An Overview of Fault Tolerance Techniques for Real-Time Operating Systems

Reza Ramezani* Yasser Sedaghat**

Dependable Distributed Embedded Systems (*DDEmS*) Laboratory: *http://ddems.um.ac.ir*
Department of Computer Engineering
Ferdowsi University of Mashhad, Mashhad, Iran

reza.ramezani@stu.um.ac.ir* ***y_sedaghat@um.ac.ir****

***Abstract*- Nowadays operating systems are inseparable part of computer systems. Real-time operating systems (RTOS) are a special kind of operating systems that their main goal is to operate correctly and provide correct and valid results in a bounded and predetermined time. RTOSs are widely used in safety-critical domains. In these domains all the system's requirements should be met and a catastrophe occurs if the system fails. Hence, fault tolerance is an essential requirement of RTOSs employed in safety-critical domains. In the past decades, several fault tolerance techniques have been proposed to protect different parts of an RTOS against faults and errors. In this paper, after presenting primary concepts of RTOSs, some features of these operating systems are reviewed and then a number of fault tolerance techniques that can be applied to each feature and their impact on system reliability is investigated. The main contribution of this work is to review and categorize several fault tolerance techniques applicable to RTOSs based on the operating system's features.**

Keywords: **Real-Time Operating System, Fault Tolerance.**

I. INTRODUCTION

"An operating system acts as an intermediary between the user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which the user can execute programs in a convenient and efficient manner"[1]. In fact the main role of an operating system is to employ some methods to manage a computer system, such as scheduling processor(s), process and thread management, inter-process communication, memory management, I/O management, concurrency control, critical sections, synchronization, interrupt and event handling, controlling timers and clocks and etc. which are known as operating systems' features.

In the non-real-time world, the value domain is the sole dimension of computations and correctness of results is the sufficient condition to consider results as valid results. The inclusion of the time domain in real-time systems adds a new dimension to the computations. Real-time applications in addition to correct results, have to produce valid results too. In these applications, correctness is achieved when correct results are produced and validness is achieved when correct results are produced on-time, in a bounded and predetermined time.

Time-sharing operating systems provide an environment to run applications and produce correct results by utilizing resources fairly and efficiently. A typical RTOS monitors and controls some external processes/objects, and it should become aware of changes in the external process/object and respond to them in a timely manner. In order to meet such timing constraints, RTOSs should provide timeliness and predictability by considering real-time requirements while designing operating system's features as mentioned before. In fact, RTOSs should provide both predictability and suitable feature set for application development.

A system is called safety-critical if the occurrence of a failure in meeting system requirements causes to catastrophic effects. In addition to meeting predefined requirements, these systems should satisfy real-time constraints if they want to perform their intended functions effectively [2]. Hence RTOSs are widely used in safety-critical systems. Military and civilian aircrafts, nuclear plants, and medical devices are examples of safety-critical systems.

In safety-critical systems, in addition to hardware, applications and the host operating system ought to be fault-tolerant and their operations should not be failed. In other words, the operating systems employed in safety-critical domains should produce correct and valid results in the presence or in the absence of faults. Such feature is known as reliable computing [3]. Requirement of this reliability is to implement fault tolerance techniques on the operating system [4]. In spite of the efforts made to prevent and remove faults during development phases of safety-critical systems, software faults aren't eliminated yet completely and also the system hardware may still fail during operation because of internal or external faults. Hence, implementing fault tolerance techniques on an RTOS to tolerate faults and errors in a safety-critical system is crucial.

In this paper first some basic concepts of RTOSs are presented and then a number of the most important features of RTOSs are reviewed. Afterward, some fault tolerance techniques applicable to the mentioned features along with their impact on system reliability is investigated. The investigated techniques include both hardware-based and software-based techniques which are employed to tolerate transient and permanent faults.

The organization of this paper is as follows. Section 2 presents some basic concepts and different types of RTOSs. Section 3 investigates a number of RTOSs' features along with some fault tolerance techniques that can be applied to each feature. Finally Section 4 concludes the paper.

II. BASIC CONCEPTS

In this section first some definitions of RTOSs are presented and then three kinds of these operating systems along with their primary requirements are discussed.

A) Real-Time Operating System (RTOS)

"Real-time operating systems emphasize predictability, efficiency and include features to support timing constraints" [5]. In RTOSs all tasks should be released on-time (on *release time*) and also should be completed before particular times called *deadline*. A real-time task fails if it couldn't meet these timing constraints [6]. In other words, violating timing constraints in RTOSs leads to system failure. In order to analyze RTOSs precisely and guaranty system safety, their internal parts should be defined exactly and also their behavior should be predictable.

For example *XOberon* is a small RTOS which provides predictability and safety together [7].

B) Soft, Firm and Hard Deadlines

Deadline is an important property concerned to tasks in RTOSs and is the instant when the results should be produced before it. If a result has utility even after the deadline is passed, the deadline is classified as *soft*, otherwise it is *firm*. If severe consequences could result if a firm deadline is missed, the deadline is called *hard* [8]. In other words violating firm deadlines results in failure and violating hard deadlines results in catastrophe.

Different requirements of hard and soft RTOSs have important effects on the system design. If the system has hard real-time constraints, the designer has to spend a lot of time to guaranty system safety and predictability and also guaranty that all timing constraints (deadlines) are met. If the system timing requirements are soft, a system that has best effort to meeting timing constraints and also has minimum quality loss while violating timing constraints should be designed. Portable media players and online video conferences are examples of soft real-time systems. Examples of hard real-time systems include drive-by-wire systems in automobiles, fly-by-wire systems in avionics, missile control systems and autonomous space systems.

Hard real-time operating systems focus on timing constraints as the most important issue in the system design and don't pay attention to fault tolerance as much as timing constraints. Since occurrence of failure in the RTOS either would causes to produce incorrect results because of wrong computations or would causes to produce invalid results because of missing deadlines, implementing fault tolerance techniques should be considered in the system design. In this paper several primary features of RTOSs along with a number of fault tolerance techniques that could be applied to each feature are presented.

III. RTOSs' Features and Fault Tolerance Techniques

In the previous sections, the importance of implementing fault tolerance techniques on RTOSs, especially those that are employed in safety-critical domains was discussed. In this section, a number of RTOSs' features along with some fault tolerance techniques that could be applied to each feature are presented.

A) Memory Management

In order to protect operating systems' components prone to failure, fault tolerance begins with memory protection. Sine programs behavior depends to data in memory, the existence of faults in these data would cause to program error and failure.

Since the flexibility and functionality of applications are being increased and also they need dynamic access to memories, *dynamic storage allocation (DSA)* algorithms play an important role in the operating systems. In addition to flexibility, real-time applications require predictability too, i.e. memory should be managed dynamically in a bounded and predetermined time. The use of DSA leads to uncertainty in RTOSs, because of the unconstrained response time of DSA algorithms and the fragmentation problem. In [9] a DSA algorithm called *TLSF* has been developed to be employed in RTOSs. *TLSF* provides explicit allocation and de-allocation of memory blocks with a bounded and acceptable timing behavior $\Theta(1)$. Using bitmaps and the aid bitmaps is another technique to make allocating and de-allocating memory safely and reliably. This technique was introduced by [10] to be employed in *RTEMS* RTOS.

Operating systems use *memory management units (MMU)* to run tasks in their protected memory address. Nevertheless some RTOSs disable MMU and don't use it [9]. OSEK-VDX, µITRON and RTAI are examples of such RTOSs that disable MMU [11]. By disabling MMU the operating system and all processes are run in the same address space and each task has access to operating system's and other processes' codes and data. Hence a bad written code or a bug in a code, for example in managing pointers, would cause to failure in the kernel, resulting in the operating system crash. Without memory address protection, also some bugs would cause to special corruptions that are difficult to detect. For example in PowerPC processors, RAM is often located at physical address 0, so even a NULL pointer dereference may not be detected [12]. In order to prevent such failures, RTOSs must use MMU. By enabling MMU, whenever the stack of a task overflows, an overflow exception is raised and the operating system stops the task execution. Instead of stopping the task execution, the operating system can suspend the task and solve the problem of stack limitation by migrating the overflowed task to a new memory address space with a larger capacity, by regarding reserved and unreserved spaces and then re-executing the suspended task. The RTOS designer should take the migration time into account when analyzing system.

Redundancy is one of the most important techniques in fault tolerance [3]. This technique can be applied to memory in a way that when a process is loaded, the operating system duplicates its data and states in more than one place/memory (three places/memories to imitate TMR). Whenever a task's data/states are changed, these changes are applied to all replicas. Whenever the task wants to read data from memory, a voting is done on replicas to determine if data are changed inadvertently or are corrupted (for any reason, such as heavy ion radiation) and also to determine which data is correct and could be used. Memory redundancy could be supported in both software level and hardware level [13].

In addition to redundancy, a fault-tolerant memory management system could be constructed by four concurrent mechanisms as: a first recording mechanism that is activated to record memory update (write) events, a second recording mechanism that records at least a limited number of memory update events, an activator to activate the first recording mechanism in the event of a fault event and a memory reintegration mechanism that is utilized to data recovery by reintegrating some parts of memory [14]. In this memory management system, error recovery can be done rapidly and efficiently by reintegrating memory pages identified in the first and second recording mechanisms by considering memory updates log.

Error-correcting code memory (ECC memory) is an instrument to improve operating systems reliability from the memory protection perspective. ECC memory is a type of computer data storage that has ability to detect and correct many kinds of internal data corruption. This memory is resistant to single-bit errors: the data that is read from each word is always the same as the data that has been written to it, even if a single bit has been flipped to the wrong state [15]. Some non-ECC memories with parity support allows errors to be detected, but not corrected. The reliability of a fault-tolerant RTOS would be improved by employing this kind of memory. In contrast to these hardware-based techniques, software-based memory error detection and correction methods such as [16] would provide both reliability and flexibility.

B) Kernel Considerations

Error detection could be done by hardware or software methods, such as *"Transient fault detection via simultaneous multithreading"* [17] which is an example of software methods. The kernel of a fault-tolerant RTOS should provide a mechanism that whenever an error occurs, a notification is sent to an agent that has duty to perform some types of error recovery actions. This agent is called *supervisor* and must be run in an isolated address space, because data in the address space containing faulty task may be corrupted. For example in Nooks which is a reliability subsystem, *Nooks Recovery Manager* is an agent for error recovery [18]. *VxWorks* RTOS employs a tree structure to manage error notifications produced by OS's components by higher-level components in the hierarchical tree [19]. The supervisor would recover the faulty task by using backward- or forward- recovery or by re-starting it from the beginning. The selected recovery strategy should be considered and defined in the system analysis.

The kernel also has to provide an event logging mechanism to determine the root of an explicit error by analyzing everything that has been happened in the system, such as kernel service calls, task context switches and interrupts, prior to the fault [12]. To detect implicit errors, the kernel should provide a software watchdog capability to be notified whenever a task is not run in its expected code sequence or time slices. This mechanism also is useful in the control flow checking technique [20]. For example *QNX* RTOS uses *Critical Process Monitor (CPM)* module and *VxWorks* RTOS uses *Failover Management System (FMS)* module to detect malfunctioning system's components.

As a technique for error prevention, fault-tolerant RTOSs should protect themselves against improper invoking system calls and passing invalid parameters. For example some RTOSs send an actual pointer of kernel objects (e.g. semaphores) to tasks and then dereference this pointer when changed and passed into other kernel service calls made by the tasks. In this sequence if a task after receiving a pointer fails, this failure would cause to pointer corruption and as a result passing the corrupted pointer to the kernel and using it by the RTOS may leads to the RTOS crash [12]. To make this kind of failures impossible, RTOSs' kernel must validate the parameters sent to all service calls. This validation could be done by employing descriptors for application's references to kernel objects or by using n-copy programming technique [3].

Availability is an important part of dependable computing which can be achieved by providing replications of operating nodes. These replicas are operated concurrently and their internal data and states are synchronous and equivalent. OSs would detect nodes failure by sending and receiving heartbeat message to and from active nodes via reliable channels. When the heartbeat message fails to arrive, the active node is discarded and one of the redundant nodes is tagged as active node and then it can be taken into processing. Figure 1 depicts this scenario. In RTOSs it's preferable to use *Active Replication* instead of *Passive Replication* when using redundancy techniques [8].

Fault-tolerant RTOS also should prevent the spread of faults to the kernel. This goal can be achieved by reducing the size of the kernel by keeping fundamental services inside the kernel and excluding others, especially those that are prone to errors, such as drivers [21]. *VxWorks* RTOS provides such isolation by inserting protection boundaries between different components [19].

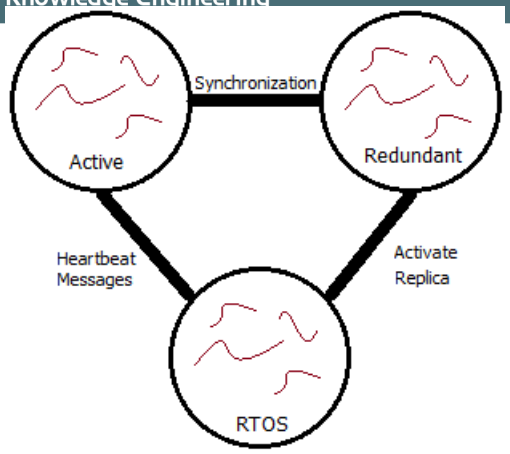

Figure 1 - Redundancy in Operating Nodes

C) Process and Thread Management

Similar to time-sharing OSs, process definition and activation is one of the most important roles of RTOSs. But, there are some differences between these two kinds of OS in managing processes because of timing constraints in RTOSs. Time-sharing OSs do their best effort to activate and release tasks timely. But in contrast, RTOSs should activate a process once and release it once or periodically and also guaranty each release is started on-time and is finished before its deadline. In order to adhere these timing constraints, an RTOS must guaranty the availability of the processes' required resources.

If tasks' behavior is not monitored and controlled by the RTOS, a task may, as a result of malicious or careless execution of another task, cannot use processor or other system resources. When a task creates a new task or another kernel object, the kernel allocates some system resource, especially a chunk of memory and CPU time to this new task. A bug or a fault in the application would cause a situation where this task creates too many other tasks or kernel objects and exhausts system resources. As a result other tasks may fail because of their inability in acquiring required resources and resulting in deadline miss. In a fault-tolerant RTOS, a mechanism must exist to prevent such failures caused by resources shortage. One possible solution is to determine the maximum required resources, especially memory space and CPU time by processes before the execution, so the RTOS can reserve required resources for each process and as result none of processes are stopped because of resources shortage. In this method none of processes can acquire more than reserved resources and if they want to use more than their quota, this act is regarded as an error and should be discarded. Since in systems with static tasks, the attributes of all tasks are known in advance, a more relax approach can be chosen in resource allocation in a way that the RTOS allocates resources to each process from its reserved resources and from the remaining resources that are free and also aren't reserved by other processes. For example a framework called *RRES* has been introduced by [22] for resource reservation that with a little coding improves system reliability significantly.

In fixed-priority systems, tasks' priority would be changed incorrectly because of fault occurrence in process table. A possible technique to solve this problem is to acquaint process manager with the importance of the tasks (e.g. hard RT task versus soft RT task or critical task versus normal tasks) by using partitions in the memory. The concept of partition process management is a major part of ARINC Specification 653, an Avionics Application Software Standard Interface [23]. The ARINC 653 partition process manager runs partitions, or address spaces, according to a timeline provided by the system designer. Each address space is placed into one

or more windows of execution in a hyper period. During each window, all tasks in other address spaces cannot be run, and only tasks within the currently active address space are selected by process manager to be run. When the hard/critical processes' window is active, its processing resource is guaranteed and soft/normal processes cannot be run and take away processing time from the hard/critical process. An implementation of ARINC 653 in *RTEMS* RTOS has been addressed in [24].

D) Scheduling

Scheduler is the heart of an RTOS. In fact in order to guaranty system safety, the scheduler by considering tasks attributes have to determine what task should be released and should be preempted at what times. There are different scheduling algorithms in RTOSs. The most important of them are as follows [25]:

- **RM**: *Rate Monotonic (RM)* is a fixed-priority scheduling algorithm which tasks priority is defined in advance and tasks with smaller period have higher priority.
- **EDF**: *Earliest Deadline First (EDF)* is a dynamic scheduling algorithm which tasks priority is defined dynamically in run-time in a way that tasks with closer deadline have higher priority.
- **LLS**: similar to EDF, *Least Laxity First (LLF)* is a dynamic scheduling algorithm. It assigns priority based on the slack time of a process. Slack time is the amount of time left after a job if the job was started now. In LLS processes with smaller slack time have higher priority [26].

Scheduler as the most important task of an RTOS has to be protected against failures. If the scheduler fails, other system tasks are not scheduled and released correctly and as result the system crashes. If the scheduler is fixed-priority, its misbehavior can be detected by using a pre-constructed static scheduling table and comparing the output of the scheduler with this table. This table has to be constructed for a hyper period. N-copy programming (NCP) is another fault tolerance technique that can be employed for both fixed-priority and dynamic scheduling algorithms. With this technique, n copies of a scheduler ($n \geq 3$) are run concurrently in different address spaces. Then the right scheduling can be done by taking votes of these replicas.

In addition to be fault-tolerant, the scheduler should take the required time to handle faulty tasks into its time analyses. As it was mentioned before, a faulty task can be re-executed from the beginning or can be restored from the last checkpoint prior to fault. This recovery and re-executing of faulty tasks, wastes time and could violate timing constraints. In order to guaranty system safety in the presence of failures, fault-tolerant RTOSs must consider these wasting times in system analysis and scheduler design. In system analysis it should be explicitly determined for each task at-most how many re-execution is possible, if task recovery is done by re-starting it from the beginning, and in situation of using backward recovery as fault tolerance scheme, by having fault rate, error detection latency and required time for saving and restoring checkpoints data, at-most how many failures could be recovered and also how many checkpoints has to be taken to do that, [27, 28]. This analysis can be done statically in advance or dynamically in run-time. RTOS more tend to use *g-state* instead of *checkpoint* [8].

In addition to recovering tasks from errors, fault-tolerant RTOSs should be able to recover processors from transient and permanent faults too. If a processor fails temporarily and its internal states and assigned tasks are not recovered, this failure leads to violate timing constraints and system crash. By sending heartbeat messages, a processor failure could be detected and by having checkpoints of the entire processor states and the assigned tasks, in disks, the faulty processor would be recovered from transient faults correctly [29]. In situation of permanent faults, after recovering faulty processor states, task migration must be done to run recovered tasks on another hale processor. Also tolerating more than one faulty processor preferably must be taken into account while system design [30]. Since disks have low speed, using diskless checkpointing schemas and storing checkpoints data on other processors' memory would help to decrease waste times [31].

A research on implementing a fault tolerance scheduler in *RTEMS* RTOS has been presented in [32]. In addition to fault tolerance, energy management and dynamic voltage scaling issues could be considered in time analysis especially for embedded systems [33].

E) Communications

In all operating systems, processes need to communicate with each other through some mechanisms, such as message passing or memory sharing. Message passing methods causes to uncertainty in the system timing, because of systems architecture features, i.e. it's impossible to determine exactly how long a message passing takes. In an RTOS the maximum latency of message passing should be determined. To achieve such determinacy some token based techniques such as Ring and TDMA can be employed [34]. Moreover if the reliability of communication channels is not 100%, some techniques such as dynamic time redundancy in the lower levels of the communication protocols or using QoS services could be employed to increase the communication channels reliability significantly [34].

In addition to physical and dynamic time redundancy, there are other approaches and methodologies to increase the reliability of inter-processes communications. For example in [35] a facility has been introduced that provides supports for fault-tolerant process groups by a family of reliable multicast protocols that can be employed in fault-tolerant RTOSs. In this facility a protocol that guaranties delivery orderings has been introduced which ensures the processes belonging to a fault-tolerant process group will see consistent orderings of events that affect the system reliability including process failures, recoveries, migration, and dynamic changes to group properties like member rankings. This is done by using some broadcast primitives, such as: group broadcast (GBCAST), atomic broadcast (ABCAST) and causal broadcast (CBCAST).

Similar to inter-process communications, inter-processors communications should be reliable too. In [36] a subsystem called *Transis* has been introduced that by using *reliable multicast message services*, supports reliable communication among processors. *VxFusion* is a run-time extension to support inter-processor communication that is employed by *VxWorks* RTOS [19]. In addition to the mentioned mechanisms, there are different channel models that by using appropriate encoders and decoders, guaranty cannels reliability [37]. In order to select a model for a communication, several factors must be considered. These factors include the physical and statistical nature of the channel disturbances, the information available to the transmitter and receiver, the presence of any feedback link from the receiver to the transmitter, and the availability of the transmitter and receiver of a shared source (independent of the channel disturbances) [37].

RPC is a remote communication method which in order to meet requirements of fault-tolerant RTOSs, should be done in a reliable manner. *Sun Batching RPC* is a variation of RPC

that performs reliable and dependable telecommunications. It typically uses reliable byte stream protocols (like TCP) for its transport [38] which in addition to guaranty reliable communication, guaranties at-most-once semantics and ordered delivery of messages. These features qualify *Sun Batching RPC* to be employed by fault-tolerant RTOSs.

The implementation of a fault-tolerant RPC based grid applications was discussed in [39].

F) I/O Management

RTOSs should manage the order of I/O accesses in a way that interferences are prevented and also all tasks (especially hard real-time tasks) could meet their timing constraints. In addition to considering timing constraints, fault-tolerance RTOSs must provide some fault tolerance techniques to tolerate faulty I/O devices. There are many fault tolerance techniques for I/O devices that are concerned to the target device. Replication is the most common technique that can be employed by duplicating I/O devices. The main I/O device is called *active (primary)* and the replicated ones are called *backup*. Once an active device fails and its failure is detected by heartbeat messages, one of the backup devices must perform the duties of active device from the fault point. Such duplications via PCIs have been investigated in [40]. In order to mitigate wasting times, it's desired to design backups as *active redundancy*. For example RAID is an example of active redundancy in secondary storage devices [41].

Robustness is an important system quality feature which is defined by the IEEE standard glossary of software engineering terminology as: *"The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions"* [42]. Avizienis et al. also define robustness as *"dependability with respect to erroneous input"* [43]. When the input data are missed or incorrect, robustness techniques try to fix or calculate the exact or approximate value of the input data. One technique in robustness is to request correct data from the sender or user by considering correct data format which has been defined in a predefined data pattern table. Another technique is to use last correct data instead of the missed/incorrect input data or to approximate correct value of the input data by applying some machine learning algorithms to previous input data in similar situations. Such techniques don't guaranty a correct behavior in the system, but they would alleviate the side effects of data loss.

The robustness of an OS would be measured by the ability of its APIs in handling exceptional input parameters which consists of detecting invalid parameters and tolerating them [44]. Experiments on 233 functions of 13 POSIX OSs reveals a 6% to 19% robustness failure rate for single-OS tests that by employing N-version technique this rate was reduced to 3.8%.

The desired robustness model should be selected while the system development. Based on study in [45], almost 47% of researches consider robustness in *verification & validation* phase of the system development and only 35% of researches consider it in the *system design* phase. Other researches do it in different phases.

G) Interrupt Handling

OSs have several types of interrupts with different priorities and execution times. Interrupts with higher priority need a faster response time. When internal data structures are being manipulated by and during service calls, other interrupts especially timer's scheduler should be disabled because otherwise a related service call may be executed and cause an access to inconsistent data. In fact in order to handle lower priority interrupts reliably, the higher priority interrupts are discarded or hindered unboundedly which cause to indeterminacy in timing system calls that is undesired for RTOSs [12]. A static analysis approach to obtain the WCET of system calls in *RTEMS* RTOS has been introduced in [46].

Fault-tolerant RTOSs should guaranty both predictability and reliability while handling interrupts. To achieve these goals, all kernel service calls should be revertible, so that the RTOS can preempt the service call, restore carried out operations and restart it later. Therefore the time to get back to the scheduler may take by a few instructions and the higher priority interrupts are always executed with an absolute minimum latency. This method improves system predictability and reliability in terms of avoidance of access to inconsistent data.

H) Programming Languages

Since fault-tolerant RTOSs have special requirements, in order to meet them special programming languages should be employed as well. Some features of traditional programming languages are prone to problems that using them in fault-tolerant RTOSs is discouraged, such as: pointers, dynamic memory allocation and de-allocation, unstructured programming, multiple entry points and exit points, variant data, implicit declaration and implicit initialization, procedural parameters, recursion, concurrency and race, and interrupts aware programming [47]. In addition to considering these programming features, real-time applications has to guaranty correct responses within strict timing constraints. In other words the maximum required time to respond to a request or to complete a work by a process should be accountable. To attain this time, the maximum time that each part of a program takes should be determined explicitly. Hence for example in the real-time programming, variable loops with undetermined or unbounded iteration are not acceptable.

In general, real-time programming languages are employed in three real-time programming models as *synchronous*, *scheduled*, and *timed* that differ in time they take to complete and in their compiler to meet corresponding requirements [48]. In addition to considering these models, programming languages employed in fault-tolerant RTOSs should support some error detection and error correction techniques. *Ada* is one of the most widely used programming language in fault-tolerant real-time domains because of its major strengths, such as: the well-defined language semantics, the strong type checking, structuring mechanisms like packages and supporting the development of code analysis, verification and testing tools [49]. *Euclid* is another fault-tolerant real-time programming language that employs exception handlers and import/export lists to provide comprehensive error detection, isolation, and recovery. The philosophy of this language is that every exception detectable by the hardware or the software should have an exception handler associated with it. Moreover, *Euclid* forces everything in the language to be time- and space-bounded [50]. Using such programming languages would cause to improve the system reliability [51].

IV. Conclusion

Real-time operating systems are widely used in safety-critical domains to interact with controlled objects in the external environment and should provide correct and valid results in a bounded and predetermined time. In these domains, the costs of a system failure leads to catastrophe and exceeds the initial investment in the computer and in the controlled object. To prevent such failures, system designer must guaranty that the system can meet requirements as specified in the domains of

both value and time during all anticipated operational situations, even when an error occurs. To attain this goal, the employed RTOS should be able to tolerate faults and errors appear in the system.

Similar to traditional operating systems, RTOSs have some primitive features that are essential for a basic RTOSs to meet value and time domains requirements. Implementing fault tolerance techniques on these features would cause to improve the reliability of the RTOS and the whole system as well.

In this paper first some definitions of RTOSs along with their requirements was reviewed followed by investigating some primitive features of an RTOS such as *Memory Management*, *Kernel Considerations*, *Process and Thread Management*, *Communications*, *I/O Management*, *Interrupt Handling* and *Programming Languages*. Then a number of fault tolerance techniques that could be applied to each mentioned features were presented. This paper in fact categorizes several fault tolerance techniques applicable to RTOSs based on some primitive features of operating systems. Some techniques could only deal with transient faults and some could tolerate both transient and permanent faults. Some techniques are only software-based and some rely on the involved hardware. In order to have a fault-tolerant RTOS, the system designer has to consider the requirements of the intended fault tolerance techniques in the *requirement analysis* and *system design* phases while developing system.

V. References

[1] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating system concepts*: J. Wiley & Sons, 2009.
[2] J. S. Ostroff, "Formal methods for the specification and design of real-time safety critical systems," *Journal of Systems and Software*, vol. 18, pp. 33-60, 1992.
[3] L. L. Pullum, *Software fault tolerance techniques and implementation*: Artech House Publishers, 2001.
[4] P. J. Denning, "Fault tolerant operating systems," *ACM Computing Surveys (CSUR)*, vol. 8, pp. 359-389, 1976.
[5] J. A. Stankovic and R. Rajkumar, "Real-time operating systems," *Real-Time Systems*, vol. 28, pp. 237-253, 2004.
[6] P. A. Laplante, "Real-Time Systems Design and Analysis," 1993.
[7] R. Brega, "A real-time operating system designed for predictability and run-time safety," in *Proceedings of The Fourth International Conference on Motion and Vibration Control (MOVIC)*, 1998, pp. 379-384.
[8] H. Kopetz, *Real-time systems: design principles for distributed embedded applications* vol. 25: Springer, 2011.
[9] M. Masmano, I. Ripoll, A. Crespo, and J. Real, "TLSF: A new dynamic memory allocator for real-time systems," in *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, 2004, pp. 79-88.
[10] H. Li and C. Yin, "Analysis and Improvement of RTEMS Memory Management," in *Education Technology and Computer Science, 2009. ETCS'09. First International Workshop on*, 2009, pp. 107-111.
[11] R. Yerraballi, "Real-time operating systems: An ongoing review," in *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS'2000), WIP Section, Orlando Fl*, 2000.
[12] David Kleidermacher and M. Griglock, "Real-time Operating System Requirements for Use in Safety Critical Systems," Green Hills Software, Inc2001.
[13] K. S. Gray, "Memory redundancy techniques," ed: Google Patents, 2002.
[14] E. J. Williams, "Memory management in fault tolerant computer systems," ed: EP Patent 0,817,053, 2003.
[15] F. Qin, S. Lu, and Y. Zhou, "Safemem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, 2005, pp. 291-302.
[16] C. Borchert, H. Schirmeier, and O. Spinczyk, "Generative software-based memory error detection and correction for operating system data structures," in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, 2013, pp. 1-12.
[17] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *ACM SIGARCH Computer Architecture News*, 2000, pp. 25-36.
[18] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 23, pp. 77-110, 2005.
[19] A. K. Sood, "Digging Inside the VxWorks OS and Firmware (The Holistic Security)," SecNiche Security Labs.
[20] I. Koren and C. M. Krishna, *Fault-tolerant systems*: Morgan Kaufmann, 2010.
[21] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Construction of a highly dependable operating system," in *Dependable Computing Conference, 2006. EDCC'06. Sixth European*, 2006, pp. 3-12.
[22] A. Mancina, D. Faggioli, G. Lipari, J. N. Herder, B. Gras, and A. S. Tanenbaum, "Enhancing a dependable multiserver operating system with temporal protection via resource reservations," *Real-Time Systems*, vol. 43, pp. 177-210, 2009.
[23] A. Specification, "653," *Avionics Application Software Interface, Annapolis, MD*, 1997.
[24] J. Rufino, S. Filipe, M. Coutinho, S. Santos, and J. Windsor, "ARINC 653 interface in RTEMS," in *Proc. DASIA*, 2007.
[25] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, pp. 46-61, 1973.
[26] M. L. Dertouzos and A. K. Mok, "Multiprocessor online scheduling of hard-real-time tasks," *Software Engineering, IEEE Transactions on*, vol. 15, pp. 1497-1506, 1989.
[27] Y. Zhang and K. Chakrabarty, "A unified approach for fault tolerance and dynamic power management in fixed-priority real-time embedded systems," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 25, pp. 111-125, 2006.
[28] I. J. Bate, "Scheduling and timing analysis for safety critical real-time systems," Ph.D., University of York Department Of Computer Science-Publications-Ycst, 1999.
[29] G. Bournoutian and A. Orailoglu, "Dynamic transient fault detection and recovery for embedded processor datapaths," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2012, pp. 43-52.
[30] X. Ping and Z. Xingshe, "Security-Driven Fault Tolerant Scheduling Algorithm for High Dependable Distributed Real-Time System," in *Parallel Architectures, Algorithms and Programming (PAAP), 2011 Fourth International Symposium on*, 2011, pp. 29-33.
[31] G.-M. Chiu and J.-F. Chiu, "A new diskless checkpointing approach for multiple processor failures," *Dependable and Secure Computing, IEEE Transactions on*, vol. 8, pp. 481-493, 2011.
[32] B. Zhang, X. Xu, and B. Li, "Research on the design of software fault tolerance based on RTEMS," in *Computer, Mechatronics, Control and Electronic Engineering (CMCE), 2010 International Conference on*, 2010, pp. 402-405.
[33] T. Wei, P. Mishra, K. Wu, and J. Zhou, "Quasi-static fault-tolerant scheduling schemes for energy-efficient hard real-time systems," *Journal of Systems and Software*, vol. 85, pp. 1386-1399, 2012.
[34] A. S. Tanenbaum, *Modern operating systems* vol. 2, 1992.
[35] K. P. Birman and T. A. Joseph, "Reliable communication in the presence of failures," *ACM Transactions on Computer Systems (TOCS)*, vol. 5, pp. 47-76, 1987.
[36] Y. Amir, D. Dolev, S. Kramer, and D. Malki, "Transis: A communication subsystem for high availability," in *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, 1992, pp. 76-84.
[37] A. Lapidoth and P. Narayan, "Reliable communication under channel uncertainty," *Information Theory, IEEE Transactions on*, vol. 44, pp. 2148-2177, 1998.
[38] R. Thurlow, "RPC: Remote procedure call protocol specification version 2," 2009.
[39] Y. Tanimura, T. Ikegami, H. Nakada, Y. Tanaka, and S. Sekiguchi, "Implementation of fault-tolerant GridRPC applications," *Journal of Grid Computing*, vol. 4, pp. 145-157, 2006.
[40] S. L. Blinick, J. C. Elliott, and E. Q. Garcia, "Redundant and fault tolerant control of an I/O enclosure by multiple hosts," ed: Google Patents, 2011.
[41] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: High-performance, reliable secondary storage," *ACM Computing Surveys (CSUR)*, vol. 26, pp. 145-185, 1994.
[42] J. Radatz, A. Geraci, and F. Katki, "IEEE standard glossary of software engineering terminology," *IEEE Std*, vol. 610121990, p. 121990, 1990.
[43] A. Avizienis, J.-C. Laprie, and B. Randell, *Fundamental concepts of dependability*: University of Newcastle upon Tyne, Computing Science, 2001.
[44] P. Koopman and J. DeVale, "Comparing the robustness of POSIX operating systems," in *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, 1999, pp. 30-37.
[45] A. Shahrokni and R. Feldt, "A systematic review of software robustness," *Information and Software Technology*, 2012.
[46] A. Colin and I. Puaut, "Worst-case execution time analysis of the RTEMS real-time operating system," in *Real-Time Systems, 13th Euromicro Conference on, 2001.*, 2001, pp. 191-198.
[47] I. I. P. Ltd., "An Introduction to Safety Critical Systems," 1997.
[48] C. M. Kirsch, "Principles of real-time programming," in *Embedded Software*, 2002, pp. 61-75.
[49] T. S. Taft and R. A. Duff, *Ada 95 Reference Manual. Language and Standard Libraries: International Standard ISO/IEC 8652: 1995 (E)* vol. 1246: Springer, 1997.
[50] E. Kligerman and A. D. Stoyenko, "Real-time Euclid: A language for reliable real-time systems," *Software Engineering, IEEE Transactions on*, pp. 941-949, 1986.
[51] V. Barr and S. Montenegro, "BOSS/Ada: An Open Source Ada 95 Safety Kit A Dependable open source embedded operating system for GNAT," *Ada Deutschland Tagung*, pp. 53-66, 2002.