Fault Tolerance in Highly-Reliable Ethernet-based Industrial Systems

Inés Álvarez, Alberto Ballesteros, Manuel Barranco, David Gessner, Sinisa Djerasevic, Julian Proenza
Departament de Matemàtiques i Informàtica, Universitat de les Illes Balears, Spain,
{ines.alvarez, a.ballesteros, manuel.barranco, julian.proenza}@uib.es

***Abstract*—Many industrial systems have specific requirements derived from the applications they execute. Specifically, the interaction of a Distributed Embedded Control System (DECS) with the real world imposes strict real-time and reliability requirements. For a system to be real-time it has to produce a proper result in a bounded time. On top of that, for a system to be reliable it has to operate continuously during its mission time and, in cases in which very high reliability is needed, Fault Tolerance (FT) techniques are used. Moreover, these systems are often deployed in dynamic environments where the operational conditions may change in an unpredictable manner. Therefore, there is an increasing interest in creating DECS that are capable of modifying their behaviour autonomously and dynamically in response to unexpectedly changing requirements or conditions. In recent years, there is a growing trend towards using Ethernet as the network technology for DECS. Unfortunately, the original specification of this technology lacks appropriate services to fulfil the most demanding requirements of industrial systems. In this regard, many Ethernet-based protocols and standards have been proposed along the last years to deal with these limitations. In this work we survey solutions that have been proposed to achieve FT in Ethernet-based DECSs, considering faults both in their nodes and communication subsystem. Additionally, we discuss adaptive FT techniques that can be used to increase the flexibility of adaptive DECS. Finally, we identify future trends and open challenges to build highly-reliable DECS in the future.**

***Keywords*—industrial systems, real time, dependability, fault tolerance, Ethernet**

I. Introduction

In the last decades there has been a growing trend towards the automation of all kinds of processes in all kind of applications. This trend can be traced back to the systems used to initiate the automation of workshops and factories and it experienced a significant boost with the incorporation of the electronics technology. These processes typically consist in controlling a dynamic system, e.g., to continuously maintain said system in a given state. This implies interacting with the real world and, thus, not only the control system must produce a correct control output, but it also must produce it in a bounded pre-specified time that corresponds to the physical dynamics of the system under control, i.e., it must be delivered in *real-time* [1].

Additionally, many of these systems require some level of *dependability*. We use this concept as a general label for referring to different attributes such as *availability*, *reliability* or *safety*, as it is done in [2].

Many of these systems have to be highly *available*, i.e., the time in which they cannot operate, due to some failure, must be significantly lower than the time in which they operate correctly. This is a frequent requirement, especially in industrial automation applications, as any downtime in a factory costs a non-negligible amount of money. Another frequent requirement in many applications is that the system has to present a high *safety*, which means that it has to have a high probability of either operating correctly or fail in a manner that does not have catastrophic consequences for humans or the environment. A more demanding attribute is *reliability*, which is the one requested when even a brief interruption of the system service is catastrophic by itself (there is no possible safe state when the system fails). In this case, it is expected that the system operates correctly in a continuous way for a significant interval of time know as *mission time*.

It is noteworthy that a highly-reliable system is by definition highly available and safe but the opposite is not the case. As it can be expected, high reliability is the hardest to get and most expensive of these attributes, and typically leads to the most complex systems. In this paper we will put the main focus precisely in highly-reliable systems since their sophistication deserves special attention. However, we will also discuss systems that are more oriented to the other attributes due to their widespread use and industrial interest.

Another adjective that is used to describe highly-dependable systems is *critical*. Although the adjective can be used to refer to highly-safe systems we will use it in this paper as equivalent to highly-reliable.

Fault tolerance is the family of techniques typically used to achieve dependability in general, and high-reliability in particular. It encompasses all the mechanisms that ensure that the system is able to deliver a correct service, even in the presence of faults. It is true that a certain level of reliability can be achieved using simpler approaches such as improving the quality of the system components, but to reach the values of reliability that are expected in critical applications, it is necessary that the system is able to tolerate faults.

Another interesting system characteristic that is receiving an increasing amount of attention in the last years is *adaptivity*. An adaptive system is able to change its operation strategies (e.g. its real-time guarantees or its type of fault tolerance) autonomously and dynamically to cope with changes in the operational requirements, the environment or the system itself. Besides the obvious advantages of such capacity in terms of an efficient use of the internal resources of the system, this ability

can also be used to further increase the level of reliability achievable by the system.

Although all the above discussion of system attributes that are relevant in critical applications has been presented mostly in the industrial context, the truth is that the kind of capabilities that have been mentioned are also relevant beyond the strict factory automation, for instance in all kinds of vehicles such as planes or cars. Therefore we will consider a wider focus that includes all embedded systems that are potentially useful for critical automation applications.

Many automation systems are nowadays constructed in a *distributed manner*, i.e., as a set of interconnected *nodes*. The reasons for introducing distribution are quite diverse and include the actual physical separation of the elements under control, and the increased modularity and composability that distribution brings. These systems are called Distributed Embedded Control Systems (DECSs) or Distributed Embedded Systems (DESs) for short. Each of the nodes of a DES executes one or several *tasks*, which are the smallest units of computation. Moreover, nodes must cooperate to deliver the required service. In order for the nodes to cooperate, they must exchange information through a *network*, which will be also called *communication subsystem*. Besides the communication media for connecting the nodes to each other, the network can include other components that we will call *network devices*.

When DESs are to be used for critical real-time applications, the corresponding real-time and high-reliability requirements must be enforced at every level of the system architecture. Specifically, they must be enforced both at the network level, i.e., messages must be delivered on time and faults must not affect the communication; and at the node level, i.e., tasks must finish the work before their deadlines and the faults must not affect the execution. In fact it has been demonstrated that for reaching a truly high reliability it is not enough to tolerate faults in the communication subsystem but it is mandatory to tolerate node faults [3], [4]. This is because nodes are typically the most complex components in a DES and, thus, they have the highest probability of being affected by faults.

In recent years, there is a growing trend towards using Ethernet as the network technology for DESs. Ethernet's main advantages are low cost, high bandwidth, compatibility with IP-based networks and high scalability. Unfortunately, the original specification of this technology lacks appropriate services to fulfill the demanding requirements of critical real-time applications. In this regard, many Ethernet-based protocols and standards have been proposed along the last years to deal with these limitations.

In this work we survey mechanisms that have been proposed to achieve fault tolerance in Ethernet-based DESs, considering faults both in their nodes and communication subsystem. Therefore the paper will cover two different aspects; the different protocols that have been proposed to extend Ethernet's capabilities to make it more suitable for critical applications and the different techniques that can be used to achieve node fault tolerance in systems that use Ethernet as basic technology for building their communication subsystem.

The remainder of the document is structured as follows. Section II provides an overview of the most relevant RT concepts for this paper; whereas Section III describes the most important fault-tolerance concepts. In Section IV we discuss fault tolerance aspects of industrial Ethernet-based protocols and in Section V we survey solutions to implement node-level fault tolerance. Finally, Section VI concludes the paper providing a summary of the most relevant concepts discussed as well as open issues to be addressed by the community when building highly-reliable real-time Ethernet-based DES.

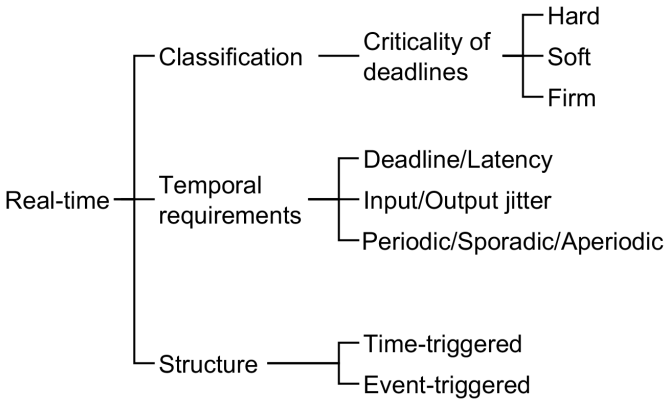


Fig. 1: Aspects of the real-time systems. Figure based on [5].

II. Real Time Concepts

As introduced previously, a system provides a real-time service if it does so timely, i.e. before a specific deadline expires [5]. Fig. 1 depicts the main aspects of real-time systems that will be described in this section. Generally speaking a system handles events by means of one or more tasks, thereby providing a set of services. An event can be internal, e.g. the need of taking a sample; or external, e.g. a sensor value reaching a specific threshold. Moreover, an event can be periodic, aperiodic or sporadic (aperiodic with a minimum inter-arrival time). Depending on the strategy a system uses to handle its events, it can be classified as either *time-driven*, when it executes its functions following a given internal time basis; or as *event-driven*, when it executes them on demand when the corresponding events happen. Each one of these strategies has its pros and cons. For instance, a time-driven system is naturally well suited to handle periodic events, but it may not be as reactive as an event-driven one to handle aperiodic/sporadic ones. Conversely, although more reactive, an event-driven system may be vulnerable to event showers, which may exhaust the system resources and prevent it from meeting its deadlines.

With respect to the consequences of missing a deadline, a system can be roughly classified as either *hard real-time*, when missing a deadline leads the system to fail in a catastrophic manner; *soft real-time*, when missing a deadline is acceptable as it merely degrades—even temporarily—the service; or *firm real-time*, when the system has a rigid deadline, i.e., missing a deadline invalidates the results but the consequences are not catastrophic.

In order to fulfill its real-time requirements, i.e. to provide its services timely, both the execution of tasks and the transmission of messages (in a distributed system) have to be appropriately scheduled so as for tasks and messages to meet their own deadlines. To achieve this, it is fundamental that the worst case response time of tasks and the maximum end-to-end delay of messages are both deterministic and bounded. Moreover, to achieve a good control quality, control systems normally require tasks and messages to exhibit a low *jitter*; where the jitter can be understood as the variability with which a given task is released/ends, or the variability with which a message is transmitted/received.

There are two main types of real-time schedulers, namely *cyclic executive schedulers* and *priority-based schedulers*. Basically, a cyclic executive scheduler calculates beforehand the instants of time at which each task is going to be executed (or each message is going to be transmitted) so that all tasks (or messages) meet their deadlines; whereas a priority-based scheduler assigns priorities to tasks (and messages) so that the order in which they access the resources in case of conflict, e.g. the CPU or the communication medium, thanks to these priorities allow them to meet their deadlines.

In the context of communication networks, a cyclic executive scheduler has traditionally been used in what are called *Time-Triggered* (TT) networks [6]. In this kind of networks the application (tasks) and/or the network itself trigger/s the transmission of messages at predefined instants of time. The traffic these messages constitute is commonly referred to as *Time-Triggered traffic*, and it is specially adequate for supporting time-driven systems. More specifically, these networks divide the communication in rounds generally called *communication cycles* that are periodically triggered to allocate the different *streams*. A stream can be understood as a message (or set of messages) related to a given piece of information and that has (or have) specific RT requirements, e.g. a message that conveys the value of a given sensor and that has to be transmitted with a period and deadline of n ms. Conversely, a priority-based scheduler has traditionally been used in what are called *Event-Triggered* (ET) networks, in which the application triggers on demand the transmission of the messages that constitute the streams. The traffic in this case is known as *Event-Triggered traffic*, and it is specially well suited for event-driven systems.

Both TT and ET real-time networks must rely on a deterministic medium access control (MAC) mechanism, i.e. a MAC mechanism that ensures that each message has a deterministic and bounded maximum end-to-end delay. In this sense, these networks must rely either on a static or on a contention-free dynamic MAC mechanism. The most natural choice for a TT network is to use a static MAC mechanism such as *Time-Division Multiple-Access* TDMA; but a contention-free dynamic one, e.g. polling-based, can also be used. In the case of an ET network, the preference is to use a contention-free dynamic MAC protocol, e.g. based on polling, token passing, etc.

TT and ET networks have pros and cons that are more or less analogous to the ones of time- and event-driven systems respectively. In this sense TT networks are preferred in critical real-time applications, because all the communication resources are reserved in advance. This means that, the messages that are transmitted at every instant are known, which makes it easier to proof their determinism. However, when compared with ET networks, TT ones present limitations to efficiently convey traffic that is inherently event-triggered, e.g. alarms.

Currently several real-time communication networks combine and extend the mechanisms of traditional TT and ET networks to adequately support both types of traffic (TT and ET). The capacity of doing so is known as *real-time flexibility*. A usual strategy to provide this capacity consists in dividing each communication cycle into *communication windows* to accommodate different types of traffic.

Moreover, on the quest for supporting adaptivity, a number of newer communication networks also provide what is known as *operational flexibility*. This is the ability to change the RT requirements and the schedule of the traffic online, to provide an adequate communication as the operational requirements of (and the services executed by) the system change.

Another important aspect of RT systems is that they usually require time synchronization mechanisms, so that tasks (in the same or in different nodes) can adequately coordinate among them, and messages are timely transmitted and forwarded by the different network devices. In some systems, each node and network device has its own local timer and are not tightly synchronized. When so, the nodes and network devices implement some mechanisms to bound bursts of activity so that they can share the resources and operate in a real-time manner. However, most systems that are time-driven or that rely on a TT network, require a tight time synchronization. Moreover, a tight time synchronization simplifies the real-time and the fault-tolerance mechanisms, e.g. the forwarding of messages at specific instants of time, the detection of errors like message omissions, or the management of the consistency among devices that perform the same tasks. This is so because this kind of synchronization allows having more knowledge about the current and future actions that should occur in the different parts of the system, e.g. the simultaneous reception of two copies of the same message through two different links.

There are two main strategies to provide tight time synchronization. One of them consists in having a global (system-wide) notion of time. This strategy is commonly known as global *clock synchronization* and it can be implemented in several ways. For instance, some systems include a privileged device or node that provides a master clock, which then is disseminated among the clocks of the rest of devices and nodes. In other systems devices and nodes synchronize their own local clocks by exchanging information about their visions of the current time and then implementing a convergence function. The other main strategy to achieve a tight time synchronization consists in having a privileged device or node that polls the actions and the communications in the rest of the system according to its own clock.

As we anticipated in the introduction, there is great interest in using Ethernet for industrial applications. Nevertheless, Ethernet was not designed to provide RT guarantees and, thus, it provides no mechanism for real-time communication [7]. Shared Ethernet relies on a contention-based (collision-based) MAC mechanism called CSMA/CD, which cannot guarantee access to the communication medium in a deterministic and bounded

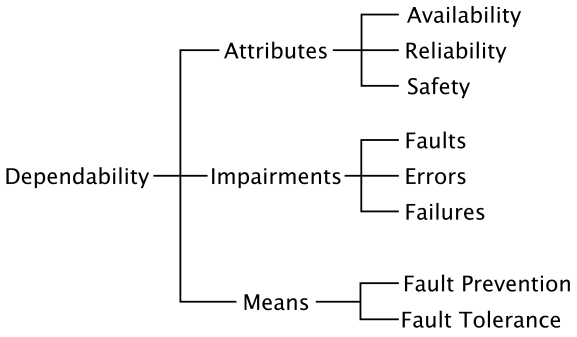


Fig. 2: Subset of the dependability tree showing the concepts considered in this paper and how they are classified. Figure based on one appearing in [2].

time, and which cannot bound the jitter either. Moreover, full-duplex Ethernet switches no longer rely on CSMA/CD, and prevent contentions (collisions) by using a dedicated uplink and a dedicated downlink per node. Nevertheless, switches may also introduce non-deterministic and unbounded forwarding delays and jitter. This is due to frames being buffered upon reception in the switch and then queued before retransmission. Also, Ethernet does not provide any sort of time synchronization mechanisms (except the ones needed to synchronize the receiver's communication controller with the beginning of each frame to not misinterpret its bits).

Fortunately, as we will indicate in Section IV, several mechanisms can be added to Ethernet to provide RT guarantees. Some of these techniques include MACs based on TDMA, polling or token passing; full-duplex switches with adequate queuing policies; *traffic shaping*, in which the network devices delay the transmission/forwarding of certain types of traffic during a limited amount of time to provide bounded latencies to other kinds of traffic; *resource reservation*, which consists in checking that are enough communication resources prior to registering a new or modified stream; *traffic policing*, which consists in monitoring the traffic to ensure that it meets a set of requirements previously specified (e.g. the maximum reserved resources); clock synchronization protocols, etc.

III. Fault Tolerance Concepts

A system is highly reliable if it has a high probability of performing its function continuously, without failing due to faults. There are two main ways to procure reliability, namely *fault prevention*, i.e. to prevent the occurrence of faults, for instance, investing in the components quality; and *fault tolerance* (FT), i.e. to include mechanisms for the system to provide its service even when it suffers from faults. Even if fault prevention is used, in complex systems like DESs faults eventually happen; thus, it is fundamental to provide them with adequate FT mechanisms if high reliability is required.

Figure 2 shows the relationship among the different dependability concepts considered in this article. Furthermore, it classifies them into three groups: attributes, impairments and means. A description of these concepts can be found next.

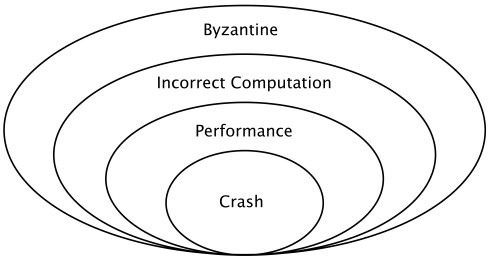


Fig. 3: Inclusion hierarchy diagram that depicts the presented failure modes. The inner failure modes are more restricted and benevolent; whereas outer failure modes are less restricted and harder to deal with.

A. Fault Model and Failure Modes

In order to correctly design an FT system it is important to take into account the *fault model*, i.e., the types and number of faults the system has to deal with. Furthermore, it is also important to identify the *failure mode*, i.e., the deviation in the service provided by the system (or in the service provided by a subsystem within the system) which is caused by the faults.

An exhaustive classification of faults can be found in [2]. Roughly speaking, we can differentiate between 1) hardware vs software, 2) temporary vs permanent and 3) simultaneous vs sequential.

On the other hand, failures can be classified following different criteria. For instance, they can be hierarchically classified according to their degree of restriction, as proposed in [6]. For the sake of succinctness, we will only refer to the next classes:

- Byzantine: lack of restrictions on the way the system can behave. It includes two-faced behaviours and impersonations.
- Incorrect computation: the system delivers incorrect results, either in the value or the time domain.
- Performance: the system delivers correct result in the value domain, but fails to do it in the time domain.
- Crash: the system omits the delivery of results from the moment the failure happens on.

Figure 3 depicts an inclusion hierarchy of the described failure modes. The inner failure modes are more restricted and benevolent; whereas outer failure modes are less restricted and harder to deal with.

B. Achieving fault tolerance

Figure 4 shows a classification of the different means to achieve Fault Tolerance and how they are related. FT is carried out by means of *error processing* and *fault treatment* [8]. Error processing, consists in eliminating errors from the state before they cause the failure of the system. In contrast, fault treatment consists in preventing faults from causing errors again.

Moreover, there are two different techniques for error processing. On the one hand, *error recovery* consists in detecting the error and replacing the erroneous state by an error-free state. If the error-free state is a past state, it is called

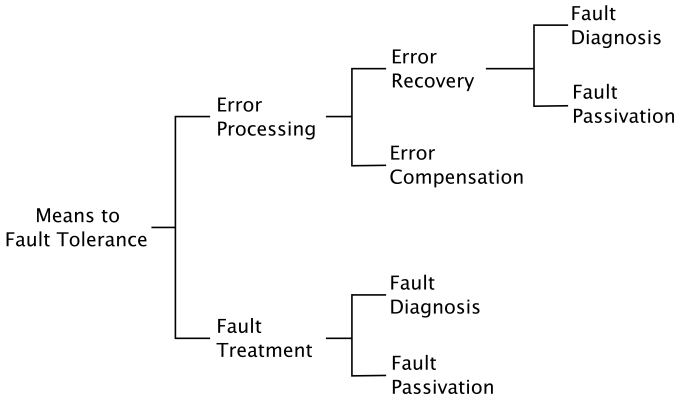


Fig. 4: Tree that classifies the different techniques used to achieve fault tolerance.

backward recovery; and if the error-free state is a new state, it is called *forward recovery*. On the other hand, *error compensation* consists in designing the system with enough redundancy to produce correct results even in the presence of faults.

In a DES, fault tolerance can be achieved at different levels of the system's architecture and using different types of redundancy. Specifically, we distinguish four different types of redundancy [9]:

- *Hardware redundancy* (also known as *spatial redundancy*) consists in providing the system with more hardware components than needed if faults could not occur. Examples of hardware redundancy are the use of redundant (multiple) sensors, nodes, switches or communication links.
- *Software redundancy* consists in providing the system with additional software beyond what is needed to carry out the operation of the system. An example of software redundancy is the use of redundant tasks to carry out the same action.
- *Time redundancy* consists in carrying out the same action several times with the same hardware and software. An example of time redundancy is message retransmission.
- *Information redundancy* consists in introducing more information than needed if faults could not happen.

A specific type of redundancy is *replication*, in which the system is provided with several identical copies of software and/or hardware components. With replication a fault affecting one of the replicas can be tolerated thanks to the non-faulty ones that take over the operation. There are three main types of replication [6]. Firstly, with *active replication* all the replicas perform the same operation in parallel, i.e., all the replicas provide the same service and, in absence of faults, the result can be obtained from any of them. In order to tolerate potential faults, the result is actually obtained by voting on the values proposed by each replica. This corresponds to the concept of error compensation presented above.

Secondly, with *semi-active replication* one of the replicas is in charge of the non-deterministic decisions. This replica is commonly referred to as central or *leader*. The leader must inform the rest of the replicas about the results of the decisions. The rest of the replicas are commonly referred to as *followers*. In the case of deterministic decisions, the results can be obtained just from the leader or from all the replicas; like in active replication. If the leader fails, its failure is detected and one of the followers becomes the leader.

Finally, with *passive replication* only one of the replicas, the primary replica, performs the operation while the rest, the standby replicas, remain inactive. If the primary replica fails, its failure is detected and one of the backup replicas takes over the operation of the primary replica, typically by starting in a previously stored (using checkpointing) state of the faulty primary replica. This corresponds to the concept of backward error recovery presented above.

Note, however, that redundancy is not only used to achieve fault tolerance. An example is the use of the *Frame Check Sequence* (FCS) in Ethernet frames. FCS is a way of information redundancy used to perform error detection. Another example is the use of software redundancy to carry out specific checks to assess the correctness of the results provided by a task.

In DESs with stringent RT requirements (e.g. short deadlines), error compensation represents the most suitable solution to increase the reliability. This is so because error compensation allows to tolerate faults without introducing a recovery (fail-over) time, which could prevent the system from meeting its deadlines. In this case it is said that the system *seamlessly tolerates* faults or that the system has *seamless redundancy*. Furthermore, when error compensation is used, the system does not need to be aware of the existence of faults to produce a correct result. This is known as *fault masking*.

It is also noteworthy that replication is specially suitable to eliminate *Single Points of Failure* (SPoF). A SPoF is any single component within a system whose failure could cause the failure of the whole system, even if that component is not directly responsible for calculating application results. Thus, eliminating any SPoF is essential to increase the reliability of any DES.

However, to properly implement replication, it is fundamental to address two issues. On the one hand, faults affecting different replicas must be independent of each other, i.e., that when a fault affects a replica, the same fault does not affect any other. For instance, it would be desirables that replicas are powered by different power supplies, so that a failure in one power supply only affects the replica it is connected to. On the other hand, faults affecting a replica must not propagate to other correct subsystems [1]. Note that this is important since otherwise a single fault can affect the whole system. Nevertheless, this can be solved by means of *error containment*. Specifically, it is necessary to define one *error-containment regions*, i.e., regions from which errors cannot propagate to other parts of the system, for each replica. Finally, note that error containment can be used to restrict the potential failure modes of a subsystem, i.e., restrict its *failure semantics*. Restrict the failure semantics of the subsystems allows to significantly simplify the FT mechanisms of the system, as dealing with byzantine subsystems is much harder than dealing with crashed ones.

C. Fault treatment

As said above, FT is carried out by means of error processing and fault treatment [8], where fault treatment consists in preventing faults from causing errors again.

Fault treatment is achieved by means of *fault diagnosis* and *fault passivation*. Fault diagnosis aims at identifying the fault that is causing errors; whereas fault passivation aims at preventing the fault from being activated and causing errors again. Fault passivation can be achieved by preventing the faulty subsystem from participating in any other process.

As any system, systems with redundant components eventually suffer faults. When those faults are permanent and affect redundant components, even if the system tolerates the faults, it suffers a reduction of the available redundancy. Such reduction is known as *redundancy attrition*.

Redundancy preservation [6] mechanisms can be used to keep the available redundancy as high as possible. In principle redundancy preservation can be attempted by simply using fault diagnosis together with error recovery (to drive a temporarily faulty replica into a non-faulty state). This first attempt is specially important to prevent the unnecessary redundancy attrition that can happen when redundant components that are only temporarily faulty are discarded as if they were permanently faulty. However, in DESs in which subsystems must coordinate their actions, a simple error recovery may not be enough to guarantee the correct operation of a subsystem after a temporary failure. In these cases *reintegration* [10] should be used. Reintegration is similar to error recovery in the sense that it implies replacing the erroneous state of a component. However, in this case, the new state is obtained as a result of reaching an agreement with other system components.

Nevertheless, in case the fault is permanent and in some other complex scenarios it may not be possible to carry out neither recovery or reintegration of components. In these cases *restoration* can be used to prevent redundancy attrition. Restoration consists in replacing a faulty redundant component by a non-faulty one. An example of restoration would be removing a faulty replica of a task and starting a new replica either in the same node or in a different one.

From the previous discussions we can conclude that redundancy plays a key role in FT. This is so, as redundancy represents a systematic way to provide FT. Nonetheless, the amount of redundancy and how it is used depend on aspects such as the probability of fault occurrence, number of faults to be tolerated, the system's architecture, among others. Specifically, the fault model has a great impact on the type of redundancy needed.

D. Tolerating permanent faults

Permanent faults can be tolerated by means of replication based on spatial redundancy. Moreover, in DESs with hard RT requirements, faults have to be seamlessly tolerated. In these cases, active replication is typically used.

To ensure the correct operation of the system when using active replication, all non-faulty replicas must produce consistent (equivalent and often identical) results. This is known as *replica determinism* [6] and we distinguish two levels:

- Internal replica determinism in which replicas produce corresponding results, as long as they start in the same initial state and they are provided with identical inputs. This determinism is basically achieved avoiding the use of non-deterministic program constructs, e.g. random numbers, or any other technique that could cause non-deterministic decisions, such as relying on information that is only known to the local replica.
- External replica determinism in which replicas are provided with corresponding inputs to carry out their operations. This can be achieved by forcing replicas to agree on a state or set of values, e.g. by forcing replicas to exchange and vote on their state/values to reach a consensus.

An important feature related to agreement is *consistency*. This feature is desirable if not fundamental for any DES (even if it is not fault-tolerant), since a DES consists of different subsystems (replicated or not) having to have a common view of a given piece of information, e.g. the set of messages exchanged so far. To achieve consistency in a highly-reliable way, it is possible to use similar techniques as those to provide replica determinism.

E. Tolerating temporary faults

Time redundancy is specially suitable to tolerate temporary faults; as it is more cost-effective and simpler than using spatial redundancy. Time redundancy is a common technique used in the communication subsystem of DESs. The most widespread techniques are *Automatic Repeat reQuest* (ARQ) and *proactive retransmissions*.

ARQ solutions rely on the transmission of *acknowledgement* (ACK) and *negative acknowledgement* (NACK) messages to trigger the retransmission of frames when these are lost. This technique is not the most suitable to tolerate temporary faults of RT systems, as the bandwidth and time required to complete the transmission are non-deterministic. Moreover, the jitter introduced by these solutions is high and the worst-case response time has to include the transmission of the additional ACK and NACK messages, and the timeouts to detect the omissions of frames.

Proactive retransmissions consist in transmitting several copies of each frame in a preventive manner, to ensure that at least one copy reaches the destination in the presence of faults. This is a more suitable solution for RT systems as it is deterministic in the resource consumption, introduces less jitter and requires less time in the worst case.

IV. Fault Tolerance in Ethernet Networks

As introduced in section I it is essential to provide an adequate degree of reliability in each level of the system's architecture if we want to achieve adequate reliability for the overall system. In distributed systems this includes the communication subsystem. As introduced in section III, fault tolerance is key to reach a high reliability in the operation of DESs. For this reason, a multitude of communication protocols include FT mechanisms.

Networks can take advantage of almost any FT technique. From error recovery to error compensation and from fault

diagnosis to fault passivation. Moreover, these techniques can be combined to achieve highly-reliable communication services. Common implementations of these techniques are recovery of network devices, such as switches; use of alternative paths, when the main path suffers a fault; retransmission of frames; and error containment to prevent a faulty component from jeopardizing the communication among fault-free ones.

Nevertheless, as we already anticipated, not all the mentioned techniques and mechanisms are suitable for highly-reliable real-time DESs. This is so because for a system to be reliable it has to provide a correct service continuously. Thus, mechanisms that introduce failover times that prevent the system from meeting the deadlines are not suitable for said systems.

The network architecture can be divided in layers following the Open Systems Interconnection (OSI) model [11]. The OSI model helps enforcing interoperability among different communication systems and protocols. To do so, it divides the different functionalities a network can offer into layers, where each layer provides a service to the immediately upper layer. The original version of the model defines seven layers, which we will briefly introduce from the lowest to the highest.

- Physical (layer 1): responsible for the transmission of raw data bits, includes aspects such as the topology, the transmission medium and the bit-encoding.
- Data Link (layer 2): commonly referred to simply as link-layer. It is responsible for the physical addressing, controlling how components gain access to the medium, frame synchronization and error detection.
- Network (layer 3): responsible for the logical addressing and routing of frames from the source to the destination when these are not directly connected.
- Transport (layer 4): performs the end-to-end communication control, providing a certain quality of service for the communications. Mechanisms such as flow control, packet segmentation/reassembly and error management are performed in this layer.
- Session (layer 5): responsible for controlling remote actions. It establishes, manages and terminates connections between local and remote applications.
- Presentation (layer 6): maps the syntax and semantics of the application data to the ones used by the network.
- Application (layer 7): provides common services to different classes of application to use the communication network.

Although the OSI model presents great advantages, such as modularity and interoperability, it presents some drawbacks too. Specifically, each protocol at each layer will add control information to the frames forwarded down the stack. Thus, this approach causes important computation and communication overheads, which may not be bearable for RT DESs. Moreover, the services provided by certain layers may not be needed depending on the network. For instance, many fieldbuses do not need the services of the network layer as they comprise a single network. For these reasons, RT networks commonly use the OSI collapsed model. This version of the model only includes physical, link and application-layers. Some functionalities from the network and transport layers (such as routing or error management) are moved to the link-layer; whereas more sophisticated functionalities of the missing layers are moved to the application-layer.

As we focus on hard RT systems, in this survey we only include protocols that could be part of the OSI collapsed model and that are based on Ethernet. Note then, that IT protocols such as the Transport Control Protocol (TCP), the Internet Control Message Protocol (ICMP), Virtual Router Redundancy Protocol (VRRP), Transparent Interconnection of Lots of Links (TRILL), link aggregation protocols, Automatic Repeat Request (ARQ) protocols, etc. are out of the scope of this article.

Moreover, clock synchronization is key to provide hard RT guarantees in many of the protocols surveyed. More concretely, the IEC 61588 [12] standard for the Precision Time Protocol (PTP) is used by many of the protocols. Therefore, we discuss the vulnerabilities and the fault tolerance mechanisms of PTP and some of its profiles; i.e. standards based on PTP that include certain variations.

We then survey protocols that represented the first steps towards achieving high reliability in Ethernet-based networks; even though some of these protocols are not adequate to be used in highly-reliable hard real-time DESs. We later move to more complete solutions that provide real-time and reliability, and we classify them according to the degree of fault tolerance they provide. Note that during the protocols description we use the terminology chosen by the specific designers of each solution when describing their protocols. Tables I and II show the specific terminology and its correspondence with a more general one.

A. Fault Tolerance in the Precision Time Protocol

PTP is a master-slave synchronization protocol that can be used in networks that rely on frames to convey information [13]. Specifically, PTP provides the network with global clock synchronization; which means that all the systems share the same notion of time as described in Section II. To that, PTP supports the selection of a master clock to which all other clocks (slave clocks) in the network synchronize.

The first version of PTP was standardized in 2002 in the IEEE Std 1588–2002 [12] and it was revised in 2008 in the IEEE Std 1588–2008 [14]. Many profiles have been defined throughout the years to cover the needs of different networks and applications. Nevertheless, we focus on two recent profiles due to their high relevance and impact and due to their orientation for using PTP in combination with other protocols discussed later on in this document. Specifically, we discuss the IEEE Std 802.1AS–2011 [15] standardized by the Audio Video Bridging Task Group and the IEEE 802.1AS–rev [16] that is currently under revision by the Time-Sensitive Networking Task Group.

In PTP, the clock selected as master clock is critical; meaning that a failure of the master clock would cause the loss of synchronization and the system failure. For this reason, PTP's revisions and profiles have different fault tolerance mechanisms to deal with the failure of the master clock, which we will briefly discuss next.

In the IEEE Std 1588-2002 slave clocks can detect the failure of the master clock and select a new master clock

TABLE I: Terms used in the fault-tolerance protocols.

Protocol	Node	Switch	Transmitter	Receiver	Comm. Manager
STP	End System	Bridge	Transmitter	Receiver	-
SPB	End System	Bridge	Transmitter	Receiver	-
MRP	End Node	Media Redundancy Client	Source	Destination	Media Redundancy Manager
PRP	Node	Switch	Source	Destination	-
HSR	Node	-	Source	Destination	-

among the non-faulty slave clocks. Nevertheless, the detection of a master clock failure takes a non-negligible amount of time. Furthermore, the selection of a new master clock and recalculation of the synchronization is also time consuming. Actually, the whole process can take several seconds [17], and during that time the system is unsynchronized; which can result in the failure of the whole system.

To reduce the time required to detect and recover from a master clock failure, the IEEE Std 1588–2008 introduces the concept of *alternate master*. The alternate master broadcasts its time to slave clocks in parallel to the master clock. This reduces the time required to synchronize after a master clock failure, as there is no need to select a new master clock and slave clocks already have information from the alternate master when the master clock fails. Nevertheless, the failure of the master clock is still not transparent for slave clocks. Slaves must still detect the failure and synchronize to the new master clock, which still introduces non-negligible failover times [17].

The IEEE Std 802.1AS–2011 profile (AS for short) does not use the concept of alternate master. Nevertheless, it introduces mechanisms to reduce the failover time produced after a master clock failure. On the one hand, AS allows to send the synchronization frames through specific routes different from the ones used for data frames. This allows to use the optimal synchronization path and reduce the time needed to find a new master clock after a failure. On the other hand, the drift from the master clock is calculated in an accumulative manner in each slave clock. This means that each slave clock knows its relative drift from the master clock which reduces the time needed for synchronization.

Finally, to the best of the authors' knowledge, the IEEE Std 802.1AS–rev aims at supporting several active master clocks. This would make resynchronization transparent for slave clocks after a master clock failure. Moreover, each master clock will have its own path for synchronization communication, making this approach more robust in front of failures in the network components between the master clock and slave clocks.

B. Fault Tolerance Protocols over Ethernet

This section comprises different protocols that provide Ethernet with a single fault tolerance service, such as spatial redundancy. It is important to note that in this section we took into consideration certain protocols that do not meet the stringent requirements of highly critical applications. Nevertheless, these protocols are a key part of Ethernet and can be used to understand the need of further mechanisms and protocols to suit the needs of highly-critical RT DESs. Table I show the correspondence between the terminology used in each protocol and a more general one.

1) Spanning Tree Protocols: The Spanning Tree Protocol (STP) is a link-layer protocol that allows to establish a single loop-free logical topology. STP was standardised as part of Ethernet in the document IEEE Std 802.1D–1998 [18]. Nevertheless, STP was superseded by the Rapid Spanning Tree Protocol (RSTP) in the standard IEEE Std 802.1D–2004 [19].

STP was devised to be used on *bridged* (switched) Ethernet networks, with mesh topologies in which components can be connected to each other through redundant physical paths. STP is implemented on the bridges and allows them to monitor the network topology in order to decide on a unique logical path to connect each pair of *end-systems* (nodes). Specifically, STP can identify redundant links to logically disconnect them from the network and create a logical tree topology.

One bridge in the network is selected to be the root bridge. The root bridge is then used to select which ports are active and which are not in all the bridges of the network. A non-root bridge calculates which is the link with the highest bandwidth that connects it to the root and selects it as preferred link.

Bridges can detect changes in the topology, e.g. the failure of a link or the connection of a new component. Whenever a bridge detects a change in the topology, it notifies the root that, in turn, will notify all bridges in the network for them to age out their forwarding information. A new tree will be calculated. If the root bridge fails, bridges will select as the new root bridge the one with the lowest identifier.

This way, STP is a recovery protocol that enables the use of spatial redundancy for both, the network and the root bridge, as long as the network has enough redundant bridges and links. However, STP introduces a high failover time of up to 50s; which improves availability but is unacceptable in highly-reliable systems. To deal with this drawback, the IEEE proposed the Rapid Spanning Tree Protocol (RSTP).

RSTP is an update of the STP that achieves error recovery with shorter failover times. Just like STP, RSTP is a link-layer protocol that allows to create loop-free logical paths between any pair of end-systems in the network. RSTP selects which ports are active and which are not in each bridge, in a similar way STP does. However, in RSTP all bridges exchange information with their neighbors to detect which neighbor has the lowest-cost path to the root.

This allows to reconfigure the network without the root bridge intervention, which significantly reduces the time required for reconfiguration in case of failure. RSTP provides spatial redundancy with a failover time of up to 10 seconds, which significantly reduces the time required by STP, but it can still be problematic in highly-reliable systems.

The Multiple Spanning Tree Protocol (MSTP) is an evolution of RSTP that supports the deployment of more than one spanning tree on the same network; thus, allowing to use VLANs. MSTP also operates on the link-layer of the

network architecture. It is part of the IEEE Std 802.1Q–2005 standard [20]. MSTP is compatible with RSTP and STP.

The number of spanning trees created by MSTP will depend on the number of existing regions. Each MSTP tree can have a single or multiple VLANs assigned and each VLAN will use one and only one spanning tree. This allows MSTP to operate in large networks, as the number of trees is not dependent on the number of VLANs, as in other proprietary solutions such as Cisco's Per-VLAN Spanning Tree (PVST) or Rapid Per-VLAN Spanning Tree (RPVST) [21].

MSTP also supports the creation of new paths whenever a network device fails, as well as the definition of new root device whenever one fails. The failover times are similar to those provided by the RSTP; improving availability but making it not suitable for highly critical systems.

2) Shortest Path Bridging: Shortest Path Bridging (SPB) is a link-layer protocol devised to enable the creation of loop-free communication paths while supporting multipath routing. It was standardised as part of the IEEE Std 802.1aq and merged in the IEEE 802.1Q–2012 [22].

It has been devised to replace STP, RSTP and MSTP; as it can support larger mesh networks, provides shorter recovery times and allows load share across the multiple active paths. In order to prevent loops SPB relies on a control plane in which each bridge has a global view of the network. This also reduces the time required for reconfiguration in case of failure. All of this is done while guaranteeing that the route follows the shortest path tree and that the forward and reverse paths are symmetric, feature needed by certain synchronization protocols.

To achieve this, SPB creates several logical networks on top of the existing physical network. Frames transmitted by an end-system are only transmitted to the members of the same logical network. To achieve this, each logical network must have a unique identifier. *Edge bridges* (bridges attached to the nodes) are responsible for adding membership information into the frames; which will in turn be extracted from the frame at the receiving edge bridge.

SPB relies on Intermediate System to Intermediate System (IS-IS) to provide bridges with a complete view of the network. IS-IS supports IEEE Std 802.1ag [23] Continuity Check Messages, which monitors, detects and reports failures to IS-IS.

All of these make SPB a standard solution to further support active spatial replication and, therefore, meet the needs of highly-reliable systems. Nevertheless, SPB itself can not be used for RT applications, as it does not provide mechanisms to support RT data communications. As we will explain in the following section, SPB has been extended to provide support for such kind of communications.

3) Media Redundancy Protocol: The Media Redundancy Protocol (MRP) is a link-layer recovery protocol standardised in the IEC 62439 document, part 2 [24].

MRP relies on a ring topology that counts with a central element, called the Media Redundancy Manager (MRM). The MRM is the responsible for the configuration of the network and for the management of network failures. The rest of the nodes in the network are called Media Redundancy Clients (MRCs). MRCs follow the configuration established by the MRM and can detect the failure of an attached link or a neighbor device and notify about it to the MRM.

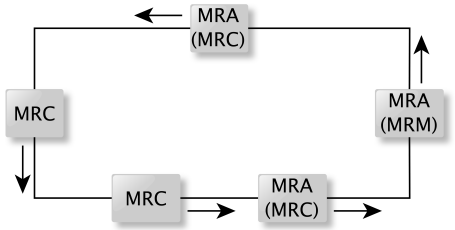


Fig. 5: Example of 5 nodes interconnected using an MRP ring. The network counts with three MRAs; one acting as the MRM and two acting as MRCs. The other nodes are simple MRCs. The arrows show the direction of frames in a fault-free state.

The revision of the standard in 2016 [25] describes a new element, called Media Redundancy Automanager (MRA). MRAs are components that can act as MRM or MRC. During network start-up, MRAs vote to decide which one will act as MRM. The selected MRA switches to MRM mode; whereas the other MRAs switch to MRC mode. Just one MRM can be active at the same time.

Figure 5 shows an MRP network comprised of 5 nodes. Three of these nodes are MRAs, from which one is the MRM and the others act as MRCs. The other two nodes are plain MRC nodes. The arrows represent the transmission of frames. To avoid loops, MRP turns the ring into a logical line topology. When the ring is fault-free, the MRM blocks a ring-port to not receive or propagate any data frames; whereas all MRCs have both ring-ports active and can receive and forward through both of them. Once a fault occurs, the MRM activates both ring-ports to re-establish the connectivity.

The MRM can detect failures in the network by transmitting a special frame periodically through one of the ports. If the MRM does not receive the frame through the other port it will detect that the ring is open and will activate its blocked port.

MRP considers the presence of more than one MRA in the network, but only one MRA can act as MRM at any time. Whenever the MRM fails, MRCs must detect its failure. If there are other MRAs in the network, they can vote and select a new MRM. Nevertheless, switching to the new MRM introduces a failover time that interrupts the communication until the switch over is completed.

MRP can tolerate one permanent fault in a ring link or MRC with failover times of up to 26.2 ms with 50 nodes. Moreover, MRP does not include any time redundancy mechanisms. This together with the fact that spatial redundancy is passive, makes MRP vulnerable to temporary faults.

Moreover, MRP does not define the failure semantics assumed for the nodes. In this sense, nodes that exhibit byzantine failure semantics could compromise the correct operation of the system. This is so as nodes are responsible for propagating frames from other nodes and could corrupt their content; e.g. introducing a bit flip that could cause the frame to be dropped by the rest of nodes. To prevent this, the failure semantics of the nodes should be restricted using additional mechanisms not devised in the standard.

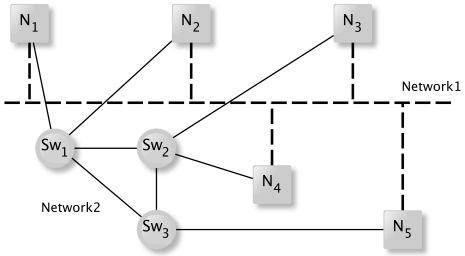


Fig. 6: Example of 5 nodes interconnected through two networks using PRP. Network 1 has a bus topology; whereas Network 2 has mesh topology.

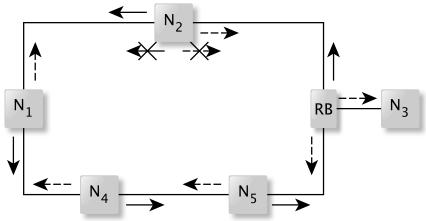


Fig. 7: Example of 5 nodes interconnected using HSR. Nodes $N_{1,2,4,5}$ have two ports through which they connect to the ring; whereas Node N_3 is attached through a RedBox. The arrows represent the transmission of a broadcast message by Node N_2 through both interfaces in parallel.

All of these make MRP an adequate protocol for systems that require high availability, but not for highly-reliable ones.

4) Parallel Redundancy Protocol: The Parallel Redundancy Protocol (PRP) is a link-layer protocol that provides spatial redundancy by connecting nodes to two independent and similar networks. The transmission of frames is done through both networks in parallel and the receiver must detect and discard duplicated frames. PRP is standardised in the IEC 62439 document, part 3 [26].

Both networks must use the same link-layer protocol but can differ in performance and topology, which can be bus, ring or mesh. Figure 6 shows 5 nodes connected using PRP; where one of the networks is a bus and the other one is mesh. This can result in different delays and in the reception of out-of-order frames. Moreover, both networks must be isolated from each other and must be fail-independent.

Nodes are modified to support the parallel connection to both networks; to duplicate frames on transmission; and to detect and eliminate duplicates on reception. Nevertheless, commercial off the shelf (COTS) nodes can be attached to a single network or they can be attached to both networks using a switching device called RedBox. Moreover, RedBoxes also allow connecting a duplicated network to a simple network.

The redundant ports of each modified node and RedBox have the same MAC and a single set of IP addresses. This way, frames transmitted through both ports are exact duplicates. Moreover, this also allows each network to operate without having to be aware of the existence of the other network.

PRP includes a mechanism to detect and discard frame duplicates. To do it, PRP relies on sequence numbers. Basically, each frame is assigned a sequence number. Whenever two frames received have the same sequence number and the same MAC address, the receiver knows they are the same frame and discards the duplicate.

This mechanism allows PRP to tolerate permanent faults in the networks seamlessly, i.e. with a failover time equal to 0. The number of failures that can be tolerated in each network will depend on the number of redundant paths. Moreover, this active replication allows tolerating temporary faults in one network, as long as they are not concurrent to permanent faults in the equivalent devices of the other network.

Moreover, PRP is compatible with protocols for error recovery such as STP or RSTP. Therefore, these protocols can be used when a permanent fault affects a network device of one of the redundant networks to establish new paths and prevent redundancy attrition. Note that even though this recovery takes time, frames will continue to flow through the independent parallel network.

PRP does not describe any mechanisms to restrict the failure semantics of the nodes or switches. Therefore, the system is vulnerable to byzantine behaviours that could cause inconsistencies, e.g. a node could send frames with different content through each redundant port, causing some nodes to receive the right information and others the wrong one.

PRP provides a certain degree of reliability, but it is not suitable for highly-reliable networks. Furthermore, it does not provide mechanisms to support RT data communications and can not be deployed in RT DES by itself. Nevertheless, note that some of the protocols described in subsection IV-C use PRP to provide fault tolerance.

5) High-Availability Seamless Redundancy: High-Availability Seamless Redundancy (HSR) is a network protocol that, like PRP, aims at providing zero failover time by means of spatial redundancy. It is a link-layer protocol and is also standardised in the part 3 of the IEC 62439 [26].

HSR is based on a ring topology, to which nodes are directly attached through two ring ports. Each node acts also as a switch for the traffic received from other nodes. Figure 7 shows an HSR network where Node 2 transmits a broadcast message. Just like in PRP, the redundant ports of a node share the same MAC and the same set of IP addresses. This simplifies the identification of frames coming from the same node as duplicates.

To tolerate one permanent fault, nodes send frames through both ring ports at the same time. Multicast frames are read by the destination nodes and forwarded through the other ring port; unicast frames are not forwarded by the destination node.

To prevent frames from indefinitely circulating the network, nodes do not forward frames through a port if they already forwarded said frame through said port. To preserve the ring topology, COTS nodes can only be connected to the ring through a RedBox that acts as proxy for the node and as switching device for the messages traversing the ring.

HSR can also support the connection of multiple rings. Two rings can be connected through a QuadBox, device used to forward frames from one ring to the other. Even though

TABLE II: Terms used in the fault tolerance and real-time protocols.

Protocol	Node	Switch	Stream	Communication Cycle	Communication Window	Transmitter	Receiver	Time-Triggered	Event-Triggered
HSE FF	Device	Switch	-	Macrocycle	-	Transmitter	Receiver	Scheduled	Unscheduled
PROFINET	I/O-controller, I/O-device	-	Application Relation	Bus Cycle	Channel	-	-	Time-Triggered	Isochronous Real-Time
SERCOSIII	Slave	-	-	Cycle	Channel	Transmitter	Receiver	Time-Triggered	Event-Triggered
EtherCAT	Slave	-	-	Cycle	-	Transmitter	Receiver	Cyclic	Acyclic
EPL	Controlled Node	-	-	POWERLINK Cycle	Phase	-	-	Isochronous	Asynchronous
AFDX	End-System	Switch	Virtual Link	-	-	Transmitter	Receiver	Scheduled	-
AeroRing	-	T-AeroRing	Contract	-	-	Source	Destination	-	Event-Triggered
TTEthernet	End-System	Switch	-	Cluste Cycle	-	Transmitter	Receiver	Time-Triggered	Rate-Constrained
AVB	Station	Bridge	Stream	-	-	Talker	Listener	-	Time-Sensitive
TSN	Station	Bridge	Stream	Cycle	Window	Talker	Listener	Scheduled	Time-Sensitive
FTTRS	Node	Switch	Stream	Elementary Cycle	Window	Publisher	Subscriber	Synchronous	Asynchronous

one QuadBox is enough to connect two rings, the standard recommends using two Quadboxes to avoid introducing a SPoF. Quadboxes apply the same technique as nodes to prevent flooding the network with several copies of the same frame.

Moreover, HSR also allows to connect a ring to a PRP network, using a RedBox for each PRP network. These RedBoxes must be capable of transforming PRP frames into HSR frames and vice versa. Moreover, it must allow to identify replicas received through each RedBox, as PRP nodes will send one copy of each frame through each network.

Actually, HSR allows to create mesh topologies as long as all nodes involved are QuadBoxes. Nodes will forward frames through all the ports, except for the one through which the frame was received, and those ports will forward the frame except if they already forwarded said frame.

Moreover, even though HSR does not define specific mechanisms to enforce hard real-time response, it does suggest the separation of traffic in classes and the allocation of specific moments in time to transmit each type of traffic.

HSR does not include any time redundancy mechanisms to tolerate temporary faults. Thus, temporary faults can only be tolerated using the existing spatial redundancy, as long as such redundancy is available.

HSR supports time synchronization among the nodes of the system. Even though it does not specify the protocol to be used for this purpose, it is fully compatible with PTP. Note that as discussed in Subsection IV-A, the available versions of PTP are vulnerable to master clock failures, as it introduces non-negligible failover times.

Finally, HSR does not define any error containment mechanisms, other than detecting frames that traverse a node several times. This could lead to the propagation of errors from one faulty-node to the rest of the network. This is specially critical as HSR does not include mechanisms to restrict the failure semantics of the components. Thus, byzantine nodes could introduce inconsistencies that could have catastrophic consequences in the system's operation. These reasons make HSR not adequate to support high reliability, but provides certain degree of reliability.

C. Fault Tolerance and Real Time Protocols over Ethernet

In this section we describe protocols that offer all the services needed to deploy them in fault-tolerant real-time DES. We describe protocols that provide different real-time guarantees and reliability levels and can be used in different systems. Table II shows the equivalence between the terminology used by the designers of each protocol and a more general one.

1) FOUNDATION Fieldbus High-Speed Ethernet: FOUNDATION Fieldbus (FF) is an application-layer communication protocol designed by the Fieldbus FOUNDATION to suit the needs of industrial systems. FF counts with a specification on top of High Speed Ethernet (HSE) included to support manufacturing applications. It is stacked on top of TCP-UDP/IP. This allows to use standard Ethernet on the link and the physical layer. HSE FF is standardised in the document IEC 61158 [27].

The FF H1 bus provides hard real-time guarantees for the underlying communications. To do so, applications rely on holistic scheduling to enforce the timing requirements of each application. Each H1 bus counts with a scheduler that polls the transmission of scheduled (time-triggered) traffic, but also nodes can request transmissions.

HSE FF can be used as a backbone to connect FF buses to each other. This is done through the HSE Linking Device (LD). Nevertheless, this connection does not offer any RT guarantees as the connected FF buses do not share the scheduler.

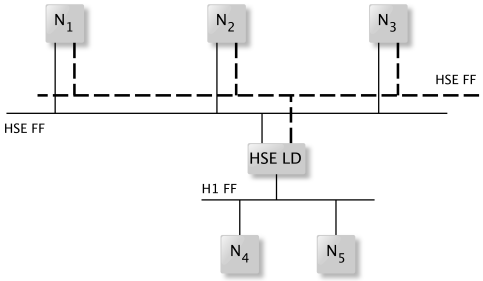


Fig. 8: Example of 5 nodes interconnected using HSE FF and H1 FF. The HSE FF is duplicated to tolerate permanent faults.

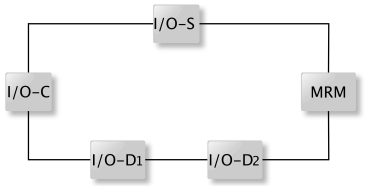


Fig. 9: Example of 5 nodes interconnected using PROFINET, where redundancy is managed using MRP. The network counts with an I/O-Controller, an I/O-Supervisor, two I/O-Devices and one MRP MRM (Media Redundancy Manager).

HSE FF supports spatial redundancy by allowing to have two completely independent networks. Devices must have two HSE interfaces to connect them to both networks. This allows to tolerate at least one permanent fault in any network component with zero failover time. Moreover, even though HSE FF does not include any time redundancy mechanisms to tolerate temporary faults, these can be tolerated using spatial redundancy as long as it is not degraded by permanent faults. Figure 8 shows a redundant HSE FF network, connected to a non-redundant H1 FF bus through an HSE LD.

Moreover, HSE FF supports device redundancy, using passive replication. If the primary device fails, the backup device becomes active. Both devices need to share the configuration to ensure fast and fault-free switch-over in case the primary device fails. Nevertheless, HSE FF does not define mechanisms to carry out the switch-over nor to ensure replica determinism.

HSE FF provides mechanisms to monitor the status of the network. Each device and HSE LD contains a component that keeps track of the healthy and faulty network components and connected devices. Each component monitors the network independently, by sending and receiving redundancy diagnostic messages. These messages are used to track the health of redundant devices and ports; and allows the device to change the transmission port in consequence.

Finally, to the best of the authors' knowledge HSE FF does not propose mechanisms to restrict the failure semantics of devices or to contain the errors that may derive from faulty ones. Thus, HSE FF provides reliability, but is not suitable for highly-reliable networks.

2) PROFINET: PROFINET stands for PROcess Field NET and it is a link-layer protocol that supports the transmission of real-time traffic over standard Ethernet. It was specifically designed for industrial applications and is standardised in IEC 61158 and IEC 61784 [27].

PROFINET supports the use of ring, star and tree topologies. To do so, nodes are always connected to the network through a switch; which can be either embedded in the device or a separate component. To provide fault tolerance, PROFINET relies on ring topologies.

It divides the communication in cycles, called bus cycles. Each cycle is in turn divided into three channels, namely the Real-Time (RT), the Non-Real-Time (NRT) and the Isochronous Real-Time (IRT) channel. It follows a provider/consumer model for data exchange.

Each channel uses a different policy to provide timing guarantees. The RT channel uses priority-based scheduling, in which the RT traffic has the highest priority to prevent it from being blocked. Moreover, IRT traffic has tighter timing constrains and, to meet them, this traffic is scheduled following a cyclic executive scheduler. This allows to avoid collisions and buffering, but it requires dedicated hardware and a tight synchronization among devices. UDP/IP traffic is transmitted if there is time available after the IRT slots, in the NRT channel.

PROFINET differentiates three types of devices: I/O-Controller, which controls the automation task; I/O-Device, a field device that receives commands from the I/O-Controller; and I/O-Supervisor, which configures and monitors the network.

Each I/O-Device can diagnose its own faults (such as defective voltage) and inform the I/O-controller. The information related to faults is transmitted as an alarm. Moreover, I/O-devices take advantage of the scheduled communication to detect errors in the providers.

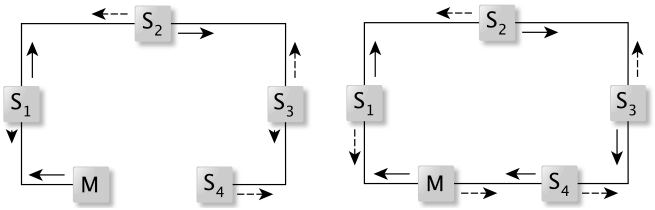
PROFINET defines methods to use ring topologies to tolerate permanent faults. More precisely, it defines the use of the previously described MRP if failover times can be accepted and the use of Media Redundancy for Planned Duplication (MRPD) otherwise. MRPD relies on a ring topology to provide fault tolerance against permanent faults. Figure 9 shows a PROFINET network with a ring topology managed by MRP.

On the other hand, PROFINET does not include any time redundancy to tolerate temporary faults. Thus, it can only tolerate temporary faults if two networks work in parallel, as long as there are no permanent faults.

Even though PROFINET relies on master/slave clock synchronization to ensure that the real-time requirements of the network are met, it does not include any mechanisms to provide the master clock with fault tolerance. Thus, the master clock represents a SPoF.

Finally, to the best of the authors' knowledge PROFINET does not include error containment mechanisms or restrict the failure semantics of the devices. Thus, a device with byzantine failure semantics could interfere with the correct operation of the system. For these reasons, PROFINET is a suitable protocol to provide safety to the network, but not high reliability.

3) SERCOS III: SERCOS stands for SErial Real-time COmmunications System. SERCOS III is the third generation and it is an automation bus based on Ethernet. It is a data



(a) Example of 5 nodes connected using SERCOSIII logical ring topology.

(b) Example of 5 nodes connected using SERCOSIII physical ring topology.

Fig. 10: Examples of 5 nodes connected using the logical and physical ring topologies proposed in SERCOSIII. The M box represents the Master; whereas the S_i boxes represent the slaves that communicate.

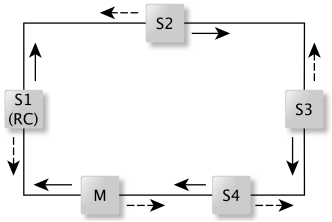


Fig. 11: Example of 5 nodes interconnected using EtherCAT. The network counts with an EtherCAT master, a slave that also acts as reference clock (RC) and three regular slaves.

link-layer protocol that provides real-time guarantees and low bandwidth consumption to support automation applications. It is standardised as part of the IEC61784 standard part 2 [28].

SERCOS follows a master/multi-slave architecture; in which the master manages the communication among slaves. SERCOS III supports line and ring topologies, as shown in Figure 10. Nevertheless, regardless of the physical topology SERCOS III enforces a logical circular topology. To do so, each slave sends the telegrams (frames) received in one of its ports through the other one. In a line topology, the last slave will send the telegrams through the receiving port once they are processed, as shown in Figure 10a. In a ring topology, the last slave is connected to the master and the master sends each telegram in both directions simultaneously, as shown in Figure 10b.

SERCOS divides the communication time into cycles, and each cycle is in turn divided into two parts, called channels. The first channel is devoted to the transmission of real-time time-triggered traffic; the second channel is used for the transmission of event-triggered traffic with no real-time requirements. The channels are isolated to provide RT guarantees to TT traffic.

Moreover, the master transmits a telegram that contains the schedule for that specific channel. After that, the master transmits the Acknowledge Telegram (AT), which is populated by the slaves that are scheduled with the data they need to transmit. Slaves process the telegrams on-the-fly, reducing the time required to carry out the data exchange.

SERCOS III relies on clock synchronization to ensure real-time capabilities. The master is used as a reference clock and transmits synchronization information at the start of every cycle. Even though it is clear that the master is key for the operation of the system, SERCOS III does not include any mechanisms to eliminate the SPoF it represents.

Moreover, the telegram used to synchronize and trigger the communication is a SPoF too. This is so because if the telegram is lost, nodes can not communicate during the first channel. This is specially critical, as no time redundancy mechanisms are included to tolerate the loss of said telegram.

The ring topology allows the system to tolerate a single permanent fault affecting a link or a slave. Any slave can detect the failure of a neighbor slave in less than one cycle time and start transmitting as in a line topology. Moreover, SERCOS III supports the hot-plugging of devices and the same mechanism can be used to reintegrate components after a fault.

The use of a single telegram to transmit the data of all slaves helps ensuring data consistency among slaves, as all slaves will receive the AT with the same information when the last slave sends it back to the master. However, a slave with byzantine failure semantics could corrupt the information of any node. Moreover, when using a ring topology, a node could introduce different information in the ATs that it receives in opposite directions, creating inconsistencies.

Moreover, SERCOS III supports oversampling to allow slaves to send several values in a single AT every cycle. Nevertheless, as all the values are sent in a single AT, it does not provide any benefits in front of temporary faults. Thus, SERCOS III can only tolerate a single temporary fault as long as the network does not suffer any permanent faults.

For the reasons discussed above, SERCOS III is a suitable protocol to provide safety, but not high reliability.

4) EtherCAT: EtherCAT stands for Ethernet Control Automation Technologies. It is a data link-layer protocol that provides real-time guarantees and high bandwidth efficiency for automation applications. It is standardised as part of the IEC61158 standard part 1 [27].

EtherCAT is based on a master/multi-slave communication architecture; in which the master manages the slaves. EtherCAT supports line, tree, star and ring topologies. Nevertheless, regardless of the physical topology EtherCAT enforces a logical circular topology. To do so, each slave sends the frames received in one of its ports through the other one. In a line topology, the last slave will send the frames through the receiving port once they are processed. In a ring topology, the last slave is connected to the master and the master sends each frame in both directions simultaneously, as shown in Figure 11.

As mentioned, in EtherCAT the master manages the communication. To do so, it transmits an Ethernet frame that is then processed and populated by the slaves on-the-fly; reducing the time required for transmitting and receiving. This way, the master in EtherCAT can be implemented using COTS Ethernet hardware, but slaves must use specialized hardware to process frames on-the-fly.

EtherCAT relies on clock synchronization to ensure real-time capabilities. Similarly to the communication, the EtherCAT master is responsible for managing the synchronization among slaves. To do so, it periodically transmits a special synchroniza-

tion frame. Nevertheless, it is not the reference clock. Instead, the first slave attached to the master is the reference clock. When the reference clock receives the synchronization frame, it writes its local time on it. The rest of the slaves use this information to resynchronize. Note that synchronization frames are only transmitted through one port, even in ring topologies.

In EtherCAT the master is key for the correct operation of the system, as it is responsible for triggering both, the communication and the synchronization. Nevertheless, there are no mechanisms to tolerate its failure. Moreover, the frame used to synchronize the slaves is a SPoF. This is specially critical, as no time redundancy mechanisms are included to tolerate the loss of said frame.

When connected in a ring topology, the master transmits the Ethernet frame for communication through both ports simultaneously. Slaves populate the first master frame they receive and forward the redundant one without information. This allows the system to seamlessly tolerate a single permanent fault affecting a link or a slave.

The use of a single frame to transmit the data of all slaves helps ensuring data consistency among slaves. However, a slave with byzantine failure semantics could corrupt the information of any node. Moreover, when using a ring topology, a node could introduce different information in the master frames that it receives in opposite directions, creating inconsistencies. Moreover, temporary faults in links could corrupt the frame, cause the loss of the information transmitted by all the slaves in a given cycle.

Moreover, EtherCAT provides mechanisms to detect the failure of a single slave and to enforce retransmissions. This allows to tolerate temporary faults in the slaves. Nevertheless, this mechanisms rely on information provided by the slave. Therefore, it may not be possible to detect the failure of a slave that fails with byzantine mode, as it can introduce incorrect status information.

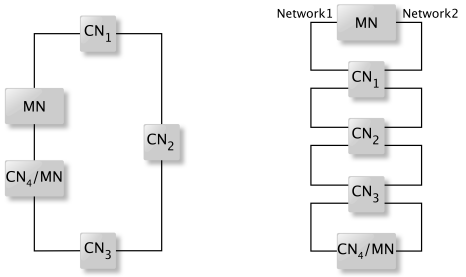
For the reasons discussed above, EtherCAT is a suitable protocol to provide safety, but not high reliability.

5) Ethernet Powerlink: Ethernet Powerlink (EPL) [29] is a link-layer protocol that extends Ethernet's link-layer with a scheduler to support hard real-time communications. It is designed to support real-time applications. The protocol can be implemented in software, using standard Ethernet hardware to enable interoperability with standard Ethernet devices.

It is included to operate in bus topologies, supporting the use of hubs, but it can operate in ring, star, tree or daisy-chained topologies. The use of switches is not recommended as queuing delays are not considered by the scheduler and could jeopardize the timeliness of real-time traffic.

EPL is based on a master/slave architecture, in which the master is called Manager Node (MN) and the slaves are called Controlled Nodes (CNs). The MN enforces synchronization in the network and controls the communication among the CNs.

EPL provides hard real-time guarantees for time-triggered traffic, but not for event-triggered traffic. ET traffic can be transmitted with no timing guarantees. To support both, real-time and best-effort traffic, EPL divides the communication in cycles that are, in turn, divided into three phases. The first phase is used by the MN to synchronize the CNs. The second phase is divided in slots, each assigned to a CN. The transmission in each slot is triggered by the MN. In the third phase, the MN grants access to the channel to a node to transmit ET traffic.



(a) Example of 5 nodes connected using an EPL ring topology.

(b) Example of 5 nodes connected using two independent daisy-chained EPL networks.

Fig. 12: Examples of 5 nodes connected using a ring and two independent networks using EPL. The MN node represents the Manager Node, the CN_i nodes represent the Controlled Nodes that communicate; and the CN_4/MN node acts as a Controlled Node and a passive replica of the Manager Node.

EPL provides fault tolerance against permanent faults. Specifically, it supports two types of channel redundancy, as shown in Figure 12:

- Ring redundancy: nodes are daysichained into a ring, this way, when one line (link) is affected by a permanent fault the ring topology becomes a line topology, maintaining the communication. However, this change requires the time equivalent to one cycle to become effective and, therefore, the redundancy is not seamless. Figure 12a shows an EPL ring topology.
- Independent networks: as in HSR, PRP and many other protocols, EPL allows to connect the nodes to two failure-independent networks, providing seamless redundancy. Figure 12b shows an EPL replicated network.

Moreover, EPL describes redundancy mechanisms of the Management Node, to avoid the SPoF a simplex MN would represent. More precisely, EPL uses passive replication of the MN, allowing to have more than one MN in the network. A back-up MN is seen as a CN by the active MN, nevertheless, it continuously monitors the network to ensure that the switch-over can be done on-the-fly when the active MN fails. This mechanism allows to ensure that the network continues to operate correctly even if the first MN fails.

EPL does not use time redundancy to tolerate temporary faults in the channel. Moreover, to the best of the authors' knowledge, there are no mechanisms to support the time redundancy of the synchronization frame, which is a critical frame as it is used to synchronize the start of the cycle in all the nodes.

Finally, EPL counts with the POWERLINK Safety protocol, which can detect errors in the communication to avoid catastrophic consequences. POWERLINK Safety is an error detecting protocol as it does not aim at tolerating faults but at

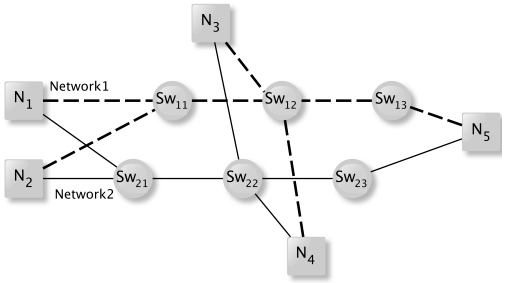


Fig. 13: Example of 5 nodes interconnected using two independent switched-Ethernet networks using AFDX.

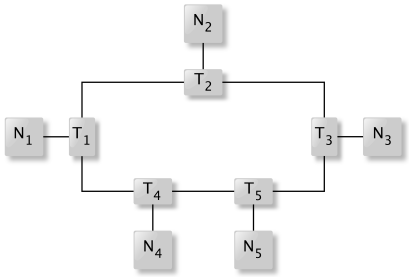


Fig. 14: Example of 5 nodes interconnected using Aeroring. Each node is connected to the ring using a T-AeroRing. Reproduced as in [32].

detecting them and reporting them for the system to evolve to a potential safe state.

6) Avionics Full-Duplex: Avionics Full-DupleX (AFDX) is a communication protocol originally developed by Airbus that aims at providing hard real-time and high-reliability over Ethernet networks for airplane control communications. To do so, AFDX proposes a series of mechanisms to extend Ethernet's link-layer. Nowadays, AFDX has become an ARINC standard and is specified in part seven of ARINC 664 [30].

AFDX relies on the use of two parallel fail-independent switched-Ethernet networks to carry out the communication [31]. These networks must have the same topology. Figure 13 shows an example of 5 nodes connected using redundant networks managed by AFDX. Each frame transmitted by a node is transmitted through the two parallel networks. Receivers must identify and eliminate duplicates upon reception. Moreover, AFDX guarantees that frames are delivered in the same order they were transmitted.

AFDX provides bounded latencies for the transmission of data traffic. To do so, AFDX relies on resource reservation. This guarantees the availability of resources during the communication and limits the bandwidth used by each transmitter. Resources must be reserved for each type of traffic a node wants to transmit. Moreover, resources must be reserved off-line and can not be changed in runtime.

The spatial redundancy allows to tolerate permanent faults in any of the redundant networks with zero failover time. On top of that, to support reconfiguration after a failure in the network, AFDX supports the existence of passive alternative paths that are also scheduled off-line and can be activated in the event of a permanent fault in the active path.

Moreover, AFDX does not use time redundancy to tolerate temporary faults. Thus, temporary faults can only be tolerated as long as the network is not affected by any permanent faults.

AFDX switches include mechanisms to enforce error containment. More precisely, in case a node or switch fails to meet the bandwidth restrictions, switches will prevent the traffic from flooding the network. Nevertheless, to the best of the authors' knowledge, AFDX does not describe mechanisms to deal with byzantine failures. Thus, to prevent inconsistent behaviours that may jeopardize the system's operation, the failure semantics of nodes should be restricted.

For the reasons discussed above, we can conclude that AFDX has mechanisms to provide the network with a certain degree of reliability. Nevertheless, to achieve high reliability must be combined with other protocols to prevent inconsistencies.

7) AeroRing: AeroRing is a communication protocol compatible with AFDX that relies on a ring topology to provide tolerance in front of permanent faults [32]. As AFDX, AeroRing relies on Ethernet, and is designed to support airplane control communications.

In order to support ring topologies, AeroRing defines a specific component called T-AeroRing, a three-port switch that connects to an end node through one port and to the ring through the other two ports. Figure 14 shows an example of AeroRing network with 5 nodes connected through T-AeroRings.

AeroRing defines four traffic classes. To provide hard RT guarantees Aeroring relies on cut-through switches with static priorities; which allows to isolate the different classes of traffic.

T-AeroRing components implement traffic shapers to enforce the requirements established for each type of traffic. Moreover, AeroRing uses traffic policing to ensure that each traffic class consumes the right resources. If a T-AeroRing detects that a traffic class consumes more resources than entitled it discards the excess frames.

T-AeroRings send critical traffic through both ring-ports in opposite directions, making it possible to tolerate the permanent fault of a single T-AeroRing or link. Non-critical traffic is sent through a single port, that corresponds to the shortest path to the destination.

Moreover, AeroRing describes mechanisms to allow T-AeroRings to detect the failure of a network device. In this way, the other T-AeroRings can change the routing tables accordingly to ensure that the transmission of non-critical traffic is re-established. Moreover, T-AeroRings also support the reintegration of network devices.

AeroRing supports another level of redundancy by allowing devices to be connected to two independent and active rings. To that, the nodes should have two ports, each connected to one of the rings. In this case, additional frame replica detection and elimination mechanisms must be implemented in the nodes to discard replicated frames received through both rings.

AeroRing does not use time redundancy to tolerate temporary faults. Thus, temporary faults can be tolerated as long as permanent faults do not affect any network device. When using

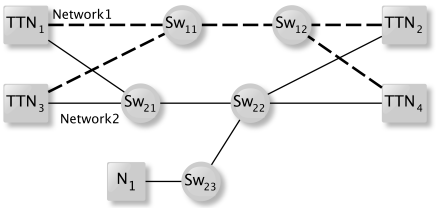


Fig. 15: Example of 5 nodes interconnected using TTEthernet. The TTN_i nodes represent the Time-Triggered nodes, which are interconnected through two independent networks. The N_1 node represents a COTS node that is connected to the rest using a single network.

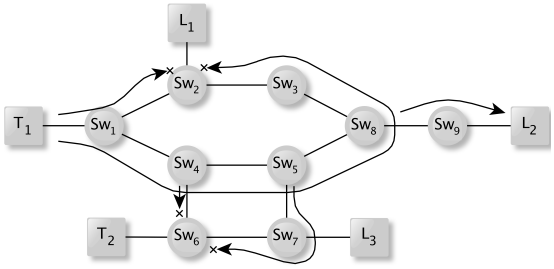


Fig. 16: Example of 5 nodes interconnected using spatial redundancy for AVB. The arrows show the logical paths established to communicate the talker T_1 to the listener L_2. Reproduced as in [36].

two rings at the same time, one permanent and one temporary fault that happen simultaneously can be tolerated.

In order to prevent frames from traversing the network several times T-AeroRings discard the frames they sourced. Moreover, T-AeroRings can detect frames with erroneous source addresses by comparing them to the routing table. This allows to eliminate impersonations. Nevertheless, to the best of the authors' knowledge AeroRing does not include the use of mechanisms to eliminate two-faced behaviours in the T-AeroRings. For these reasons, AeroRing provides the network with reliability, but not high reliability.

8) Time-Triggered Ethernet: Time-Triggered Ethernet (TTEthernet) is a real-time protocol compatible with IEEE 802.3 Ethernet. It provides link-layer services to support time-triggered hard real-time traffic over Ethernet devices [33] [34]. It is designed to support safety applications and cyber-physical systems.

TTEthernet is based on switched Ethernet and supports both star and ring topologies. Moreover, it proposes to use redundancy on the network to tolerate the permanent fault of any *system*. In order to do that, it provides support for using replicated star topologies. Figure 15 shows a TTEthernet network with two stars used to tolerate faults.

TTEthernet supports different classes of traffic, with different timing and reliability guarantees. Time-Triggered scheduled traffic; Rate-Constrained (event-triggered with bounded transmission rate) traffic; and Best-Effort standard Ethernet traffic with no guarantees.

In TTEthernet the communication follows a predefined schedule, which allows to provide hard RT guarantees to the communication. Moreover, TTEthernet relies on clock synchronization. Specifically, on a clock synchronization that is based on a convergence function; i.e. the time reference is calculated as a function of the values of different physical clocks. Thus, the synchronization mechanism is fault-tolerant, as it does not rely on a central component.

TTEthernet supports redundancy in the switches and links; eliminating any possibly existing hardware SPoF. Actually, a TTEthernet end-system (node) can have up to three different ports, each attached to a separate network. Being more specific, TTEthernet supports two different replicated topologies:

- Replicated Star: it can support up to three independent stars connecting any pair of nodes. TT and RC traffic is transmitted in parallel through all the available stars. Replicas of the traffic are eliminated at the receiving end-system. This topology allows to tolerate the failure of any network device.
- Ring: TTEthernet-switches are connected forming a ring and each end-system is connected directly to its own switch, which acts as its physical interface with the rest of the network. Switches forward TT and RC frames through both ring-ports; and eliminate replicas and forward one to the attached node. This topology allows to tolerate the failure of a single inter-switch link.

On the other hand, TTEthernet does not provide any time redundancy mechanisms to tolerate temporary faults in the links. Thus, temporary faults can only be tolerated as long as the spatial redundancy is available.

Finally, switches are provided with error containment mechanisms to prevent faulty systems from jeopardizing the correct operation of the overall system. More precisely, the error containment mechanisms proposed in TTEthernet can eliminate byzantine behaviours; such as two-faced behaviours or impersonations. For all these reasons, TTEthernet provides high reliability to the network.

9) Fault Tolerance over Audio Video Bridging: Audio Video Bridging (AVB) is the first generation of IEEE standards that aimed at providing Ethernet with real-time capabilities. These standards operate at the link-layer and were devised to support the transmission of audio and video traffic [35].

Even though AVB standards can not provide timing guarantees on their own, they provide the means to do so. In fact, due to the relevance and the impact of these standards there are several analyses to provide AVB with timing guarantees. Some examples of this can be found in [37]–[40].

Given that AVB is standard Ethernet, it supports mesh topologies. Nevertheless, there are restrictions to the size of the network imposed by certain standards. For instance, the clock synchronization standard only guarantees synchronization accuracy under 1μs for systems up to 7 hops apart. For systems that require highly-accurate synchronization this could lead to a violation of the timing guarantees in larger networks.

To provide soft RT guarantees while keeping the plug&play nature of Ethernet, AVB includes the IEEE Std 802.1Qat [41] Stream Reservation Protocol (SRP). SRP allows end-systems

(nodes) that want to communicate to reserve resources along the path that connects them. In this way SRP reduces the probability of frame loss, as there is no possibility of buffer overflow; and increases the determinism in the end-to-end delay.

AVB is comprised of two more standards to enforce real-time guarantees. The first one is IEEE 802.1Qav [42], which describes what is called the Credit-Based shaper; and the second one is the already introduced IEEE Std 802.1AS–2011 [15], which describes a clock synchronization mechanism.

AVB was designed to provide soft real-time guarantees, but not high reliability via FT. Nevertheless, the interest in extending AVB to provide services for automation and automotive applications motivated the design of fault tolerance mechanisms. Specifically, SRP was extended to support the transmission of frames through several paths in parallel.

To this end, in [36] the authors propose a mechanism to allow SRP to establish redundant streams. These redundant streams are registered whenever two or more end-systems signal their wish to communicate. Figure 16 shows an example of the redundant logical paths established using the mechanism to interconnect T_1 and L_2.

Nevertheless, in order to manage the redundancy during the transmission an additional redundancy control protocol needs to be used. If RSTP is used there is still a failover time whenever a network device fails, as the new logical path needs to be established before frames can be forwarded through it. Nevertheless, if used together with other redundancy control protocols, such as PRP or HSR, the failover time is zero, as the redundant paths are active simultaneously.

Regardless of these efforts to increase the reliability of AVB networks, there are some limitations to their deployment in critical systems. For instance, as discussed in Subsection IV-A, the IEEE Std 802.1AS standard is vulnerable to failures in the master clock.

Moreover, AVB does not include any time redundancy mechanisms to tolerate temporary faults in the links. Thus, if passive replication is used for spatial redundancy, temporary faults in a link could cause the loss of frames.

Finally, there are no mechanisms to restrict the failure semantics of the devices or to achieve error containment. Therefore, inconsistencies in the communications may happen, among other potential error scenarios. For the discussed reasons, AVB provides availability, but not high reliability.

10)Time-Sensitive Networking: Time-Sensitive Networking is a Task Group of the IEEE that is currently working on the standardization of hard real-time, high-reliability and on-line configuration services for standard Ethernet [43]. The set of standards developed by this group are usually referred to as Time-Sensitive Networking (TSN) too. TSN is an evolution of AVB, which aims at providing mechanisms to support automation, control and automotive networks.

In TSN each standard defines a service and these services can be combined to create tailored networks that meet the requirements of a great variety of applications; including the ones supported by AVB. Similar to AVB, TSN supports mesh topologies, but like in AVB there are restrictions to the accuracy of clock synchronization achievable in systems separated by more than 7 hops.

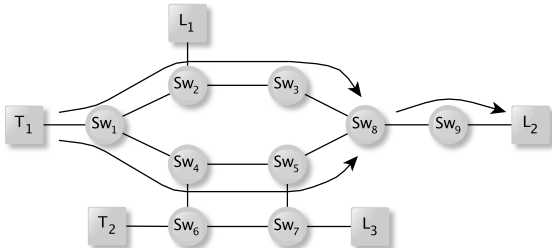


Fig. 17: Example of 5 nodes interconnected using TSN. The arrows show the logical paths established to communicate the talker T_1 to the listener L_2.

TSN supports time-triggered hard real-time traffic with the standard IEEE Std 802.1Qbv [44] Time-Aware Shaper; event-triggered soft real-time traffic is supported with the AVB standard IEEE Std 802.1Qav Credit-Based Shaper. Other traffic shapers for TT traffic are defined in the standard IEEE Std 802.1Qch [45] Cyclic Queuing and Forwarding and IEEE Std 802.1Qcr [46] Asynchronous Traffic Shaper.

To provide the timing guarantees required by control applications, TSN relies on clock synchronization. As it was already introduced in Subsection IV-A, the TSN Task Group is currently working on a revision for the IEEE 802.1AS standard to provide the clock synchronization mechanism with higher reliability.

Moreover, the standard IEEE Std 802.1Qcc [47] is an evolution of the IEEE Std 802.1Qat SRP. Qcc introduces mechanisms to carry out the reconfiguration of the network in a centralised manner. Specifically, Qcc describes two new configuration models; the *Centralized Network/ Distributed User* (CN/DU) and the *Fully Centralized* (FC) models.

The CN/DU model proposes the use of a Centralized Network Configuration (CNC) entity, responsible for managing and configuring the network. Stations can transmit their network-related requirements to the bridge they are attached; which will in turn forward it to the CNC. Finally, the CNC process all the requests and distributes the modifications through the network.

On the other hand, the FC model proposes the creation of the Centralized User Configuration (CUC) entity, responsible of managing the station requirements. The CUC can receive application-related requirements and transform them into network-related requirements for the CNC to process them.

These models allow to significantly reduce the time required for reconfiguration, as well as to support new enhanced services for reliability. Nevertheless, the specification of the CNC and the CUC is out of the scope of the standards. Moreover, to the best of the authors' knowledge there are no mechanisms to support their replication. Therefore, the centralised architectures introduce new SPoFs.

In order to increase the reliability of data frames, the TSN Task Group proposed three different standards. The first two standards are concerned with spatial redundancy. More precisely, the IEEE Std 802.1Qca [48] allows to establish multiple redundant logical paths to connect nodes that want to communicate; whereas IEEE Std 802.1CB [49] allows to

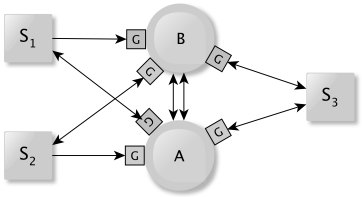


Fig. 18: Example of 3 nodes interconnected using FTTRS. A and B represent the two HaRTES switches; the S_i boxes represent the communication nodes and the G boxes represent the port guardians.

create redundant streams on top of the logical paths created by Qca. Specifically, Qca is an extension of the SPB that supports seamless redundancy and resource reservation. Figure 17 shows an example of the redundant logical paths established using Qca and CB to interconnect T_1 and L_2. It is important to note that the CB standard is independent from the IEEE Std 802.1Q, in the sense that it is not an amendment nor a revision of the latter. Thus, CB can be used to provide spatial redundancy to other protocols.

There can be as many redundant logical paths as the physical topology allows. Thus, the number of permanent faults that can be tolerated will depend on the physical topology. Moreover, Qca also allows to establish a new logical path whenever a network device fails, reducing the redundancy attrition. Furthermore, as long as there are several active paths, this reconfiguration will be done seamlessly. Nevertheless, CB introduces the possibility of frames arriving in a different order they were transmitted.

TSN does not include any time redundancy mechanisms specifically designed to tolerate temporary faults on the network. Therefore, in order to tolerate both, temporary and permanent faults happening simultaneously, there must be at least three redundant paths to connect any pair of nodes.

The last standard, IEEE Std 802.1Qci [50] Per-Stream Filtering and Policing defines two mechanisms for error containment. The first one allows to identify and eliminate frames that arrive out of time, to prevent them from interfering with the rest of the traffic. The second mechanism allows to detect components failing as babbling idiots, i.e. components that get stuck transmitting a frame over and over again. Frames arriving from the failing component are dropped to prevent them from flooding the network.

Nevertheless, TSN does not propose any mechanisms to detect and eliminate byzantine behaviours, such as two-faced behaviours. These behaviours may arise due to spatial replication and could cause inconsistencies in data transmissions. For the reasons discussed above, TSN provides a certain degree of reliability, but it is not suitable for highly reliable networks.

11) Flexible Time-Triggered Replicated Star: The Flexible Time-Triggered Replicated Star (FTTRS) is a highly reliable network architecture built on top of the Flexible Time-Triggered (FTT) communication paradigm. Moreover, FTTRS uses Ethernet as the link and physical layer protocol. FTTRS' FT mechanisms are placed on the link layer [51].

TABLE III: Summary of protocols grouped by dependability attribute.

Attribute	Protocols
Availability	STP, SPB, MRP, AVB
Safety	PROFINET, SERCOS III, EtherCAT, EPL
Reliability	PRP, HSR HSE FF, AFDX, AeroRing, TSN
High Reliability	TTEthernet, FTTRS

FTT is based on a master/multi-slave architecture, where the master acts as a centralised controller that manages the communication among the slaves. FTTRS is based on an existing FTT implementation which uses a specifically-designed switch called Hard Real-Time Ethernet Switching (HaRTES), in which an FTT master is embedded. The used topology is a mono-hop replicated star with HaRTES switches, shown in Figure 18.

FTTRS is interoperable with standard Ethernet, supporting the connection of both, COTS nodes and switches, as it can transmit standard Ethernet frames as best-effort traffic without interfering with the RT traffic.

The communication in FTT, and therefore in FTTRS; is divided in communication cycles called Elementary Cycles (ECs). Each EC is in turn divided into three windows, that isolate the transmission of the different types of traffic. At the start of the EC, there is a window for the master to transmit a message that serves to both synchronize the slaves and to communicate them the transmission schedule (which slave has to transmit what) for the current EC. After that, the RT time-triggered and event-triggered traffic are transmitted in two different and consecutive windows. Finally, if there is enough time, best effort traffic is sent.

Moreover, FTT allows to change the traffic requirements (e.g. the periodic messages to be exchanged, their actual periods, etc.) online. Slaves can request changes to the master, which decides whether those specific changes can be made or not and carries out the configuration of the network.

FTTRS provides spatial redundancy to tolerate permanent faults in the communication channel. As mentioned, FTTRS is based on a replicated HaRTES star, where both HaRTESs are active. FTTRS provides mechanisms to ensure replica determinism between the two active replicas of the master. All these mechanisms are designed to tolerate at least the concurrent occurrence of one permanent and one temporary fault.

FTTRS provides time redundancy of frames to tolerate temporary faults in the links. This is specially important in the case of the Trigger Message. The TM is used to synchronize, trigger the communication and notify changes in the network to the slaves. In FTTRS the TM is replicated in the time domain to eliminate the SPoF a simplex transmission would represent.

Finally, FTTRS proposes error containment mechanisms that guarantee that the rest of the network is not affected even if slaves have byzantine failure semantics. These mechanisms are

TABLE IV: Classification of the protocols in terms of fault tolerance aspects and network features they fulfil.

Protocol	Fault Tolerance Aspects					Network Features		
	Path Redundancy	Time Redundancy	Manager Redundancy	Master Clock Redundancy	Network Consistency	Real-Time	Multihop	Any Flexibility
STP	Failover	None	Failover	-	No	No	Yes	Yes
SPB	Failover	None	-	-	No	No	Yes	Yes
MRP	Failover	None	Failover	-	No	No	Yes	Yes
PRP	Seamless	None	-	-	No	No	Yes	Yes
HSR	Seamless	None	-	Failover	No	No	Yes	Yes
HSE FF	Seamless	None	Failover	Failover	No	Hard	Yes	No
PROFINET	Seamless	None	Failover	None	No	Hard	Yes	No
SERCOSIII	Seamless	None	None	None	No	Hard	Yes	No
EtherCAT	Seamless	Failover	No	Failover	No	Hard	Yes	No
EPL	Seamless	None	Failover	Failover	No	Hard	Yes	No
AFDX	Seamless	None	-	-	No	Hard	Yes	No
AeroRing	Seamless	None	-	-	No	Hard	Yes	Yes
TTEthernet	Seamless	None	-	Seamless[a]	Yes	Hard	Yes	No
AVB	Failover	None	-	Failover	No	Soft	Yes	Yes
TSN	Seamless	None	None	Seamless	No	Hard	Yes	Yes
FTTRS	Seamless	Seamless	Seamless	Seamless	Yes	Hard	No	Yes

Seamless: redundancy allows to tolerate faults without interruption in the service.
Failover: the service is interrupted for certain time after a fault occurs.
None: there is no redundancy to tolerate faults.
- : does not apply to the protocol.
[a] TTEthernet's synchronization is actually based on a convergence function; i.e. the time is calculated as a function of different physical clocks.

implemented in the *port guardians*. These devices are placed between the nodes and the HaRTESs and frames that correspond to two-faced behaviours, impersonations and timing faults.

For all the reasons previously discussed FTTRS provides the network with a high level of reliability.

D. Summary

We have described protocols that use fault tolerance in different ways to offer different services. We started discussing the clock synchronization protocol PTP and some of its profiles. Specifically, we discussed the proposals to tolerate the failure of the master clock. The standards IEEE Std 1588-2002, IEEE Std 1588-2008 and the profile IEEE Std 802.1AS provide mechanisms that introduce failover times when the master clock fails. On the other hand, the IEEE 802.1AS-rev that is under development aims at providing seamless redundancy of the master clock.

We moved to communication protocols and we distinguished protocols that offer a single fault tolerance solution from those that also provide real-time services. We can further group the protocols described into four different categories depending on the dependability attribute they improve: availability, safety and reliability.

All STP versions, SPB, MRP and the AVB proposal improve the availability of the network. PROFINET, SERCOSIII, EtherCAT and EPL all increase the safety of the system. PRP and HSR provide reliability with no real-time; whereas HSE FF, AFDX, AeroRing and TSN provide reliability together with timing guarantees. Nevertheless, these protocols are not the best fit for highly reliable networks, as to the best of the authors' knowledge these protocols do not deal with byzantine behaviours. Finally, TTEthernet and FTTRS provide FT services to achieve high reliability for RT applications. Table III shows a summary of this classification.

On top of that, Table IV shows a classification of the protocols in terms of the fault tolerance aspects discussed. Furthermore, we included in this classification relevant network features to better understand the kind of applications that each protocol can support. Specifically, we distinguish the type of RT guarantees of the protocols; whether they can operate in multi-hop networks and whether they provide any flexibility.

We see that there are few protocols that provide real-time guarantees and flexibility. Specifically, STP, SPB, MRP, PRP and HSR offer certain degree of operational flexibility, as they allow to modify the operation of the network to deal with failures or to support the connection of new devices. Nevertheless, these changes are not done with any real-time guarantees; i.e. the time required for the changes is not bounded. Moreover, they do not provide real-time flexibility, as they do not support RT communications.

On the other hand, we find more sophisticated protocols such

as AeroRing, AVB and TSN that provide higher degrees of flexibility. First, all these protocols support real-time flexibility; even though AVB does not support hard real-time traffic. Moreover, AeroRing, AVB and TSN provide operational flexibility, as they support changes in the traffic, but do not bound the time required to perform the changes.

Regarding TTEthernet, we see that it provides the services required by highly-reliable systems and can operate in multi-hop networks. Moreover, it supports real-time flexibility, as it can convey different types of traffic. Nevertheless, it does not provide any operational flexibility. Finally, FTTRS supports both, operational flexibility, as it allows to modify the traffic requirements online and within a bounded time; and real-time flexibility, as it supports hard, soft and non-real-time traffic. Nevertheless, FTTRS is a mono-hop architecture and can not be deployed in larger systems.

Therefore, we see that there is need to develop a communication protocol capable of providing high reliability to real-time, multi-hop and adaptive systems.

V. Tolerance to Node Faults

As introduced previously, to achieve a high level of reliability in the system it is not enough to tolerate faults affecting the network but it is also necessary to be able to tolerate faults affecting the nodes. This is because nodes are typically the most complex components in a DES and, thus, they have the highest probability of being affected by faults [3], [4]. In fact, the fault tolerance mechanisms deployed at different levels of the system must be designed jointly to ensure that they function in a coupled manner, i.e., the fault tolerance design must be holistic. Moreover, this tight coupling needed from a fault tolerance perspective is also required from a real time perspective. Specifically, to consider a DES to operate in real time, both tasks and messages must be scheduled jointly.

In this section we survey what we call *complete infrastructures* for implementing highly-reliable DESs. By complete infrastructure we mean the set of interrelated hardware and software components (the architecture) and the set of in-built mechanisms that make it possible to fulfil the system requirements, i.e., the real-time and reliability requirements, both at the node and the network level.

The rest of the section is organized as follows. First, in Sec. V-A and Sec. V-B we put some of the fault-tolerance concepts introduced in Sec. III in the context of tolerance to node faults. Specifically, we conclude that task replication is the most flexible, efficient and fault tolerant approach to tolerate faults affecting the nodes and present the fault tolerance mechanisms that can be used to avoid the redundancy attrition that task replicas can suffer. Then, in sec. V-C we provide our classification of the complete infrastructures that make use of task replication. Finally, in Sec. V-D, we describe some examples of complete infrastructures to give a more detailed perspective of the design and operation of these solutions.

A. Design Approaches to Tolerate Node Faults

Faults affecting a node can be tolerated either by replicating said node or by replicating the tasks it executes. Please, recall from the introduction that a DES can be considered as composed by a set of (computational) nodes each executing one or more minimum computational units called tasks. In this sense replicating a node means to replicate the node's hardware, e.g. to use different identical nodes each performing the same computations; whereas replicating a task consists in carrying out its operations by means of several task replicas executed in the same or in different nodes.

Replicating the tasks is preferable over replicating the nodes for several reasons. First, replicating the tasks requires less specialized hardware. This is because replicating the nodes implies implementing the replication mechanisms in hardware. Second, nowadays the available hardware makes it possible to execute several tasks in the same node. Thus, each one of a given set of nodes can execute one or more replicas of different tasks, thereby allowing to execute a number of tasks replicas higher than the number of available nodes. Third, replicating the tasks provides the necessary level of granularity to tolerate, in a given node, faults that only affect one of the tasks being executed there. This is an interesting capacity, because typically tasks have different criticality levels and, thus, said granularity allows adjusting, independently for each task, the number of replicas to be executed in a given node according to that task criticality. Finally, replicating the tasks allows implementing more flexible fault-tolerance mechanisms. For instance, note that the assignment of tasks to nodes, i.e. what we call here *task allocation*, is normally decided at design time. However, some DES provide what here we refer to as *dynamic task allocation*, i.e., the ability to start/stop and then migrate tasks among nodes at runtime. This kind of flexibility can improve the functionality, efficiency and the fault tolerance of the system.

As also pointed out in Sec. III-B, there are two main types of replication, namely active and passive. For the specific case of task replication, actively replicating a task means that all the replicas of that task perform the same operations in parallel; whereas passively replicating a task consists in having one of the task replicas (the primary one) performing the operations, while a set of backup task replicas remain inactive until the primary fails (in which case one of the backup replicas takes over).

When choosing between active and passive replication, there is a trade-off between the time needed for taking over once a fault occurs, i.e. the fail-over time, and how complex it is to implement the replication strategy. As said in Sec. III-B, active replication is specially suitable to provide a zero fail-over time, i.e. to provide seamless fault tolerance (seamless redundancy). However, it is noteworthy that active replication is hard to implement. One important reason of this higher complexity in the case of task replication is that actively replicated tasks normally need to be tightly synchronized among them to adequately carry out their operations in parallel, e.g. to exchange and vote on their state to reach a consensus. In contrast, passive replication is normally easier to implement, but it introduces a significant delay between the failing of the primary replica and the instant of time in which one of the backup replicas takes over. Since real-time DESs with stringent RT requirements can benefit from seamlessly tolerating faults, the current section pays special attention to active replication.

B. Avoiding Redundancy Attrition in Replicated Tasks

Replicating tasks is costly as the amount of resources used to execute one operation needs to be doubled or even tripled. Moreover, additional mechanisms have to be deployed in the system to manage said replication. Therefore, it is fundamental to avoid *redundancy attrition* (Sec. III-C). This is of paramount importance as the investment put on the replication can be easily lost; specially when temporary faults may occur, since they are more likely to occur than permanent ones and can unnecessarily lead the tasks to be perceived as permanently faulty. *Redundancy preservation* [6] mechanisms can be used to avoid redundancy attrition. Although these mechanisms were introduced in Sec. III-C, let us revisit them in the context of task replication.

The first mechanism is called *recovery* and consists in replacing the erroneous state of a task with an error-free one. This error-free state can be obtained either from a stored copy of a state previous to the occurrence of the fault, or by inferring a new valid state from which the task replica can continue its operation. Note that this recovery can only succeed if a temporary fault provoked a corruption of the internal state of the task replica. It will not succeed if the faults permanently affected any hardware or software component.

In some cases, however, recovery does not suffice and, thus, the faulty task replica needs to *reintegrate*. Reintegration is similar to recovery in the sense that it implies replacing the erroneous state of the faulty replica by a correct one. However, in this case, the faulty task replica obtains the correct state as a result of reaching an agreement with the other non-faulty replicas. This is particularly relevant in replicated environments in which a faulty task replica can only return to operation if it adequately coordinates with the non-faulty ones.

Moreover, when the fault does not affect the applications being executed but other underlying software components or the hardware itself, task redundancy preservation can only be accomplished if the faulty task replica is *restored*, i.e. if the faulty task replica is removed and a new task replica is started either in the same node or in a different one at runtime. In any case, once restored, the new task replica needs to further recover or reintegrate to correctly operate. Task restoration is useful when the internal state of the faulty task replica has been corrupted in a way that it cannot recover nor reintegrate. Another scenario in which task restoration is also useful is when the node in which the task is being executed is affected by a permanent hardware fault which prevents said task replica from correctly operating. In this case the task can be reallocated to a different non-faulty node.

C. Classification of Infrastructures for Highly-Reliable DESs

In this section we present the classification we propose for the complete infrastructures that allow to implement highly-reliable DESs. Note, in this regard that, although the focus of this paper is put on the fault tolerance, the basic mechanisms used to achieve it have not changed in decades. Consequently, this cannot be a key aspect to differentiate among infrastructures. Instead, we propose to classify them depending on the kind of system they aim to support. Specifically, we identify two classes: *classical* and *contemporary*.

The classical infrastructures class comprises all those infrastructures that were developed between the 1970's and 2000's. These infrastructures were focused on fulfilling the requirements imposed by the DESs used during these decades. Specifically, these systems which tended to be small, not very complex and isolated. Lets explain these properties in a more detailed manner. First, they were composed of few interconnected nodes. Second, the operation they have to carry out was not very complex and, thus, the software and hardware required was also not very complex. Finally, it was not necessary to connect such systems with other systems and, thus, the kind of networks and communication technologies mostly used tended to be specialized. Some of the most relevant projects addressing the development of such infrastructures are: the Software Implemented Fault Tolerance (SIFT) computer [52], the Maintainable Real-Time System (MARS) [53], the Delta-4 [54] or the Generic Upgradable Architecture for Real-time Dependable Systems (GUARDS) [55].

The contemporary infrastructures class comprises all those infrastructures that were developed approximately from the 2000's until now. Note that at the end of the 1990's there was a huge growth in the size and complexity of the DESs, driven by a reduction in the price of the hardware components. This, in turn, provoked the development of new DESs with new requirements that had to be fulfilled. Next we list the most relevant needs that arose from that evolution of the DESs:

- *Need for more processing power*. As the hardware components became cheaper, the interest in improving the automation of processes and/or automatizing new processes growth. This called for more powerful hardware capable of implementing more functionalities while maintaining the fulfilment of the real-time and dependability requirements. In this regard, we identify two relevant hardware advances that allowed to achieve this goal:
  - *Heterogeneous hardware*. The evolution of the consumer-oriented microelectronics has make them very powerful and cheap. In this sense, there is a willingness to exploit the capabilities of the new devices in areas such as DESs. The main idea is to integrate multiple off-the-shelf components to construct DESs. Of course, since these components are not usually designed to operate to meet hard real-time and high reliability, new challenges have to be tackled.
  - *Multi-core systems*. DESs have been traditionally constructed as multiple distributed and interconnected mono-core processors. This is because each processor can constitute an error-containment region and, thus, tasks are physically isolated. However, it is difficult to make DESs scale using this scheme.

    To solve this issue contemporary infrastructures propose to use multi/many-core processors that integrate the execution of multiple tasks in one chip. Specifically, they propose to construct multi-core processors using the SoC paradigm, in which multiple cores are interconnected by means of a reliable and real-time on-chip network,

i.e. the so-called *Network-on-a-Chip* (NoC). Moreover, these multi-core processors are still interconnected through an industrial communication network.

The challenge is then to create these new error-containment regions in the scope of each processor in order to execute multiple tasks with different real-time and dependability requirements, considering, also, that these tasks would need to coordinate with other tasks in other processors connected to the network. It is noteworthy that most of the challenges met in regular industrial networks, like the scheduling of messages, will have to be addressed in the new on-chip network technologies that are appearing in the context of industrial SoCs.

- *Need for more network bandwidth*. As the number of nodes of the DES increases the quantity of data that has to be exchanged increases. Moreover, sensors become more complex and, thus, the amount of information each individual node can produce also increases. For instance, nowadays it is common to use images as data to be processed. This calls for network technologies with large bandwidth that can convey all this data. Specifically, as will be seen later, there is a trend of using Ethernet-based networks as the underlying communication technology for communication the nodes of DESs. In particular, TTEthernet (see Sec. IV-C8) is selected in most of infrastructures, as it is the de-facto standard for real-time communications. Moreover, TTTech, the company responsible for this technology, is involved in many of the European projects for developing dependable DESs.
- *Need for more efficient implementations*. The more a system grows, the bigger the amount of resources needed to support such scaling. Consequently, it is desirable that the system implements mechanisms to ensure that it is cheap and energetically efficient, despite its size. On the one hand, from the perspective of the processing power, mono-core processors do not scale appropriately. However, multi-core processors can be used to provide a higher performance in a more efficient manner. On the other hand, from the point of view of the fault tolerance, redundancy is typically, used to tolerate network and node faults. Redundancy implies additional hardware and software resources that are only used for this purpose. In order to overcome this issue, many DESs are constructed to support mixed-criticality functionalities. A mixed-criticality system makes all its resources available for both critical and non-critical functionalities, thus, avoiding the need for providing specific resources for each kind. Note, in this regard, that the system has to still ensure that the individual real-time and dependability requirements of the functionalities are fulfilled.
- *Need for more generic solutions*. The complexity and size of the DESs has been increasing continuously, but the ability of the developers to generate code for them cannot increase. One approach to address this issue is to design generic systems that can be adopted and tuned for different domains. This is done by identifying the pieces whose implementation can diverge in the different fields and specifying how they should be implemented. By doing this a set of core components are well-defined and, then, the rest of components, which are field-specific, are specified for each domain.
- *Need for integration with other systems*. As the DESs become more complex they need to communicate and cooperate with other systems in order to carry out their operation. This means that contemporary complex DESs are not constructed as an isolated and dedicated set of interconnected nodes, but as set of linked critical and non-critical subsystems that interact among them. Note that this is what the *system of systems* approach proposes, i.e., to interconnect several dedicated systems to create a more complex system that provides more functionality and performance than one that can be achieved with the sum of the individual systems [56].
- *Need for more adaptivity*. Any DES is prone to changing operational conditions. These changes can affect: (1) the functional requirements, i.e., what the system has to do; (2) the environment, i.e., the external conditions under which the system has to operate; and (3) the system itself, i.e., intentional or unintentional (faults) changes in the software or hardware. The classical way to address these changes is by providing the systems with enough resources to cope with the worst case scenario. An example of this is the additional software and hardware needed to tolerate the faults that could occur in the most harsh environment. However, this is very inefficient as most of the time most of the resources are under use. The ability to modify the behaviour of the system to face these changes is an interesting feature to take advantage as most as possible of the available resources in each operational scenario.

Some of the most relevant projects addressing the development of such infrastructures are: the Generic Embedded System (GENESYS) [57], the Industrial Exploitation of the GENESYS Cross-Domain Architecture (INDEXYS) [58], the Embedded Multi-Core Systems for Mixed-Criticality Applications in Dynamic and Changeable Real-Time Environments (EMC^2) [59] the Distributed Real-time Architeture for Mixed-Criticality Systems (DREAMS) [60] and the Dynamic Fault Tolerance for Flexible Time-Triggered (DFT4FTT) [61].

It is important to remark that, although the infrastructures belonging to the second class are more modern, they do not invalidate the ones from the first class. On the one hand, from a dependability perspective, the level of fault tolerance that can be achieved by the infrastructures can be the same, as all of them use the same basic fault tolerance principles. On the other hand, each infrastructure was designed having a set of goals in mind and, thus, each one is specialized in a given domain or context. Consequently, a given classical infrastructure can be more suitable than any contemporary one depending on the requirements of the system to implement.

Note also that this classification is not strict in the sense that some classical infrastructures can implement support to some of the above needs, to some extent. Note in this regard that GUARDS can represent the point of transition between the two classes. For instance, as will be explained in Sec. V-D2,

GUARDS provides some level of generality as it allows to create instances of itself to attain different levels of fault tolerance. Another example is it ability to support COTS components that can be upgraded over time.

Another aspect to highlight from the second class is that the complexity of the infrastructures increases exponentially with the addition of new functionalities. This is because, as previously explained, ensuring a real-time and dependable behaviour requires applying specific mechanisms at all the levels of the system. Consequently, the corresponding infrastructure must include specific mechanisms to ensure that these requirements are also fulfilled for the functionalities, which can be very hard. A good example of this is the use of multi-core processors to execute mixed-criticality functionalities. Note, in this regard, that interconnected tasks executed in different cores of a processor calls for an holistic scheduling that considers the allocation of tasks into cores and the communications carried out through the on-chip network. At the same time, task isolation must be ensured; i.e. ensure that a fault affecting one task executed in one core cannot propagate to other task in another core.

D. Examples of Infrastructures for Highly-Reliable DESs

In this section we describe more in detail some of the most representative infrastructures for implementing highly-reliable DESs for which there is available information. There are several aspects to highlight from this list. First, since we are now interested in faults affecting the nodes, the specific network utilized is not the focus in this part of the survey. We will consider any infrastructure that relies on a network that provides the necessary services to ensure that the system requirements can be met. Second, this list is ordered chronologically. We will first describe some relevant references that may not rely on Ethernet and, then, we will describe some more modern references that do use Ethernet as their underlying communication network and that are representative of the last years of this research area. Third, during the following description we use the terminology chosen by the specific designers of each solution when describing their infrastructures. Finally, in some cases, what we call infrastructure is referred as architecture by some authors. Consequently, note that, when describing the infrastructures, the term architecture can also include the logic that implements the real-time and fault tolerance services.

Regarding the organization of information, for each infrastructure we will describe: its background; its architecture and execution model, i.e., how the tasks are considered and how they are executed, depending on its relevance; its scheduling mechanisms; the network technologies supported and additional communication services provided; and, finally, its fault tolerance capabilities.

1) Delta-4 (1992) [54]: This was a 5-nation collaborative project with the purpose of designing and implementing an open, real-time and dependable distributed computing architecture. Specifically, an infrastructure to support large-scale LAN-based information processing systems like the ones that can be found in Computer-Integrated Manufacturing (CIM). By open we mean that the system can: be constructed using COTS components; support the heterogeneity of the hardware and software components; and provide re-usability to the software. By real-time we mean that it provides services to ensure that the deadlines imposed by the software can be met. Finally, regarding the dependability, Delta-4 is aimed at applications for which the reliability, availability and security is relevant, but not applications for which safety is necessary.

In Delta-4 the tasks are called *capsules*, which are logical run-time units of computation and data encapsulation that communicate with each other by means of messages. Capsules are created, coordinated and managed thanks to *Deltase* (Delta-4 Application Support Environment), a software infrastructure included in the Delta-4 architecture. Deltase provides a virtual environment that hides the inherent complexity of managing capsules in a distributed and fault-tolerant system.

Regarding the real-time services provided by Delta-4, note that openness has to be sacrificed when a strict real-time response is required and, thus, two variants of the architectures are proposed. On the one hand, the Delta-4 Open System Architecture (D4-OSA) allows to implement non-real-time systems in which hardware and software heterogeneity is a requirement. On the other hand, the Delta-4 Extra Performance Architecture (D4-XPA) is tailored for systems with explicit real-time requirements. However, the main assumption is that nodes are homogeneous and fail-silent.

As concerns the communication network, Delta-4 has to be deployed on top of a communication network that is also real-time. In this regard, there are implementations for token bus (ISO 802.4), token ring (ISO 802.5) and FDDI (ISO 9314). These communication networks are based on token passing medium access technique, which makes the communications deterministic and, thus, suitable for real-time systems. Moreover, in those cases in which dependability is a requirement, a dual communication media scheme can be used. This makes it possible to tolerate the failure of one of the media and isolate any failing node.

Media redundancy it is enough to ensure that messages are delivered properly in a distributed environment. Note, for instance, that a message issued by a capsule has to be delivered to corresponding recipient capsules consistently (either all replicas receive the message or none of them receives it) and in the same order. To ensure these properties, Delta-4 includes an atomic multicast protocol on top of the communication network called AMp in the D4-OSA and xAMp in the D4-XPA.

Regarding its fault tolerance features, Delta-4 is able to tolerate faults affecting the hardware of the nodes and, to a lesser extent, accidental design faults affecting the software. On the one hand, hardware faults are tolerated by means of capsule replication, i.e., critical operations are carried out by a group of capsule replicas (see Fig. 19). The supported replication techniques are: active, passive and semi-active replication. On the other hand, software faults are tolerated thanks to design diversity [62].

Note, however, that the set of fault-tolerance features provided depend on the architecture variant. D4-OSA is focused on openness which means, among other things, that the system is constructed using off-the-shelf computers that do

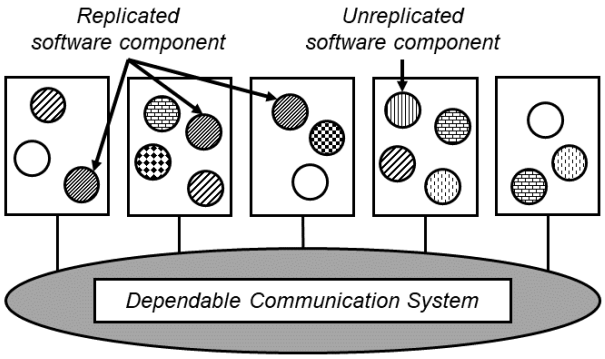


Fig. 19: Replicated Software Components in Delta-4 (reproduced as in [54]).

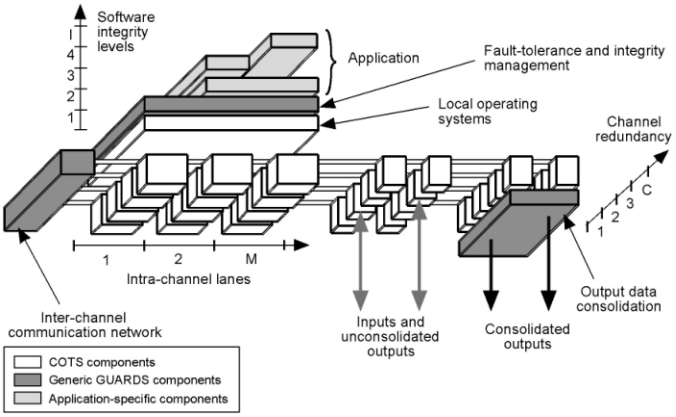


Fig. 20: GUARDS architecture (adapted from [63]).

not include dedicated fault-tolerance features and that can fail in an uncontrolled manner. Consequently, D4-OSA provides the necessary error-processing and fault-treatment mechanisms to ensure that hardware faults can be tolerated in these type of nodes. In contrast, as explained previously, D4-XPA needs to ensure a real-time behaviour and, thus, the fault-tolerance mechanisms provided are adapted accordingly. On the one hand, D4-XPA requires that nodes are fail-silent, that is why it makes it possible to restrict the failure semantics of a node from fail-uncontrolled to fail-silent by embedding in it a so-called Network Attachment Controller (NAC). On the other hand, the only replication technique supported is semi-active replication. This is because, contrary to active replication, the atomic multicast protocol used has a bounded delay.

Finally, Delta-4 includes *fault-treatment* mechanisms. First, Delta-4 is able to perform *fault diagnosis*, i.e. localize existing faults affecting a replica group and determine if faults can be passivated. Second, *fault passivation* can be used to prevent a faulty node or capsule to provoke additional errors. Finally, *system reconfiguration* and *system maintenance* allow to re-allocate and re-initialize faulty replica capsules to carry out redundancy preservation. In case the available computational resources are not enough to perform said operations, non-critical capsules can be deallocated in favour of the critical ones. Moreover, if this is not enough, the reconfiguration can be deferred until a node recovers.

2) GUARDS (1999) [55]: The Generic Upgradable Architecture for Real-time Dependable Systems was an European project whose main goal was to tackle the cost and time issues derived from development and validation of highly-reliable DESs. Specifically, GUARDS developers state that building new products from previous fault tolerance developments is very complex, which increases the time-to-market. Consequently, GUARDS proposes a complete generic infrastructure for implementing highly-reliable DESs. By generic we mean that GUARDS is able to implement DESs from different domains and, thus, to cover a wide range of real-time and dependability requirements.

Moreover the rapid advancements in microelectronics make the obsolescence of the hardware components an issue. For this reason the GUARDS architecture was designed to be constructed using COTS which can be upgraded during the lifetime of the system, i.e., up-to-date version of the hardware and software components can be incorporated. This makes it possible to add new functionalities requiring more processing power or additional new components.

The GUARDS architecture, as can be seen in Fig. 20, is defined along three dimensions of error containment: *channels*, *lanes*, and *integrity levels*. An instance of the GUARDS architecture is defined by the three dimensional parameters $\{C, M, I\}$, which correspond to the number of channels, lanes and integrity levels respectively.

- The channel dimension provides the first error containment region for physical faults affecting the nodes, by means of task replication. For this, each channel executes one task replica following the active replication scheme. It is noteworthy that not all the GUARDS instances require the same number of channels. This results in various possible fault tolerance mechanisms. For instance, $C = 2$ implements duplication, which is used as a mean to detect errors. This is done by comparing the outputs of both replicas, and, if they diverge, one of the two suffered from a fault. This is useful to implement a safe failure mode like *fail-stop*, where upon the detection of a fault replicas stop their operation. $C = 3$ implements Triple Modular Redundancy (TMR) where voting is used to mask the faults without a service interruption. Finally, as will be further explained later, channels are interconnected by means of an Inter-Channel Network (ICN).
- The lane dimension provides the secondary error containment region for physical faults affecting the nodes and corresponds to the presence of multiple processors in a single channel. This additional processors can be used for several purposes. First, for improving the fault diagnosis capabilities within a channel, by comparing the result obtained by several computational replicas. Second, for improving the availability of a channel, e.g., by passivating a node that is diagnosed as permanently faulty. Finally, another reason for including multiple lanes in an instance

is to have parallel processing to improve performance and isolation of software of different integrity levels.
- The integrity level provides containment regions with respect to the software design faults. Specifically, it is tailored to protect critical components from the propagation of error provoked by design faults in lower-critical components. For this, each *application* is assigned to a specific integrity level. When the integrity level is higher, the consequences of the failure of the application are more severe. The protection is achieved by means of an integrity policy that controls the communication between applications of different integrity levels. Specifically, this policy uses a so-called *Validation Objects* that make use of fault tolerance mechanisms to allow information flows from low to high integrity level applications.

GUARDS is capable of supporting a range of real-time computational and scheduling models. However, in any case, all the real-time attributes are known at design time and, thus, the scheduling is done off-line. First, a schedulability analysis is carried out for each channel. This ensures that tasks in each channel can execute without missing any deadline. After that, taking into account the attributes of the tasks, the scheduling for the communications through the ICN is built, similarly as cyclic executive schedules are constructed.

The central point of the GUARDS architecture is a custom network called Inter-Channel Network, which was already mention above. This network interconnects all the channels while providing them global clock synchronization and interactive consistency [64] on non-replicated data. For this, it includes multiple ICN-managers, one for each channel. These managers are interconnected by means of a Ethernet-based broadcast communication infrastructure. Specifically, every ICN-manager contains C Ethernet links (being C the number of channels), one link to transmit to the other ICN-managers and $C-1$ links to receive from the other ICN-managers. This kind of topology makes it possible to reduce the complexity of implementing consistent broadcast communication among channels.

The basic fault tolerance mechanisms of GUARDS have already been introduced previously as they are in-built into its architecture. The channel and lane dimensions allow to implement several different fault tolerance strategies based on active replication. Moreover, the integrity level dimension allows applications of different levels of criticality to safely coexist and interact, i.e., it supports mixed-criticality systems.

Additionally, GUARDS includes mechanisms that allow it to diagnose a channel as lost, due to a temporary fault, and then reintegrate it. Specifically, a reintegration algorithm including a clock re-synchronization and a state restoration is defined. The complexity is in the state restoration process, which consists in retrieving the *channel context*, i.e., the value of all the internal variables of the channel, from the other channels.

3) EMC^2 (2017) [59]: The Embedded multi-core systems for Mixed-Criticality applications in dynamic and Changeable real-time environments was an European project that aimed at creating an open and interoperable service-oriented architecture approach for mixed-criticality applications in dynamic and changeable real-time environments using multi-core processors. Next we describe in more detail what we mean.

- Open means that EMC^2 can be used in different industrial domains like automotive, avionics, space, or health care. Each of these domains has specific requirements. For instance, all of them have different dependability requirements. Consequently, the EMC^2 architecture has been designed to be generic and, then, it can be adapted for the specific industrial domain.
- Interoperable means that EMC^2 allows to implement systems that can dynamically interact with other systems. Moreover, these systems do not need to be industrial.
- The Service-Oriented Architecture (SOA) is a software design pattern in which multiples services, i.e., units of functionality, are implemented in a distributed environment and, thanks to a well-defined interfaces, can interact among them to achieve some objective. Moreover, the SOA approach makes it possible to create and remove services and to modify their communications during the operation of the system. For all these reasons, SOA eases the development and maintenance of the software.
- A mixed-criticality system is one that is able to integrate multiple applications with different levels of safety and security on a single computing platform.
- Dynamic and changeable real-time environments means that the runtime context can change at any time during the operation of the system. These changes can be external to the system, i.e., changes in the environment, or internal to the system, i.e., changes in the availability and quality of available platform resources. In order for the system to operate properly under these conditions it must be adaptive, i.e., it must be able to change its behaviour dynamically and autonomously.
  Furthermore, EMC^2 also makes it possible to dynamically modify the software being executed in the system in a service-on-demand fashion, similarly as it happens with the applications on a mobile phone. This means that the system permits to start new services and stop any being executed.
- A multi-core processors is a processing unit composed of multiple cores. This type of processor is nowadays powerful yet cheap and make it possible to improve the system integration, efficiency and performance. Consequently, it comes to replace the mono-core processors for implementing complex computational systems.
  Moreover, System-on-a-Chip (SoC) is the technology suitable for implementing this multi-core processors in industrial systems. A SoC is a chip that contains a set of interconnected heterogeneous components, i.e., technology suppliers can select which hardware functionalities they need and construct a chip that includes them. A typical SoC for industrial systems is composed of multiple application-purpose cores and additional control components interconnected through a reliable and real-time on-chip network, i.e., a Network-on-a-Chip (NoC).

In EMC^2 the scheduling is carried out at two levels. On the one hand, a global scheduling is performed to guarantee the real-time behaviour of the end-to-end communication. On the other hand, a local arbitration is performed at each processor. By

doing this, EMC^2 ensures a deterministic and exclusive access to the NoC. For this to work applications have to negotiate their access to the communications with the scheduling modules, the so-called Resource Managers (RMs).

Concerning the type of scheduling carried out, EMC^2 studied the scheduling for virtualized systems, i.e., the real-time scheduling mechanisms that make it possible to develop a real-time scheduler and a resource sharing protocol for a safety critical operating system targeting mixed-critical automotive applications. Specifically, the technique selected was hierarchical scheduling, which encapsulates different tasks in different partitions, each with a given processor execution budget. Hierarchical scheduling is a good candidate for mixed-criticality systems as it ensures time isolation between tasks mapped in different partitions.

For the off-chip communication EMC^2 does not impose any specific network technology. In contrast, due to the openness of the solution, the most suitable network is selected in each industrial domain. For instance, in the automotive domain CAN, CAN-FD and automotive Ethernet are selected. Nevertheless, EMC^2 assumes Ethernet whenever possible as it provides new mechanisms for achieving dependable and high accuracy time synchronization over Ethernet.

For the on-chip communication EMC^2 proposes the use of the so-called Time-Triggered Network-on-a-Chip (TTNoC). This network technology allows a deterministic communication among multiple chip components.

Regarding the fault tolerance services provided by EMC^2 , it is important to remember that, as explained previously, its openness allows it to be applied in different industrial domains and, thus, support different safety requirements. For this to work, EMC^2 is kept generic, i.e., it does not explicitly define the safety mechanisms. Instead, these mechanisms are defined in each specific industrial domain. We will now discuss the fault tolerance mechanisms designed in the context of the EMC^2 project for transport automation systems in general and railway systems in particular. Note that we select this example because, contrary to other domains, railway systems have reliability requirements.

Railway systems have stringent reliability and availability requirements. To meet these requirements a middleware layer is proposed. This middleware provides services for synchronization, communication, fault detection by means of voting, and health-monitoring. From the point of view of the replication of tasks different configurations are supported. First, with no replication health monitoring and software diversity is used to achieve safety. Second, two replicas can be used as mean to detect errors (see V-D3). Finally, three replicas can be used to implement a TMR scheme (see V-D3).

4) DREAMS (2017) [60]: The Distributed REal-time Architecture for Mixed criticality Systems was an European project with the purpose of developing an architecture for mixed-critical distributed systems executed on networked multi-core chips. By mixed-critical system we mean a system that is able to simultaneously execute multiple applications with different levels of criticality, while guaranteeing their operational requirements. In particular, DREAMS addresses the execution of applications with different security, safety and

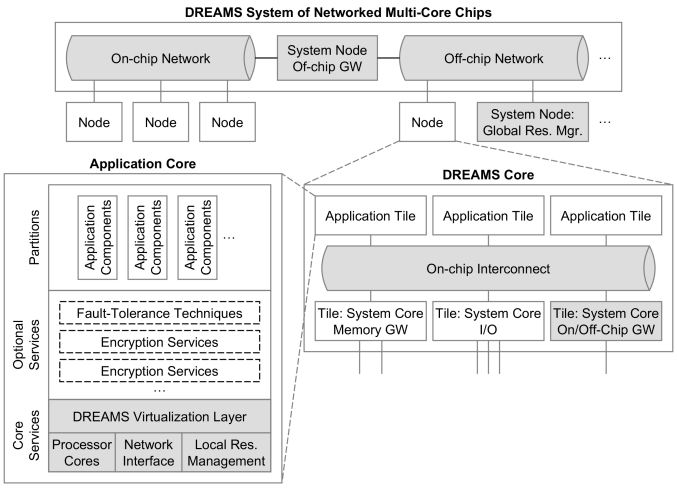


Fig. 21: DREAMS architecture (reprinted from [60]).

real-time requirements. Moreover, to avoid the propagation of faults in different applications and unintended side effects due to their integration, DREAMS introduces mechanisms that ensure a temporal and spatial partitioning of the application subsystems at different levels. First, at the software execution level, DREAMS provides an *hypervisor*, i.e., a software layer that is able to create independent software execution environments. Second, at the distributed system level, DREAMS incorporates network building blocks that makes it possible to partition the communications among the nodes using heterogeneous time- and/or event-triggered networks. Finally, at the chip level, DREAMS isolates the execution of the software in specific cores of the multi-core processors.

The architecture proposed in the context of the DREAMS project, as shown in Fig. 21, covers several levels. The smallest unit of computation is called *Application* and several of them can be confined into an *Application Tile*, which ensures a proper partitioning. Several Application Tiles can be inserted in a node. Moreover, they can exchange messages thanks to an on-chip network to coordinate their operation. Note in the figure that additional tiles are required to carry out system-related operations. Finally, several nodes can exchange messages thanks to one or various off-chip networks in order to coordinate the operation of applications being executed in different nodes. Note, in this regard, that this architecture provides end-to-end communication channels over several heterogeneous and mixed-criticality on-chip and off-chip networks. This horizontal and vertical integration is possible thanks to *gateways*.

DREAMS schedules the computational resources to ensure that each task obtains the required computational time of a core and that its execution meets the deadlines. On the one hand, a static scheduling is carried out off-line to obtain the pre-computed scheduling decisions for every instant. On the other hand, a dynamic scheduling can be used during runtime to reconfigure the system when faults do occur and, thus, do a recovery.

Concerning the exchange of messages among Applications,

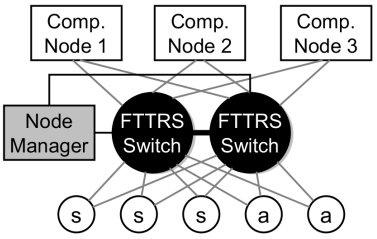


Fig. 22: DFT4FTT architecture.

note that, as already explained, communications are carried out both at the off-chip and on-chip level. DREAMS does not impose any specific network technology in any of these two levels, as long as they provide the required real-time and fault-tolerance services. On the one hand, off-chip networks are the ones that interconnect the nodes of the system. These are regular industrial LANs like TTEthernet and EtherCAT. On the other hand, on-chip networks are the ones that interconnect the Application Tiles inside a chip. These are internal chip networks that allow a predictable communication in System-on-a-Chip (SoC) architectures. Some examples are AEthereal [65], Spidergon Network-on-Chip (STNoC) [66] or Time-Triggered Network-on-Chip (TTNoC) [67].

As DREAMS specifies a general architecture, it does not impose any specific redundancy technique. Instead, it supports active replication and voting for those systems in which dependability is a main concern. With this type of redundancy DREAMS is able to perform error detection and error masking. Moreover, the DREAMS architecture ensures the replica determinism of the replicated safety-critical-related components. Finally, as will be explained at the end of the DREAM project description, DREAMS also includes adaptivity mechanisms that allow it to recover from faults.

DREAMS also includes mechanisms to support adaptivity. Specifically, it allows to modify dynamically the amount of resources assigned to the Applications. For this purpose, as seen in Fig. 21, a dedicated system node called Global Resource Manager (GRM) dynamically computes new configurations of resources. A configuration includes for each partition, for instance, the scheduling, the assigned processor core or the amount of memory available. Note that changes in the configurations are triggered by changes in the diagnosis information gathered by the local resource monitors in the nodes. This information includes both the availability of resources and the timing behaviour.

5) DFT4FTT (2018) [61]: The Dynamic Fault Tolerance for Flexible Time-Triggered is an on-going Spanish project which aims at developing a complete infrastructure for implementing the so-called Adaptive Distributed Embedded Systems (ADESs), i.e., a type of DES that has the ability to modify its behaviour autonomously and dynamically in response to changing operational requirements or conditions. Specifically, here the focus is on the mechanisms that provide flexibility both from a functional and a fault tolerant perspective.

In DFT4FTT each of the functionalities the system has to execute is implemented by means of an *application* which, in turn, is composed by a set of distributed *tasks* that are executed in a sequential and/or parallel manner. A task in the minimum unit of computation and has various operational attributes like periodicity with which it executes, the time it needs to execute or the list of other tasks with which its interacts.

Tasks can be executed in the nodes of the distributed system. Specifically, a node can hold various tasks, as long as it has enough resources and the executions of said tasks are schedulable. The allocation of tasks in the nodes is decided automatically and dynamically by a central element called *Node Manager* (NM). For this, the NM carries out three processes. First, it obtains the *system state* by monitoring the system. The system state includes information like: the list of nodes, as well as the links, and if they are faulty; the list of running tasks and their health; or the failure rate of the nodes and the bit error rate of the links of the nodes. Second, in the decision process the NM checks that the system state fulfills the *system requirements*. The system requirements is the list of applications the system must execute. In case the system requirements are not fulfilled the NM searches a new configuration, i.e., a new task allocation, that fulfils them. Finally, in the configuration process the NM instructs the nodes to perform the required communication and computational changes to implement the new configuration.

The operational flexibility is achieved thanks to the modification of the system requirements list in the NM. Specifically, the nodes or the NM itself can modify this list to indicate changes in the operational requirements.

As concerns the scheduling, DFT4FTT proposes a holistic scheduling that determines for each application, a sequence of task executions and message transmissions that allows them to meet their deadlines. Additionally, one schedulability analysis must be carried out in each node to ensure that the executions of the tasks in said nodes meet all their deadlines.

As can be seem in Fig. 22, DFT4FTT relies on an switched-Ethernet-based network called FTTRS which, as already explained in Sec. IV-C11, takes as a basis the HaRTES, whose most relevant feature is its ability to change dynamically the communication of the network nodes, while ensuring a real-time response in the transmission of messages. Then, FTTRS uses spatial and time replication to achieve fault tolerance. This flexibility, together with the real-time and the fault-tolerance services makes FTTRS a crucial piece in the DFT4FTT infrastructure. Finally, in the context of the DFT4FTT project, FTTRS will be extended to provide it with flexibility in its fault tolerance mechanisms.

The fault tolerance mechanisms of DFT4FTT at the node level are based in active replication with majority voting. Specifically, the critical tasks are replicated following the TMR approach (see Sec. V-D2). However, the level of replication is managed automatically and dynamically by the NM based on the information contained in the system requirements and the system state. Specifically, one of the operational attributes contained in the system requirements is the level of reliability of each task, i.e., the probability with which each task must be operational during the mission time. Moreover, the system state contains updated information about the system and the environment. Consequently, upon the modification of the

reliability level of a task, due to a change in the operation requirements, the NM is able to automatically increase or decrease the level of replication. Another example, upon the occurrence of a fault affecting a node, the NM is able reallocate the tasks being executed in said node to other available ones. Likewise, if the environment changes in way that it affects the failure rate of the nodes, the level of replication of tasks can also be modified to meet the reliability requirements. In this regard, note that DFT4FTT supports: no replication, when the task is not critical; TMR, when the task is critical; and 5-Modular Redundancy, when the task is critical and the system operates in a extremely harsh environment.

Flexibility for fault tolerance was kept in mind all the time while designing the DFT4FTT infrastructure. This is why it is very versatile in reconfiguring the system to achieve as much reliability as possible. Actually, the next steps are towards developing the so-called *adaptive fault tolerance*, i.e., the ability to change the fault tolerance strategy, from N-Modular Redundancy in this case, to improve even more the level of reliability that can be achieved.

E. Summary

In this section we have discussed how to tolerate faults affecting the nodes of a DES. Specifically, we have stated that the most suitable manner of doing so is by using what we call a *complete infrastructure*. A complete infrastructure is the set of interrelated hardware and software components (the architecture) and the set of in-built mechanisms that make it possible to fulfil the system requirements, namely, the real-time and reliability requirements, both at the node and the network level. The reason for doing so is because the development of the mechanisms that make it possible to fulfil these requirements must be done in a holistic manner. This means that they must be developed jointly and at all the levels of the architecture.

We have explained the mechanisms that are most suitable for tolerating node faults, which are also the ones that these infrastructures implement. Specifically, we have explained why to actively replicating the critical tasks (software computational units) and execute them in different nodes is better—for hard RT and FT DESs— than replicating the nodes (hardware) themselves. In particular, replicating the tasks is preferable because it allows the system to include mechanisms to dynamically manage at runtime the execution of tasks, thereby paving the way for implementing flexible fault-tolerance mechanisms. For instance, when a node suffers from a hardware fault, redundancy attrition can be avoided by migrating (restoring) the task replicas affected by that fault to other non-faulty nodes.

Then, we have presented a classification for the infrastructures. This classification is based on the kind of DESs they have to support. More precisely, we distinguish two classes: *classic* and *contemporary* infrastructures. The first class encompasses all those infrastructures developed between the 1970's and the 2000's which aim at supporting small, not very complex and isolated DESs. The second class encompasses all those infrastructures developed from the 2000's until now and are characterized for including new features required by the new DESs developed in the last decades. In this regard, we have summarized the most important needs that have been arisen due to the significant evolution of the hardware components at the end of the 1990's.

Finally, we have further described some of the infrastructures we consider more relevant. In particular, for each of them we have described: its background; its architecture and execution model, when relevant; its scheduling mechanisms; the network technologies supported and additional communication services provided; and, finally, its fault tolerance capabilities. A summarized version of this information can be found in Table V.

VI. Conclusions

In the last decades there has been a growing trend towards the design of systems for automating all kinds of processes in all kind of applications. These systems typically have to operate in real time and many of them have requirements regarding their dependability level. The most sophisticated among them are those providing a high reliability using fault tolerance techniques.

When these systems are distributed, one of their fundamental components is the network that is used to exchange information, so the different nodes can coordinate their actions. For highly-reliable real-time distributed systems, all levels of the architecture, including the network, need to provide adequate services and be able to tolerate their faults.

There are several factors that have limited the development of highly-reliable systems. First, due to their complexity is hard to create a general purpose solution. Nowadays, these systems are designed as a whole, knowing in advance the requirements of the application to be executed and the environment in which they are deployed. Second, the complexity of the solutions are reflected in the cost. This means that, many times, these systems are only implemented when it is strictly necessary; i.e. forced by laws or certifications.

Nevertheless, due to the relevance of these systems, some communication protocols tailored for automation applications, have evolved or have been proposed to provide hard RT and/or highly-reliable communications; whereas some complete infrastructures have been proposed to provide these attributes at the level of the nodes as well.

There is a growing trend towards using Ethernet as the network technology for this kind of systems. Ethernet's main advantages are low cost, high bandwidth, compatibility with IP-based networks and high scalability. Unfortunately, the original specification of this technology lacks appropriate services to fulfil the demanding requirements of highly-reliable real-time applications. In this regard, many Ethernet-based protocols and standards have been proposed along the last years to deal with these limitations.

In this paper we survey the communication protocols and the complete infrastructures that are more relevant to provide hard RT and high reliability for DESs based on Ethernet.

We exclusively survey protocols based or proposed for Ethernet; whereas we survey infrastructures that are not necessarily based on Ethernet, but that can serve as an inspiration for future infrastructures based on this technology.

First, we discussed some of the fault tolerance mechanisms proposed for one of the most relevant clock synchronization

TABLE V: Summary of the complete infrastructures for implementing highly-reliable real-time DES.

Infrastructure	Real-Time	Communication		Mixed-Criticality	Multi-Core	Fault-Tolerance
		Network	Add. Services			
Delta-4	D4-OSA: No D4-XPA: Open	Token bus, Token ring or FDDI	Media redundancy and Atomic broadcast	No	No	Task replication Design diversity
GUARDS	Off-line scheduler	Ethernet	Clock synchronization Interactive consistency	Yes	No	Task/Node replication Integrity levels
EMC^2	On-line scheduler	Off-Chip: Open On-Chip: TTNoC	Clock synchronization when Ethernet	Yes	Yes	Open
DREAMS	Off/On-line scheduler	Off-chip: Open On-chip: Open	-	Yes	Yes	Open Task replication
DFT4FTT	On-line scheduler	FTTRS	-	Yes	No	Task replication Autonomous reconf.

protocols, the Precision Time Protocol (PTP). Second, we showed that there exist a series of protocols that provide just one FT mechanism, based on different types of redundancy, e.g. spatial redundancy. These protocols are not designed to support hard RT and highly-reliable DESs, but still they are interesting as they can be potentially combined to provide a suit of FT mechanisms for future hard RT and highly-reliable Ethernet-based networks.

Then, we described Ethernet protocols that provide one or more FT mechanisms together with mechanisms for supporting hard RT communications. Nevertheless, not all these protocols provide high reliability. Instead, some protocols provide other dependability-related attributes such as availability or safety. We finish describing the ones with FT mechanisms that are specially designed to provide high reliability for hard RT applications.

Moreover, some of protocols we survey provide flexibility mechanisms to support different types of RT traffic, or even to change the RT requirements of the traffic at runtime. This is specially interesting for the development of hard real-time and highly reliable infrastructures.

The most adequate manner of implementing a highly-reliable DES in general, and fault tolerance for the nodes in particular, is by means of a complete infrastructure. This is due to the fact that the real-time and dependability requirements these DES must satisfy can only be fulfilled by developing the appropriate mechanisms in a holistic manner, i.e., they must be developed jointly and at all the levels of the architecture.

In these infrastructures fault tolerance at the node level is achieved by replicating the software, i.e., the tasks, instead of replicating the hardware, i.e., the nodes. This is because task replication is most cost-effective and it makes it easy to provide flexible fault tolerance mechanisms. Moreover, in order to avoid the redundancy attrition tasks can suffer, most of these infrastructures also include specific mechanisms to recover, reintegrate or restore faulty tasks.

These infrastructures can be classified considering the DESs they aim to support. In this regard, we distinguish classical and contemporary infrastructures. While classical infrastructures are tailored for small, not very complex and isolated DESs, contemporary infrastructures give support to more complex DESs. Note that this complexity involves new challenges that have to be overcome. This, in turn results in new requirements related to the performance, efficiency, generality, connectivity and adaptivity of the DESs that have to be fulfilled by the infrastructures.

Regarding the future work in the development of highly-reliable Ethernet-based DESs, the authors have distinguished several open challenges that have to be addressed in the next years. At the network level, there is still space for further development, as discussed in the summary of Sec. IV. Specifically, by looking at Table IV, we see that many of the surveyed protocols are not suitable for highly-reliable or adaptive systems. Therefore, we see that there is need to develop a communication protocol capable of providing high reliability to real-time, multi-hop and adaptive systems. At the system level, adaptivity represents an interesting challenge that open the room for more efficient and effective fault tolerance. This is possible since an adaptive systems should be able to autonomously and automatically select its level or strategy of fault tolerance at each instant, depending on the operation context.

Finally, it is important to highlight that there is always work to do in the fault tolerance domain. This is because there is no silver bullet allowing to make a system completely reliable. Every system is unique and it must be studied individually to detect the kind of faults it can suffer and apply the appropriate fault tolerance mechanisms properly at all the levels of the architecture. Consequently, although the basic fault tolerance principles have not change in decades, applying them is an ad-hoc procedure which is complex.

VII. Acknowledgements

This work is supported in part by the Spanish Agencia Estatal de Investigación (AEI) and in part by FEDER funding through grant TEC2015-70313-R (AEI/FEDER, UE). Sinisa Djerasevic was supported by a scholarship of the EUROWEB Project, which is funded by the Erasmus Mundus Action II programme of the European Commission. The authors also wish to thank the reviewers for their useful and relevant comments on this article.

References

[1] H. Kopetz, *Real-Time Systems*, ser. Real-Time Systems Series. Boston, MA: Springer US, 2011.

[2] J.-C. Laprie, *Dependability: Basic Concepts and Terminology*. Springer-Verlag, 1992.

[3] M. Barranco, J. Proenza, and L. Almeida, "Reliability improvement achievable in CAN-based systems by means of the ReCANcentrate replicated star topology," in *2010 IEEE International Workshop on Factory Communication Systems Proceedings*, May 2010, pp. 99–108.

[4] M. Barranco, J. Proenza, and L. Almeida, "Quantitative Comparison of the Error-Containment Capabilities of a Bus and a Star Topology in CAN Networks," *IEEE Transactions on Industrial Electronics*, vol. 58, no. 3, pp. 802–813, mar 2011.

[5] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*, 4th ed. USA: Addison-Wesley Educational Publishers Inc, 2009.

[6] S. Poledna, *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*. Norwell, MA, USA: Kluwer Academic Publishers, 1996.

[7] B. M. Wilamowski and J. D. Irwin, *Industrial Communication Systems (The Industrial Electronics Handbook)*, second edi ed., CRC Press, Ed., 2011.

[8] T. Anderson and P. Lee, Eds., *Fault Tolerance – Principles and Practice*. Prentice Hall, 1981.

[9] B. W. Johnson, *Design and Analysis of Fault Tolerant Digital Systems*. Addison-Wesley Publishing Company, 1988.

[10] J. Proenza, "RCMBnet: A distributed Hardware and Firmware Support for Software Fault Tolerance," Ph.D. dissertation, 2007.

[11] "Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model," 1994.

[12] "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems," *IEEE Std 1588-2002*, pp. i–144, 2002.

[13] R. Zurawski, Ed., *Industrial Communication Technology Handbook*, 2nd ed. CRC Press, 2015.

[14] "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems," *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, pp. 1–300, July 2008.

[15] "IEEE Standard for Local and Metropolitan Area Networks - Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks," *IEEE Std 802.1AS-2011*, pp. 1–292, March 2011.

[16] "IEEE Draft Standard for Local and Metropolitan Area Networks - Timing and Synchronization for Time-Sensitive Applications," *IEEE P802.1AS-Rev/D6.0 December 2017*, Jan 2018.

[17] A. Mahmood, F. Ring, A. Nagy, T. Bigler, A. Treytl, T. Sauter, and N. Kerö, "High precision and robustness in network-based clock synchronization using ieee 1588," in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, Sep. 2014, pp. 1–4.

[18] "IEEE Standard for Information Technology- Telecommunications and Information Exchange Between Systems- Local and Metropolitan Area Networks- Common Specifications Part 3: Media Access Control (MAC) Bridges," *ANSI/IEEE Std 802.1D, 1998 Edition*, pp. i–355, 1998.

[19] "IEEE Standard for Local and Metropolitan Area Networks: Media Access Control (MAC) Bridges," *IEEE Std 802.1D-2004 (Revision of IEEE Std 802.1D-1998)*, pp. 1–281, June 2004.

[20] "IEEE Standard for Local and Metropolitan Area Networks—Virtual Bridged Local Area Networks," *IEEE Std 802.1Q-2005 (Incorporates IEEE Std 802.1Q1998, IEEE Std 802.1u-2001, IEEE Std 802.1v-2001, and IEEE Std 802.1s-2002)*, pp. 1–300, May 2006.

[21] (2019, February) Understanding Rapid PVST+. https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus5000/sw/configuration/guide/cli/CLIConfigurationGuide/RPVSpanningTree.html#31778.

[22] "IEEE Standard for Local and Metropolitan Area Networks–Media Access Control (MAC) Bridges and Virtual Bridges," *IEEE Std 802.1Q, 2012 Edition, (Incorporating IEEE Std 802.1Q-2011, IEEE Std 802.1Qbe-2011, IEEE Std 802.1Qbc-2011,IEEE Std 802.1Qbb-2011, IEEE Std 802.1Qaz-2011, IEEE Std 802.1Qbf-2011,IEEE Std 802.1Qbg-2012, IEEE Std 802.1aq-2012, IEEE Std 802.1Q-2012*, pp. 1–1782, Dec 2012.

[23] "IEEE Standard for Local and Metropolitan Area Networks - Virtual Bridged Local Area Networks Amendment 5: Connectivity Fault Management," *IEEE Std 802.1ag - 2007 (Amendment to IEEE Std 802.1Q - 2005 as amended by IEEE Std 802.1ad - 2005 and IEEE Std 802.1ak - 2007)*, pp. 1–260, 2007.

[24] "Industrial Communication Networks – High Availability Automation Networks – Part 2: Media Redundancy Protocol (MRP)," 2010.

[25] "Industrial Communication Networks – High Availability Automation Networks – Part 2: Media Redundancy Protocol (MRP)," 2016.

[26] "Industrial Communication Networks – High Availability Automation Networks – Part 3: Parallel Redundancy Protocol (PRP) and High-availability Seamless Redundancy (HSR)," 2012.

[27] "Digital Data Communications for Measurement and Control – Fieldbus for Use in Industrial Control Systems (All Parts)," 2014.

[28] "Industrial communication networks – Profiles – Part 2: Additional fieldbus profiles for real-time networks based on ISO/IEC 8802-3," 2010.

[29] (2019, April) POWERLINK Basics: System Overview. https://www.ethernet-powerlink.org/uploads/media/POWERLINKBasics_brochure_e.pdf.

[30] "Aircraft Data Network-Part 7, Avionics Full Duplex Switched Ethernet Network (AFDX)," 2009.

[31] C. Fuchs, "The Evolution of Avionics Networks From ARINC 429 to AFDX," vol. 65, 01 2012.

[32] A. Amari, A. Mifdaoui, F. Frances, J. Lacan, D. Rambaud, and L. Urbain, "AeroRing: Avionics Full Duplex Ethernet Ring with High Availability and QoS Management," in *Proceedings of the 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, Jan. 2016.

[33] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer, "The time-triggered Ethernet (TTE) design," in *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, May 2005, pp. 22–33.

[34] H. Kopetz, "The Rationale for Time-Triggered Ethernet," in *2008 Real-Time Systems Symposium*, Nov 2008, pp. 3–11.

[35] "IEEE Standard for Local and Metropolitan Area Networks–Audio Video Bridging (AVB) Systems," *IEEE Std 802.1BA-2011*, pp. 1–45, Sept 2011.

[36] O. Kleineberg, P. Fröhlich, and D. Heffernan, "Fault-Tolerant Ethernet Networks with Audio and Video Bridging," in *ETFA2011*, Sept 2011, pp. 1–8.

[37] J. Cao, M. Ashjaei, P. J. L. Cuijpers, R. J. Bril, and J. J. Lukkien, "An independent yet efficient analysis of bandwidth reservation for credit-based shaping," in *2018 14th IEEE International Workshop on Factory Communication Systems (WFCS)*, June 2018, pp. 1–10.

[38] M. Ashjaei, G. Patti, M. Behnam, T. Nolte, G. Alderisi, and L. Lo Bello, "Schedulability analysis of Ethernet Audio Video Bridging networks with scheduled traffic support," *Real-Time Systems*, vol. 53, no. 4, pp. 526–577, Jul 2017.

[39] J. Cao, P. J. L. Cuijpers, R. J. Bril, and J. J. Lukkien, "Tight worst-case response-time analysis for ethernet AVB using eligible intervals," in *2016 IEEE World Conference on Factory Communication Systems (WFCS)*, May 2016, pp. 1–8.

[40] U. D. Bordoloi, A. Aminifar, P. Eles, and Z. Peng, "Schedulability analysis of Ethernet AVB switches," in *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, Aug 2014, pp. 1–10.

[41] "IEEE Standard for Local and Metropolitan Area Networks—Virtual Bridged Local Area Networks Amendment 14: Stream Reservation Protocol (SRP)," *IEEE Std 802.1Qat-2010 (Revision of IEEE Std 802.1Q-2005)*, Sept 2010.

[42] "IEEE Standard for Local and Metropolitan Area Networks - Virtual Bridged Local Area Networks Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams," *IEEE Std 802.1Qav-2009 (Amendment to IEEE Std 802.1Q-2005)*, pp. C1–72, Jan 2009.

[43] (2019, April) IEEE802—Time-Sensitive Networking (TSN) Task Group. https://1.ieee802.org/tsn/.

[44] "IEEE Standard for Local and Metropolitan Area Networks – Bridges and Bridged Networks - Amendment 25: Enhancements for Scheduled Traffic," *IEEE Std 802.1Qbv-2015 (Amendment to IEEE Std 802.1Q—as amended by IEEE Std 802.1Qca-2015, IEEE Std 802.1Qcd-2015, and IEEE Std 802.1Q—/Cor 1-2015)*, March 2016.

[45] "IEEE Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks–Amendment 29: Cyclic Queuing and Forwarding," *IEEE 802.1Qch-2017 (Amendment to IEEE Std 802.1Q-2014 as amended by IEEE Std 802.1Qca-2015, IEEE Std 802.1Qcd(TM)-2015, IEEE Std 802.1Q-2014/Cor 1-2015, IEEE Std 802.1Qbv-2015, IEEE Std 802.1Qbu-2016, IEEE Std 802.1Qbz-2016, and IEEE Std 802.1Qci-2017)*, pp. 1–30, June 2017.

[46] (2019, April) P802.1Qcr-Bridges and Bridged Networks Amendment: Asynchronous Traffic Shaping. https://1.ieee802.org/tsn/802-1qcr/.

[47] "IEEE Draft Standard for Local and Metropolitan Area Networks–Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks Amendment: Stream Reservation Protocol (SRP) Enhancements and Performance Improvements," *IEEE P802.1Qcc/D2.2, March 2018*, Jan 2018.

[48] "IEEE Standard for Local and Metropolitan Area Networks– Bridges and Bridged Networks - Amendment 24: Path Control and Reservation," *IEEE Std 802.1Qca-2015 (Amendment to IEEE Std 802.1Q-2014 as amended by IEEE Std 802.1Qcd-2015 and IEEE Std 802.1Q-2014/Cor 1-2015)*, March 2016.

[49] "IEEE Standard for Local and Metropolitan Area Networks–Frame Replication and Elimination for Reliability," *IEEE Std 802.1CB-2017*, Oct 2017.

[50] "IEEE Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks–Amendment 28: Per-Stream Filtering and Policing," *IEEE Std 802.1Qci-2017 (Amendment to IEEE Std 802.1Q-2014 as amended by IEEE Std 802.1Qca-2015, IEEE Std 802.1Qcd-2015, IEEE Std 802.1Q-2014/Cor 1-2015, IEEE Std 802.1Qbv-2015, IEEE Std 802.1Qbu-2016, and IEEE Std 802.1Qbz-2016)*, Sept 2017.

[51] D. Gessner, J. Proenza, M. Barranco, and A. Ballesteros, "A Fault-Tolerant Ethernet for Hard Real-Time Adaptive Systems," *IEEE Transactions on Industrial Informatics*, pp. 1–1, 2019.

[52] J. Wensley, L. Lamport, R. Shostak, C. Weinstock, J. Goldberg, M. Green, K. Levitt, and P. Melliar-Smith, "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proceedings of the IEEE*, vol. 66, no. 10, pp. 1240–1255, 2008.

[53] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, "Distributed Fault-Tolerant Real-Time Systems: the Mars Approach," *IEEE Micro*, vol. 9, no. 1, pp. 25–40, feb 1989.

[54] D. Powell, Ed., *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer Berlin Heidelberg, 1991.

[55] ——, *A Generic Fault-Tolerant Architecture for Real-Time Dependable Systems*. Boston, MA: Springer US, 2001.

[56] C. B. Nielsen, P. G. Larsen, J. Fitzgerald, J. Woodcock, and J. Peleska, "Systems of Systems Engineering: Basic Concepts, Model-Based Techniques, and Research Directions," *ACM Comput. Surv.*, vol. 48, no. 2, pp. 18:1—-18:41, 2015.

[57] R. Obermaisser, H. Kopetz, S. Kuster, B. Huber, C. El Salloum, R. Zafalon, F. Auzanneau, V. Gherman, K. Kronlof, H. Waris, G. Cristau, G. Edelin, P. Millet, M. Borth, C. Couvreuer, N. Suri, P. Bokor, D. Dobre, M. Serafini, and M. Hiller, *GENESYS: A Candidate for an ARTEMIS Cross-Domain Reference Architecture for Embedded Systems*, 2009.

[58] A. Eckel, P. Milbredt, Z. Al-Ars, S. Schneele, B. Vermeulen, G. Csertán, C. Scheerer, N. Suri, A. Khelil, G. Fohler, R. Obermaisser, and C. Fidi, "INDEXYS, a Logical Step beyond GENESYS," in *Computer Safety, Reliability, and Security*, E. Schoitsch, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 431–451.

[59] W. Weber, "Embedded Multi-Core Systems for Mixed Criticality Applications in Dynamic and Changeable Real-Time Environments," https://www.artemis-emc2.eu/, 2017.

[60] A. Larrucea, I. Martinez, J. Perez, V. Brocal, S. Peiró, H. Ahmadian, and R. Obermaisser, "DREAMS : Cross-Domain Mixed-Criticality Patterns," 2017.

[61] A. Ballesteros, J. Proenza, M. Barranco, and L. Almeida, "Reconfiguration Strategies for Critical Adaptive Distributed Embedded Systems," in *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2018.

[62] Avizienis and Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," *Computer*, vol. 17, no. 8, pp. 67–80, 1984.

[63] D. Powell, J. Arlat, L. Beus-Dukic, A. Bondavalli, P. Coppola, A. Fantechi, E. Jenn, C. Rabejac, and A. Wellings, "GUARDS: a Generic Upgradable Architecture for Real-Time Dependable Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 580–599, jun 1999.

[64] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," *Journal of the ACM*, vol. 27, no. 2, pp. 228–234, apr 1980.

[65] K. Goossens, J. Dielissen, and A. Radulescu, "Æthereal Network on Chip:Concepts, Architectures, and Implementations," *IEEE Design and Test of Computers*, vol. 22, no. 5, pp. 414–421, 2005.

[66] K. Tatas, K. Siozios, D. Soudris, and A. Jantsch, "The Spidergon STNoC," in *Designing 2D and 3D Network-on-Chip Architectures*. New York, NY: Springer New York, 2014, pp. 161–190.

[67] R. Obermaisser, H. Kopetz, and C. Paukovits, "A Cross-Domain Multiprocessor System-on-a-Chip for Embedded Real-Time Systems," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, pp. 548–567, nov 2010.

Inés Álvarez received the degree in computer science with a specialization in computer engineering from the University of the Balearic Islands (UIB), Palma de Mallorca, Spain, in 2014 and a master in computer engineering from the same University in 2016.

She is currently working towards the Ph.D. degree in information and communications technologies at UIB and she is a Lecturer with the Department of Mathematics and Informatics, UIB. Her research interests include dependable and real-time systems, fault-tolerant distributed systems and dependable communication topologies.

Ms. Alvarez is a Student Member of the IEEE since 2016.

Alberto Ballesteros received the first degree in informatics engineering (Ingeniería Superior en Informática) and the Master's degree in computing engineering from the University of the Balearic Islands, Palma de Mallorca, Spain, in 2012 and 2016, respectively.

He is currently working towards the Ph.D. degree in information and communications technologies in the same university. His research interests include real-time and fault tolerant distributed systems, adaptive systems and industrial networks.

Manuel Barranco received the first degree in informatics engineering (Ingeniería Superior en Informática) and the doctorate in informatics from the University of the Balearic Islands (UIB), Palma de Mallorca, Spain, in 2003 and 2010, respectively.

He is currently a Lecturer with the Department of Mathematics and Informatics, UIB. His research interests include dependable and real-time systems, fault-tolerant distributed systems, adaptive systems, dependable communication topologies, and field-bus and industrial networks.

Dr. Barranco is a member of the IES Technical Committee on Factory Automation.

David Gessner received the first degree in informatics engineering (Ingeniería Superior en Informática), the Master's degree in information and communication technologies and the doctorate in information and communication technologies from the University of the Balearic Islands, Palma de Mallorca, Spain, in 2010, 2011 and 2017 respectively.

His research interests include dependable and real-time systems, fault-tolerant distributed systems, adaptive systems, dependable communication topologies, and field-bus and industrial networks.

Sinisa Djerasevic received the first degree in informatics engineering from the University of Belgrade, Serbia, and the doctorate in information and communication technologies from the University of the Balearic Islands, Palma de Mallorca, Spain, in 2018.

His research interests include dependable and real-time systems, fault-tolerant distributed systems, adaptive systems and dependable communication topologies.

Julián Proenza (SM'12) received the first degree in physics (Licenciatura en Ciencias Físicas) and the doctorate in informatics from the University of the Balearic Islands (UIB), Palma de Mallorca, Spain, in 1989 and 2007, respectively.

He is currently a Lecturer with the Department of Mathematics and Informatics, UIB. His research interests include dependable and real-time systems, fault-tolerant distributed systems, adaptive systems, clock synchronization, dependable communication topologies, and field-bus and industrial networks.

Dr. Proenza is a Member of the IEEE Industrial Electronics Society (IES) since 2009 and Senior Member since 2012. He is also a member of the IES Technical Committee on Factory Automation.