iPRP—The Parallel Redundancy Protocol for IP Networks: Protocol Design and Operation

Miroslav Popovic, Maaz Mohiuddin, Dan-Cristian Tomozei, and Jean-Yves Le Boudec, *Fellow, IEEE*

***Abstract*—Reliable packet delivery within stringent delay constraints is of paramount importance to mission-critical computer applications with hard real-time constraints. Because retransmission and coding techniques counteract the delay requirements, reliability may be achieved through replication over multiple fail-independent paths. The existing solutions, such as the parallel redundancy protocol (PRP), replicate all packets at the media access control layer over parallel paths. PRP works best in local area networks; however, it is not viable for IP networks that are a key element of emerging mission-critical systems. This limitation, coupled with diagnostic inability and lack of security, renders PRP unsuitable for reliable data delivery in these IP networks. To address this issue, we present a transport-layer solution: the IP parallel redundancy protocol (iPRP). Designing iPRP poses nontrivial challenges in the form of selective packet-replication, and soft-state and multicast support. iPRP replicates only time-critical unicast or multicast user datagram protocol traffic. iPRP requires no modifications to the existing monitoring application, end-device operating system, or to the intermediate network devices. It only requires a simple software installation on the end devices. iPRP has a set of diagnostic tools for network debugging. With our implementation of iPRP in Linux, we show that iPRP supports multiple flows with minimal processing-and-delay overhead. It is being installed in our campus smart-grid network and is publicly available.**

***Index Terms*—Availability, industrial communication, IP networks, redundancy, telecommunication network reliability, wide area networks.**

I. Introduction

SPECIFIC mission-critical computer applications have hard delay constraints. Failure to satisfy these constraints can result in economic losses or, even worse, human lives can be endangered in cases when these failures affect safety mechanisms. Notable examples of such applications (often built on top of cyber-physical systems) are process-control and emergency-management applications in the oil and gas industries [1], real-time detection of pollutants in the water/wastewater industry [2], vehicle-collision avoidance in car industry [3], automatic train control [4], process control in chemical industry [5], state estimation in smart grid [6], high-frequency trading [7], and distributed online gaming [8].

Manuscript received June 12, 2015; revised October 22, 2015 and January 24, 2016; accepted January 26, 2016. Date of publication February 12, 2016; date of current version October 31, 2016. Paper no. TII-15-0924.

The authors are with School of Computer and Communication Sciences - Laboratory for Communications and Applications (I&C-LCA2), École Polytechnique Fédérale de Lausanne, Lausanne 1015, Switzerland (e-mail: miroslav.popovic@epfl.ch).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TII.2016.2530018

Reliable and timely packet delivery, even in the order of 10 ms, is of utmost importance in satisfying the hard-delay constraints. The classic approaches to reliable communication through coding and retransmission are not compatible with the hard delay constraints. An alternative is to achieve reliability through replication over multiple fail-independent paths, which is the focus of this paper. More precisely, we present a solution for packet replication over multiple paths in IP networks. Indeed, as we discuss next, the existing solutions apply to media access control (MAC)-layer networks and cannot be transposed to IP networks that are a requirement for many of the aforementioned applications.

A. From MAC- to IP-Layer Parallel Redundancy Protocol

The parallel redundancy protocol (PRP) IEC standard [9] was proposed as a solution to packet replication over multiple fail-independent paths for local area networks (LANs) where there are no routers. Communicating devices need to be connected to two cloned (disjoint) bridged networks. The sender tags MAC frames with a sequence number and replicates them over its two interfaces. The receiver discards redundant frames based on sequence numbers.

In addition to extending PRP functionality to IP networks, the new design should also avoid the drawbacks of PRP. The most limiting feature of PRP is that the two cloned networks need to be composed of devices with identical MAC addresses. This contributes to making network management difficult. Furthermore, PRP duplicates all the traffic unselectively, which is acceptable for use in a local environment, but which cannot be done in general IP networks, because some links can be expensive and unnecessary traffic should be avoided. Moreover, PRP has no security mechanisms.

Note that, a PRP for IP networks needs to support IP multicast, as this is used in many of the aforementioned applications.

For a running example, we use the smart-grid communication network depicted in Fig. 1. Here, measurements are streamed periodically (every 20 ms for 50-Hz systems) by phasor measurement units (PMUs) to phasor data concentrators (PDCs), where the information about the electrical network state is expected in quasi-real time. PRP cannot be directly deployed here: devices are multihomed and each interface is assigned a different IP address. Most devices have two interfaces connected to a main network cloud made of two fail-independent network subclouds labeled *A* and *B*, while some have a third interface connected to a 4G cellular wireless service (labeled "Swisscom Long-term evolution (LTE) backbone" in this figure). It is assumed that paths between

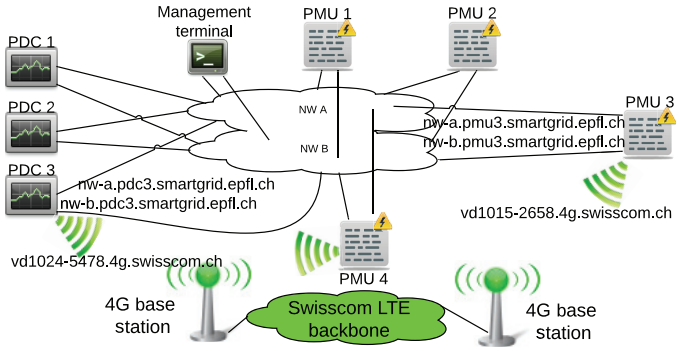


Fig. 1. Typical iPRP use-case in the context of smart grids. Devices PDCs and PMUs are connected to two overlapping network subclouds (labeled *A* and *B*). Some devices use an additional LTE connection providing a low-latency cellular service [10]. Every PMU streams data to all PDCs, using UDP and IP multicast.

interfaces connected to the A network subcloud stay within it (and similarly with B). The A and B network subclouds could be physically separated; however, in practice, they are most likely interconnected for network management reasons.

A simple way to achieve the arrangement described before is to divide the network into two logical subclouds A and B. Then, by adjusting the routing weights of the links interconnecting the A and B subclouds, we can ensure that $A \rightarrow A$ and $B \rightarrow B$ traffic stays within A and B subclouds, respectively, thereby giving rise to fail-independent paths. In such a setting, the interconnections will be used only for $A \leftrightarrow B$ traffic.

We need a solution that, similarly to PRP, takes advantage of network redundancy for increasing reliability and works in scenarios such as the one in Fig. 1. The existence of fail-independent paths is fundamental for the optimal operation of such a solution. However, in the event of a network-component failure, the paths can partially overlap. Then, the solution should reap the maximum possible benefits by operating in a degraded-redundancy mode. In other words, if complete end-to-end redundancy is no longer possible, the solution should continue to work.

In order for our solution to be easily deployed, we also require it to be transparent to both the application and network layers: it should only require installation at end devices and no modifications to running application software or to intermediary network devices (routers or bridges). In addition, real-time applications typically use user datagram protocol (UDP) rather than transmission control protocol (TCP) because TCP does not work with IP multicast or TCP retransmissions, the following packet losses, require several round-trip times and can be both detrimental and superfluous. Hence, we target a solution that supports UDP applications only.

In this paper, we present the design and implementation of iPRP (the IP PRP), a transport layer solution for transparent replication of unidirectional unicast or multicast UDP flows on multihomed devices.

B. iPRP

An iPRP host has to send different copies of the same packet over different paths. With the current technology, a device cannot control the path taken by an IP packet, beyond the choice of a destination address, exit interface and a type-of-service value. Other fields, such as the IPv6 flow label or source routing header extensions, are either ignored or rejected by routers. Also, the type-of-service field is used by applications and should not be tampered with by iPRP. Hence, we assume that a choice of the path is done at the sources by choosing communication interface and the destination address. The job of iPRP is then to transparently replicate packets over the different interfaces for the UDP flows that need it, match corresponding interfaces, remove duplicates at the receiver, and do this in a way that is resilient to crashes (see Section V-G).

Not all traffic requires replication, only certain devices and certain UDP flows do (time-critical data). Hence, replication needs to be selective: a failure-proof mechanism, transparent to applications, is required for detecting and managing packet replication. It needs to correctly match the interfaces, so that independent paths are used whenever they exist.

The iPRP protocol design is such that it does not interfere with the existing security mechanisms and does not introduce any new security weaknesses (see Section VI).

iPRP assumes that the network is traffic engineered; the critical UDP data streams receive enough resources and are not subject to congestion. iPRP instantly repairs packet losses due to failures or transient problems, such as transmission losses. It does not solve congestion problems due to underdimensioned network links. TCP flows are not affected.

Our iPRP implementation is for IPv6, as it is being installed in our smart-grid communication network (smartgrid.epfl.ch), that uses IPv6 (following the argument that new network environments should avoid future transition problems and embrace IPv6 from the start). Our implementation is available at http://goo.gl/N5wFNt. Adaptation to IPv4 is straightforward.

II. Related Work

As mentioned in Section I, iPRP overcomes the limitations of PRP [9]. The authors of [11] are aware of the fact that PRP is limited to LANs and suggest a direction for developing PRP in an IP environment. Their suggestion is neither fully designed nor implemented. Also, it requires that the intermediate routers preserve the PRP trailers at the MAC layer, which in turn requires changes in all of the routers in the networks. It does not address all the shortcomings of PRP (diagnostic tools, lack of multicast support, and need of special hardware). In contrast, our transport layer approach does not have these drawbacks.

Multipath TCP (MPTCP) [12] is used in multihomed hosts. It allows TCP flows to exploit the host's multiple interfaces, thus increasing the available bandwidth for the application. Like MPTCP, iPRP is a transport layer solution and is transparent to network and application. Unlike MPTCP, iPRP replicates the UDP packets on the parallel paths, while MPTCP sends one TCP segment on only one of them. In a case of loss, MPTCP resends the segment on the same path until enough evidence is gathered that this path is broken. So, a lost packet is repaired after several round-trip time (RTTs) (not good for time-critical flows).

Similarly, link aggregation control protocol (LACP) [13] and equal-cost multipath routing (ECMP) [14] require seconds for

failover. LACP enables the bundling of several physical links together to form a single logical channel. The failure of a link is discovered through the absence of keep-alive messages that are sent every 1–30 s. ECMP can be used together with most routing protocols in order to balance traffic over multiple best paths when there is a tie. In a case of failure, it relies on the reconfiguration of the underlying routing protocol that is commonly detected by the absence of keep-alive messages.

Network coding exploits network redundancy for increasing throughput [15], and requires intermediary nodes to recode packets (specialized network equipment needed). Also, it is not suitable for time-critical applications as typically packets are coded across "generations," which introduce decoding delays. Source coding (e.g., fountain codes [16]) can be useful for the bursty transmissions of several packets. However, it adds delay, as encoding and decoding are performed across several packets (not suitable for UDP flows with hard-delay constraints).

Multiprotocol-label-switching transport-profile (MPLS-TP) $1 + 1$ protection feature [17] performs packet duplication and feeds identical copies of the packets in working and protection path. On the receiver side, there exists a selector between the two; it performs a switchover based on some predetermined criteria. However, some time is needed for fault detection and signaling to take place, after which the switchover occurs. Hence, a 0-ms repair cannot be achieved.

Multitopology routing extends the existing routing protocols (e.g., [18]) and can be used to create disjoint paths in a single network. It does not solve the problem of transparent packet replication, but can serve as a complement to iPRP in the following way. On top of the underlying network (base topology), additional class-specific topologies can be created as a subset of base topology. We can use this feature to define fail-independent A and B subclouds in order to ensure fail-independent paths between sources and destinations.

Another method to ensure the discovery of fail-independent paths is software-defined networking (SDN) [19]. Centralized controller is aware of the overall network topology and can impose routing rules in a way that guarantees independent paths/trees between all the hosts.

III. A Top-Down View of iPRP

In this section, we first go over high-level design decisions we had to make during the development of the iPRP protocol and then succinctly describe the resulting design. iPRP aims to provide reliable end-to-end communication between multihomed hosts. The very first question that emerges is the choice of a TCP/IP layer where iPRP should be placed. Among others, iPRP needs to support scenarios where the traffic is carried over a shared network infrastructure, e.g., a telecom network operator provides connectivity for smart-grid services [20]. In this case, end users do not control intermediate routers. Hence, the routers should not be aware of the existence of iPRP. Consequently, we put network transparency as a requirement, which leads us to a solution that places iPRP above the network layer. Taking this into account, a possible solution can be to place iPRP at the application layer. This would imply that all legacy applications that are traditionally used would need to undergo changes in order to be compatible with iPRP protocol. Again, we opt for an application-transparent solution that leads us to a choice of a transport-layer solution.

The next choice to make was whether all the traffic originated at a sender should be replicated. Having in mind that bandwidth over IP-network links can be of limited capacity we opt here for a solution that replicates traffic selectively. The control of this feature is left to users through a simple configuration of the UDP port numbers that correspond to services that demand high reliability of packet delivery.

The next choice was that of a mechanism to inform a sender about the alternate IP addresses (unicast or multicast) of the receivers, so as to establish redundant paths. Classic solutions like PRP, use cloned address networks. Cloned IP addresses are not an option of iPRP as users do not necessarily control and manage all the interconnecting networks.We solve this problem by designing lightweight, secure, and crash-resilient signaling protocol. It also takes into account specificities of the multicast communication, e.g., it avoids flooding of senders with signaling messages from large groups of multicast receivers. It is a plug and play protocol initiated whenever a new sender emerges and completely transparent to the application layer.

Furthermore, we needed a mechanism for discard of duplicates, which can cope with packet reordering due to the network, and crash failures of the hosts. The existing duplicate-discard mechanisms such as the one used by PRP do not perform well under such packet-reordering and host failures. So, we also put in place, a stateless protocol for redundant packets removal.

The resulting design of iPRP is as follows. iPRP senders and receivers are expected to have multiple network interfaces. Applications are identified by the UDP port used to receive data. To enable iPRP for a certain application at the receiver, the port on which the application is listening needs to be added to the list of iPRP monitored ports (see Section IV-C). Receiving data on an iPRP monitored port automatically triggers an iPRP session between the sender and the receiver (see Section V). Within this session, the iPRP software running on the source host learns the receiver's network interfaces from the iPRP software running on the receiver. It uses the configured rules to match local interfaces to the receiver's remote ones. It then proceeds to capture the application's outgoing packets, encapsulates them in iPRP data messages addressed to the receiver's remote interfaces (according to the determined matching), and replicates them over the local interfaces. At the receiver, the iPRP software decapsulates the original packets, discards duplicates (Section V-E), and delivers them to the receiver application.

Hosts having multiple interfaces is not a strict requirement, but is desirable. In cases where the sender or the receiver have a single interface, iPRP still works, but the paths taken by the replicated packets join at a certain point, which becomes a single point of failure.

IV. Operation of iPRP

A. How to Use iPRP

iPRP is installed on end devices with multiple interfaces on: streaming devices (the ones that generate UDP flows with

hard-delay constraints) and receiving devices (the destinations for such flows).

Streaming devices (such as PMUs) do not require special configuration. Streaming applications running on such devices benefit from the increased reliability of iPRP without being aware of its existence. iPRP operates as a modification to the UDP layer.

On receiving devices, the only thing that needs to be configured is the set of UDP ports on which replication is required. For example, say that an application running on a PDC is listening on some UDP port for measurement data coming from PMUs. After iPRP is installed, this port needs to be added to the list of iPRP monitored ports in order to inform iPRP that any incoming flows targeting this port require replication. The application does not need to be stopped and is not aware of iPRP.

Nothing else needs to be done for iPRP to work. In particular, no special configuration is required for intermediary network equipment (routers and bridges).

B. General Operation: Requirements for Devices and Network

iPRP provides $1 + n$ redundancy. It increases, by packet replication, the reliability of UDP flows. It does not impact TCP flows.

iPRP-enabled receiving devices configure a set of UDP ports as *monitored*. When a UDP packet is received on any of the monitored ports, a one-way soft-state iPRP session is triggered between the sender and the receiver (or group of receivers, if multicast is used). Soft-state means that the state of the communication participants is refreshed periodically, and the entire iPRP design is such that a state-refresh message received after a cold start is sufficient to ensure proper operation. Consequently, the state is automatically restored after a crash, and devices can join or leave an iPRP session without impacting the other participants.

Within an iPRP session, each replicated packet is tagged with an iPRP header (Section V-D). It contains the same sequence number in all the copies of the same original packet. At the receiver, duplicate packets with the same sequence number are discarded (Section V-E). The original packet is reconstructed from the first received copy and forwarded to the application.

In multicast, all devices in the group of receivers need to run iPRP. If by mishap only part of the receivers support iPRP, these trigger the start of an iPRP session with the sender and benefit from iPRP; however, the others stop receiving data correctly. The use of source-specific multicast (SSM) is recommended (see [21]).

All iPRP-related information is encrypted and authenticated. The existing mechanisms for cryptographic key exchange are applied (security considerations in Section VI).

C. UDP Ports Affected by iPRP

iPRP requires two system UDP ports (transport layer) for its use: 1) the iPRP control port; and 2) the iPRP data port (in our implementation 1000 and 1001, respectively). The iPRP control port is used for exchanging messages that are part of the soft-state maintenance. The iPRP data port receives data messages of the established iPRP sessions. iPRP-capable devices always listen for iPRP control and data messages.

The set of monitored UDP ports, over which iPRP replication is desired are *not* reserved by iPRP and can be any UDP ports. UDP ports can be added to/removed at any time from this set during the iPRP operation. Reception of a UDP packet on a monitored port triggers the receiver to initiate an iPRP session. If the sender is iPRP capable, an iPRP session is started (replicated packets are sent to the iPRP data port), else regular communication continues.

D. Matching the Interconnected Interfaces of Different Devices

One of the design challenges of iPRP is determining an appropriate matching between the interfaces of senders and receivers, so that replication can occur over fail-independent paths. To understand the problem, consider Fig. 1 where the PMUs and PDCs have at least two interfaces. The A and B network subclouds are interconnected. However, the routing is designed such that, a flow originating at an interface connected to subcloud A with a destination in A, will stay in subcloud A. A potential problem can arise if a sender's interface, say SA, intended to be connected to the A subcloud, is mistakenly connected to the B subcloud, and vice versa. Then, one path from source to destination will go from SA (on subcloud B) to the destination interface DB (on subcloud B), and conversely to the other path. Following the routing rules, these flows will use interconnecting links between A and B subclouds. This is not desirable as these links can be of insufficient capacity because they are not intended to carry such traffic. Furthermore, it is no longer guaranteed that such paths are disjoint. PRP avoids this problem by requiring two physically separated and cloned networks. iPRP does not impose these restrictions. Hence, iPRP needs a mechanism to match interfaces connected to the same network subcloud.

To facilitate appropriate matching, each interface is associated with a 4-bit identifier called iPRP network subcloud discriminator (IND), which qualifies the network subcloud it is connected to. The iPRP software in end devices learns each of the interfaces' INDs automatically via simple preconfigured rules. Network routers have no notion of IND. A rule can use the interface's IP address or its DNS name. In our implementation, we compute each interface IND based on its fully qualified domain name. In Fig. 1, the rule in the iPRP configuration maps the regular expression `nw-a*` to the IND value `0xa`, `nw-b*` to IND `0xb`, and `*swisscom.ch` to IND `0xf`, respectively.

The receiver periodically advertises the IP addresses of its interfaces, along with their INDs to the sender (via `iPRP_CAP` messages). The sender compares the received INDs with its own interface INDs. Only those interfaces with matching INDs are allowed to communicate in iPRP mode. In our example, IND matching prevents iPRP to send data from a PMU A interface to a PDC B interface. Moreover, each iPRP data packet contains the IND of the network subcloud where the packet is supposed to transit (see Section V-D). This eases the monitoring

Algorithm 1. (At the receiver) Soft-state maintenance (keeps the list of active senders up-to-date)

```
while true do
    remove inactive hosts from the list of active senders (last-seen timer expired);
    for every packet received on one of the monitored ports or on iPRP Data Port do
        if the source is in the list of the active senders then
            update associated last-seen timer;
        else
            put sender in the list of active senders;
        end
    end
end
```

and debugging of the whole network. It allows us to detect misconfiguration errors that cause a packet expected on an A interface to arrive on a B interface.

V. Protocol Description

The iPRP message exchange is divided into two planes: 1) control; and 2) data planes. The control plane is responsible for exchange of messages required to establish and maintain an iPRP session. The data plane is responsible for replication and deduplication of time-critical UDP flows. Note that, control plane messaging is nontime critical and far less frequent than data plane (data plane—ms and control plane—s) .

The data plane operation is divided into two phases: 1) replication phase; and 2) duplicate-discard phase. Next, we discuss the operation of each plane and the description of key elements of the iPRP protocol in detail.

A. Control Plane

The control plane is used for exchange of messages to establish and maintain an iPRP session. The iPRP session establishment is triggered when a UDP packet is received at some monitored UDP port p. In Fig. 2, UDP port p is made monitored at t_1 at the receiver, by adding it to the list of monitored ports. This triggers the establishment of an iPRP session, i.e., the receiver's soft-state-maintenance functional block (Fig. 3) adds the sender to the list of active senders (Algorithm 1).

The iPRP-capability-advertisement functional block (Fig. 3) at the receiver, sends `iPRP_CAP` to the control port of the sender every T_{CAP} seconds (t_2 in Fig. 2, Algorithm 2). This message informs the sender that the receiver is iPRP enabled and provides information required for selective replication over alternative paths. It contains the iPRP version; INDs of the network subclouds to which the receiver is connected to facilitate IND matching (see Section IV-D); the source and destination UDP port numbers of the packet that triggered the iPRP-session establishment; in multicast, the multicast IP address of the group; in unicast, IP addresses of all receiver interfaces; and

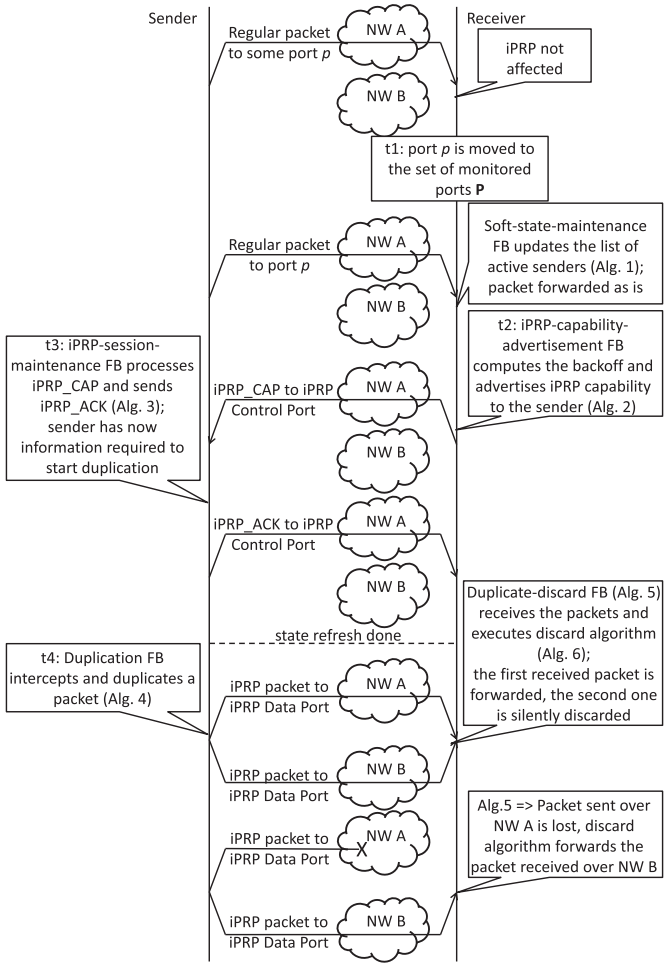


Fig. 2. Message sequence chart for typical scenario when iPRP-capable devices are starting iPRP operation.

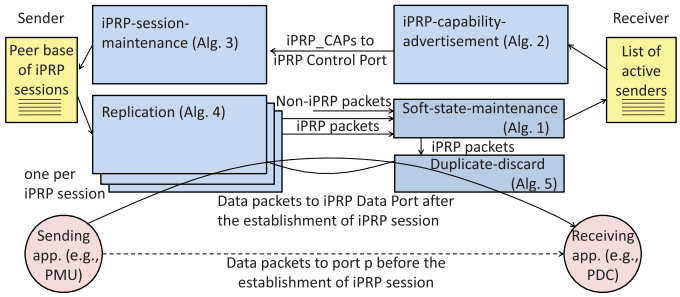


Fig. 3. Overview of the functional blocks.

a symmetric, short-lived cryptographic key for authentication and encryption of the iPRP header (Section VI).

On receiving the `iPRP_CAP`, the iPRP-session-maintenance functional block (Fig. 3) at the sender acknowledges it with an `iPRP_ACK`. The `iPRP_ACK` contains the list of sender IP addresses that are used by the receiver to subscribe to alternate network subclouds to receive data through SSM. In multicast, the receivers send `iPRP_CAP` after a back-off period (Section V-F) to avoid flooding. The `iPRP_ACK` message also serves as a terminating message for impending `iPRP_CAPs`, thereby preventing a flood (Algorithm 2).

Algorithm 2. (At the receiver) iPRP capability advertisement

```
1  while true do
2  |  compute T_backoff (Section V-F);
3  |  listen for iPRP_ACKs until T_backoff expires;
4  |  send iPRP_CAP messages to all hosts in the list of active
   |  senders from which no iPRP_ACKs are received;
5  |  sleep T_CAP − T_backoff;
6  end
```

Algorithm 3. (At the sender) iPRP session maintenance

```
1  while true do
2  |  remove aged entries from the peer-base;
3  |  for every received iPRP_CAP message do
4  |  |  if there is no iPRP session established with the
   |  |  destination then
5  |  |  |  if IND matching is successful then
6  |  |  |  |  establish iPRP session by creating new entry
   |  |  |  |  in the peer-base;
7  |  |  |  |  send iPRP_ACK message;
8  |  |  |  end
9  |  |  else
10 |  |  |  update the keep-alive timer;
11 |  |  end
12 |  end
13 end
```

To complete the iPRP session establishment, the iPRP-session-maintenance functional block performs IND matching (Section IV-D) and creates a peer-base entry (t_3 in Fig. 2, Algorithm 3). The peer-base contains all information needed by the sender for replication of data packets.

The second goal of control plane it to maintain an iPRP session. To this end, the iPRP_CAP messages are used as keep-alive messages (Algorithm 3). The iPRP session is terminated if no iPRP_CAP message is received for a period of $3T_{\mathrm{CAP}}$. These messages are sent to a sender as long as it is present in the list of active senders. The list of active senders is maintained by the soft-state-maintenance functional block by updating the last-seen timer (Algorithm 1) when a new data packet is received. Sessions that are inactive for more than $T_{\mathrm{inactivity}}$ are terminated.

For each new iPRP session, a corresponding iPRP session establishment is triggered. If any of the required steps could not be completed due to message loss or to an iPRP incapability, an iPRP session is not established and packets are not replicated.

Addition or removal of new interfaces at the sender or receiver is communicated by the iPRP_CAP messages and the peer base is updated accordingly. Specifically, when an iPRP_CAP is received for already established iPRP sessions, the peer base is updated in the following ways. Newly received INDs, which are successfully matched, are added to the peer base. On the contrary, INDs in the peer base that cannot be matched with any of the received INDs, are removed from the peer base after confirmation from multiple consecutive iPRP_CAPs (to handle the effect of the backoff algorithm in Section V-F).

Algorithm 4. (At the sender) Packet replication

```
1  for every outgoing packet do
2  |  check the peer-base;
3  |  if there exists an iPRP session that corresponds to the
   |  destination socket then
4  |  |  replicate the payload;
5  |  |  append iPRP headers incl. seq. number;
6  |  |  send packet copies;
7  |  else
8  |  |  forward the packet unchanged;
9  |  end
10 end
```

B. Data Plane: Replication Phase

The replication phase occurs at the sender to send out data plane messages once the iPRP session is established. The replication functional block (Fig. 3) on the sender intercepts all outgoing packets destined to UDP port p of the receiver. These packets are subsequently replicated and iPRP headers (Section V-D) are prepended to each copy of the payload. iPRP headers are populated with the iPRP version, a sequence-number-space ID (SNSID—unique identifier of an iPRP session), a sequence number, an original UDP destination port, and IND. The 32-bit sequence number is the same for all the copies of the same packet. The destination port number is set to the iPRP data port for all the copies. An authentication hash is appended and the whole block is encrypted. Finally, the copies are transmitted as iPRP data messages over the different matched interfaces (see Algorithm 4, t_4 in Fig. 2).

C. Data Plane: Duplicate-Discard Phase

The duplicate-discard phase occurs at the receiver, once an iPRP session is established to ensure that only one copy of replicated packets is forwarded to the application. Upon reception of packets on the iPRP data port, the associated last-seen timer is updated (see Algorithm 1) and the packets are forwarded to the duplicate-discard functional block (Algorithm 5). It decrypts the iPRP header at the beginning of the payload using the symmetric key used in iPRP_CAP message. Then, function isFreshPacket (Section V-E - Algorithm 6) is called. Based on the sequence-number-space ID (SNSID—unique identifier of an iPRP session) and the sequence number, the packet is either forwarded to the application or discarded. The first received copy should reach the application, subsequent copies are discarded. The replication is thus rendered transparent to the sender and receiver applications. In Fig. 2, we show two scenarios after the time t_4; in one case, both copies are delivered, in the other, one packet is lost.

D. iPRP Header

Fig. 4 shows the position and the fields of the iPRP header used in data packets. The SNSID is used to identify an iPRP session. This identifier is unique across all iPRP sessions terminating at the same receiver, thereby allowing multiple iPRP

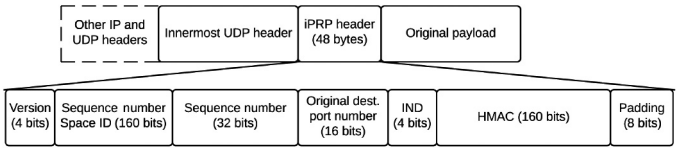


Fig. 4. Location and fields of the iPRP header.

sessions on the same machine. In our implementation, it is chosen as a concatenation of the source IPv6 address, the source UDP port number of the socket to which the application writes the packet and a 16-bit reboot counter.

The SNSID is used by a receiver to tie the packets with different source IP addresses that belong to the same iPRP session. When a new receiver joins a multicast group with an already established iPRP session, it uses the source IP address in the SNSID to uniquely identify the sender of the packets and the source port number in the SNSID to uniquely identify the streaming application on the sender. However, in case of a crash and reboot of the sender, the sequence number is reset. Then, a new reboot counter in the iPRP header differentiates packets belonging to the new iPRP session from those of the old iPRP session, thereby ensuring a seamless recovery at the receiver.

To maintain the format of the iPRP header for an IPv4 implementation, we suggest repeating source IPv4 address four times at the place of source IPv6 address. The original destination UDP port number is included to allow for the reconstruction of the original UDP header. The iPRP header is placed after the inner-most UDP header. So, iPRP works well, even when tunneling is used (e.g., 6–4).

Like many protocols (such as DTLS, VPN, VXLAN, and 4in6), iPRP adds its own header to the packet payload. In order to avoid packet fragmentation, we adopt the same solution as any tunneling protocol: at the sender, iPRP reduces the interface MTU size to the minimum of 1280 bytes required by IPv6. In practice, typical MTU values are closer to the IPv6-recommended 1500 bytes. This leaves a margin for the inclusion of the iPRP and other tunneling protocol headers.

E. Discard Algorithm

The redundant copies of a packet are eliminated by a discard algorithm running at the receiver. In scenarios where the packets are received out-of-order, the discard algorithm proposed for PRP [22] delivers several copies of the same packet to the application. The function `isFreshPacket` (Algorithm 6) avoids this issue. It is used by Algorithm 5 to decide if a packet sequence number corresponds to a fresh packet. We use 32-bit unsigned integer sequence numbers, large enough to avoid the wrap-around problem.

Algorithm 6 tracks the following variables per iPRP session, identified by a sequence number space ID (`SNSID`):

1) `HighSN`—highest sequence number of a packet received before the current packet;
2) `ListSN`—sequence-number list of delayed packets.

`ListSN` is bounded to a maximum of $\texttt{MaxLost} < 2^{31}$ entries. `MaxLost` is the maximum sequence-number difference accepted by the application. In practice, we can take $\texttt{MaxLost} > R \times T_{\text{late}}$, where R is an upper bound on packet rate of the streaming application that corresponds to an iPRP session and T_{late} is the time after which packets are deemed out-of-date, thus irrelevant. Consequently, if a packet is received with a sequence number that precedes `HighSN` by more than `MaxLost`, it is deemed "very late" and dropped.

The value of `MaxLost` is configurable and depends on the targeted application. For example, in our smart-grid setting, there is a hard delay constraint of 20 ms (any packet older than this can be safely discarded). To be conservative, we allow packets with the delays of up to $T_{\text{late}} = 50$ ms. We set `MaxLost` to 1024, high enough to support any realistic PMU streaming rate.

Algorithm 5. (At the receiver) Duplicate discard

```
1  for every packet received on iPRP data port do
2      get sequence number space ID (SNSID);
3      get sequence number (SN);
4      if it is the first packet from this SNSID then
5          SNSID.HighSN ← SN                          // Bootstrap
6          remove iPRP header;
7          reconstruct original packet;
8          forward to application;
9      else
10         if isFreshPacket(SN, SNSID) then
11             remove iPRP header;
12             reconstruct original packet;
13             forward to application;
14         else
15             silently discard the packet;
16         end
17     end
18 end
```

Algorithm 6. Function to determine whether a packet with sequence number `CurrSN` corresponds to a fresh packet in the sequence number space ID `SNSID`. The test "x follows y" is performed for 32-bit unsigned integers using subtraction without borrowing as "$(\texttt{x} - \texttt{y}) >> 31 == 0$".

```
1  function isFreshPacket(CurrSN, SNSID)
2  if CurrSN == SNSID.HighSN then
3      return false;                          // Duplicate packet
4  else if CurrSN follows SNSID.HighSN then
5      put SNs [SNSID.HighSN+1, CurrSN-1] in SNSID.
       ListSN;
6      remove the smallest SNs until SNSID.ListSN has
       MaxLost entries;
7      SNSID.HighSN ← CurrSN                  // Fresh packet
8      returntrue;
9  else
10     if CurrSN is in SNSID.ListSN then
11         remove CurrSN from SNSID.ListSN;
12         return true;                       // Late packet
13     else
14         return false       // Already seen or very late
15     end
16 end
```

iPRP and its discard algorithm are able to recover after unexpected events (crashes and reboots). A problem can occur if, after a reboot of a sender, the same sequence numbers are reused. Then, fresh packets can be wrongly discarded as the receiver would be deceived into believing that it had already delivered such packets. This problem can be fixed by imposing handshakes between senders and receivers. However, such a solution is not appropriate if multicast is used and, furthermore, it would violate soft-state property. Our solution is to have a sender maintain a reboot counter that defines different sequence-number spaces within the same sender machine (see Section V-D). Therefore, when a new reboot counter is encountered, the receiver creates a new `SNSID`, thereby resetting `HighSN`. Following a reboot of a receiver, all the receiver's counters are initialized upon the reception of the first iPRP data packet.

As mentioned earlier, the algorithm keeps track of one variable and one list per iPRP session. The most expensive operation is searching the list (line 10). However, in practice, `ListSN` is limited to few entries. The algorithm can be further optimized for a $\mathcal{O}(1)$ time complexity by using a hash table implementation for `ListSN`. Additionally, the algorithm is designed to have a fixed memory usage: size(`ListSN`) bytes.

Before stating the correctness of the algorithm, we need to introduce some definitions. We say that a received packet is *valid* if it arrives in order or if it is out-of-order but not later than T_{late}. Formally, this means that a packet received at time t with $\texttt{SN} = \alpha$ is not valid if some packet with $\texttt{SN} = \beta > \alpha + \texttt{MaxLost}$ was received before t.

Furthermore, let Δ be an upper bound on the delay jitter across all network subclouds. Formally, for any two packets i, j sent over any two network subclouds k, l: $\Delta \geq \left(\delta_i^k - \delta_j^l\right)$, where δ denotes the one-way network delay. Also, recall that $T_{\text{inactivity}}$ is used to terminate inactive sessions (Section V-A).

Theorem 1 (Correctness of the discard algorithm): If $R \times \Delta < 2^{31}$ and $R \times (T_{\text{inactivity}} + \Delta) < 2^{31}$, then Algorithm 6 guarantees that: 1) no duplicates are forwarded to the application; and 2) the first received valid copy of any original packet is forwarded to the application.

The proof is lengthy and is given in the appendix. To understand the practicality of the conditions in the theorem, note that, $T_{\text{inactivity}}$ is in the order of seconds and is much larger than Δ. Therefore, the only condition to verify is $R \times (T_{\text{inactivity}} + \Delta) < 2^{31}$, which for, say $T_{\text{inactivity}} = 10$ s and $\Delta = 100$ ms, requires $R < 2 \times 10^8$ packets/s (pps)—a rate much higher than ever expected.

F. Backoff Algorithm

The soft state in a multicast iPRP session is maintained by periodic advertisements (`iPRP_CAP`) sent to the source by each member in the multicast group of receivers. We want to prevent "message implosion" at the source for groups of receivers ranging from several hosts to millions. Failing to do so can have a similar effect as a denial-of-service attack. The source would be overwhelmed with processing `iPRP_CAPs` if all the multicast group members would send them. Nevertheless, if the source waits too long before receiving at least one `iPRP_CAP`, the start of the iPRP operation would be delayed. This is why, we also require the source to receive an `iPRP_CAP` within at most $D = 10$ s after the start of the loop in Algorithm 2 (executed periodically every $T_{\text{CAP}} = 30$ s) .

A similar problem was studied in the literature on reliable multicast, where ACK implosion at the source needs to be avoided. To the best of the authors' knowledge, the solution that best fits our scenario was proposed by Nonnenmacher and Biersack [23]. We adopt it in our design: each receiver performs a random backoff before transmitting an `iPRP_CAP`. The source acknowledges each `iPRP_CAP` by an `iPRP_ACK`. The reception of an `iPRP_ACK` before the expiry of the backoff timer inhibits any receiver from sending its `iPRP_CAP`. The backoff timer follows a flipped truncated exponential distribution (inaptly called "exponential" in [23]), defined by a PDF on $[0, D]$ that increases toward D, $f_X(x; \lambda, D) \stackrel{\text{def.}}{=} \lambda e^{\lambda x} (e^{\lambda D} - 1)^{-1} \cdot \mathbb{1}_{\{x \in [0,D]\}}$.

Due to the back-off algorithm, a multicast iPRP sender may not receive `iPRP_CAPs` from the same receiver in two consecutive cycles. As different receivers can have interfaces with different INDs' active, the consecutive `iPRP_CAPs` seen by the sender can have different INDs. In cases when an `iPRP_CAP` is received from a receiver with fewer interfaces, if the missing INDs would be immediately removed from the sender's peer base, the replication on these network subclouds would be adversely affected. To mitigate this problem, removal of INDs from the peer base is done only after confirmation from multiple consecutive `iPRP_CAPs` (see Section V-A).

We implement the backoff computation of [23] by CDF inversion. A uniform random variable $U \in [0, 1]$ is obtained via a random number generator. Next, the backoff is set to $T_{\text{backoff}} = \lambda^{-1} \ln(1 + (e^{\lambda D} - 1)U)$ (Algorithm 2, line 2). We pick $\lambda = 25/D$ (see [21] for further discussion).

G. Robustness and Soft State

iPRP is a soft-state protocol that is robust against host failures and supports joining or leaving the hosts from the network at any time, independently of each other. In a multicast case, it is expected that a new iPRP-capable receiver can show up (or simply crash and reboot) after an iPRP session with other receivers was established. Then, the new receiver will immediately be able to process packets received at the iPRP data port without the need to exchange control messages.

The iPRP control-message exchange does not rely on the availability of any particular network subcloud, making our protocol robust to network failures. Once the soft-state maintenance functional block learns about alternative network subclouds, `iPRP_CAP` messages are sent over all of them. Furthermore, the control plane communication to the reserved iPRP control port is secured (see Section VI). The security algorithm for iPRP header protection can be chosen as a part of the configuration.

VI. Security Considerations

The iPRP protocol design is such that it does not interfere with upper-layer security protocols. However, in addition, we needed to provide security for the iPRP header itself, as

there are attacks that can stay undetected by upper-layer security protocols. Concretely, if an attacker manages to alter the sequence-number field of iPRP packets transmitted over one (compromised) network subcloud, the discard algorithm can be tricked in a way that the packets from both (compromised and noncompromised) network subclouds are discarded. Note that, similar attacks exist for PRP, where an attacker, with access to one network, can force the discard of valid frames on another network. For example, say an attacker has access to network subcloud A. A PRP frame is represented as $A5$, where A is the network subcloud it belongs to and 5 is the sequence number. If $A5$ and $B5$ were received and the attacker retransmits the frame $A5$ by altering the sequence number as $A6$, then the actual $A6$ and $B6$ frames will both be discarded. In other words, an unsecured PRP or iPRP could weaken the network instead of making it more robust. Yet another argument for protecting the iPRP protocol is that by doing so, we minimize the exposure for prospective attacks in the future.

The iPRP control messages are encrypted and authenticated. This guarantees that the security of replicated UDP flows is not compromised by iPRP and that it does not interfere with application layer encryption/authentication.

Specifically, `iPRP_CAP` messages and the corresponding `iPRP_ACK` messages are transmitted over a secure channel. The iPRP header inserted in the data packets is authenticated and encrypted with a preshared key. Thus, replay attacks and forged messages insertion are avoided.

We establish the secure channel for the transmission of `iPRP_CAP` messages depending on the type of communication, unicast, or multicast. Details are as follows.

Unicast: In unicast mode, a datagram transport layer security (DTLS) [24] session is maintained between the sender and the receiver. It is initiated by the receiver upon the arrival of the first UDP datagram from the source. `iPRP_CAP` messages are transmitted within this session. So, the iPRP capabilities of the receiver are transmitted only to an authenticated source. `iPRP_ACK`s are not required in unicast (since message implosion can occur in multicast only).

Unicast `iPRP_CAP` messages contain a symmetric key used to authenticate and encrypt the iPRP header. This key is updated periodically during a unicast iPRP session. Hosts keep a small fixed number of valid past keys to prevent losing the iPRP session because of delayed reception of a new key. The oldest key is discarded upon reception of a new one.

Multicast: iPRP relies on any primitive that establishes a secure channel with the multicast group. For example, multicast security (MSEC) [25] can be used for group key management and for establishing a group security association.

In this setting, both `iPRP_CAP` and `iPRP_ACK` messages, as well as the iPRP headers inserted in the replicated packets, are authenticated and encrypted with the group key. Thus, there is no need to include an additional key in the `iPRP_CAP`.

VII. iPRP Diagnostic Toolkit

As iPRP is designed to be IP friendly, it facilitates the exploitation of the diagnostic utilities associated with TCP/IP. The diagnostics include verification of connectivity between hosts and the evaluation of the corresponding RTTs (similar to `ping`), the discovery of routes to a host (similar to `traceroute`), etc. Furthermore, the toolkit also adds some more tools that are specific to iPRP and it gives iPRP a significant edge in network diagnostics and statistics collection over PRP. The toolkit comprises the following tools:

```
iPRPtest <Remote IP Address><Port>
            <Number of packets> <Time period>
iPRPping <Remote IP Address>
iPRPtracert <Remote IP Address>
iPRPsenderStats <IP Address>
iPRPreceiverStats <IP Address>.
```

Imagine a typical scenario where an application on an iPRP-enabled host that is subscribed to a particular multicast group (G) experiences packet losses. To troubleshoot this problem, the user at the receiving host would use the `iPRPreceiverStats` tools to consult the local list of active senders, to check for the presence of an iPRP session associated with any host sending multicast data to group G. If an iPRP session exists, then the tool returns the statistics of packets received over different networks in the iPRP session. Then, to understand if the problem is caused by multicast routing or lossy links, the user moves to the sending host.

First, with `iPRPtest` and by using the remote IP address of the receiver, the user establishes a temporary, unicast iPRP session with the host. If successful, the `iPRPping` tool is used to obtain the packet loss and RTT statistics over the multiple networks. Also, the `iPRPtracert` tool is used to verify the hop-by-hop UDP data delivery over multiple networks. For any iPRP session between two hosts, the `iPRPsenderStats` is used by the sending host to query the remote host about the statistics of the packets accepted and dropped by the duplicate-discard functional block on that remote host. The operation of each tool is described in detail in [21].

VIII. Implementation

We opted for a Linux-based user-space implementation that has the following properties.

1) Enable the selective filtering of IP packets, so that the iPRP sequence of operation can be applied.
2) Allow for packet mangling where the iPRP header can be inserted and packets can be replicated at the sender and duplicates can be discarded and original packet can be restored at the receiver.
3) Minimal CPU overhead.

To this end, we use the `libnetfilter_queue` (NF_QUEUE) framework from the Linux iptables project. NF_QUEUE is a userspace library that provides a handle to packets queued by the kernel packet filter. It requires the `libnfnetlink` library and a kernel that includes the `nfnetlink_queue` subsystem (kernel 2.6.14 or later). It supports all Linux kernel versions above 2.6.14. We use the Linux kernel 3.11 with iptables-1.4.12.

The main challenge encountered in the implementation was to ensure that the delay and processing overhead was low. For this purpose, we categorized the various instructions in iPRP

TABLE I
SPECIFICATIONS OF HOSTS USED IN THE TEST BED

Component	Version specifications
CPU	Intel C2Duo, 2.53 Ghz
Ethernet controller	Intel 82567LM Gigabit Card
Wireless controller	Intel Wifi N d5300
Operating system	Ubuntu 12.04 LTS, 64-bit
Kernel	3.6.11-rt29 (RT-Linux Patch)

as time critical or nontime critical. For instance, adding the iPRP header to a packet is time critical, whereas updating the peer base to enable replication on a new network is nontime critical. Then, we used batching of nontime-critical instructions to reduce the total number of system calls. In this way, we achieved a lower overhead while maintaining the same real-time performance.

Currently, we are deploying iPRP on our EPFL smart-grid communication network (smartgrid.epfl.ch).

IX. PERFORMANCE EVALUATION

In order to evaluate the performance of our implementation, we have setup a lab test-bed and do two types of assessments. The first one is to evaluate the operation of iPRP and its discard algorithm in different scenarios. The latter set of experiments is to assess the processing delay due to iPRP and the additional CPU usage used by iPRP software of our proof-of-concept implementation. Our test bed consists of two Lenovo ThinkPad T400 laptops with a 64-bit Ubuntu OS. The laptops (Table I) are connected over two Ethernet-based wired networks (one via USB adapter) and one ad-hoc Wi-Fi network. We label the interface eth0 as nwA, eth1 as nwB and wlan0 as nwC. To evaluate the real-time operation, we patched the Linux kernel 3.11.6 with the associated real-time Linux patch *rt29*.

A. iPRP Behavior in the Presence of Asymmetric Delays and Packet Losses

Our goal here is to validate the design and implementation of iPRP by quantifying the packet losses and delays perceived by an application. We stress test the discard algorithm with heavy losses and asymmetric delays and compare the performance with that in theory. The packet losses and delays are emulated using the Linux `tc-netem` [26] tool on the test bed described in Table I.

In Table II, we summarize the settings used in different scenarios. To mimic the traffic created by PMUs, we send a 280-byte UDP datagram every 20 ms, long enough to have stationary behavior. We emulate delays that are uniformly distributed within $10\ \text{ms} \pm 5\ \text{ms}$ (small differences in network topologies and/or loads), and within $1\ \text{s} \pm 0.2\ \text{s}$ (significant differences in network topologies or serious perturbations in network functioning). We emulate both independent and bursty losses. In both cases, the overall packets' loss rate is 5%. To produce 5% bursty losses with `tc-netem`, we use Gilbert–Elliot model [27] with $p = 0.01$, $r = 0.19$, $1 - k = 0.01$, and $1 - h = 0.81$, where p and r are the transition probabilities between the bad and the good states, $1 - k$ is the loss probability in the good state, and $1 - h$ is the loss probability in the bad state.

TABLE II
SCENARIOS USED FOR PERFORMANCE EVALUATION

Scenario	`tc-netem` delay : loss nature		
	nwA	nwB	nwC
0	S:IL	S:IL	S:IL
1	Z:IL	S:IL	Not used
2	Z:BL	S:BL	Not used
3	Z:IL	L:IL	Not used
4	Z:BL	L:BL	Not used
5	S:IL	S:IL	Not used

[a]Scenarios used for performance evaluation. $\texttt{tc} - \texttt{netem}$ added delay: "Z" means 0, "S" means small uniform $10\ \text{ms} \pm 5\ \text{ms}$, and "L" means large uniform $1\ \text{s} \pm 0.2\ \text{s}$. Loss nature: "IL" means 5% independent and "BL" means 5% bursty losses.

TABLE III
LOSS PERCENTAGES IN VARIOUS SCENARIOS

Scen.	nwA	nwB	nwC	iPRP	theory
0	5.061	4.913	5.1537	0.0126	0.0128
1	5.057	5.002	Not used	0.253	0.254
2	5.132	5.059	Not used	0.259	0.254
3	5.014	5.013	Not used	0.251	0.249
4	5.022	4.981	Not used	0.247	0.249
5	5.051	5.002	Not used	0.251	0.253

We use Scenario 0 to evaluate the operation of iPRP in the presence of more than two networks. In Scenarios 1–4, we test the discard algorithm by making asymmetric delays and losses, thus forcing it to keep track of delayed/missing packets. With Scenario 5, we test the expected iPRP side benefit of having lower average one-way network delay, given that the iPRP duplicate-discard functional block always forward the first packet delivered over any of the available networks. We measure delays and losses over individual networks and as experienced by an application located on top of iPRP.

In Table III, we show the measurement results. We assume that the losses on different networks are independent. Under this assumption, the expected actual loss percentage can be approximated with the product of observed loss percentages on different networks. We compare the observed actual losses (iPRP column) with the expected actual loss percentage (theory column). A deviation would mean anomalies in the iPRP protocol and implementation. The accordance between the last two columns in Table III shows that iPRP performs as expected in significantly reducing the actual packet losses.

In Fig. 5, we show the CDF of one-way network delay for a specific packet (d_{iPRP}) for Scenario 5. In theory, it should be $d_{\text{iPRP}} = \min(d_{\text{nwA}}, d_{\text{nwB}})$, where $d_{\text{nwA}}, d_{\text{nwB}}$ are the one-way network delays of the same packet on network subclouds A and B. What we measured matches the theory very well. This is a confirmation of the anticipated side benefit of iPRP: the delays perceived by the application are improved when iPRP is used, compared to those when only one of the individual networks is used.

CDFs are not shown for Scenarios 1–4 as, by construction, it is almost deterministic which network has the shortest delay. For example, in Scenario 1 most of the times $d_{\text{iPRP}} = \min(d_{\text{nwA}}, d_{\text{nwB}}) = d_{\text{nwA}}$.

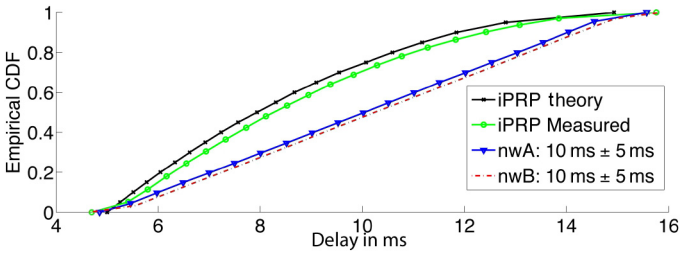


Fig. 5. iPRP side benefit: The delays perceived by the application are improved when iPRP is used, compared to those when only one of the individual networks is used.

TABLE IV
CPU USAGE WITH IPRP AND VARYING LOADS

Number of sessions	Exper. 1: Aggregate of pps for all iPRP sessions kept constant to 1000		Exper. 2: pps per iPRP session kept constant to 10	
	Send. (%)	Rec. (%)	Send. (%)	Rec. (%)
0 (Idle)	3.7	0.9	3.7	0.9
1	14.5	11.8	4.5	2.2
2	14.1	11.9	5.6	2.4
4	15	11.3	5.7	2.3
10	15	12	7.3	3.2
20	15	12	10	5.2

B. Processing Overhead Caused by iPRP

In this section, we evaluate processing delays and the additional CPU load when iPRP is used on the test bed described in Table I. We conduct several runs of Scenario 1 (see Table II) and use GNU gprof [28] to assess the average processing delay incurred by an iPRP data packet at the iPRP sender daemon (ISD) and the iPRP receiver daemon (IRD). In an ISD, a data packet encoutners only the replicator function that adds the iPRP header and replicates packets over multiple interfaces. This operation takes 0.8 μs on average. In an IRD, a data packet encounters three functions. The packet handler copies a packet into user space, verifies the fields of the iPRP header, and prepares a packet for the duplicate-discard function, which indicates if a packet is to be dropped or forwarded. These operations take 0.8 and 0.4 μs on average, respectively. Finally, if a packet is to be forwarded, the iPRP header is removed and checksum is recomputed in 2.4 μs. On average, a data packet incurs a delay overhead of 4.4 μs due to iPRP.

In order to assess the additional CPU load when iPRP is used, we perform two experiments in which we record the CPU usage by iPRP daemons on the sender and on the receiver. The results are summarized in Table IV. In Experiment 1, we keep constant aggregate packet rate of 1000 pps for all established iPRP sessions (1 iPRP session of 1000 pps, 2 iPRP sessions of 500 pps each, etc.). The CPU usage with iPRP is quasi-constant at 15% for the sender and 12% for the receiver. In Experiment 2, we keep a constant packet rate of 10 pps for every individual iPRP session (1 iPRP session—10 pps in total, 2 iPRP sessions—20 pps in total, etc.). At the sender, the CPU usage increases, at first, at a rate of 0.9% per iPRP session and the increase rate per iPRP session decreases to 0.32% for a larger number of iPRP sessions. At the receiver, the increase rate of CPU usage per each additional iPRP session goes from 0.8% to 0.22%.

Following the results from Table IV, the additional % of CPU usage at sender (U_s) and receiver (U_r) due to iPRP can be approximated with the relations

$$U_s = 3.7 + 0.28 \times (\#\text{ of iPRP sessions}) + 0.01 \times (\text{packets/s}) \quad (1)$$

$$U_r = 0.9 + 0.08 \times (\#\text{ of iPRP sessions}) + 0.01 \times (\text{packets/s}). \quad (2)$$

X. CONCLUSION AND FUTURE WORK

We have designed iPRP, a transport layer solution for improving the reliability of UDP flows with hard-delay constraints, such as smart-grid communication, industrial processes, high-frequency trading, and online gaming. iPRP is application and network transparent, which makes it plug-and-play with the existing applications and network infrastructure. Furthermore, our soft-state design makes it resilient to software crashes. Besides unicast, iPRP supports IP multicast, making it a suitable solution for low-latency industrial automation applications requiring reliable data delivery. We have equipped iPRP with diverse monitoring and debugging tools, which is quasi-impossible with the existing MAC layer solutions. With our implementation, we have shown that iPRP can support several sessions between hosts without any significant delay or processing overhead. To achieve a low delay and processing overhead, we use batching of nontime-critical instructions, thereby reducing the total number of system calls.

Interworking with legacy systems could be handled by developing adequate proxies. Legacy hosts that cannot be upgraded with iPRP software could be placed behind an iPRP proxy that would handle all iPRP functions and could thus communicate with an iPRP-enabled host. This would add redundancy to the path between the iPRP proxy and the other end of communication. Nevertheless, the iPRP proxy would now be a single-point-of-failure. A very interesting case arises if the legacy host is equipped with PRP. In such cases, the solution with an iPRP proxy still applies but could be improved using the concept of split proxies in order to remove the single-point-of-failure. Split proxies would each be singly attached to the PRP LAN and to the IP network. They would insert the iPRP header to each duplicate packet received from the PRP LANs. The challenge, left for future work, is to design a distributed algorithm between the split proxies in order to ensure consistency of iPRP sequence numbers.

Our implementation is publicly available and is currently being installed in our campus smartgrid [6]. In the future, we intend to do extensive measurements on our smartgrid and study the performance of iPRP in real networks.

To further reduce the delay overhead due to iPRP, one might think of a more efficient kernel-space implementation. However, given the low delay overhead of our user-space implementation and the problems of stability associated with a kernel-space implementation, this is not advisable. Hence, we would like to push the implementation of iPRP functionalities to the network adapter itself, similarly to the TCP segmentation offload technique [29]. Also, given the slow adoption of IPv6, porting the implementation to IPv4 can be of interest.

APPENDIX
PROOF OF THEOREM 1

To prove the statement of Theorem 1, we need the following lemmas.

Lemma 1: If $R \times \Delta < 2^{31}$ and $R \times (T_{inactivity} + \Delta) < 2^{31}$, then the wrap-around problem does not exist.

Proof: Proof: The wrap-around problem can arise in two cases.

Case 1: A late packet arrives with $\texttt{CurrSN} < \texttt{HighSN} - 2^{31}$. As $R \times \Delta < 2^{31}$, the time required by the source to emit 2^{31} packets is longer than Δ. Hence, HighSN cannot precede CurrSN for more than 2^{31} and this scenario is not possible.

Case 2: A fresh packet is received with $\texttt{CurrSN} > \texttt{HighSN} + 2^{31}$. This means that from the point of view of the receiver, there were more than 2^{31} iPRP packets lost in succession. As $R \times (T_{\text{inactivity}} + \Delta) < 2^{31}$, the time for more than 2^{31} consecutive packets to be sent is greater than $(T_{\text{inactivity}} + \Delta)$. Hence, the time between reception of any two packets differing by SNs more than 2^{31} is greater than $(T_{\text{inactivity}})$. Therefore, during this time, the iPRP session would be terminated and a new session will be initiated when the fresh packet is received. Hence, this scenario is also not possible.

Therefore, in the rest of the proof, we can ignore the wrap-around problem and do as if SNs of received packets were integers of infinite precision. Also, a notation such as $\texttt{HighSN}_{t^-}$ ($\texttt{HighSN}_{t^+}$, respectively) denotes the value of HighSN just before (after, respectively) time t.

Lemma 2 (Monotonicity of HighSN): If at time t, a packet with $\texttt{SN} = \alpha$ is received, then $\texttt{HighSN}_{t^+} = \max(\texttt{HighSN}_{t^-}, \alpha)$. Therefore, HighSN increases monotonically with time.

Proof: From Algorithm 6, when $\alpha > \texttt{HighSN}_{t^-}$ (line 4) then the value of HighSN is changed to α (line 7). Otherwise, when $\texttt{HighSN}_{t^-} \geq \alpha$ (lines 2 and 9), HighSN is unchanged, i.e., $\texttt{HighSN}_{t^+} = \texttt{HighSN}_{t^-}$. The two cases combined together give $\texttt{HighSN}_{t^+} = \max(\texttt{HighSN}_{t^-}, \alpha)$. ■

Lemma 3 (Fresh packet is never put in ListSN*):* If at time t, a packet with $\texttt{SN} = \alpha$ is forwarded to the application then $\alpha \notin \texttt{ListSN}_{t'^+} \forall t' \geq t$.

Proof: Let us prove by contradiction. Assume that $\exists t' > t$ such that $\alpha \in \texttt{ListSN}_{t'}$. Hence, $\exists t_1 \in (t, t']$ when α was added to ListSN. As $t_1 > t$, from Lemma 2, we conclude that $\texttt{HighSN}_{t_1^-} \geq \texttt{HighSN}_{t^+} \geq \alpha$. Now, from Algorithm 6, we know that only SNs $> \texttt{HighSN}_{t_1^-}$ can be added to ListSN. Hence, α cannot be added to ListSN at time t_1. Therefore, we have a contradiction. ■

Lemma 4: At any time t, $\texttt{HighSN}_{t^-}$ is equal to SN of a packet received at some time $t_0 < t$ or no packet has been received yet.

Proof: HighSN is modified only at line 7, where it takes the value of the SN received. Hence, HighSN cannot have a value of an SN that has not been seen yet. ■

Now, we proceed with the proof of the theorem. First, we prove statement (1). Assume we receive a duplicate packet with $\texttt{SN} = \alpha$ at time t. It means that a packet with $\texttt{SN} = \alpha$ was already seen at time $t_0 < t$. Then, from Lemma 2, it follows that $\alpha \leq \texttt{HighSN}_{t^-}$. Then, either $\alpha = \texttt{HighSN}_{t^-}$ (line 2) or $\alpha < \texttt{HighSN}_{t^-}$ (line 10).

Case 1: When $\alpha = \texttt{HighSN}_{t^-}$, the packet is discarded according to line 3.

Case 2: When $\alpha < \texttt{HighSN}_{t^-}$, line 10 is evaluated as false due to Lemma 3. Hence, the packet is discarded by line 14.

Next, we prove statement (2) by contradiction. Assume we receive a first copy of a valid packet with $\texttt{SN} = \alpha$ at time t but we do not forward it. This can happen either due to line 3 (case 1) or due to line 14 (case 2).

Case 1: Statement from line 2 was evaluated as true, which means that $\alpha = \texttt{HighSN}_{t^-}$. As $\texttt{SN} = \alpha$ is seen for the first time, Lemma 4 is contradicted. Hence, this case is not possible.

Case 2: Statement from line 10 was evaluated as false, which means that $\alpha < \texttt{HighSN}_{t^-}$ and $\alpha \notin \texttt{ListSN}_{t^-}$. We show by contradiction that this is not possible, i.e., we now assume that $\alpha < \texttt{HighSN}_{t^-}$ and $\alpha \notin \texttt{ListSN}_{t^-}$. Now, there are three cases when $\alpha \notin \texttt{ListSN}_{t^-}$ can be true.

1) $\texttt{SN} = \alpha$ was added to and removed from ListSN before time t because it was seen (line 11) which is impossible as the packet is fresh.
2) $\texttt{SN} = \alpha$ was added to ListSN and later removed at time $t_0 < t$ because the size of ListSN is limited to MaxLost entries (line 6). This means that at time $t_0 < t$ a packet with $\texttt{SN} = \beta$ was forwarded and $\beta - \alpha > \texttt{MaxLost}$ (line 6). However, this means that the packet with $\texttt{SN} = \alpha$ was not valid at time t_0 and therefore is also not valid at time $t > t_0$.
3) $\texttt{SN} = \alpha$ was never added to ListSN. Consider the set $\mathbb{T} = \{\tau \geq 0 : \texttt{HighSN}_{\tau^+} > \alpha\}$. $\mathbb{T}$ is nonempty because $t \in \mathbb{T}$, by hypothesis of our contradiction. Let $t_0 = \inf \mathbb{T}$. Then, necessarily $\texttt{HighSN}_{t_0^-} \leq \alpha < \texttt{HighSN}_{t_0^+}$ (say, $= \beta$). β is the SN of a packet received at time t_0. Since α is valid, $\beta - \alpha <$ MaxLost. Otherwise, α would be invalid at time t_0, therefore at time t, which is excluded. Then, we have two subcases possible.
   a) $\texttt{HighSN}_{t_0^-} < \alpha$. Then, by line 5, α is added to ListSN, which is a contradiction.
   b) $\texttt{HighSN}_{t_0^-} = \alpha$. But, by Lemma 4, a packet with $\texttt{SN} = \alpha$ must have been received before t_0, which is a contradiction because α is a fresh packet at $t \geq t_0$.

REFERENCES

[1] S. Petersen, P. Doyle, S. Vatland, C. Aasland, T. Andersen, and D. Sjong, "Requirements, drivers and analysis of wireless sensor network solutions for the oil gas industry," in *Proc. IEEE Conf. Emerg. Technol. Factory Autom. (ETFA)*, Sep. 2007, pp. 219–226.

[2] J. Capella, A. Bonastre, R. Ors, and M. Peris, "A wireless sensor network approach for distributed in-line chemical analysis of water," *Talanta*, vol. 80, no. 5, pp. 1789–1798, 2010.

[3] W. Chen and S. Cai, "Ad hoc peer-to-peer network architecture for vehicle safety communications," *IEEE Commun. Mag.*, vol. 43, no. 4, pp. 100–107, Apr. 2005.
[4] H. Dong, B. Ning, B. Cai, and Z. Hou, "Automatic train control system development and simulation for high-speed railways," *IEEE Circuits Syst. Mag.*, vol. 10, no. 2, pp. 6–18, May 2010.
[5] P. D. Christofides, J. F. Davis, N. H. El-Farra, D. Clark, K. R. D. Harris, and J. N. Gipson, "Smart plant operations: Vision, progress and challenges," *AIChE J.*, vol. 53, no. 11, pp. 2734–2741, 2007.
[6] M. Pignati *et al.*, "Real-time state estimation of the EPFL-campus medium-voltage grid by using PMUs," in *Proc. IEEE PES Innov. Smart Grid Technol. Conf. (ISGT)*, 2015, pp. 1–5.
[7] R. Kay, "Pragmatic network latency engineering fundamental facts and analysis," cPacket Networks, White Paper, 2009, pp. 1–31.
[8] G. Schiele, R. Suselbeck, A. Wacker, J. Hahner, C. Becker, and T. Weis, "Requirements of peer-to-peer-based massively multiplayer online gaming," in *Proc. 14th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2007, pp. 773–782.
[9] H. Kirrmann, M. Hansson, and P. Muri, "IEC 62439 PRP: Bumpless recovery for highly available, hard real-time industrial networks," in *Proc. IEEE Conf. Emerg. Technol. Factory Autom. (ETFA)*, Sep. 2007, pp. 1396–1399.
[10] M. Elattar *et al.*, "Using LTE as an access network for Internet-based cyber-physical systems," in *Proc. IEEE World Conf. Factory Commun. Syst.*, 2015, pp. 1–5.
[11] M. Rentschler and H. Heine, "The parallel redundancy protocol for industrial IP networks," in *Proc. IEEE Int. Conf. Ind. Technol. (ICIT)*, Feb. 2013, pp. 1404–1409.
[12] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, "TCP extensions for multipath operation with multiple addresses," Internet Engineering Task Force, RFC 6824 (Experimental), Jan. 2013.
[13] IEEE Standard for Local and Metropolitan Area Networks—Link Aggregation, IEEE Standards Association, IEEE Standard 802.1AX-2008, 2008.
[14] C. Hopps, "Analysis of an equal-cost multi-path algorithm," Internet Engineering Task Force, RFC 2992 (Informational), Nov. 2000.
[15] C. Fragouli and E. Soljanin, "Network coding fundamentals," *Found. Trends Netw.*, vol. 2, no. 1, pp. 1–133, 2007.
[16] D. J. C. MacKay, "Fountain codes," *IEEE Proc. Commun.*, vol. 152, no. 6, pp. 1062–1068, Dec. 2005.
[17] N. Sprecher and A. Farrel, "MPLS transport profile (MPLS-TP) survivability framework," Internet Engineering Task Force, RFC 6372 (Informational), Sep. 2011.
[18] P. Psenak, S. Mirtorabi, A. Roy, L. Nguyen, and P. Pillay-Esnault, "Multi-topology (MT) routing in OSPF," Internet Engineering Task Force, RFC 4915 (Proposed Standard), Jun. 2007.
[19] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.
[20] M. Torchia, "Innovative ICT empower a better connected smartgrid," International Data Corporation (IDC), Framingham, MA, USA, White Paper, Tech. Rep., Aug. 2014 [Online]. Available: http://enterprise.huawei.com/ilink/enenterprise/download/HW_362734.
[21] M. Popovic, M. Mohiuddin, D.-C. Tomozei, and J.-Y. Le Boudec, "iPRP: Parallel redundancy protocol for IP networks," EPFL, Lausanne, Switzerland, Tech. Rep. 19948, 2014.
[22] H. Weibel, "Tutorial on parallel redundancy protocol (PRP)," Zurich University of Applied Sciences, 2011.
[23] J. Nonnenmacher and E. W. Biersack, "Scalable feedback for large groups," *IEEE/ACM Trans. Netw.*, vol. 7, no. 3, pp. 375–386, Jun. 1999.
[24] E. Rescorla and N. Modadugu, "Datagram transport layer security," Internet Engineering Task Force, RFC 4347, Apr. 2006.
[25] M. Baugher, R. Canetti, L. Dondeti, and F. Lindholm, "Multicast security group key management architecture," Internet Engineering Task Force, Tech. Rep. RFC 4046, 2005.
[26] S. Hemminger et al., "Network emulation with NetEm," in *Proc. 6th Aust. Nat. Linux Conf. (LCA'05)*, Canberra, Australia, Apr. 2005, pp. 18–23.
[27] S. Salsano, F. Ludovici, and A. Ordine, "Definition of a general and intuitive loss model for packet networks and its implementation in the Netem module in the Linux Kernel," Univ. Rome, Rome, Italy, Tech. Rep., 2009.
[28] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "GPROF: A call graph execution profiler," *ACM Sigplan Notices*, vol. 17, no. 6, pp 120–126, 1982.
[29] L. Boucher, S. Blightman, P. Craft, D. Higgen, C. Philbrick, and D. Starr, "TCP offload network interface device," U.S. Patent 7 284 070, Oct. 16, 2007. [Online]. Available: http://www.google.com/patents/US7284070.

Miroslav Popovic received the B.Sc. degree in telecommunications from the School of Electrical Engineering, University of Belgrade, Belgrade, Serbia, in 2008, and the M.S. degree in computer science from École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland, in 2010. He is currently working toward the Ph.D. degree at the School of Computer Science and Communication Systems, EPFL.

His research interests include various aspects of smart-grid communication networks.

Mr. Popovic coauthored three papers that won the Best Paper Awards at e-Energy 2011, International Conference on Emerging Networking Experiments and Technologies (CoNEXT) 2012, and IEEE World Conference on Factory Communication Systems (WFCS) 2015, and he received the EPFL School of Computer and Communication Sciences Teaching Assistant Award in 2014.

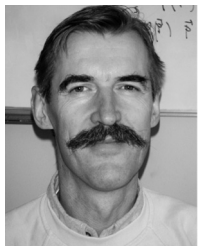
Maaz Mohiuddin received the undergraduate degree in electrical engineering from the Indian Institute of Technology Hyderabad, Sangareddi, India, in 2013. He is currently working toward the Ph.D. degree at the School of Computer Science and Communication Systems, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland.

His research interests include fault-tolerance in real-time systems and reliable communication for smart grid networks.

Mr. Mohiuddin has coauthored a paper presented at the IEEE World Conference on Factory Communication Systems (WFCS) 2015 that won the Best Paper Award.

Dan-Cristian Tomozei received the Engineering degree in computer science from École Polytechnique, Paris, France, in 2008, and the Ph.D. degree in computer science from the University Pierre et Marie Curie (UPMC), Paris, in 2011.

During his Ph.D. work he was affiliated with the Technicolor Paris Research Laboratory, where he developed distributed algorithms for congestion control and content recommendation in peer-to-peer networks. The present paper was written during his postdoctoral at École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, in the group of Prof. J.-Y. Le Boudec, where he was working on communication and control mechanisms for smart grids. Since 2015, he has been a Software Engineer with Cisco Systems, San Jose, CA, USA.



Jean-Yves Le Boudec (M'89–SM'01–F'03) received the Agrégation degree in mathematics from Ecole Normale Supérieure de Saint-Cloud, Paris, France, in 1980, and the Ph.D. degree in mathematics from the University of Rennes, Rennes, France, in 1984.

From 1984 to 1987, he was at Institut National des Sciences Appliquées Institut de Recherche en Informatique et Systémes Aléatoires (INSA/IRISA), Rennes. In 1987, he joined as a Member of Scientific Staff the Department of Network and Product Traffic Design, Bell Northern Research, Ottawa, ON, Canada. In 1988, he joined as a Manager of the Department of Customer Premises Network, International Business Machines Corporation (IBM) Zurich Research Laboratory, Ruüschlikon, Switzerland. In 1994, he became an Associate Professor and is a Professor with École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland. He coauthored a book on network calculus, which forms a foundation to many traffic control concepts in the Internet, an introductory textbook on information sciences, and is also the author of the book *Performance Evaluation* (EPFL Press, 2010). His research interests include the performance and architecture of communication systems and smart grids.