# Improving Fault-Tolerance of IMA using Safe Dynamic Reconfiguration

Tim Schubert[1*], Sven Friedrich[1], Wanja Zaeske[1], Umut Durak[1]

[1*]Safety Critical Systems & Systems Engineering, German Aerospace Center (DLR), Lilienthalpl. 7, Braunschweig, 38108, Niedersachsen, Germany.

*Corresponding author(s). E-mail(s): tim.schubert@dlr.de;
Contributing authors: sven.friedrich@dlr.de; wanja.zaeske@dlr.de; umut.durak@dlr.de;

**Abstract**

IMA is a central element of modern aircraft. It enables vendors to develop individual software and hardware components mostly independently from each other and integrate them using standardized interfaces, thereby reducing costs and shortening development cycles. Ensuring safety and fault-tolerance in systems of increasing complexity continues to present challenges to system integrators, requiring redundant deployments of many components. The software components are usually executed using a hypervisor based on ARINC 653 APEX services that provide the safety barriers necessary to ensure a deterministic run-time behavior. We present an approach that allows us to reduce the number of redundant partition deployments and improve fault-tolerance by exploiting these standardized interfaces to allow for safe run-time reconfiguration. We show how location-independence of APEX partitions can be achieved using a set of networked message routers that facilitate the communication between partitions without violating their functional real-time constraints or requiring changes to the partitions themselves.

**Keywords:** DIMA, Reconfiguration, ARINC 653, Real-Time

# 1 Introduction

Larger and more complex aircraft continually require more advanced interconnected systems to operate safely. In Integrated Modular Avionics (IMA), software components interoperate using standardized interfaces and execute using generalized computational resources that can host multiple applications of mixed criticality. This has the advantage of saving maintenance cost, Space, Weight, and Power (SWAP) by promoting the re-use of existing resources, but it requires appropriate isolation techniques, which DO-178C [11] refers to as partitioning.

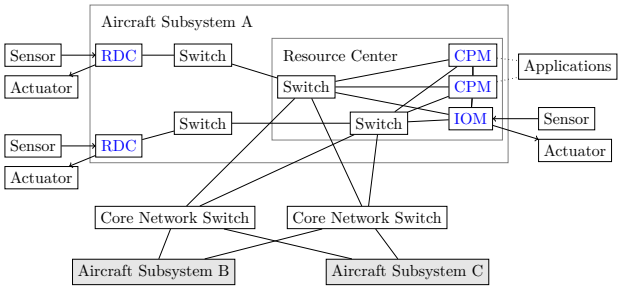


**Fig. 1** Example of a DIMA architecture adapted from ARINC 664 [6]. Subsystems A, B and C are connected by a core network. The subsystems are redundant on all levels including redundant CPMs and network switches. IO Module (IOM) connect sensors and actuators to the network and applications.

Distributed Integrated Modular Avionics (DIMA) achieves resilience to faults using redundant components on all integration levels, as shown in figure 1. This requires that additional computational and networking resources are present, even if they are not at all times required for the system to function.

Some types of faults however may be more efficiently recoverable by reassigning or reconfiguring existing resources. An example of such a fault is, when one of multiple producers of a type of data becomes unavailable, or it is detected that it produces invalid data. It may be preferable to let the same consumer of this data type consume from a number of alternate producers than to fall back to using a different redundant subsystem altogether. If the consumer however becomes unavailable, e.g. because the Compute Module (CPM) that hosts it becomes unavailable, it may be possible to start another instance of the consumer on a different CPM and redirect the output of the producer to it.

Using the standardized interfaces of Aeronautical Radio, Incorporated (ARINC) 653, the allocation of computational resources to software components can be updated at runtime, however the data paths between the producers and consumers, over the network or locally on the CPMs, remain statically configured from when the system is integrated. If a software component needs to communicate with a set of redundant peers, it needs explicit knowledge of the peers' communication end-points, which means that details of the system configuration leak into its implementation, limiting its reuse and increasing its complexity.

An alternative concept is presented by Zaeske et al. [21], who suggest that an intermediary software component should be used as a router between all other software components of a CPM. By alteration of its route table, the sources and destinations of the data-paths between attached software-components can be dynamically updated during reconfiguration, possibly in addition to changing the module schedule.

In this work, we present an implementation of such a message router and extend it by making the data-paths transparent to the network. A set of cooperating routers can extend the data paths from multiple CPMs over the subsystem- and core-networks, enabling location-independent execution of application software components and greater reuse of the available computational resources.

# 2 Background

## 2.1 IPC Primitives

ARINC 653 defines a set of operating system services, the APplication EXecutive (APEX), that application software can expect to find in compatible execution environments.

These execution environments usually get referred to as hypervisors in this context, since they take care of initializing the hardware of the CPM and manage the execution of software-components inside partitions that run sequentially and in isolation from each other.

Partitions can communicate with each other by accessing Inter-Process Communication (IPC) channels of the hypervisor. ARINC 653 itself remarks that the contents of APEX channels may transparently be transmitted between CPMs using any type of network and without the application software on either end having explicit knowledge of the transport type.

ARINC 664 Part 7 [6], also known as Avionics Full-Duplex Ethernet (AFDX), defines how data may be transmitted inside an aircraft using full-duplex switched Ethernet. It translates the concept of the APEX's IPC channels to the networking backplane, where Virtual Links (VLs) each connect one producer with a set of consumers. Multiple of these VLs can be time-division-multiplexed using the same physical network interfaces.

While ARINC 653 makes no requirements for the channels' or the aircraft network's to guarantee synchronous transmissions or guarantee data integrity; in practice though this is often required to satisfy minimum operational guarantees of the application software. ARINC 664 even demands that each VL must not exceed a maximum jitter between consecutive values and maximum end-to-end delays.

A suitable router for use in safety-critical systems must therefore be able to satisfy these constraints.

## 2.2 Concurrent Access to Processing Resources

In general-purpose operating systems, the scheduler will attempt to fairly distribute the available computation time between processes. This is done by pre-empting the execution of a process to allow another process to continue its execution, after its

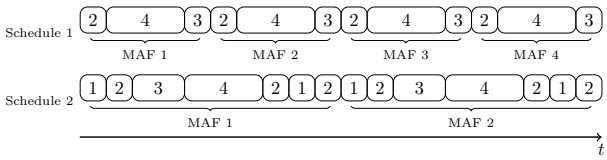


**Fig. 2** Two alternate TSP module schedules with different MAFs that each contain minor frames one to four. Each partition may be scheduled during multiple minor frames of different lengths during each MAF, but the minor frames inside each MAF remain the same, as long as the module schedule is not changed.

scheduled time has elapsed. In practice however, the work a process needs to perform depends on external events like hardware interrupts from timers and IO peripherals.

Interrupt-driven concurrency extends preemptive multitasking with the concept of programmable interrupt events [19] that hold the execution of the currently scheduled process and allow for an interrupt handler of another process to run immediately and process the interrupt. This can achieve shorter response times to external events, but makes the process scheduling more unpredictable if external events do not occur deterministically.

Instead, ARINC 653 requires that partitions execute according to one of a set of pre-determined schedules, that can be selected at run-time, but in which the schedule of each individual partition is fixed. Using Time-Space Partitioning (TSP), time is segmented into Major Frames (MAFs) inside which each partition is assigned any number of minor frames in the case of ARINC 653 P1 and exactly one for P4. A single schedule consists of repetitions of the same MAF. An example of such a TSP in shown in figure 2.

Using only the services of ARINC 653, it is not possible to interrupt the execution of a partition using an interrupt handler of another partition. Instead, ARINC 653 P1 [3] specifies that the core software should provide some means to forward interrupts from the hypervisor to individual partitions. In-case of XtratuM Next Generation (XNG), this is implemented using virtual interrupts that the partition only receives during one of its minor frames. Since this service is hypervisor-specific and not part of ARINC 653 P4, a portable router partition may not depend on it for interfacing with hardware-timers and IO devices and must perform its tasks using different mechanisms, such as polling IO.

## 2.3 Concurrent Access to Network Resources

In real-time applications, the processes have certain requirements for the latency and jitter of the data streams. The router must be able to minimize the delay and jitter, so that the required performance bounds are not exceeded. For this, it has to provide fair access to network interfaces that should be used to multiplex multiple data streams.

There are multiple different ways of achieving this, such as hardware-emulation [7], para-virtualization [18] and multiplexing access in hardware [10]. Various hypervisors provide virtualized access to underlying IO-devices using VirtIO, which the router cannot depend upon, since this interface is not yet part of ARINC 653. In theory, the block memory and file system services of ARINC 653 P2 [4] may be used for

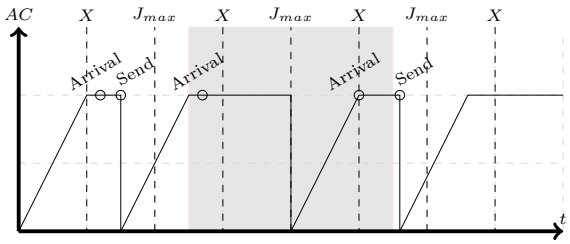


**Fig. 3** AFDX's behavior with a BAG ($X$) of 2 ms and a maximum jitter of 1 ms. The line shows the amount of tokens in the token-bucket (AC) that is used for shaping the traffic of a stream of data. The stream has new data every 2 ms (*Arrival*). The maximum allowable jitter ($J$), the time it takes to transmit a frame, is 1 ms. The sender is blocked by a transmission of another VL inside the gray area. Frames cannot be sent if they cannot be fully received within $J_{max}$, therefore the second frame is dropped, but there is sufficient time to send the third frame after the sender is unblocked.

implementing this, but P2 is not universally supported. Therefore, the router in this work must resort to directly and exclusively accessing the hardware devices. To ensure partition isolation, this access must be exclusive, and the router multiplexes accesses from multiple partitions.

Figure 3 shows how AFDX manages multiple concurrent VLs. Each VL is assigned at configuration time, an identity number, an Maximum Transfer Unit (MTU), and a Bandwidth Allocation Gap (BAG). The BAG is a measure for the length of time that is not used for transmitting data of a VL on any physical link. Therefore, combined with the link's transmission speed and MTU, the BAG also determines the maximum data rate of a VL. This maximum data-rate is enforced by the scheduler using a token-bucket. The AFDX scheduler may have to discard a frame if the link would be blocked before the maximum jitter is reached and how it delays a third frame, if the link is unblocked before the end of the maximum jitter period and before the next BAG. A system integrator must therefore take care that the transmission time of any two VLs that use the same interface do never overlap, and the link is not overutilized.

AFDX's VLs are similar to the rate-constrained traffic class of Time-Triggered Ethernet (TTEthernet). It is relevant to note that both do not require synchronized local clocks, so they cannot guarantee when traffic from a connected end-system may arrive at its destination. The only guarantees are towards the intervals and maximum jitter at which data is transmitted. This essentially means that real-time capable end-systems have to be able to receive data at any point in time. They must perform input-buffering either in hardware or in software, since the time a process reads from a buffer and the time the buffer is written to might not always align. The effects of this property on our router implementation are further investigated in section 6.2.

# 3 Related Work

Zaeske et al. [21] introduce the concept of a router partition as a possibility to increase the resilience and resource utilization of an IMA system using dynamic reconfiguration. They argue that dynamic reconfiguration is a way of making the same system resilient to multiple different environmental conditions without requiring multiple redundant deployments of the system. Dynamic reconfiguration of an ARINC

653 compliant system must be done at two levels, first the module schedule must be altered, which should be done by exchanging what partitions get ran rather than by changing the contents of the partitions, to preserve the partitioning barriers required by DO-178C [11]. The other part is dynamically altering the communication between partitions. Altering the communication in a trusted router partition, rather than selecting from multiple different inputs in the application partitions is preferable, since diagnostics can be obtained this way and the reconfiguration logic does not need to be replicated in each partition. For our demonstrator, we implement only the router and update the configuration of the router based on a pre-determined time-schedule.

The router implemented in this work provides such a dynamic multiple-producer, multiple-consumer mapping of the communication between the application partitions. This work also elaborates on the concept of multiple-producer, multiple-consumer channels by grouping input and output channels into VLs that can be multiplexed and scheduled onto a shared data network. It implements an interface that can be used by the mitigator described by Zaeske et al. [21] to switch between consecutive configurations atomically.

Van der Leest et al. [20] made a survey of the techniques used by publicly known hypervisors and implemented a prototype open source hypervisor that implements ARINC 653. They list multiple approaches to partition the usage of IO resources and point out their limitations, including an IO partition, scheduling, an auxiliary processor, and CPU interrupts. They implement two IO mechanisms from ARINC 653 based on the Xen hypervisor. Sampling ports are implemented using shared memory and queuing ports with ring buffers. For external networking, they use a separate driver that has direct access to the hardware and communicates with the partitions using the queuing ports. While this provides portability through the use of queuing ports, the network is accessed from a driver that resides inside the hypervisor which increases the potential for uncontained faults. By contrast, the router partition implemented in this paper contains the driver implementation as part of a dedicated partition.

Masmano et al. [17] present an IO-partition for Linux guests running under the XtratuM hypervisor that exclusively accesses the network devices. Instead of communicating via sampling and queuing ports, the physical network devices are exposed to the IO partition using para-virtualized VirtIO devices. Requiring VirtIO for accessing the network would require changes to the application partitions. They find that if the partitions are executed sequentially, the use of an IO partition incurs additional delays. The effective data-rate also depends on how often the IO partition is scheduled. We were able to replicate these findings.

Lee et al. [16] implement a partition that uses the APEX ports of the hypervisor to transparently translate the APEX ports of application partitions to the network, similarly to the router in this paper. Their network-manager is implemented on-top of a Linux kernel of Han et al. [14] that has added support for some APEX services such as sampling and queuing ports and acts as a hypervisor for application partitions.

Each APEX port of their Linux-hypervisor can only be accessed from one interface, while our router is able to multiplex multiple VLs onto the same physical interface. Both implementations allow at most one source for each destination, which is also a requirement of AFDX for consistency reasons. Multiple interface types are internally

abstracted using a Network Abstraction Layer, similarly to the Network Interface Layer (NIL) of our router. An important difference is that our router does not need to run as part of a system partition, while their network-manager does.

Their network-manager explicitly targets their APEX-enabled Linux kernel and relies on its configuration format, network interface layer and threading support, while our router is entirely agnostic to the underlying kernel or hypervisor, through use of a653rs, a safe and low-overhead hypervisor abstraction layer by Friedrich et al. [13].

Our router does not require any additional privileges or interfaces from the execution environment, besides what is required by ARINC-653 P4, which contains a smaller subset of the services contained in ARINC-653 P1, and for which implementations for practically all commercially available IMA execution environments exist.

Lee et al. [16] measure very low jitter for the roundtrip delay between two distributed instances of their Linux-hypervisors. The reason for this is that their application partitions are synchronized by means of a global clock, eliminating a significant source of jitter. Since AFDX-based DIMA systems are not inherently synchronized, we further investigate the effect of globally-asynchronous clocks on end-to-end latencies.

Lauer et al. [15] perform an estimation of the worst-case end-to-end delays in a system of networked software components in an IMA based on the tagged signal model and design, an evaluation method of the model that uses Integer Linear Programming (ILP). Their evaluation is more detailed than the simulation we performed in section 6.2, but they also find that the offset between the schedules of the distributed CPMs is a source of latency.

# 4 Reconfiguration

Reconfiguration of the router for different scenarios is achieved using a combination of generic programming language types, compile-time macros and run-time reconfiguration of the route table. The router is configured using a two-step process. The network interfaces and APEX ports that are available to the router are specified at compile-time, since they do not change during reconfiguration. During reconfiguration however, the router dynamically changes which interfaces are used by updating its route table.

## 4.1 Compile-Time Reconfiguration

The static part of the configuration remains the same in all alternate configurations during a deployment scenario. All resources are statically allocated inside the executable object code. This includes the APEX ports, the data rate and MTU of all network interfaces, the maximum number of virtual links, and the maximum required buffer size to transmit them all using the router. This way it is possible to statically determine how much memory the router requires and how much time must be allocated for it in a TSP schedule.

Using implementations for type generics, it is possible to exchange the scheduler, network interface driver, and hypervisor, as shown in figure 4. This enables a large degree of portability and re-usability for different platforms and use-cases. We demonstrate the portability by providing working implementations for two different

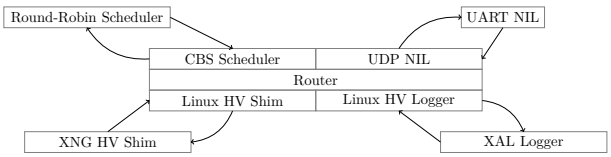


**Fig. 4** Components can be exchanged to adapt the router for use with a different hypervisor and network interface. It is possible to configure the same router with multiple interfaces of mixed types.

hypervisors, XNG [2] and a653rs-linux [13], and two different architectures, Zynq 7000 and AMD64, using two different modes of inter-hypervisor communication, via Universal Asynchronous Receiver / Transmitter (UART) and User Datagram Protocol (UDP).

## 4.2 Runtime Reconfiguration

A separate partition, called mititgator, writes the configuration containing VLs and their inputs and outputs, which can be network interfaces and hypervisor ports, to a sampling port that is read by the router. The router considers all configurations valid for which the statically configured resource limits are not exceeded and for which all referenced outputs and inputs exist.

By comparing the number of inputs and outputs contained in the configuration with the number of permittable inputs and outputs, it can be checked if the loaded configuration would fit into memory. Even more so, configurations that contain more than the maximum permissible number of inputs and outputs will not be deserialized and rejected as invalid before reconfiguration is attempted.

Since the configuration of the interfaces and VLs and their inputs and outputs are all known at design-time, it can be checked if a configuration would exceed the maximum allowable data-rate, delay or jitter at any interface.

The router will not operate until it receives a valid configuration, and it will reconfigure itself only if the provided configuration is valid. At the same time, this might include invalid configurations from a system-level perspective. For example, the router does not know if a network interface actually connects to a network on which it will be able to contact the destinations of a specific VL. This validation step may be achieved either by validating the configurations ahead-of-time using techniques like Model-Based System Engineering (MBSE) or dynamically using a validated algorithm, which are both outside of the scope of this work. If the system configuration changes over time, this situation must be recognized by alternate means and the mitigator must react to the change by updating the router's configuration for the deployed configuration to remain valid.

Since the routing configuration and the module schedule depend on each other, care must be taken to perform the switch between two system configurations without interruption or degrading the system health further. In particular, no port should be selected as the alternate source or destination of a VL before the partition that services the other end of the channel is selected to run in the module schedule. A fully featured mitigator that would also update the module schedule is out-of-scope

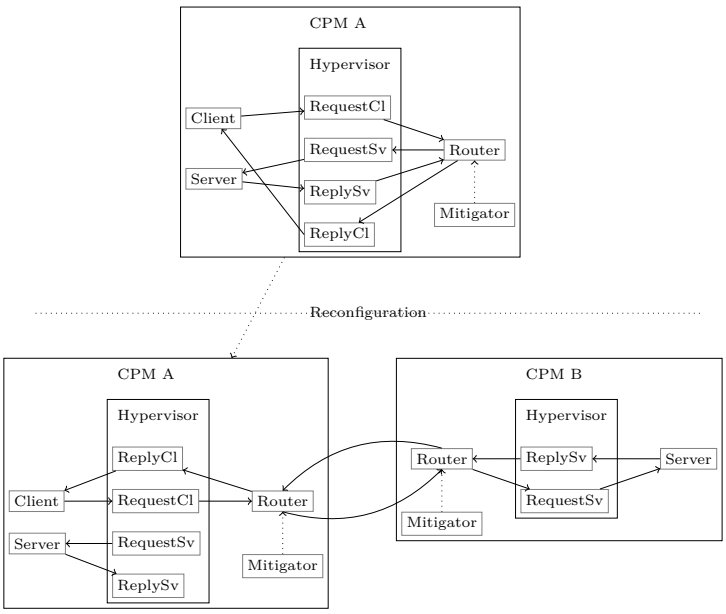


**Fig. 5** Dynamic reconfiguration of echo server and echo client running on the same node. After reconfiguration, the active client and server run on two different nodes.

for our demonstrator. Therefore, the server on CPM A is left running, but it does not participate in the system.

## 4.3 Demonstration of Dynamic Reconfiguration

Figure 5 shows how the consumer and producer of a VL can be updated dynamically at run-time. The demonstrator uses the XNG [2] hypervisor and two Cora Z7 [9] as the CPMs. UART is used to communicate between the two CPMs.

In this example, an echo client and echo server communicate using one or more instances of the router partition. The echo client periodically emits echo request messages. The echo server receives these messages and responds with echo reply messages. The client receives these echo replies and can determine the time it took to transmit the echo request and response by comparing the time-stamp against its local clock.

At first, the client and server run locally using the same hypervisor. Later, the local echo server stops responding, which would be detected by the mitigator. In our demonstrator, we emulate the execution of the mitigation strategy by letting our mitigator instruct the router to update its configuration every 20 seconds. As a result, after 20 s the router forwards the echo requests and echo replies to and from another echo server that runs on a remote CPM. The remote server becomes the new consumer of the echo requests and the new producer of the echo replies. The router acts as a bridge between the local APEX channels, transmitting their contents from one hypervisor to another using UART. All this happens without requiring changes to any of the application code or the channels available on either of the hypervisors.

By using the router and mitigator, instead of switching from one subsystem hosting the client and server to another redundant subsystem, we were able to reuse the working client and only exchange the server, by using another server using resources of another CPM. The same router and the mitigator can be used for any number of partitions on a CPM. The system integrator can use the demonstrated method to variably configure any number of spare consumers and producers that need to be replaceable at the same time, giving fine-grained control over the resource consumption.

An alternative that needs to be considered is to put the management of redundant inputs into the client and server partitions instead of placing this selection process inside the router. Would our demonstrator example use this type of redundancy management, it would require two additional channels for every possible pair of $n$ redundant clients and $m$ redundant servers. Since the memory required for storing these channels must be allocated in the hypervisor at design-time, this would take up to $(n * m) * M$ memory, where $M$ is the memory required for the sampling port contents or the First In - First Out (FIFO) queue and management data used by the hypervisor. By using the router, this memory consumption is effectively limited to $2 * (n + m) * M$. The memory consumption from any spare partition is the same when using this method and when using the router, since the memory needs to be allocated at design-time. Which partition occupies time in the schedule can always be selected by selecting different module schedules.

# 5 Resource Requirements

Since the resources of a system using DIMA are usually statically assigned to individual applications at design-time, the resource consumption of a configuration of the router has to be known at design time. We showed in section 4.1 how the maximum memory consumption of each router configuration is always known at design-time.

Another aspect is the time that is required during each MAF for the router to perform its tasks. This time has to be reserved for the router in the module schedule of the hypervisor. How this time can be distributed in the schedule depends on if the hypervisor implements ARINC 653 P1 or P4. However, the overall amount of work and time in the schedule remains identical. The following listing shows a high-level perspective of the control flow inside the router, from which a Worst Case Execution Time (WCET) can be derived based on the number of ports, interfaces, VLs, and their BAGs.

**Listing 1** High-level pseudo-code representation of the control-flow inside the router.

```
loop:
  if vl in scheduler.next():
    for s in vl.inputs:
      data = s.receive()
      for out in vl.outputs:
        out.send(data)
```

Each input or output port is part of at-most one VL. Each interface can be an output and input of any VL. The scheduler processes the next VL in the worst case

only after $BAG_{max}$. The resulting data is transmitted at worst with the smallest data-rate of any connected hypervisor port or network interface $B_{min}$. The loop will get pre-empted when the minor-frame of the router has elapsed and continued when its next minor-frame starts. The cyclomatic complexity of any run-time configuration of the router for any given compile-time configuration is therefore $W_{max}$, where $V$ is the number of VL, $P_{in}$ and $P_{out}$ are the number of input and output hypervisor ports, and $I$ is the number of network interfaces.

$$W_{max} = O(V * (P_{in} + P_{out} + I^2)) \tag{1}$$

To minimize the delay, especially if only one minor frame is available to the router, all ports, and interfaces that have data and are to be scheduled must be serviced during the minor-frame. At the same time, the limited time inside each MAF must be distributed effectively to all VLs. It must also be possible for the router to service each VL between individual BAGs. Therefore, a more precise estimate of the WCET for each VL is needed for use by system-integrators to arrive at the time $d_router$ that must be reserved for the minor frame of the router.

This estimate depends on the contents of the run-time configuration, which are the number of VLs, their respective BAGs, and their inputs and outputs as shown in equation 2.

$$d_{router} = \sum_{x}^{|VL|} (i_x * MTU_i * d_x^{receive} + \sum_{j}^{|O|} o_i * MTU_i * d_j^{send}) \tag{2}$$

The constants $d_x^{receive}$ and $d_x^{send}$ are the transmission durations of the respective network interfaces and local hypervisor ports. They depend on the deployment target and can be determined experimentally at design-time or gathered from data sheets. For example, we obtained these values by tracing the execution of the router on our experimentation platform. The predicates $i_x$ and $o_x$ indicate if a local hypervisor port or network interface is an input or an output to the virtual link $x$.

Similarly, the maximum network data-rate consumption can be derived from the BAG of each VL, the number of outputs it has, and the participating interfaces' / ports' MTUs. Since it is only known from the run-time configuration, which interfaces and ports participate in a VL, only an upper estimate can be derived at compile-time, which in most cases will be too tight to be efficient. If this limit should be optimized, the system-integrator has to make sure, either by limiting the possible configurations to a validated set or by using a validated run-time algorithm, that no run-time configuration exceeds any interface's maximum data-rate.

# 6 Evaluation

## 6.1 End-to-End Delay Variation

To determine the effect on real-time applications when using the router as an intermediary between partitions and the hypervisor, we measured the end-to-end delay between two partitions. These partitions run an echo client and an echo server, where the client periodically emits an echo request and the echo server emits an echo reply

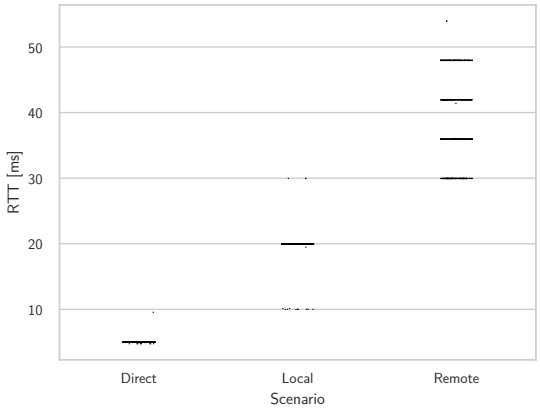


**Fig. 6** Experimentally determined Round-Trip Time (RTT) for direct transmissions (*Direct*), transmissions through the hypervisor (*Local*) and transmissions through two connected routers (*Remote*).

for every echo request it receives. Since each echo request contains a sequence number as well as a time-stamp and the server copies these contents to the echo reply, the client is able to determine reasonably precisely how much time elapsed between the creation of the echo request and the reception of the echo reply messages, which corresponds directly to the instantaneous end-to-end transmission delay between the echo client's ports and the echo server's ports.

We measure this delay in three different scenarios with the goal of distinguishing between the delays incurred from using the router as an intermediate and using two networked hypervisors. Figure 6 shows the measured end-to-end delays for a client and server that communicate directly, without an intermediate router, on the same hypervisor (*Direct*), with an intermediate router, on the same hypervisor (*Local*) and with two intermediate networked routers on separate hypervisors (*Remote*).

For the full schedule information see table 1. It is relevant to note that ARINC 653 does explicitly not guarantee that IPC between partitions is synchronous, so the time it takes for a message to traverse from one partition, e.g. the client to the router and to the server, is not guaranteed to be constant. However, using a 1 ms gap between partitions on XNG we were able to obtain reliable delays, with only some outlier values. We choose a large gap to be able to demonstrate how the different configuration scenarios affect the length of the delay.

The additional delay in the *Local* scenario stems from the additional three minor frames for the configurator, the router, and the processing delays within the hypervisor. Most of the time during each minor frame of the router is spent on hypervisor calls for performing IPC and UART transmissions. We were able to confirm this by tracing each hypervisor call and comparing the time to the total time the router took for processing each VL. However, the router always takes up a fixed time of 1 ms in the schedule, so this is what is measured as an additional end-to-end-delay, if each VL can be forwarded during each minor frame. The system-integrator has to assure

**Table 1** Schedules used for measuring the end-to-end delay between the echo client and server. Each minor frame is 1 ms wide. All minor frames are followed by a 1 ms break in the MAF.

| Scenario | Partition | Offset [ms] |
|---|---|---|
| Direct | Client | 0 |
| | Server | 2 |
| Local | Config | 0 |
| | Client | 2 |
| | Router | 4 |
| | Server | 6 |
| | Router | 8 |
| Remote (client) | Config | 0 |
| | Router | 2 |
| | Client | 4 |
| Remote (server) | Config | 0 |
| | Router | 2 |
| | Server | 4 |

this. In section 5, we show how to arrive at an estimate of the WCET for the router to perform all operations.

For the networked routers in the *Remote* scenario, we measured a sequence of central values that are distributed over a wider range. The number of same-sized minor frames and inter-minor-frame gaps that has to be traversed depends on which minor frame the client and server nodes are in when the router transmits an echo request or echo reply. We verify this assumption through simulating a model of the system.

## 6.2 Modeling and Simulation of End-to-End Delays

To investigate what exactly constitutes the inconsistent delays, our assumptions about the observed behavior, can be formulated as a model which we execute as a discrete-event simulation.

Each partition $x$ in schedule $s$ has a duration $d_x^s$ and an offset within the schedule $o_x^s$. On each hypervisor, partitions run in sequence according to a predetermined schedule until the end of each MAF with the duration $d_{MAF}^s$. Since the standard does not specify if there is time in between MAFs, for which no partition is scheduled, it should be assumed that there is a delay $d_{\overline{MAF}}^s$ between consecutive MAF. The delay within each partition almost exclusively comes from hypervisor-calls that copy data, $d_{apex}^{receive}$ and $d_{apex}^{send}$, and from accessing the network interfaces, $d_{net}^{receive}$ and $d_{net}^{send}$. We confirmed this by tracing the time the function calls to the APEX's Application Programming Interface (API) and the network interface layer took and comparing against the time it takes to process the VL. Since the client and server are not started at the same time, $d_{off}^s$ is the offset between their start times.

A partition can only completely send or receive a message to or from a port if the hypervisor-call can complete during the remaining time of the partition window. This condition can be expressed as condition 3.

$$o_x^s \leq ((t + d_{off_x}^s) \mod (d_{MAF}^s + d_{\overline{MAF}}^s)) \leq (o_x^s + d_x^s) \tag{3}$$

When a hypervisor-call or a network action started at time $t_e^s$ and completed after $d_x^s$, e.g. $d_{apex_s}^s$, the next event that processes the result of this action, like receiving a network frame or reading from a APEX port, can happen at the earliest if the condition 4 holds true at time $t$.

$$t_e^s + d_x^s \leq t \tag{4}$$

Both the condition 3 and condition 4 must hold true so that the partition can run and process the event. This essentially means that the results of all events take different times to become available, depending on when they happen during a MAF. How much time the message spends at each step along the path from the client to the server and back is influenced both by the length of the partition windows, the alignment of the two distributed schedules, the period of the echo messages and the length and variation of $\overline{MAF}$.

It is expected that this creates harmonic oscillations of the measured RTT. This hypothesis is validated through simulation and comparison with the measured data in the next section.

The simulation works by generating events according to the state-machine logic of the partitions involved at times determined by condition 3 and condition 4. The transmission of messages to and from the buffers of the hypervisor and network interfaces is modeled using events that are each assigned a time at which they are complete, and the model can act on them. Each emission of an echo message is also modeled as a periodically emitted event. Each reception of an echo reply is modeled as an event that the simulator creates a new RTT measurement for based on the time at which the echo request event was emitted and the current simulation time. The simulation time is incremented in configurable steps of at least 1 µs.

Figure 7 shows a 35 s of simulation time excerpt from the RTT during three simulation runs for different scenarios. We ran the simulation with a step size of 1 µs for 600 s of simulation time. The same pattern shown in figure 7 repeats itself every 20 s of simulation time.

The simulation shows a similar periodic behavior of the RTT. The offset of the two distributed schedules $d_{off}^s$ determines the length of the delay, as is shown by the simulation runs with *different start*-times for client and server and the *normal* run, where client and server start at the same time.

The different time between the major-frames of equal size $d_{\overline{MAF}}^s$ causes the jitter pattern with discrete RTT-levels, as shown by the run with *different inter-MAF* delays. This corresponds to the results shown in figure 6.

# 7 Discussion

## 7.1 Limitations

If the access to a networking device is done using memory mapped IO, this requires a Memory Management Unit (MMU) or Memory Protection Unit (MPU) to ensure that it is exclusive to the router. This requirement is not specific to the router itself, but to a network interface driver, which can be freely substituted depending on the deployment target, allowing the router itself to remain platform-agnostic. If the hypervisor allowed

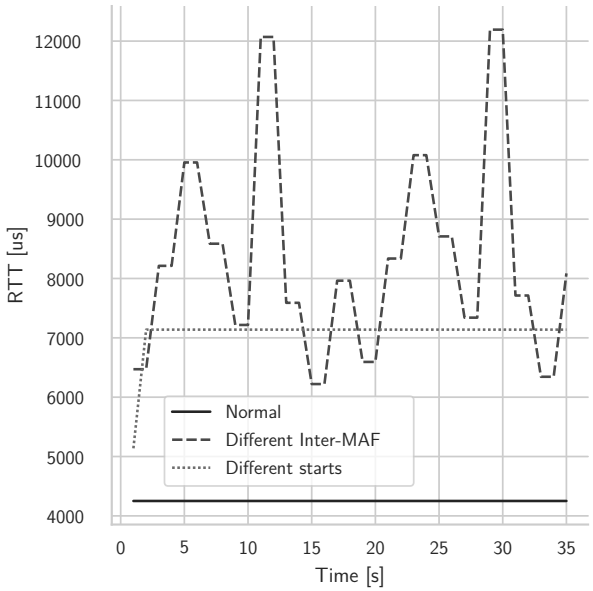


**Fig. 7** Simulated RTT. *Normal*: Both client and server have zero $\overline{MF}$ and start at the same time. *Different Inter-MAF*: The $\overline{MAF}$ of the client is 1111 µs and that of the server is zero. Both start at the same time. *Different start*: The $\overline{MAF}$ of the client and server is zero. The client starts 1111 µs after the server.

access to network interfaces using queuing ports, the router would be able to utilize this with no additional network interface drivers required.

Since the router does not store any additional state for the VLs, reconfiguration can be performed in one atomic step, enabling safe dynamic reconfiguration without downtime, provided the configuration is consistent with the module schedule.

The router does not yet implement any functionality for monitoring queue overruns and underruns, which would indicate misbehaving application partitions. This information is however trivially available and could be communicated to the health monitor partition using a reporting protocol that is currently being developed.

ARINC 653 P4 does only allow one minor frame for each partition within a MAF [5]. As a result, the latency between each two partitions for which a direct data producer-consumer relation exists, increases by up to one MAF, if the communication is done through the router compared to when performed directly. However, if multiple minor frames are allowed for the router, as is permitted in P1, the latency increases only by the additional time the router' minor frame takes up in the MAF.

The router has to be developed at least according to the highest assurance level of any partition for which it handles data [11]. The router has been developed as part of a demonstrator, so the development has not been performed according to DO-178C, however some considerations for a future compliant implementation can already be derived from the gathered experience. For one, the configuration can be considered a Parameter Data Item and is not part of the code of the router itself. DO-178C

requires that the configuration is shown to be correct, complete, and valid for DAL B and DAL A. The router performs only some correctness and completeness checks related to the schema definition of the configuration format, and checks if all inputs and outputs that are referenced in the configuration are indeed available to the router. Since the router itself does not have the necessary insights into the system health and configuration of the DIMA system, some verification, correctness, and completion checks will be performed at design-time, by the mitigator, or a combination of both run-time and design-time checks. We intend to perform this validation as part of a future demonstrator.

Another aspect that needs to be considered from an implementation standpoint is how to coordinate the switching of the router's configuration and the module schedule, so that the switch is atomic, and the system configuration remains consistent and without downtimes, also between multiple connected CPMs. For example, the UART used in the demonstrator has a very limited hardware receive buffer. If this buffer overflows because one router has switched to a new configuration that forwards messages to the UART and the remote router has not yet performed the switch, some data from the buffer may be overwritten and lost. Even if a reliable transport mechanism is used, the message may still be delayed longer than the maximum allowable delay of the VL transmitted.

## 7.2 Other concurrency concepts

As has been shown in section 6.1, synchronized clocks are required to achieve stable end-to-end delays. Otherwise, harmonic oscillations of the system of two networked nodes that each locally run synchronous schedules leads to potentially large variations of the end-to-end delay over time.

A possible approach for countering this is to use synchronized local clocks for running the TSP schedulers in the end-systems, thereby effectively making the relative timing of when the router can process the input buffer of the network interface and when the input buffer is written to by the other router constant. This synchronized network communication is already implemented in TTEthernet, which is the network protocol of choice in many aerospace applications. Attempts of retrofitting time-synchronization to AFDX are also promising in this regard [8].

Another possibility is to break up the synchronous local schedules and switch to a reactive programming model. This would let the router react to external events in a more timely manner, effectively reducing the maximum delay even further, but not keeping it constant, which may be sufficient for most applications that rely on the jitter-bounds of AFDX. The reactive control flow also would have the benefit of requiring a smaller amount of context changes and copy operations from doing hypervisor calls and scheduling partitions. This concept has been in use in the form of interrupt-driven concurrency in many real-time capable Real-Time Operating Systems (RTOSs) like RTIC [12] and FreeRTOS [1]. However, it removes some safety-barriers of TSP, increasing the barriers for the certifiability of such solutions.

The need to statically define memory areas for all possible redundant partitions at design-time remains a significant limitation to the efficient dynamic use of redundant partitions in ARINC 653. It is usually possible to define shared memory areas for

partitions that are not to be run at the same time, however this requires that no state from the partition previously occupying the memory area is left, so that the isolation of mixed-criticality partitions continues to be guaranteed.

# 8 Conclusion

We showed how a dedicated router partition or a set of networked router partitions can be used for making the data-paths inside a DIMA dynamically reconfigurable, alleviating some limitations to the reusability and resilience of current DIMA architectures.

The router uses a modular architecture and emphasizes compatibility with a wide range of execution environments by requiring only support for ARINC 653 P4, which is a minimal subset of the services defined in ARINC 653 P1 that is supported by virtually all IMA execution environments.

Our router enables network-transparent applications by providing an implementation of a portable mapping between local software-components of a CPM and the aircraft network that does not require changes to application partitions and does not rely on any specific hypervisor.

We investigated the effects of locally-synchronous – globally asynchronous communication on the end-to-end delays of applications using the router. In this context, we demonstrate both experimentally and through simulation how the limitations of ARINC 653 limits the routers use in portable real-time applications. These limitations stem from the combined use of TSP and asynchronous networks, which confirms results known from the literature.

# References


[1] (2017) The FreeRTOS Reference Manual. Manual 10.0.0 issue 1, Amazon Web Services

[2] (2018) Xng hypervisor – software user manual. Tech. Rep. 14-033.009.sum.08, Fent Innovative Software Solutions

[3] ARINC 653P1 (2021) ARINC 653P1 Avionics Application Software Standard Interface, Part 1, Required Services. Standard, Aeronautical Radio Incorporated (ARINC), Bowie, Maryland

[4] ARINC 653P2 (2019) ARINC 653P2 Avionics Application Software Standard Interface, Part 2, Extended Services. Standard, Aeronautical Radio Incorporated (ARINC), Bowie, Maryland

[5] ARINC 653P4 (2012) ARINC 653P4 Avionics Application Software Standard Interface, Part 4, Subset Services. Standard, Aeronautical Radio Incorporated (ARINC), Bowie, Maryland

[6] ARINC 664P7 (2005) ARINC 664P7 Aircraft Data Network Part 7 Avionics Full Duplex Switched Ethernet (AFDX) Network. Standard, Aeronautical Radio Incorporated (ARINC), Bowie, Maryland, USA

[7] Bellard F (2005) Qemu, a fast and portable dynamic translator. In: USENIX annual technical conference, FREENIX Track, California, USA, p 46

[8] Boulanger F, Marcadet D, Rayrole M, et al (2018) A time synchronization protocol for A664-P7. In: 2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC). IEEE, London, UK, pp 1–9, https://doi.org/10.1109/DASC.2018.8569837

[9] Digilent (2023) Cora Z7 Reference Manual - Digilent Reference. URL https://digilent.com/reference/programmable-logic/cora-z7/reference-manual

[10] Division ILA (2011) Pci-sig sr-iov primer, an introduction to sr-iov technology, revision 2.5. Tech. rep., Intel Corporation

[11] ED-12C (2012) Software Considerations in Airborne Systems and Equipment Certification. Standard, The European Organisation for Civil Aviation Equipment, Malakoff, France

[12] Eriksson J, Häggström F, Aittamaa S, et al (2013) Real-time for the masses, step 1: Programming api and static priority srp kernel primitives. In: 2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES), pp 110–113, https://doi.org/10.1109/SIES.2013.6601482

[13] Friedrich S, Engler E, Schubert T, et al (2023) Assuring apex with a versatile rust api. In: embedded world 2023 Conference Proceedings. WEKA FACHMEDIEN GmbH, pp 298–305

[14] Han S, Jin HW (2012) Kernel-level arinc 653 partitioning for linux. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing. Association for Computing Machinery, New York, NY, USA, SAC '12, p 1632–1637, https://doi.org/10.1145/2245276.2232037

[15] Lauer M, Ermont J, Boniol F, et al (2011) Latency and freshness analysis on ima systems. In: ETFA2011, pp 1–8, https://doi.org/10.1109/ETFA.2011.6059017

[16] Lee SH, Han S, Jin HW (2012) A Configurable, Extensible Implementation of Inter- Partition Communication for Integrated Modular Avionics. In: 2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp 453–458, https://doi.org/10.1109/RTCSA.2012.44, iSSN: 2325-1301

[17] Masmano M, Peiró S, Sánchez J, et al (2012) IO Virtualisation in a Partitioned System. In: Embedded Real Time Software and Systems (ERTS2012), Toulouse,

France, URL https://hal.science/hal-02192402

[18] Russell R (2008) Virtio: Towards a de-facto standard for virtual i/o devices. SIGOPS Oper Syst Rev 42(5):95–103. https://doi.org/10.1145/1400097.1400108

[19] Tannenbaum AS, Bos H (2015) Modern Operating Systems, Fourth Edition. Pearson Education Inc.

[20] VanderLeest SH (2010) ARINC 653 hypervisor. In: 29th Digital Avionics Systems Conference, pp 5.E.2–1–5.E.2–20, https://doi.org/10.1109/DASC.2010.5655298, iSSN: 2155-7209

[21] Zaeske W, Brust CA, Lund A, et al (2023) Towards enabling level 3a ai in avionic platforms. In: Groher I, Vogel T (eds) Software Engineering 2023 Workshops. Gesellschaft für Informatik e.V., Bonn, pp 189–207, https://doi.org/10.18420/se2023-ws-18