IRATI
INVESTIGATING RINA

Grant Agreement No.: 317814

IRATI

Investigating RINA as an Alternative to TCP/IP

Instrument: ***Collaborative Project***
Thematic Priority: ***FP7-ICT-2011-8***

D3.4 Third phase integrated RINA prototype over Ethernet for a UNIX-like OS

Due date of the report: Month 24
Actual date: 13th February, 2015
Start date of project: January 1st, 2013 - Duration: 24 months
version: v 1.0

Project co-funded by the European Commission in the 7th Framework Programme (2007-2013)		
Dissemination Level		
PU	Public	✓
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

FP7 Grant Agreement No.	317814
Project Name	Investigating RINA as an Alternative to TCP/IP
Document Name	IRATI D3.4
Document Title	Third phase integrated RINA prototype over Ethernet for a UNIX-like OS
Workpackage	WP3
Authors	Francesco Salvestrini (Nextworks) Vincenzo Maffione (Nextworks) Nicola Ciulli (Nextworks) Eduard Grasa (i2CAT) Miquel Tarzan (i2CAT) Leonardo Bergesio (i2CAT) Dimitri Staessens (iMinds) Sander Vrijders (iMinds)
Editor	Francesco Salvestrini (Nextworks)
Reviewers	Eduard Grasa (i2CAT) Dimitri Staessens (iMinds)
Delivery Date	13th February, 2015
Version	V1.0

Executive summary

This deliverable presents the FP7 IRATI project's third phase integrated RINA prototype over Ethernet. This prototype provides the features and meets the requirements stated in deliverable D2.4 [4], the software components described in this document have been implemented, tested and integrated into the IRATI's code-base which incrementally provide all its functionalities from an unique code-base [1][2][3]. The resulting prototype meets the requirements for the experimentation activities and has been released to WP4 as well as made available as open-source material to the public (ref. D5.3).

This document is structured as follows. Section 1 provides introductory notes and motivates the implementation approach. Section 2 and 3 provide details on the Shim IPC Process over TCP/UDP and on the SDU Protection functionalities respectively. Section 4 describes the changes to the stack's data-path [1] that were introduced during the project's third phase. Section 5 provides details on the enhancements needed to support reliable flows. Section 6 details the open-source activities related to IRATI's code-base. Section 7 summarises the outcomes of the work performed.

Table of Contents

Executive summary ... 3
Acronyms ... 6
1 Introduction ... 8
2 Shim IPC Process over TCP/UDP ... 9
2.1 The Shim IPC Process over TCP/UDP Factory ... 9
2.2 The Shim IPC Process over TCP/UDP IPC interface ... 10
2.3 Receipt of new data ... 11
3 SDU Protection ... 13
3.1 CRC32 ... 14
3.2 TTL ... 14
4 Distributed loop ... 16
4.1 The IPC Process interface ... 16
4.1.1 Binding a flow to an IPC Process ... 17
4.1.2 Unbinding a flow from an IPC Process ... 17
4.1.3 Enqueueing of SDUs ... 17
4.2 Changes in the KFA ... 18
5 Enhancements of reliable flows ... 19
5.1 Application workflow ... 19
5.2 Incoming SDUs workflow ... 20
5.3 Retransmissions/outgoing control PDUs workflow ... 20
5.4 Workflows example ... 21
5.5 Concurrency and shims adaptation ... 23
5.6 Deferred work in the RMT ... 23
5.6.1 Blocking write ... 24
6 Open source initiative ... 27
6.1 Documentation ... 27
7 Conclusions ... 29
8 Appendices ... 30
8.1 Appendix A – The serialized PDU API ... 30
8.2 Appendix B – The KFA's IPC Process interface ... 30

8.3 Shim IPC Process over TCP/UDP configuration syntax 32

8.3.1 dirEntry 33

8.3.2 expReg 33

9 References 34

Acronyms

API	Application Programming Interface
EBNF	Extended Backus-Naur Form
CRC	Cyclic Redundancy Check
DUP	Data Unit Protection
IP	Internet Protocol
IPC	Inter-Process Communication
IPCP	IPC Process
IRQ	Interrupt Request
KFA	Kernel Flow Allocator
LKM	Loadable Kernel Module
ML	Mailing List
NIC	Network Interface Card
OS	Operating System
PCI	Protocol Control Information
PDU	Protocol Data Unit
QoS	Quality of Service
RINA	Recursive Inter-Network Architecture

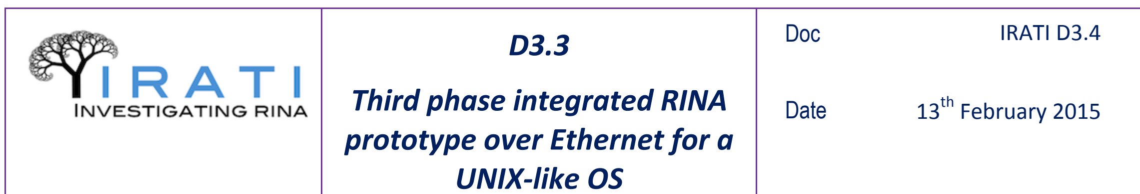
RMT	Relaying and Multiplexing Task
SDU	Service Data Unit
TCP	Transmission Control Protocol
TTL	Time To Live
UDP	User Datagram Protocol

1 Introduction

This deliverable presents the FP7 IRATI project's third phase integrated RINA prototype over Ethernet for a UNIX-like OS. This prototype - as the previous ones [1][2][3] - takes into account its corresponding phase requirements [4], the inputs from WP4 [5] and incrementally introduces new functionalities over the second phase prototype [2][3].

For the third and final prototype, the focus of WP3 has been on introducing the planned functionalities and on improving stability. Functionalities such as the shim IPC Process over TCP/UDP, SDU Protection and support for reliable flows have been introduced; efforts have been made to improve performance of the data-path. These additions only affected a subset of the software components at kernel-space and thus - just like the second prototype - the high level software architecture of the whole stack remained mostly unchanged. As a consequence, both the high-level architecture and the software design of the second phase prototype [2][3] still hold.

The software design acknowledges the requested features and meets the requirements for the project's third phase [4]. The corresponding software components have been implemented, functionally tested and integrated into a single code-base. The resulting prototype meets the stability requirements for experimentation and has been released to WP4 with incremental integration and continuous releases [2]. As reported in D5.3, IRATI's integrated software has been also made available as open source material through a set of social media [7][8][9].



2 Shim IPC Process over TCP/UDP

The Shim IPC Process over TCP/UDP [4][15] aims at wrapping the existing TCP/IP stack into a shim IPC Process. In order to comply with the IRATI design its implementation resides in kernel-space. It makes use of the kernel socket layer (IPv4, `AF_INET`) to provide the necessary functionality.

The Shim IPC process over TCP/UDP provides IPC services to regular applications or (normal) IPC Processes and supports two QoS cubes: *reliable* and *unreliable*; the former is implemented using a TCP socket - i.e. an internet IPv4 socket of type `SOCK_STREAM` - while the latter is implemented using a UDP socket - i.e. and internet IPv4 socket of type `SOCK_DGRAM`.

2.1 The Shim IPC Process over TCP/UDP Factory

The introduction of a new type of shim IPC Process requires a new IPC Process Factory that coordinates the lifecycle of IPC Process instances of that type [1]. Consequently, the Shim TCP/UDP IPC Process Factory has been implemented and integrated into the stack, as the *shim-tcp-udp* Loadable Kernel Module (LKM). This new factory exposes the following interface:

- `tcp_udp_create`: Provides a new instance of a Shim IPC Process over TCP/UDP to the caller – i.e. the KIPCM. That is, allocates and initializes the new instance's internal data-structures.

- `tcp_udp_destroy`: Destroys an instance of Shim IPC Process over TCP/UDP, as obtained through `tcp_udp_create`.

- `tcp_udp_init`: Initializes the IPC Process Factory internal data-structures. This function is executed upon factory registration, which happens at *shim-tcp-udp* module loading/initialization.

- `tcp_udp_fini`: Provides no actions. However, it is used as an assertion point. Since this function is executed by the KIPCM upon un-loading/finalization of the *shim-tcp-udp* module, the function checks if the factory's internal data structures came back to their initial state (as in `tcp_udp_init`) – i.e. the number of successfully executed `tcp_udp_create` calls must be equal to the number of successfully executed `tcp_udp_destroy` calls.

After creation, a new IPC Process instance has to be assigned to a DIF (ref. [1], section 2.1.5.1.1). For the Shim IPC Process over TCP/UDP case, this operation must provide to the instance the following information:

- A hostname that is used internally when binding listening (TCP) or receiving (UDP) sockets.

- A RINA directory that contains mappings (in terms of a list of `dirEntry` items, ref. section 8.3) between application names and internet socket names (an internet socket name comprises an IP addresses and a port number). When the IPC Process receives a flow request towards a remote application, the directory is consulted to find the IP address and port number to which each flow allocation request has to be sent.

- A list of expected application registrations (in terms of a list of `expReg` items, ref. section 8.3), that is the names of the applications that are expected to register with the shim IPC Process. Each list entry maps an application name to a unique port number. In this way, when an application registers with the shim IPC Process, the associated port is used to bind the listening socket (for the TCP socket) or the receiving socket (for the UDP socket).

Once the Shim IPC Process over TCP/UDP instance has received the aforementioned information, it becomes fully operational and applications can start using the IPC API to request its services.

2.2 The Shim IPC Process over TCP/UDP IPC interface

The Shim TCP/UDP Factory produces objects (Shim TCP/UDP instances) that have the same northbound interface as the other (shim) IPC Processes in the stack, (ref. [1], sec. 2.1.5.1). Therefore, their northbound interface provides the following operations:

- `tcp_udp_application_register`: If the application that invoked the operation is expected to register with the shim DIF, both a UDP and a TCP socket are created and bound – using `kernel_bind()` - to the port associated to the application in the expected applications registration list. Furthermore, the `kernel_listen()` operation is invoked on the TCP socket only. In this way, the registering application becomes reachable through the shim DIF. It is necessary to create both a TCP and a UDP socket because, at registration time, the QoS cube that will be requested by a remote shim IPC Process asking for a flow allocation is not yet known. When an allocation request comes, some data are written by the remote IPC Process on one of the two sockets, so that the local IPC Process can complete the flow allocation.

- `tcp_udp_application_unregister`: This operation ensures that the application is no longer reachable through the shim DIF. The TCP socket is torn down (using the `kernel_sock_shutdown()` call). Next, the UDP receiving socket and the TCP listening socket are released.

- `tcp_udp_flow_allocate_request`: This method is essentially used to perform client-side socket connection setup. A directory lookup is executed, in order to find the remote IP address and port corresponding with the destination application name. If the request asks for a reliable flow, a TCP socket is created and `kernel_connect()`

called to initiate the TCP connection to the remote socket associated with the destination application. If the request asks for an unreliable flow, a UDP socket is created and bound (`kernel_bind`) with the hostname (IP address) specified in the shim DIF configuration. If that operation fails, an error is returned to the requesting (source) application. If the operation is successful, a new flow is created and bound to the newly created socket.

- `tcp_udp_flow_allocate_response`: Invoked by the destination application in order to respond to an incoming flow allocation request. If the corresponding allocate request came from a TCP socket, an incoming TCP connection was accepted using `kernel_accept()`. If the corresponding allocate request came from a UDP socket, it means that a packet was received from a previously unknown source (IP address) which is seen as a new UDP flow allocation request (see also section 2.3). If the destination application accepts the flow, all packets that will be received by the shim IPC Process on the corresponding TCP connection (or on the corresponding UDP receiving socket) will be delivered to the flow. If any packets had arrived in the time the flow was pending – i.e. between the creation of the flow and the positive response - they are delivered to the destination application in the flow allocation response method. If the flow is denied, all received pending packets are dropped and, in case of TCP connection, the "accepted" socket is destroyed.

- `tcp_udp_sdu_write`: The SDU passed as an argument is sent as the payload of a TCP segment or UDP datagram, depending on the selected QoS-cube – i.e. reliable or unreliable, respectively. The socket associated with the flow and the SDU are passed directly to the `kernel_sendmsg()` function.

- `tcp_udp_flow_deallocate`: Invoked to instruct the IPC Process to deallocate the flow. If a TCP socket is associated with the flow, the flow de-allocation is triggered also on the opposite side by using the `kernel_sock_shutdown()` operation. Otherwise – in case of a UDP socket - the operation is local, since UDP has no explicit flow (de)allocation.

2.3 Receipt of new data

Similar to what happens with the Shim IPC Process over IEEE 802.1Q [1], the Shim IPC Process over TCP/UDP uses a callback function that is registered with the kernel and is called upon the reception of new data on one of the sockets mentioned in section 2.2. There are three cases:

- New data is received on a UDP socket. If no flow is currently associated with the remote IP address and port, the KIPCM is notified (using `kipcm_flow_arrived()`) about the incoming flow allocation request represented by the packet and the UDP payload is temporary enqueued for subsequent deliver – that happens if the flow gets

accepted by the destination application. If a flow already exists, the UDP payload is posted to the upper IPC process or application.

- New data is received on a TCP listening socket. This reports an incoming TCP connection. The `kernel_accept()` operation is executed to get the socket that gets associated to the new pending flow. The KIPCM is notified - using `kipcm_flow_arrived()` - about the incoming flow allocation request.
- New data is received on a TCP connection socket. A flow corresponding to this connection already exists. The TCP payload is posted to the upper IPC process or application.

In all cases, the UDP/TCP payload is obtained using the `kernel_recvmsg()` function.

3 SDU Protection

Development of an initial Service Data Unit (SDU) Protection infrastructure, referred to as DUP software component in the following, has been started during the project's second phase and completed in the third phase. Only normal IPC Processes have been upgraded to use the DUP component which is available at kernel-space only. Applications that wish to protect their SDUs in different ways, or do not trust the protection services offered by the DIF they are dealing with, will have to implement data protection on their own - e.g. using an external library.

SDU Protection is applied on serialized PDUs [2], at a system-wide level. It applies globally to all normal IPC Processes running in a system, and can be enabled/disabled through Kconfig (ref. [1], sec. 4.1.1). The resulting functionality is sufficient to drive experimentations, as normal IPC processes are able to support SDU Protection mechanisms.

During the integration of the DUP component, the API of the serialized PDU management component – i.e. the *pdu_ser* - has been slightly changed, in order to accomodate the DUP in the architecture. Its functionalities have been also enhanced in order to allow growing and shrinking of serialized PDUs by an arbitrary number of bytes. Refer to sec. 8.1 for the updated serialized PDU API.

Two forms of SDU protection are available in the IRATI stack:

1. CRC32: Calculates and adds CRC32 checksum to a serialized PDU (4 bytes). In each IPC Process (in the same DIF) - that forwards the PDU, the CRC32 is recomputed. Upon reception of a PDU, if the calculated checksum differs from the checksum stored in the PDU itself, the PDU is dropped. This protection is mostly used when the underlying transmission medium has no error handling mechanism.

2. TTL: Manages the Time To Live (TTL) field of the PDU. On the sending node an initial TTL is added to the PDU (1 byte). Every time an IPC process forwards the PDU - i.e. at each hop - the TTL is decremented. If the TTL reaches zero, the PDU is dropped. This protection ensures that PDUs are dropped in case of temporary/erroneous routing loops.

Although SDU protection is applied system-wide, the CRC32 and TTL mechanisms are independent from each other and therefore the kernel-space build system (ref. [1] section 4.1.1) has been adapted in order to let the users select them as needed (the components and the calls to their API are guarded by `#ifdef`/`#endif` directives). When both are required, these two policies are applied in order. In the egress direction the TTL is executed first while it is executed last in the ingress direction.

3.1 CRC32

The DU-Protection CRC32 software component provides the following API:

- `bool dup_chksum_set(struct pdu_ser * pdu);`
- `bool dup_chksum_is_ok(struct pdu_ser * pdu);`

The `dup_chksum_set()` function adds a CRC32 checksum to the beginning of the serialized PDU passed as argument. When the operation returns, the first four bytes of the serialized PDU contain the calculated CRC32.

The `dup_chksum_is_ok()` function recalculates the checksum of the serialized PDU passed as argument (skipping the first 4 octets) and compares it with the first four bytes in the PDU. The result of the comparison between the two values is returned – i.e. `false` if they differ, `true` otherwise.

The logical place where to insert the DU protection API calls was in the *serdes* component [2]:

- For outgoing traffic, before adding a CRC32 to a PDU, the serialized PDU is first grown by pushing 4 bytes in the front (see the appendix for the pdu_ser API). Then, in order to add a CRC32 checksum, `dup_chksum_set()` is called.
- For incoming traffic, the CRC32 checksum is first checked by calling `dup_chksum_is_ok()`. If the PDU contains errors, it is dropped. Otherwise, it is shrunk by popping the front by 4 bytes and normal processing continues.

3.2 TTL

The TTL software component provides the following API:

- `bool    dup_ttl_set(struct pdu_ser * pdu, size_t value);`
- `ssize_t dup_ttl_decrement(struct pdu_ser * pdu);`
- `bool    dup_ttl_is_expired(struct pdu_ser * pdu);`

The `dup_ttl_set()` function takes as input a serialized PDU and a value for the TTL. The value is put in the first byte of the serialized PDU.

The `dup_ttl_decrement()` function takes as input a serialized PDU, interprets the first byte of the serialized PDU as the TTL, decrements this value and returns it.

The `dup_ttl_is_expired()` takes as input a serialized PDU, and interprets the first byte as the TTL. If the TTL is zero, `true` is returned. Otherwise, a `false` value is returned.

In the initial implementation of the TTL management, these functions are also called from the serdes component (as well as for the CRC32 related ones, as in the previous section):

- For outgoing traffic, the serialized PDU is grown by pushing one byte in the front. Next, the TTL value is set in the PDU using the `dup_ttl_set()` function. A (user-configurable) initial value is used on the first node when the PDU is passed to RMT. The TTL is checked through the use of `dup_ttl_is_expired()`. If the TTL has expired, the PDU is dropped. Otherwise, normal processing continues.
- For incoming traffic, `dup_ttl_decrement()` is first called. The returned value is placed in the unserialized PDU. Next, the serialized PDU is shrunk in the front by one byte.

4 Distributed loop

During the first and the second phases of the IRATI project, the Kernel Flow Allocator (KFA) was the software component managing the flow of the read/write operations among IPC Processes. Read and write calls from application or IPC Processes addressed the KFA (through either `sdu_write` or `sdu_read` syscalls), which in turn redirected the operation to the proper application or IPC Process based on mappings contained within its data model (i.e. the flow data structure) [1]. This approach was chosen to ease implementation and keep the development within the project's tight timeline and planning. Although the KFA was playing the role of a centralised management entity for the data-path, this did not degrade performance significantly when only a low number of flows were established concurrently.

However, a higher degree of decoupling of the IPC Process data-path was envisioned since the beginning of the project and it has been introduced. Development during the third phase moved the workflow from centralised to distributed management of the data-path. We referred to this as either the *distributed loop* or *distributed workflow*.

The *distributed workflow* relieves the KFA from keeping the bindings between IPC Processes, i.e. the binding between the IPC Process on top, which is the user of the flow or an application, and the IPC Process below, which is maintining the flow. These bindings are now stored into each IPC Process on a per-flow basis, so *write* and *read* operations do not always rely on the KFA anymore. Thus, IPC Processes write SDUs to IPC Processes below and read SDUs from the IPC Processes on top, without the intervention of the KFA. However, the KFA remains as the manager of port identifiers and thus it can be imagined as a thin adaptation layer for applications only. Applications read and write SDUs through the KFA, while IPCPs write incoming SDUs (intended for applications) into the KFA.

The implementation of the *distributed loop* only affects the read/write data-path, and the changes required to implement the functionality affect part of the kernel-space software components, being the KFA, the (normal) IPCP, and the shims - i.e. *shim-eth-vlan* [1], *shim-tcp-udp* and *shim-hv* [2][3]. The IPC Process interface was slightly modified, adding new operations required for the integration of the distributed loop.

The following sections describe the details of the changes applied to the IPC Process interface and to the KFA data-model.

4.1 The IPC Process interface

In order to accommodate the distributed loop functionality, the set of calls defining the existing IPC Process interface was extended. These new calls are used only by the normal IPC Process and the KFA by means of its IPC Process API. They are used by the underlaying IPC Process to invoke the one on top and therefore, there is no need for the shim IPC Processes to implement them (they cannot have underlaying IPC Process).

Already existing calls related to the creation of a connection – i.e. `connection_create`, `connection_arrived` and `connection_update`, ref. [1] section 2.1.5.1.1 - have been updated with the introduction of a new parameter in their prototypes. This parameter is a pointer to the user of the flow and thus it is used to inform the N-1 IPC Process about the N IPC Process using the flow, either another IPC Process or the KFA on behalf of a user space application.

The details of the new operations introduced in the IPC Process API are described in the following sections.

4.1.1 Binding a flow to an IPC Process

When an N-1 IPC Process receives a flow request from the N IPC Process, the N-1 IPC Process invokes the `flow_binding_ipcp` operation on the N IPC Process. This operation informs the N IPC Process of the port-id of the N-1 flow just created and also passes a pointer to the N-1 IPC Process instance itself, so that the N IPC Process can use it to call the `sdu_write` operation on that port-id. The binding of port-id and N-1 IPC Process instance is stored in the N IPC Process' RMT.

The prototype of the operation is as follows:

```
int flow_binding_ipcp(struct ipcp_instance_data * user_data,
                      port_id_t                   port_id,
                      struct ipcp_instance *      n1_ipcp);
```

4.1.2 Unbinding a flow from an IPC Process

The `flow_unbinding_ipcp` operation is the opposite of `flow_binding_ipcp`. The operation is triggered by the N-1 IPC Process after an N-1 flow is destroyed, so that the N IPC Process can remove the port associated with this flow.

The prototype of the operation is as follows:

```
int flow_unbinding_ipcp(struct ipcp_instance_data * user_data,
                        port_id_t                   port_id);
```

4.1.3 Enqueueing of SDUs

The N-1 IPC Process triggers the `sdu_enqueue` operation to forward incoming SDUs to the (client) N IPC Process or to the KFA that will deliver it to the user-space application.

The prototype of the operation is as follows:

```
int sdu_enqueue(struct ipcp_instance_data * data,
                port_id_t                   id,
                struct sdu *                sdu);
```

4.2 Changes in the KFA

The main change in the KFA is the addition of the IPC Process data structures into its data-model, in order to mimic an IPC Process compliant interface (ref. [1] section 2.1.5.1).

An IPC Process instance pointer (`struct ipcp_instance *`) was added to the KFA's data model. This `ipcp_instance` was extended with the necessary data-structures (data and operations) [1] so it behaves like any other IPCP process with respect to the distributed-loop. The *data* part only contains a pointer to the KFA instance while the *operations* part is restricted to the minimal set of calls the KFA has to support (i.e. `flow_ipcp_bind`, `flow_deallocate`, `sdu_post`, `flow_sdu_write`, `name`, `enable_write` and `disable_write`). Refer to section 8.2 for details on the IPC Process' data structures as well as the changes applied to the KFA data-model.

In this way, IPC Processes - either normal or shim - can use the IPC Process interface to write SDUs to the layer on top, regardless of it being an application (through the KFA) or another IPC Process. The resulting *read/write* workflow implementing the distributed loop is depicted in the following figure.

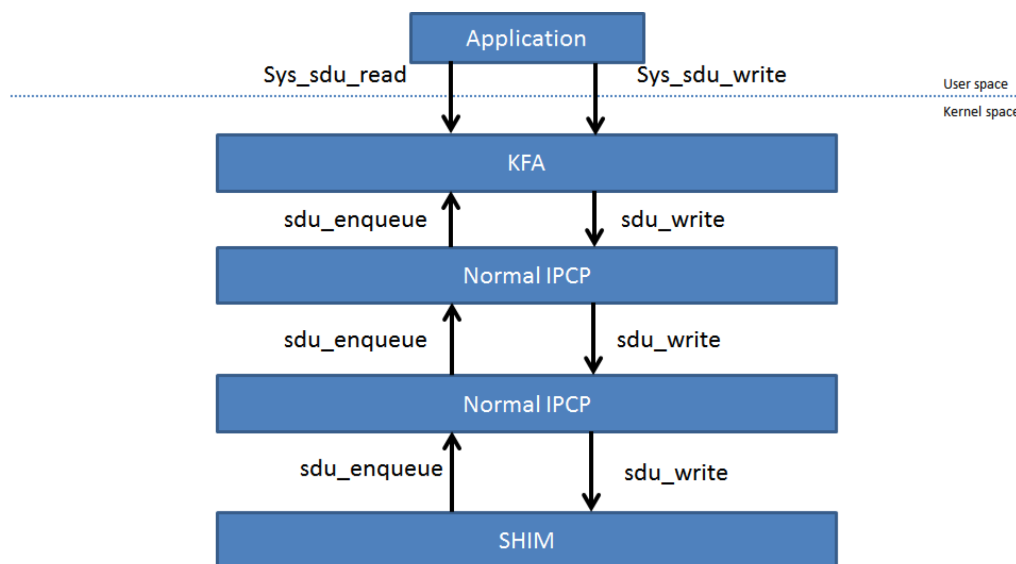


Figure 1 read/write operation (phase III)

Therefore, the KFA is now responsible for:

- Managing the `sdu_ready` queue [1] which stores the SDUs that were delivered by the underlaying IPC Process before delivery to the client application;
- Redirecting the application's *write* operation to the proper IPC Process;
- Blocking the write/read operations when a flow's credit is exhausted or there is nothing to read from that flow.

5 Enhancements of reliable flows

During the third phase, the Reliable Flows implementation from the second phase prototype [2] was further enhanced in order to minimise context switches and work deferrals in the stack's data-path.

In the following subsections, the various SDU workflows enhancements are presented separately (ref. 5.1, 5.2 and 5.3) and then grouped together in an example (ref. 5.4). Next, the refactoring of the KFA, normal IPC Process and shim IPC Processes - required for the implementation of such changes - are described (ref. 5.5 and 5.6).

5.1 Application workflow

Figure 2 depicts the workflow that begins with an application writing an SDU to a port-id. In order to minimise context switches and work deferrals, it proceeds in process context from the KFA to the IPC Process' RMT, eventually reaching a shim IPC Process. This corresponds to workflow (1) in the figure. However, it might be the case that it is not possible to reach the RMT component because the N-1 port to be used is already in use by the RMT itself (due to concurrent write operations ongoing). In this case deferral is unavoidable and thus a tasklet is scheduled to proceed at a later time with the operation, as workflow (2) in the figure shows. The work will proceed in softIRQ context from that point on, avoiding a new context switch – ref. workflow (3) in the figure.

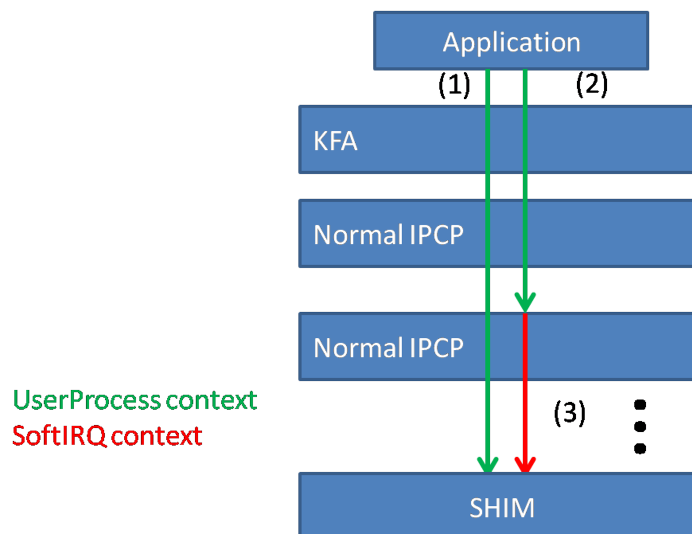


Figure 2 write workflow

5.2 Incoming SDUs workflow

For the incoming SDUs' workflow, which starts with a packet received through a Shim IPC Process, the context in which the work will proceed depends on the type of shim IPC Process (i.e. softIRQ or process contexts). The next figure shows the different possible workflows.

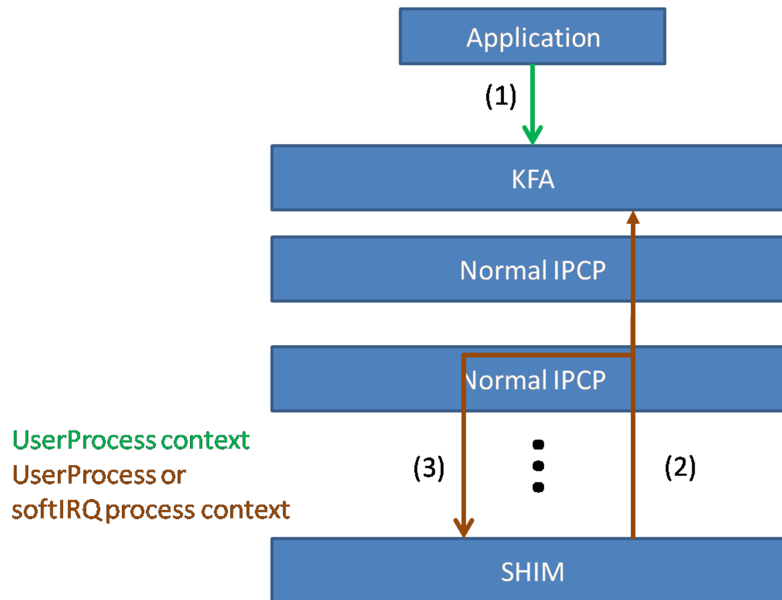


Figure 3 incoming SDUs data path

Workflow (1) in Figure 3 shows the read operation performed by an application, which is processed by the KFA. The SDUs arrive in the KFA from workflow (2). A shim IPC Process receives a packet and creates a SDU, forwarding it to a normal IPC Process and/or the KFA without deferrals or context switches. Workflow (3) corresponds to an SDU that must be relayed to a different IPC Process which is also carried out without deferrals or context switches.

5.3 Retransmissions/outgoing control PDUs workflow

Figure 4 shows the workflow either for retransmissions or for sending control PDUs in case a timer (inactivity timers, A-timer) expires.

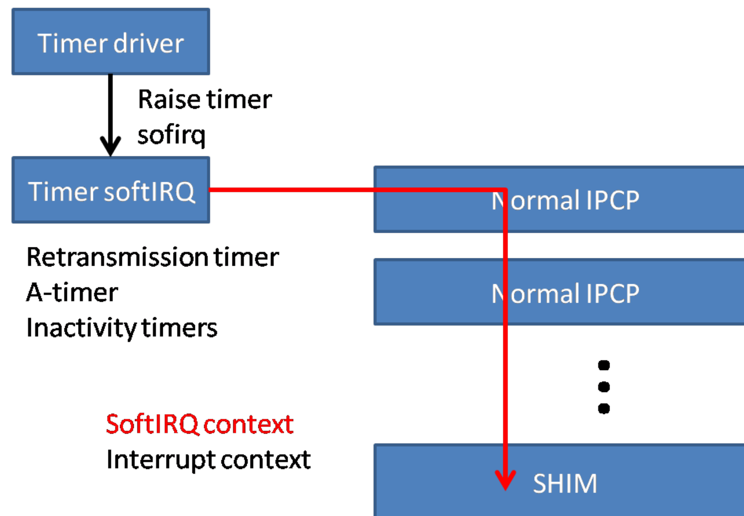


Figure 4 Workflow for retransmissions

This workflow starts in softIRQ context, and proceeds down the stack of IPC Processes without deferrals or context switches, unless a port is already in use - as in workflow (2) in Figure 2. In this case, the same procedure as in workflow (3) in Figure 2 is applied.

5.4 Workflows example

The workflows described in the previous sections are summarised in this subsection through the use of an example (ref. Figure 5), which uses the *shim-eth-vlan* [1] as bottom DIF.

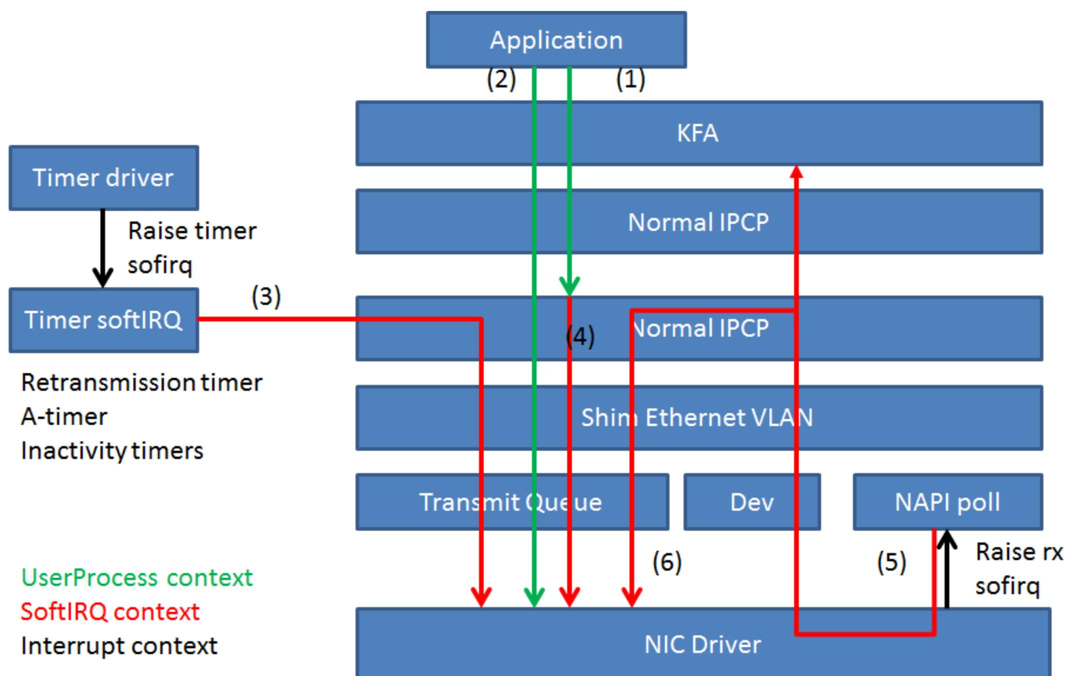


Figure 5 Different data transfer workflows with the shim-eth-vlan

Workflows (1) and (2) represent the "regular" *write* operation workflows, as triggered by an application in user-space (through the `sdu_write()` syscall). After going through the KFA, the SDU arrives at a normal IPC Process (ref. section 4).

Depending on the network configuration, the SDU may cross several normal IPC Processes (two are shown in the Figure) until it arrives at a shim IPC Process. Following workflow (2), the traversal of the normal IPC Process can be done without deferring work – by executing in process context while the N-1 port to use is not already being used by the RMT (i.e. in a concurrent *write* operation). Otherwise, if the RMT port to be used is already in use - ref. workflow (1) - the work is deferred by means of an RMT tasklet that will continue the write operation when the N-1 port becomes available again. This second part corresponds to the workflow (4), which is executed in softIRQ context.

In both workflows (2) and (4), once the SDU arrives at the shim-eth-vlan IPCP, the SDU is wrapped in an Ethernet frame and it is handed to the NIC driver for transmission.

Workflow (3) corresponds to *write* operations triggered either by an A-timer timeout (maximum time to wait until sending an ACK control PDU) or by a Retransmission timeout (maximum time to wait for an ACK before retransmitting the PDU).

Finally, workflows (5) and (6) correspond to the reception of a PDU (for the case of the shim-eth-vlan). When the NIC driver detects the arrival of a new Ethernet frame, it schedules a

softIRQ - i.e. `NET_RX_SOFTIRQ` - to signal the kernel about new frames available. Once the `net_rx_action()` softIRQ handler is scheduled, the NAPI [22] poll driver callback is invoked in order to process the received packet(s). Each received packet is intercepted by the *shim-eth-vlan* reception handler. Then the workflow continues up through the whole stack of DIFs until it reaches the KFA and then the application (5), or until it has to be forwarded to another remote IPC Process - thus a write operation is performed on an N-1 port (6).

5.5 Concurrency and shims adaptation

Although the shim-eth-vlan deals with frame reception using a softIRQ-based strategy inherited from the NAPI mechanism, this is not the case for other shims - e.g. the Shim IPC Process over TCP/UDP invokes the API of a normal IPC Process on top (in process context) from a workqueue. For this reason, in order not to impose restrictions to other shim implementations, the data-path has been updated in order to allow incoming PDUs either from process or interrupt contexts. This has been achieved by controlling operations that may involve sleeping calls (i.e. *kmalloc*) and disabling interrupts while locking protected data structures that may be accessed from different contexts at the same time. As a result, the processing will not be affected by the context in which the PDU is received.

5.6 Deferred work in the RMT

Regarding the *write* workflows (2) and (1, 4) of Figure 5, the RMT implementation always tries to send the PDU passed by the DT component [2] to the underlaying IPC Process using the port-id of the N-1 IPC Process supporting the flow. If the port is not currently in use, the operation follows workflow (2) - which corresponds with workflow (1) of Figure 6.

Figure 6 RMT internals of the write operation

The RMT controls the status of these N-1 ports and, when the required one is being used by another write operation - as it is the case of workflow (1) of Figure 6 - it follows the workflow (2), as depicted in the same Figure. The RMT pushes the PDU to an outgoing queue associated to the port, deferring the work until the port becomes available again. The deferral is implemented with a tasklet, in order to accommodate high-priority / low-latency requirements. Workflow (3) of Figure 6 depicts the work done by the tasklet in softIRQ context - that is, retrieving the PDUs from the outgoing queue and sending them to the N-1 IPC Process.

5.6.1 Blocking write

Another important feature that was required in order to successfully implement the flow control capabilities was the ability to block the upper IPC Process (or application) from writing when the flow credit is exhausted and the closed window queue is full. This was done by:

- Introducing a *waitqueue* in the KFA
- Adding state variables in the RMT's N-1 ports
- Adding new operations in the IPC Process API:
  - ```
    int enable_write(struct ipcp_instance_data * data,
                     port_id_t                   id);
    ```
    ```

- `int disable_write(struct ipcp_instance_data * data, port_id_t id);`

The following sections describe the behaviours of the data-path in case of blocking *write* operations in the KFA and normal IPC Process.

5.6.1.1 Blocking write in the KFA

The following figure shows the blocking write implementation in the KFA.

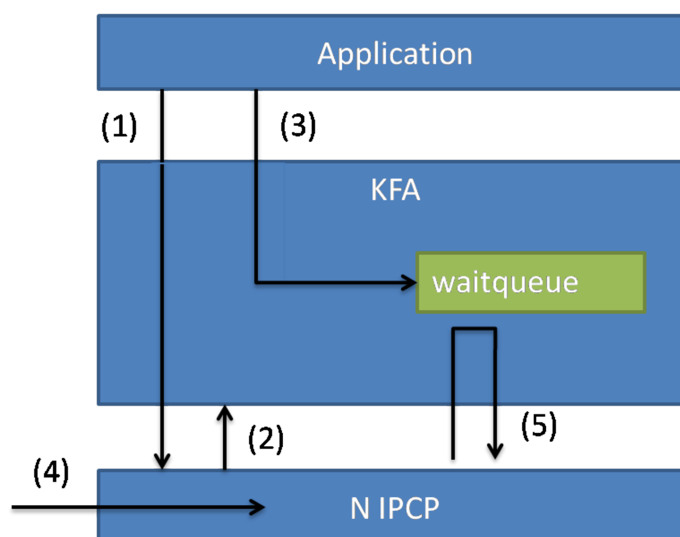


Figure 7 Write blocking in KFA

(1) The application performs an `sdu_write()` syscall that is handled by the KFA. The SDU is sent lower in the stack.
(2) If more SDUs are handed to the N-1 flow than it is allowed to send, its credit will be used up. The next SDUs will be placed in its closed window queue. If this closed window queue is also full, the N-1 flow calls the `disable_write()` function in the KFA which updates its internal state.
(3) The next `sdu_write()` operation invoked by the application is now put to wait, blocking the syscall until the N-1 IPC Process enables the write again.
(4) A control PDU, extending the N-1 flow's credit, arrives at the N-1 IPC Process. It first empties its closed window queue.
(5) If there is still credit after sending all the SDUs that were queued in the closed window queue, the N-1 IPC Process calls `enable_write()` in the KFA making the `sdu_write()` syscall wake up, finish its work and return control to the application.

5.6.1.2 Blocking write in the normal IPCP

The following figure shows the blocking write implementation in the normal IPC Process.

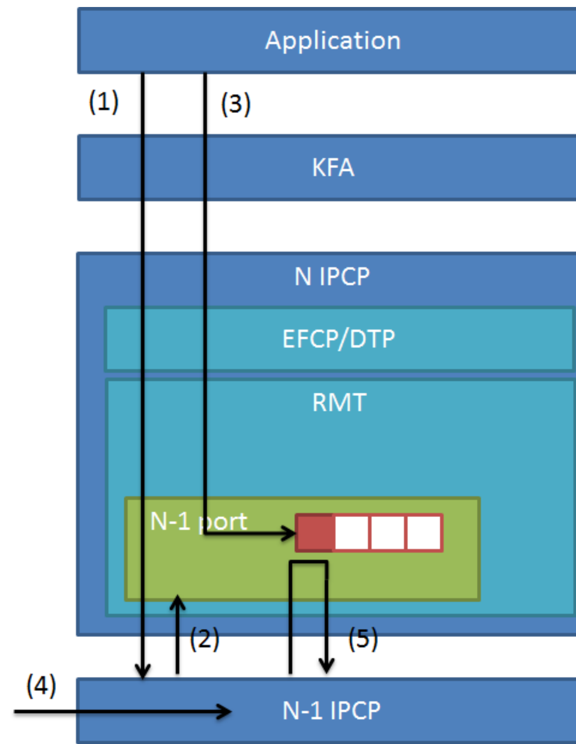


Figure 8 Write blocking in normal IPCP

(1) The application performs an `sdu_write()` syscall that is received by the KFA and, since the two N and N-1 flows still have credit, the SDU is sent lower in the stack.
(2) After sending its latest SDU, the N-1 flow supported by the N-1 IPC Process exhausted its credit and filled up its closed window queue. Therefore, the N-2 flow calls the `disable_write()` operation in the N-1 IPC Process which in turn updates its internal state, blocking the N-1 port for writing.
(3) The next `sdu_write()` issued to the N-1 IPC Process will find the operation blocked and thus, the SDU will be enqueued in the N-1 port queue and the control will return back to the caller.
(4) A control PDU, extending the N-1 flow's credit, arrives at the N-1 IPC Process. It first empties its closed window queue.
(5) If there is still credit, the N-1 IPC Process calls the `enable_write()` operation in the N-1 IPC Process, signalling the RMT to empty the N-1 port queue.

6 Open source initiative

During the third phase, the FP7 IRATI stack development transitioned from a completely closed development model - where the code was hosted on a closed git repository [6] - to an open one - where the code has been made publicly available on an open GitHub repository [7].

GitHub [10] mainly offers free git [19] repositories hosting facilities to open-source projects and can be considered as a one of the leading platforms for promoting (and maintaining) open source activities. Apart from hosting the code, Github offers issue tracking tightly bound together with automation features [21], and hosting for per-project web and wiki pages. Features such as closing an issue on the issue-tracker with a commit message on the repository, referring to a file in the repository from a wiki page and updating the project's web-site contents from the repository itself – greatly minimise the overall maintenance efforts. For these reasons GitHub was chosen over other alternatives (e.g. BitBucket [20]).

In addition, an open Mailing List (ML) [13] has been created for discussing issues related to the IRATI stack.

Therefore, the list of the freely available online services related to the IRATI open-source initiative can be summarised as follows:

- Source repository (code and low-level documentation placeholder): https://github.com/IRATI/stack
- Wiki pages (high-level documentation placeholder): https://github.com/IRATI/stack/wiki
- Issues tracking: https://github.com/IRATI/stack/issues
- Web site: http://irati.github.io/stack
- Mailing list: http://www.freelists.org/list/irati

6.1 Documentation

Given the incremental additions to the software and the slight changes applied to the high level architecture during both the second and the third phase of the project, the documentation provided in previous WP3 deliverables [1][2][3] is still valid and can be considered as a detailed description of the software architecture. The same deliverables, coupled with the WP4 ones [5][23], provide instructions for the configuration of the software (e.g. the structure of `ipcmanager.conf` configuration file) as well as the details for its execution in different environments.

Due to the RAD approach to implementation and the considerable size of the IRATI code-base, keeping the documentation up-to-date with the software requires additional planning.

The stability of the high-level design allows us to embed the software details directly into the source code. The well-known Doxygen tool [12] was adopted in user-space while Docbook [14] - the de-facto standard for documenting the kernel-sources - was inherited for the parts of the stack in kernel-space. During the development of all prototypes, the source-code has been annotated using a syntax understood by these tools. Moreover, the build infrastructure has been enhanced with the necessary mechanisms in order to detect the requirements (i.e. the Doxygen binaries) on the building host and generate the documentation automatically from source-code.

7 Conclusions

The final FP7 IRATI prototype complies with the requirements described in D2.4 [4] and provides the features required for the experimentation activities that will be described in D4.3. The functionalities - incrementally introduced during the third phase, over the previous prototypes - span part of the data-path of the IRATI stack [1]. The resulting software provides the implementation of the Shim IPC Process over TCP/UDP (ref. section 2) and system-wide SDU Protection (ref. section 3). The distributed loop functionalities (ref. section 4) introduced increases the independence of the IPC Processes regarding the KFA and allows them to interface to other IPC Processes (i.e. the one on top and the one below) directly, without the involvement of the KFA. Moreover, it also provides support to the reliable flows (ref. section 5) through a revised workflow which has impacted only the critical sections of the datapath.

The resulting software, the complete FP7 IRATI project codebase, has been tagged as version 1.0.0 and made publicly available on an open repository enhanced with a set of social media such as an open mailing list and a wiki (ref. section 6). The software provides – from within the same codebase - all the features planned for the three IRATI phases, still complying with the experimentations performed in the respective phases.

8 Appendices

8.1 Appendix A – The serialized PDU API

```
struct pdu_ser;

struct pdu_ser * pdu_ser_create_buffer_with(struct buffer * buffer);
int              pdu_ser_destroy(struct pdu_ser * pdu);

bool             pdu_ser_is_ok(const struct pdu_ser * pdu);
struct buffer *  pdu_ser_buffer(struct pdu_ser * pdu);
int              pdu_ser_buffer_disown(struct pdu_ser * pdu);

int              pdu_ser_head_grow(struct pdu_ser * pdu,
                                   size_t           bytes);
int              pdu_ser_head_shrink(struct pdu_ser * pdu,
                                     size_t           bytes);
```

8.2 Appendix B – The KFA's IPC Process interface

```
struct kfa {
        spinlock_t              lock;
        struct pidm *           pidm;
        struct kfa_pmap *       flows;
        struct ipcp_instance *  ipcp;
        struct list_head        list;
};

struct ipcp_instance_data; /* To be defined by each IPCP */

struct ipcp_instance_ops {
        int  (* flow_allocate_request)(struct ipcp_instance_data * data,
                                       struct ipcp_instance *      usr_ipcp,
                                       const struct name *         source,
                                       const struct name *         dest,
                                       const struct flow_spec *    flow_spec,
                                       port_id_t                   id);
        int  (* flow_allocate_response)(struct ipcp_instance_data * data,
                                        struct ipcp_instance *      dest_usr_ipcp,
                                        port_id_t                   port_id,
                                        int                         result);
        int  (* flow_deallocate)(struct ipcp_instance_data * data,
                                 port_id_t                   id);

        int  (* application_register)(struct ipcp_instance_data *   data,
                                      const struct name *           source);
        int  (* application_unregister)(struct ipcp_instance_data * data,
                                        const struct name *         source);

        int  (* assign_to_dif)(struct ipcp_instance_data * data,
                               const struct dif_info *     information);

        int  (* update_dif_config)(struct ipcp_instance_data * data,
                                   const struct dif_config *   configuration);

        /* Takes the ownership of the passed SDU */
        int  (* sdu_write)(struct ipcp_instance_data * data,
                           port_id_t                   id,
```

```
                    struct sdu *                sdu);

cep_id_t (* connection_create)(struct ipcp_instance_data * data,
                               port_id_t                   port_id,
                               address_t                   source,
                               address_t                   dest,
                               qos_id_t                    qos_id,
                               struct conn_policies *      cp_params);

int      (* connection_update)(struct ipcp_instance_data * data,
                               struct ipcp_instance *      user_ipcp,
                               port_id_t                   port_id,
                               cep_id_t                    src_id,
                               cep_id_t                    dst_id);

int      (* connection_destroy)(struct ipcp_instance_data * data,
                                cep_id_t                    src_id);

cep_id_t
(* connection_create_arrived)(struct ipcp_instance_data * data,
                              struct ipcp_instance *      user_ipcp,
                              port_id_t                   port_id,
                              address_t                   source,
                              address_t                   dest,
                              qos_id_t                    qos_id,
                              cep_id_t                    dst_cep_id,
                              struct conn_policies *      cp_params);

int      (* flow_binding_ipcp)(struct ipcp_instance_data * user_data,
                               port_id_t                   port_id,
                               struct ipcp_instance *      n1_ipcp);

int      (* flow_unbinding_ipcp)(struct ipcp_instance_data * user_data,
                                 port_id_t                   port_id);

int      (* sdu_enqueue)(struct ipcp_instance_data * data,
                         port_id_t                   id,
                         struct sdu *                sdu);

/* Takes the ownership of the passed sdu */
int (* mgmt_sdu_write)(struct ipcp_instance_data * data,
                       address_t                   dst_addr,
                       port_id_t                   port_id,
                       struct sdu *                sdu);

/* Passes the ownership of the sdu_wpi */
int (* mgmt_sdu_read)(struct ipcp_instance_data * data,
                      struct sdu_wpi **           sdu_wpi);

/* Takes the ownership of the passed sdu */
int (* mgmt_sdu_post)(struct ipcp_instance_data * data,
                      port_id_t                   port_id,
                      struct sdu *                sdu);

int (* pft_add)(struct ipcp_instance_data * data,
                address_t                   address,
                qos_id_t                    qos_id,
                port_id_t *                 ports,
                size_t                      size);
```

```
        int (* pft_remove)(struct ipcp_instance_data * data,
                           address_t                   address,
                           qos_id_t                    qos_id,
                           port_id_t *                 ports,
                           size_t                      size);

        int (* pft_dump)(struct ipcp_instance_data * data,
                         struct list_head *          entries);

        int (* pft_flush)(struct ipcp_instance_data * data);

        const struct name * (* ipcp_name)(struct ipcp_instance_data * data);
        int (* enable_write)(struct ipcp_instance_data * data, port_id_t id);
        int (* disable_write)(struct ipcp_instance_data * data, port_id_t id);
};

struct ipcp_instance kfa_instance {
        struct kobject              kobj;
        struct ipcp_factory *       factory;
        struct ipcp_instance_data * data;
        struct ipcp_instance_ops *  ops;
};

struct ipcp_instance_data kfa_instance_data {
        struct kfa * kfa;
};

Static struct ipcp_instance_ops kfa_instance_ops = {
        .flow_allocate_request     = NULL,
        .flow_allocate_response    = NULL,
        .flow_deallocate           = NULL,
        .flow_binding_ipcp         = kfa_flow_ipcp_bind,
        .flow_unbinding_ipcp       = kfa_flow_deallocate,
        .application_register      = NULL,
        .application_unregister    = NULL,
        .assign_to_dif             = NULL,
        .update_dif_config         = NULL,
        .connection_create         = NULL,
        .connection_update         = NULL,
        .connection_destroy        = NULL,
        .connection_create_arrived = NULL,
        .sdu_enqueue               = kfa_sdu_post,
        .sdu_write                 = kfa_flow_sdu_write,
        .ipcp_name                 = kfa_name,
        .enable_write              = enable_write,
        .disable_write             = disable_write
};
```

8.3 Shim IPC Process over TCP/UDP configuration syntax

In the following, the shim IPC Process over TCP/UDP configuration syntax is expressed using EBNF.

```
<digit>           ::= 0..9
<non-zero-digit>  ::= 1..9
<non-zero-length> ::= <non-zero-digit><digit>*
```

```
<uint8>         ::= 0 .. 255

<length-and-string> ::= 0 | <non-zero-length> <string>
<length-and-IP> ::=
     0 | <non-zero-length> <uint8> . <uint8> . <uint8> . <uint8>

<uint16>          ::= 0 .. 65535
<non-zero-length> ::= 1 .. MAXUINT

<length-and-Port> ::= 0 | <non-zero-length> <uint16>
<length-and-AP>   ::= 0 | <non-zero-length> <string>
<length-and-AE>   ::= 0 | <non-zero-length> <string>

<uint>              ::= 0 .. MAXINT
<number-of-entries> ::= <uint>
```

8.3.1 dirEntry

The `dirEntry` format can be described with the following EBNF:

```
dirEntry      ::= <number-of-entries> {:<dirEntry-item> }*
dirEntry-item ::= <length-and-AP>
                  <length-and-AE>
                  <length-and-IP>
                  <length-and-Port>
```

Example of dirEntry:

```
2:11:test2.IRATI0:8:10.1.1.34:242711:test1.IRATI0:8:10.1.1.24:2426
```

The example describes the following mappings:

1. (AP = `test2.IRATI`, AE = `0`) ⟶ (IP = `10.1.1.34`, Port = `2427`) }
2. (AP = `test1.IRATI`, AE = `0`) ⟶ (IP = `10.1.1.24`, Port = `2426`) }

8.3.2 expReg

The `expReg` format can be described with the following EBNF:

```
expReg       ::= <number-of-entries>{:<expReg-item>}*
expReg-iterm ::= <length-and-AP><length-and-AE>
```

Example of expReg:

```
1:25:rina.apps.echotime.server0:4:2426
```

The example describes the following mapping:

- (AP = `rina.apps.echotime.server`, AE = `0`) ⟶ (Port = `2426`)

9 References

1. FP7 IRATI - D3.1 - First phase integrated RINA prototype over Ethernet for UNIX-like OS. Available online at http://irati.eu/wp-content/uploads/2012/07/IRATI-D3.1-v1.0.pdf
2. FP7 IRATI - D3.2 - Second phase integrated RINA prototype over Ethernet for a UNIX-like OS. Available online at http://irati.eu/wp-content/uploads/2012/07/IRATI-D3.2-v1.0.pdf.
3. FP7 IRATI - D3.3 - Second phase integrated RINA prototype for Hypervisors for a UNIX-like OS. Available online at http://irati.eu/wp-content/uploads/2012/07/IRATI-D3.3-bundle.zip.
4. FP7 IRATI - D2.4 - Third phase use cases, updated RINA specifications and high-level software architecture. Available online at http://irati.eu/wp-content/uploads/2012/07/IRATI-D2.4-bundle.zip
5. FP7 IRATI - D4.2 Deployment Experimentation and Validation Report for Phase 2. http://irati.eu/wp-content/uploads/2012/07/IRATI-D4.2.pdf
6. The (closed) IRATI software repository. Available upon request at https://github.com/dana-i2cat/irati
7. The (open) IRATI' stack site. Available online at http://irati.github.io/stack
8. The (open) IRATI wiki. Available online at https://github.com/IRATI/stack/wiki
9. The (open) IRATI Mailing List. Available online at http://www.freelists.org/list/irati
10. GitHub. Available online at http://github.com
11. Doxyen - Generate documentation from source code. Available online at http://doxygen.org
12. The IRATI open mailing list. Available online at http://www.freelists.org/list/irati
13. DocBook 5: The Definitive Guide. Available online at http://www.docbook.org/tdg5
14. The IRATI (closed) issue tracking tool. Available upon request at https://github.com/dana-i2cat/irati/issues
15. FP7 IRATI - D2.1 - First phase use cases, updated RINA specifications and high-level software architecture. Available online at http://irati.eu/wp-content/uploads/2012/07/IRATI-D2.1.pdf
16. The socket API reference. Available online at http://pubs.opengroup.org/onlinepubs/7908799/xns/syssocket.h.html

17. FP7 IRATI - D2.3 - Second phase use cases updated RINA specification and high-level software architecture (under EC evaluation). Available online at http://irati.eu/wp-content/uploads/2012/07/IRATI-D2.3.zip
18. The (open) IRATI software repository. Available upon request at https://github.com/IRATI/stack
19. The Git distributed version control system. Available online at http://git-scm.com
20. Atlassian Bitbucket. Available online at https://bitbucket.org
21. GitHub features. Available online at https://github.com/features
22. NAPI, the extension to the device driver packet processing framework. Available online at http://www.linuxfoundation.org/collaborate/workgroups/networking/napi
23. FP7 IRATI - D4.2 - Deployment Experimentation and Validation Report for Phase 1. http://irati.eu/wp-content/uploads/2012/07/IRATI-D4.1-final.pdf

END OF DOCUMENT