BULGARIAN ACADEMY OF SCIENCES

PROBLEMS OF ENGINEERING CYBERNETICS AND ROBOTICS • 2023 • Vol. **79**, pp. 37-54
p-ISSN: 2738-7356; e-ISSN: 2738-7364
https://doi.org/10.7546/PECR.79.23.02

Redundancy Management in Dependable Distributed Real-Time Systems

Edita Djambazova[1], *Rumen Andreev*[1]

[1] *Institute of Information and Communication Technologies, Bulgarian Academy of Sciences, Sofia, Bulgaria*
Emails: edita.djambazova@iict.bas.bg, rumen@isdip.bas.bg

Abstract: Dependable distributed real-time systems are usually deployed in safety-critical applications. One of the major requirements for their operation is to be fault-tolerant. Fault tolerance is achieved by using various techniques, most of which are based on redundancy. Traditionally, redundancy in dependable distributed real-time systems is applied at a component level and in communication but it could take many forms. The presented survey investigates the methods and techniques to realize redundancy in dependable distributed real-time systems and attempts to systemize the variety of approaches to implement it. Although redundancy is a well-studied fault-tolerance method, it deserves attention in order to broaden the view on its application and seek opportunities for optimized solutions.

Keywords: *dependable distributed real-time systems, redundancy, replication, fault-tolerance techniques, system design*

1. Introduction

Dependable distributed real-time systems are often implemented in safety-critical applications where the requirements for fault-tolerant operation are essential. Incorporating fault tolerance means guarantees for error detection and removal. Redundancy is a key method to achieving fault tolerance. As its name suggests, it is something that the system could do without. Most systems operate correctly without using additional elements. For dependable systems, however, where there is a requirement for high system reliability even in the presence of faults, one of the traditional techniques to apply redundancy is replication.

Since introducing redundancy incurs excessive resources and hence increased costs, it is implemented in systems that have to guarantee that no catastrophic failure could occur. Examples of such systems are safety-critical applications, high-reliability systems, mission-critical systems, automobiles, etc. The additional costs of having redundant elements are justified by the potential consequences of a system failure which could cause severe damage to equipment, loss of critical functionality, or even injury or death of people. Researchers, designers, and engineers seek ways of achieving the needed dependability at the optimal price.

Dependable distributed real-time systems use replication techniques at different levels. Since these systems are usually intended for safety-critical applications, fault tolerance is integrated into their design. Distributed real-time systems are built out of functional units that exchange messages with each other through a communication channel in order to fulfill a common control task. Replication is applied in the functional units, in the communication channel, in the tasks, etc. Where to use replicated components is decided at the design stage of the dependable system.

The paper discusses dependable distributed real-time systems. For the sake of brevity, we will use the terms distributed systems and distributed real-time systems equivalently throughout the text.

A distributed real-time system (Fig. 1) contains relatively autonomous functional units (often called in the literature fault containment units – FCUs [14, 17, 23]) that exchange messages through a communication bus and execute a common algorithm. We use the term components for the functional units throughout the paper.

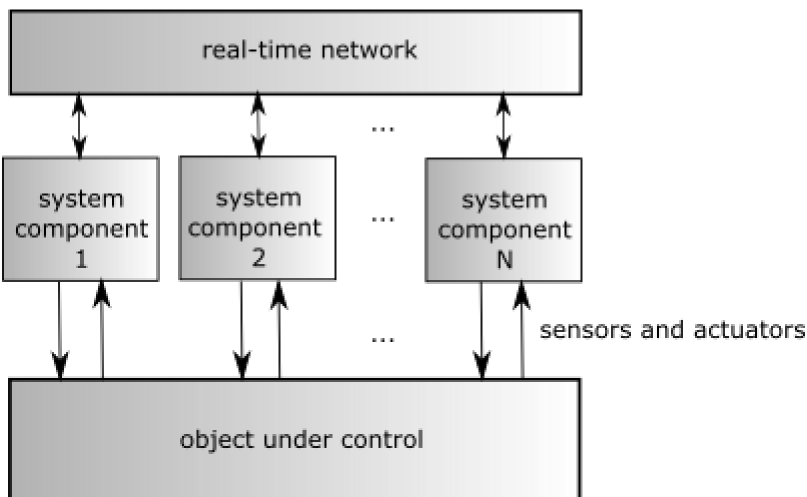

Fig. 1. Distributed real-time system

The system controls an industrial process called the object under control. The components take inputs from the sensors of the object under control, run a control program that calculates some results, and output these results to the actuators of the object. To fulfill their task, they need to communicate with the other components by exchanging data. System components are designed to have safe behaviour, i.e., a fault should not reach the outputs of a component. Fault containment is an important property of a component. It guarantees reliable system operation. To achieve a fault-tolerant behaviour of the components, they are built out of replicated modules [2, 3, 7, 9, 17]. Replication at the component level can be hardware and software. Additional error detection means are also applied in each module [15, 30, 33, 38]. The bus itself is often replicated [12, 15, 23, 39].

Our survey focuses on component redundancy. It studies the redundancy methods and replication techniques applied at a component level in an attempt to systemize their great variety and find out some possibilities for optimization.

The paper is organized into three sections studying the role and the place of redundancy in distributed real-time system design. The system design from a dependability perspective is discussed in Section 2 where the process of incorporating means for fault tolerance is described. The methods and techniques to implement redundancy in distributed real-time systems are surveyed in Section 3. The realization of redundancy in dependable distributed systems is discussed in Section 4. Section 5 concludes the survey.

2. System design

The distributed system development cycle can be presented as an iterative process. It considers the system from two viewpoints: practical and abstract. The practical view of the system is its *implementation*. In its working environment, the system is prone to various impairments that prevent it from correct operation. The abstract view of the system is its *model*. The system model takes into account the system specifications, the intended application, the desired functionality, and the system architecture. To design it some theoretical knowledge is needed. These viewpoints of the system have to exchange data with each other to achieve a complete system model that can be validated and verified.

There are two approaches to the system design: top-down and bottom-up. The top-down strategy takes the way from a high level of abstraction through the application of certain requirements to a model relevant to the application. The bottom-up strategy is to take some specific technical parameters of the system and go all the way to the design and realization of a system to build a model to verify and validate the system. The two approaches should meet in the design of a system.

Initially, the bottom-up strategy defines the problem under consideration. The problem is a practical issue that needs to be resolved. It usually has a real-world origin, e.g. a system suffering from a malfunction. In the context of the paper, the studied problem is *dependability* as a general concept (Fig. 2). The designed system needs to be dependable since it will be operated in critical applications. System engineers try to find a solution to that problem. Firstly, they need to understand the issue, i.e., to see the structure of the problem. To achieve the *fault tolerance* required by the application they have to decide which methods are appropriate. System reliability is one of the attributes of dependability [1, 19, 20]. There are various means to obtain a certain level of reliability – fault prevention, fault tolerance, fault removal, and fault forecasting [1, 19]. Fault tolerance is the focus of the presented survey. It is mostly achieved through *redundancy*. The types and modes of redundancy are discussed in Section 3.

System designers develop a strategy to reach the solution. In this effort, they look for some basic knowledge. At this point, the research methods and models may help find better solutions. Many fault-tolerance techniques are used in distributed real-time systems. Most of them are based on redundancy, such as N-modular redundancy, N-version programming, software diversity, recovery blocks, etc.

To establish the strategy for problem resolution the developers also use some specific properties of the operational environment of the system and its context. The context is the application area of the system under consideration. It influences those characteristics of the environment that are relevant to the developed system. The result of this iterative process is a plan for the realization of the system that is a solution to the initial problem.

In terms of redundancy, the described process looks as follows (Fig. 2). A distributed system controls an industrial process, i.e. the distributed computing environment. Along with the issues concerning communication among the constituent components, the system is subject to faults. These faults disturb its operation and threaten to harm the process under control leading to undesired consequences. The problem with the system's *dependability* becomes a design issue: how to make the system *fault-tolerant*. Faults could have many causes and could occur under various conditions, their instance is unknown. And they are inevitable, they are always present in the systems. Since the faults are unpredictable, the system should have the resources to deliver its intended service even in the presence of faults. *Redundancy* is one of the strategies to resolve the problem with the needed fault tolerance. The engineering issues with redundancy implementation, such as coordination among the replicas, designing self-checking tools, choosing a fault/failure mode, etc., have to be combined with the research solutions, e.g., defining a system model, developing component models, studying different operational scenarios with the models, fault-tolerance techniques, reliability assessment, etc. We combine them under the name *dependability*

assessment. The models and their parameters depend on the *application* (e.g., cyber-physical systems, safety-critical systems, automobiles, etc.) and the operational environment, i.e., the *distributed computing environment*. The dependability attributes are determined based on the particular application. The results obtained from the models' investigation are used in the system's design. The appropriate *replication technique* is identified.

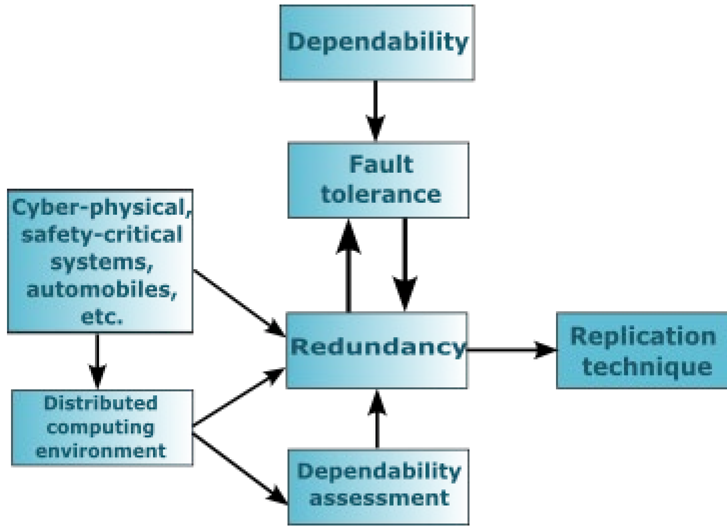

Fig. 2. The process of applying redundancy.

Component replication can be viewed through the prism of the system design strategy. After identifying the appropriate replication technique the design process can enter the known schemes for system design. There are well-established approaches to system design in general and the design of engineering systems in particular. Our general approach gives a view to dependable system design and helps see the opportunities to implement redundancy in a dependable distributed system. It offers input knowledge to the process of system design introducing the fault-tolerance perspective.

System design schemes help construct a system from the initial idea to the final implementation [36]. The process is standardized and broadly applied to develop systems with the desired properties. This is known as System Development Life Cycle (SDLC). The SDLC is applied in the following design stages: planning, analysis, design, implementation, and maintenance[1]. Many models include the stage of testing before the implementation[2] since it is an important part of the process. During the planning stage, the concept and the

[1] https://en.wikipedia.org/wiki/Systems_development_life_cycle
[2] https://www.clouddefense.ai/blog/system-development-life-cycle

general system requirements are identified. As the process matures, the functional requirements of the system are defined. Based on them, the system architecture is developed. The development process enters into more detail and the system hardware and software are designed. The system is ready for implementation but it should go through component testing, integration testing, system testing, and acceptance testing.

In [25], the process of reliable systems development, called Design for Reliability (DfR), is presented in six engineering activities: identify, design, analyze, verify, validate, and control. It aligns with the described general process of problem solving (Fig. 2).

The described process of determining the replication technique taking into consideration many aspects (Fig. 2) can be used as a prerequisite to the SDLC scheme or be incorporated into it, thus specifying dependability requirements.

3. Redundancy in dependable distributed real-time systems

As already outlined, redundancy is a functionality or a component of a computer system that is not needed for the system to perform its normal operation but is necessary in case of fault occurrence to guarantee system service delivery in these exceptional circumstances. It is a method of implementing fault tolerance in dependable computer systems and is in turn realized by replication techniques.

Redundancy can be structural, time, or functional [7, 10, 18, 34, 35]. Structural redundancy represents the introduction of additional elements (hardware or software) to the system architecture to take over the function of the failed ones. It can take different replication styles and degrees. Replication styles define if the replication is active, passive, or a combination of them [8, 22]. Active structural redundancy is realized through hot spare components that operate concurrently with the primary one executing the same task and comparing their results [34, 35]. Passive structural redundancy uses spare modules for the primary module. One of the spares becomes active when the primary module fails [34, 35]. The replication degree [8, 22], determines the number of replicas of a component – one, two, or three.

Time redundancy is realized through repetition. A control routine can be executed twice and the results can be compared. In case of a detected fault, the sending component can remain silent and try to recover. Another way to apply temporal redundancy is to send a failed message again. Time redundancy has to be applied carefully in real-time systems to preserve their correctness in the real-time domain.

Functional redundancy means the replication of some functions with different means, e.g. software diversity, acceptance tests, etc. Here, the replicas are not identical but their functions are similar.

Information redundancy is used in the coding theory and is applied in computing systems but is out of the scope of this paper.

The redundancy types are described in more detail in the next subsections.

3.1. Replication style

The replication style defines the way the replicated components perform their operation. The fault-tolerant components have replicated modules and only one of them, called primary, issues the output result. The other replicates are secondary. Depending on the replication style the redundancy can be passive or active.

Sparing

Sparing is a passive form of redundancy [7, 10, 20]. Only the primary module executes the application, and the other replicas are its spares. Spare modules are added to the primary one in order to take over its functions in case of its fault. In hardware, this is implemented as a standby sparing. The standby module is identical to the active unit but does not perform in the system operation until the primary module fails. In that case, it becomes active and the failed module can be diagnosed and eventually repaired. Sparing can be applied in two ways depending on how the state of the primary module is transferred to the spare. In cold sparing, a spare can take the functions of the primary only when it crashes and then retrieves the state from a log saved on shared storage. In warm sparing, the spares are in standby mode and periodically receive state updates from the primary.

Replication

Replication means that identical modules execute the same functions on the same inputs, and compare their results [7, 10, 20, 34, 35]. The replicas could work as active or passive spares (see Sparing). The active spares work together concurrently but only the primary component issues the result to the object under control.

Replication techniques can be implemented at different parts of the dependable system's architecture. They can be realized in the hardware, the software, the communication, and the time-dependent elements of the designed system. Passive and active redundancy can be applied together making many combinations.

3.2 Replication degree

The replication degree determines the number of modules in a component. Depending on the importance of a component for the system operation, it can have one or more replicates, or not be replicated at all.

In hardware, the replication takes the form of N-modular redundancy (NMR) [7, 18, 20, 34, 35]. It is mostly applied as dual modular redundancy (DMR)

and triple modular redundancy (TMR). In DMR, the two replicas compare their results and, in case of discrepancy, the component does not issue any result, remaining fail-silent. Usually, the modules have additional fault-tolerance mechanisms (called self-checking units) to decide which module is faulty. In TMR, there are three active modules and a voter. Voting over the results allows for the masking of one fault. Depending on the failure mode, the component can continue operating with two (or even one) non-faulty modules equipped with self-checking tools.

The replication in software is realized as recovery blocks (RB) [31, 32] and N-version programming (NVP) [31]. In the RB approach, two alternates are produced from a common service specification and an acceptance test decides whether the result is correct. The acceptance test is applied sequentially to the results of the alternates. If the results of the primary alternate do not pass the acceptance test, the second alternate is executed. The RB approach corresponds to the stand-by sparing in hardware.

In the N-version programming, there are N ($N \geq 2$) variants of the software that are executed simultaneously, and their results are compared. The variants are software routines that are written by different programming teams and possibly using different algorithms. This is supposed to avoid the common errors that programmers tend to do. The results of the software versions are voted upon and the majority result is issued. The hardware equivalent of the NVP is the NMR.

N Self-Checking Programming (NSCP) [31] is another approach to redundancy applied in software. In the NSCP approach, N self-checking software components are executed; one of them is considered as acting and the other self-checking components are considered as hot spares. Upon failure of the acting component, the operation is switched to a spare self-checking component.

3.3. Time redundancy.

Time redundancy requires additional time to be allocated to the task execution [7, 18, 34, 35]. The control routine is executed twice and the results are compared [15] in order to override the effect of a transient fault. In case of a difference, the component remains silent. Another form of temporal redundancy in dependable distributed systems is to send a message again if a fault is reported before the first transmission [15, 34, 35].

The fault in the message is detected by some embedded mechanisms such as CRC code, comparison with the results of the other components, etc. Time redundancy has lower overhead compared to structural redundancy but it may impact the system’s performance and should obey the real-time constraints.

3.4. Functional redundancy.

Functional redundancy is implemented in the software. In [10], it is defined as qualifying the system's behaviour relative to its inputs/outputs relationships. According to the definition given in [10], a system has functional redundancy in three cases: (i) if some theoretically possible input values are not applicable according to the system's specifications; (ii) if some theoretically possible output values are not produced according to the system's specifications; and (iii) if some theoretically possible input/output values never occur according to the system's specifications. Functional redundancy is useful in error detection.

4. Redundancy realization

The described forms of redundancy can be applied in combination and/or at different system levels [37]. In dependable distributed systems, the components are physically replicated (hardware redundancy), the software modules are allocated at different processors (software redundancy), the communication channels could also be replicated, and time redundancy is often applied.

Some systems use a kind of redundancy hierarchy. They define the levels of importance of their elements and introduce a communication procedure among the levels. In [28], integrity and criticality levels are determined.

Dependability in distributed real-time systems for safety-critical applications became a part of their design. All parameters and system components that are important for the fault-tolerant system operation are included in the SDLC. The system design point of view on structural redundancy is illustrated in Fig. 3.

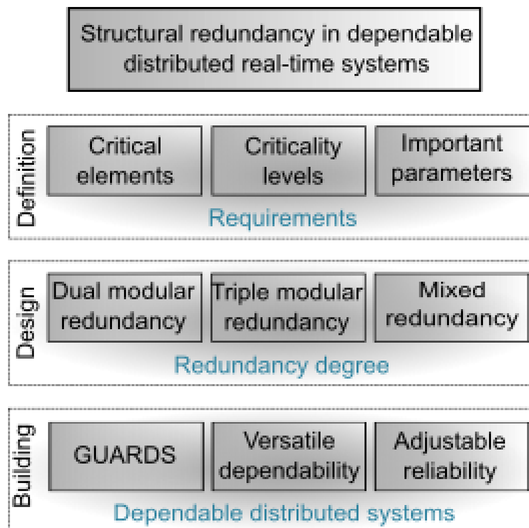

Fig. 3. Applying SDLC in structural redundancy for dependable distributed systems

Introducing structural redundancy involves the definition of the critical elements, the important system parameters, and the criticality levels (if they are to be included). The components of the distributed system control different parameters of the object under control with different significance for the fault-tolerant operation. At the stage of defining the requirements in SDLC, the controlled parameters should be defined and the criticality levels should be outlined. The components controlling the important parameters will receive a high criticality level and need to be fault-tolerant. At the stage of system design, the replication degree and style should be determined. The replication degree defines if SMR (single modular redundancy), DMR, or TMR will be applied. The replication style chosen determines whether active or passive replication will be used. Modules in system components can be evenly distributed, i.e., all components can have an equal number of modules, or can use mixed redundancy. The actual building stage of the SDLC implements the decided system architecture, e.g., GUARDS [28, 29], versatile dependability [8, 22], or adjustable reliability [5], as shown in Fig. 3.

Replication is rarely applied in pure types in dependable distributed systems. There is always a mixture of replication styles at various system levels. In hardware, the system's components can be replicated identically or depending on their criticality. Software components can also have a different number of copies on different components/processors to achieve better use of system resources. In the scheduling of dependable distributed systems, approaches for mixed criticality are implemented that meet the fault-tolerance and real-time requirements [4, 13].

4.1. Implementing different replication degrees

Equal redundancy for all components

The most straightforward way of applying redundancy is to replicate the active components and compare their results. In distributed systems, the nodes can be built out of two or three identical modules executing the same task (Fig. 4).

The results of the replicas are compared (as in [11, 15, 31, 38]) or voted in the case of TMR (as in [14, 17]). If no discrepancy is shown, the presumably correct result is issued to the object under control. In case of a mismatch, the result is not issued and the node is put in a safe state according to the system conventions.

The system's hardware components have equal replication degrees but the software components (execution tasks) may have different redundancy. For example, there are three tasks in Fig. 4 – A, B, and C; task A has three replicates, task B has one, and task C has two. The replicates may reside on different system components, thus allowing for error isolation.

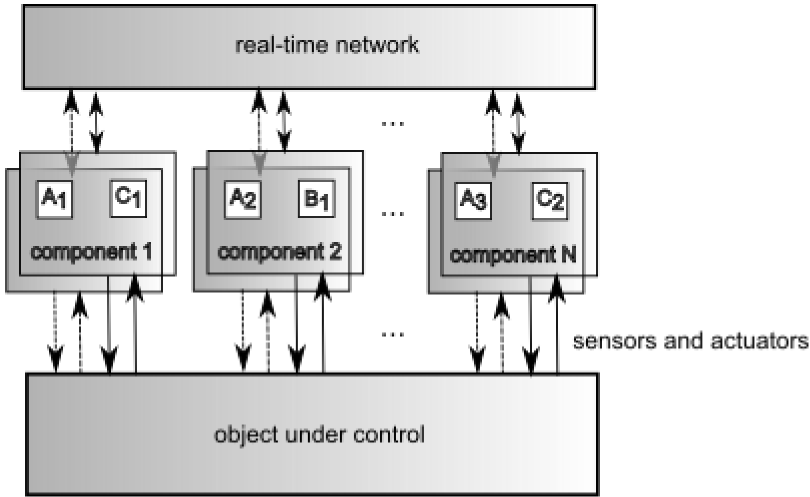

Fig. 4. Dependable distributed real-time system

The two historically established directions in dependable distributed systems are to build an open architecture as in Delta-4 [27, 30] or a strictly determined architecture as in MARS [15]. The Delta-4 project defines a real-time computer architecture that uses replicated software components allocated at different host computers. Host computers are fault-tolerant (hardware replication). A homogeneous set of fault-tolerance techniques is determined that allows building various systems and applications with different dependability and performance requirements to be configured without redesigning the software components.

In MARS [15], the aim is to build a predictably dependable distributed real-time system based on hard real-time constraints. The system is designed to preserve its functionality even in peak-load conditions. System components use active redundancy and self-checking, the communication bus is also duplicated, and time redundancy is used for the tasks. The active redundancy is physically applied by DMR, i.e., two components execute the same tasks. The time redundancy is achieved by sending each message twice. The MARS components have self-checking mechanisms that guarantee their fail-stop behaviour.

Many dependable distributed systems are built to offer re-usability of the components, lower design costs, and introduce flexibility to the application requirements. The DEAR-COTS system [26, 38] is an approach to reducing design costs by implementing commercial off-the-shelf (COTS) components into dependable distributed systems. The system is built using distributed nodes to execute hard real-time applications. The hard real-time subsystem (HRTS) of DEAR-COTS is based on the software integration of hardware COTS components. One hard real-time application is divided into several tasks, which

execute in different nodes of the HRTS. Each node has its own COTS kernel and hardware and the HRTS support software provides the application distribution and replication management. Active replication of dissimilar replicated task sets is used in DEAR-COTS.

The use of COTS components also lies in the architecture of GUARDS [29]. The generic fault-tolerant computer architecture is defined along three dimensions of fault containment: integrity levels, lanes, and channels. Fault tolerance and integrity management are software-implemented, with a minimum of specialized hardware. Integrity levels are intended to prevent design faults from propagation outside a containment region. The aim is to protect critical components from errors due to design faults in less-critical components. The integrity level reflects the degree to which an application object is trustworthy. The GUARDS architecture uses multiple processors or lanes to detect and diagnose physical faults within a channel. Channels are the ultimate line of defence against physical faults in a single channel. Fault tolerance is achieved through active replication of application tasks over the set of channels.

The Time-Triggered Architecture (TTA) [14, 16, 21] extends the MARS approach to give a way of building dependable distributed real-time systems. A TTA component is a fault-containment unit. Fault-tolerant units are built out of replicated components to mask arbitrary faults in components. TTA components interact with the environment solely by the exchange of messages. They have message-based linking interfaces that are independent of the implementation technology of the components. The TTA architecture provides a globally synchronized time base. The message exchange among the modules runs over two replicated channels. The smallest replaceable unit (SRU) [16, 17] consists of a host subsystem and a communication subsystem. The host subsystem executes the application software and the communication subsystem is responsible for communication protocol execution. One or more SRUs can form a fault-tolerant unit.

Different redundancy for the components

Some systems use different degrees of replication for their components. Some of the controlled system parameters or some system components are more important for the system's operation than others. The importance of such components is defined by the consequences of their failure. If the system cannot afford to lose some of its functionality because that would lead to a catastrophic failure, the component is considered critical.

In GUARDS [29], there are integrity levels and criticality levels. The integrity levels are defined according to the degree to which a system component can be trusted – the more trustworthy a component is, the higher its integrity level [28]. The degree of trust placed on a component depends on its criticality. Critical components are considered those whose failure consequences are severe. Such

components have a higher degree of integrity. Integrity levels are a part of the integrity policy that prevents flows of information from low to high integrity levels. Criticality levels are related to integrity levels but address different issues. Integrity levels, through the integrity policy, define the allowed data flows between the levels and resource utilization by the components of different levels. Criticality levels define the critical system components in terms of the potential consequences of failures of components at each level [29].

The DEAR-COTS model of replication [26, 38] allows for defining the replication degree of specific parts of the real-time application accordingly to the reliability of the components and the desired level of reliability for the application. An N-replicated component is defined. There is a Replica Manager layer to provide the resources for communication between the distributed tasks and replicated components. A deterministic replica execution must be guaranteed.

The DECOS project [11, 24] proposes an integrated distributed architecture to support mixed-criticality systems. Mixed-criticality systems consist of distributed application parts with different criticality levels on top of the same physical hardware. The DECOS system is based on the principles of "strong fault encapsulation, fault tolerance by means of replication and redundancy, and separation of safety-critical from non-safety-critical functionality [11]." The functional distribution following these principles leads to the structure with a number of Distributed Application Subsystems (DASs) and jobs. The system service is executed by clusters that are sets of distributed node computers interconnected by a time-triggered network. The node computers provide the protected execution environment for the jobs through encapsulated partitions. According to the federated approach, application parts with different criticality levels are implemented as self-contained units with their own processing and input/output systems. In DECOS, the non-interference of the encapsulated partitions allows the nodes to host multiple jobs belonging to different DASs, with different levels of criticality.

Versatile dependability [8, 22] is an approach to building dependable software architectures considering three important aspects – fault tolerance, performance, and resources. It provides a set of tools, called "knobs", for tuning the trade-offs between these aspects. The versatile dependability framework distinguishes between low-level and high-level knobs. The internal fault-tolerance properties, such as the replication style, the minimum number of replicas, the checkpointing intervals, the fault monitoring intervals, etc., are the low-level knobs. The high-level knobs correspond to the external properties of the system, e.g., scalability and availability. The versatile dependability framework is based on fault-tolerant middleware specifications. A distributed asynchronous system is assumed. A special software module, a replicator, manages groups of client and server replicas. The replication is implemented at the process level, since a process

may contain several objects that have to be recovered in case of a process crash. The replicator supports active and passive replication styles.

Most of the dependable distributed real-time systems follow the architectural style depicted in Fig. 4. They employ equally replicated physical components and mixed redundancy of the software components. There are approaches that propose different degrees of replication of the hardware components (Fig. 5), as in the dependable distributed system with adjustable reliability [5].

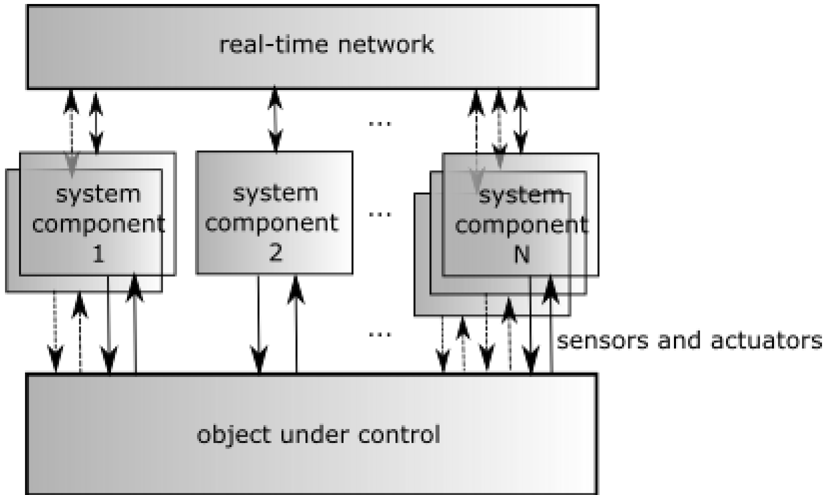

Fig. 5. Dependable distributed real-time system with adjustable reliability

The system with adjustable reliability [5] applies modules with different replication degrees – single modules, modules with dual modular redundancy, and modules with triple modular redundancy. Modules are self-contained units according to dependability terminology, i.e., no fault can propagate from one component to another. The replication degree is determined according to the total reliability required by the application. Using the approach of adjustable reliability, the module redundancy distributions that satisfy the application requirements are identified. The necessity to have components with different replication degrees is related to their criticality. Unlike the approaches implementing mixed criticality of the software application parts executed over evenly replicated hardware, the reliability adjustment approach proposes a component's redundancy degree to be determined according to its criticality at the design stage and the fault-containment components to operate with mixed redundancy. Although this approach may not offer the flexibility of the software-implemented dependability solutions the results of its simulation modelling [6] show that there are redundancy distributions that achieve the total system reliability required by the application.

5. Conclusion

Implementing redundancy is an essential technique to achieve fault tolerance in dependable distributed real-time systems. Different forms of replication are used to guarantee fault containment, error detection, and safe operation of the system. Dependable distributed real-time systems use redundant hardware and software components, replicate communication channels, and apply time redundancy. The extensive use of various forms of redundancy renders the systems complex and resource-consuming and affects their performance. Many approaches have been proposed to building highly reliable distributed systems that make efficient use of their resources, satisfy the dependability requirements of a real-time application, and allow to relax the design efforts.

We have presented the process of redundancy implementation from the perspective of the system design cycle. Identifying the replication technique is a process that considers both the realization specifics and the theoretical knowledge about the designed system. The engineering perspective includes the requirements of the application, the real system context, the fault-tolerance techniques, etc. The abstract point of view proposes the system model that takes into account the dependability attributes and assumptions relevant to the system, i.e., reliability, availability, the fault/failure mode, the system reliability assessment, the possible fault scenarios, etc. The chosen replication technique as a result of that effort can then be included in the system design cycle.

Various forms of redundancy implementation were developed to suit the intended system applications. To render the dependable distributed systems more flexible and effective from the performance viewpoint mixed replication styles and replication degrees are incorporated. They reflect the criticality levels determined for the fault-containment components. Using different degrees of replication in dependable distributed real-time systems allows the efficient use of system resources without sacrificing the dependability requirements dictated by the application.

Acknowledgment

This research is supported by the Bulgarian FNI fund through the project "Modelling and Research of Intelligent Educational Systems and Sensor Networks (ISOSeM)", contract КП-06-Н47/4 from 26.11.2020.

References

1. Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing, IEEE Transactions on Dependable and Secure Computing 1(1), 11–33 (2004), DOI: 10.1109/TDSC.2004.2.

2. Barranco, M., Derasevic, S., Proenza, J.: An Architecture for Highly Reliable Fault-Tolerant Adaptive Distributed Embedded Systems. Computer 53, 38–46 (2020), DOI: 10.1109/MC.2019.2944337.
3. Birman, K. P.: Reliable Distributed Systems Technologies, Web Services, and Applications. Springer New York, NY (2005).
4. Burns, A. and Davis, R. I.: Mixed Criticality Systems – A Review. University of York, UK (2022).
5. Djambazova, E.: A Fault-Tolerant Real-Time System with Adjustable Reliability, ACM International Conference Proceeding Series, CompSysTech'21 - Ruse, Bulgaria, Association for Computing Machinery (ACM), New York, USA, pp. 76–80 (2021), DOI: 10.1145/3472410.3472415.
6. Djambazova E.: Achieving System Reliability Using Reliability Adjustment, ACM International Conference Proceeding Series, International Conference on Computer Systems and Technologies 2022 (CompSysTech '22), Ruse, Bulgaria, ACM, New York, USA, pp. 64–68 (2022), DOI: 10.1145/3546118.3546129.
7. Dubrova, E.: Fault-Tolerant Design. Springer Science+Business Media New York (2013), DOI: 10.1007/978-1-4614-2113-9.
8. Dumitras, T., Srivastava, D. and Narasimhan, P.: Architecting and Implementing Versatile Dependability. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds) Architecting Dependable Systems III. LNCS, vol. 3549 (2005), DOI: 10.1007/11556169_10.
9. Erciyes, K.: Distributed Real-Time Systems Theory and Practice. Computer Communications and Networks Series, Springer Cham (2019), DOI: 10.1007/978-3-030-22570-4.
10. Geffroy, J.-C. and Motet, G.: Design of Dependable Computing Systems. Springer Netherlands (2002), DOI: 10.1007/978-94-015-9884-2.
11. Herzner, W., Schlick, R., Schlager, M., Leiner, B. et al.: Model-Based Development of Distributed Embedded Real-Time Systems with the DECOS Tool-Chain, SAE Technical Paper 2007-01-3827 (2007), DOI: 10.4271/2007-01-3827.
12. Isermann, R.: Fault Diagnosis Systems. An Introduction from Fault Detection to Fault Tolerance. Springer, New York (2006), DOI: 10.1007/3-540-30368-5.
13. Islam, S., Lindstrom, R., and Suri, N.: Dependability driven integration of mixed criticality SW components. In Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06), Gyeongju, South Korea, pp. 485-495 (2006), doi: 10.1109/ISORC.2006.26.
14. Kopetz, H. and Bauer, G.: The time-triggered architecture. In Proceedings of the IEEE, 91(1), 112–126 (2003), DOI: 10.1007/0-306-47055-1_14.
15. Kopetz, H., Damm, A., Koza, C., Mulazzani, M., Schwabl, W., Senft, C., Zainlinger, R.: Distributed fault-tolerant real-time systems: The Mars approach. IEEE Micro 9 (1), 25–40 (1989), DOI: 10.1109/40.16792.
16. Kopetz, H.: The time-triggered model of computation. In Real-Time Systems Symposium (RTSS98). IEEE Press. Madrid, Spain. pp. 168–177 (1998), DOI: 10.1109/REAL.1998.739743.
17. Kopetz, H.: Real-Time Systems, Design Principles for Distributed Embedded Applications. 2nd ed. Real-Time Systems Series, Springer (2011).

18. Koren, I. and Krishna, C. M.: Fault-Tolerant Systems (2nd ed.). Elsevier Inc. (2021), DOI: 10.1016/C2018-0-02160-X.
19. Laprie, J.C.: Dependability: Basic Concepts and Terminology. Springer: Berlin/Heidelberg, Germany (1992).
20. Lee, P. A., Anderson, T.: Fault tolerance: principles and practice, 2nd ed. Springer, (1990).
21. Maier, R., Bauer, G., Stöger, G., and Poledna, S.: Time triggered architecture: A consistent computing platform, IEEE Micro 22(4), 36–45 (2002), DOI: 10.1109/MM.2002.1028474.
22. Narasimhan, P., Dumitras, T. A., Paulos, A. M., Pertet, S. M., Reverte, C. F., Slember, J. G. and Srivastava, D.: MEAD: support for Real-Time Fault-Tolerant CORBA. In Concurrency and Computation: Practice and Experience 17, 1527–1545. Wiley InterScience (2005), DOI: 10.1002/cpe.882.
23. Obermaisser, R., Kopetz, H., Kuster, S., Huber, B., El Salloum, C., Zafalon, R., Auzanneau, F., Gherman, V., Kronlof, K., Waris, H., et al. GENESYS: A Candidate for an ARTEMIS Cross-Domain Reference Architecture for Embedded Systems, Suedwestdeutscher Verlag fuer Hochschulschriften (2009).
24. Obermaisser, R., Peti, P., Huber, B., and El Salloum, C.: DECOS: An integrated time-triggered architecture. Elektrotech. Inftech. 123, 83–95 (2006), DOI: 10.1007/s00502-006-0323.
25. O'Connor, P. D. T., and Kleyner, A.: Practical Reliability Engineering. Fifth Edition, John Wiley & Sons, Ltd (2011), DOI:10.1002/9781119961260.
26. Pinho, L. M., Vasques, F., and Wellings, A.: Replication management in reliable real-time systems. Real-Time Systems 26, 261–296 (2004), DOI: 10.1023/B:TIME.0000018248.18519.46.
27. Powell, D. (Ed.): Delta-4: A Generic Architecture for Dependable Distributed Computing, Springer: Berlin/Heidelberg, Germany (1991).
28. Powell, D. A Generic Fault-Tolerant Architecture for Real-Time Dependable Systems, Springer: Boston, MA, USA (2001).
29. Powell, D., Arlat, J., Beus-Dukic, L., Bondavalli, A., Coppola, P., Fantechi, A., Jenn, E., Rabe´jac, C., and Wellings, A.: GUARDS: A Generic Upgradable Architecture for Real-Time Dependable Systems, In IEEE Trans. Parallel and Distributed Systems, special issue on dependable real-time systems, 10(6), 580–599 (1999).
30. Powell, D., Bonn, G., Seaton, D., Verissimo, P., and Waeselynck, F.: The Delta-4 Approach to Dependability in Open Distributed Computing Systems. In Proceedings of the Eighteenth International Symposium on Fault-Tolerant Computing. Digest of Papers, 246–251 (1988), DOI: 10.1109/FTCS.1988.5327.
31. Randell, B., Laprie, J.-C., Kopetz, H., and Littlewood, B.: Predictably dependable computing systems. Springer (1995).
32. Randell, B. and Xu, J.: The evolution of the recovery block concept, in M. Lyu (Ed.), Software Fault Tolerance, Wiley, 1995, pp. 1–21.
33. Rostamzadeh, B., Lonn, H., Snedsbol, R., and Torin, J.: DACAPO: A Distributed Computer Architecture for Safety-Critical Control Applications. In Proceedings of the Intelligent Vehicles'95. Symposium, 376–381 (1995), DOI: 10.1109/IVS.1995.528311.

34. Shooman, M. L.: Reliability of Computer Systems and Networks: Fault Tolerance, Analysis, and Design. John Wiley & Sons, Inc. (2002).
35. Sorin, D. J.: Fault Tolerant Computer Architecture. Morgan and Claypool Publishers (2009).
36. Stetter, R.: Fault-Tolerant Design and Control of Automated Vehicles and Processes, Insights for the Synthesis of Intelligent Systems. Studies in Systems, Decision, and Control 201, Springer Nature Switzerland AG (2020), DOI: 10.1007/978-3-030-12846-3.
37. Tanenbaum, A. S. and van Steen, M.: Distributed Systems: Principles and Paradigms. 2nd ed. Pearson Prentice Hall (2017).
38. Veríssimo, P., Casimiro, A., Pinho, L. M., Vasques, F., Rodrigues, L., and Tovar, E.: Distributed Computer-Controlled Systems: the DEAR-COTS Approach. In IFAC Proceedings 33(30), 113–120 (2000), DOI: 10.1016/S1474-6670(17)36739-3.
39. Veríssimo, P. and Rodrigues, L.: Distributed Systems for System Architects. Springer, Boston, MA. (2001), DOI: 10.1007/978-1-4615-1663-7.