, some of these daemons may need extended abilities to, for example, configure the forwarding tables in the network I/O daemon.

4.1.2 Network Nodes

The prototype network is a small scale network operating inside a single administrative domain. Compared to a solution that covers multiple domains, some functions are simpler, for example the rendezvous and topology management. The assumptions made during the first iteration round are therefore simpler, since there is no need for developing solutions for communication and information exchange between different domains that potentially have different preferences, for example, with respect to the choice of rendezvous system.

The nodes that are needed in the network are a set of forwarding nodes, providing both some rendezvous functionality as well as forwarding functions, a rendezvous point, providing the "root point" of the rendezvous mechanism, and end-hosts that are publishing and subscribing data. Note that the Rendezvous function in this case is limited to a simple root node: this is possible when the prototype is only a single domain implementation. To cover multiple domains, the Rendezvous mechanism will be developed further and the second iteration round will provide a prototype of that inter-domain system.

4.1.3 Signaling Model

Figure 7 represents our current understanding of the prototype signaling model, i.e. how internal and external signals flow between the functions and network nodes in the prototype during the supported operations. Because signal headers and payload formats are still open, they are not drawn in detail. Below the figure each signaling step is described in more detail.

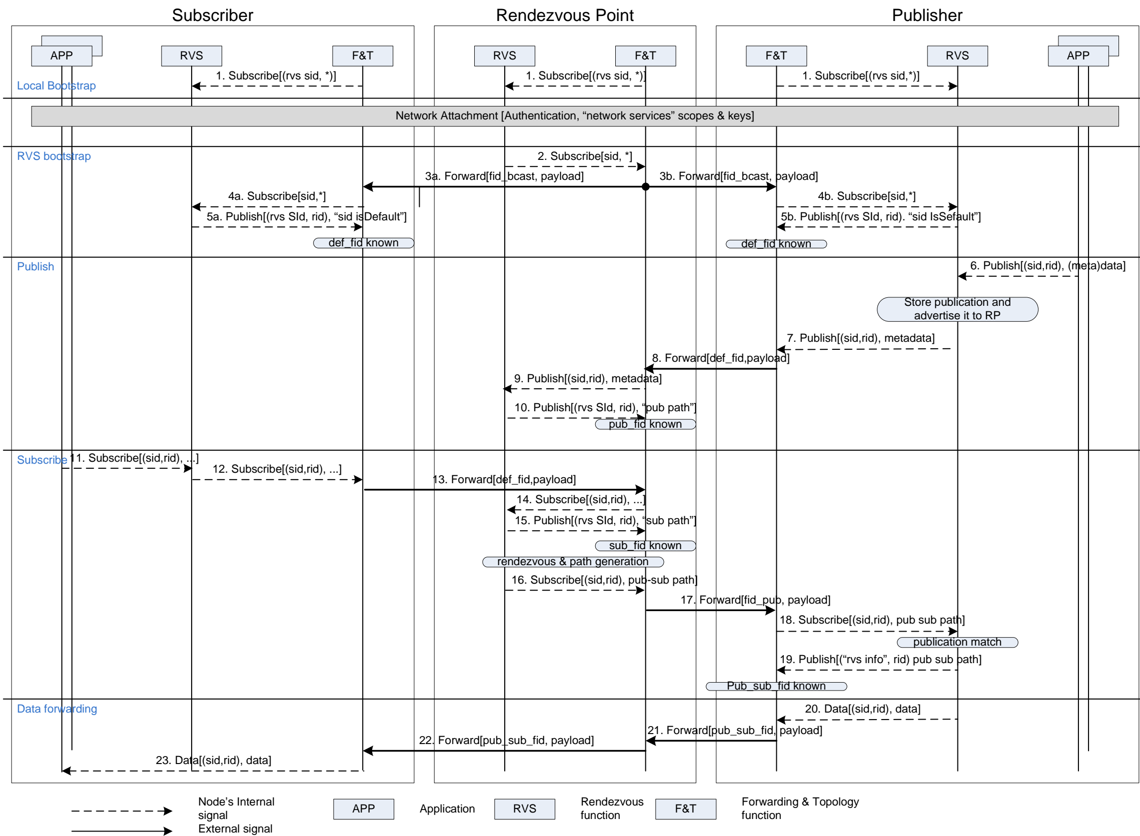

Figure 7 - The Signaling Model

4.1.3.1 Local Node Bootstrap

1. The topology function in each node subscribes to all local host rendezvous information. Rendezvous information carries forwarding table updates, which are published by the (local) rendezvous function.

4.1.3.2 Network Attachment

In the Network attachment phase, end nodes are authenticated and attached to the network. During this signaling all domain specific network service scopes, such as domain rendezvous scope and domain topology scope, are provided to the nodes.

4.1.3.3 Rendezvous Bootstrap

2. The rendezvous function of the Rendezvous Point (RP) subscribes to the domain's default rendezvous scope. This signal is advertised by the forwarding function to all nodes belonging to a domain.
3. The subscribe signal sent from the RP's forwarding function will first enter Publisher/Subscriber nodes via their forwarding function (fast path).
4. The Publisher/Subscriber node forwarding function drops the signal to the slow path and then provides it to the node's rendezvous function.
5. The subscribe signal triggers generation of rendezvous information in the publisher/subscriber rendezvous function. The publisher/subscriber topology function is subscribed to it and will receive it. The content of the update is "default rendezvous path".

4.1.3.4 Publish

6. One the publisher's applications uses the pubsub-api to publish data. Eventually the publish call will enter the publisher's rendezvous function, where the publication is stored and an advertisement for the publication in form of a new publish signal is triggered.
7. The publisher's rendezvous function uses the Advanced API to trigger a publish signal that advertises the publication to the RP. This signal is passed to the publisher's forwarding function
8. The publisher's forwarding function receives the signal, chooses the right forwarding identifier (FId) and then forwards the signal towards the RP using the domain's default rendezvous path.
9. The forwarding function of the RP receives the signal, lifts it to the slow path and provides it to the local rendezvous function, where the publication is added to the rendezvous databases accordingly.
10. The rendezvous function publishes a rendezvous update that contains a path to the publisher. The topology function has subscribed to it, so it will receive it locally and generate/update the corresponding forwarding table entry.

4.1.3.5 Subscribe

11. Some subscriber's application wishes to subscribe to the publication that was published in steps 6-10. Subscribing happens using the pubsub-api with appropriate parameters, at least SId and RId. Eventually, the subscribe call reaches the rendezvous function, which then generates and triggers a subscribe signal, which passes by the local forwarding function in the same manner as in the publish phase.
12. The subscriber's forwarding function receives the signal, chooses the right FId and forwards the signal towards the RP using again the domain's default rendezvous path.
13. The RP's forwarding function receives the subscribe signal, lifts it to the slow path and provides it to the local rendezvous function.
14. The local rendezvous function receives the subscribe signal and checks whether it has related publication in the databases - In this case it has.
15. The rendezvous function publishes a rendezvous update that contains the path to the subscriber. The topology function has subscribed to it, so it will receive it locally and generate the corresponding forwarding table entry. This entry may be needed e.g. in case the subscriber wants notification about the successful subscribe before the actual data. In this message sequence chart such a notification is not used.
16. The rendezvous function must notify the publisher about an existing subscription to enable the data transfer. The rendezvous function generates a forwarding path between publisher and subscriber and adds this information as metadata to the received subscribe signal and triggers signaling towards the publisher using the pubsub-api.
17. The RP's forwarding function will send the subscribe signal using the publisher's Fid.
18. The publisher's forwarding function receives the subscribe signal, lifts it to the slow path and provides it to the rendezvous function. It is matched with the cached data and data transfer is initiated.
19. The received subscribe signal triggers generation of rendezvous information in the rendezvous function. The publisher's topology function has subscribed to it so it will receive it locally. The content of the update is path information from the publisher to the subscriber.

4.1.3.6 *Data Transfer*

20. It is assumed that the rendezvous function would map the data and trigger data sending, but it may also be some other entity.
21. The publisher's forwarding function forwards data packets towards the subscriber using the related FId.
22. The RP's forwarding function forwards data packet towards the subscriber – this all happens on the fast path.
23. The subscriber's forwarding function receives the data packet and maps the data to the application memory page, who then can read the data.

4.2 Helper Functions

The PSIRP prototype is a combination of system level functions that are located both in the kernel and in the user space, as well as a set of user space helper functions that use system level functions to implement the desired set of prototype functions. In this section, the implementation plans for the helper function are described.

4.2.1 Rendezvous

In Figure 8 the architecture of the rendezvous function is presented.

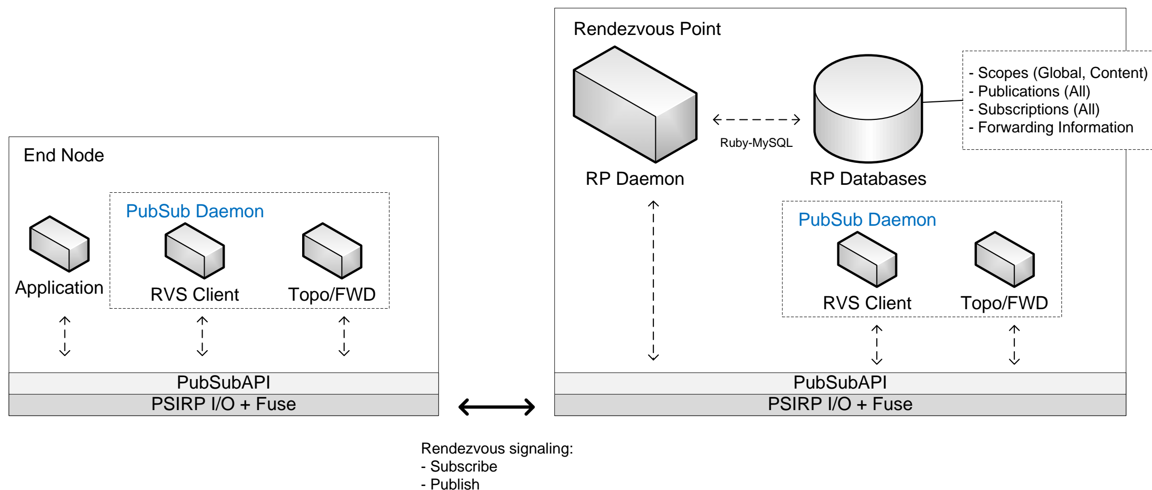

Figure 8 - The rendezvous prototype architecture

As defined in the conceptual architecture, the rendezvous function is distributed and every node in the architecture takes part in it one way or the other. In the first PSIRP rendezvous prototype, the target is to implement essential parts of the intra-domain rendezvous function and to integrate it to other existing prototype function modules. In practice, the rendezvous function tightly co-operates with the topology and forwarding function, both of which are implemented in a user space demon process.

The rendezvous implementation will consist of two functional entities:

- The Rendezvous Client (RVS Client), also referred to as minimal rendezvous function, implements the rendezvous functionality needed in the nodes that wish to subscribe and/or publish, even only locally (intra-node). In the PSIRP architecture this means that in practice every node in the topology must implement this functional entity.

- The Rendezvous Point (RP) implements part of the rendezvous needed to support distributed operations beyond one node. The RP is the location where inter-node publish and subscribe signals will meet. In other words, a domain RP is the entity which is aware of all subscriptions and publications within a domain. Further a rendezvous point adopts one of the three different functional modes:
    - *Interior Rendezvous Point* (IRP) supporting only intra-domain rendezvous
    - *Exterior Rendezvous Point* (ERP) supporting only inter-domain rendezvous
    - *Rendezvous Point* (RP) supporting both intra- and inter-domain rendezvous

In the first prototype inter-domain rendezvous is out of scope and therefore IRP and RP are used as synonyms.

It is likely that the conceptual architecture will also require a rendezvous entity in the forwarding nodes, but the forwarding node rendezvous function will not be implemented in the first stage of prototyping.

The current conceptual architecture has several options of how intra-domain forwarding could be implemented. This is naturally important for rendezvous, because the forwarding function will be used to forward the rendezvous signaling. The lack of decision on how the forwarding in the intra-domain level will ultimately be done, given the possibility that there are currently several currently equally good options, is one of the reasons the forwarding node rendezvous entity is left out. Said decision mainly affects the implementation of the forwarding node rendezvous function.

4.2.1.1 Rendezvous Functional Entities

Here we assume that no distinction will be made in the RC or RP functions in the handling of metadata and data. In other words, data stored in the trusted node's database is assumed to be valid at all times and the question whether it is data or metadata describing some data is out of scope for rendezvous. Therefore the primitive handling will be identical in all cases. In the RVS Client this means:

- If subscribe results match an entry in the local node cache (pubsub daemon), no further checks for the data type etc. will be made, but the data will be mapped to the requesting process and further rendezvous signaling is not needed.
- If subscribe does not result in a local match, rendezvous signaling will be triggered towards the RP.

Rendezvous Client

The rendezvous client is assumed to implement at least the following functionalities:

- Initiating subscribe signaling towards RP.
- Initiating publish signaling towards RP.
- Listening to the rendezvous advertisement(s) from the network and bootstrapping of the rendezvous function in the local node, if needed. These advertisements are coming from RPs or from the on-link forwarding node as defined by the current conceptual architecture.
- Maintain active publish and subscribe states.

The current understanding is that the rendezvous client will be implemented as part of the pubsub daemon using C. Alternatively the client can be implemented using Ruby as with the RP rendezvous.

Rendezvous Point

The Rendezvous Point enables at least the following functions:

- Periodical rendezvous advertisement (subscribe messages carrying RP SId(s)) needed to enable intra-domain rendezvous bootstrap and advertising which scopes can be reached through a specific RP. Advertisement is done using the pubsub-api.
- Maintain scopes database based on the configuration of the RP and in later phases (inter-domain) based on received inter-domain rendezvous advertisements.
- Maintain subscription database for the received subscriptions
  - o Pre-establishing subscriptions should be supported.
- Maintain publication database for the received publications.
- Share database information with other domain RPs to prevent a single point of failure
  - o Not in the first prototype.
- Forwarding notification signal (subscribe) to the publisher, if a valid entry exists in the publication database.
- Provide forwarding information updates to the topology function based on contents in the databases and received rendezvous signals.
  - o Routing of actual data is based on the forwarding table.

The RP rendezvous function will be implemented as a separate module using Ruby (and C extensions) in the same way as the network attachment application is implemented in Python. The module is planned to use PSIRP primitives from the pubsub-api to enable the intra-domain Rendezvous Point function. Because the pubsub-api is implemented in C, one of the implementation tasks is to implement an appropriate wrapper interface between the RP and the API.

4.2.1.2 Rendezvous Function (intra-domain)

The intra-domain rendezvous function is responsible for enabling subscriptions and publications to meet each other. However, before this can happen, the rendezvous system and the forwarding path towards the Rendezvous Point must be bootstrapped within the domain.

Rendezvous Bootstrap

The rendezvous bootstrap is initiated through the RP's periodical rendezvous advertisements which in practice are subscribe messages carrying one or more SId(s) that are accessible through the RP. The scope ID(s) in the advertisement contains at least one SId that enables the default forwarding path to the local domain RP. The RP floods this advertisement to all interfaces. According to the conceptual architecture forwarding nodes will forward this rendezvous advertisement onwards until all nodes within the domain have received it and thus know through which interface a certain RP is reachable. Notice that there can be several RPs within a domain and a RP may be reachable from several interfaces. However, in the first prototype there is only one RP.

Subscribe

When an application wishes to receive some information, it uses the pubsub-api to subscribe to the content. In practice subscribe is a system call provided by the pubsub-api and pubsub_io kernel module. Valid subscribe system call contains at least a scope identifier and a rendezvous identifier as its parameters.

The system call eventually reaches the RVS Client, where rendezvous signaling is triggered, if needed. First, before initiating signaling, the client checks for a possible cache match for the subscribed metadata/data:

- The subscribe signal is forwarded using a FId mapped to (SId, RId), if no data exists. The default mapping is towards the domain RP.
- If data exists, it is mapped to the requesting process

Secondly, if no valid cache match was found, the client checks that the scope in the subscribe signal, defined by the application, allows for signaling outside the local node. Actual forwarding of the subscribe message is the responsibility of the local forwarding function. After triggering the subscribe message the RVS Client prepares to receive a response from the RP:

- Subscribe Notification, when RP confirms that subscription was received, but the rendezvous is still on-going.
    - This is not implemented in the first prototype.
- Not found signal, when a publication identified by a (SId, RId) pair does not exist in the known rendezvous system.
    - This may be implemented in the first prototype.

Or from the publisher:

- Data, when sending the actual data identified by (SId, RId) pair.

Eventually the subscribe message sent by the end-node's local forwarding function reaches the RP rendezvous function. The rendezvous function in the RP implements at least the following tasks for each received subscribe message:

- Valid subscription information is added to the subscription database, with an appropriate lifetime. The lifetime is needed to handle the pre-established subscription case, i.e. when a subscription happens before the actual publication exists. The lifetime may be part of the subscribe message or it might be based on some RP policy.
- The subscription is checked against the publication database.
    - If a publication entry exists in the database, a forwarding information update is generated and the subscribe signal message is forwarded towards the publisher.
- If a publication entry with data exists in the publication database, the data is forwarded towards the subscriber and no further signaling is needed.
    - In the first prototype data is always provided from the publisher to the subscriber.
- If a publication entry does not exist in the publication database, the subscription may remain in the subscription database until the publication arrives at the Rendezvous Point or the lifetime entry expires.
    - In the first prototype implementing pre-established subscriptions is low priority.

Publish

When an application wishes to publish some information it uses the publish system call provided by the pubsub-api. The publish system call requires at least scope identifier, rendezvous identifier and the data from the application. Of these parameters the data is mandatory, whereas the identifiers can be generated later on in the pubsub daemon, if the caller application didn't provide.

As in the case of subscribe, the publish system call will reach the pubsub daemon and RVS Client where rendezvous signaling is triggered, if requested publications did not exist in the local cache. The RVS Client checks the semantics of the publish system call and generates a corresponding publish signal. The actual sending of the publish signal is the responsibility of the forwarding module. During the system call the publication is stored by the pubsub daemon.

Eventually the publish message sent by the Publisher's local forwarding function reaches the RP rendezvous function. The rendezvous function in the RP implements at least the following tasks for each received publish message:

- Valid publication information is added to the publication database, with appropriate lifetime provided by the publisher.
- Publication information is checked against subscription database to find possible pre-established subscriptions.
    - If pre-established subscriptions do not exist, no further action is needed.
    - If pre-established subscriptions do exist, the publisher will be notified about it and data transmission from the publisher begins towards subscriber(s) via the RP.

4.2.2 Topology

In the first iteration round, the topology management function is not yet implemented as a separate entity. Because topology management is responsible for creating forwarding information and updating the necessary forwarding table entities, some operations are needed and our solution to handle this is to implement a minimal set of topology functions as tightly integrated with other functions, such as the forwarding function.

4.2.3 Network Attachment

The first prototype, the *network attachment* (NA) daemon is implemented as a user space application that uses publish and subscribe operations for communication. It implements a network attachment protocol that provides a framework for:

- acquiring information about potential attachment points (APs)
- establishing a control channel between a node and its access point (i.e., two NA daemons)
- negotiating about services and compensation
- exchanging configuration information, e.g. identifiers needed for setting up rendezvous

The protocol will also include support for opportunistic authentication and message integrity protection.

An example message exchange between one NA daemon acting as an initiator (X) and another one acting as a responder (Y) is shown in Figure 9. In that message sequence chart, Y initially advertises itself by broadcasting publications on the link. At some point, X joins the link and subscribes (internally) to these advertisements (A) received from its network interface. From those messages X can learn an algorithmic identifier set (P) that Y has subscribed to, and use it to publish a service/compensation contract proposal, a subscription to return messages (I), and some other information (as needed), directed to Y. Then, Y subscribes to messages coming specifically from X (R), and continues the contract negotiation and configuration information exchange. Finally, Y authorizes X to use the network and publishes the corresponding contract. After this, the channel between X and Y remains and can be used, e.g., for updating configuration or compensation data.

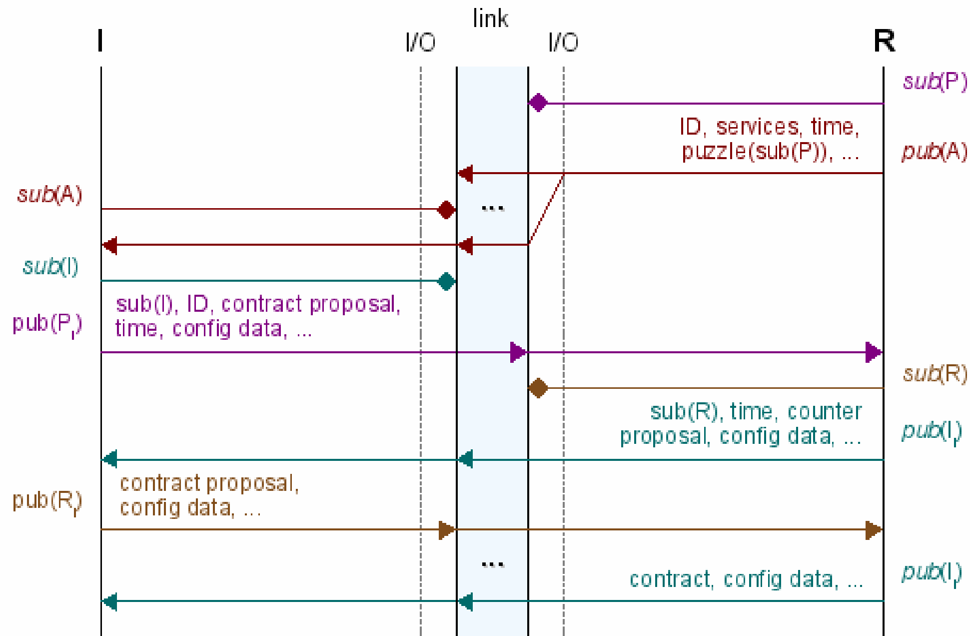

Figure 9 – An example network attachment message sequence chart

4.2.3.1 Implementation

The prototype NA daemon is implemented in Python. It uses the API provided by libpsirp. Since that interface is written in C, a wrapper that makes the functions available in Python and that converts C's data types into Python objects and vice versa, is generated with SWIG. In addition, the daemon should be able to access the "low-level" forwarding mechanism in order to set up the initial state for communication on a link. As long as the native publish/subscribe I/O is lacking networking functionality, the attachment protocol implementation may be tested between separate nodes by using an independent publish/subscribe engine and UDP sockets for link emulation. This architecture is shown in Figure 10.

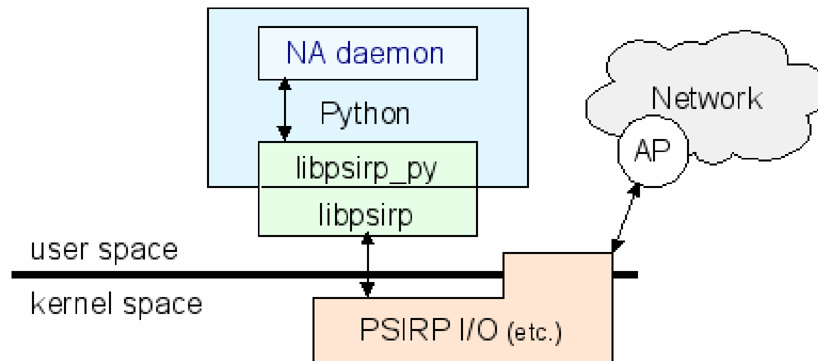

Figure 10 – The network attachment implementation: High level architecture

4.3 PSIRP Daemon and I/O

This section describes the implementation and operation of the PSIRP Daemon and the PSIRP I/O module. The operation is described with use cases for the different functions, such as publish and subscribe, and also implementation specific functions such as creating and modifying a publication. Possibilities given by using the NetFPGA as an implementation and evaluation tool are also described in this section.

4.3.1 PSIRP I/O module

The PSIRP I/O module is a loadable FreeBSD kernel module that implements the base for the API, the Advanced API and acts as a gateway between user level applications and daemons and the PSIRP Daemon. It implements new system calls on top of which the PSIRP Library, libpsirp, can be built.

4.3.2 PSIRP Daemon

The PSIRP Daemon is a multi-thread user space daemon written in C. It will host a number of functions:

- Forwarding
- Network I/O
- Local publication list
- Rendezvous Client
- Security functions, e.g. PLA

As publications are presented to the node as memory areas, we need a way to send out published memory areas and receive memory areas from nodes for publications that have been subscribed to. A convenient way to implement this is to use the existing virtual memory pager mechanism in FreeBSD. In FreeBSD there are different pagers to choose from, but none of them is suitable for our purposes without heavy modifications. As we try to avoid changing the existing kernel, we have chosen to use the vnode pager, which reads and writes memory pages into files. By creating our own file system in user space, with the help of the FUSE framework [5], we can handle the memory requests in the PSIRP Daemon and map them to network I/O. The publication list stored in the daemon is then presented as a file system through the VFS interface [6]. A separate thread in the daemon process is used to handle file system requests.

In the other thread, we handle sending and capturing Ethernet frames. We have a forwarding table and for each incoming PSIRP message we search for a match. If there is no match, the frame is dropped. If there is a match the forwarding table gives a function pointer to the handling function. The handling function may then either store the data from the frame into memory (publication) or forward the message out on some other network interface (if the policy says that the node is a forwarding node).

4.3.2.1 PLA

Packet Level Authentication (PLA) will be implemented and integrated with the PSIRP daemon as a software module. The implementation will consist of adding a PLA header into packets and validating them using software based cryptographic solutions.

4.3.3 PSIRP Daemon and PSIRP I/O Interaction

The following subsections define the publication operations. The operations cover the creation and edition of a publication at the host, as well as publish and subscribe operations for delivering the publication in the network.

4.3.3.1 Create

Figure 11 illustrates the set of procedures that are required at the node to create a publication. Initially, an application calls the new ‘create’ system call to allocate memory for its data. The data stored in this memory area will be published in the later phase after calling a separate ‘publish’ system call. The ‘create’ system call has a close analogy to the existing ‘malloc’ system call, i.e. they work in a similar way, from the application developer’s point of view. The main difference is that the ‘create’ system call ends up in the PSIRP kernel I/O module that creates a virtual memory (vm) object.

The virtual memory object is mapped to a set of free memory pages where the data will be stored. In a way, the vm-object works as a handle for the full publication in the kernel, while the memory pages can be seen as 4096 byte chunks of the full publication. We use the standard FreeBSD based virtual memory management primitives to handle the memory access between the application and memory pages. The PSIRP kernel module maps the vm-object to the calling application’s virtual memory (vm) map and returns the memory address of the beginning of the allocated memory area. The virtual memory map contains a mapping between the address space that is visible at the user-space for the application and the actual memory pages stored in the kernel-space. It is important to mention here that the created vm-object will also be mapped to the PSIRP daemon’s vm-map after the ‘publish’ system call. The PSIRP kernel module allocates an additional memory page for the metadata that is related to the data stored to the rest of the memory pages. Typically, the application fills the data in the memory area and calls the ‘publish’ system call.

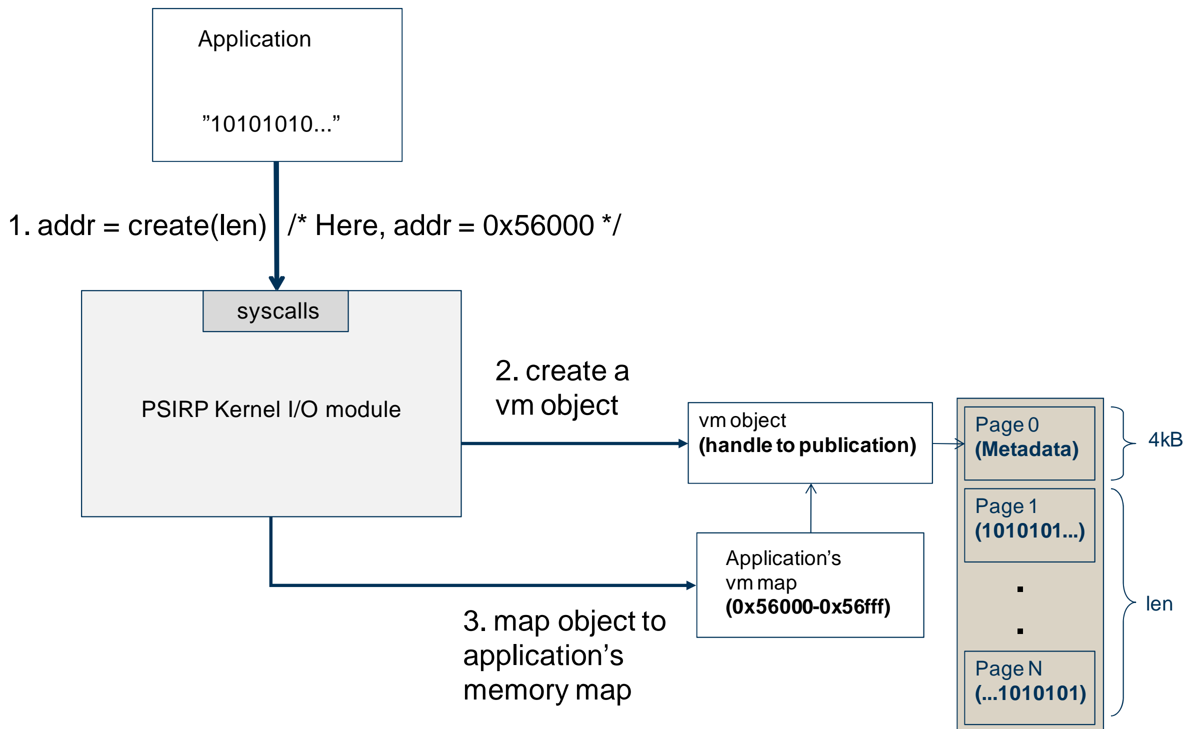

Figure 11 - Creating a Publication

4.3.3.2 Publish

Figure 12 illustrates the set of procedures that are required to publish data at the node. The application gives at least the memory address and the length for the 'publish' system call. It is important to note that the application is able to publish only part of the created data. Basically, the memory address passed to 'publish' may point to any part of the memory earlier allocated area via 'create' with the length informing the system how many bytes should be published starting from that memory address. The application may also give the SId and RId in the system call. However, if the values are not given, the PSIRP daemon generates the IDs for the data to be published.

The 'publish' system call ends up at the PSIRP kernel module that maps the vm-object to the PSIRP daemon's vm-map. The first memory address stored in this created vm-map entry works as a file name in the 'open' call in the kernel. The 'open' call ends up at the FUSE daemon's 'open' call-back function. It is important to notice that the 'open' call ends up at the PSIRP daemon because the /pubsub directory in the FreeBSD virtual file-system is a mount point for our own PSIRP file-system. The PSIRP daemon stores the memory address i.e. the file name, to its local publication list. As a result, the daemon is able to access the memory pages created by the application and publish the metadata to the RP. Once the 'open' call returns, the PSIRP kernel module sets a copy-on-write flag on the memory pages.

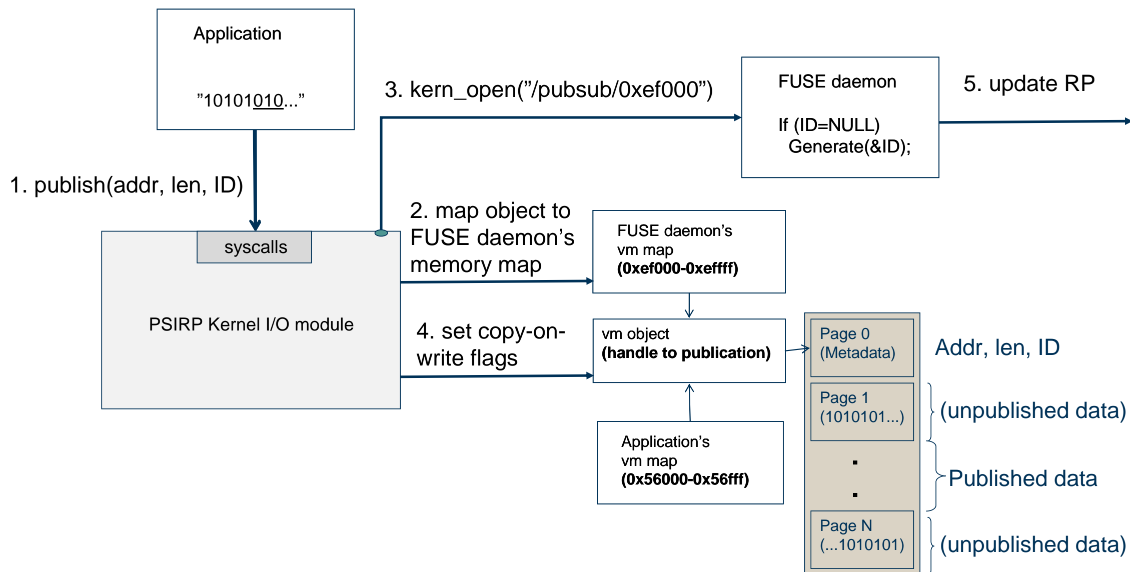

Figure 12 - Publishing a Publication

4.3.3.3 Edit

Once the application writes to a memory area that was earlier published or subscribed there will be a page fault. This is illustrated in Figure 13. In this case, page fault means that the application was writing on a memory page that had copy-on-write flag on. This is a default FreeBSD kernel behaviour and not implemented in the project. The hardware page fault is handled by the kernel virtual memory management that will eventually create a vm-shadow-object for the existing vm-object. The shadow object is mapped to the original vm-object having a mapping to the original memory pages. The memory page that is affected by the write operation is copied and mapped to the vm-shadow-object. In addition, the application's virtual memory map is updated to point to the shadow object. The copied and edited memory page can be seen as a delta chunk from the versioning point of view. In a way, the original vm-object works as a handle for the original publication, the shadow-object works as a handle for the delta. This kind of functionality has a strong relationship to the well-known version controlling systems like CVS [8] and SVN [9]. In addition, it is possible to see the analogy between vm-shadow-objects and ZFS [10] snapshots that are both based on "deltas", i.e. storing only the differences between versions instead of complete versions.

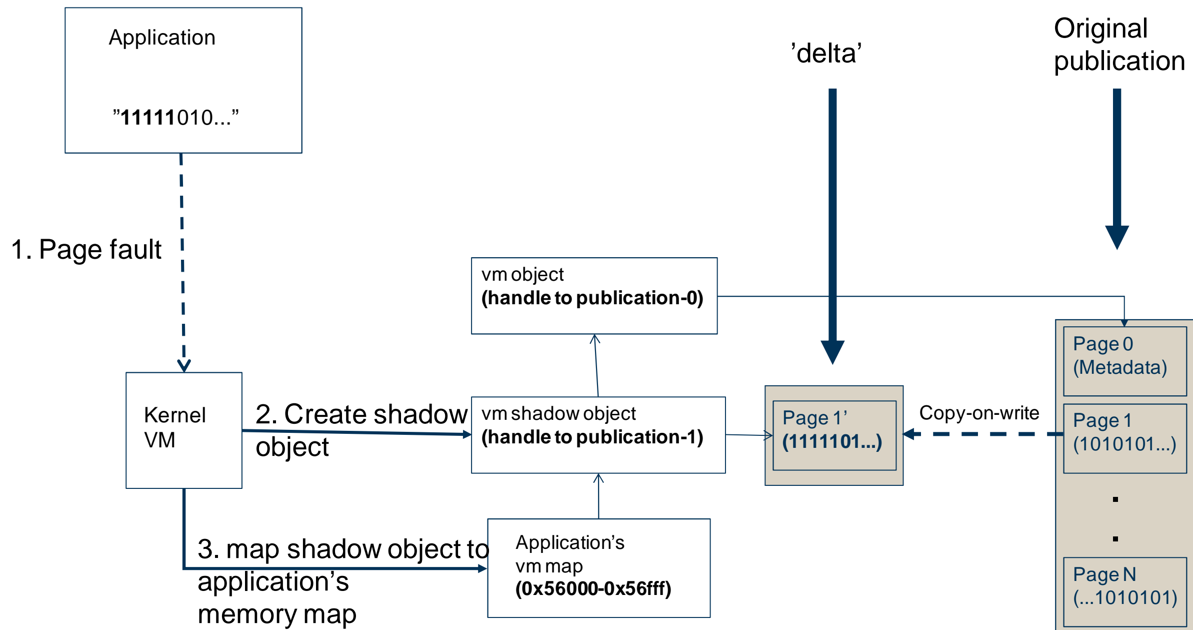

Figure 13 - Writing to a Publication

4.3.3.4 Subscribe

Figure 14 illustrates the set of procedures that are required at the node to subscribe to data. The 'subscribe' system call contains at least the RId. The system call ends up at the PSIRP kernel module. The RID is given as a filename in the 'read' call to the PSIRP daemon. The daemon returns the memory address of the metadata location if the metadata is already locally cached at the host. Otherwise, the daemon triggers a subscription towards the network. Depending on the parameters given in the 'subscribe' system call, the 'open' call from the kernel module to the file system may be non-blocking or blocking. In the former case, the 'open' returns right after triggering a subscription in the network. In the latter case, the 'open' call returns once all the data is received and stored locally at the host.

Once the 'open' call returns, the kernel module creates a new vm-object and associates it with a vnode pager. Using the vnode pager it is possible to have the page faults end up with the PSIRP daemon. (Otherwise, the FreeBSD's default pager would page out memory pages to the swap partition.). If the 'open' call returns before metadata is received, it allocates a PSIRP-INITIAL size of memory for the data. It may be possible to resize the memory area after getting the metadata, the metadata containing the length of the subscribed data. Based on this information the kernel module is able to reserve the right number of memory pages that are mapped to the vm-object. In addition, the vm-object is mapped to the application's virtual memory map. The 'subscribe' system call returns a memory address to the application. The subscribed data is available for the application starting from this memory address.

As earlier mentioned, if the daemon did not fetch data after the 'open' call, the memory pages do not contain the subscribed data. When the application accesses the memory area that was allocated for the data there will be a page fault. The page fault ends up with the PSIRP daemon via the vnode-pager. The daemon triggers a subscription procedure for that specific memory page. The 4096 byte memory page is the smallest size of data chunk that is used in the architecture. It is carried in jumbo Ethernet frames in the link. Therefore, the prototype does not need to implement fragmentation between the end-points.

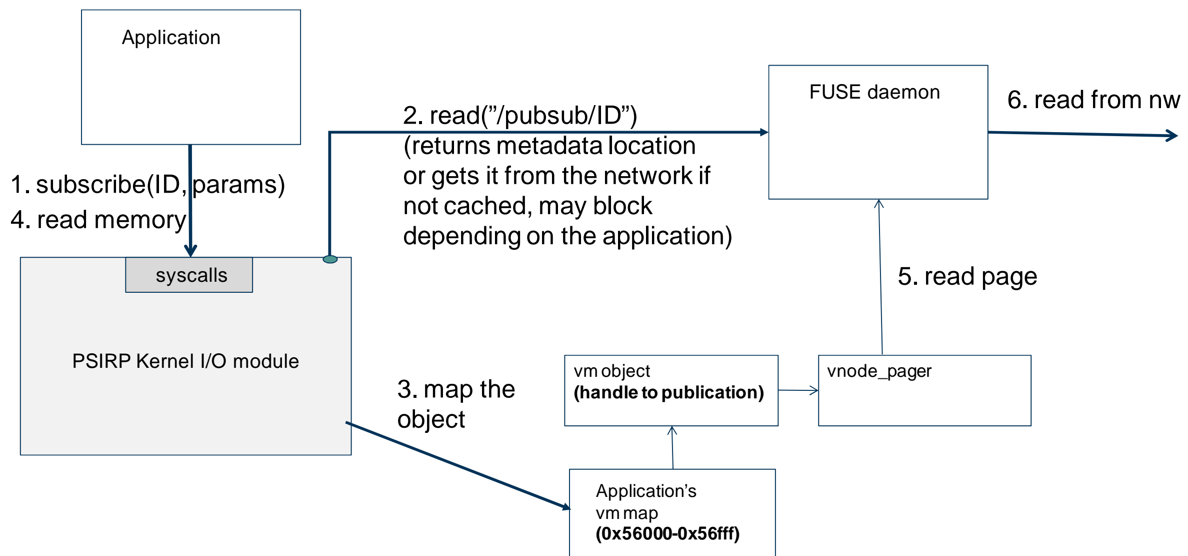

Figure 14 - Subscribing to a Publication

4.3.4 Network I/O

The network I/O module is part of the PSIRP daemon. The network I/O takes care of capturing frames from the link and transmitting frames to the link. The PSIRP daemon is implemented using multiple threads. The main thread is listening to events from the FUSE library. In addition, the capturing and transmission procedures are implemented in own threads using queues per direction. The frame error detection is strongly bound to the security approaches taken in this project. Basically, the error detection can be based on different kinds of hash algorithms and public-key technology. Error detection may result in re-subscribing the publication or part of the publication, depending on the application.

4.3.5 NetFPGA

NetFPGA [23] is an open platform that allows for development and evaluation of highly customized network routers, switches and interfaces. It enables customization up to the MAC layer, since only the Ethernet physical layer (PHY) is fixed in hardware. The platform includes a PCI card and sources (with BSD license) for reference software (Ethernet NIC, router and switch) that can be used as a starting point for the development. PHY layer, some memory circuits, connectors (e.g. SATA, PCI, Ethernet) and FPGA circuit are integrated on the PCI card. The Ethernet PHY supports four 1 gigabit ports. The presented features make the platform suitable for the needs of the PSIRP project to implement e.g. fast forwarding prototype with required authentication functions.

The most important feature of the NetFPGA is that the FPGA circuit can be used to implement some network functionality at the hardware level. The FPGA is configured by using a hardware description language, providing the advantage that it can be used to implement truly concurrent functionality.

It is planned to modify the current router implementation so that various custom routing functions can be tested on hardware level. The goal is to find out what can be and what should be implemented in hardware and to find out what is required from the hardware to do. Also depending on the implemented routing functions it might be interesting to make comparative performance measurements between the current TCP/IP based implementation and these customized implementations on the same platform.

NetFPGA is used in various universities for research and educational purposes and therefore it is likely that research results based on this platform will be published by other parties. That would allow reliable performance comparison between our implementation and others with the same platform. At the first stage, the forwarding function of the PSIRP prototype will be implemented on the card. This is done by gradually modifying either existing router or switch implementation. At this stage, the plan is to replace parts that are used for IP based forwarding with our own implementation that is based on forwarding labels. In practice the implementation derives a forwarding label from the packet and makes a decision on which port the packet is forwarded based on the label.

The part that makes the decision about forwarding will be implemented so that it would be easy to customize. After that it is quite easy to test and refine different methods for forwarding, those can include for example lookup table based forwarding or z-filters.

4.4 PSIRP API and Module Interfaces

The following subsections define the required interfaces for the application development, including both the user applications as well as helper applications. The helper applications have different kinds of needs than the normal user applications, thus a separate interface with some additional functionality is created for them. In addition, there are internal interfaces that are required to support the communication between different modules.

4.4.1 Libpsirp

Libpsirp is a shared C library that hides the system call specific details from applications and can also perform other helper functions. The main purpose at the beginning is that it uses system calls provided by the PSIRP I/O kernel module to implement the API and the Advanced API.

The Simplified Wrapper and Interface Generator (SWIG) [7] are used to generate the corresponding Python API (Network Attachment) and Ruby API (Rendezvous Point) from the libpsirp C code. Parameterizations for the platform specific API calls are supposed to be the same as in the C library.

4.4.2 API

The API provides the following functions for applications:

- create
- publish
- subscribe
- unsubscribe
- release

Create is similar to malloc() or mmap() in the standard C library. It simply allocates the requested amount of memory to the calling process and returns the address of the allocated memory area. This memory area can be later used to publish data. This function is specific to the prototype implementation due to the memory management and process structure properties of the FreeBSD operating system, not because of the requirements from the conceptual PSIRP architecture. On some other operating system this function may be unnecessary and existing memory allocation library calls can be used.

Publish creates a snapshot of a memory area of given length from the given address. It can take parameters such as the Scope ID and the Rendezvous ID, if the application desires to specify those itself. If these are left unspecified, the PSIRP Daemon will automatically allocate the IDs. In addition, parameters in the publish system call may include, but are not limited to, e.g. the lifetime and the type of the publication. The application gets the IDs allocated to the publication after a successful function call.

Subscribe maps a publication to a memory location. The subscribe call requires a Rendezvous ID of the publication and optionally also the Scope ID. If the Scope ID is left unspecified, it defaults to the local scope ("localhost"). Subscribe call may be blocking or non-blocking. In the case of blocking, there is a fixed timeout. In the case of non-blocking, the subscribe call returns and only accessing the publication data will cause blocking when the missing parts are fetched from the network or from the cache. As the Rendezvous client module is also using the same API, there needs to be a special flag to prevent "rendezvous loops".

Unsubscribe unmaps the publication from the calling process, and removes the subscription from the subscription table and forwarding table. Unsubscribe requires a Rendezvous ID and optionally also the Scope ID.

Release is the counterpart of create. The application may call this when it doesn't want the publication to be mapped to its memory space any more. This does not affect copies of publications in other nodes or in other processes in the same node. Release takes a memory address as the parameter, as the calling process may have (for some reason) mapped the same publication in several memory locations by calling subscribe with the same Rendezvous ID several times.

4.4.3 Advanced API

The helper applications (Network Attachment, Rendezvous Point, etc.) use the same API as normal user applications, but they need extended functionality, which is present in the "Advanced API". One example of such functionality is the ability to configure the forwarding tables (like co-located Rendezvous Point and forwarding node in the first prototype). Another one is the system call needed for the PSIRP Daemon to register the FUSE file system mount point into the PSIRP I/O kernel module during the start-up.

On top of the system call library API, the Simplified Wrapper and Interface Generator (SWIG) is used to generate corresponding Python API (Network Attachment) and Ruby API for the system call interface.

4.4.4 Module interfaces

In the first prototype, we try to use the pubsub-api in as many places as possible. However, there do exist extra interface besides our pubsub-api:

- The Ruby - MySQL interface is required for the database implementation in the rendezvous point implementation.
- The interface between libpsirp and the I/O kernel module is used for transmitting psirp API related system calls to the kernel module. A new system call number is defined for the interface in the FreeBSD kernel.
- The kernel module communicates with the daemon using the virtual file system (VFS) calls and vnode-pager with the daemon. In addition, the daemon may use the libpsirp API to store the received publications locally at the node.
- The interface between PSIRP daemon and Ethernet is implemented using 'libpcap' [11] and 'libnet' [12] library routines.

5 Abbreviations

AId	Application Identifier
ASIC	Application Specific Integrated Circuit
DoS	Denial of Service
DTN	Delay Tolerant Network
ECC	Elliptic Curve Cryptography
FId	Forwarding Identifier
FPGA	Field Programmable Gate Arrays
FUSE	File system in User space
PSIRP	Publish Subscribe Internet Routing Paradigm
PLA	Packet Level Authentication
LoLI	Lower Layer implementation
NREN	National Research and Education Network
RId	Rendezvous identifier
RP	Rendezvous Point
SId	Scope Identifier
SWIG	Simplified Wrapper and Interface Generator
TTP	Trusted Third Party
UpLI	Upper Layer implementation
VFS	Virtual file system

6 References

[1] PSIRP: D2.2: “Conceptual Architecture”, http://www.psirp.org/files/Deliverables/FP7-INFSO-ICT-216173-PSIRP-D2.2_ConceptualArchitecture_v1.1.pdf

[2] PlanetLab, http://www.planet-lab.org

[3] Onelab3: http://www.one-lap-2.org

[4] PSIRP: D3.1, “Implementation Plan”, http://www.psirp.org/files/Deliverables/FP7-INFSO-ICT-216173-PSIRP-D3%201_PrototypePlan.pdf

[5] File system in User space, http://fuse.sourceforge.net/

[6] Marshall Kirk McKusick and George V. Neville-Neil. “The Design and Implementation of the FreeBSD Operating System”. 1994.

[7] Simplified Wrapper and Interface Generator, http://www.swig.org/

[8] Per Cederqvist et al. “The CVS manual — Version Management with CVS”. ISBN: 0-9541617-1-8. 2006.

[9] Subversion, http://subversion.tigris.org/

[10] ZFS, http://opensolaris.org/os/community/zfs/

[11] libpcap, http://www.tcpdump.org/

[12] libnet, http://www.packetfactory.net/libnet/

[13] Altera, HardCopy Structured ASICs: technology for business [online], 2008, available from: http://www.altera.com/products/devices/hardcopy/hrd-index.html

[14] B. Brumley and K. Nyberg, “Differential properties of elliptic curves and blind signatures,” In Proc. of Information Security, 10th International Conference - ISC '07, volume 4779 of Lecture Notes in Computer Science, pp. 376-389, Springer-Verlag, 2007.

[15] C. Candolin, “Securing military decision making in a network-centric environment,” doctoral dissertation, Helsinki University of Technology, Finland, 2005.

[16] J. Forsten, K. Järvinen, and J. Skyttä, Packet level authentication: Hardware subtask final report, Technical report [online], 2008, available from: http://www.tcs.hut.fi/Software/PLA/new/doc/PLA_HW_final_report.pdf [Accessed 25th July 2008].

[17] K. Järvinen, J. Forsten, and J. Skyttä, “FPGA design of self-certified signature verification on Koblitz curves,” In Proc. of the Workshop on Cryptographic Hardware and Embedded Systems, CHES 2007, Vienna, Austria, September, 2007, pp. 256-271, Springer-Verlag LNCS 4727.

[18] Kobliz, “Elliptic Curve Cryptosystems,” Mathematics of computation, Volume 48, pp. 203-209, 1987.

[19] D. Lagutin, “Redesigning Internet - The Packet Level Authentication architecture,” licentiate's thesis, Helsinki University of Technology, Finland, June 2008.

[20] V. Miller, “Use of elliptic curves in cryptography,” In Proc. of the Advances of Cryptology – Crypto '85, Santa Barbara, USA, August 1985.

[21] NS-3, http://www.nsnam.org

[22] PSIRP: D4.1, “Preliminary Validation Plan and Selection of Tools”, http://www.psirp.org/files/Deliverables/FP7-INFSO-ICT-216173-PSIRP-D4.1_ValidationPlan-1.pdf

[23] NetFPGA, http://www.netfpga.org