Operating System Issues in Future End-User Systems

Invited paper

Kimmo E. E. Raatikainen, *Member*, *IEEE*

***Abstract*—Traditional monolithic operating systems have conceptually remained almost unchanged since the UNIX, that is, since the late 70s. Several experimental operating systems from the research community have been based on alternative paradigms. Today we are facing the dawn of mobile or wireless Internet. This new operational environment calls for new solutions. We discuss four significant research issues for future end-user systems: self-awareness, detection and notifications, system integrity, and power management. A paradigm shift in operating system design, as demonstrated in x-kernel, microkernels, exokernels, and TinyOS, can help us to lay the software foundation for reconfigurable end-user systems.**

***Index Terms*—mobile networking, operating systems, wireless access.**

I. Introduction

Operating systems have been in the core of computer science from the very beginning. Therefore, one can ask is there some issues still open. If the answer is yes then one can ask why fifty years have not been enough in solving all relevant issues. Clearly operating systems have evolved a lot during the years. However, changes in operational requirements have changed so that today we need to reconsider even the fundamentals of operating systems.

The need of this reconsideration has its roots in the fundamental changes in usage patterns. Communication and computing devices move; users move and change their devices; (sub)networks in cars, trains and airplanes move; software moves from one execution environment to another. These changes can be characterized as personal networking domains. By personal networking we mean not only body and personal area networks but also protocol aspects of networking in the personal domain, digital home and other smart places like shopping centers, public and private transportation vehicles, ad-hoc communities, and networked (i.e., infrastructure provided) services. In addition, the solutions should also work in a reasonable manner in non-smart places that do not have a high-bandwidth connectivity or any connectivity at all. The ultimate objective is that the solution stacks for different domains are as similar as possible.

All of us hope that we can reuse existing software technology as far as possible. It saves us a lot of money now but may in the future turn out to be extremely expensive. If we are not all the time ready for a revolution, we may miss the train and we may find ourselves at the trap of basing next releases of our products on existing legacy. We do not claim that today is the right time to forget all legacy systems. However, tomorrow it is even more costly to replace them. We should ask ourselves whether or not we want to produce pullovers for dinosaurs although the climate has already started to cool and sooner or later the dinosaurs will disappear.

In this paper we discuss research challenges in operating systems for future end-user systems. We start by stating our assumptions about future mobile applications. Based on the widely accepted vision of wireless future we examine the functional requirements. In Section III we take a look at some milestones in operating systems. We briefly summarize key contributions of x-kernel, microkernels, exokernels, and TinyOS. In Section IV we elaborate the fundamental research issues for future generations of operating systems.

Kimmo Raatikainen is a professor at the Helsinki University Computer Science Department, P.O. Box 68 (Gustaf Hällströmin katu 2b), FIN-00014 UNIVERSITY OF HELSINKI, Finland (kimmo.raatikainen@cs.helsinki.fi). He also works as a part-time research fellow at Nokia Research Center, P.O. Box 407 (Itämerenkatu 11-13), FIN-00045 NOKIA GROUP, Finland (kimmo.raatikainen@nokia.com) and s part-time principal scientist at Helsinki Institute for Inofrmation Technology, P.O. Box 9800 (Tamma-saarenkatu 3), FIN-02015 TKK, Finland (kimmo.raatikainen@hiit.fi).

II. Functionality in Future Mobile Systems

Published visions of the wireless future ([1]-[6]) can be summarized as follows: future applications will be context-sensitive, adaptive and personalized, and future systems will be reconfigurable. These properties have been examined by WWRF [7] in details. Below we briefly elaborate the functional requirements behind the concepts of reconfigurability, context-awareness, adaptability, and personalization.

A. *Reconfigurability*

In essence reconfigurability means that system's hardware and software configurations can seamlessly change in run-time.

The end-user devices of today are primarily integrated units like PDAs, laptops, or mobile phones. However, the situation will change in the future. A personal trusted device, we call it *FuturePhone*, will be the core of the personal networking system. It probes its surrounding looking for suitable peripheral devices such as displays, input devices, processors, fast access memories and access points to communication channels. It dynamically builds up the most appropriate end-user system that can be autoconfigured. The *FuturePhone* also probes for other similar devices in order to establish suitable ad-hoc communities and different kinds of sensors in order to extract context information associated to the current smart place. The *FuturePhone* also tries to detect actuators which provide the means to affect properties of the smart place.

The scenario above implies that the system is able to do device detection and service discovery, as well as hardware and software configuration management.

Efficient device detection and service discovery require that the system is able to monitor the environment and to deliver notifications when something new appears, something old disappears, and something existing changes its state. A notification should only be delivered to those who are interested in it. This, in turn, implies that notifications need to be filtered.

When the state of the systems or its environment changes, then the system needs to decide whether or not the system needs to be reconfigured. This decision engine needs to have a model of its environment (external context) and of its own configuration and capabilities. In other words, the system needs to be self-aware (or reflective). The decision engine also needs a "target function" that indicate how preferable different configurations are to the user. In that sense the target function reflects the personal preferences of the user. It is impossible to assume that all the capabilities and preferences used by the decision engine are complete from the very beginning. Therefore, the models need to be learned, and the system must have learning capabilities.

When the system reconfigures itself, it must maintain its integrity. This requires protection against unauthorized modification. When the system includes a new piece of code into its execution base, it must trust that the code is neither malfunctioning nor introducing undesirable side-effects. An alternative is that the system can verify or validate the code. In addition, the system needs to trust, verify, or validate the pieces of information so that the models used by the decision engine are coherent and reliable. When the system moves into a new administrative domain, it needs to establish a trust relationship in that domain. The establishment needs, at least, mutual authentication, usually also some kind of authorization to use certain resources and services.

B. *Context-Awareness*

Context-awareness means that one is able to use context information. In principle, almost any piece of information available at the time of interaction can be context information. If nobody is utilizing a piece of information, then that piece does not belong to context information. Since the set of running applications is dynamic, the context information is a set the content of which evolve in time.

An application is context-aware if it can extract, interpret and use context information and adapt its functionality to the current context of use. The grand challenge is to create a flexible context modeling framework. The objective is to have efficient means of presenting, maintaining, sharing, protecting, reasoning, and querying context information.

Traditional data management solutions are not sufficient for context-awareness. We can, however, use a database metaphor: all available information is the content of the database, and the pieces in use—that is context information—is the current Database view. The primary difference to the traditional data management is the high rate of changes over time both in the available information and in the context information. Another difference is that context information is, by its nature, distributed and also, to some extent, bound to time and location.

Context reasoning introduces an additional flavor to the requirements: the semantics of context information must be presented in an understandable way. The current Semantic Web and ontologies are insufficient. It is implausible to assume that all necessary terms needed could be defined beforehand. Therefore, context-awareness needs a language that can be used to describe the exact semantics of a new term. The requirement is that a reasoning engine implemented today can understand the semantics of a term to be introduced tomorrow.

C. *Adaptability*

The basic principle of adaptability is simple: the behavior of an application changes when the circumstances change. Conceptually adaptability is quite close to reconfigurability, but we want to keep them separate. In our terminology reconfigurability is a system level concept and adaptability is an application level concept: system reconfigures itself, but an application adapts its behavior.

If an application wants to be able to adapt its behavior according to the changed circumstances, then the system must be able to notify the application of the changes. In some cases the required adaptation can be achieved by changing an algorithm internal to the application. However, in many cases an alternative approach is much more plausible through replacing some components of the application or relocating them. Therefore, an adaptive application may request the system to reconfigure itself or to supply a service from another provider. An alternative is that the system configuration remains the same, but the application components are changed and/or relocated. The user-centric approach requires that all such modifications are automatic, but guided and controlled by user preferences.

In some situations reactive adaptation is not enough. If the connectivity is lost, synchronization, downloads and uploads are impossible. However, if the application had been notified in advance that the connectivity will soon be lost, then the application could have made some proactive preparations. Such proactive actions require predictions that are essential enablers of adaptive applications. A prediction service must support both requests and subscribed notifications. An application should, for example, be able to request the probability of being able to transmit a certain amount of data in a given time. “Inform me as soon as the probability of losing connectivity in the next two minutes exceeds 75%” is an example of a subscribed prediction.

D. *Personalization*

Personalization is considered to be a key enabler for the success of future applications and services. It tackles the phenomenon of information flooding met, already today, almost everywhere. The objective is to increase the usefulness of information and applications by tailoring the content and presentation according to the needs and wishes of the user.

In order to provide personalization, the system needs to be aware of wishes and preference of the user. In mobile and dynamic environments these preferences depend on available capabilities. Enumeration of preferences and capabilities beforehand is a mission impossible. Therefore, the system needs modeling and learning capabilities.

In many respects, the requirements from personalization on software infrastructure are quite similar to those from context-awareness. In personalization, the focus is on extracting user preferences and on utilizing those pieces of information in applications. One essential challenge is to take cultural differences into account.

E. *Summary*

In this Section we have elaborated functional requirements for future mobile systems. The key enablers include:

- environment monitoring,
- device detection and service discovery,
- event notification and filtering,
- hardware and software configuration management
  - auto-configuration
- decisions engines
  - when and how to reconfigure
- modeling and learning capabilities
- maintaining system integrity

Clearly, all these issues are not solved on the operating system level alone. Support form hardware, protocol stack, and middleware is needed to for a feasible solution. Operating system support is needed in monitoring, detection, notifications, configuration management, and system integrity.

III. Some Milestones in Operating Systems

The structure and abstractions (see Fig. 1) in the traditional monolithic operating systems have not practically evolved since the Unix [8]. Other milestones in operating systems include THE [9], “Nucleus” [10], Multics [11], and Hydra [12]. Edsger Dijkstra’s recollections [13] and Fernando Corbato’s retrospective [14] give interesting information about early problems in operating systems.

In the late 1980s the research community introduced the microkernel approach that minimized the size of the kernel and implemented most of operating system services as servers outside the kernel. The first generation of microkernels, such as Amoeba [15], Mach [16], and V [17], introduced too much overhead to be more than academic exercises. However, they laid the foundation for future developments.

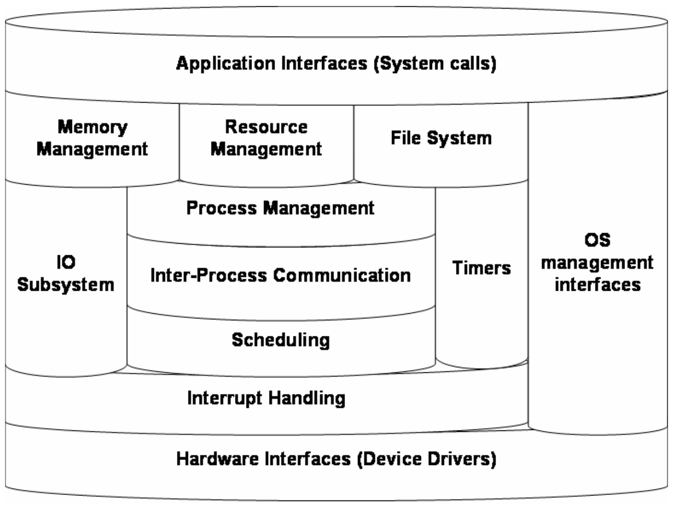

Fig. 1. Structure of traditional operating systems.

The x-kernel [18], [19] at the University of Arizona, L4 kernel [20], [21] at GMD, exokernel [22], [23] at MIT, and TinyOS [24], [25] at UC Berkeley introduced interesting solutions and new concepts. Below we briefly summarize the key ideas in these four systems. A recent survey of customizability in operating system [26] provides additional information.

A. *The x-Kernel*

The x-kernel [18], [19] was designed to facilitate efficient implementation of communication protocols. It included components that manage processes, memory, and communication. The process and memory managers were quite similar to those in other operating systems. The x-kernel supports multiple address spaces so that multiple light-weight processes can execute in each address space. Synchronization of processes within an address space is based on kernel-supported semaphores.

The novel aspect of the x-kernel was the communication manager that provides an object-oriented infrastructure for composing protocols. The communication manager was designed to balance generality (x-kernel can implement a wide variety of network protocols) and efficiency (none of the protocols in the x-kernel suffer severe performance penalties).

The communication manager of the x-kernel provides an object-oriented infrastructure for implementing and composing protocols. The design is based on two abstract communication objects: protocols and sessions. Both objects export a well-defined set of operations implemented by each network protocol the kernel will support. This helped in keeping the protocol interface clean and in eliminating special cases. All protocols assume that another protocol sits above them in the protocol dependency graph.

When a message arrives at a network device, a kernel process shepherds it up through a sequence of protocol and session objects. When a kernel-level protocol invokes a user’s

Demux operation, a shepherd process crosses the boundary and continues executing in user mode. Since the kernel area is part of each address space, no address space switch is needed. When a user process generates a message, the process crosses the user-kernel boundary and shepherds the message down through a sequence of protocol and session objects in the kernel.

The x-kernel demonstrated in practice that a platform for efficient protocol stack is feasible when the operating system is designed on terms of communication needs.

B. L4 μ-Kernel Family

L4 [20], [21] was developed at GMD. Its underlying design philosophy is based on the claim that efficiency and flexibility require a minimal set of general microkernel abstractions and that microkernels are processor dependent. L4 is a lean kernel featuring fast message-based synchronous IPC, a simple-to-use external paging mechanism and a security mechanism based on secure domains. Fiasco [27], [28]–developed at the TU Dresden–implements the L4 interface [29] in C++. Another microkernel in the L4-family is Pistachio [30] from University of Karlsruhe.

The L4 µ-kernel is based on two basic concepts, threads and address spaces. A thread is an activity executing inside an address space. Cross-address-space communication (IPC) is designed to be efficient.

A basic idea of L4 is to support recursive construction of address spaces by user-level servers outside the kernel. The initial address space essentially represents the physical memory. Further address spaces can be constructed by granting, mapping and unmapping logical pages. All address spaces are thus constructed and maintained by user-level servers, also called pagers. Only the grant, map and unmap operations are implemented inside the kernel. I/O ports are treated as parts of address spaces so that they can be mapped and unmapped in the same manner as memory pages.

Hardware interrupts are handled as IPC. The µ-kernel transforms an incoming interrupt into a message to the associated thread. This is the basis for implementing all device drivers as user-level servers outside the kernel. In contrast to interrupts, exceptions and traps are synchronous to the raising thread.The kernel simply mirrors them to the user level.

The main contribution of the L4 team was the demonstration that IPC can be implemented in an efficient manner.

C. Exokernel

Traditional operating systems hide information about hardware resources behind high-level abstractions such as processes, files, address spaces and interprocess communication. A group at MIT designed a new operating system architecture, called exokernel [22], [23], which only securely multiplexes available hardware resources. All traditional operating system abstractions are implemented entirely at application level by untrusted software.

In order to provide applications control over machine resources, an exokemel defines a low-level interface. The design rational is based on the observation that the lower the level of a primitive, the more efficiently it can be implemented, and the more latitude it grants to implementors of higher-level abstractions. This approach also allows separation of protection from management. In contract to the virtual machine approach [31], the exokemel exports hardware resources rather than emulating them.

The exokernel approach is also motivated by the familiar “end-to-end” argument [32]. Applications know better than operating systems what the goal of their resource management decisions should be. Therefore, the applications should be given as much control as possible in resource management decisions.

In order to provide the maximum opportunity for application-level resource management, the exokernel architecture consists of a thin exokemel core that multiplexes and exports physical resources securely through a set of low-level primitives. Library operating systems implement higher-level abstractions.

The design challenge is to give library operating systems maximum freedom in managing physical resources while protecting them from each other. In separating protection from management, an exokemel performs three important tasks: 1) tracking ownership of resources, 2) ensuring protection by guarding all resource usage or binding points, and 3) revoking access to resources. To achieve these tasks, an exokemel employs three techniques. First, using secure bindings, library operating systems can securely bind to machine resources. Second, visible revocation allows library operating systems to participate in a resource revocation protocol. Third, an abort protocol is used by an exokemel to break secure bindings of uncooperative library operating systems by force.

The MIT exokernel team laid the foundation but the primary contribution later came from University of Cambridge. The Nemesis [33] and particularly the Xen [34], [35] demonstrated that paravirtualization introduces only a small overhead, typically 2-4%.

D. TinyOS

TinyOS [24] is a very small microthreaded operating system design for netwoks of sensor nodes (SmartDust [36]). The TinyOS is usually accompanied by Maté [25], a really minimalistic virtual machine. TinyDB [37] and the programming language nesC [38] belong to the same solution family.

The TinyOS draws on previous architectural work on lightweight thread support and efficient network interfaces. According to the design team, the core challenge they faced was the operating system framework that retains the characteristics of the hardware design by managing the hardware capabilities effectively, while supporting concurrency-intensive operation in a manner that achieves efficient modularity and robustness.

TinyOS solves the design challenge by selecting an event model so that high levels of concurrency can be handled in a very small amount of space. The TinyOS also adopts the two-level scheduling structure from CILK [39] so that a small amount of processing associated with hardware events can be performed immediately while long running tasks are interrupted.

A complete system configuration consists of a tiny scheduler and a graph of components. A component has four interrelated parts: a set of command handlers, a set of event handlers, an encapsulated fixed-size frame, and a bundle of simple tasks. Tasks, commands, and handlers execute in the context of the frame and operate on its state. To facilitate modularity, each component also declares the commands it uses and the events it signals.

Commands are non-blocking requests made to lower level components. Typically, a command will deposit request parameters into its frame and conditionally post a task for later execution.

Event handlers are invoked to deal with hardware events, either directly or indirectly. The lowest level components have handlers connected directly to hardware interrupts, which may be external interrupts, timer events, or counter events. An event handler can deposit information into its frame, post tasks, signal higher level events or call lower level commands.

Maté is a tiny communication-centric virtual machine designed for sensor networks. Its high-level interface allows complex programs to be very short, usually under 100 bytes. Code is broken up into small capsules of 24 instructions, which can self-replicate through the network. Maté is a bytecode interpreter. It is a single TinyOS component that sits on top of several system components. There are two stacks: an operand stack and a return address stack. Most instructions operate solely on the operand stack. Maté has three execution contexts that can run concurrently at instruction granularity.

The Tiny solution family clearly indicates that a small and efficient operating system and development environment can be build. The primary lesson is similar to that from the x-kernel: Design the operating system for a specific target system. In addition, the Maté shows that a tiny virtual machine is useful for installing new applications.

E. *Summary*

The four examples presented above clearly demonstrate the advantages of a paradigm shift in operating system design. Other significant experiments in operating system research include operating system frameworks (Flux OS Kit [40], Choices [41], Pebble [42]), portal-based operating systems (Kea [43], Space [44]), reflective operating systems (MetaOS [45]) and dynamically adaptive operating systems (Synthetix [46]). Additional interesting results have been in the area of software protection (Software Fault Isolation [47], Proof-Carrying Code [48]).

IV. Some Research Challenges in Operating Systems

As already discussed the need of reconsideration comes from the new operational requirements: reconfigurability, context-awareness, adaptation, and personalization. The problems with the current plug-and-play are a clear indication that new approaches are needed. Our claim is that we need a paradigm shift: ***forget the end-user terminal and start thinking about end-user systems!***

The fundamental challenge is that we must tackle reconfigurability and adaptation issues on hardware level and on all levels of software (operating system, protocol stack, middleware, applications). We must also remember that the system needs to be usable anywhere, anyhow, anytime, and by everybody. This implies that the end-user systems will also contain carry-on, battery-powered devices.

In the research space of operating systems we have identified four key issues: self-awareness, detection-notifications, system integrity, and power management. In this Section we briefly summarize the current state-of-the-art and highlight some research topics in these four areas.

A. *Self-Awareness*

In order to be reconfigurable, the system needs to be self-aware. In other words, the system needs to understand its own state; the operating system must understand its own state and configuration, and the state and the configuration of the resources it is controlling and managing.

The understanding requires a model of hardware and operating system level software configuration. This model, or these models, will be used to describe the configurations and to reason about the configurations.

Operating systems have such models, but their use is limited to operating system's internal bookkeeping. Particularly in reasoning, the operating system needs to pass the model to other software.

B. *Detection and Notifications*

In order to reconfigure a system, the system needs to detect new devices/subsystems and services that have come available. The system needs also to detect the subsystems or services that have disappeared, or whose properties have significantly changed.

Issues in this area include interrupt handling and event notifications. We need also revisit many optimization techniques currently used in operating systems. For example, lazy update should not be used if we must take into account that the persistent storage subsystem may disappear.

Most of detection in the operating system level is done in interrupt handlers and device drivers. Interrupt handling is also affected by sensors. When a carry-on device provide motion detection, we will get interrupts very frequently. The traditional way of handling interrupts would consume too much CPU cycles.

An event system—such as CORBA Notification Service [49], Java Message Service [50] Siena [51]—is a typical middleware service to deliver notifications. In current operating systems event notifications are typically delivered in one-to-one communication pattern. When more than one subscriber is interested in an event, scheduling should take into account the timeliness requirements of event notifications.

C. *System Integrity*

The success of the future Internet will totally depend on consumers' trust. The current Internet is vulnerable to worms, viruses, spam and fraud. It clearly demonstrates that "*fix it later*" does not work. Security, trust and privacy must be addressed from the very beginning of system design and on all levels: hardware, operating system, protocols, middleware.

Authenticated boot is a fundamental enabler for system integrity. Trusted Computing Group [52] has specified one way of providing the necessary chains of trust. In dynamically reconfigurable systems, chains of trust must be re-established each time the configuration changes. How this can be done efficiently enough is a research challenge. The re-establishment of the chains of trust is also needed each time a new piece of software is to be included. The Proof-Carrying Code [48] is a prominent way of verifying what the received piece of code will do. Alternatives include various sandboxing techniques.

Without sufficient hardware support, system integrity cannot be provided. For example, most game applications run their own logic on display processor. If the display processor is DMA-enabled, hardware support is necessary to prevent the application code from modifying internal data structures of operating system.

D. *Power Management*

In future systems, available energy, as embodied by the system battery, will have an increasingly important role. Despite wide-spread recognition of the importance of energy, current operating systems do not provide application developers a convenient abstraction of the energy resource. There have been broad efforts to better manage the energy use of individual devices, but there has been relatively little attention to managing energy as a resource.

In order to manage energy, the operating system must have a model for power consumption. There are two primary problems in addressing specific energy-related goals in operating system level. The first one is to develop resource management policies. The second one is related to adaptation in application behavior.

The Smart Battery interface in the ACPI specifications [53] is a fundamental enabler for operating system to manage the battery resource. It allows the system to query the status of the battery. However, the operation of querying the interface is too slow to be useful for gathering power consumption data at a sufficiently fine grain for resource management functions without introducing unacceptable overhead.

Energy management at the operating system level can be viewed in two dimensions.

1) There are a wide variety of devices in the system that concurrently consume power and that are amenable to very different management techniques. 2) The devices in the system are shared by multiple applications. It is a hard problem to accurately attribute power consumption to the correct processes. However, solving this accounting problem is a prerequisite to managing the battery resource. It involves three fundamental issues. First, we must understand the nature and determining the level of resource consumption. Second, we must appropriately be able to charge for use of the various devices in the system. Finally we must be able to attribute those charges to the responsible entity.

The special issue on power-aware embedded computing [54] addresses the system-level design gap between a given algorithm specification and the selected target architecture platform. Some of the fundamental research issues identified in the issue include:

- Which decisions and optimizations should be statically defined?
- Which require a synergy between both approaches?
- How can such decisions and techniques adapt to dynamically varying working conditions and/or performance targets?
- What is the cost-effectiveness of the required hardware assists (if any)?
- Which components, features, and parameters of a system architecture should be statically tuned or specialized to the needs and requirements of a target embedded application so as to improve energy efficiency?
- Which ones should be dynamically reconfigurable and/or adaptive?
- Which fundamental characteristics of an embedded application influence or impact the above design choices?

V. Conclusions

We have shown that a paradigm shift in operating system design is necessary to meet the needs of future end-user system. We proposed to take reconfigurability as the main concern. This will be a similar to the design principles in x-kernel [18] and TinyOS [24].

If we are not all the time ready for a revolution, we may miss the train and we may find ourselves at the trap of basing next releases of our products on existing legacy. We do not claim that today is the right time to forget all legacy systems. However, tomorrow it is even more costly to replace them. We should ask ourselves whether or not we want to produce pullovers for dinosaurs although the climate has already started to cool and sooner or later the dinosaurs will disappear.

In order to support reconfigurability, the operating system research must solve several research issues related to self-awareness, detection and notifications, system integrity, and power management. The fundamental challenge is to find a reasonable balance of solutions in the areas of hardware, system software (operating system, protocol suite, middleware), and development tools.

References

[1] M. Weiser, “The Computer for the Twenty-First Century,” *Scientific American*, September 1991, pp. 94-104.

[2] M. Weiser, “Some Computer Science Issues in Ubiquitous Computing,” *Communications of the ACM*, July 1993, pp. 74-84.

[3] R. Bagrodia, W.W. Chu and L. Kleinrock, “Vision, Issues, and Architecture for Nomadic Computing,” *IEEE Personal Communications*, December 1995, pp. 14-27.

[4] M. Satyanarayanan, “Pervasive Computing: Vision and Challenges,” *IEEE Personal Communications*, August 2001, pp. 10-17.

[5] K. Ducatel et al., *Scenarios for Ambient Intelligence in 2010*. Technical report, ISTAG, February 2001.

[6] G. Banavar, J. Barton, N. Davies and K. Raatikainen, “Special feature on middleware for mobile & pervasive computing,” *ACM SIGMOBILE Mobile Computing and Communications Review*, October 2002, pp. 16-24.

[7] R. Tafazolli, Ed., *Technologies for the Wireless Future*. Wiley, 2005.

[8] D. M. Ritchie and K. Thompson, “The UNIX Time-Sharing System,” *Communications of the ACM*, July 1974, pp. 365-375.

[9] E. W. Dijkstra, “The Structure of the “THE” Multiprogramming System,” *Communications of the ACM*, May 1968, pp. 341-346.

[10] P. Brinch Hansen, “The Nucleus of a Multiprogramming System,” *Communications of the ACM*, April 1970, pp. 238-250.

[11] E. I. Organick, *The Multics System: An Examination of its Structure*. MIT Press, 1972.

[12] W. Wulf et al., “Hydra: The kernel of a multiprocessing operating system,” *Communications of the ACM*, July 1974, pp. 337-345.

[13] E. W. Dijkstra, “My Recollections of Operating System Design,” Manuscript EWD1303 (Oct. 2000/Apr. 2001). Reprinted in *ACM SIGOPS Operating Systems Review*, April 2005, pp. 4-40.

[14] F. J. Corbató, “On building systems that will fail,” *Communications of the ACM*,” September 1991, pp. 72-81.

[15] A. S. Tanenbam et al., “Amoeba System,” *Communications of the ACM*,” December 1990, pp. 46-63.

[16] D. Golub, R. Dean, A. Forin and R. Rashid, “Unix as an application program,” *Proceedings of the Usenix Summer Conference* (Anaheim, Calif., June 1990). Usenix Association, 1990, pp. 87–96.

[17] D. R. Cheriton, G. R. Whitehead and E. W. Sznyter, “Binary emulation of Unix using the V kernel,” *Proceedings of the Usenix Summer Conference* (Anaheim, Calif., June 1990), pp. 73–86.

[18] L. Peterson, N. Hutchinson, S. O’Malley and H. Rao, “The x-kernel: A Platform for Accessing Internet Resources,” *IEEE Computer Magazine*, May 1990, pp. 23-33.

[19] N. C. Hutchinson and L. L. Peterson, “The x-Kernel: An Architecture for Implementing Network Protocols,” *IEEE Transactions on Software Engineering*, January 1991, pp. 64-76.

[20] J. Liedtke, “On µ-Kernel Construction,” *Proceedings of the 15th ACM Symposium on Operating System Principles*, December 1995, pp. 237-250.

[21] H. Härtig et al., “The Performance of µ-Kernel-Based Systems,” *Proceedings of the 16th ACM Symposium on Operating System Principles*, October 1997.

[22] D. R. Engler et al., “Exokernel: An Operating System Architecture for Application-Level Resource Management,” *Proceedings of the 15th ACM Symposium on Operating System Principles*, December 1995, pp. 251-266.

[23] M. F. Kaashoek et al., “Application Performance and Hexibility on Exokernel Systems,” *Proceedings of the 16th ACM Symposium on Operating System Principles*, October 1997, pp. 52-65.

[24] J. Hill et al., “System Architecture Directions for Networked Sensors,” *Proceedings of ASPLOS 2000*, ACM, November 2000, pp. 93-104.

[25] P. Levis and D. Culler, “Maté: A Tiny Virtual Machine for Sensor Networks,” *Proceedings of ASPLOS X*, ACM, October 2002, pp. 85-95.

[26] G. Denys, F. Piessens and F. Matthijs, “A Survey of Customizability in Operating Systems Research,” *ACM Computing Surveys*, December 2002, pp. 450–468.

[27] M. Hohmuth and H. Härtig, “Pragmatic Nonblocking Synchronization for Real-time Systems,” *Proceedings of the 2001 USENIC Annual Technical Conference*.

[28] M. Hohmuth, *The Fiasco kernel: Requirements definition*. Technical Report TUD–FI–12, TU Dresden, December 1998. http://os.inf.tudresden.de/fiasco/doc.html.

[29] J. Liedtke, *Lava Nucleus (LN) Reference Manual*. Technical Report. IBM Thomas J. Watson Research Center. 1998.

[30] *The L4Ka::Pistachio Microkernel* (white paper). System Architecture Group, University of Karlsruhe, May 1, 2003.

[31] R. J. Creasy, “The Origin of the VM/370 Time-Sharing System,” *IBM J. Research and Development*, September 1981, pp. 483-490.

[32] J. H. Saltzer, D. P. Reed and D. D. Clark, “End-to-End Arguments in System Design,” *ACM Transactions on Computer Systems*, November 1984, pp. 277-288.

[33] D. Reed, A. Donnelly and R. Fairbairns, eds., “Nemesis Kernel Overview,” http://www.cl.cam.ac.uk/Research/SRG/netos/old-projects/pegasus/publications/overview/brief-overview.html.

[34] P. Barham et al., “Xen and the Art of Virtualization,” *Proceedings of the 19th ACM Symposium on Operating System Principles*, October 2003, pp. 164-177.

[35] K. Fraser et al, “Safe Hardware Access with the Xen Virtual Machine Monitor,” OASIS ASPLOS 2004 workshop, Boston, Mass., October 9, 2004. http://www.cl.cam.ac.uk/Research/SRG/netos/papers/2004-oasis-ngio.pdf.

[36] B. Warneke, M. Last, B. Liebowitz and K. S. J. Pister, “Smart Dust: Communicating with a Cubic-Millimeter Computer,” *IEEE Computer Magazine*, January 2001, pp. 44-51.

[37] S. Madden, M. J. Franklin, J. M. Hellerstein and Wei Hong, “TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks,” *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*. December 9-11, 2002.

[38] D. Gay et al., “The nesC Language: A Holistic Approach to Networked Embedded Systems,” *Proceedings of PLDI'03*, June 9–11, 2003.

[39] R. D. Blumofe, “Cilk: An efficient multithreaded runtime system.,” *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995, pp. 207-216.

[40] B. Ford et el., “The flux OSKit: A substrate for kernel and language research,” *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 1997, pp. 38–51.

[41] R. Campbell et al., “Designing and implementing Choices: an object-oriented system in C++,” *Communications of the ACM*, September 1993, pp. 117–126.

[42] K. Magoutis et al., ”Building appliances out of reusable components using pebble,” *Proceedings of the 9th ACM SIGOPS European Workshop*, 2000.

[43] A. Veitch and N. Hutchinson, “Kea—a dynamically extensible and configurable operating system kernel,” *Proceedings of the 3rd Conference on Configurable Distributed Systems*, 1996.

[44] D. Probert, J. Bruno and M. Karzaorman, “Space: a new approach to operating system abstraction,” *Proceedings of the International Workshop on Object Orientation in Operating Systems*, 1991, pp. 133–137.

[45] M Horie et al., ”Using meta-interfaces to support secure dynamic system reconfiguration,” *Proceedings of the 4th International Conference on Configurable Distributed Systems*, 1998.

[46] C. Pu et al., “Streamlining a commercial operating system,” Proceedings of the 15th ACM Symposium on Operating System Principles, 1995.

[47] R. Wahbe et al., ”Efficient software-based fault isolation,” *ACM SIGOPS Operating Systems Review*, December 1993, pp. 203–216.

[48] G. C. Necula and P. Lee, “Safe kernel extensions without run-time checking,” *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, 1996, pp. 229–243.

[49] *CORBA Notification Service 1.1*, OMG document formal/2004-10-11, October 2004.

[50] Sun Microsystems, *Java Message Service Documentation*, June 2001.

[51] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf, "Design and Evaluation of a Wide-Area Event Notification Service," *ACM Transactions on Computer Systems*, August 2001, pp 332-383.

[52] Trusted Computing Group, *TCG Specification: Architecture Overview*, Revision 1.2, 28 April 2004, https://www.trustedcomputinggroup.org/downloads/TCG_1_0_Architecture_Overview.pdf

[53] Hewlett-Packard Corporation et al., *Advanced Configuration and Power Interface Specification*, Revision 3.0, September 2, 2004.

[54] M. Jacome and F. Catthoor (eds), “Special issue on power-aware embedded computing,” *ACM Transactions on Embedded Computing Systems*, August 2003.