A New OS Architecture For Connected Devices

J.Y. Astier

SynapOS

Summary

Current computer operating systems architectures are not well suited for this new coming world of connected devices, known as Internet of Things (IoT), for multiple reasons: poor communication performances in both point-to-point and broadcast cases, poor operational reliability and network security, excessive requirements both in terms of processors power and memory sizes leading to too high an electrical power consumption. We introduce a new computer operating system architecture well adapted to connected devices, from the most modest to the most complex, and more generally able to tremendously raise the input/output capacities of any communicating computer. This architecture rests on the principles of the Von Neumann hardware model and is composed of two types of asymmetric distributed containers, which communicate by message passing. We describe the sub-systems of both of these types of containers, where each sub-system has its own scheduler, and a dedicated execution level.

What is the Architecture of an Operating System?

There are a number of general theories on “systems”, each giving its own definition of what a “system” is. We will be using the simple definition stating that a “system” is a “set of entities talking through interfaces (and according to protocols)". We will consider here, that in the case of a computer operating system, what are respectively called “entities” and “interfaces” in that definition are what are usually called “sub-systems” and “APIs” in computer speak. What we will be calling “architecture” is a description of these sub-systems and of its APIs. The description of a sub-system will be that of its scheduler, of the scheduler's modes of context switching, of its performance, but also of the services provided by the sub-system. The interfaces between the different sub-systems will be “procedural” or using “message passing” (Fig. 1):

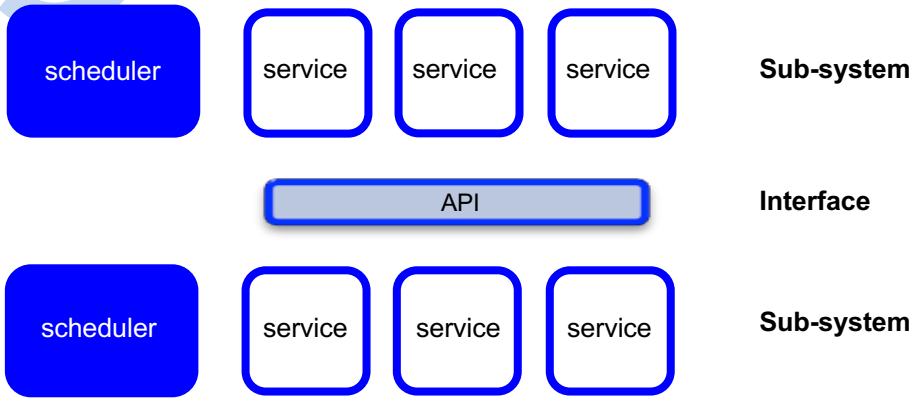

Fig. 1 - The “system” API organization

Why introduce a new architecture?

The architecture of the Multics system was among the most ambitious computer operating system architectures. Its definition started in 1964, and it can be considered finished in 1972, the year of the publication of an article titled *Multics – the first seven years* (we are leaving out the implementation of the system, which went on after the article was published). The following sentence comes from the article's conclusions:

In closing, perhaps we should take note that in the seven years since Multics was proposed, a great many other systems have also been proposed and constructed; many of these have developed similar ideas. In most cases, their designers have developed effective implementations which are directed to a different interpretation of the goals, or to a smaller set of goals than those required for the complete computer utility.

This sentence turned out to be prophetic, because well after 1972, numerous computer operating system architectures are still influenced by that of Multics, to such a point that certain architectural traits will be passed down from generation to generation, although their use will be long gone due to the evolution of the hardware and of the use of computers.

Recall that the first machine used to develop Multics, the General Electric Model 654, had at most 1024 kilo-words of central memory, with each word being 36 bits (for a total of 4.5 megabytes), had up to 4 processors, each capable of executing 500,000 instructions per second. It was the size of a very large room. By comparison, a modern smartphone has 500 times more memory, and 1000 times more processing power. The smallest systems-on-a-chip for IoT are the size of the smallest coins, have 0.5 megabytes of memory and can run 2,000,000 instructions per second. The use of a smartphone, or of any IoT has therefore nothing to do with that of the General Electric 654, which allowed a few tens of users to perform scientific computing, or to develop Multics using slow and loud teleprinters.

In the following paragraphs, we will analyze some of the downsides of architectural traits inherited from a long time ago, and ill-suited to modern communicating machines. We cannot overemphasize the fact that the consequences are not the result of a particular implementation, but rather of the architectures themselves. The past decade has seen a number of projects coming from developers whose aims were to rewrite part or all of existing architectures, without achieving significant gains. We will end this article by introducing a new operating system architecture meant for IoT and for communicating computer machines. In the following of this text we will use the term of communicating machine for naming machines using richer I/Os than a simple IoT device, that is to say using communication links, but also being able to process and store real time data flows.

Telecommunication and Input/Output Problems

We will distinguish three types of data flows which often exhibit telecommunication performance problems: high bandwidth point-to-point flows, high bandwidth broadcast flows, and low bandwidth flows of infrequent broadcast messages. In a more general manner, all input/outputs of our connected devices and computers are suffering of serious performances issues, more specifically file systems. We will proceed by pointing out five architectural deficiencies which account for the performance problems. Some of these deficiencies are so prevalent that they have driven the introduction of palliative hardware solutions which are often very expensive.

SynapOS

Almost all existing operating systems exhibit poor performance as soon as protocol stacks such as HTTP/TCP/IP meet sustained flows nearing half of the transmission bandwidth. To see this, one only needs to start streaming an HD video through a broadband Internet connection to a PC. When the streaming is started, a waiting animation is always shown, for up to a few seconds, even though the video should start playing instantly. Then, sometimes the video will pause, which should not occur. In that case, novices blame the PC, the network or the server of being too slow. Actually, the terminals' operating systems are to blame for these hiccups; that can easily be proved by replacing the terminal by a device with a suitable operating system. Now, on the same connection, the same files located on the same server can be played without problems. For a number of users however, DSL TV is not good enough, driving the telecommunications operators to replace copper wire infrastructure by fiber too early given the investments made.

The set-top box operating systems also show poor performance when receiving broadcast streams, both by cable or by terrestrial and satellite antennas. If it is possible to record on a hard drive a single TV program, without experiencing too many glitches, recording 7 or 8 is nigh impossible. The total data flow of 8 high definition videos is only 16 megabytes per second, and the slowest of disks, at 5400 rotations/minute, have a data rate of 80 megabytes per second when correctly used. To cope with these deficiencies of the terminal operating systems, hard-drive manufacturers sell highly expensive models whose rotation speed is of 10,000 rotations/minute or more, and whose firmware is specially modified for video recording.

The most energy efficient home-automation radio protocols use small broadcast messages, which are not repeated. For instance, there are small electrical switches which broadcast exactly one small message over the radio. Although the speed is only 115,200 bauds, some systems can sometimes randomly lose a few bytes of the message making it unusable, even though the hardware is fully able to receive them without any loss. Although the radio protocols were specially designed to limit the electronics for a modem directly connected to an asynchronous serial port, the radio modems manufacturers are driven to add a small system-on-a-chip, whose sole purpose is to store the bytes received for the few milliseconds that are sometimes required by the operating systems to take them into account.

These problems are caused by multiple architectural weaknesses, which combine and add up. Among them, the five most egregious are the high context switching times, the useless data copies, the excessive data buffering, overly general file systems, and lastly, inappropriate hardware interruption handling mechanisms. Once again, these are not implementation problems of these operating systems, but rather problems of the architectures themselves. The five issues are presented in further detail below.

Excessive Context Switching Times

The context switching time is the time spent at every transition from one activity to another, that is the time needed by the operating system to perform ad-hoc actions. The architectural choices of what has to be done at every context switch determines the time it takes: whether to change the address space or not, whether to change the stack or not, and the extent of the modifications to the internal tables of the system. We purposefully use the term of “activity”, rather than “task” or “thread”. In fact – and this is an important architectural design decision – there exists a technique called finite state automaton engine, which is not very widespread in the software world because it is difficult to master, but which nonetheless allows for context switching in virtually no time by eliminating the notions of task and threads. We will see later how to gainfully use it.

SynapOS

Useless Data Copying

Useless copies in the central memory of the machine can occur with certain operating systems, because, as a result of the isolation mechanisms of the tasks and the system itself, copying is the only data transfer method between two tasks or between a task and the system itself. Architectural design decisions are the sole cause.

Excessive Data Buffering

Buffering consists in accumulating an amount of data before even starting their processing. Often, these buffers are found to be 200 to 10,000 times too large and are in the megabytes or tens of megabytes rather than kilobytes. That's why users who start streaming must wait multiple seconds before seeing the first image even appear: the data begins to arrive from the server as soon as the request is made, but only when megabytes of video data have arrived does the video decoding begin. One may ask why such large buffers are used, causing such long wait times. There are in fact two causes for this. The first is the operating system context switching times; using buffers a thousand times smaller means a thousand fold increase in number of context switches by unit of time, which leads to a system collapse as soon as the activities are organized as tasks, no matter how "lightweight". The second is an implementation issue. Many binary data flow parsers are unable to stop their work at any point in the input flow, but only at certain points, such as at the beginning of a new image. These parsers can only be started when a buffer contains an amount of data at least equal to the largest possible compressed image size. This second point is not an architectural defect in and of itself. The architect having identified that the overly coarse parsers are a cause of system collapse can choose to integrate critical features in the system itself or in its libraries, so that they can be well implemented and optimized. With this, it is clear what the benefit is obtained by specializing an operating system: providing optimized business-specific functionality to developers, which they cannot optimize sufficiently due to a lack of time. Moreover, modern hardware sometimes comes with hardware parsers which developers don't use or under-use because of the complications this entails. The integration of all the parsers for high bit rate flow allows for the use of such hardware support. It is important to realize that using more powerful processors would not solve anything, because it would not diminish the context switching costs by a factor 1000 or 10,000 and would do **nothing** to reduce the number of bytes needed before starting these parsers. This last point is of utmost importance, because it explains why it is absurd to always increase the processing power of our smartphones: this extra power is of no use, because that was never the source of the sluggishness. And because of it, the battery life goes down. In fact, video decoding is the only reason why there are 8 cores in a cell phone, function for which there are dedicated processors (DSPs) which can reduce the power draw by a factor ten compared to software decoding on an eight-core processor.

Ill-Suited File systems

To be clear, the term "file system" from a strict point of view refers to the organization of data on a physical support. A file system is labelled "general file system" when it is both capable of containing a large number of small files as well as large files. In its general acceptation, which we will use, "file system" also refers to the operating system components which implement and maintain this organization. For the same disk data organization specification, such as FAT 32, there are multiple ways to maintain this organization, and the methods chosen will have impacts greater than the organization itself. Depending on the use-case, file system strategies will favor small files by allowing fast creation and destruction or will benefit the throughput for large files. In the case of communicating machines simultaneously receiving large bit rate data flows, it will be necessary to use file systems well-adjusted to that use case, where the movements of the disk's read-and-write head are minimized.

synapse

Poor interrupt handling

Modern electronics, including the smallest *Systems on a Chip* used for IoT, all have interrupt controllers allowing nested interrupts. Software can configure the priority of each interrupt, and when two interrupts arrive nearly at the same time, the handling code for the first one can be interrupted if the priority of the second one is greater. The software must assign the greatest priorities to the shortest and most frequent handlers, and the lowest priorities to the longest and rarest handlers. That way a long handler will be interrupted by short ones, and no latency is induced in the handling of the frequent interrupt, which leads to the maximum speeds. A short handler cannot be interrupted by a long one, but this is of no importance. Of course, the short handler codes must be carefully optimized, and will not perform slow calls to the system. No interrupt handling can call a blocking function, such as grabbing a semaphore. Certain architects consider that the considerable power afforded by the modern processors eliminates the need to make use of the interrupt priorities, and that it is not a problem to call slow system procedures from interrupt handlers even though simple order of magnitude computations prove the contrary. This is the reason why certain operating systems, even when run on powerful processors lose the last bytes of small radio messages transmitted at 115,200 bauds: if a UART has a 16-byte deep FIFO, it can be filled in 1.5 milliseconds, and the following bytes will be lost if the interrupt handler is not run during this time period. In the case of a communicating operating system, which because of its job, is exposed to sustained high interrupt rates, the interrupt handling must be performed swiftly and cannot be postponed and placed on a run queue, because doing so only increases the amount of ineffective activities in the system, increases the latencies, and worsens the problems due to interrupt misses.

Safety problems

In the case of developer workstation operating systems, the system must be fully protected from application bugs during the testing phase, no matter how expensive the protections are. Certain systems go so far as to protect components of the operating system or the drivers by automatically stopping the failing piece of software. An operating system meant for communicating machines is not designed to be used for its own development as was Multics, or to run text editors and compilers. In addition, in embedded communicating objects, it would be absurd to stop the software components allowing precisely the communication, thereby isolating the device. It is in fact better to restart the system. Most importantly, what matters in the end is the safety of the operation of the whole communicating machine, which is a combination of hardware, operating system and application software.

As far as the operating system is concerned, the safety of its operation is guaranteed by a large number of automated, off-line checks. Each component of the system must be submitted to checks called “unit tests”; the component is tested alone and in isolation of the other components. A major issue is the comprehensiveness of the test coverage, that is the guarantee that all possible code paths have been explored at least once during the tests. For this last point, the architecture chosen for the operating system is not without consequence. In particular, the use of the previously mentioned finite state automaton engine technique lets one automatically detect if test coverage is complete or not.

SynapOS

As far as the applications are concerned, the choice of suitable architectures can also be essential, especially for the smallest of IoT devices. Let's consider two examples: when the language used to implement an application is compiled to machine code, it is impossible to prevent the application from inappropriately accessing its own data, and it is expensive to limit its address space in order to isolate the code, the system data and the peripheral registers. With architecture called “language-based computers”, the combination of the hardware and its operating system do not offer any other choice than a single interpreted language for the development of the applications. It becomes impossible for an application to overwrite its own data, the code or data of the system, and it can't access the peripheral registers.

Security problems

The protection of the data collected or stored on connected devices, but also the integrity of the software of the object makes up what we will call the “security”. Surprisingly, certain architects consider that security is not an architectural concern. It is as if it were possible to secure an operating system after the fact, by adding “security layers” and whose implementation as also outside the scope of the architect's responsibility. To illustrate, it is as if it were possible to draw up the plans for a building without taking account its security, to finish the construction, and only then try and achieve a high level of security by adding surveillance systems and access controls. When building a bunker or simply a locker room, the architect must from the design stage ban windows, add sufficiently thick walls, and use strong materials.

It behooves the architect to design a system which can start without relying on a shell. The architecture should by design prevent the execution of unauthenticated contents, prevent a user from consenting to disable security mechanisms. And of course, the architecture should allow for the replacement of open source telecommunications stacks by proprietary codes, of which must be required to come with automated test with extensive coverage.

Frugality and Scaling Problems

IoT must be frugal, both in terms of energy consumption as well as communication bandwidth. The architecture of the operating system impacts the power consumption, from the moment it allows a reduction in required memory sizes, and when it lets the processor clock remain low and even stop completely, which is only possible if the system can restart quickly, typically in less than a millisecond. For IoT of the smaller kind, communicating using a low bandwidth radio, software updates can be realized on a component by component basis, in order to reduce the update sizes, which is only possible if the architecture was designed for this. From this point of view, the Language-based Computer architecture is particularly well suited. Indeed, the natural functional decoupling is strong in that case: the interpreter itself is an independent component, each library is an independent component, and the interpreted application is another. It is easy to add an independent version number to each of the blocks, allowing true upgrade of a single component, without needing to upgrades the others which tends to happen with compiled languages. Because of a single function address table, it is possible to upgrade the functions of a library one by one, without needing to change more than one address in all the code, the address in the function table of the interpreter.

SynapOS

The capability of an operating system to both shrink in the smallest of objects and make use of the most complex is what we call its “scalability”. The amount of necessary code must be minimal, all while letting more code be added to provide more features. For this also, a Language-Based Computer architecture turns out to be very efficient. The interpreter code is only a few tens of kilobytes, and the number of functions it can call can be updated thanks to the use of an indirect call table. It is therefore required that the P-codes of the instructions be at least 16 bits long, which rules out those whose instruction size is only 8 bits.

Fleet of Objects Management Problems

The deployment of large quantities of small objects in homes poses three problems of fleet management: the management of the fleet of objects inside one home, the management of objects of the same type across homes, and finally the management of the data which we allow to leave the premises.

Management of the Objects in one Home

The reason for adding communication abilities to small objects is first and foremost a local one: the objects in one home must interact with and receive commands from the residents. Some standards such as EnOcean allow controller type objects to discover locally sensor and actuators, and to create links with them. Specific IP protocols such as Simple Service Discovery Protocol (SSDP) also allow the discovery of local equipment. Even if they are simple compared to TCP/IP, these protocols are nonetheless too complex to be used by application software. It is therefore desirable to integrate them to the operating system.

Object Maintenance

The software upgrades in the case of IoT, where we only have low bandwidth radio channels occurs through gateways, which have an Internet connection as well a connection to the radio channel. If we compare the total embedded code sizes, with the bandwidths of the radio channels, it become obvious why partial updates are required for code embedded in objects. From this point of view also, Language-based Computer type approaches are beneficial because the part of the application most often upgraded, the application, is wholly separated from the rest of the code, because it is written in a different language. It is highly desirable, and it may be required that all modifications to the code be authenticated by the operating system.

Exported Data Management and Permissions

The personal data collection and their mining has become a source of profit. Google for instance, to better sell targeted advertising, is increasing the sources of collection and cross-matching of our personal data, in particular the contents of our mails (Gmail) which are systematically analyzed by artificial intelligence software called “bots”. All of the manufacturers of IoT products also wish to collect and store the data collected in our homes for their own benefit. To do so, the collected data are directly transmitted using the Internet to back-end servers under the control of the manufacturers of objects, and only through these servers can users access their own data. Even if end users are doubtlessly ready to let this silent data collection happen, if they even realize it is going on at all, the same cannot be said of the managers of industrial sites, or of housing domains, who are justifiably afraid of security breaches. The operating systems of IoT controllers will have to offer the means to encrypt the transmitted data, but also to block confidential data from getting out.

SynapOS

Hardware input/output architectures

Historians generally consider that the first machine deserving to be called a computer is the ENIAC (Electronic Numerical Integrator and Computer). This computer was designed at the Ballistic Research Laboratory of the US Army, at Aberdeen in Maryland. This machine was designed by John Presper Eckert, based on ideas of John William Mauchly professor of physics, who had realized that computation of ballistic tables could be performed electronically. Before the ENIAC was fully functional, a second project called EDVAC (Electronic Discrete Variable Automatic Computer) is launched in 1946 under the direction of the same Eckert and Mauchly, based on a documented written in 1945 by John Von Neumann: First Draft of a Report on the EDVAC. As the name suggests, the document is a first draft, and is very incomplete. The subject of the document is in no way the description of a computer architecture but does have going for it that it explains how to use vacuum tubes to increase the speed of computation. The central chapter is this document, *6.0 E-Elements* gives a model of the "elementary electronic neuron", and the following chapters describe how to build an adder, a multiplier, and a subtractor, binary floating points, and central memory using this elementary building block. By way of introduction, the document starts with a paragraph called *2.0 Main subdivisions of the system*, which is, without saying so, a simplified functional description of the ENIAC. But because this article was published in 1945, before the 1948 article "The ENIAC", and because achievements were considered more important than publications in these final years of World War II, the historians mistakenly both attribute the invention of the architecture to John Von Neumann and tie it to the EDVAC.

The two fundamental points of the architecture of a computer are on one hand the description of its buses and of its instruction set, the two being closely tied. The 1945 article by Von Neumann does not address the notion of a bus at all, and only defines the different classes and sub-classes of instructions. Bear in mind that the article contains no architectural diagram. Nonetheless on-line literature is filled with diagrams entitled "Von Neumann architecture," most of which do not contain a bus, which is a central element without which the understanding of a computer is impossible, and which both the ENIAC and the EDVAC had. The following architectural diagram was draw by us from the February 1948 article *The ENIAC* (J.G. Brainerd et T.K. Sharpless), in particular the chapter *Machine Components (Fig. 2)*:

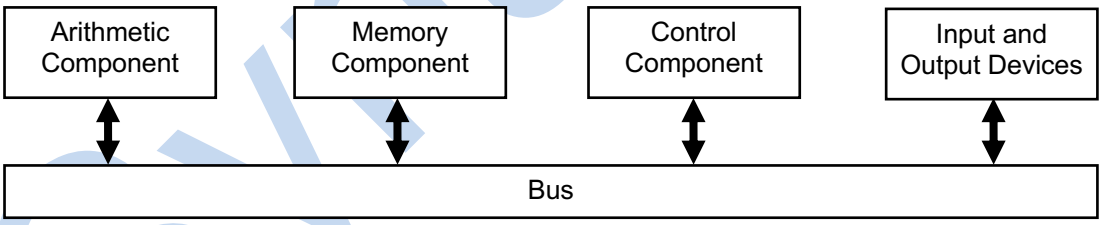

Fig. 2 - A universal bus of ENIAC Machine

It's important to realize that the four rectangles at the top of this diagram have little to do with the physical architecture of the ENIAC which is found in *Figure 2 ENIAC Floor Layout* of the same article. What is called Arithmetic Component is a set of electronic bays in which the addition, the subtraction, the division, but also some memory; 20 accumulators and 3 function tables. The set of bays corresponding to the Memory Component also contains 20 other accumulators and 3 function tables identical to the previous ones. The *Input and Output Devices* rectangle represents a set of various hardware setups among which are card readers, a card puncher, led and button panels, each of them directly connected to the bus, which is made up of coaxial cables.

SynapOS

It is essential to remember that both the ENIAC and the EDVAC were machines meant for performing computation - for ballistic trajectory computation to be precise. In no way were these machines meant for data processing in the modern sense of the term. They were computers in the sense of automatic electronic calculators. Only ten years later would the distinction between computer and calculator disappear for good. All the attention and energy of the ten or so engineers working on building the ENIAC was concentrated on the creation of the electronic bays of the *Arithmetic Component*, *Memory Component*, and *Control Component*. All the *Input and Output Devices* were well tested devices bought "off the shelf" from different companies such as the card readers and punches were from IBM, these types of devices having been used for 30 years at that point (IBM was founded in 1911). Plugging these devices to the computer was a minor issue, little thought and effort was given to the problem. In 1948, in a famous article called *The Eniac*, two engineers who took part in the project, J.G. Brainerd and T.K. Sharpless, wrote the following: "Current developments in large-scale general-purpose digital computing devices are devoted to a considerable extent to obtaining speeder input and output mechanisms."

These developments bore fruit. In 1978, Hewlett-Packard started selling a machine considered to be one of the first "workstations", that is meant for a single user. The HP 9845A comprises a graphical screen, a keyboard, a printer, and a tape reader. Like all Hewlett-Packard hardware, it is very easy to use and perfectly well built, making it a subject of envy and admiration. In the design of its architecture, the effort spent on input/output goes far beyond what was done for the program execution sub-system (Fig. 3):

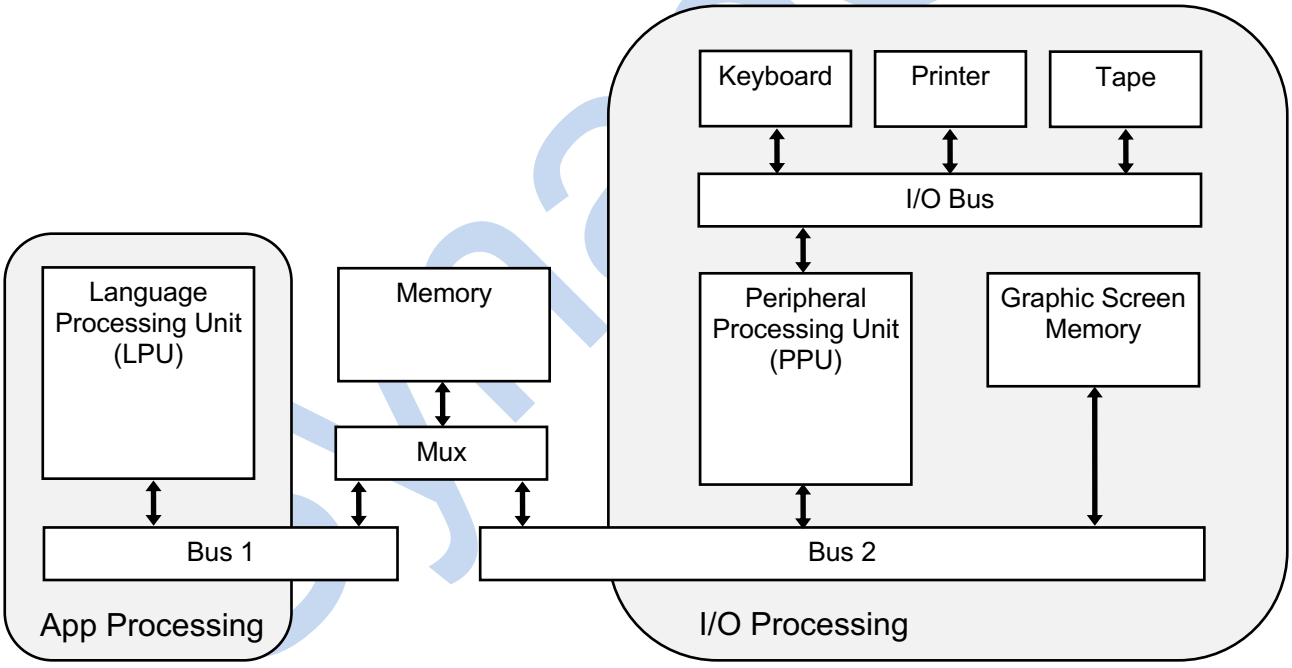

Fig. 3 - The first "workstation" principal construction

The LPU executes a BASIC interpreter. The BASIC program is stored in memory with two access ports for the LPU and PPU. When the BASIC program requests an input/output, a request is written in the two port memory, and is executed by the PPU. While the PPU executes the input/output, the LPU can proceed with the execution of its BASIC program.

A new architecture for communicating machines

The remarkable hardware architecture of the HP 9845A of 1978 described above had only one important problem, and that was its cost because it used two 16-bit processors, the LPU and PPU running at 5.7 MHz. By comparison, the first micro-computers IBM PC 5150 of 1981 only had a single Intel 8088 processor running at 4.77 MHz. But both the removal of the processor dedicated to the input/output and the design of the BIOS low-level software layers which did not expose the hardware interruptions gave the IBM PC 5150 poor input/output performance. Only with OS/2 (1987) and Windows NT (1993) did decent input/output handling systems emerge, and they were still far from the true capabilities of the hardware devices. In 1999, in an article entitled *Introduction to "The Eniac"* (referencing the 1948 article *The Eniac*), W. Burks (one of the main designers of the electronics of the ENIAC) and E. Davidson wrote:

"... and once again, as with the ENIAC, computation rate is not the performance-limiting factor, rather it is still the communication, the I/O, the setup for the computation. It seems that communication science may be at the heart of the problem after all."

In reference to this still relevant observation, we introduce a new distributed operating system architecture meant for IoT and communicating machines. It is based on the notion of "containers", that is a set of self-sufficient codes that can either run on bare metal without any other software, or on top of any third-party operating system. The architecture introduced is composed of two types of containers which talk only by messages, the app containers and the I/O containers.

Every time it needs to perform input/output an app container sends one and only one request message to the I/O container, which will return exactly one response message. We have a client/server relationship, the app container being the client, and the I/O container being the server. A single machine can host either only an app container, only an I/O container, or both. Every machine in the same local network has a unique number called its node number.

The request and response messages have exactly the same structure, called an event. An event carries two addresses, one for the recipient and one for the sender. An address is a triple of integer numbers: node number, automaton number, and way number. The automaton number designates a functionality of a container, while way numbers distinguish the different instances of the same software functionality.

An app container can contain an RTOS, C applications, interpreters, an HTML renderer or even another operating system. In the latter case, the container must include a software component for that system to converts all the input/output commands to request messages.

An I/O container contains drivers, protocols, services, and file systems. When a message reception is requested, it can stack a protocol or a file system on top of a driver. It is also able to create pipes which are unidirectional data flow within the container. It has two schedulers called VMIT and VMIO. The first preempts the second with no latency, with the very next instruction running the VMIT.

SynapOS

The following diagram represents one communicating machine and two IoT devices on the same local network, typically Ethernet and WiFi (Fig. 4):

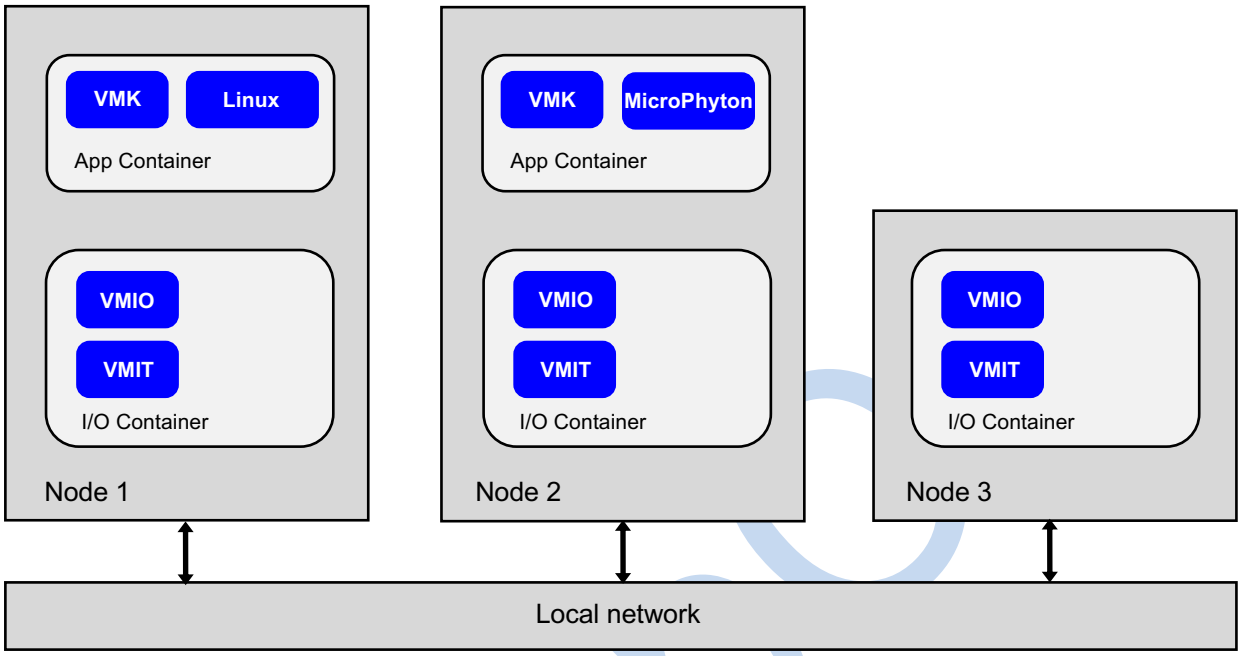

Fig. 4 - Communication of several IoT devices

Node 1 will typically be a router, a NAS (Network Array Storage) file server, it can have a display, such a TV or a set-top box. It contains four sub-operating systems, each having their use and a different latency. The VMIT responds to hardware interrupts, with a response time much less than the microsecond. The VMIO receives I/O requests and has an average response time of less than 100 microseconds. The VMK is a non-preemptive RTOS which allocates time to the Linux kernel. The Linux kernel is stripped of any driver, protocol, network service or file system. All of the input/output requests coming from the Linux applications are transformed into request events which are directly deposited to an I/O container, which can be in any of the nodes 1, 2, or 3. For a Linux binary, all the devices of the three nodes are seen as local devices. Without any modification, the Linux kernel benefits from distributed system features. We therefore have four sub-systems, where each has a specific role for which it is specialized, and from the architecture of its scheduler has a different response time. Of course, the most feature-rich sub-system will be the slowest, and conversely the most lightweight will be the fastest.

Node 2 will typically be equipped with a 4 MHz processor, with 16 kilobytes of RAM and 128 kilobytes of code, which gives it capabilities equivalent to that of the 1978 HP 9845A. Like this machine, node 2 is a *Language-based Computer:* its programming language, in this case MicroPython is interpreted. Like the HP 9845A, it is responsible for all the input/output. Unlike the 9845A, a single processor is used. The VMIT preempts the VMIO and the VMK with zero latency, and the VMIO preempts the VMK with zero latency.

Node 3 is a small sensor which does not contain an application container. It behaves like an input/output server; it handles requests which can come from the other nodes.

Application Containers

Like the Language processing Unit of the HP 9845A, an application container contains everything that allows an application to run, except for the input/output. What constitutes the application itself will be there, C code, MicroPython code, but also HTML files. The container also includes the necessary code for that code, such as mathematical functions, a MicroPython interpreter or an HTML renderer. It contains an RTOS which will support the application written in C, the interpreter or the HTML renderer.

The VMK

The features that the RTOS must provide are very limited, because all of the input/output will be transferred to the I/O container. We call VMK this simplified RTOS. It must provide primitives for task handling, for semaphores, and for event queues. Every task of the VMK has a stack, and a message queue of its own events, and optionally its address space. Finally, the VMK must have the means to send events to the nodes, so that the I/O requests can reach the various recipient container nodes.

HTML renderer

The code bases of many large software projects such as HTML renderers usually have abstraction layers for all of their input/output: telecommunications, files, and graphics. They are therefore easily embeddable in application containers, using a conversion layer that transforms all procedural input/output calls to request events. Furthermore, experience shows that 90% of what is believed to be part of the HTML rendering in itself is actually code used by the various abstraction layers meant for the various operating systems and the remaining size will go down from 200 to 20 megabytes of code.

Application-Oriented Operating System

An application-oriented operating system such as Unix or Linux can easily be embedded in an app container. It is easy to strip them of all of their drivers, protocols, network services, and file systems and to create once and for all, a conversion layer like with the HTML renderer, which turns the procedural input/output requests to request event deposits. Having done this, the size of the kernel is only a few hundreds of kilobytes. In this way, we have as a single VMK task, a Unix/Linux kernel where almost all security flaws have disappeared, because the weakest components have been replaced. Previously unobtainable throughputs are achieved, and the data flows inside pipes within the I/O containers. Finally, the strict separation between the kernel and the I/O located in distinct containers and communicating by passing messages, makes using hardware peripheral register access controls (a trusted zone) easy, further increasing the security of the data.

One should note that the client/server model used between applications and for input/output means that the requests made by the application does not need to go through the VMK and are directly sent to the recipient input/output container. The same is true for the response events.

SynapOS

Input/Output Containers

An input/output container is the equivalent of the Peripheral Processing Unit of the HP 9845A. It receives input/output and pipe configuration requests. It contains drivers, protocols, network services, and file systems. It is composed of two sub-systems which are: VMIT, an interrupt handler, and VMIO, a monolithic input/output monitor.

Interrupt handler

At the lowest level is a sub-system called VMIT, and whose role is to handle hardware interrupts with different priorities and which can therefore be nested. To each hardware interrupt priority corresponds one and only one stack. The address space is unique and is that of the input/output monitor. This sub-system handles the low-level drivers, that is all the code which depends on the peripherals. For every type of device or peripheral, there is a specific *Hardware Abstraction Layer*, that is a specification of both the interface of the procedure meant to control the type of device, and second the events submitted by the interrupt handlers. These messages can only be deposited to the local input/output monitor, located in the same address space. All of the code of the VMIT depends on the specific hardware, including the scheduler which depends on the CPU and the interrupt crossbar wiring.

Monolithic Input/Output Monitor

Above the VMIT is a monolithic input/output monitor called the VMIO, which executes high-level drivers, the protocols, the services, and the file systems, all implemented as automaton transition tables. There is no notion of a task at this level, nor of semaphores which makes going from one function to the next a latency-free operation, which in turn allows for high bit rates even with multiple concurrent flows. In order to achieve zero-latency, the VMIO uses a single stack for itself and all its automatons, a single address space and a single set of registers. Every driver, protocol, service or file system component is composed of a transition table (state, event) and of a set of C procedures, which are all transactional handlers for a transition. These C procedures, these automaton transition handlers, when they need to send commands to a device under their control, will use directly call without any context switching the C procedures which implement the *Hardware Abstraction Layer* for the device. There is a one to one correspondence between high-level drivers and the specifications of the low-level drivers. All of the code for the VMIO, including the scheduler, is written in C, and is strictly portable without any modification or conditional compilation. The VMIO has a pipe mechanism, which allows a unidirectional data transfer from automaton to automaton. A pipe is configured and started using request events, and then the client does not need to intervene anymore for the data to flow from automaton to automaton. Thus, the various data flows occur without any context switching, and in particular no copy and no address space change. If for instance the application configures a pipe ETHERNET => TCP => HTTP => FAT32, a high-speed download feature is realized.

In this example, the TCP protocol, the HTTP service and the FAT32 file system are executed within the monolithic monitor, and not within the RTOS, which runs in the app container. This is a major trait of the architecture we introduce. Some have noted that the RTOSes are two slow for demanding I/O tasks and have tried to develop faster RTOSes by reducing the context switching time. This is pointless, as it reduces the functionality of the RTOS, without even truly reaching the goal of a sufficient speed-up. By having both an input/output monitor and separately a RTOS, we can both perform I/O faster than a RTOS would, while also having a feature-rich RTOS if needed.

SynapOS

Media Home Gateway Example

A Media Home Gateway will typically have a hard-drive, and an Internet connection be it Ethernet or WiFi. It hosts an application container and an input/output container (Fig. 5):

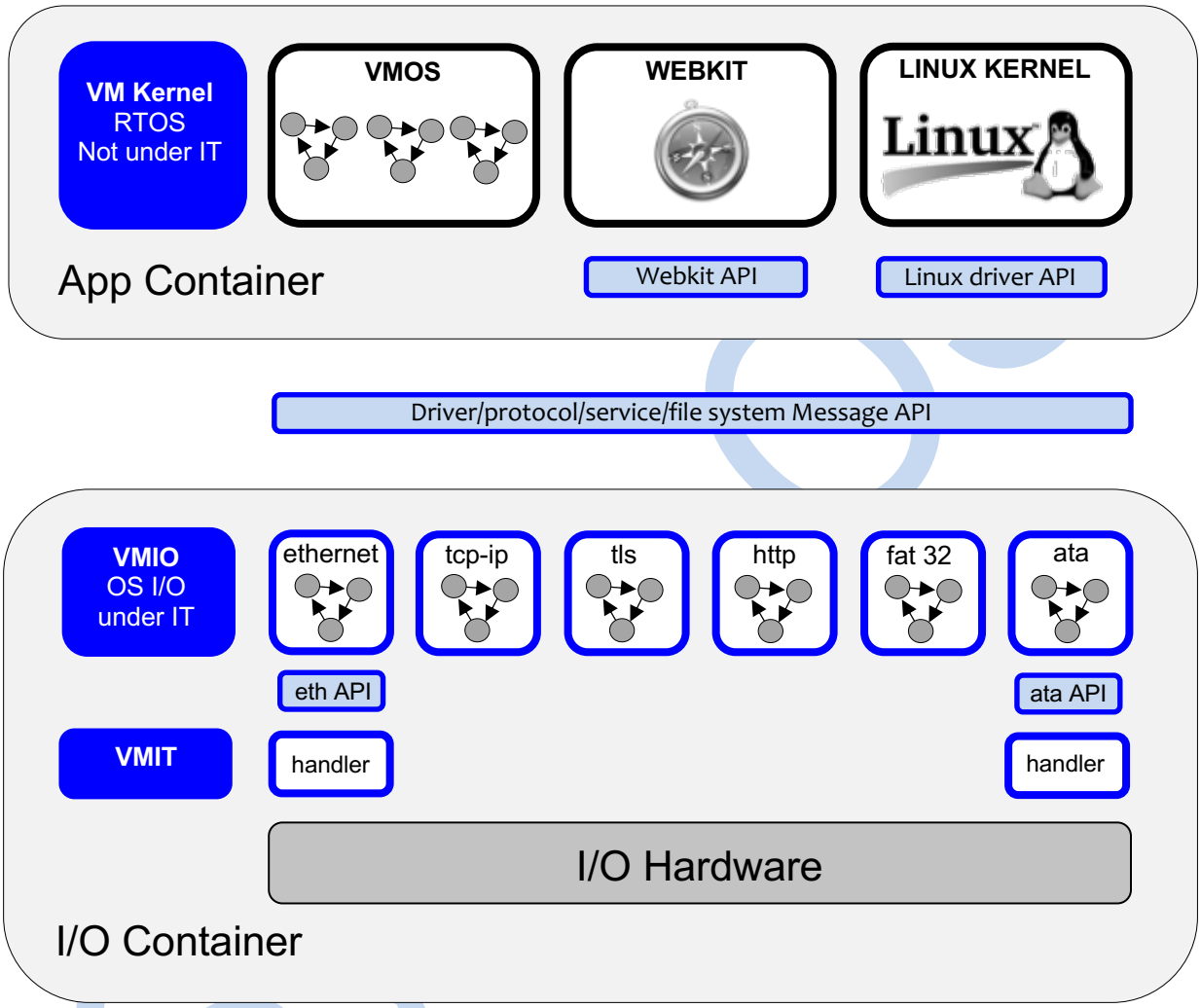

Fig. 5 - Home media Gateway device software organization

The application container embeds a first application operating system VMOS which contains interpreters built as finite state automatons. The VMOS executes the *middleware* of the *Media Home Gateway*. Second, there is a HTML renderer, for which an abstraction layer was implemented, which the input/output requests of the renderer are converted in events deposited to the I/O container. Third is a Linux kernel stripped of all of its drivers, protocols, services, and file systems. An abstraction layer converts all Linux input/output requests to request event deposits.

The I/O container has two low-level drivers, that is hardware-specific code, one for the Ethernet controller and one for the ATA disk. The two interrupt handlers are called by the VMIT. Each low-level drivers needs to follow the specification for the type of hardware, the low-level Ethernet driver must implement an *Ethernet API* which is not the same as the *ATA API*. The VMIO input/output monitor is a finite state automaton engine, which executes the automatons that are independent of the hardware. It contains the “high-level” drivers for Ethernet and ATA, but also the TCP/IP and TLS protocols, the HTTP service, and the FAT32 file system.

An IoT with Application Example

An IoT *Language Based Computer* will typically have an Ethernet connection and multiple radio modems such as Bluetooth Low Energy or EnOcean. It hosts an application container and an I/O container (Fig. 6):

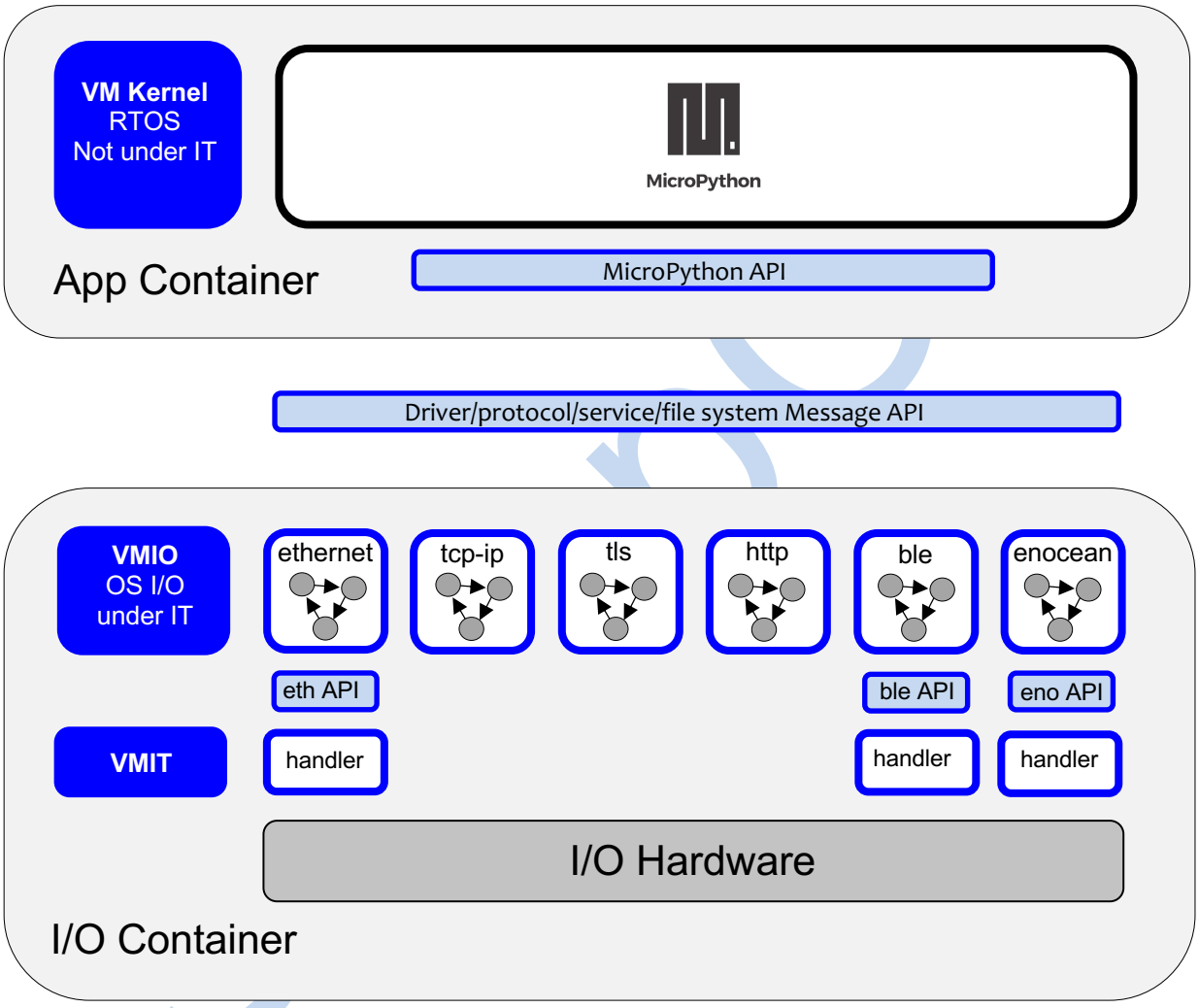

Fig. 6 - Example of IoT device software with Application level

The app container embeds a MicroPython interpreter and an application written in MicroPython. An abstraction layer converts the MicroPython input/output request to request events.

The input/output container has three low-level drivers, that is hardware dependent code, the one for the Ethernet controller, the one for the Bluetooth Low Energy modem and the one for the EnOcean modem. All three interrupt handlers are called by the VMIT. Each low-level driver must implement a specification specific to the nature of the hardware, the low-level Ethernet driver must follow the *ethernet API*, which is not the same as the *ble API* (Bluetooth API), or the *eno API* (EnOcean API). The VMIO I/O monitor executes six automatons which are independent of the hardware: the high-level drivers for Ethernet, BLE and EnOcean, but also TCP/IP and TLS, the HTTP service.

A Basic IoT Example

A basic IoT device will typically have an EnOcean radio modem, and binary I/O using GPIOs. It only contains an I/O container (Fig. 7):

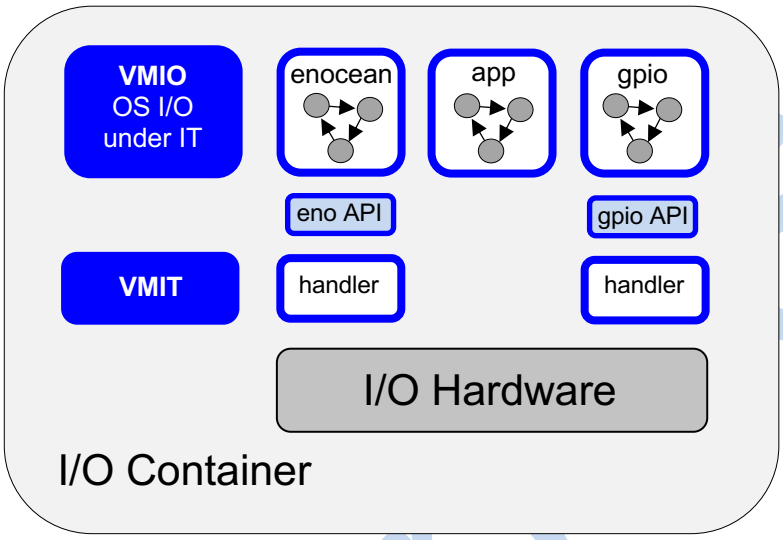

Fig. 7 - The basic software set for IoT device

The input/output container has two low-level drivers, one for the EnOcean radio modem, another one for the digital and analog GPIOs. The two interrupt handlers are called by the VMIT. The VMIO input/output monitor executes three automatons which are independent of the hardware; the high-level EnOcean and GPIO drivers, but also a micro-application called app.

The small application called app is written as a finite state automaton and is inside of the input/output container. Typically, the app automaton receives remote requests and exerts elementary command control logic. This is an additional feature afforded by this architecture, allowing to further shrink of the code by eliminating the application container and the VMK.

The basic IoT can simultaneously behave like a remove device in the case of a distributed system, being an input/output server, but also as an IoT able to handle messages which are not events thanks to the small app automaton which can contain a small message parser of another format, and simple application logic.

SynapOS

Conclusions

In a world where hardware technologies have long followed Moore's law, the principles of operating system design evolve at a entirely different pace, where the major traits of the architectures of the systems such as Windows, iOS, and Linux are derived three decades of work dating back to 1964 (Multics architecture).

The study of the new architecture introduced here has started in 1986, with the need to implement a distributed satellite image processing system. In order to transparently chain processing performed on one hand on a mini computer running FORTRAN code, and on the other hand on an image processing machine, the two being connected by a high-speed interface, the software was split in an application container, and two image processing containers. The first container comprised a FORTRAN interpreter, which made requests to the image processing software located on the same computer and also to the container for the image processing machine. On the 27th of April 1986, two days after the Tchernobyl nuclear plant accident, one of the first infrared picture of the site is transmitted by the SPOT satellite to be worked on using this application composed of three containers distributed across two machines.

All of the architectural traits described here have been validated separately through the implementation of various military and civilian projects between 1986 and 1999. From 1999 to 2008, a first version of our operating system has been implemented following all of the architecture, proving that it is indeed a very efficient solution to all of the problems identified at the start of this article. This culminated in real projects, delivering consumer electronics that reached the market.

The architecture is based on concepts and ideas which all existed previously. We did not invent the notions of container, of *Language-based Computer*, of finite state machine engine, of input/output monitor, of interrupt dispatcher, of abstraction layers, of sub-system separation, of preemption, or of execution levels. We did not invent a model for hardware architecture such as the one attributed to Von Neumann, nor did we discover the importance of logical automatons of which Alan Turing has shown that they are a model of any machine capable of computation.

We did combine all of these concepts and have thus obtained a new architecture and thanks to this we believe we have solved the major problem of input/output, for which W. Burks observed as late as 1999 that a lot remained to do from a conceptual point of view. Moreover, this architecture allows improvements in *machine to machine* relationships, in operational safety, in security, in frugality with regards to hardware resources and electrical power, in scalability, and in deployed fleet management. The use of finite state automaton in the input/output container, allowing both greatly improved performances and the comprehensiveness of unit tests, is an essential aspect.

This architecture can be used fully or partially according to the needs. It is first and foremost meant for true IoT. It is also applicable to all systems doing input/output whatever they are. But it can also be used in hybrid objects, that is objects which both communicate while also performing more complex software functionality: control systems, data collection and storage, or local artificial intelligence.

The integration of small rule-based programming, also called “artificial intelligence”, inside our home clouds seems essential in order to use all the hardware capabilities of our communicating machines. These will have to be installed and configured in as fully automated manner as possible, be able to perform “reflex arc” type actions and request assistance from the outside when needed.

SynapOS