MECHANISMS FOR RELIABLE DISTRIBUTED REAL-TIME OPERATING SYSTEMS pdf

1: On Distributed Embedded Systems

Buy or Rent Mechanisms for Reliable Distributed Real-Time Operating Systems as an eTextbook and get instant access. With VitalSource, you can save up to 80% compared to print.

The network is homogeneous Real-Time A real-time system is a system in which the timeliness of operation completion is a part of the functional requirements and correctness measure of the system. I opened an SO question here to try to clarify this. We reserve the real-time term, sometimes qualified by soft or hard, for systems which are incorrect when time constraints are not met. Note that many of the concerns summarized in the fallacies above intersect with timeliness. See also the real-time tag wiki It is useful to note that RT and DRT systems exist on a continuum of requirements, with "deterministic" or conventionally, hard real-time at one extreme. However, plenty of systems have very important time constraints which are nevertheless non-deterministic. Especially in the context of DRT systems, it is also useful to separate the concept of activity urgency from activity priority. In large systems where latency and failure are real and non-trivial factors, the explicit management of computing and communication resources to effect timeliness and other design requirements becomes more important, and the separation of these two dimensions becomes important. Composing Distributed with Real-Time Explicit timeliness requirements â€" What are the requirements, how are they mapped to activities, are they true trans-node timeliness requirements, how will the time constraints be represented explicitly in the design and implementations, and how will failures be detected, reported, and recovered? Time synchronization â€" What are the requirements for and mechanisms for achieving clock synchrony? Wiki on clock synchronization ; many applications require only NTP ; more stringent requirements may necessitate special hardware e. Synchrony requirements â€" What are the synchrony assumptions constraining and requirements for system synchrony? This is connected to clock synchrony, but not identical. Some thoughts on formal models from Doug Jensen ; wikipedia on Asynchronous System and Synchronous ; SO question on partial event ordering ; Design patterns â€" What are the moving parts, and how do they relate over the transport? In particular, how do these relationships affect timeliness? Middleware â€" How are you going to encode the distributed aspects of the system? Time Constraints â€" How are you going to document, measure, and enforce time constraints in the system? Partial Failure â€" A real-time system typically has reliability requirements. The seminal paper on the topic is On the impact of fast failure detectors on real-time fault-tolerant systems , by Aguilera, Le Lann, and Toueg. This paper is heavy sledding, but rewards every ounce of intellectual investment.

MECHANISMS FOR RELIABLE DISTRIBUTED REAL-TIME OPERATING SYSTEMS pdf

2: TenAsys INtime RTOS - scalable, hard real-time, dynamic, determanistic OS for PC

Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel deals with the Alpha kernel, a set of mechanisms that support the construction of reliable, modular, decentralized operating systems for real-time control applications.

The concepts of real-time remote and distributed real-time remote interfaces are introduced in order to facilitate the design and implementation of real-time and distributed The concepts of real-time remote and distributed real-time remote interfaces are introduced in order to facilitate the design and implementation of real-time and distributed real-time threads that call remote objects. Wellings - Computer Systems Science and Engineering 8 3 , " The incorporation of unbounded components i. We present an architectural model that caters for the thre We present an architectural model that caters for the three main approaches to integrating unbounded components imprecise computation, sieve functions and multiple versions. This architectural model is feasible because it is supported by schedulability tests that will guarantee the bounded tasks. These test are defined in the paper. A simple computational model that uses preemptive priority-based dispatching is required. The wide-spread use of techniques such as imprecise computation will only happen if they are integrated into standard software engineering methods. We therefore show how the techniques can be realised in Ada 9X. Show Context Citation Context We are unaware of anyone who has combined preemptive priority based scheduling using time attributes such as deadline or period with a time-value scheduling of non-crucial tasks. We consider the problem of recovering from failures of distributable threads with assured timeliness. When a node hosting a portion of a distributable thread fails, it causes orphansâ€”i. We consider a termination model for recovering fr We consider a termination model for recovering from such failures, where the orphans must be detected and aborted, and failure-exception notification must be delivered to the farthest, contiguous surviving thread segment for resuming thread execution. We show that AUA and TP-TR bound the orphan cleanup and recovery time, thereby bounding thread starvation durations, and maximize the total thread accrued timeliness utility. A distributable thread is a single thread of execution with a globally unique identifier that transparently extends and retracts through local and This paper presents a mechanism for modeling timing, precedence, and data-consistency constraints on concurrently executing processes. The model allows durations and intervals between events to be specified. An algorithm is provided to detect schedules which may be unsafe with respect to the constra An algorithm is provided to detect schedules which may be unsafe with respect to the constraints. This work, motivated by the design and validation of autonomous error-recovery strategies on the Galileo spacecraft, appears to be applicable to a variety of asynchronous real-time systems. These autonomous processes are constrained at the event level by timing, precedence, and data-consistency rules. A schedule ordering of events that violates these constraints can jeopardize the spacecraft and is labeled unsafe. Safety involves those correctness properties required by the static portions of the specification The following assumptions hold: Each process has a correct initial state, runs to completion, and produces correct results if executed alone. These assumptions exclude from consideration those ca On multiprocessor utility accrual real-time scheduling with statistical timing assurances by Hyeonjoong Cho, Haisang Wu, Binoy Ravindran, E. We establish several properties of gMUA including optimal total utility for a special case , conditions under which individual activity utility lower bounds are satisfied, a lower bound on system-wide total accrued utility, and bounded sensitivity for assurances to variations in execution time demand estimates. Under conditions 1 and 2 , gMUA always completes the allo

MECHANISMS FOR RELIABLE DISTRIBUTED REAL-TIME OPERATING SYSTEMS pdf

3: What are the essentials of real-time distributed systems? - Stack Overflow

Reviewer: Charles N. Schroeder This book describes a set of kernel-level mechanisms that support the construction of modular, reliable, decentralized operating systems for real-time control applications.

Erick Redwine and JoAnne L. Holliday Santa Clara University Making a distributed system reliable is very important. The failure of a distributed system can result in anything from easily repairable errors to catastrophic meltdowns. A reliable distributed system is designed to be as fault tolerant as possible. Fault tolerance deals with making the system function in the presence of faults see Fault-Tolerant Systems. Faults can occur in any of the components of a distributed system. This article gives a brief overview of the different types of faults in a system and some of their solutions. Component Faults There are three types of component faults: A transient fault occurs once and then disappears. If the operation is repeated then the system will behave normally. An intermittent fault arises, then goes away, rises again and so on. A common cause of an intermittent fault is a loose contact on a connector. These faults are very annoying and hard to fix because of their sporadic nature. Lastly there are permanent faults caused by faulty components. The system will not work until the component is replaced. Burnt out chips, software bugs and processor failure explored in Processor Faults are all examples of permanent faults. Processor Faults A special type of component is the processor, and it can fail in three ways, fail-silent, Byzantine and slowdown. All lead to a different kind of failure. A fail-silent, also known as fail-stop, fault occurs when a processor stops functioning. It no longer accepts input and outputs nothing, except perhaps to say it is no longer functioning. Byzantine faults occur when a faulty processor continues to run, giving wrong answers and maybe working with other faulty processors to give the impression that they are working correctly. Compared with fail-silent faults, Byzantine faults are hard to diagnose and more difficult to deal with. A slowdown fault occurs when a certain processor executes very slowly. That processor might be labeled as failed by the other processors. Network Failures Network failures keep processors from communicating with each other. We will look at the failures resulting in total loss of communication along parts of the network. Two problems arise from this: One-way links cause problems similar to processor slowdown see Processor Faults. For example, processor A can send messages to processor B but cannot receive messages from B. Processor C can talk to both A and B. So each processor has a different idea of which processors have failed. Processor A might think that B has failed since it does not receive any messages from it. Processor C thinks both A and B are working properly since it can send and receive messages from both. Network partitions occur when a line connecting two sections of a network fail. So processors A and B can communicate with each other but cannot communicate with processors C and D and visa versa. Let us say processor A updates a file and processor C updates the same file but in a different way. When the partition is fixed, the file will not be consistent among all processors. It is not clear to the processors how to make the data consistent again. Agreement To be reliable, a distributed system must be able to reach agreement see Consensus and Byzantine Agreement , even if the system is faulty. This means all processors in a distributed system must agree on some value. Two cases must be explored to understand the problem of agreement. The first involves perfect processors but faulty communication lines, and the second problem looks at Byzantine processors. For very similar analogies, refer to Tanenbaum [2]. Two processors cannot be in agreement if the communication lines between them are faulty. The perfect example of this is the well-known, two-army problem illustrating the trouble of getting two perfect processors with faulty communication between them to agree on one bit of information. Two allied armies, led by General Bonaparte and General Alexander, are encamped on opposite sides of the enemy army. If both armies attack together then they will emerge victorious, but if they attack at different times they will be defeated. Their only mode of communication is a messenger who is subject to capture by the enemy army. See you at dawn. Both armies ready themselves for attack. Later that day, Bonaparte realizes that Alexander does not know if the messenger made it back safely. So Bonaparte sends the messenger back across enemy lines to deliver a message of

confirmation to Alexander. Upon receiving this message of confirmation, Alexander sends a message back to tell Bonaparte that he received the message and that the attack is still on. No matter how many messages are sent, neither General will attack because both Generals are unsure if the messages were received. The next problem is more complex. It mimics the case of perfect communication lines and faulty processors and is commonly called the Byzantine generals problem. In this problem, n allied generals surround the enemy, but m of the generals are traitors faulty processors. The traitors try to prevent the loyal generals from reaching agreement by giving them contradictory and false information. Unlike the two-army problem, the loyal generals can reach agreement. Agreement in this situation is defined as the generals exchanging the troop strengths of their armies. Each army sends a message to all other armies, telling them of their troop strengths. Then the generals send their finding on to all the other generals. A Byzantine general will send faulty information to all generals but those faulty signals will be picked up and the loyal generals will know the traitor is Byzantine. For more information on this, refer to the article on Byzantine Agreement. For algorithms to handle Byzantine faults refer to chapter 11 of Chow [1]. Solutions To System Failures Before we explore some of the common solutions to system failures, we must learn the difference between synchronous and asynchronous systems. In a synchronous system the amount of time required for a message to be sent from one system to another has a known upper bound. Therefore, processor A sends a message to processor B and waits a given time for a response. If A does not receive a response within that time, it knows an error has occurred and it will send the message again. After a set number of resends, B is labeled as failed. In an asynchronous system, none of this is true. A processor will wait an infinite time for a response from the other processor. Many solutions for fault tolerance cannot be implemented in an asynchronous system. A processor experiencing slowdown is impossible to differentiate from a dead processor see Impossibility of Consensus. The most common approach to handling faults is redundancy. There are three types of redundancy: Information redundancy involves adding extra bits to allow recovery from distorted bits. An example is adding a Hamming code to data in order to recover from noise in the transmission line. With time redundancy an action is performed and if need be it is performed again. An aborted transaction can be redone without harm to the system. For more information refer to Chapter 3 of Tanenbaum [2]. Time redundancy is the most frequently used solution for intermittent and transient faults. Physical redundancy involves adding more components to a system in case a component fails. Physical Redundancy There are two types of physical redundancy: The advantages and disadvantages of each must be weighed when determining which type of physical redundancy will be implemented. Consider a circuit with devices A, B, and C linked in sequence. If all devices are working properly then the final result will be correct. But if one of the devices is faulty then the final result will probably be incorrect. Now we will look at a circuit utilizing TMR. First devices A, B and C are replicated three times and then three voters are added after each stage of the circuit. Why are three voters required in this system? The answer is that the voters are components too and might fail. Each voter has three inputs but only one output. The majority of inputs become the output of the voter. If two or all inputs are the same then that becomes the output. If all three inputs are different then the output is undefined. Now let us see if the system will be fault tolerant when different components fail. First, consider a simple case where A1 fails. The voters will pass on the value of A2 and A3 since that value is in the majority.

MECHANISMS FOR RELIABLE DISTRIBUTED REAL-TIME OPERATING SYSTEMS pdf

4: Reliability of Distributed Systems

Additional info for Mechanisms for Reliable Distributed Real-Time Operating Systems. The Alpha Kernel. Sample text. The capabilities to be passed, either into or out of an object, arc specified as part of the object's interface in the formal parameter list of the appropriate operation.

Description[edit] Structure of monolithic kernel, microkernel and hybrid kernel-based operating systems A distributed OS provides the essential services and functionality required of an OS but adds attributes and particular configurations to allow it to support additional requirements such as increased scale and availability. To a user, a distributed OS works in a manner similar to a single-node, monolithic operating system. That is, although it consists of multiple nodes, it appears to users and applications as a single-node. This separation increases flexibility and scalability. The kernel[edit] This article needs attention from an expert in Computing. The specific problem is: See questions asked as comments in the "kernel" section. WikiProject Computing may be able to help recruit an expert. A kernel of this design is referred to as a microkernel. These components are the part of the OS outside the kernel. These components provide higher-level communication, process and resource management, reliability, performance and security. The components match the functions of a single-entity system, adding the transparency required in a distributed environment. In addition, the system management components accept the "defensive" responsibilities of reliability, availability, and persistence. These responsibilities can conflict with each other. A consistent approach, balanced perspective, and a deep understanding of the overall system can assist in identifying diminishing returns. Separation of policy and mechanism mitigates such conflicts. Architecture and design must be approached in a manner consistent with separating policy and mechanism. In doing so, a distributed operating system attempts to provide an efficient and reliable distributed computing framework allowing for an absolute minimal user awareness of the underlying command and control efforts. This is the point in the system that must maintain a perfect harmony of purpose, and simultaneously maintain a complete disconnect of intent from implementation. However, this opportunity comes at a very high cost in complexity. The price of complexity[edit] In a distributed operating system, the exceptional degree of inherent complexity could easily render the entire system an anathema to any user. As such, the logical price of realizing a distributed operation system must be calculated in terms of overcoming vast amounts of complexity in many areas, and on many levels. This calculation includes the depth, breadth, and range of design investment and architectural planning required in achieving even the most modest implementation. Each of these design considerations can potentially affect many of the others to a significant degree. This leads to a massive effort in balanced approach, in terms of the individual design considerations, and many of their permutations. As an aid in this effort, most rely on documented experience and research in distributed computing. History[edit] Research and experimentation efforts began in earnest in the s and continued through s, with focused interest peaking in the late s. A number of distributed operating systems were introduced during this period; however, very few of these implementations achieved even modest commercial success. Fundamental and pioneering implementations of primitive distributed operating system component concepts date to the early s. These pioneering efforts laid important groundwork, and inspired continued research in areas related to distributed computing. These breakthroughs provided a solid, stable foundation for efforts that continued through the s. The accelerating proliferation of multi-processor and multi-core processor systems research led to a resurgence of the distributed OS concept. The introduction focused upon the requirements of the intended applications, including flexible communications, but also mentioned other computers: Finally, the external devices could even include other full-scale computers employing the same digital language as the DYSEAC. For example, the SEAC or other computers similar to it could be harnessed to the DYSEAC and by use of coordinated programs could be made to work together in mutual cooperation on a common taskâ€¦ Consequently[,] the computer can be used to coordinate the diverse activities of all the external devices into an

effective ensemble operation. Each member of such an interconnected group of separate computers is free at any time to initiate and dispatch special control orders to any of its partners in the system. As a consequence, the supervisory control over the common task may initially be loosely distributed throughout the system and then temporarily concentrated in one computer, or even passed rapidly from one machine to the other as the need arises. It was completed and delivered on time, in May This was a " portable computer ", housed in a tractor-trailer , with 2 attendant vehicles and 6 tons of refrigeration capacity. Lincoln TX-2[edit] Described as an experimental input-output system, the Lincoln TX-2 emphasized flexible, simultaneously operational input-output devices, i. The design of the TX-2 was modular, supporting a high degree of modification and expansion. This technique allowed multiple program counters to each associate with one of 32 possible sequences of program code. These explicitly prioritized sequences could be interleaved and executed concurrently, affecting not only the computation in process, but also the control flow of sequences and switching of devices as well. Much discussion related to device sequencing. The full power of the central unit was available to any device. The TX-2 was another example of a system exhibiting distributed control, its central unit not having dedicated control. Intercommunicating Cells[edit] One early effort at abstracting memory access was Intercommunicating Cells, where a cell was composed of a collection of memory elements. A memory element was basically a binary electronic flip-flop or relay. Within a cell there were two types of elements, symbol and cell. Each cell structure stores data in a string of symbols, consisting of a name and a set of parameters. Information is linked through cell associations. Information was accessed in two ways, direct and cross-retrieval. Direct retrieval accepts a name and returns a parameter set. Cross-retrieval projects through parameter sets and returns a set of names containing the given subset of parameters. This was similar to a modified hash table data structure that allowed multiple values parameters for each key name. Cellular memory would have many advantages: The constant-time projection through memory for storing and retrieval was inherently atomic and exclusive. The authors were considering distributed systems, stating: We wanted to present here the basic ideas of a distributed logic system with We must, at all cost, free ourselves from the burdens of detailed local problems which only befit a machine low on the evolutionary scale of machines.

MECHANISMS FOR RELIABLE DISTRIBUTED REAL-TIME OPERATING SYSTEMS pdf

5: What is real-time operating system (RTOS)? - Definition

Summary. Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel deals with the Alpha kernel, a set of mechanisms that support the construction of reliable, modular, decentralized operating systems for real-time control applications.

Operating System The real-time operating system used for a real-time application means for those applications where data processing should be done in the fixed and small quantum of time. It is different from general purpose computer where time concept is not considered as much crucial as in Real-Time Operating System. RTOS is a time-sharing system based on clock interrupts. RTOS used Priority to execute the process. When a high priority process enters in system low priority process preempted to serve higher priority process. Real-time operating system synchronized the process. So that they can communicate with each other. Resources can be used efficiently without wastage of time. RTOS are controlling traffic signal; Nuclear reactors Control scientific experiments, medical imaging systems, industrial system, fuel injection system, home appliance are some application of Real Time operating system Real time Operating Systems are very fast and quick respondent systems. These systems are used in an environment where a large number of events generally external must be accepted and processed in a short time. Real time processing requires quick transaction and characterized by supplying immediate response. For example, a measurement from a petroleum refinery indicating that temperature is getting too high and might demand for immediate attention to avoid an explosion. In real time operating system there is a little swapping of programs between primary and secondary memory. Most of the time, processes remain in primary memory in order to provide quick response, therefore, memory management in real time system is less demanding compared to other systems. Time Sharing Operating System is based on Event-driven and time-sharing the design. In event-driven switching, higher priority task requires CPU service first than a lower priority task, known as priority scheduling. Switching takes place after fixed time quantum known as Round Robin Scheduling. In these design, we mainly deal with three states of the process cycle 1 Running: When a process has all the resources require performing a process, but still it is not in running state because of the absence of CPU is known as the Ready state. Interrupt latency is time between an interrupt is generated by a device and till it serviced. RTOS support static as well as dynamic memory allocation. Both allocations used for different purpose. Like Static Memory, the allocation is used for compile and design time using stack data structure. Dynamic memory allocation used for runtime used heap data structure. The primary functions of the real time operating system are to: Manage the processor and other system resources to meet the requirements of an application. Synchronize with and respond to the system events. Move the data efficiently among processes and to perform coordination among these processes. The Real Time systems are used in the environments where a large number of events generally external to the computer system is required to be accepted and is to be processed in the form of quick response. Such systems have to be the multitasking. So the primary function of the real time operating system is to manage certain system resources, such as the CPU, memory, and time. Each resource must be shared among the competing processes to accomplish the overall function of the system Apart from these primary functions of the real time operating system there are certain secondary functions that are not mandatory but are included to enhance the performance: To provide an efficient management of RAM. To provide an exclusive access to the computer resources. The term real time refers to the technique of updating files with the transaction data immediately just after the event that it relates with. Few more examples of real time processing are: Air traffic control system. Systems that provide immediate updating. Systems that provide up to the minute information on stock prices. Real time operating systems mostly use the preemptive priority scheduling. These support more than one scheduling policy and often allow the user to set parameters associated with such policies, such as the time-slice in Round Robin scheduling where each task in the task queue is scheduled up to a maximum time, set by the time-slice parameter, in a round robin manner. Hundred

of the priority levels are commonly available for scheduling. Some specific tasks can also be indicated to be non-preemptive. A process might not be executed in given deadline. It can be crossed it then executed next, without harming the system. Example are a digital camera, mobile phones, etc. A process should be executed in given deadline. The deadline should not be crossed. Examples are Airbag control in cars, anti-lock brake, engine control system, etc. Dinesh authors the hugely popular Computer Notes blog. Where he writes how-to guides around Computer fundamental , computer software, Computer programming, and web apps. For any type of query or something that you think is missing, please feel free to Contact us.

MECHANISMS FOR RELIABLE DISTRIBUTED REAL-TIME OPERATING SYSTEMS pdf

6: Real-time operating system - Wikipedia

Distributed real-time systems are inherently asynchronous, dynamic, and non-deterministic, and yet are nonetheless www.enganchecubano.com increasing complexity and pace of these systems precludes the historical reliance solely on human operators for assuring system dependability under uncertainty.

Stochastic digraphs with multi-threaded graph traversal Intertask communication and resource sharing[edit] A multitasking operating system like Unix is poor at real-time tasks. The scheduler gives the highest priority to jobs with the lowest demand on the computer, so there is no way to ensure that a time-critical job will have access to enough resources. Multitasking systems must manage sharing data and hardware resources among multiple tasks. It is usually unsafe for two tasks to access the same specific data or hardware resource simultaneously. Many embedded systems and RTOSs, however, allow the application itself to run in kernel mode for greater system call efficiency and also to permit the application to have greater control of the operating environment without requiring OS intervention. On single-processor systems, an application running in kernel mode and masking interrupts is the lowest overhead method to prevent simultaneous access to a shared resource. While interrupts are masked and the current task does not make a blocking OS call, the current task has exclusive use of the CPU since no other task or interrupt can take control, so the critical section is protected. When the task exits its critical section, it must unmask interrupts; pending interrupts, if any, will then execute. Temporarily masking interrupts should only be done when the longest path through the critical section is shorter than the desired maximum interrupt latency. Typically this method of protection is used only when the critical section is just a few instructions and contains no loops. This method is ideal for protecting hardware bit-mapped registers when the bits are controlled by different tasks. Mutexes[edit] When the shared resource must be reserved without blocking all other tasks such as waiting for Flash memory to be written , it is better to use mechanisms also available on general-purpose operating systems, such as a mutex and OS-supervised interprocess messaging. A non-recursive mutex is either locked or unlocked. When a task has locked the mutex, all other tasks must wait for the mutex to be unlocked by its owner - the original thread. A task may set a timeout on its wait for a mutex. There are several well-known problems with mutex based designs such as priority inversion and deadlocks. In priority inversion a high priority task waits because a low priority task has a mutex, but the lower priority task is not given CPU time to finish its work. But this simple approach gets more complex when there are multiple levels of waiting: Handling multiple levels of inheritance causes other code to run in high priority context and thus can cause starvation of medium-priority threads. The simplest deadlock scenario occurs when two tasks alternately lock two mutex, but in the opposite order. Deadlock is prevented by careful design. Message passing[edit] The other approach to resource sharing is for tasks to send messages in an organized message passing scheme. In this paradigm, the resource is managed directly by only one task. When another task wants to interrogate or manipulate the resource, it sends a message to the managing task. Although their real-time behavior is less crisp than semaphore systems, simple message-based systems avoid most protocol deadlock hazards, and are generally better-behaved than semaphore systems. However, problems like those of semaphores are possible. Priority inversion can occur when a task is working on a low-priority message and ignores a higher-priority message or a message originating indirectly from a high priority task in its incoming message queue. Protocol deadlocks can occur when two or more tasks wait for each other to send response messages. Interrupt handlers and the scheduler[edit] Since an interrupt handler blocks the highest priority task from running, and since real time operating systems are designed to keep thread latency to a minimum, interrupt handlers are typically kept as short as possible. This can be done by unblocking a driver task through releasing a semaphore, setting a flag or sending a message. A scheduler often provides the ability to unblock a task from interrupt handler context. An OS maintains catalogues of objects it manages such as threads, mutexes, memory, and so on. Updates to this catalogue must be strictly controlled. For this reason it can be problematic when an interrupt handler calls an

OS function while the application is in the act of also doing so. There are two major approaches to deal with this problem: RTOSs implementing the unified architecture solve the problem by simply disabling interrupts while the internal catalogue is updated. The downside of this is that interrupt latency increases, potentially losing interrupts. The segmented architecture does not make direct OS calls but delegates the OS related work to a separate handler. This handler runs at a higher priority than any thread but lower than the interrupt handlers. The advantage of this architecture is that it adds very few cycles to interrupt latency. As a result, OSes which implement the segmented architecture are more predictable and can deal with higher interrupt rates compared to the unified architecture. It is generally wrong to write real-time software for x86 Hardware. First, for stability there cannot be memory leaks memory that is allocated but not freed after use. The device should work indefinitely, without ever needing a reboot. For this reason, dynamic memory allocation is frowned upon. Another reason to avoid dynamic memory allocation is memory fragmentation. With frequent allocation and releasing of small chunks of memory, a situation may occur where available memory is divided into several sections and the RTOS is incapable of allocating a large enough continuous block of memory, although there is enough free memory. Secondly, speed of allocation is important. A standard memory allocation scheme scans a linked list of indeterminate length to find a suitable free memory block, [7] which is unacceptable in an RTOS since memory allocation has to occur within a certain amount of time. Because mechanical disks have much longer and more unpredictable response times, swapping to disk files is not used for the same reasons as RAM allocation discussed above. The simple fixed-size-blocks algorithm works quite well for simple embedded systems because of its low overhead.

7: Distributed operating system - Wikipedia

Add tags for "Mechanisms for reliable distributed real-time operating systems: the Alpha kernel". Be the first.

This will help draw similarities to traditional distributed-systems development while also recognizing that there are unique systems-design implications, implementation issues, and approaches to solution development. Scenarios Automotive Telematics Scenario In the automotive and avionics industry, embedded systems provide the capability of reaching new levels of safety and sustainability that otherwise would not be feasible, while adding functionality, improving comfort, and increasing efficiency. Examples of this include improved manufacturing techniques, driver-assistance systems in cars that help prevent accidents, and advanced power-train management concepts that reduce fuel consumption and emissions. It will have sensors, actuators, and smart embedded software, ensuring that neither the driver nor the vehicle is the cause of any accident. This concept extends to all aspects of the driving experience: For example, the car would know who is allowed to drive it and who is driving it; where it is; where it is going and the best route to its destination; and it would be able to exchange information with vehicles around it and with the highway. It would monitor its own state fuel levels, tires, oil pressure, passenger compartment temperature and humidity, component malfunction, need for maintenance and the state of the driver fatigue, intoxication, anger. The car would first advise and then override the driver in safety-critical situations, use intelligent systems to minimize fuel consumption and emissions, and contain an advanced on-board entertainment system. Challenges and Issues To enable this scenario, components would need to be embedded in long-lived physical structures such as bridges, traffic lights, individual cars, and perhaps even the paint on the roads. Some components will be permanently connected to a network, but many would be resource constrained for example, in terms of power while computing data and thus communicating it wirelessly only when necessary. The many pieces of such a system will of necessity be heterogeneous, not only in form but also in function. There may be subsystems that communicate to consumers in private vehicles, others that relay information from emergency vehicles to synchronize traffic lights, still others that provide traffic data and analysis to highway engineers, and perhaps some that communicate to law enforcement. How information will be communicated to those interacting with the system is of great importance in such an environment. Safety is a critical concern, and issues of privacy and security arise as well, along with concerns about reliability. Precision Agriculture Scenario Incorporating DES technology into agriculture is a logical development of the advances in crop management over the last few decades. Despite deep understanding and knowledge on the part of farmers about how to adjust fertilizers, water supplies, and pesticides, and so on, to best manage crops and increase yields, a multitude of variations still exist in soil, land elevation, light exposure, and microclimates that make general solutions less than optimal, especially for highly sensitive crops like wine grapes and citrus fruit. The latest developments in precision agriculture deploy fine-grained sensing and automated actuation to keep water, fertilizer, and pesticides to a minimum for a particular local area, resulting in better yields, lower costs, and less pollution-causing runoff and emissions. Furthermore, the data collected can be analyzed and incorporated as feedback control to adjust irrigation flow rate and duration tuned to local soil conditions and temperature. Sensors that can monitor the crop itself for example, sugar levels in grapes to provide location-specific data could prove very effective. In the future, DES might be used to deploy sensors for the early detection of bacterial development in crops or viral contamination in livestock, or monitor flows of contaminants from neighboring areas and send alerts when necessary. In livestock management, feed and vitamins for individual animals will be adjusted by analyzing data from networks of ingestible sensors that monitor amounts of food eaten, activity and exercise, and health information about individual animals and the state of the herd as a whole. Challenges and Issues In this scenario, embedded components must be adaptive, multimodal, and able to learn over time. They will need to work under a wide range of unpredictable environmental conditions, as well as to interact with fixed and mobile infrastructure and new elements of the system as they are added and

removed at varying rates of change. Aviation and Avionics Scenario The European Commission has set goals for the aviation industry of reducing fuel consumption by 30 percent by through the use of embedded systems. This may be a high goal to be achieved solely through the use of technology. But in this industry, the goals appear to very ambitious across the board. The unmanned aerial vehicle UAV for use in surveillance and in hazardous situations such as fire fighting promises to be cheaper, safer, and more energy efficient to operate than conventional aircraft. There are apparently many different kinds of UAVs under development: The aircraft of the future will have advanced networks for on-board communication, mission control, and distributed coordination between aircraft. These networks will support advanced diagnosis, predictive maintenance, and in-flight communications for passengers. For external communication, future aircraft will communicate with each other in spontaneous, specific-for the-purpose ways similar to peer-to-peer networks. Challenges and Issues The aviation and avionics industry has specific needs in terms of security, dependability, fault tolerance and timeliness, stretching the limits of distributed embedded-systems design and implementation. The whole system, if not each of its embedded components, needs to be high precision, predictable, and robust for percent operational availability and reliability. It must enable high bandwidth, secure, seamless connectivity of the aircraft with its in-flight and on-ground environment. It should support advanced diagnosis and predictive maintenance to ensure a to year operational life span. DES design environments and tools will need to provide significant improvements in product development cycles, ongoing customizations, and upgrades beyond those achievable with current distributed-systems development tools. Design advances in fast prototyping, constructive system composition, and verification and validation strategies will be required to manage this complexity. Manufacturing and Process-Automation Scenario Embedded systems are important to manufacturing in terms of safety, efficiency, and productivity. They will precisely control process parameters, thus reducing the total cost of manufacture. Potential benefits from integrating embedded control and monitoring systems into the production line include: One is a need for better man-machine interactions in what is fundamentally a real-time, man-plus-machine control loop. Providing better interactions will improve quality and productivity by ensuring that there are no operator errors, as well as by reducing accidents. Availability, reliability, and continuous quality of service are essential requirements for industrial systems achieved through advanced control, redundancy, intelligent alarming, self-diagnosis, and repair. Other important issues are the need for robustness and testing, coherent system-design methodology, finding a balance between openness and security, integrating old and new hardware with heterogeneous systems, and managing obsolescence. Medical and Health-Care Scenario Society is facing the challenge of delivering good-quality, cost-effective health care to all citizens. Medical care for an aging population, the cost of managing chronic diseases, and the increasing demand for best-quality health care are major factors in explaining why health-care expenditures in Europe are already significant 8. Medical diagnosis and treatment systems already rely heavily on advances in embedded systems. New solutions that mix embedded intelligence and body-sensing techniques are currently being developed [3], and current advances in this area address patient-care issues such as biomedical imaging, remote monitoring, automatic drug dispensing, and automated support for diagnosis and surgical intervention. Challenges and Issues The medical domain represents a complex and diverse arena for extraordinary developments that deploy a wide range of embedded systems. Implanted devices such as pacemakers and drug dispensers are commonplace, but need to become more sophisticated, miniaturized, and connected to networks of information systems. Wearable devices for monitoring and managing cholesterol, blood sugar, blood pressure, and heart rate must be remotely connected to the laboratory and to the operating room in a secure and reliable manner from beginning to end. Mobility Scenario Combining mobile communications with mobile computing is allowing people to talk to others and access information and entertainment anywhere at any time. This requires ubiquitous, secure, instant, wireless connectivity, convergence of functions, global and short-range sensor networks and light, convenient, high-functionality terminals with sophisticated energy management techniques. Such environments will enable new forms of working with increased productivity by making

information instantly available, when needed in the home, cars, trains, airplanes and wider-area networks. Imagine a hand-held or wearable device giving easy access to a range of services able to connect via a range of technologies including GSM, GPS, wireless, Bluetooth and via direct connection to a range of fixed infrastructure terminals. Potential applications and services include: Entertainment, education, internet, local information, payments, telephony, news alerts, VPNs, interfaces to medical sensors and medical services, travel passes and many more. Challenges and Issues Devices would need to reconfigure themselves autonomously depending on patterns of use and the available supporting capabilities in environment or infrastructure and be able to download new services as they became available. To develop such infrastructure, the gap between large, enterprise systems and embedded components would need to be bridged. Significant developments are required in technology for low-power and high performance computing, networked operating systems, development and programming environments, energy management, networking and security. Issues that need to be resolved in the infrastructure to support these kinds of scenarios include the provision of end-to-end ubiquitous, interoperable, secure, instant, wireless connectivity to services. Simultaneously the infrastructure must allow unhindered convergence of functions and of sensor networks. Addressing the constraints imposed by power management energy storage, utilization and generation at the level of the infrastructure and mobile device poses a major challenge. Home-Automation and Smart Personal Spaces Scenario By deploying DES in the home, an autonomous, integrated, home environment that is highly customizable to the requirements of individuals can be foreseen. Typical applications and services available today include intruder detection, security, and environmental control. But in the future, applications and services to support the young, elderly, and infirm will be developed, and these may have the ability to recognize individuals and adapt to their evolving requirements, thereby enhancing their safety, security, and comfort. Challenges and Issues Multidisciplinary, multiobjective design techniques that offer appropriate price and performance, power consumption, and control will have to be used if we are to realize the potential of embedded systems for home entertainment, monitoring, energy efficiency, security, and control. Such systems will require significant computational, communication, and data-storage capabilities. The mix of physical monitoring and data-based decision support by some form of distributed intelligence will rely on the existence of seamlessly connected embedded systems and the integration of sensors and actuators into intelligent environments. These systems will be characterized by ubiquitous sensors and actuators and a high-bandwidth connection to the rest of the world. Technologies will need to be developed that support sensing, tracking, ergonomics, ease-of-use, security, comfort, and multimodal interaction. Key to achieving this result will be developing wireless and wired communications and techniques for managing sensor information, including data fusion and sensor overloading. The challenges are to make such systems intelligent, trustworthy, self-installing, self-maintaining, self-repairing, and affordable, and to manage the complexity of system behavior in the context of a large number of interoperable, connected, heterogeneous devices. These systems will need to operate for years without service, be able to recover from failure, and be able to supervise themselves. Managing these embedded systems will require support of all aspects of the life cycle of the application and service infrastructures, including ownership, long-term storage, logging of system data, maintenance, alarms, and actions by the provider emergency, medical, or security services, authorization of access and usage, and charging and billing under a range of different conditions of use. Technical Imperatives Some of the most challenging problems facing the embedded-systems community are those associated with producing software for real-time and embedded systems. Such systems have historically been targeted to relatively small-scale and stand-alone systems, but the trend is toward significantly increased functionality, complexity, and scalability, as real-time embedded systems are increasingly being connected via wireless and wired networks to create large-scale, distributed, real-time, and embedded systems. These combined factors [4] require major technical imperatives to be addressed by both industry and research establishments: Self-Configuration and Adaptive Coordination Self-configuration is the process of interconnecting available elements into an ensemble that will perform the required functions at the desired performance level.

MECHANISMS FOR RELIABLE DISTRIBUTED REAL-TIME OPERATING SYSTEMS pdf

Self-configuration in existing systems is realized through the concepts of service discovery, interfaces, and interoperability. But embedded systems, which appear in hybrid environments of mobile and static networks with nodes of varying capability, energy availability, and quality of connectivity, are plagued by diverse and energy-limited wireless connectivityâ€”making low power discovery a challenge. Also scalable discovery protocols, security, and the development of adequate failure models for automatically configured networks will require that solutions be developed. Adaptive coordination involves changes in the behavior of a system as it responds to changes in the environment or system resources. Coordination will not necessarily be mediated by humans because DES could be so large and the time scale over which adaptation needs to take place too short for humans to intervene effectively. Achieving adaptive coordination in DES will draw on the lessons learned from adaptive coordination in existing distributed systems, but it will also require meeting the radical new challenges posed by the physically embedded nature of collaborative control tasks and large numbers of nodes, and further complicated by the relatively constrained capabilities of individual elements. Thus, to achieve adaptability in DES, solutions are needed in decentralized control and collaborative processing, and techniques must be developed to exploit massive redundancy to achieve system robustness and longevity. Trustworthiness If we can expect DES to be deployed in large numbers and become an essential part of the fabric of everyday life, five technical imperatives much be taken into account in their design from the outset: On reliability, current monitoring and performance-checking facilities, and verification techniques are not easily applicable to DES because of their large number of elements, highly distributed nature, and environmental dynamics. In terms of safety or the ability to operate without causing accidents or loss, bounded rational behaviors are essential, especially in the face of real-time systems and massive DES likely to exhibit emergent or unintended behaviors. It may be virtually impossible to distinguish physical from system boundaries in a large-scale DES, making security a big challenge. And, considering that networking of embedded devices will greatly increase the number of possible failure points, security analysis may prove even more difficult. Privacy and confidentiality policies will be exacerbated by the pervasiveness and interconnectedness of DES. Users will be monitored, and vast amounts of personal information will be collected. Thus, implementing privacy policies, such as acquiring consent in a meaningful fashion in large scale networks, will be very difficult. Related to all of the above, embedded systems must be usable by individuals who have little or no formal training. Unfortunately, usability and safety often conflict, so trade-offs will need to be made. Understanding the mental models people use of the systems with which they interact is a good way for designers to start addressing issues of usability and manageability of embedded systems. Computational Models New models of computation are needed to describe, understand, construct, and reason about DES effectively. Understanding how large aggregates of nodes can be programmed to carry out their tasks in a distributed and adaptive manner is a critical research area.

MECHANISMS FOR RELIABLE DISTRIBUTED REAL-TIME OPERATING SYSTEMS pdf

Teaching middle school mathematics for all James A. Telese Appendix 2: The Brown priests : biographical data? V. 4. Six French plays. Sam Choy the Makaha Sons A Hawaiian Luau The Official Patients Sourcebook on Blastomycosis Animal Crackers (Marx Brothers) Purdue owl mla Canadian materials for history, poetry, and romance Discovering Animal Behaviour Best arduino books 2017 Papa can you hear me piano sheet music Colder than hell Threats, countermeasures and advances in applied information security Hilda Doolittle (H. D.) Nirvana : University of Washington, Seattle, January 6, 1990 Tim Hughes Inclusive Classrooms from A to Z Jail : August 1942 The Intertextuality of the Epistles Kids Say the Best Things About God Thomas and the trucks The Legacy of Baskets Lymphocyte cell surface The world and his wife Bringing a dream to life Practical knowledge and direction of fit. Angular 2 ui development The selected works of Cesare Pavese How to discover relevant primary authority : using reference services and other secondary sources How bluegrass music destroyed my life K.O.d in the rift Trade Remedies for Global Companies Son of an otter, son of a wolf Moral instruction and fiction for children, 1749-1820 OneKey Student Access Kit Four Galaxies in Peace One-Liners from God Genetic engineering techniques Fearless traveler The resourceful teachers handbook Pressure Ulcer Risk