Technical Report

CoS: A New Perspective of Operating Systems Design for the Cyber-Physical World

Vikram Gupta

Eduardo Tovar

Nuno Pereira

HURRAY-TR-120703

Version:

Date: 7/4/2012

CoS: A New Perspective of Operating Systems Design for the Cyber-Physical World

Vikram Gupta, Eduardo Tovar, Nuno Pereira

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail: vigu@isep.ipp.pt, emt@dei.isep.ipp.pt, nap@isep.ipp.pt

http://www.hurray.isep.ipp.pt

Abstract

Our day-to-day life is dependent on several embedded devices, and in the near future, many more objects will have computation and communication capabilities enabling an Internet of Things. Correspondingly, with an increase in the interaction of these devices around us, developing novel applications is set to become challenging with current software infrastructures. In this paper, we argue that a new paradigm for operating systems needs to be conceptualized to provide aconducive base for application development on Cyber-physical systems. We demonstrate its need and importance using a few use-case scenarios and provide the design principles behind, and an architecture of a co-operating system or CoS that can serve as an example of this new paradigm.

CoS: A New Perspective of Operating Systems Design for the Cyber-Physical World

[Forward-looking Paper]

Vikram Gupta[†‡], Eduardo Tovar[†], Nuno Pereira[‡]
[†]CISTER Research Center, ISEP, Polytechnic Institute of Porto, Portugal
[‡]Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, USA
vikramg@ece.cmu.edu, {emt, nap}@isep.ipp.pt

***Abstract*—Our day-to-day life is dependent on several embedded devices, and in the near future, many more objects will have computation and communication capabilities enabling an Internet of Things. Correspondingly, with an increase in the interaction of these devices around us, developing novel applications is set to become challenging with current software infrastructures. In this paper, we argue that a new paradigm for operating systems needs to be conceptualized to provide a conducive base for application development on Cyber-physical systems. We demonstrate its need and importance using a few use-case scenarios and provide the design principles behind, and an architecture of a *co-operating system* or *CoS* that can serve as an example of this new paradigm.**

I. Introduction

The penetration of embedded-systems in our daily lives is increasing at a tremendous rate, and even today, a human being can have tens of devices around him that have computational capabilities. In the future, not only the number of such devices is set to increase further, their capability to communicate among themselves and accomplish complex and distributed logic will become more widespread. In addition to the current *smart* devices such as mobile-phones, music players and tablets, even the *dumb* devices such as lamps, tables and chairs may have computation capabilities and contribute to ambient intelligence. This possible trend has led researchers in academia and industry to foresee an *Internet of Things*, where all (or most) of the objects will be connected to each other and the internet. Such a highly connected world will further enable several applications like home automation, intelligent ambience, green buildings and so on. However, full potential of highly-connected cooperating objects is still difficult to perceive, as there is scope for diverse and revolutionary applications that may not have been conceived yet.

To enable the development of such new applications, new paradigms for embedded-systems software are required. We believe that the currently available operating systems and programming abstractions may not encourage an environment for active application development for future networked embedded systems. In this paper, we argue that the design of the operating systems for networked embedded systems needs to be thought from a different perspective than the one already taken in the popular solutions like TinyOS [1], Contiki [2], Nano-RK [3] etc. Most of the popular research works in the direction of facilitating programming on sensor networks assume that the existing operating systems are the *de-facto* platforms upon which the middleware or the programming abstractions have to be built. This assumption needs to be thought again from a top-down perspective where the new goal is to support dynamic deployment and management for network-level applications.

Existing operating systems were designed to ease the programming of specific hardware that was developed as prototypes for wireless sensor networks. Programming these devices on *bare-metal* is complex and requires high degree of expertise in embedded systems. Platforms like MicaZ and TelosB are resource-constrained yet powerful-enough devices that can easily support a small operating system, custom communication stacks and one or more applications. Operating systems were designed from the perspective of easing the application development process on individual devices because even in their standalone operation they are complex systems with a processor, a radio, several sensors, a programming/communication interface over the USB or the serial port and so on. These hardware and software platforms have contributed a lot towards the development of ground-breaking research and proof-of-concept ideas. Moreover, the research in these areas provided a vision for the future of networked embedded systems. To achieve the goal of ubiquitous connectivity of embedded devices described earlier, there is a need to design (distributed) operating systems from scratch that completely isolate the users from node-level intricacies, and take the application development to a higher level where the whole network ecosystem can be viewed as a single organism. We believe that revamping the way operating systems are designed is a first step towards this goal.

By networked embedded systems we refer to the broader area of Cyber-Physical Systems (CPS) that react to the environment in addition to just sensing the physical quantities as in the case of wireless sensor networks. Timeliness is an important requirement of CPS, because of the tight integration of sensing and actuation. We believe that it is time we move from an operating system to a *co-operating system* or *CoS*, that embodies all fundamental functionalities necessary for encouraging application development for networked embedded systems directly above it. *CoS* is a truly distributed operating system, in the way that it provides a geographically distributed view of the operating system to the user rather than abstracting the network as a single machine. In the rest of this paper, we

describe a few key principles that can motivate the design of such a cooperating-system, and we propose a possible architecture that can satisfy those principles.

II. State of the Art

Many solutions have been designed that aim to provide an environment for convenient application development for networked embedded systems. From the perspective of allowing the development of diverse applications on cyber-physical systems, we classify them into three major classes.

A. Operating Systems

Earlier operating systems and even the more recent ones provide several convenient abstractions for programming the hardware. The popular sensor network operating systems like Contiki, TinyOS, etc., all allow one or more applications to be developed for hardware platforms, and the network-level coordination is the responsibility of the application programmer. These operating systems facilitated and supported computer scientists familiar with programming of general-purpose computers to develop applications for embedded hardware.

Some newer operating systems like LiteOS [4] provides a UNIX-like interface for sensor networks and each device can be accessed or written-to like a file. HomeOS [5], [6] allows connectivity of heterogenous devices such that a typical user can develop complex logic using the appliances in a modern home. HomeOS is a centralized design that connects the deployed devices and provides an interface for configuring the devices according to the needs of the users, based on access privileges and time of the day, for example. HomeOS is an interface above the home automation infrastructure and is closer to being a middleware-layer rather than an OS.

B. Middleware and Abstractions

Facilitating the development and the deployment of applications on heterogenous sensor networks has been a key driver behind the design of several middleware and programming abstractions proposed in the past. Most of the solutions discussed in the recent survey by Mottola and Picco [7] allow programming the network as a whole, while abstracting the user from lower-level complexities. Several different solutions have been proposed that serve varied goals, but anecdotal evidence suggests that almost none of those have been adopted in actual deployments or test-beds other than those for which these systems were built. When a new application is conceived, researchers typically find it more convenient to develop their own middleware/programming framework on top of a popular operating system to deploy and test their application instead of using an existing solution. The reasons behind this relatively less enthusiastic adoption of middleware can be several. Visibility of the solution, maturity of the software, hardware platforms supported and application domains covered, are few such factors that dictate the popularity of a middleware or a programming abstraction.

In addition to those, developers generally are not able to place confidence in third-party middleware for their applications because of the lack of robustness and support. Debugging may eventually require delving into the middleware code and its interactions with the underlying operating system. We aim to design a co-operating system to overcome these limitations, such that all the functionality of the underlying hardware and the possible interactions of devices can be visible to the user-interface, while providing easy development.

C. Standards

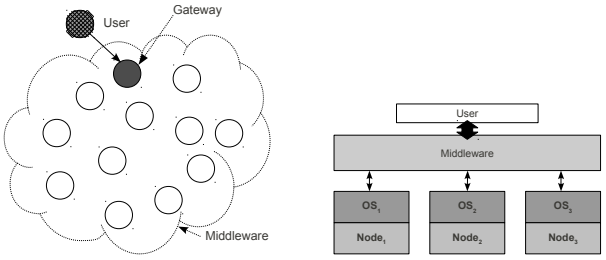
Several standards are being promoted by the industry and academia to foster interoperability between various appliances at home, and hence allow development of applications. Examples of such protocols are DLNA [8], Z-Wave [9] and OSIAN [10]. These standards can be great contributors towards distributed application development for appliances, and any programming system should be able to leverage these standards for communicating with heterogenous devices.

III. Use-case Scenarios

The operating system we envision (propose) should be generic-enough for most of the cyber-physical applications and should allow rapid application development as well. We consider the following two broad application scenarios, on which we can base the design decisions behind a *CoS*.

Intelligent surroundings (Distributed Applications): Cyber-physical systems embody interacting (cooperating) objects, where based on sensed inputs by one or more devices, a distributed action might be taken. We take the example of a conference room in an office building with several chairs, an LCD screen and lighting infrastructure. We also assume that all these devices (the chairs, the lights and the screen) have sensing and actuation capabilities, a computation platform capable of running a *CoS*, and compatible radio-transceivers. Several applications can be conceived on this infrastructure. For example: *i)* adjusting the brightness and color temperature of the LCD screen based on the distance of the chairs from the screen and properties of the ambient light, *ii)* turning the lights on/off based on the occupancy of the room, and *iii)* turning on the heat in the chairs based on the occupant-preferences and so on. Considering that these devices could be manufactured by different vendors, developing distributed applications is challenging. A *CoS* should provide a conducive environment where applications can be deployed across diverse devices with different capabilities. End-to-end device connectivity, application-code dissemination are some of the basic functionalities that should be provided, and higher level device interactions should be specified by the programmer.

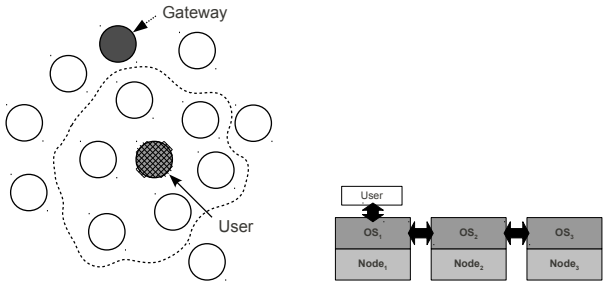
Communicating Vehicles (Dynamic Topologies): A relevant example of connected devices in dynamic topologies is a set of cars at an intersection that are stuck in a traffic jam. In such a scenario, where different vehicles meet at a junction for brief periods, having a centralized middleware or a common abstraction for directing them to carry out a certain goal may or may not be possible. A practical solution can be a modern operating system, that has coordination of dynamic-topologies as a basic functionality, and allows programming this ecosystem in a non-centralized way. A traffic policeman



(a) User interaction through a middleware

(b) Layered diagram showing the user interaction

Fig. 1. Programming with the help of a middleware, to emphasize the centralizing aspect



(a) User interaction through a *CoS*

(b) Layered diagram showing application dissemination using *CoS*

Fig. 2. Programming using a *CoS*

can now simply broadcast a traffic-jam resolving algorithm to all cars for that situation. For example, the policeman can command the cars arriving at the intersection from the north lane to start going into the west lane and so on. The operating system running on the vehicles' will allow the reception of such commands, and take useful action like notifying the driver or driving accordingly if it is an autonomous vehicle.

IV. Design Principles

The motivation behind *CoS* is driven by the observation that the existing operating systems were only designed for facilitating node-level application development, and middleware solutions designed to allow *macro-* or network-level programming are limited in scope and are highly dependent on the underlying operating systems. We discuss some of the principles that stress on the need for a new perspective in the design of operating system for cyber-physical systems.

A. *Programming using* CoS

Traditional network programming approaches typically involve a middleware or a programming abstraction that inevitably tends to centralize the network topology. A user has to interact with a programming layer, that generally resides on a central server or a gateway. As shown in Figure 1, the middleware abstracts the network complexities from a user with the help of a hierarchical setup that can be rigid and highly application specific.

In contrast, *CoS* is designed to facilitate application development in a distributed way for networked objects of the future. The programmer interacts with the operating system directly for creating network-level applications, instead of a middleware or a gateway. *CoS* makes it possible that the user can interact with the system at any logical or topological location in the network, as shown in Figure 2(a). *CoS* manages the communication and dissemination of application program to other nodes, based on the user requirements embedded in the logic of the application. The communication between the nodes can be transparent to the user, and the interaction among them is dictated by the application logic. The user may deploy an application over one or more nodes, each running an instance of *CoS*. Then the application is distributed to other participating nodes by *CoS* as exemplified in Figure 2(b).

B. *Truly Distributed Design*

Traditional distributed operating systems were designed from a perspective of abstracting away the presence of more than one machine from the user. According to Tanenbaum *et al.* in [11]:

> *"As a rule of thumb, if you can tell which computer you are using, you are not using a distributed system. The users of a true distributed system should not know (or care) on which machine (or machines) their programs are running, where their files are stored, and so on."*

This presents a major disconnect from cyber-physical systems, where the devices are not only logically distributed but geographically as well, and more often than not, it is important for the applications to associate the geographical location in their logic. For example, controlling the window blinds based on light level readings in specific rooms *A, B and C*. Most middleware and programming abstractions tend to centralize the network, and cause overheads in maintaining connectivity to all the nodes and may also involve participation of the nodes that may otherwise not be required. A middleware would require a hierarchical architecture to allow deployment of applications. Similar to the example of vehicles at a junction, the nodes may not provide enough support for an active higher layer in dynamic topologies. Hence, the operating system executing on the nodes needs to allow application deployment in a decentralized way, and then execute distributed logic. A distributed operating system for cyber-physical systems (logically and geographically) can decentralize the operation of the network. It should allow a user to deploy applications by connecting to any one or more nodes in the network. *CoS* should obviate the requirement of having a middleware and/or a programming abstraction to program the network.

C. *Other Key Features*

Modular: *CoS* should be modular in design supporting dynamic loading-unloading of modules and application. This can help a programmer create powerful applications with significant ease by making use of existing modules, or creating new ones. Modules can either reside on the system flash memory, or can be delivered over-the-air, if needed.

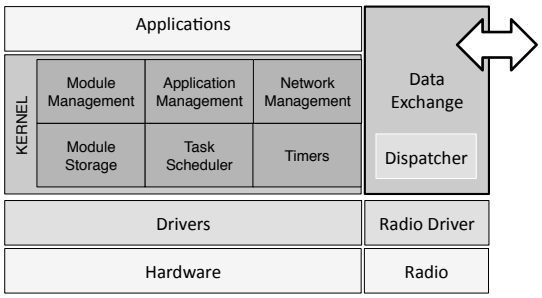


Fig. 3. Typical architecture of a *CoS*

Integrated Network Management: *CoS* should support tightly integrated network management, such that a device is able to discover its neighboring devices and thus allowing updating of routing information. This information should be made available to the programming interface to network-wide application development.

Programming Interface: The traditional way of programming sensor nodes via a direct one-to-one connection to a computer, is designed to facilitate application development while making good use of on-board peripherals. The programming interface of the proposed operating system should be at the network-level by design, rather than being a layer on top of node-level programming abstraction. The programming interface should have information about the network topology and capability of the nodes to provide a global view to the programmer in an intuitive way.

Isolation of Applications: Multiple independent applications on a network of nodes should be supported, while making sure that the data exchange and operation of the applications remains isolated. The data may need to be multiplexed in the network to save energy, and then demultiplexed to deliver to each user.

Support for Heterogenous Platforms: The real-world applications of sensor networks especially in the context of cyber-physical systems may require diverse hardware, including processor, sensors, actuators and communication peripherals. The operating system should be designed such that it supports this varied ecosystem of hardware, and provides suitable programming provisions.

V. *CoS* Architecture

After discussing the design features of *CoS*, we can now describe its architecture. An outline of the architecture showing various components is provided in Figure 3. The following important components constitute *CoS*: Kernel, drivers, exchange plane and the applications on top. We will describe each of those in detail next.

A. Applications

One of the key motivating principles behind the design of *CoS* is easy and convenient deployment of applications directly on top of the *operating system*, rather than having a network-wide middleware or a programming framework. The status-quo in the programming of networked embedded systems involve either copying operating system images with embedded applications onto to the flash, or using a network-level virtual-machine delivery system. Following from the typical trend of writing applications on top of an OS for general-purpose computing systems, *CoS* should allow installing applications at runtime, without the need of a middleware. Given the resource-constrained nature of the embedded systems, a programmer can create and compile applications on a PC, and then deliver the binaries to *CoS*.

As explained earlier, the user may not need to depend on a gateway or a central-server to deploy the application, *CoS* allows any one or more nodes to act as *point of delivery*. The distribution of an application to participating nodes is handled by the data-exchange plane. The kernel provides a *mount-point* to the newly deployed application and adds its information in a local list. The mount-point is a pointer to the memory location where the application resides, so that it can be executed according to the scheduling policy of *CoS*, and the criticality or the priority of the application.

Each application has to specify a scope, both geographic and logical, to help determine the nodes to be associated with that application. In case of more than one application submitted by independent users, *CoS* ensures isolation between the state of the applications both at node- and network-level, thus making sure that the data is delivered to the intended destination in a seamless manner and appropriate action is taken in case of *sense-and-react* applications. This may require conflict resolution or deadlock avoidance support from the kernel. For example, in an intelligent surroundings scenario, if one application requires lights off in the night, and another requires the lights to be turned on if the window shades are down, it may happen that they can be in conflict at some point in time. This conflict has to be resolved among participating applications, and this responsibility lies with the kernel.

B. Kernel

The kernel handles the core functionality, including managing the applications, task-scheduling and timing. Scheduling the applications is one of the most important functions of the kernel. The underlying hardware platform may be significantly resource-constrained that may not support more than a certain number of applications, and the resource usage of applications have to be limited within certain bounds. The kernel ensures that the applications do not misbehave, and their timing requirements are met in the best manner possible. For this purpose, kernel from the NanoRK [3] operating system can be adapted, as it has support for real-time scheduling and task-level resource reservations. In addition to these optimizations, the kernel should be able to resolve conflicts in case of dissimilar requirements of applications. The kernel can make use of priorities, or assign default behaviors to the peripherals. In the example of conflict in the state of lights provided earlier, the kernel may choose to keep the lights off at night, until over-ridden manually.

The kernel should be modular in design so that drivers or other modules can be added to enable required functionalities.

Cyber-physical systems can consist of varied sensor and actuator peripherals, and providing out-of-the-box support for such possibly large number of devices may not be practical. Programmers or users should be able to install modules on the nodes covered by their applications. The kernel should allow dynamic loading and unloading of modules in a manner similar to the SOS [12] operating system. The kernel can achieve this with the help of module management and storage components.

As *CoS* may be run on battery-powered devices, minimizing the power consumption is important. A power-management module tries to put the device to sleep for as long as possible. Nodes may operate at very low duty-cycles, hence the power-management module can ensure that different applications execute in way to maximize the sleep interval.

C. Drivers

Hardware support for the peripherals on a node, including the radio, the sensors and the actuators, is provided through drivers. In addition to the default drivers available with *CoS*, drivers can be loaded as modules at the runtime. Such design allows easy integration of heterogenous devices and dynamic behavior in the long-term. The operating system does not need to be *flashed* again if some peripheral devices are added or removed. In addition to the peripherals, drivers can help applications to configure the communication layer as well. Radio configuration, medium-access control and routing can be implemented as modules and changed on-the-fly, if needed.

D. Exchange Plane

One of most important components of the *CoS* architecture is the data-exchange plane. The data-exchange plane handles all the communication to and from the node. Applications created by the user are delivered to the nodes through this plane, and are further relayed to other nodes that participate in the given application. Other responsibilities of the data-exchange plane are ensuring isolation between the applications, delivering data to the nodes involved, and also directing actuation based on the distributed logic of an application.

The data-exchange plane uses information from the network management module in the kernel about the topology and routing information in order to maintain the communication across a multi-hop network. It can use a device-advertisement phase to construct a topology map of the system. The advertisements allow the exchange-plane to maintain information about the capabilities of the neighboring nodes. The radius of the neighborhood may be pre-decided as a design-parameter or specified by the applications. Developing an application may require knowledge about the capabilities of the devices in the network and hence, the advertisements available to the data-exchange plane should be provided to the programmer so that a distributed logic can be implemented, in accordance with the truly distributed design principle explained in Section IV-B.

The flexibility of *CoS* lies mainly in the configurability of the data-exchange plane and how conveniently a programmer can access and adapt this plane in her application. It allows on-demand information gathering about the devices around and topology formation according to the application needs. For more dynamic network topologies, the maintenance of network information and device advertisements can be more frequent if an application requires so. Otherwise, the network may remain relatively dormant if no application-level updates are required.

VI. Conclusions

We proposed a new paradigm in operating system design called *Co-operating System* or *CoS*, that aims to ease the application development for cyber-physical systems. We argued that the current operating systems like TinyOS, Contiki or Nano-RK are designed with a goal to facilitate the programming of individual nodes in a network of embedded devices. Middleware or network programming frameworks are the other end of the spectrum that may reduce the flexibility of applications and jeopardize the reliability and robustness. Perhaps this is the reason that even with the development of several such solutions, not many have been widely adopted, and researchers still depend heavily on developing applications directly on top of an operating system. We provided the design principles behind *CoS* and discussed its architectural aspects that may enable significant changes in the way applications are developed and distributed for networked embedded systems. It can be argued that *CoS* may not be significantly different from a middleware running on top of a traditional OS in terms of the software-architecture, but the fresh perspective of creating network applications directly on *CoS* can provide a conducive setup for rapid and diverse application development for cyber-physical systems.

References

[1] T. T. 2.x Working Group, "Tinyos 2.0," in *the 3rd international conference on Embedded networked sensor systems*, ser. SenSys '05. San Diego, California, USA: ACM, 2005, pp. 320–320.

[2] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *the 29th IEEE International Conference on Local Computer Networks*, ser. LCN '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 455–462.

[3] A. Eswaran, A. Rowe and R. Rajkumar, "Nano-RK: an Energy-aware Resource-centric RTOS for Sensor Networks," *IEEE Real-Time Systems Symposium*, 2005.

[4] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He, "The liteos operating system: Towards unix-like abstractions for wireless sensor networks," in *Information Processing in Sensor Networks, 2008. IPSN '08. International Conference on*, april 2008, pp. 233 –244.

[5] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, and V. Bahl, "An operating system for the home (to appear)," in *Proceedings of the 9th USENIX conference on Networked systems design and implementation*, ser. NSDI'12, 2012.

[6] ——, "The home needs an operating system (and an app store)," in *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets. Monterey, USA: ACM, 2010, pp. 18:1–18:6.

[7] L. Mottola and G. P. Picco, "Programming wireless sensor networks: Fundamental concepts and state-of-the-art," *ACM Computing Surveys*, 2010.

[8] "http://www.dlna.org/."

[9] "http://www.z-wave.com/modules/zwavestart/."

[10] "http://osian.sourceforge.net/."

[11] A. S. Tanenbaum and R. Van Renesse, "Distributed operating systems," *ACM Comput. Surv.*, vol. 17, pp. 419–470, December 1985.

[12] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *the 3rd international conference on Mobile systems, applications, and services*, ser. MobiSys '05. Seattle, USA: ACM, 2005, pp. 163–176.