Journal of Communication and Computer 12 (2015) 273-285
doi:10.17265/1548-7709/2015.06.001

A Review of Operating System Infrastructure for Real-Time Embedded Software

Luis Fernando Friedrich and Mario A. R. Dantas

Department of Computer Science, PPGCC, University of Santa Catarina, CEP 88040-900, Florianopolis/SC, Brazil

Abstract: Since their early applications in the 1960s, embedded systems have come down in price and increased dramatically in processing power and functionality. In addition, embedded systems are becoming increasingly complex. High-end devices, such as mobile phones, PDAs, entertainment devices, and set-top boxes, feature millions of lines of code with varying degrees of assurance of correctness. Nowadays, more and more embedded systems are implementing in a distributed way, and a wide range of high-performance distributed embedded systems have been design and deployed. As many aspects of embedded system design become increasingly dependent on the effective interaction of distributed processors, it is clear that much effort needs to be focus on software infrastructure, such as operating systems, to ensure that they provide functionality to fulfill these requirements. This paper reviews some of the approaches associated with the operating systems used to fulfill these needs.

Key words: Embedded systems, real-time operating systems, multi-core.

1. Introduction

An embedded system can be defined as an combination of devices that includes a programmable computer and perhaps additional parts, either mechanical or electronic, and is designed to perform a dedicated function, but is not itself intended to be a general purpose computer. The word embedded reflects the fact that these systems are typically a fundamental part of a larger system. Embedded systems are also call embedded real-time systems. Embedded systems play an important role, in a wide variety of applications, from small stand-alone systems, like a network router, to complex DREs (distributed real-time embedded systems) supporting large-scale mission-critical domains such as avionic applications.

All embedded systems have dedicated functionality and are therefore dedicated systems. “Dedicated functionality”, means that the system has been design for a specific purpose and for a set of pre-defined tasks. Moreover, the system functionality predefine in the hardware and software.

The wide variety of applications implies that the properties, platforms, and techniques on which embedded systems are based on vary widely. Hardware needs can sometimes be meet with general-purpose processors. For instance, high-end devices, including mobile phones, PDAs, and consumer electronics (entertainment devices such as TVs and DVD players, set-top boxes, etc.), have incorporated microprocessors as a core system component, instead of using application-specific hardware. However, in many systems specific processors are required, for instance, DSP devices to perform fast signal processing. In exceptional cases where required reaction times are extremely short, microprocessor technology cannot always satisfy the timing constraints and hard-wired electronic logic devices must be use instead.

Functionality can be modified or adjusted through software changes. It is possible to add functionality to a device through a software upgrade as long as the hardware does not need to be modified, and the available memory space is large enough to

Corresponding author: Luis Fernando Friedrich, Dr. Eng., professor, research fields: distributed and real-time embedded operating systems.

accommodate the changes. For instance, high-end devices feature millions of lines of code with varying degrees of assurance of correctness. They may incorporate third-party components, and even complete operating systems (such as Linux) that can be installed by the manufacturer, suppliers and even the end user. In such cases, it becomes impossible for embedded system vendors to provide guarantees about the behavior of the device, when supporting such devices using a traditional unprotected real-time executive approach. Failure or malicious behavior of a single software component on the device will affect the whole device.

It is also possible to modify part of the hardware functionality, using for example, technology such as FPGA (field-programmable gate array). Memory management capabilities are necessary in some systems to provide memory protection and virtual memory. Special purpose interfaces are also need to support a variety of external peripheral devices, control energy consumption, and carry out other functions.

In mission-critical systems besides having to meet deadlines, tasks are essentially critical and require special real-time responsiveness. However, beyond survivability mission-critical systems must also satisfy the same rigid requirements of reliability and fault tolerance. Dealing with such requirements demands a high degree of system function adaptability.

Nowadays, the use of processor-based devices has increased dramatically for most human activities, both professional and leisure. Rapid progress in processor and sensor technology combined with the expanding diversity of application fields is placing enormous demands on the facilities that software infrastructure such as operating systems must provide to ensure that they provide functionality to fulfill the requirements.

This paper reviews the current state of embedded systems from small stand-alone systems to distributed real-time systems, looking at some of the software infrastructure used to provide the functionality they need. Software infrastructure includes embedded operating systems, real-time operating systems and other forms of middleware. First, the paper presents the concepts and characteristics of embedded systems. Next, one presents some of the requirements that are generally useful for embedded systems. In the next section, we present some alternatives for the software infrastructure that are intend to provide the necessary functional and non-functional requirements for embedded system software to execute. Finally, we present conclusions on software infrastructure for embedded applications.

2. A Review of Embedded Systems

An embedded system is hided inside a system or environment, performing some dedicated function. The hosting system may be a specific system such as a car or an aircraft, a machine or a factory, but it may also be a person in the case of an intelligent pace maker or some hearing device, where the embedded system replaces or extends the human capabilities. Embedding computation into the environment and everyday objects (also called pervasive computing) would enable people to cooperate with information-processing devices in a more informal way than they currently do, and independent of their location or situation they find themselves.

As there is an enormous variety of embedded systems from small, intelligent sensors to vast aircraft control systems and vehicle simulators, the functions they performed may vary a lot. Examples of how real-time and embedded systems provide us with services are in Automotive or Avionics, Health and Medical Equipment, Consumer Electronics and Intelligent Homes, and Telecommunications.

2.1 Categories of Embedded Systems

Unlike PCs and workstations that execute regular non-real-time general-purpose applications, such as our editor and network browser, the computers and networks that run embedded real-time applications are often hide from our view. Concurrently, embedded systems are becoming smaller and smaller, heading more and more towards networked. In this paper, we aim embedded systems, which in turn we have subdivided in two categories: stand-alone embedded systems and networked embedded systems.

2.1.1 SES (stand-alone embedded systems)

The last two decades have witnessed a significant evolution of stand-alone embedded systems. These systems are being assembly from IP(intellectual property) components, which are assembly on a SoC (system-on-chip). SoCs offer a potential to embed complex functionalities and to meet demanding performance requirements of applications such as DSPs, network, and multimedia processors.

Most of CE (consumer electronics) devices are classify as SES. For instance, the explosion of the CE market over the past decade has generated products mainly in three categories:

Low-end devices generally built around application specific hardware like ASIC or SoC with small amounts of program memory (ROM), usually around 256 Kbytes, and inexpensive processors;

Mid-range consumer devices, such as video cameras, are characterized by moderate amounts of program memory like 1 to 2 Mbytes;

High-end devices, such as smart phones and set-top boxes, usually have much more memory, up to 32 Mbytes. In most cases, they use powerful processors and are develop by large programming teams.

2.1.2 NES (Networked embedded systems)

Networked embedded systems may come in many different forms. Fundamentally, networked embedded system is a collection of spatially and functionally distributed embedded nodes interconnected by means of wireline or wireless communication infrastructure and protocols, with some sensing and actuation elements interacting with the environment [1].

In order to be in accordance to the different forms that Networked Embedded Systems may come, in this paper we consider three types of networked embedded systems, based on what is proposed in [2]: Embedded Systems, Sensor Systems, and Distributed Real-Time and Embedded.

ES (embedded systems) are systems where the computing components are embed into some other purpose built device (an aircraft, a car, or a home). Here the characteristic is that these systems are usually not mobile and often not all devices are connected, usually with only one other server machine and most of the time not to external networks.

SN (sensor networks) are most of the time composed by a large number of possibly tiny devices having a single task which is monitoring some conditions within an environment and report back to a central server. Wireless sensor networks are a widely deployed example of networked embedded systems. There is a great interest both the industry and academia in wireless sensor networks technologies that enable deployment of a wide range of applications, such as military, environmental monitoring, e-health, etc.

DRES (distributed real-time and embedded systems) play an increasingly important role in modern application domains, including military command and control, avionics and air traffic control, and medicine and emergency response. Distributed real-time and embedded systems outline a computational infrastructure of many large-scale mission-critical domains to control a variety of artifacts across a number of sites. In life-critical military DRE systems [3], to provide the right answer at the right time is crucial.

2.2 Properties of Embedded Systems

Traditional, embedded software can be quite complex and have a number of requirements. These have implications both for the application and for the software infrastructure, such as the operating system. According to Crespo [4], embedded software have several common features (properties) such as the following:

Rc (resource-constrained computing), meaning they are frequently rigorously constrained regarding available resources. As a result of these restrictions, the system needs to use efficiently its computational resources. For instance, the operating system must be able to operate in resource-constrained environments;

Rt (real-time requirements) are needed because many embedded applications interact deeply with the real world, they often have strict real-time requirements to fulfill such as temporal requirement, or deadline;

Ec (embedded control systems), because most of the embedded systems perform control activities involving input data acquisition (sensing) and output delivery (actuation);

Qs (quality of service), having an efficient use of the system resources is necessary in embedded systems. Feedback based approaches are being used to adjust the performance or quality of service of the applications as a function of the available resources;

Mc (multi-Core platforms) are becoming very common. The great majority of CPU manufacturers have migrated to multicore technology as a mean to keep improving the performance of their product lines;

Embedded systems are getting more and more complex, dynamic, and open, while interacting with a progressively more demanding and heterogeneous environment. As consequence, the reliability and security of these systems have become major concerns. For example, safety-critical systems must not only guarantee real-time behavior but furthermore they require absolute dependability and availability of system service;

The Dp (notion of dependability) includes aspects of reliability and availability. Reliability and availability relate to the probability of working continuously for a given duration and the percentage of uptime, respectively;

Sf (Safety) is concerned with the prevention of the loss of life and/or serious damage to people, property or the environment. This is straightforward for medical devices, but the characteristic is also valid for an aircraft, a car, etc.;

Sc (Security) is concerned with the capability to prevent information and system resources from being use or altered by unauthorized users. A secure system is one where only intended use of the system will be permit.

As embedded systems get networked, the scenario gets a little more complex and while some common features become more stringent, other requisites become very important to be provided.

Now, supporting properties such as Fh (failure handling) are relevant to NES (networked embedded systems). In NES, nodes may fail or experience problems due to several reasons. The failure of individual nodes should not affect the overall task of a networked embedded system thus leading to an increased need for providing mechanisms to ensure fault tolerance to the applications.

Table 1 summarizes the different categories of embedded systems (presented in Sections 2.1.1 and 2.1.2) and properties (introduced in Section 2.2) which are usually required in each of them.

In order to establish the importance of the properties to the different embedded systems they are classify as M (mandatory) in case you must fulfill the requirement, W (wanted) when it is advantageous to have it, O (optional) when to have it or not is not a big issue.

3. A Review of Operating System Infrastructure for Embedded Systems

Embedded systems have been around at least as long as the microprocessor. The software for these systems has been build, more or less successfully, using several different paradigms. The manufacturer builds some systems from scratch with all software being created specifically for the device in question. Not all components of embedded systems need to be design from scratch. For instance, software infrastructure

Table 1 Properties of embedded systems.

	Rc	*Rt*	*Ec*	*Qs*	*Mc*	*Dp*	*Sf*	*Sc*	*Fh*
SES									
Low-end	M	M	M	W	M	W	M	W	W
Med-end	W	W	M	M	M	W	W	W	W
High-end	O	W	M	M	M	W	W	W	W
NES									
ES	M	M	M	M	W	M	M	W	M
SN	M	W	M	M	O	M	W	M	M
DRES	W	M	M	M	W	M	M	M	M

such as OS (operating systems), are examples of reusable software components. Such components are available from independent software vendors and in some cases as open source software.

The rapid progress in processor and sensor technology combined with the expanding diversity of application fields is placing huge demands on the facilities that an embedded operating system must provide. The variety of applications where embedded systems are being important also implies that the properties, platforms, and techniques on which embedded systems founded can be very different. All the types of embedded systems, from SES (stand-alone embedded systems) to NES (networked embedded systems), need some specific type of service from software infrastructure such as operating system. For instance, these services are supposed to be prepared to attend to functional and non-functional requirements. In this section, we give an overview of presently available software infrastructures for embedded systems, starting with operating systems architectures. In addition, it presents an overview of existing operating systems divided in two categories: real-time embedded systems and domain specific embedded systems.

For each one of these categories it presents a table in order to survey how the embedded systems properties, introduced in Section 2.2, have being support by these operating systems. The table gives you a view on how the operating systems provide and enforce a certain property: A (adequately) when the OS provides and enforces the property; P (poorly) when the OS provides the property but does not enforce it and N (none) when the OS is not able to support the property.

3.1 Real-Time Embedded Operating Systems Architectures

Embedded systems typically have requirements that are very different from the ones for desktop computers, and hence operating systems for embedded systems are diverse from GPOS (general-purpose operating systems). Operating systems for embedded systems usually are design to be tailor for a specific application and therefore are more static than GPOS. In addition, most embedded systems are application specific and require real-time guarantees to function correctly. Therefore, they need RTOS (real-time operating systems), which can be flexible and customized towards the application at hand.

Designing a proper RTOS architecture needs some delicate decisions. The basic services like process management, inter-process communication, interrupt handling, or process synchronization have to be provided in an efficient manner making use of a very restricted resource budget. The kernel design, one of the most common Operating Systems architectures has been around for almost 40 years and offers a clear separation between the operating system and the application running on top of it. The processes can use the kernel functionality by performing system calls. System calls are software interrupts which allow switching from the application to the operating system. Various techniques such as library-based approaches, monolithic kernels, microkernels, or virtual machines have been develop in order to structure the OS, each of them dedicated to specific demands [5]. The challenge is how to implement applications that can execute efficiently on limited resource and that meet non-functional requirements such as timeliness, robustness, dependability, performance, and so on.

3.1.1 Library-Based Architecture

Not all embedded systems need to be support by operating system functionality. Usually, the simplest embedded systems are building without an explicit operating system. Such systems do not require complex mechanisms or real-time scheduling of concurrent tasks, and therefore the implementation of then use a simple main loop or executive cyclic approaches. In addition, dedicated networked embedded systems usually do not need an operating system for its implementation, but only a stand-alone protocol stack. For instance, in systems without MMUs, RTOS are building as a library linked together with the application. The result is one single executable that executed in one single address space. In this particular case, system calls simply implemented as function calls and no context-switches are required when calling an operating system function. In addition, no loader is required to load applications at run-time. This is often more efficient and less time consuming as a full context switch with address space changes when having an RTOS implemented as a kernel in a separated address space. However, when using a library based RTOS running on systems without MMU, we have no security through hardware memory separation. The implementation of all application and operating system activities are in the same address space. Bugs in one part of the system easily propagate to the whole system. On small microcontrollers where only one application execute this disadvantage is acceptable.

3.1.2 Monolithic Kernel Architecture

The monolithic approach of building a kernel asserts that all functionality provided by the OS are within the kernel itself, “The structure is that there is no structure” [6]. The kernel composes by a set of procedures that are able to call each other without any restrictions. These procedures include service functions that are the entry points for the interrupts to the kernel. The advantage of monolithic kernel organization is performance, low overhead in doing system calls. However, the big problem in the case of monolithic kernels is that any single fault occurring within the kernel functions can lead to a total crash of the whole system. In most cases, device drivers included in a monolithic kernel are very error-prone. Several studies on software dependability report fault densities of 1 to 16 bugs per 1,000 lines of executable code. Drivers, which typically comprise 70% of the operating system code, have a reported error rate that is 3 to 7 times higher.

3.1.3 Microkernel Architecture

In order to better organizing monolithic kernels the microkernel architecture was develop. The basic assumption is to reduce the services provided by the kernel dramatically by putting all services, which are not essentially necessary for the microkernel, into user space as isolated processes. In this case, to use a service a client application process needs to send a message requesting the service to a server process which receives the request, completes the request and sends back a response message to the client application process. Usually, the following services are located into the microkernel itself: Dispatcher, Scheduler, and Memory Manager. The advantage of microkernel’s if compared to monolithic kernels is the separation of services from the kernel itself. As a result, making the kernel a very small piece of software is more dependable and maintained more easily than a monolithic kernel. However, to get better structuring and dependability, it does need a lot of inter-process communication through message passing and much context switching. In comparison to monolithic kernels, it also has an effect on the real-time behavior. In microkernel architecture, system calls do not necessarily execute at the time they have been initiate, because the services behave like regular processes that have to be schedule by the real-time scheduler.

3.1.4 Multi-Core Architecture

Multi-core architectures are an attempt to solve the lack of computational power in embedded systems by enabling computational concurrency. By using several slower cores together with lower-level hardware support for very specific tasks instead of cost-expensive high performance processors, you can get very good parallel performance and correspond to the cost constraints of the embedded system market. However, multi-core architectures imply further challenges on the software and hence on operating systems that support these architectures. The design of an operating system applied in multi-core systems is strongly dependent on the underlying system architecture. The software design process is strongly coupled or even an inherent part of the hardware design. Depending on the architecture, the cores in a multi-core system typically share many more things, such as interrupt controllers, devices, and caches. Multi-core architectures need special techniques for process management, memory management, and synchronization. Although multicores and operating systems can be mix in a wide variety of configurations, there are some use cases that are more common than others are. When utilizing a multicore, there are two fundamental ways the operating system can manage the cores: SMP (symmetric multiprocessing) and AMP (asymmetric multiprocessing).When using SMP, one operating system is controlling multiple cores. When a core becomes available, it will run the next thread on the ready queue. AMP, on the other hand, is using one operating system per core. It could be two copies of the same OS or two very different OSes such as an RTOS and a general purpose OS like Linux.

There are advantages and disadvantages of both approaches, and the best choice depends on the application. SMP has the advantage that it can load-balance threads over the cores as long as there are more threads that typically can run than there are processors. In addition, it is generally easier to manage and develop for only one operating system. AMP, on the other hand, is more efficient because there is less synchronization needed between the cores. In addition, AMP has other advantages such as dependability, if one core goes down, the rest can continue to operate; and scalability, AMP typically scales to a larger number of cores.

Regarding the scheduling algorithms, Brandenburg [7] categorizes the scheduling approaches that an Operating System can adopt into partitioned, global and clustered. The partitioned approach employs processor affinity, forcing each thread to run into a specific core, and scheduling each core individually. The global scheduling approach permits threads to migrate from one core to another, thus the cores share a single run queue. Finally, a clustered approach, which groups some cores together and provides a single run queue for each group, using global scheduling within a group.

3.1.5 Virtual Machine Architecture

The main idea of system virtual machines is to provide an exact copy of the available hardware for every virtual machine. Therefore, a small control program is necessary to assign the available hardware to the virtual machines. This program is called the VMM (virtual machine monitor) or hypervisor. A hypervisor is a piece of code that manages one or more operating systems that run on top of it. This program is the only code executed in supervisor mode and ensures that the virtual machines are clearly isolated from each other. It does that by creating partitions (or virtual boards) that run on top of the hypervisor, by virtualizing certain aspects of a system. For example, by virtualizing the CPU, you can run multiple virtual CPUs on top of one physical CPU or core. In addition, virtualization usually applied to memory in order to split the physical memory up so that multiple partitions can use some part of the real memory. Finally, when multiple virtual boards need to share devices such as serial ports, Ethernet ports, graphics, and so forth, you need to virtualize the devices as well.

There are two types of hypervisors. Type 1 is a dedicated hypervisor (also called embedded hypervisor) that is small and runs directly on the hardware. Type 2 (also called server hypervisor) typically runs on top of or in conjunction with an operating system and uses the resources of that host operating system. The main differences between an embedded hypervisor and a server hypervisor are due to very different requirements. For instance, performance and isolation are the two main goals for an embedded hypervisor while for a server hypervisor it is very important to be able to run a guest OS unmodified.

3.2 Real-Time Embedded Operating Systems

Traditional RTOS (real-time embedded operating systems) such as VxWorks [8], and QNX [9], are typically large (requiring hundreds of KB or more of memory), general-purpose systems consisting of a binary kernel with a rich set of programming interfaces. Such RTOS target systems with greater CPU and memory resources, and generally support features such as full multitasking, memory protection, TCP/IP networking, and POSIX-standard APIs that are undesirable (both in terms of overhead and generality) for sensor network nodes. They provide memory protection given the appropriate hardware support.

QNX Neutrinois a RTOS with a microkernel architecture [10]. In the traditional SMP scheduling style, the QNX Neutrino allows threads to migrate from one processors to another, once all cores are peers for the scheduling of threads. This scheduling style is equivalent to the global approach. Besides the SMP and AMP styles mentioned before, the QNX Neutrino also employs a third, and more restrictive, style of Symmetric Multiprocessing, called BMP (bound multiprocessing). In this style, a single instance of an OS is responsible to manage all CPUs, like in the ordinary SMP, however it applies “processor affinity” [10], where it is not allowed thread migration from one core to another. The BMP falls under the Brandenburg’s partitioned scheduling approach.

The Wind River’s VxWorks RTOS is implemented as a monolithic kernel that supports both Symmetric and Asymmetric Multiprocessing [11]. The former is as an add-on under the name of VxWorks SMP. On SMP systems, the VxWorks allows CPU affinity (partitioned scheduling approach), or it can automatically load balance the threads among the available cores (global scheduling approach). Along with the SMP capabilities, VxWorks provides new features such as Spin-locks, Read-write semaphores, Atomic operators and Thread-local storage.

Several embedded systems have taken a component-oriented approach for application-specific configurability [12]. These systems consist of a set of components that are wired together to form an application. Components vary in size from fine-grained, specialized objects to larger classes and packages.

The most widely adopted free, open source RTOS, eCos (embedded Configurable operating system) [13] was release in 1986. It provides a graphical-configuration tool and a command line-configuration tool to customize and adapt the RTOS to meet application-specific requirements. eCos kernel configuration can be done with the bitmap scheduler or the MLQ (multilevel queue) scheduler. Both schedulers support priority-based scheduling with up to 32 priority levels. The bitmap scheduler is somewhat more efficient and only allows one thread per priority level. The MLQ scheduler allows multiple threads to run at the same priority.

Contemporary OS such as Linux has been use for embedded applications because of its capabilities supporting soft real-time applications. Linux also has extensions that enable them to support hard real-time applications. These RTOSes such as RTAI [14] are only suitable for large real-time systems due to footprint required. Other Linux extensions such as Embedded Linux [15] and µClinux [16] are more embedded compliant but do not provide real-time support. These extensions have an architecture that is very much alike the Linux architecture, except they do not have MMU (memory management hardware) support. Table 2, summarizes some operating systems for embedded systems and how they provide the properties (requirements) of section 3. Most of the RTOSes for embedded systems provide the necessary functionalities such as multitasking, memory management, file system, network, etc., through its API. They provide various architecture approaches such as monolithic kernel, micro-kernel based, components based, and library based. However, they do not provide and enforce some important properties that are necessary in embedded systems especially the properties usually required in networked embedded systems. In addition, most of the RTOS listed required large amount of memory to run and do not deal with low power consumption. According to 2012 Embedded Market Survey [17], 40% of embedded systems developers are using commercial operating systems or RTOS products for their current project, 31% are using open-source operating systems in their projects, and 20% are developing their own operating systems for their projects.

3.3 Domain Specific Embedded Operating Systems

A Domain Specific Embedded Operating System provides specific characteristics for different domains of embedded systems such as automotive, avionics, mission critical systems and sensor networks systems, without loose OS functionality. This section presents two specific domains: Safety-critical Systems and Embedded Sensor Networks.

3.3.1 RTOS for Safety Critical Systems

Computer systems that operate systems of critical responsibility called safety critical systems. Typically, a small deviation in the environment or the system's behavior, a failure or an error appearing within such a system can yield in hazardous situations and may cause catastrophes. Safety-critical systems therefore must not only guarantee real-time behavior but furthermore they require absolute dependability and availability of system service. To free application developers from implementing safety and real-time mechanisms into each application, operating systems serve as the underlying platform designed towards supporting real-time and all safety-incorporating non-functional features.

As recent trends are heading towards the integration of applications of different criticality levels on one single platform, operating systems for safety-critical applications face the challenge of guaranteeing the availability of the processor time as well as the availability of resources.

Significant work has been performed within the avionics domain to achieve the stated objectives. The main body of work has been performed under the banner IMA (integrated modular avionics). The ARINC 653 [18] is a standard that specifies a programming interface for a RTOS. In addition, it establishes a particular method for partitioning resources over time and memory. ARINC 653 defines an APEX (application executive) for space and time

Table 2 Embedded systems properties supported by RTOSes.

	Rc	*Rt*	*Ec*	*Qs*	*Mc*	*Dp*	*Sf*	*Sc*	*Fh*
VxWorks	P	P	A	A	A	P	P	A	P
QNX	P	P	A	A	A	A	P	A	P
eCos	A	P	A	P	P	N	P	P	N
RTAI	N	A	P	P	P	P	N	P	N
uClinux	A	N	A	P	N	N	N	P	N

partitioning that may be used wherever multiple applications need to share a single processor and memory, in order to guarantee that one application cannot bring down another in the event of application failure. Each partition in an ARINC 653 system represents a separate application and makes use of memory space dedicated to it. MILS (multiple independent levels of security and safety) approach [19] is proposed to provide a reusable formal framework for high assurance system specification and verification. Separation of kernel is the big issue. In the MILS architecture, the kernel only does four very simple things. A MILS kernel is responsible for enforcing data isolation, control of information flow, periods processing and damage limitation policies, and nothing else. Each of these policies counters one or more of the basic foundational threats to system assurance. The MILS kernel has two special characteristics. First, the kernel is the only code that runs in supervisor or privileged mode. The second characteristic is that because the separation kernel is so simple it can be very small, approximately 4,000 lines of C language source code.

LynxSecure [20] and PikeOS [21] are examples of MILS compliant OS, used especially in military and avionics industries. Another system, RTEMS [22] that is not MILS compliant also used for military and avionics applications.

LynxSecure provides a robust environment within which multiple secure and non-secure operating systems can perform simultaneously with no compromise of security, reliability, or data. Lynx Secure expands on the proven real-time capabilities of the LynxOS® RTOS (real-time operating system) with time-space partitioning and operating system virtualization. It conforms to the Multiple MILS architecture, with strict adherence to data isolation, damage limitation and information flow policies identified in this architecture. To fulfill the separation kernel concept of MILS architecture, LynxSecure utilizes virtualization.

PikeOS is a microkernel-based real-time operating system, targeted at safety and security critical embedded systems. It provides a partitioned environment for multiple operating systems with different design goals, safety requirements, or security requirements to coexist in a single machine. The goal of PikeOS is to provide partitions that comprise a subset of the system's resources. Processing time is one of those resources. It is expected that the partitions to host a variety of guest operating systems with different requirements regarding timely execution. PikeOS combines resource partitioning and virtualization, in order to fulfill the MILS separation kernel concept. PikeOS provides a build-in Health Monitoring feature that implements all features described in the ARINC-653 standard.

RTEMS (real-time executive for multiprocessor systems) is a free open source RTOS (real-time operating system) designed for embedded systems. On a conceptual level, RTEMS is defined for three layers: hardware support, kernel and APIs. The user then develops his application by using the available APIs. The hardware support layer encompasses the processor and board dependent files as well as a common hardware library. RTEMS provides a notion of executive that encapsulates the API layer and the kernel.

Table 3 summarizes how the operating systems for avionics domain embedded systems provide the properties (requirements) of Section 2.

All systems tend to be compliant to ARINC and MILS specifications, where the big issue is Security and Safety. However, some basic requirements for

Table 3 Embedded systems properties RTOS for avionics.

	Rc	*Rt*	*Ec*	*Qs*	*Mc*	*Dp*	*Sf*	*Sc*	**Fh**
Lynxsecure	P	P	P	A	A	A	A	A	P
PikeOS	P	P	P	A	A	A	A	A	P
RTEMS	P	A	P	A	A	A	A	A	P

embedded systems such as resource constrains and real-time are not adequately handled. In addition, it seems that failure handling is not a very concern for these systems.

3.3.2 Operating Systems for Embedded Sensor Networks

Recently, the availability of cheap and small tiny sensors and low power wireless communication allowed the large-scaled deployment of sensor nodes in ESN (embedded sensor networks). An embedded sensor network is a network of embedded computers placed in the physical world that interacts with the environment. These embedded computers, or sensor nodes, are often physically small, relatively inexpensive computers, each with some set of sensors or actuators [23].

Given the recent advances in SN technology, it is possible to construct low-cost and low-power miniature sensor devices that are spreading across a geographical area in order to monitor their physical environment. Consisting of nodes equipped with a small processing unit, memory, a sensor, a battery and a wireless communication device, SNs enable a myriad of applications ranging from human-embedded sensing to ocean data monitoring. Since each single node has only constrained processing and sensing capabilities, coordination among devices is necessary [24]. Due to their specific nature, sensor networks have different requirements compared to standard systems, such as self-configuration, energy-efficient operation, collaboration, in-network processing, as well as, a useful abstraction to the application developer.

Given these requirements, a SN OS must have a very small footprint. At the same time, it must provide a limited number of common services for application developers , such as hardware management of sensor, for sensing and data delivery to neighbors; task coordination, using event-based or preemptive approaches to run tasks; power management, using different power management techniques.

The sensor networking community typically uses embedded (and, possibly, real-time) versions of existing operating systems such as Linux for the larger devices. These embedded versions provide largely the same programming support as their regular counterparts, but with additional device-level support for embedded controllers, flash memory, and other peripherals specific to these devices. As such, not much research has been required on new operating systems support for these larger devices. On the other hand, the smaller devices (such as the motes) have required novel directions in operating system design.

TinyOS [25], a very efficient OS for SNs and widely used by many research groups as well as in some segments of industry, deviates significantly from the traditional multi-threaded model of modern operating systems. TinyOS uses event based task coordination in order to run on very resource-constrained nodes. The execution model is similar to a finite state machine. It consists of a set of components that are included in the applications when necessary. TinyOS addresses the main challenges of a sensor network: constrained resources, concurrent operations, robustness, and application requirement support. Each TinyOS application consists of a scheduler and a graph of components. The concurrency model in TinyOS consists of a two-level scheduling hierarchy: events preempt tasks, but tasks do not preempt other tasks. Each task can issue commands or put other tasks to work. Events initiate by hardware interrupts at the lowest levels. They travel from lower to higher levels and can signal events, call commands, or post tasks. Wherever a component cannot accomplish the work in a bounded amount of time, it should post a task to continue the work. This is because a non-blocking approach implemented in TinyOS.

MOS (MantisOS) [26]. The MOS (mantis operating system) is a SN OS designed to behave similarly to UNIX and provides a larger functionality than TinyOS. It is a lightweight and energy-efficient multithreaded OS for sensor nodes. In contrast to TinyOS, the

Table 4 Embedded systems properties OS for sensor networks.

	Rc	*Rt*	*Ec*	*Qs*	*Mc*	*Dp*	*Sf*	*Sc*	*Fh*
TyniOS	A	P	A	P	N	P	N	N	N
Contiki	A	P	A	P	N	P	N	N	N
MantisOS	A	P	A	P	N	P	N	N	N

MANTIS kernel uses a priority-based thread scheduling with round-robin semantics within one priority level. To avoid race conditions within the kernel, binary and integer semaphores are support. The OS offers a multiprogramming model similar to that present in conventional OS, i.e., the OS complies with the traditional POSIX-based multithreading paradigm. All threads coexist in the same address space. The kernel of Mantis OS also provides device drivers and a network stack. The network stack is implemented using user-level threads and focuses on the efficient use of the limited memory.

Contiki [27]. The Contiki operating system provides dynamic loading and unloading of programs and services during run-time. It also supports dynamic downloading of code enabling the software upgrade of already deployed nodes. To provide this it uses more memory than TinyOS but less than Mantis OS. The main idea of Contiki is to combine the advantages of event-driven and preemptive multithreading in one system: the kernel of the system is event-driven, but applications desiring to use multithreading facilities can simply use an optional library module for that. In Contiki system, the partition (core and loaded programs) is determined at compilation time. The core comprises the kernel, program loader, run time libraries, and communication system.

Table 4 summarizes how operating systems for sensor networks domain provide the properties (requirements) of Section 2.

One of the characteristics of OSes for sensor networks is to be adequately able to deal with resource constraints such as memory and low power. They all provide and ensure this particular requirement. However, most of the systems do not provide an adequate real-time behavior. Regarding some of the important requirements for distributed embedded systems, these OSes perform very poorly. Therefore, it seems that those requirements have to be supported in a different level than operating system, such as middleware level.

4. Conclusions

The paper presented various operating systems tailored for different types of embedded systems, from stand alone embedded systems such as adigital câmera to distributed real-time embedded systems such as a military defense system. For SES (stand-alone system) the important requirements are basically supported by the OSes. However, there is currently no OS that can provide and enforce most o the requirements for network edembedded systems. For instance, the many types of DRE systems all have one requirement in common: to deliver the right response at the right time. This is crucial for life-criticalmilitary DRE systems, such as those defending ship sagainstmissile attacks. It is also crucial for safety-criticalcivilian DRE systems, such as those regulating the temperature of coolant in nuclear reactors and maintaining the safe operation of steel manufacturing machinery. It is hard to design DRE systems that implement their requiredQoS (quality of service) capabilities, are dependable and predictable, and are cost-conscious in their use of computing resources, with only the support of operating system software infrastructure. It is even harder to build the mon time and within budget. As a result, distributed real-time and embedded systems are now being built using a common layerof software infrastructure, called middle ware, that serves two purposes. The first is to ease application development by abstracting a way the particular details of the hardware and operating system a teach computing site.

The second is to provide a family of services that are common to many applications, simplifying component design and increasing reusability while all owing specific optimizations for a particular deployment.

Networked embedded systems, especially DRE systems can span a variety of network types, to pologies and scales, ranging from next-generation local sensor/actuator networks tolarge-scale traffic or power-grid management systems. A general theme of these systems is that independently of network characteristics, the stringent application-specific constraints must been forced by all in frastructure software services, including OS, middle ware and applications as well as end-to-end by the network. Moreover, applications with different constraints will share the network and other physical and logical resources.

How to provide different service classes and ensure the proper allocation and protection of shared resources consistently across all layers, including operating systems, is another important challenge. Creating the proper interfaces between network infrastructure and applications is still an open research issue for operating systems area.

References

[1] NRC. 2001. *Embedded, Everywhere: A Research Agenda for Networked Systems of Embedded Computers*. Committee on Networked Systems of Embedded Computers, National Research Council.

[2] FP6-IP-RUNES. 2005. "D5.1 Survey of Middleware for Networked Embedded Systems, January 2005. http://www.ist-runes.org/docs/deliverables/D5_01.pdf.

[3] Lee, I., Leung, J., and Son, S. 2008. "Handbook of Real-Time and Embedded Systems." *Chapman & Hall/CRC Computer &Information Science Series*.

[4] Crespo, A., Ripoll, I., Gonzalez, H. M., and Lipari, G. 2008. "Operating System Support for Embedded Real-Time Applications." *EURASIP Journal on Embedded Systems* 2008 (February): 1-2.

[5] Ecker, W. 2009. *Hardware-Dependent Software-Chapter 2*. Berlin: Springer Science.

[6] Tanenbaum, A. 2008. *Modern Operating Systems 3rd Edition*. New Jersey: Prentice Hall, Upper Saddle River.

[7] Wolf, W., and Jerraya, A. 2004. *Multiprocessor Systems-On-Chips*. San Mateo: Morgan Kaufmann.

[8] VxWorks. 2008. http://www.windriver.com.

[9] Hildebrand, D. 1992. "An Architectural Overview of QNX." In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, 113-26.

[10] www.qnx.com/products/neutrino-rtos/.

[11] www.windriver .com/products/vxworks/.

[12] Friedrich, L., Stankovic, J., Humphrey, M., Marley, M. and Haskins, J. 2001. "A Survey of Configurable Component-based Operating Systems for Embedded Applications."*IEEE Micro* (May): 54-68.

[13] Massa, A. 2003. *Embedded Software Development with eCos*. New Jersey: Prentice Hall.

[14] Dozio, L., and Mantegazza, P. 2003. "Real Time Distributed Control Systems Using RTAI." Presented at the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Hakodate, Hokkaido, Japan.

[15] Embedded Linux. 2008, www.linuxdevices.com/.

[16] µClinux. 2008. Embedded Linux Microcontroller Project. www.uclinux.org/.

[17] UBM Electronics. 2012. Embedded.com and EETimes, 2012 Embedded Market Survey. www.eetimes.com/electrical-engineers/education-training /webinars/4369712/2012-Embedded-Market-Study.

[18] ARINC. 1996. ARINC 653: Avionics Application Software Standard Interface (Draft 15). Airlines Electronic Engineering Committee (AEEC), June 17th, 1996.

[19] Alves, F. J., Harrison W.S., Oman, P. and Taylor, C. 2006. "The MILS Architecture for High-Assurance Embedded Systems." *International Journal of Embedded Systems*: 239-47.

[20] DeLong, R. J. 2007. *LynxSecure Separation Kernel—A High-Assurance Security RTOS*. LynuxWorks, San Jose, CA. www.lynuxworks.com.

[21] Kaiser, R., and Wagner, S. 2007. The PikeOS Concept History and Design, SYSGO. www.sysgo.com.

[22] RTEMS. 2008. www.rtems.com/.

[23] Heidemann, J., and Govindan, R. 2004. "An Overview of Embedded Sensor Networks." In: *Handbook of Networked and Embedded Control Systems*, Springer-Verlag.

[24] Stojmenovic, I. 2005. *Handbook of Sensor Networks: algorithms and Architectures*. New York: Wiley.

[25] TinyOS. 2008. www.tinyos.net/.

[26] Bhatti, S. 2005. "MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms." ACM/Kluwer Mobile Networks &Applications (MONET), *Special Issue on Wireless Sensor Networks* 10 (August): 563-79.

[27] Dunkels, A., Gronvall, B., and Voigt, T. 2004. "Contiki— A Lightweight and Flexible Operating System for Tiny Networked Sensors." In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors*.