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

DELIVERABLE D3.5

Final Description of the Implementation

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	33 months, until 30.9.2010
Document Title	Final Description of the implementation
Date of preparation	6.5.2010
Authors	Petri Jokela (LMF), Janne Tuononen (NSNF), Jimmy Kjällman (LMF), Borislava Gajic (RWTH), George Xylomenos (AUEB), Konstantinos Katsaros (AUEB), Jukka Ylitalo (LMF), Dmitrij Lagutin (AALTO-HIIT), Walter Wong (LMF/AALTO-HIIT), Kaloyan Petrov (IPP-BAS), Vladimir Dimitrov (IPP-BAS), Ventzislav Koptchev (IPP-BAS), Dirk Trossen (UCAM)
Responsible of the deliverable	Petri Jokela (LMF) Phone: +358 9 299 2413 Fax: +358 9 299 3535 Email: petri.jokela@ericsson.com
Reviewed by	Arto Karila
Target Dissemination Level	Public
Status of the Document	Completed
Version	1.0
Document location	http://www.psirp.org/deliverables/
Project web site	http://www.psirp.org/

Table of Contents

1 Introduction 3
1.1 Abbreviations 3

2 PSIRP Prototype 4
2.1 Blackhawk 4
2.2 Rendezvous Node 14
2.3 Topology Management 16
2.4 Application 20
2.5 Language Bindings 22
2.6 Example of Current Integration Plans 22

3 Open Source Release 23
3.1 Blackhawk 23
3.2 NetFPGA forwarding implementation 23
3.3 Packet Level Authentication (PLA) 24
3.4 Documentation 24

4 PSIRP Testbed 24
4.1 Testbed Set-up 24
4.2 Virtual Test Bed Set-up 25
4.3 Applications 26

5 Evaluation of the Work 26
5.1 Rendezvous Node 27
5.2 Blackhawk 27
5.3 Topology Management 27

6 Conclusions 27
7 References 28

1 Introduction

It has been the declared goal of PSIRP to complement the architecture and technology development within the project with a clear implementation effort that realises and tests our developed technologies in a realistic setting, i.e., with available technology. The intertwined nature of such implementation with the development of our architecture and protocol technologies has been crucial for the project since the lessons learned in implementation have been directly fed into the progress on the architectural and protocol levels.

In this document we describe the current status of the PSIRP prototype implementation. The implementation itself has been done in separate modules, e.g. Rendezvous, Topology, host internal blackboard based publication management and packet forwarding. In addition, security features in the form of Packet Level Authentication (PLA) have been implemented, as well as various test applications that utilize the PSIRP prototype platform. We describe these different modules through this document.

The final target of the project is to produce an integrated PSIRP prototype, providing an implementation of the designed PSIRP architecture. This integrated prototype is to be show-cased as well as tested within a growing multi-site testbed. This document describes the setup and status of the testbed efforts. We also outline the applications that are envisioned for this testbed.

Last but not least, this document provides a brief overview of the level of integration between the different components. This integration work is currently ongoing as specified in deliverable D3.4 [3].

1.1 Abbreviations

LId – Link Identifier

RId – Rendezvous Identifier

SId – Scope Identifier

vRId – Version-RId

pRId – Page-RId

PIT – Publication Index Table

Pubi – Publication Index

RVS – Rendezvous

TCC – Traffic and Congestion Control

SMC – State Machine Compiler

ECN – Explicit Congestion Notification

RN – Rendezvous Node

RP – Rendezvous Point

LSA – Link State Advertisement

TM – Topology Manager

PLA – Packet Level Authentication

2 PSIRP Prototype

The PSIRP prototype development has followed the generic design as defined in the early phase of the project. The core of the implementation is the so-called blackboard based host implementation, which handles publication management inside the host. The implementation of the blackboard system is called “Blackhawk” and it is presented in Section 2.1. The Blackhawk implementation also defines an Application Programming Interface (API) that all applications, including the PSIRP architecture helper applications, use for publishing and subscribing to publications. The core functions of this API are *publish*() and *subscribe*().

The two fundamental helper functions, defined in the PSIRP architecture, namely **Topology management** and **Rendezvous**, are implemented on top of the Blackhawk core. The Rendezvous module, presented in Section 2.2, handles the publication management by maintaining the publication metadata information and matching subscriptions to publications. The module also handles the publication management by maintaining the publication metadata information, and matching subscriptions to publications. The Topology management module, presented in Section 2.3, is responsible for maintaining and exchanging the network’s topology information on an intra-domain level. It also creates paths and related Forwarding Identifiers for subscribed publications based on the requests from the Rendezvous system.

The reader is assumed to be familiar with the terms and structure of the PSIRP system architecture as described in the conceptual architecture [1], the contents of the implementation progress report D3.3 [2] and the integration and demonstration plan D3.4 [3].

2.1 Blackhawk

Figure 21 depicts the architecture of the Blackhawk prototype on a high level. The architecture itself has not changed since D3.3 [2] and D3.4 [3], as the focus of the prototype development work has been on increasing stability, adding features needed for integration of different components and API usability improvements.

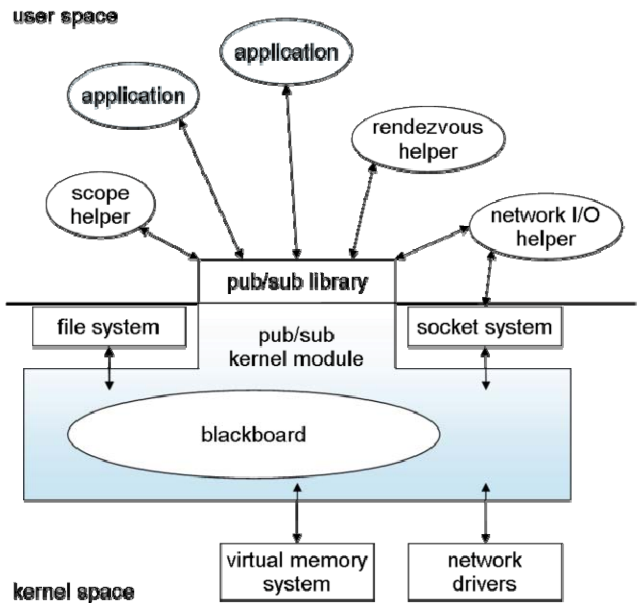


Figure 1 Blackhawk architecture

2.1.1 The pub/sub kernel module

The blackboard is the local repository for publications residing in the memory of each computer running Blackhawk. Both the content (data) and metadata of publications is stored in the blackboard. Next, we briefly describe the design of the blackboard and how it is integrated with different parts of the FreeBSD operating system.

Virtual Memory System Integration

The blackboard resides in the operating system's kernel space as a loadable module and is integrated with the virtual memory system. Thus, the content of a publication can be accessed in a very natural and efficient manner: the data are mapped into an application's memory space so that the application can get a pointer to that memory area. This memory area can be accessed just like any other allocated memory.

In FreeBSD's virtual memory system [8], data is (virtually) stored in pages with a default size of 4096 bytes. Each of the pages belongs to a virtual memory object and has a specific index within that object. VM object shadowing is employed for the sharing of pages between objects. In our blackboard implementation, we keep two VM objects for each publication: one for metadata and one for data.

The metadata object size is currently one page per publication. It acts as a placeholder for the RId and size of the publication, as well as other essential information. Applications in user space get indirect read-only access to this information through accessors in the pub/sub library (i.e., *libpsirp*, described later in this document).

The data object, on the other hand, points to memory pages that hold the actual content of the corresponding publication. This object (or only a part of it) is mapped to applications using the copy-on-write concept. This means that several applications can be given read-write access to the same memory if they subscribe to the same publication. If they modify the subscribed data, copies will be made of the changed pages, while the unmodified ones remain shared. In order to make those changes visible to other node-local subscribers, or the original creator, the data needs to be re-published. This results in a new version of the publication.

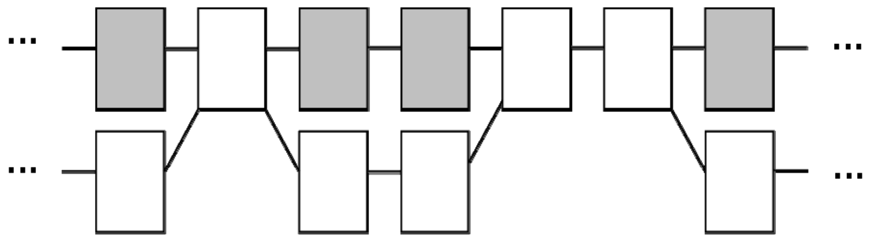


Figure 2 Publication versions can share pages

File and Event System Integration

The blackboard is integrated with the node's file system which provides multiple advantages. Firstly, each publication can be accessed, currently only with read access, through a normal-looking file on the computer. This provides us with an alternative, legacy-compatible API (Figure 3) for the blackboard system. Secondly, it is easier to support kernel event (*kevent*) queues (*kqueues*) for generating notifications when new versions of publications are created. Thirdly, it enables demand paging over the network, although this feature is still unimplemented in Blackhawk. However, the concept has been presented already in earlier deliverable D3.2 [9] and has been used in one of the early demonstrators [3]. Having our own file system type (*psfs*) in the kernel is more efficient than using one in user space (e.g. via FUSE). It also simplifies the required interactions between the file system part and the rest of the blackboard components.

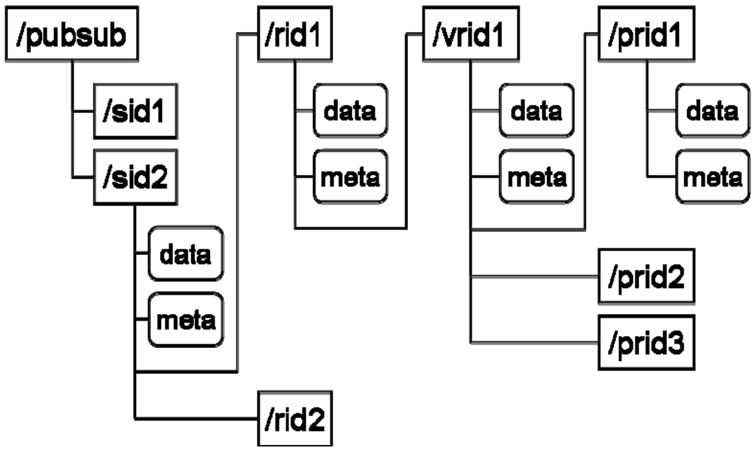


Figure 3 Pub/Sub file system view

In user space, applications get an open file descriptor when they subscribe to a publication, or publish a newly created one. Using that descriptor, the application can register to *kevents*. This may happen implicitly, depending on the API functions used. When a publication is updated, a special event notification is dispatched from the kernel via a *vnode* to the subscribing entities. Here we have created our own *vnode* type, called *pnode*, and use a special type of *knote*, NOTE_PUBLISH, that the subscribers have registered to listen at.

Data Object Model

Next, we have a look at the hierarchy of publication objects stored in the blackboard. Blackhawk currently has four different publication types: Scope, Data (or concept), Version, and Page publications, as shown in Figure 4.

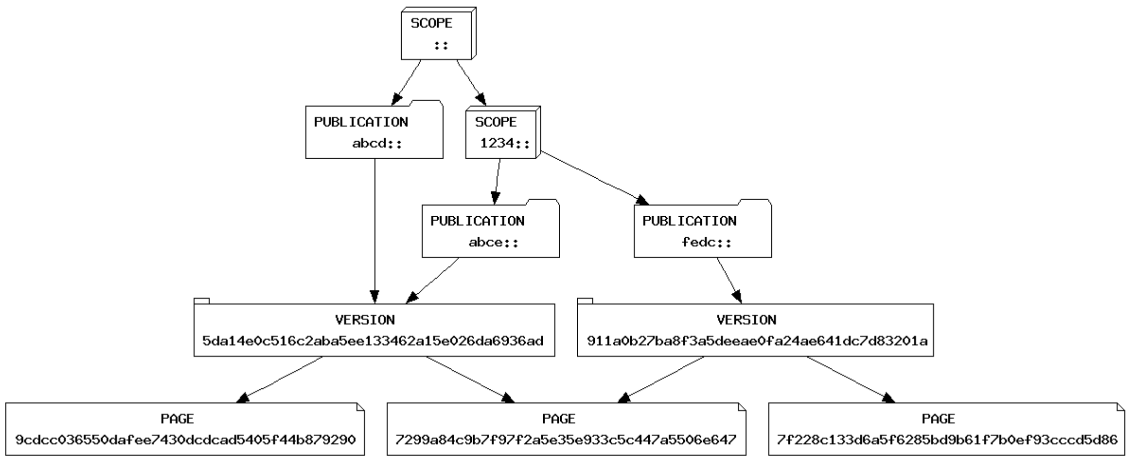

SCOPE
::
PUBLICATION
abcd::
SCOPE
1234::
PUBLICATION
abce::
PUBLICATION
fedc::
VERSION
5da14e0c516c2aba5ee133462a15e026da6936ad
VERSION
911a0b27ba8f3a5deeae0fa24ae641dc7d83201a
PAGE
9cdcc036550dafee7430dcdcad5405f44b879290
PAGE
7299a84c9b7f97f2a5e35e933c5c447a5506e647
PAGE
7f228c133d6a5f6285bd9b61f7b0ef93cccd5d86

Figure 4 Publication hierarchy generated from a real system

Conceptually, scope publications are containers for data publications (see Figure 4). Scopes are, proverbially, identified by SIds. On the implementation level they contain a list of RIds in their data object. New RIds are added to scopes by the scope daemon, which gets notified about all publish-operations in the node. This helper application simply subscribes to a scope that may have been updated, modifies its contents if needed, and re-publishes it.

A data publication is identified using a RId, and it refers to some data item whose content does not need to be immutable.

Since a data publication can have several versions, the publication hierarchy includes version publications with immutable content. The identifier of a version publication is a content-dependent version-RId (vRId), implemented as the root hash of a (skewed) Merkle hash tree of the data.

Finally, we find page publications on the lowest level in this hierarchy. They correspond directly to the memory pages in the data objects, and are identified by page-RIds (pRIds) that are formed by hashing page contents, e.g., with SHA-1. The content of a page publication is also immutable.

Node-local Rendezvous

The blackboard needs to provide node-local, in-kernel rendezvous. This means that if somebody has published data under a SId/RId pair, subscribers should get that data when they subscribe to the same pair. In addition, it is also possible to subscribe to specific data with vRIds and pRIds. For this purpose, the blackboard contains a lookup table called *Publication Index Table* (PIT). The publication identifiers, i.e., SId, Rid, vRId, and pRId, are used as keys for accessing that hash table.

The PIT can consist of multiple, potentially swappable, VM pages. Each page has a number of buckets that contain a couple of PIT entries. For optimization reasons, entries on sub-pages can be moved "upwards" when they are accessed. Each PIT entry, if filled, contains an Id, and a mapping from that Id to a *Publication Index* (pubi). A pubi is a data structure that holds additional metadata needed only in the kernel, including e.g., pointers to the data and metadata VM objects as well as to the vnode for the publication.

To complete the in-kernel operations, we need to describe the actual lookups when subscribing to a data publication in a specific scope, or when re-publishing something and implicitly checking for already existing versions. As described above, scopes have a list of RIds in their data object. Similarly, data publications have a list of version-RIds and version publications have a list of page-RIds in their metadata object. To find a data publication in a scope, we first look up the scope in the PIT by the SId, get the data page from the pubi, and check whether the RId is in that data. If the RId is found, we perform a second lookup using the RId as the key. This time we can get the metadata and data objects and map them for the user space application subscribing to the publication. If we subscribe using only a SId and a RId, the latest version of the publication is returned.

The procedure is the same for finding a specific version under a data publication, except that the data publication's RId would be used as the SId, and its metadata object would be used as the "scope" in the first lookup. Similarly, the metadata of a version can be used as a scope when subscribing to a specific page. However, these steps are not required in a normal subscription with a SId/RId pair.

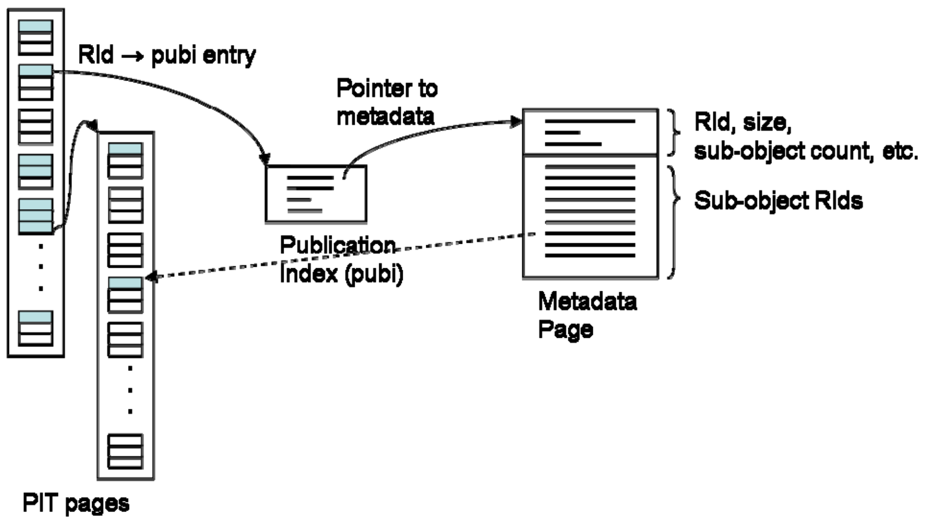


Figure 5 PIT, pubi, and metadata

Figure 5 shows the PIT on the left-hand side. As one hash bucket is full, it further points to a sub-page. A PIT entry points to a publication index in the middle, which in turn points to a metadata page of, e.g. a data publication or a version on the left-hand side. The metadata object contains RIds that can be used in further PIT lookups.

2.1.2 Native pub/sub API

The kernel module has a system call interface towards the user space. In practice, this interface is not supposed to be used as an API in normal applications which should use the libpsirp API library, which provides instead all the essential pub/sub functions. Applications in user space use this library to communicate with each other through the blackboard.

New publications can be created with the *psirp_create(len, *pub)* function, where *len* refers to the requested size of the content. It returns a handle (*pub*), which is a pointer to a data structure containing a pointer to the data, length, Ids, and file descriptor of the publication, among other things. However, the structure is not accessed directly, but through *getter* functions.

Publishing the created publications is done with *psirp_publish(sid, rid, pub)* where *sid* and *rid* are identifiers, and pub is the handle. *psirp_subscribe(sid, rid, *pub)* is used for subscribing to a publication. By default it returns the newest local version.

The API also contains other variants of the functions mentioned above, for instance for synchronous (blocking) subscribing, as well as auxiliary functions needed, e.g., for operations on identifiers. The API reference can be found at http://code.psirp.org.

The kevent/kqueue mechanism in FreeBSD can be used for registering and listening to publication update events. This is similar to using, e.g., the select() call to wait for file or network events.

2.1.3 Helper applications for pub/sub networking

The architecture defines helper applications (in user space) that take care of certain special functions in the prototype and correspond to some of the components in the PSIRP component wheel. The core prototype includes two of these helpers at the moment: the network I/O daemon and the local rendezvous daemon.

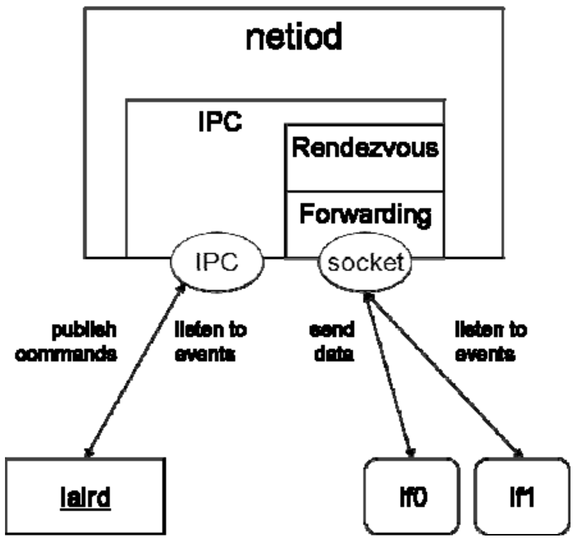


Figure 6 Internal architecture of the network I/O daemon

Network I/O Daemon

The network I/O daemon (*netiod*, Figure 6) sends and receives packets on the link layer through sockets. Currently it uses broadcast Ethernet frames, but (overlay) networking over UDP has also been planned. Secondly, this helper implements packet forwarding with zFilters. It implements the basic forwarding functionality, i.e., matching FIds to the Link IDs (LIds) of outgoing interfaces.

The LIds used by netiod are statically configured in a configuration file, /etc/netiod.conf. One of those LIds should "point" to the node itself, thus providing a “virtual interface” which, if present in the incoming zFilter, actually takes the packet to node internal processing instead of just forwarding it to others or possibly dropping it.

Since normal Ethernet links can only send and receive ~1500 bytes per frame, a 4096-byte page with headers does not fit into a single packet. For this reason, the network I/O daemon needs to handle publication fragmentation and reassembly. The complete publications that it has received from the network, it publishes in the local node's blackboard.

When compared to the v0.2 release, Blackhawk v0.3 alpha features rudimentary multicast support: when packets are put to outgoing packets queue, the existing queued chunks are scanned. If identical chunks are found, their FId is updated, and the new packet does not need to be added. The number of simultaneous pending data subscriptions has also been increased.

For the prototype, the following packet format has been specified: The packet begins with a forwarding header that includes a FId, TTL value, and other forwarding-related information. This header is followed by a rendezvous header, containing the SId, RId, vRId, and sequence number of the publication / data chunk. If security features are enabled, the PLA header, also called the security header, follows the rendezvous header. If the packet is used for signalling published metadata, a subscription to metadata, or a subscription to data, a placeholder for metadata is included. This element contains a SId, RId, vRId, the return-FId to be used for forwarding packets back from the receiver to the sender, the data length and the signal type. Alternatively, if the packet contains only a chunk of a publication’s content, the last part of the packet is just generic payload (e.g. 1024 bytes).

Local Rendezvous Daemon

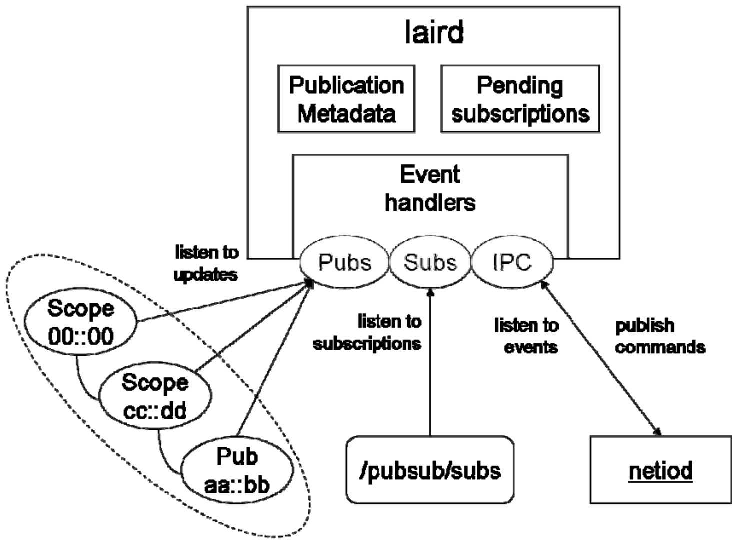


Figure 7 Internal architecture of the local rendezvous daemon

The local network rendezvous helper (*laird*, Figure 7) deals with pub/sub operations between nodes in a LAN. In other words, it enables publish/subscribe operations between nodes in the local network.

This daemon listens to all subscribe-events that occur in the local node. It also listens to updates to all local publications: initially it monitors "Scope 0", the so-called *node-local root scope* in Blackhawk, and afterwards it recursively registers to listen to all sub-scopes and all publications in those scopes.

The local network rendezvous daemon issues *publish* and *subscribe* commands to the network when these operations occur locally. When a new publication is created or an old one is updated, metadata is sent to a local rendezvous node that caches the metadata. Each node knows a default FId pointing towards one default rendezvous node, and they use that FId for sending out any information if no other destination has been explicitly defined.

When the rendezvous node receives a subscription and is aware of a data source for that publication, it can relay the data subscription to that source. When the subscription then reaches the source, the local rendezvous daemon in that node tells the network I/O daemon to send the publication data to the subscriber using a collected reverse-path FId (found in the metadata).

When compared to Blackhawk v0.2, the local rendezvous system has as a new task to maintain state for pending subscriptions. In the current version of Blackhawk, v0.3 alpha at the time of writing, subscriptions can be persistent. This means that a node can request to get all future versions, possibly the current one included, from the rendezvous node. When new versions get published, and the local rendezvous daemon learns about them, the node receives notifications (i.e., metadata) about those versions and can fetch the corresponding data.

In addition, support for subscribe-before-publish has been added. This means that the local rendezvous node maintains state for pending subscriptions from other nodes until it finds out a matching publication.

Helper-to-helper IPC

Blackboard-based IPC is used as the sole communication mechanism between the network I/O and local rendezvous helpers within a node:

- Both helpers have two common initial RIds in Scope 0 (the node-local root scope)
- Both helpers subscribe and listen to their own "IPC-RId", and publishe updates with the other entity's current RId
- These RIds can be updated when, e.g., the version number limit for one RId is coming close
- A common IPC publication format is known by both parties

Using this scheme, *laird* publishes information about publications and subscriptions to *netiod*, and *netiod* publishes received information (metadata) to *laird*.

Network Signalling Model

Figure 8 illustrates the local rendezvous signalling in a simple pub/sub scenario. First, a publisher publishes the metadata (identifiers, size, path to source) of a new publication to the local rendezvous node. When a subscriber wants to get the publication, it first subscribes to the metadata, since it needs to know at least the publication's version-RId and length. The metadata can be sent from the RVS node's cache. Finally, the subscriber subscribes to the actual data and the RVS node forwards this request to the publisher who finally sends the publication in chunks to the subscriber.

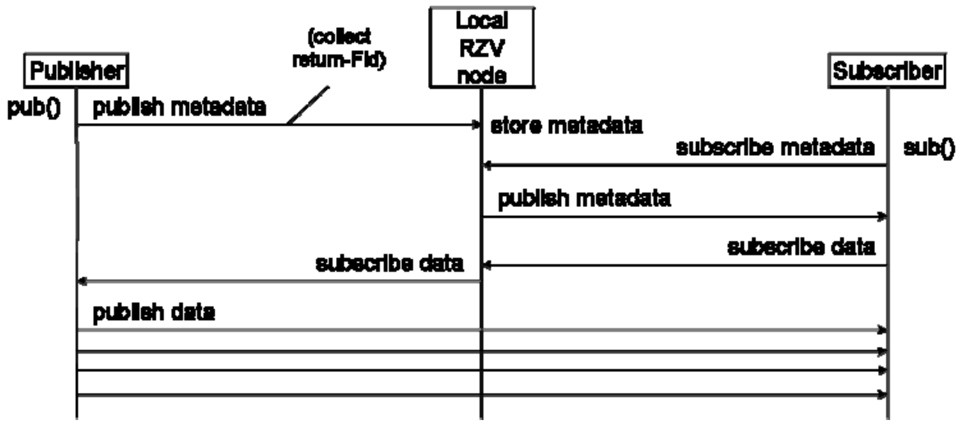


Figure 8 Message Exchange Example

2.1.4 Packet Level Authentication

In [2], the main features of the PLA implementation are described. Since then, the implementation has been integrated with the Blackhawk v0.3 implementation.

2.1.5 Transport and Caching Publication chunk requests

In addition to the existing 'subdata' packet type used in Blackhawk by subscribers to request all publication packets, a 'subdatachunk' request is implemented allowing subscribers to request specific publication chunks. The mechanism is used by the 'Lost chunk recovery', 'Data caching' and 'Subscriber controlled Traffic and Congestion Control (TCC)' schemes, as described below. The requested publication chunks are specified by their sequence numbers and are included in the 'subdatachunk' packet payload as an array with the number of requested packets as the first element, followed by the sequence numbers of the packets.

Lost chunk recovery

A 'flow' sequence number is added to the packet header where 'flow' stands for a given 'subdatachunk' request. In general a 'subdatachunk' packet requests a subset of all publication chunks, which makes the 'flow' sequence number different from the publication sequence number. The 'flow' sequence numbers are sequential, with the last sent packet of a given request having number zero.

Lost chunks are detected in two ways:

- When the last packet of a given request (recognized by a zero 'flow' sequence number) is received, any missing packets from that request are considered lost and the subscriber sends a 'subdatachunk' request containing only those.
- Cases when the last packet of a request is lost are handled by a program running in a separate thread. For each active publication the program keeps track of the system time the last packet was received. The program checks all active publications at regular time intervals, the default value being one second. If for an active publication it is determined that there are missing requested packets and the time elapsed since the last packet was received (or the active publication created) is greater then a predetermined constant value (one second by default), all requested packets which are missing are considered lost and a 'subdatachunk' request containing only those is sent. If necessary the process is repeated up to 5 times by default, after which if there are still missing packets of the given request, no further attempts are made and the active publication is freed.

Data caching

The caching mechanism consists of a single cache store per node servicing all network interfaces with the following configurable parameters:

- Cache Check Interval - the time interval at which the cache is checked for old entries. The default value is 60 seconds.
- Cache Timeout - the time interval since a cache entry was entered or last used after which the entry is considered old and removed. The default value is 5 minutes.
- Cache Size - the size of the cache in number of packets. The default value is 1000. If the cache size is exceeded new entries are not stored until space is available.

The cache stores clones of all incoming data packets. Duplicate entries are not stored. If a 'subdatachunk' packet is received by a node the cache is checked for the requested content. Any found packets are sent back to the requesting node and if necessary a modified 'subdatachunk' request containing only the requested chunks which were not found in the cache is forwarded towards the publisher. At this point, if a 'subdata' request is received by a node the cache is not checked. The reason for this is that even though the caching mechanism can find and return all available chunks for a given publication, the node has no information to determine if any chunks were not found in the cache and thus cannot forward a modified 'subdatachunk' request containing only the missing chunks. A workaround which is not implemented at this point could be for the node to send a 'negative subdatachunk' request containing the chunk numbers which are not needed. Once such a request is received by a node it would send all but the packets included in the request.

The data caching is implemented as a shared library written in C++. The C interface for use in Blackhawk consists of three functions:

- *void encache(pkt_ctx_t* pubdata)*, where the argument is a pre-parsed clone of the original 'pubdata' packet received by the node. The clone ownership is transferred to the cache. The clone is deleted by the cache cleanup thread when it is determined to be old.
- *pkt_ctx_t** getpubdata(pkt_ctx_t* subdatachunk, unsigned int* len).* The first argument is the 'subdatachunk' packet received by a node containing the sequence numbers of the requested chunks. The function returns an array containing all or a subset of the requested chunks and sets the second (output) argument to the length of the array.

- *cache_unlockItem(pkt_ctx_t* pubdata)*. Cache entries are returned in a locked state which prevents the cache cleanup thread from removing them. This function is needed to release the locks after the entries have been processed.

TCC - sender controlled

In this TCC version the rate towards the congested area is controlled at the nodes that send publication data. For that purpose, all nodes have a separate output queue for each traffic flow. The queues are serviced in a round-robin fashion and each queue rate is controlled by a token bucket filter. A node communicates a congestion condition to the previous node in a path via a special choke packet. A congestion condition is determined by either a packet loss or an output queue getting full. In addition, all nodes have a single input queue where all incoming packets wait to be processed with the goal of using this queue as an indicator of node overload. At this point, packet loss is only detected at the subscriber. When a node receives a choke packet, it reduces the sending rate of the queue corresponding to a classId placed in the choke packet's payload. Currently, the rate is dropped to a predetermined level after which it gradually raises with each sent packet if no other choke packets concerning this particular queue are received.

The module is implemented as a shared library written in C++ with a C interface:

- *void tbf_enqueueOut(const psirp_fid_t* fid, if_list_item_t* iface_out, const void* buf, size_t len, int flags)*; The first argument is used to classify traffic flows. It is followed by the output interface, the beginning of the packet, the packet length and the flags to use with the sendto syscall when the packet is dequeued.
- *unsigned int tbf_congestionCheck(const psirp_fid_t* fid, char* classId)*; checks if the output queue corresponding to the FID provided as the first argument is getting full. Returns a congestion status: 0 (no congestion), 2 (the queue is filled above 50%), 4 (the queue is full). Sets the second argument to the ID of the queue. The caller uses the classId to initialize the payload of the choke packet. A node that receives a choke packet uses the classId to modify the sending rate of the particular queue corresponding to the classId.
- *char* tbf_classify(const psirp_fid_t* fid)*; Returns the classId corresponding to the FID. Used for sending choke packets when the input queue is congested.
- *void tbf_reduceRate(char* classId)*; Reduces the sending rate of the corresponding queue.

TCC - subscriber controlled

In this TCC version, the rate towards the congested area is controlled indirectly by the subscriber through the number of chunks it requests at a time. The subscriber starts by requesting one, then two, four, eight and so on packets if each previous request was answered successfully - meaning all requested packets were received. If a packet loss is established the subscriber requests only the lost chunks and increases the number of chunks in its next request from this new starting point.

2.1.6 Transport Simulator in NS3

We are using SMC - The State Machine Compiler - to generate a transport state machine for ns-3 simulations. SMC uses the well-known 'State' design pattern for class generation. We believe that this is the first time that SMC is used together with ns-3 development. The usage of SMC improves code quality and speeds up the transport design cycles. In practice, we can avoid several coding bugs in the transport state machine and change the state machine design fast while sustaining short iteration cycles. The transport state machine is described in the SMC meta language that is compiled to C++/Python classes used in the ns-3 simulator. Figure 9 illustrates the transport state-machine class diagram and the dependency with SMC.

The transport state-machine is divided into two separate sub-state-machines, namely, the requestor and receiver parts. The two sub-state-machines share a common protocol context. The protocol context contains information about the number of unsent requests and window size. Both sub-state-machines update these values in the protocol context. We have applied the well-known 'Strategy'-pattern to implement different kinds of state-machines for the receiver. Currently, we have identified two different strategies for the receiver, namely, strategies for Explicit Congestion Notification (ECN) supported and unsupported forwarding paths. Our design allows the receiver to update dynamically its strategy. In addition developers can design and implement new alternative strategies easily. We have not applied the 'Strategy'-pattern to the requestor side due to the single state-machine implementation in that part. However, it is possible to use the same pattern as with the receiver also on the requestor side to simultaneously support alternative requestor strategies.

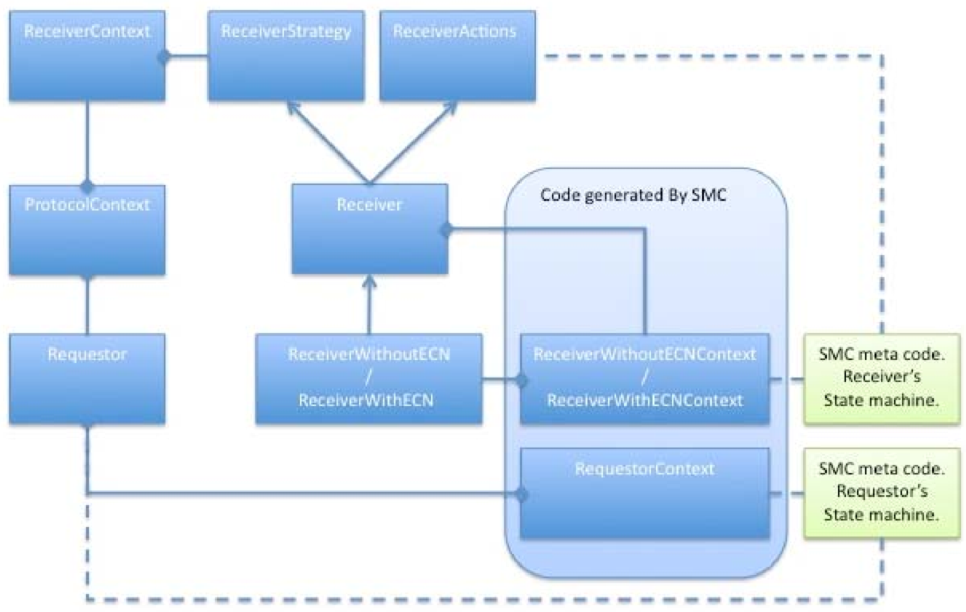

ReceiverContext
ReceiverStrategy
ReceiverActions
ProtocolContext
Receiver
Code generated By SMC
Requestor
ReceiverWithoutECN / ReceiverWithECN
ReceiverWithoutECNContext / ReceiverWithECNContext
SMC meta code. Receiver's State machine.
RequestorContext
SMC meta code. Requestor's State machine.

Figure 9 Transport state machine class diagram

2.2 Rendezvous Node

The Rendezvous Node (RN) prototype implementation is shortly summarized and the changes since D3.4 [3] are highlighted in this section. The local rendezvous helper, as part of the Blackhawk platform, is not in the scope of this text. Instead, this section covers the rendezvous node implementation for the global rendezvous system.

The rendezvous node implementation is, to some extent, an example realization of the PSIRP rendezvous concept defined in the conceptual architecture [1]. It supports establishing a rendezvous network (a group of rendezvous nodes in a policy compliant tree topology), and provides a basic rendezvous service in that network, including publish, subscribe and pre-established subscribe services. The rendezvous concept provides global scalability by interconnecting multiple rendezvous networks using, e.g., hierarchical Distributed Hash Tables (DHTs). However, the interconnection layer of the concept has not been implemented.

The expected final rendezvous implementation architecture is illustrated in Figure 10,

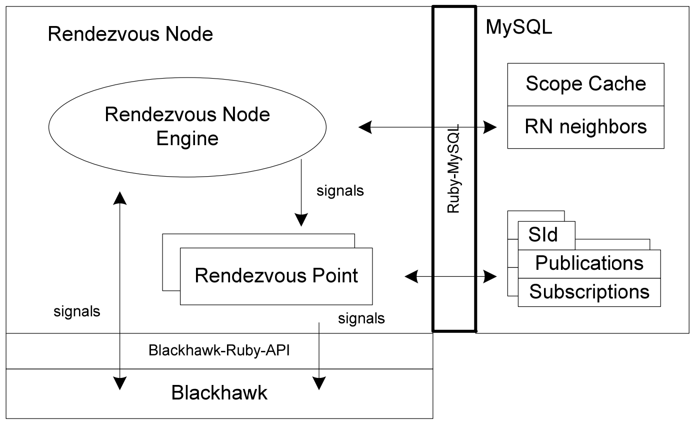


Figure 10 Rendezvous Implementation Architecture

The Rendezvous Node prototype has been implemented using Ruby1.8, while a Ruby C extension is used for rendezvous signal creation and parsing. MySQL data tables are used for storing both rendezvous node and rendezvous point data structures and these structures are accessed through the standard ruby-mysql interface. The rendezvous prototype initially used UDP for the signal transport, but as part of the successful finalization of the first phase rendezvous integration with the Blackhawk implementation, the UDP forwarding was replaced with the Blackhawk-Ruby-API. The underlying Blackhawk platform is now responsible for providing the required forwarding service for the Rendezvous Node prototype. The topology stub that was earlier simulating the operations of the Topology Management has also been removed from the architecture figure. As part of the integration work, the responsibility of handling that information was moved to the topology manager and the Blackhawk platform.

Since the submission of the Integration and Demonstration Plan [3] in September 2009, the focus of the implementation work has been on integration. During the first phase of the integration, in addition to the already mentioned communication module replacement, the prototype code was updated to re-establish support for the features it supported before the integration task. The first phase of integration was finished in October 2009 and after that the resources were used for the detailed planning of the second phase. In the beginning of 2010, the implementation work was started again with modifying the rendezvous signal formats. The previously used tag based character string packet formats were re-designed and replaced with a more conventional and network like, option based, implementation.

This change in signal format will provide easier signal synchronization in the second phase integration. The new design of the rendezvous signal format is a composition of a consecutive chain of predefined options, where the order and selection of options is rendezvous signal type specific. The new signal format definitions, option types, and the tools for creating and parsing rendezvous signals are implemented in a new Ruby C extension, which uses the native Ruby C extension API. The rest of the code is still implemented using Ruby.

We expect that some changes in the signal format may be needed during the second phase, and at this point we give only a couple of examples so that the reader can grasp the idea of the option based rendezvous signal formats. For instance, the new “publish” rendezvous signal format looks like this

| OPT_TYPE_RP (2 bytes) | LEN option (2 bytes) | RP Sid (32 bytes) | credential (32 bytes) |

| OPT_TYPE_PUB | LEN option | Sid (32 bytes) | Rid (32 bytes) |

| DATA_TYPE_NID | LEN option | Node ID (32 bytes) |

The first option contains rendezvous point information that is needed to enable rendezvous point creation and/or usage in the rendezvous node for the publish signal. The second option contains the RIds of the published publication. The third option is a new topology management related “Node ID” option. The NId is needed both from the publisher and the subscriber in the case when rendezvous for a publication happens and the rendezvous point requests an optimal path from the topology management server.

As a second example, we present the “subscribe” signal, which is actually similar to the publish signal, but without the rendezvous point option.

| OPT_TYPE_SUB | LEN option | Sid (32 bytes) | Rid (32 bytes) |

| DATA_TYPE_NID | LEN option | Node ID (32 bytes) |

The very latest development in the integration work has an impact to the generic prototype communication model and therefore also to the rendezvous signalling model. The currently evaluated new model indicates that the generic pub/sub API will be extended with such functionality that the previously used “transport publication” may not be needed anymore when two rendezvous entities are communicating. The new communication model will be described in detail in the forthcoming implementation and prototype technical report.

2.2.1 Relation to Blackhawk

The Rendezvous Node implementation relation with the Blackhawk implementation is three-fold. Firstly, the implementation uses the Ruby version of the pub/sub API, just like any other application, to send rendezvous signals to other rendezvous entities and the topology management server. Secondly, the implementation has a specific requirement (extended functionality): the pub/sub API has to be capable of providing the recorded "reverse zFilter" of the received rendezvous signal together with the payload to the RN process, if requested. Thirdly, in certain cases, the rendezvous point is responsible for updating the BlackHawk forwarding table with rendezvous ID forwarding ID mappings

The second requirement is a prerequisite for the third requirement. The two latter ones are not implemented yet and they are the main activities for the integration work. The preliminary plan for implementing reverse zFilter provisioning is to re-use the idea in the "socket options". The forwarding table updating will be done either using specific local pub/sub signalling or through a specific API.

2.2.2 Relation to Topology Management

The Rendezvous Point will communicate with the Topology Manager (TM) in a pure publish-subscribe way. When a rendezvous occurs between an existing publication and an incoming subscription, the rendezvous point will publish an update to the well-known publication that the TM server has subscribed to. This update contains the publication’s (Sid, Rid) pair and the Node IDs of the publisher and subscriber. Based on this publication the TM server can construct an optimal path between the publisher and subscriber and publish it to the publisher together with the publication information.

2.3 Topology Management

The Topology Management module (see Figure 12) performs topology discovery by gathering connectivity information and link-relevant information published from the helper modules of forwarding nodes in its domain of operation. The Topology Manager subscribes to three different publications to receive the required link state information from the network. The basic link information is published in LSA (Link State Advertisement) publications. As optional information, the LinkMSG publications contain more detailed information about the capabilities of the links in the network, and the Application capability publications contain possible application requirements in various nodes. Using this collected information, the Topology Manager can compute optimal forwarding paths for delivering publications through the network. Our current Python based implementation of Topology Management is compliant with the current Blackhawk v0.3 prototype.

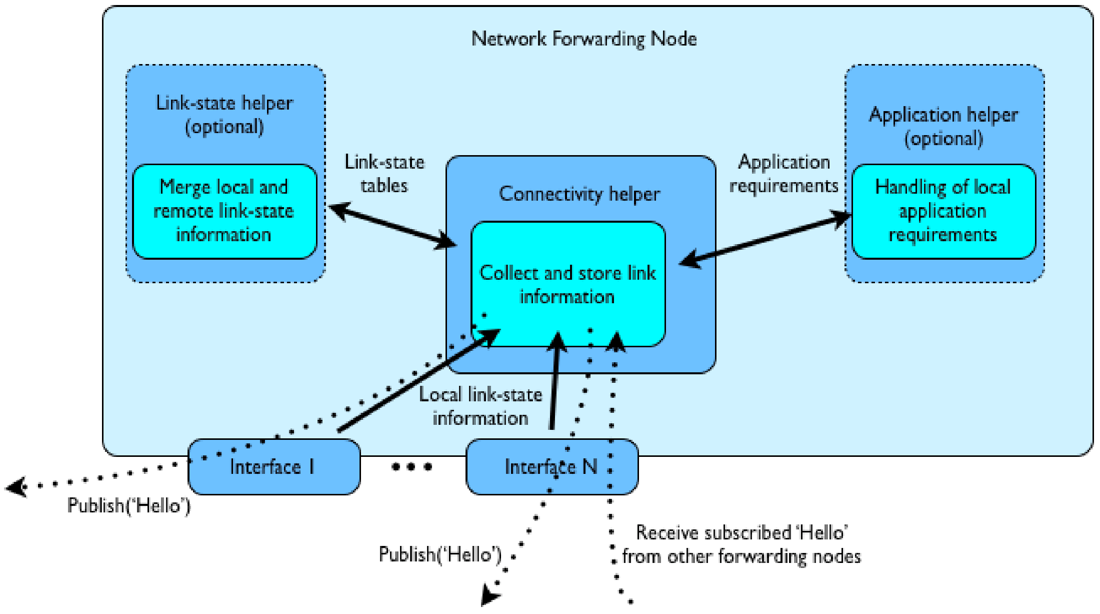

Network Forwarding Node
Link-state helper (optional)
Merge local and remote link-state information
Link-state tables
Connectivity helper
Collect and store link information
Application requirements
Application helper (optional)
Handling of local application requirements
Local link-state information
Interface I
Interface N
Publish('Hello')
Publish('Hello')
Receive subscribed 'Hello' from other forwarding nodes

Figure 11 Forwarding Node's Topology Modules

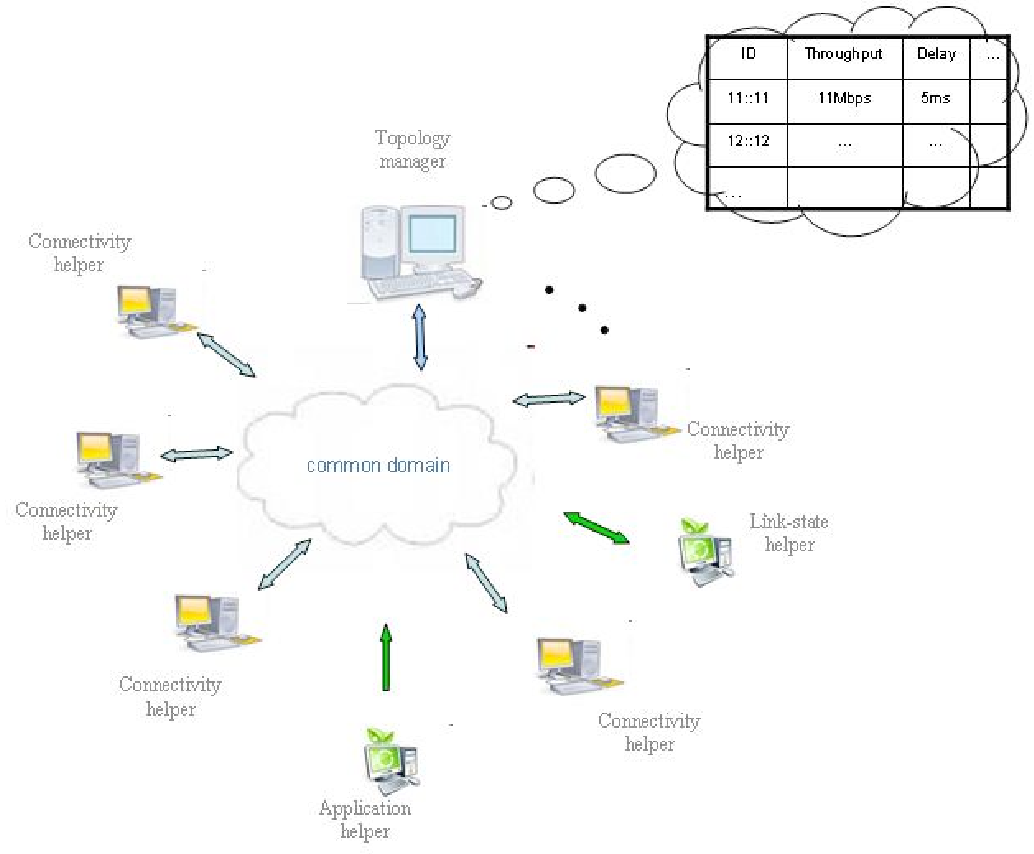
We have defined a set of helper modules that are required for feeding the Topology Manager with network state information (see Figure 11). These helpers include the *connectivity helper,* the *application helper* and the *link-state helper.* The main role of a connectivity helper is to discover local connectivity information. Along with connectivity information, the connectivity helper gathers physical level link information, e.g., delay, through an integrated physical level helper function. In the implementation, these functions collect the information into a link state table and update it when there are changes in the local network environment. The connectivity helper creates a 'Hello' publication, and sends it to its neighbours. This publication contains very simplistic networking information, e.g., the node's identifier, and its interfaces. When the connectivity helper has received the neighbouring nodes' 'Hello' publications, it creates the LSA publication and publishes it. As mentioned above, the Topology Manager is subscribed to this publication, and it will receive it. The Connectivity Helper must reside on each of the forwarding nodes.

The optional Link-state helper module is responsible for collecting and maintaining more detailed link information. The link-state helper maintains a table of "known" links along with other related and available information on them, e.g., throughput and delay values, which can be utilized for path optimization. Each link-state helper also publishes its known table of link information in a 'FileMSG' publication, thus distributing the information to its neighbouring nodes. At the same time, the Link-state helper is subscribed to the 'FileMSG' publications and receives them from the neighbours. Once it has collected this information into its table, it creates a 'LinkMSG' publication from the table and publishes it. This publication will be delivered further to the Topology Manager.

Finally, the Application helper collects requirements for the local applications, and publishes this information in Application publications, to which the Topology manager is also subscribed.

Once the Topology Manager has acquired knowledge of the network topology, it can calculate optimal paths, taking also into account the optional information about the link and application capabilities, for publication delivery through the network. This Topology Manager's path calculation module is triggered, e.g., when the Rendezvous system requests a data transmission between two nodes.

The basic mode of operation, exchange of topology-related messages, followed by building network state based on information about the existence of network entities, is explained in [2] and [3]. In this deliverable we provide more detailed description of the heavily extended topology management module.



ID	Throughput	Delay	...
11::11	11Mbps	5ms	
12::12	...	...	
...			

Figure 12 Illustration of Topology Management functionality within a single domain

2.3.1 Implementation details

Connectivity Helper

The Connectivity Helper periodically publishes information about its existence in the form of Hello messages. In order to obtain its basic configuration information, each connectivity helper parses a configuration file defined in the command line (e.g., netiod.conf) obtaining information about its unique ID (stored in the configuration file) and outgoing interface ID (stored as “LID” in the configuration file). Each connectivity helper can acquire information about possible connecting links as well (stored as def in configuration file), if such information is provided in the configuration file. This is expected to be, e.g., information about the default route to the Rendezvous Node. The current implementation relies only on knowledge of the connectivity helper’s own identity, without requiring any awareness of existing connections, but it leaves open the possibility of obtaining additional information from the configuration file. The mandatory identity information together with additional optional info, e.g., SId, RId, time stamp, is contained in the Hello message.

Every Connectivity helper is subscribed to Hello messages through the corresponding thread. The thread issues relevant callbacks for processing the subscribed publications. The type of callback depends on the publication that has been received, but its main role is storing incoming publications into a specific, predefined list. Accessing and processing of received publications is done separately in a dedicated thread. Upon the discovery of a connecting node and the corresponding link, the connectivity helper updates the list of its neighbours. Besides the information about surrounding nodes the connectivity helper maintains a record about the exact interface from which the publication came. Each neighbour's lifetime in the connectivity helper storage is associated with a decreasing time to live entry, which is immediately restarted upon receipt of neighbour's publication. If the TTL of the connectivity helper expires, it is removed from the list of neighbours. In the case of any change in the entry list, a LSA (Link State Advertisement) message containing neighbour IDs together with the list of their interfaces is asynchronously published under a predefined SId/RId pair. Otherwise, a LSA publication is issued periodically within predefined intervals.

Topology Manager

As described earlier, the Topology manager subscribes to various publications where different types of link and usage information are published. Every subscription has a corresponding callback for placing received publications into the capability list, saving the time of their immediate processing and letting them to be processed by another thread when needed. Having information about link properties and application requirements, path creation can be influenced by putting appropriate weights to the links.

Based on the publications received, the topology manager uses a corresponding topology graph creation class. More precisely, if it receives input from the link-state and application helpers providing link status and application requirements, the topology manager leverages the module which will take into account all parameters at its disposal while calculating paths. On the other hand, if there is no input neither from applications nor the link-state helper, the topology manager will simply rely on the basic module calculating shortest paths based simply on the connectivity information obtained through neighbour discovery.

Link-state helper

The Link-state helper is in charge of parsing a link conditions file containing information about links in the form of a table, where rows represent linkIDs (the outgoing interface OR the virtualID of the destination) and columns refer to observed link parameters. The link-state helper periodically publishes link information obtained from its locally available link-info file, which is generated by physical level helper functions within the connectivity helper. Simultaneously, it subscribes to the same information published by its neighbours. After receiving a link-state info publication, the helper merges the information received from the neighbourhood and publishes an updated version of the link information in a linkMSG publication. The SId/RId pair of this publication is predefined. On the Topology Manager, the last updated link info publication is used for link weight assignment and path calculation.

The functionality of merging all available link information is envisioned to reside only on dedicated nodes in the network, thus avoiding unnecessary data re-computation and traffic overload. Locally stored link information is obtained by a physical level helper function incorporated into the connectivity helper's functionality. It continuously monitors network condition changes and updates the results in the form of a table. The current implementation provides link delay status updates by simply managing publication time stamps while the throughput information is hard coded for the case of a single domain.

Application helper

Application requirements are obtained and distributed throughout the network by a dedicated application helper. It consists of a simple process that parses command line parameters, translates them into a corresponding dictionary of parameter-value pairs and publishes the acquired info under a predefined SId/RId pair. This publication is intended help optimizate the path calculation procedure carried by the topology manager.

2.3.2 Integration

Recent implementation efforts have been focused on running the topology management module over the network, i.e., using the networking daemon of the Blackhawk prototype. The current implementation of subscribing to topology relevant information relies on *kevent* registration and enforces the reception of publications coming over the Rendezvous Node. Therefore, the connectivity helper's locally generated publications are not prioritized over publications originated on different connectivity helpers coming over the network.

A first performance evaluation of the topology management implementation has been carried over one-domain and networking scenarios. In a single one-domain setup, all nodes communicate directly using predefined SId/RId pairs, without the presence of a separate Rendezvous point as the dispatching entity between publishers and subscribers. In order to test the topology management functionality using the networking daemon, we created a network using Virtual Machines running the Blackhawk prototype. In the current setup, we deploy a network of four nodes, where three of them are native publishers/subscribers, acting as simple network entities. Each of them runs single instances of the topology manager, connectivity helper, link-state helper and application helper functions. The remaining node is responsible for dispatching publications and subscriptions accordingly, thus it has the function of a Rendezvous node. Results obtained from these initial tests are shown in [6].

2.4 Applications

2.4.1 BitTorrent

We have started developing since the first year of the project a PSIRP-based content distribution 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 [4]. 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 does not need to be based on simple tit-for-tat, and the relationship between the RIds of the pieces, which could be based on algorithmic identifiers. Due to limitations of the prototype implementation of the PSIRP architecture however, and especially the lack of inter-domain rendezvous facilities, it will be impossible to use the prototype to experiment with these aspects of the application before the end of the project. As it would make no sense to demonstrate a large scale content distribution application over a broadcast based local area network, we have shifted the design and evaluation effort for this application to the simulation platform which provides these facilities via overlays.

2.4.2 Socket emulator

Any proposal that seeks to radically change the architecture of the Internet must plan to co-exist with the existing Internet for an extended period of time. In particular, in order to be deployed, a new architecture must ensure that it will be possible to execute existing applications on top of it. While many applications, especially content distribution ones, can reasonably be expected to be rewritten so as to operate optimally over an information-centric architecture, there is a vast number of existing, endpoint-centric, applications that will have to operate in some type of compatibility mode, preferably without the need to even recompile them. Since most existing Internet applications were written on top of the widespread Sockets API, the most direct way to make them compatible with a new architecture is to develop a shim layer that transparently translates Socket API calls to the underlying information-centric calls offered by the new architecture.

To this end, we have designed and implemented a Socket API emulator for PSIRP, which allows unmodified Internet applications to operate on top of a native publish/subscribe protocol stack. The emulator transparently maps IP/TCP/UDP addresses to PSIRP SId/RId pairs and translates most of the socket API calls to the calls provided by the *libpsirp* API. Due to the lack of a reliable transport protocol in PSIRP, the effort has focused on fully emulating datagram sockets, with the full emulation stream sockets left for future work. The socket emulator is currently working in datagram mode, albeit with limitations on the number of packets that can be sent to the same socket; this limitation is due to a corresponding limitation in the number of versions that each publication can have. The implementation of the emulator follows the evolution of the PSIRP prototype and will be integrated to the main code release before the end of the project.

2.4.3 Trivial File Transfer Protocol

The Trivial File Transfer Protocol (TFTP) [5] is an Internet standard for lightweight file transfer applications, such as remote booting and configuration. It is intentionally very easy to implement, as it operates on top of UDP, using a stop and wait scheme transport scheme, which essentially takes care of error, flow and congestion control. As a result, it is an ideal test case for the socket emulator, which only emulates datagram sockets. We have implemented TFTP on top of the socket emulator (as well as over native UDP/IP) as a test and demonstration application.

In order to assess the overhead due to the socket emulator and compare the performance of native PSIRP as opposed to emulated socket applications, we have also written a publish/subscribe version of TFTP, that is, a modified application that employs *libpsirp* calls instead of socket API calls. This application uses the same approach as the original TFTP, that is, the stop and wait transport scheme, in order to make the two applications directly comparable and assess the overhead due to the socket emulator. While this is not an ideal design for a publish/subscribe network, it is an experiment in straightforward porting of an Internet application for the PSIRP prototype that could eventually be compared to a publish/subscribe based design approach. Both the emulated and the native versions of this application are currently working, with some limitations however due to the corresponding limitations of the socket API.

2.4.4 Firefox Plugin

Many Web applications nowadays are publish/subscribe in nature, e.g. newsletter, stock prices, weather forecasts, but the underlying communication paradigm is based on the send/receive model. As a consequence, protocols are more complex to build and consume more network resources than necessary. For example, the HTTP long polling protocol requires constant re-subscription of data after receiving an update, incurring extra control overhead and bandwidth consumption.

In order to enable Web browsers with native publish/subscribe capabilities, we implemented a plug-in for the Firefox Web-browser as a front-end for the Blackhawk prototype in the Web. The plug-in enables PSIRP calls from the user interface and supports Web browsers working both in a Blackhawk-enabled node or a non Blackhawk-enabled node. In the latter case, we implemented a pub/sub proxy that encapsulates native pub/sub messages into UDP packets and vice-versa. In this case, users in the IP domain can use the Firefox plug-in to transparently publish and subscribe to publications in the psirp protocol domain without constraints.

The usage of publish/subscribe in the Web presents three main benefits: it allows for real-time notification capabilities, it reduces the hardware requirements for Web-servers and it also reduces the overall energy consumption. By subscribing to content instead of placing a request every time that an update is delivered, we are able to support real-time notifications since the receiver's channels are always open, instead of reopening them through the HTTP request/response model. Second, the usage of the publish/subscribe paradigm reduces state maintenance in the server since we do not need to maintain open TCP connections with clients, thus reducing the hardware requirements for Web server. Finally, the reduced number of messages exchanged in the publish/subscribe paradigm results in lower energy consumption, being an important parameter for embedded devices such as mobile phones and notebooks. A thorough evaluation of these gains will be conducted in the near future as part of the testbed integration.

2.5 Language Bindings

The PSIRP prototype implementation is written in C, therefore C applications can directly employ the *libpsirp* API. The current Blackhawk release also includes additional language bindings for Python and Ruby, two interpreted languages that are ideal for rapid prototyping. The local rendezvous helper, for example, uses the Python API, while the inter-domain rendezvous component uses the Ruby API.

The API for Python is, in fact, the most developed one at the moment. Since Blackhawk v0.2 this API has been object-oriented - that is, each publication corresponds to an object in this system - and includes a set of auxiliary tools that make certain tasks much easier. For example, it provides an easy way to handle publication updates in an event-driven manner.

Language bindings for Java are also currently being developed in the project. Potentially it will be possible to develop fast pub/sub applications while still taking advantage of Java's garbage collected environment. Moreover, developers can utilize a variety of external tools and libraries (data structures, graphical user interfaces, etc) developed by the large Java open source community.

2.6 Example of Current Integration Plans

In D3.4, the integration plan was presented. The following example describes the current plans that are being implemented. However, the final design will be described in an additional technical report that will be published at the end of the project.

The following message sequence chart represents the assumed signalling taking place in the prototype system to enable a subscriber to subscribe a publication. Some explanation to the terms in the figure:

- R-FId X, return zFilter from the current node to Y. This is collected automatically, as the packet is forwarded through the network topology.
- NId-X, a node ID that is used to recognize end node X within the network topology. The TM server is assumed to be able to generate a path from its topology information based end NIds. TM clients are assumed to advertise their NId within "hello messages".
- FId X->Y, a zFilter containing path from X to Y.

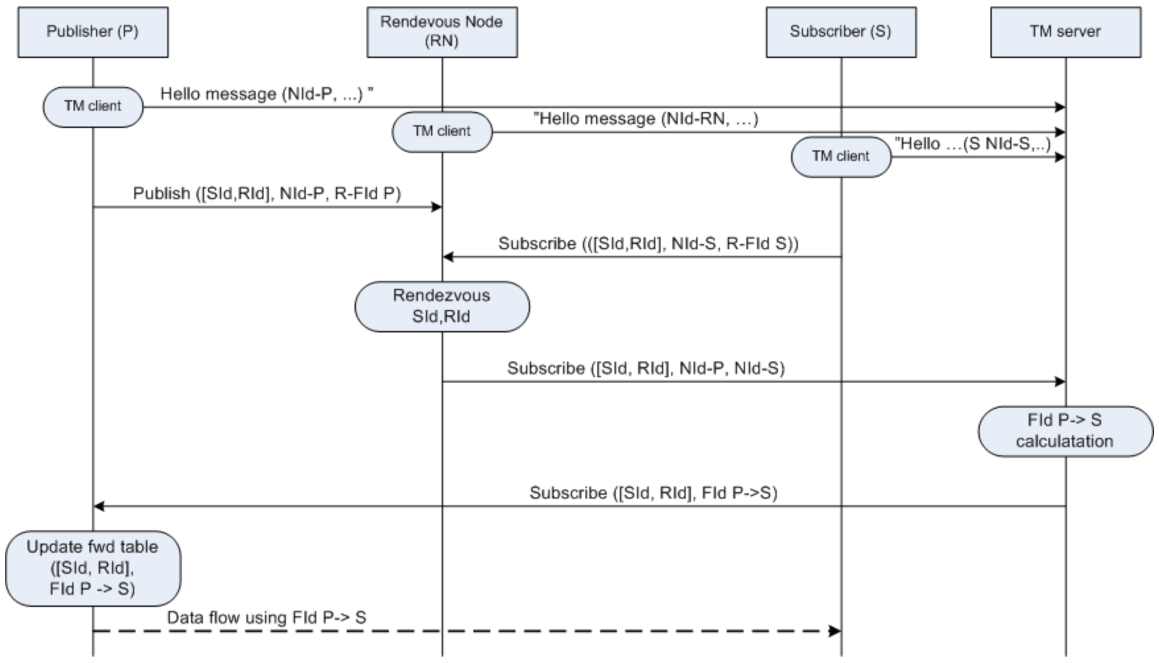

Publisher (P)
Rendevous Node (RN)
Subscriber (S)
TM server
TM client
Hello message (NId-P, ...) "
TM client
"Hello message (NId-RN, ...)
TM client
"Hello ...(S NId-S,..)
Publish ([SId,RId], NId-P, R-FId P)
Subscribe (([SId,RId], NId-S, R-FId S))
Rendezvous SId,RId
Subscribe ([SId, RId], NId-P, NId-S)
FId P-> S calculatation
Subscribe ([SId, RId], FId P->S)
Update fwd table ([SId, RId], FId P -> S)
Data flow using FId P-> S

Figure 13 Basic Publish and Subscribe

3 Open Source Release

During the planning phase, openness was seen as an important element. Accordingly, the project will publish the implementation as open source to provide external partners with the possibility of verifying our work, and further develop thei own applications on top of it. During the project, the core parts of the implementation have been released and they support the basic publication management with the related API, networking, and packet forwarding inside the network.

In the following subsections, the released implementations are described in more details.

3.1 Blackhawk

The prototype has been released both as a source code package requiring an existing FreeBSD OS installation, and as a FreeBSD Virtual Machine image, which can directly be used for testing by running it in a virtual machine monitor. During the project, two versions of the prototype code have been released (v0.1 and v0.2). The third release has been scheduled for May 2010.

The released prototype code has followed the development in the architecture and prototype. The main features for the upcoming version 0.3 have been described in Section 2 of this document.

3.2 NetFPGA forwarding implementation

In the release implementation, we identified all unnecessary parts from the NetFPGA reference switch implementation and removed most of the code that is not required in our system (see Figure 14: on the left side there is the original reference switch design, and on the right side the zFilter forwarding design). The removed parts were replaced with a simple zFilter switch [10].

The current 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 interface. We are using our own

EtherType for identifying zFilter packets. The implementation drops incoming packets with wrong ethertype, invalid zFilter, or if the TTL value has decreased down to zero.

Our prototype has been implemented mainly in the new output port selector module. This module is responsible for the zFilter matching operations, including the 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.

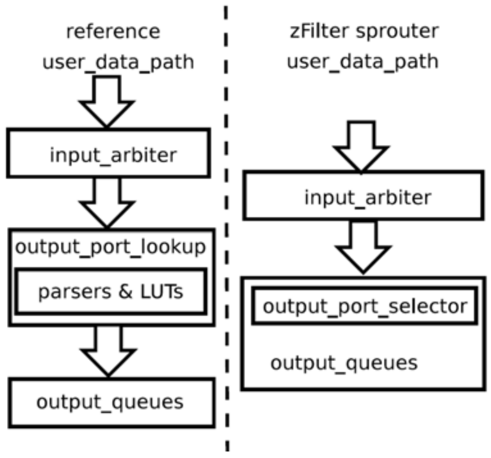


Figure 14 Reference switch modifications

3.3 Packet Level Authentication (PLA)

The source code of the PLA library (*libpla*) v0.1 has been released under the GPL and FreeBSD licences. The released libpla library basically contains all the PLA functionality used by the PSIRP prototype as described in D3.3. It supports PLA header generation and verification, along with functionality for certificate and key management.

The library has been tested on FreeBSD and Linux, and it can be also used with other applications and networking solutions. For example, the libpla library has been used in a master's thesis describing a wireless WPLA architecture [11]. The thesis also utilizes *libpcap* and *libnet* libraries for IP communication in conjunction with the libpla library.

3.4 Documentation

The documentation with installation instructions is maintained in the PSIRP project’s wiki pages [7]. The wiki page was selected due to easiness of maintenance. External partners are also allowed to ask for permission to edit the pages.

4 PSIRP Testbed

During its 3-month extension, PSIRP will establish a testbed facility that will enable us to showcase the integration work, as well as isolated technologies that have been developed within PSIRP. The following section outlines the setup, utilized implementations and applications within this testbed.

4.1 Testbed Set-up

The testbed utilizes the current node implementation, i.e., the Blackhawk implementation. Several nodes, depending on availability of hardware, are installed at various partner sites (see below). The local nodes at each site are directly connected through Ethernet. Each local site is interconnected with other sites through an openVPN configuration, i.e., Ethernet frames are tunnelled via the public Internet. This will also allow for public demonstrations through laptop setups, i.e., a local demo (with one or more laptops) can be connected to the overall setup, assuming that proper openVPN configuration and authorization is in place. The openVPN server is currently installed at the University of Essex.

Initially, the different sites are administered as a single PSIRP domain with a single topology manager [12]. Eventually, however, each site will be configured as a single PSIRP domain to fully enable inter-domain operations. Also, a flexible rendezvous configuration will be implemented, based on the current rendezvous point implementation. All forwarding elements implement packet forwarding with in-packet Bloom filters. The Link Identities will be based on either fixed LIDs or Z-formation based LID calculations (using Z-formation requires support from the Topology Manager). Furthermore, several partners have expressed interest to setup forwarding nodes based on the existing NetFPGA implementation for the Bloom filter forwarding approach, based on fixed LITs and/or Z-formation based LITs.

The currently envisioned setup for the testbed until the end of September is:

- Essex University (non-PSIRP partner): 5 PSIRP nodes, acting as publishers and/or subscribers as well as forwarding nodes
- Cambridge University: 2 PSIRP nodes
- RWTH-Aachen University: up to tens of PSIRP nodes through cluster machines setup
- Athens Universityof Economics and Business: 2 PSIRP nodes
- HIIT: Two NetFPGA forwarding nodes and other dedicated nodes
- IPP-BAS: at least 2 dedicated machines

With this initial setup, in the order of ten or more distributed machines are configured while having additional cluster capabilities. As an additional site, discussions are ongoing with MIT to setup at least one or two machines as a PSIRP domain in the US.

4.2 Virtual Test Bed Set-up

Psnet is a set of Bash scripts that automate the process of using virtual machines for PSIRP Blackhawk prototype testing, validation and development. The scripts were developed to provide a shell (vmShell) to the developer that can create, destroy, configure and etc. the virtual setup.

All tasks for starting, stopping, logging into, configuring VMs are automated by single and simple commands.

Using predefined custom topology files the 'load' command can build the defined network topology. Interconnection between VMs and the host is performed automatically using TAPs and bridges. If a *netiod* configuration is not available, it is generated for all VMs. zFilters are configured properly even if the route to the RN node is going through forwarding nodes. For now only setups with 1 rendezvous node are supported.

On the created topology we can run different scenarios. With the command 'run_sc' we can automatically execute the predefined set of commands on specified VMs. Commands are sent through SSH. The VMs will log the results from those commands and the host will collect the logs after the scenario has finished.

Basic Usage

A scenario can be executed with only 3 commands.

```
[User@host psnet]$ ./psnet
[vmShell] load setup5
....
[vmShell] vm load
...
[vmShell] run_sc sc1
...
[vmShell] quit
```

Details

All VMs are using only one "FreeBSD 8" image in read-only mode. On boot, each VM uses a small script (vm_cnf) to identify itself based on its network interface MAC address. NFS is used to share data between the host and VMs. Because most of the operations in VMs are network or virtual memory related (/pubsub is a virtual directory), NFS performance is not an issue.

Logging

All Blackhawk helpers generate output, which can be stored in log files. In order to be able to read logs, colouring in the helpers output should be switched off. For easier debugging of pslog files, we wrote a colouring scheme for the text editor gEdit.

Results

The developer (tester) is supported while working with a large number of virtual PSIRP nodes. The full functionality of the Blackhawk prototype can be tested/evaluated without any additionally programming on a single server or laptop.

4.3 Applications

Within the multi-site testbed facility, the following applications are envisioned (without any guarantee that all of them will be realized):

- Simple file transfer: one publisher, one subscriber – see Section 2.4.3
- Non-realtime video streaming: one frame per publication with re-publication of new frames, one or more subscribers
- Collaborative working (e.g., shared applications): real-time audio/video conferencing, requiring bidirectional traffic. This is likely to be implemented only at the beta-stage
- Legacy applications through a socket emulator: the socket emulator, implemented at AUEB, will be utilized for, e.g., audio streaming applications over standard IP – see Section 2.4.2
- Web applications with modification to the HTTP model: utilizes the Firefox plugin for the psirp protocol, i.e., metadata document for the rendering is sent with an embedded RId for active objects such as weather or sensor information – see Section 2.4.4

5 Evaluation of the Work

This section gives an overview of the main components and evaluates their degree of integration. Evaluation of performance and stability of the components is outside the scope of this document since it is covered by the upcoming deliverable D4.5.

5.1 Rendezvous Node

The first phase of the rendezvous integration with the local node Blackhawk platform version 0.2 proceeded and finished as planned in D3.4 during autumn 2009: The UDP based communication module used earlier was replaced with the publish/subscribe Ruby API provided by Blackhawk and the basic rendezvous mechanism was established on top of it.

The second phase of this integration has been started and is planned be finished during the first half of 2010. Key elements to be finished in this process are signal format harmonization and network support (from rendezvous IDs to forwarding IDs).

The interconnection layer part of the rendezvous concept will not be implemented as part of this work.

5.2 Blackhawk

The PLA has been integrated with Blackhawk, first with version 0.2, and then the necessary modifications have been made to support the soon to be released version 0.3.

The Network Attachment functionality has not been integrated. The current prototype still requires manual, configuration file based bootstrap. To support Network Attachment, the Rendezvous mechanism needs to be adjusted to support advertisements towards the edges of the network, so that the initial RVS zFilter can be collected for the attaching host. The integration has not yet been done, and the integration is planned to be done before the final demonstration.

The transport functionality has been tightly integrated with the Blackhawk implementation. It was not implemented as a separate module, but merely directly into the system.

5.3 Topology Management

The Topology Management has been adjusted to operate with Blackhawk release 0.3. The implementation work has been done in close co-operation with the Blackhawk enhancements since release 0.2.

6 Conclusions

This document is the last deliverable documenting the implementation efforts within PSIRP. It documents the integration of the various components into a coherent and single prototype. The integration of implementation towards such a single prototype has been progressing since the writing of D3.4. For this, we presented the status of this work and the steps remaining to complete in the remainder of the project.

It can be seen from our presentation that core components like the node platform (Blackhawk), Topology Management and Rendezvous Node have been coming together through our integration efforts. This has allowed for envisioning a true networked setup of a PSIRP network. Such networked setup will be implemented during the extension phase of the PSIRP project until September 2010 and it will encompass all major partner sites. The multi-site testbed will allow for showcasing the capabilities of our technologies, the overall architecture and the potential for new applications. Furthermore, this testbed setup will be utilized to finalize any integration efforts that are still needed. It will also serve as a starting point for future efforts as well as an engagement tool with external project efforts.

Overall we conclude that the implementation efforts have been progressing as planned with major components being successfully implemented in a coherent prototype. Several functions, such as the inter-domain topology formation function, are still missing due to the ongoing design efforts in this space. But we are confident that the chosen platform approach and the current integrative prototype will provide the necessary basis for future extensions.

7 References

[1] D. Trossen (ed.), “Architecture Definition, Components Descriptions and Requirements”, PSIRP deliverable D2.3, February 2009.

[2] P. Jokela (ed), "Progress Report and Evaluation of Implemented Upper and Lower Layer Function", PSIRP deliverable D3.3, June 2009.

[3] D. Trossen (ed), “Integration and Demonstration Plan”, PSIRP deliverable D3.4, September 2009.

[4] G. Xylomenos, K. Katsaros, and V. Kemerlis. Peer assisted content distribution over router assisted overlay multicast. In Euro-NF Future Internet Architecture Workshop, 2008.

[5] K. Sollins, The TFTP Protocol (Revision 2), Internet Request for Comments 1350, 1992.

[6] J. Riihijärvi (ed.), "Final architecture validation and Performance Evaluation Report", PSIRP deliverable D4.5, April 2010.

[7] PSIRP Prototype Documentation, http://wiki.hiit.fi/display/psirpcode/Home

[8] FreeBSD Architecture Handbook, “http://www.freebsd.org/doc/en/books/arch-handbook/”, referred 29th of April, 2010.

[9] P. Jokela (ed.), “Implementation Plan based on Conceptual Architecture”, PSIRP deliverable D3.2, September 2008.

[10] Jari Keinänen, Petri Jokela, Kristian Slavov, Implementing zFilter based forwarding node on a NetFPGA, NetFPGA Developers Workshop, August 13-14, 2009, Stanford, CA.

[11] A. Al Hasib. Design and implementation of Wireless Packet Level Authentication (WPLA). Master's thesis, Helsinki University of Technology, Espoo, Finland, 2009.

[12] M. Ain (ed.), “Update on the Architecture and Report on Security Analysis”, PSIRP deliverable D2.4, September 2010..