R5-COP

Reconfigurable ROS-based Resilient Reasoning Robotic Cooperating Systems

D12.22: Software Interface Specification

Csanád Erdős, Tamás Dabóczi, Gábor Naszály, András Szél, Gábor Wacha (BME), Mateusz Maciaś (PIAP), Jesus Alonso (TTS)

Project	R5-COP	**Grant agreement no.**	621447
Deliverable	D12.22	**Date**	31.07.2015 (M18)
Contact Person	Csanád Erdős, Tamás Dabóczi	**Organisation**	BME
E-Mail	erdos@mit.bme.hu, daboczi@mit.bme.hu	**Diss. Level**	PU

Grant agreement no.	**621447**
Project acronym	**R5-COP**
Project full title	**Reconfigurable ROS-based Resilient Reasoning Robotic Cooperating Systems**
Dissemination level	PU
Date of Delivery	31.07.2015 (M18)
Deliverable Number	D12.22
Deliverable Name	Software Interface Specification
AL / Task related	12.2
Author	Attila Szarvas (BME), Tamás Dabóczi (BME)
Contributors	Csanád Erdős (BME), Gábor Naszály (BME), András Szél (BME), Gábor Wacha (BME), Mateusz Maciaś (PIAP), Jesus Alonso (TTS)
Keywords:	semantic technologies, domain specific languages, ontologies, software component reuse, component plugability, embedded virtualization
Abstract	This document gives an overview and evaluation of software technologies that can significantly increase the reconfigurability of robotic systems. The presented virtualization techniques provide a general interface between hardware and software components, and semantic web technologies provide general data-exchange solutions, that represent the most generally applicable form of software interfaces to date.

Document History			
Ver.	**Date**	**Changes**	**Author**
0.1	29/1/2015	Added sections for general and embedded virtualization Added preliminary versions of Abstract interfaces and Sensor interfaces in security robots sections	Mateusz Maciaś (PIAP) Gábor Naszály (BME) Attila Szarvas (BME) Gábor Wacha (BME)
0.2	30/1/2015	Added full draft for investigating sensor data representation standards, and platform and embedded virtualization (QEMU). Section discussing semantic data formats and software interfaces extended.	Tamás Dabóczi (BME) Csanád Erdős (BME) András Szél (BME)
0.3	5/2/2015	Added full draft for sensor interfaces in security robots.	Mateusz Maciaś (PIAP)
1.0	6/2/2015	Introduction and conclusion added.	Attila Szarvas (BME)
1.1	16/2/2015	Added info related to audio & video sensor interfaces	Jesus Alonso (TTS)
1.2	26/06/2015	Added section about MoM for seamless interfacing	Andrija Feher (SYN), Sönke Michalik (TUBS)
1.3	26/07/2015	Updated section about MoM, ROS 2.0 for seamless interfacing	Andrija Feher (SYN), Sönke Michalik (TUBS)
1.4	31/07/2015	Updated and extended section 3.3	Jesus Alonso (TTS)
FIN.	31/07/2015	Integration and finalization	Csanád Erdős (BME)

Note: Filename should be

"R5-COP_D##_#.doc", e.g. „R5-COP_D91.1_v0.1_TUBS.doc"

Fields are defined as follow

1. Deliverable number ***.***

2. Revision number:

draft version **v**

approved	**a**
version sequence (two digits)	***.***
3. Company identification (Partner acronym)	*****

Contents

List of Acronyms

GPIO	General Purpose Input/Output
HVM	Hardware Virtual Machine
IRI	Internationalized Resource Identifier
MMU	Memory Management Unit
OCU	Operator Control Unit
OGC	Open Geospatial Consortium
OKL	Open Kernel Labs
OWL	Web Ontology Language
PV	Paravirtualization
PVH	Paravirtualization in a HVM Container
QEMU	Quick Emulator
RDF	Resource Description Framework
RTOS	Realtime Operating System
RTSP	Real Time Streaming Protocol
SOS	Sensor Observation Service
TCB	Trusted Computing Base
TLB	Translation Lookaside Buffer
UGV	Unmanned Ground Vehicles
URI	Universal Resource Identifier
VMI	Virtual Machine Interface
VMM	Virtual Machine Monitor
WSDL	Web Service Description Language

1 Introduction

1.1 Summary (abstract)

The R5-COP project aims to create solutions that allow robotic system designers to rapidly reconfigure their tools using standardized interfaces and modular, exchangeable components.

This document summarizes solutions allowing *Seamless integration* of components through standardized and/or abstract *software interfaces.*

The lowest level interface pertaining to software we have identified is the one separating hardware resources and code execution. Section 2 gives a detailed overview of *Virtualization* techniques and their applicability to embedded and cyber-physical systems. These solutions aim to separate hardware and software design, and allow us to use programs written once on different kinds of CPUs and microcontrollers. Besides a general overview of virtualization we put a special emphasis on the QEMU embedded virtualization solution.

Section 3 focuses on a higher level software interfaces, namely the message oriented data exchange formats. These can take place locally or over the Internet and are closely related to Web technologies. We give an overview of these technologies focusing on semantic technologies, which encapsulate data together with its interpretation. We introduce self-describing XML based languages, the resource description framework technique, and standardized data storage and exchange formats such as SensorML created by the Open Geospatial Consortium. Finally, subsection 3.1 gives an overview of standardized software interfaces already in use in the robotic industry, and an outlook on upcoming standardized software interfaces.

1.2 Purpose of document

This document presents various interfacing techniques using virtualization and semantic technologies, and evaluates the potential gain by using them. It focuses on techniques such as virtualization, semantic web technologies.

This is a draft document of the final deliverable due in M18, which will also contain sections regarding message oriented middleware solutions.

1.3 Partners involved

Partners and Contribution	
Short Name	**Contribution**
BME	Overview and evaluation of virtualization techniques with a particular focus on embedded virtualization. Overview and evaluation of semantic web techniques providing an abstract, self-describing means for data exchange.
PIAP	Overview of standard software interfaces in the robotic industry, and examples of related technologies such as laser scanning.
TTS	A short draft related to audio/video sensor interfaces
TUBS	

2 Improving component interoperability through embedded virtualization

2.1 Virtualization in general

Before we discuss embedded virtualization and the ways it can improve component interoperability let's discuss virtualization in general. Virtualization can happen at different levels. Depending on the level at which virtualization operates we can distinguish between **system virtualization** and **process virtualization**.

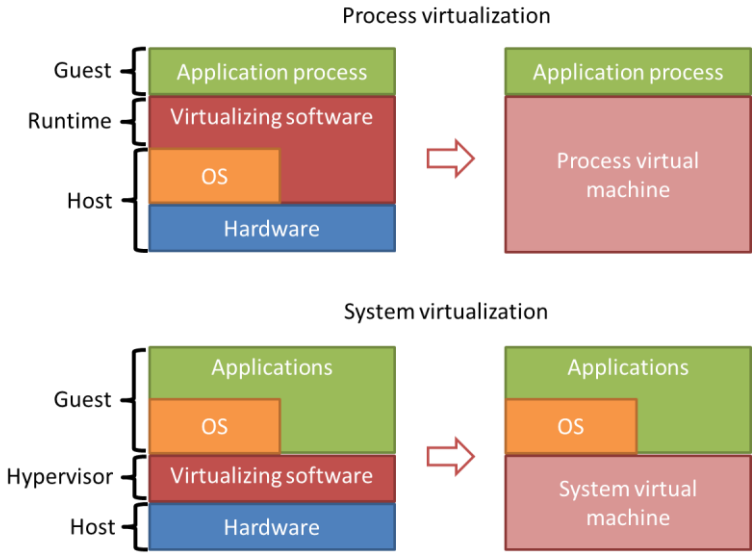


Figure 2-1. Process virtualization vs. System virtualization

From now on under virtualization we mean system virtualization. System virtualization makes an additional copy (or copies) of the environment provided by the hardware to the software running on it. This virtual environment is called **virtual machine**. And the component realizing the virtualization is called **hypervisor** or **virtual machine monitor (VMM)**. The paper of Popek and Goldberg (1974) [1] is a well cited starting point in the field of virtualization. In their work they defined three requirements against hypervisors:

1. Equivalence criteria: the virtual environment shall be essentially indistinguishable from the real one by the software.[1]
2. Efficiency criteria: the applications running inside the virtual environment suffer only minor decreases in speed.[2]
3. Control criteria: the hypervisor shall be in complete control over the resources of the real environment.[3]

[1] Implementing virtualization needs time and resources. For these reasons the virtualized environment behaves in a different way from the timing point of view compared to the real environment. Furthermore it has less resources. However from all the other aspects the virtualized environment must be identical to the real one.

[2] In practise it means that the virtualization can be considered as efficient if the majority of the instructions can be executed without any modifications.

[3] In other words the hypervisor must catch the moments when the code running in the virtualized environment wants to get information on the state of resources in the system or wants to alter it. In these cases the hypervisor must show the virtual copy of these states instead of the real ones.

2.1.1 Classic virtualization

In their paper Popek and Goldberg also declared a sufficient condition to test whether a given architecture can support virtual machines or not:

"... a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions."

By **sensitive instructions** they mean those instructions whose execution is either depending on the state of the resources in the system or whose execution alters the state of these resources. (Examples for these resource states can be the amount of available memory, the current execution mode of the processor or whether interrupts are enabled or not.)

By **privileged instructions** they mean those instructions whose execution is allowed only in the **privileged mode** of the processor (this mode often called **supervisor** or **kernel mode**). Whereas any attempt of the execution of these instructions in the **unprivileged** (also called **user**) **mode** generates an **exception** (also called **trap**). It is important to note that simply omitting the execution of these instructions in the unprivileged mode is not enough. The exception is has to be asserted in these cases.

Let's consider some software we want to run in a virtualized environment. The software can be divided into two parts. Some of the code (in our example an operating system) is executed in kernel mode (which is needed because normally the OS is supposed to be in control of the resources in a system). The remainder part of the software (in our example the processes) runs in user mode.

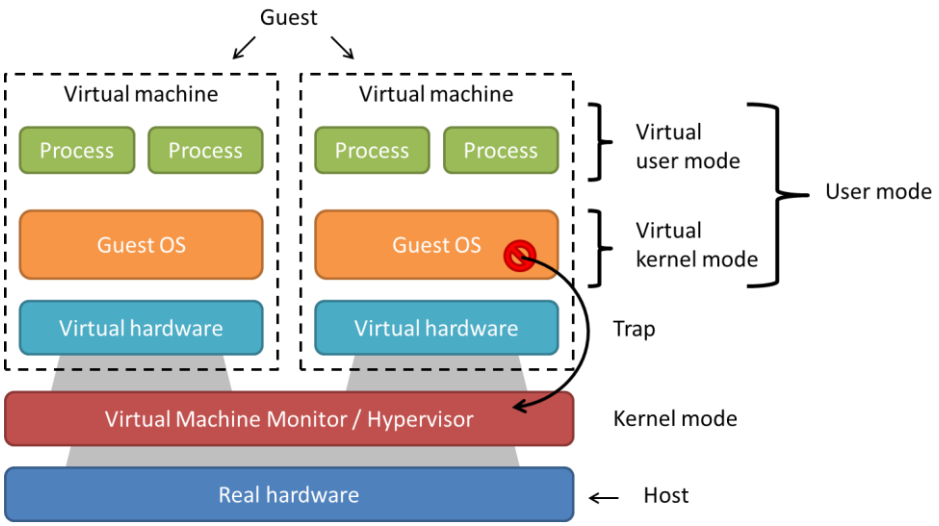


Figure 2-2. Classic virtualization

What happens if Popek and Goldberg's sufficient condition on virtualizability holds true meanwhile we try to execute the OS in user mode (in other words we deprivilege it)? Every time the OS executes a sensitive instruction (in other words it tries to read or alter the state of the resources in the system) a trap occurs. Processors jump to a well defined program memory address if they receive a trap. (This is because under normal circumstances (e.g. not virtualized) a trap usually indicates that something went wrong. The code located at that memory address is supposed to handle the situation.)

This behaviour is quite handy for virtualization purposes. After the OS failed to execute the sensitive instruction and trap has been arisen all we need to do is to put the entry point of a hypervisor to the address where the processor just jumped.

This way the hypervisor now can "fool" the OS. If it wanted to read the state of the real system the hypervisor can show the state of the virtual environment. Likewise if the OS

wanted to alter the state of the real system the hypervisor alters the state of the virtual environment.

This technique (**trap and emulate**) is also called as **classic virtualization**. In this case the hypervisor is the only component running in kernel mode. The original kernel mode is now called **virtual kernel mode** and the original user mode is called **virtual user mode**. The real system is the **host** and the virtual system(s) is (are) the **guest**(s).

It is worth to mention that not all architecture can be virtualized in the above described classic way. An example for such an architecture is the well know x86 (without the various virtualization support extensions added by Intel and AMD after realizing the lack of support for classic virtualization in the original x86 architecture).

It is also worth to note that not being virtualizable in the classic way does not mean not being virtualizable at all. Remember, Popek and Goldberg's condition on virtualization is just a sufficient condition, not a necessary one. This leaves the way free for other virtualization techniques to meet the three criteria (namely the equivalence, efficiency and control) against a hypervisor. [2]

2.1.2 Dynamic binary code translation

If the classic way of virtualization cannot be applied we can theoretically create a hypervisor using an **interpreter**. The interpreter simply **fetch**es the next instruction from memory, then **decode**s it and finally also **execute**s it. However this technique fulfils two of Popek and Goldberg's criteria (namely the equivalence and control), it cannot be said to be an efficient one. [2]

A somewhat similar technique is the **binary (code) translation**. The idea is that only those sensitive instructions need special attention whose execution in user mode does not generate a trap. All of the other instructions can be executed natively (there is no need to interpret them).

Binary translation thus has to be able to locate those problematic instructions. To make this task a little bit easier and faster, the code about to run is first divided into small chunks called **basic block**s. A basic block has exactly one entry point and exactly one exit point (like a jump, subroutine call or a return). The search for problematic instructions happens within these basic blocks. And if one is found that instruction needs to be handled. It will be usually a call to the hypervisor. Execution also returns to the hypervisor if the exit point has been reached. [3] After the problematic instruction has been replaced the effect is somewhat similar to as if a trap had been asserted (which also supposed to give the execution to the hypervisor).

To speed up this technique the translated blocks are stored in a cache. This way the negative impact on speed caused by binary translation happens only once (at the cost of increased memory consumption). As time goes on more and more part of the code will be translated. If at the end of a basic block the execution is about to move to another basic block (which has been already translated) we do not need to return back to the hypervisor. The next basic block can simply be directly invoked. [3]

It is worth to note that in virtual user mode we do not need to translate the code. Simply omitting these problematic instructions by the processor is adequate. [3] This way we further can speed up this technique by switching off binary translation (**direct execution**) every time the system running in the virtual environment switches to user mode. [2]

The method described so far is precisely called **dynamic** binary (code) translation because translation happens on the fly every time a basic block is about to be executed.

At first we may think that binary translation is a slower technique compared to the trap and emulate method. However practice shows mixed result. This is because a modern processor usually has **instruction cache** and **branch prediction circuit**. Their content is invalidated every time a trap occurs. And this is expensive. [2] For this reason a hypervisor may decide to use binary translation even in those cases where classic virtualization also could be used.

2.1.3 Paravirtualization

To accomplish system virtualization there is another interesting idea called **paravirtualization**. This technique intentionally allows the software within the virtual environment to be aware not running on the real system. Although this way we sacrifice one of Popek and Goldberg's criteria (equivalence) we can increase the efficiency criteria (as we do not need relatively expensive techniques like trap and emulate or binary translation). The guest operating system simply invokes **hypervisor calls** where it normally wanted to execute a sensitive instruction. This is somewhat similar when a process invokes **system calls** if it wants something only the operating system has right to do so. This way a **programming interface** can be defined between the operating system and the hypervisor.

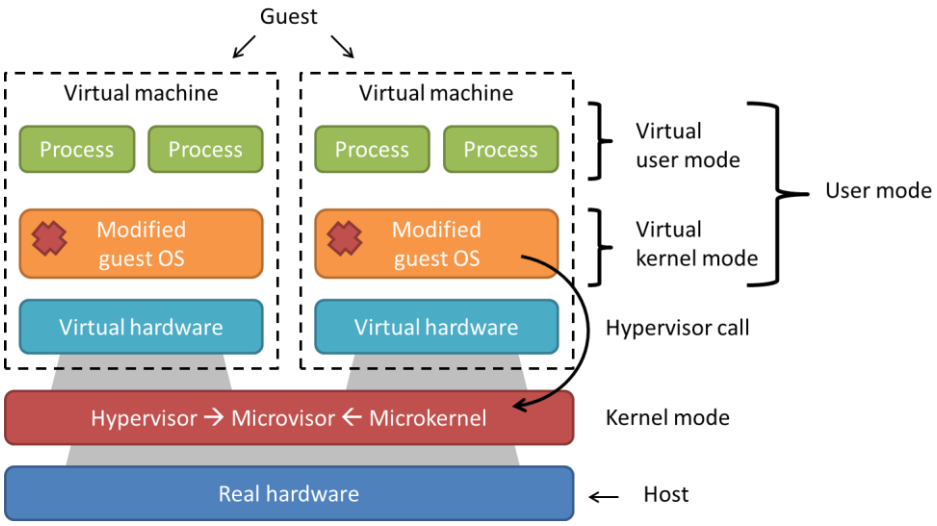


Figure 2-3. Paravirtualization

It is worth to note that in this case the hypervisor tends to be similar to a **microkernel**. A microkernel implements by definition[4] only those services that are absolutely necessary for the operation. All other service (like a device driver or a file system implementation) runs within user mode processes. In other words: a microkernel implements only **mechanisms**. But those **policies** what tell us how to use the mechanisms provided by the microkernel are not part of the kernel. (A possible example can be scheduling. In this case the mechanisms the microkernel has to provide is finding the highest priority task ready to run and to manage the context switch. However those policies what tells how to assign priorities to the tasks are implemented in user mode. [3])

Hypervisors are working in a quite similar way in the paravirtualized case: they are a relatively thin software layer running in kernel mode and provide fundamental services to the operating system above them. All of the other services are implemented within the operating system in user mode. For this reason a paravirtualized hypervisor is often called as a **microvisor** (which is a term made up from the words microkernel and hypervisor). [4]

As we can see paravirtualization requires the guest operating system to be modified (this is the cost we must pay for the increased efficiency). As soon as we modify our operating system it won't be able to run directly on the bare metal. Furthermore there exists quite a few hypervisors. This would require to modify the operating system according to every hypervisor. To overcome these difficulties a **standard virtual machine interface** can be defined. In theory operating system developers needs to modify their product for this

[4] An often cited statement for the definition of microkernels is from Joachen Liedtke: *a concept is tolerated inside the µ-kernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality*. **A megadott forrás érvénytelen.**

interface only. Hypervisor developers expected to produce a thin mapping layer which translates from the standard interface calls to the actual hypervisor calls. And hardware manufacturers are expected to produce a similar mapping layer to translate the standard

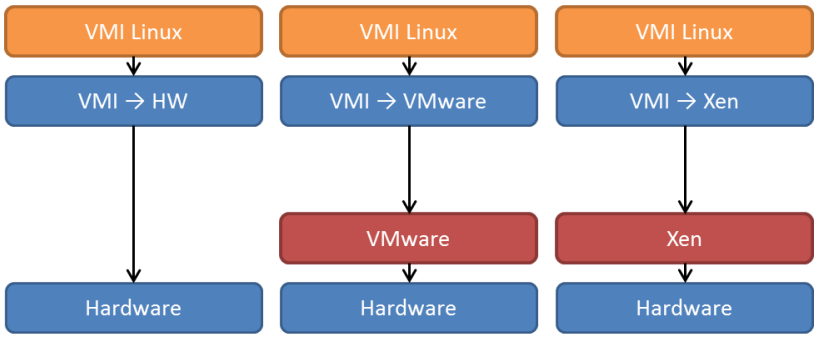


Figure 2-4. Virtual Machine Interface

interface calls to the real hardware. An example for such a standard interface is **VMI** (**Virtual Machine Interface**). It has been developed by VMware in 2006. [5]

After the appearance of paravirtualization some refer to the two older techniques (classic virtualization, binary translation) as **full virtualization** [6]. This is because those methods fulfil all three criteria on hypervisors defined by Popek and Goldberg. Whereas paravirtualization sacrifices one: equivalence criteria.

Some define classic virtualization as **pure virtualization** and call binary translation and paravirtualization as **impure virtualization**. This is because classic virtualization does not require the code to be modified whereas binary translation alters the code in runtime and paravirtualization alters the code at design time. [7]

2.1.4 Location of the hypervisor

According to the previous three chapters we can distinguish between hypervisors by the technique they implement. However there is another distinction between the hypervisors based on the position they located in a system.

If the hypervisor is the software component running directly on the hardware we call it **type-1 hypervisor** (or **bare-metal hypervisor**). If the hypervisor is a process running on top of an operating system we call it **type-2 hypervisor** (or **hosted hypervisor**).

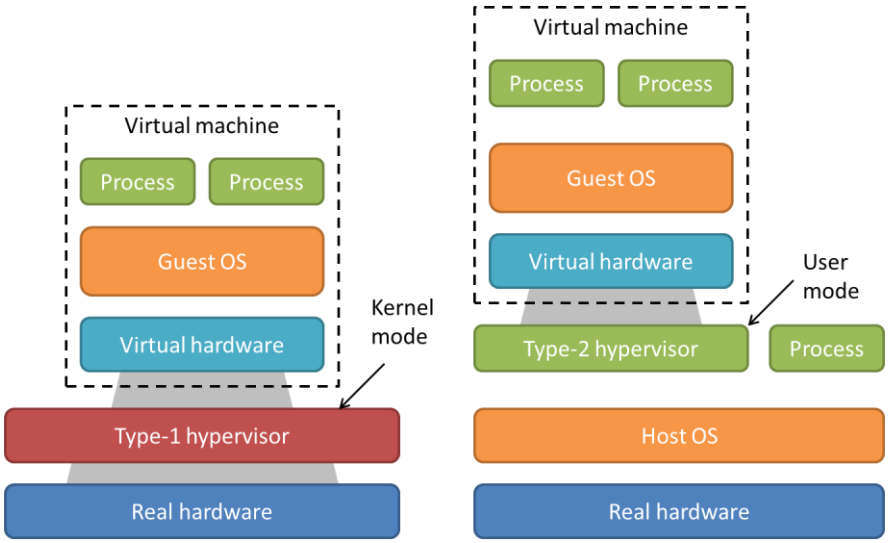


Figure 2-5. Possible locations of a hypervisor

Type-1 has the advantage of being more effective:

- It runs directly on the hardware. So there is no need to go through various operating system calls.
- Other processes do not decrease the amount of resources available for the hypervisor.

Type-2 has the advantage to be more convenient:

- The user can keep their operating system.
- It has greater hardware support as popular operating systems usually have wider support for various devices than hypervisors.

The above mentioned leads to the fact that type-1 hypervisors are more often used in case of servers (where efficiency is important but convenience is not). And type-2 hypervisors are more often used on desktop and mobile machines (where efficiency is not too important but convenience is).

Examples for type-1 hypervisors: VMware ESXi, Xen Project Hypervisor, Oracle VM Server[5], Microsoft Hyper-V, OKL4, Xvisor. Examples for type-2 hypervisor: VMware Workstation (and VMware Player), QEMU, Oracle VM VirtualBox, Microsoft Virtual PC.

2.1.5 Virtualization of other devices

The previous chapters covered some aspects of virtualization regarding the CPU. However it is worth to note that in the field of informatics a device in almost all of the cases is not just a CPU but has memory and other I/O devices as well. This means if we want to virtualize a whole system we have to virtualize the memory management and the I/O devices too. Although it is beyond the scope of this document to describe these techniques in detail it is worth to mention them at least in a few words.

2.1.5.1 Virtualization of memory management

In a modern processor besides the CPU we can found also an **MMU** (**Memory Management Unit**) device too. The aim of this device is to help with *virtual* memory management. It is very important to note that at this point the word *virtual* has nothing to do with the field of virtualization we mentioned in this document so far. The word *virtual* refers to a technique which makes it possible for an operating system to show address spaces to the processes independent from the physical memory. These address spaces are continuous, independent from each other and usually has the extent of the whole address space the processor is able to address.

The above mentioned means that first, these *virtual* memory address spaces have to be mapped to physical memory locations second, not all *virtual* memory can be mapped directly to physical memory. In this case an auxiliary device (like a HDD) with greater storage capacity but lower speed is used to store data which is not mapped directly to physical memory.

The mapping information is stored in **page table**s. These data structures are located in memory and every process has one. The CPU uses ***virtual* address**es and the MMU translates these addresses to **physical address**es using the mapping information stored in page tables. And also the MMU is the device which eventually puts these physical addresses to the memory address bus of the system.

If we want to virtualize this memory management we can define a very similar task to the one described above. The hypervisor needs to map the physical addresses valid within the virtual machines to the physical addresses valid within the real system (for the sake of simplicity we call this latter **machine address**).

[5] Internally uses the Xen hypervisor.

The problem is that without modification the MMU cannot be used for this task by the hypervisor. This is because the MMU not just translates between the virtual addresses to physical addresses but also puts the physical addresses to the memory address bus.

For this reason the first technique invented by software engineers is the use of **shadow page table**s. These tables are administered by the hypervisor and maps the *virtual* addresses to machine addresses. To accomplish this task the hypervisor needs some trickery to catch the events when the guest operating system updates its page tables and shall update its shadow page tables accordingly. This process has certain computing overhead and also needs some extra memory as to every page table belongs a shadow page table. So we need as many shadow page tables as many processes exist in the sum of the virtual environments.

After realizing the computational and memory overhead caused by the technique of shadow page tables hardware engineers extended the functionalities of the MMU by adding support for **second level address translation**. The second level page tables map physical addresses to machine addresses. An MMU supporting second level address translation is capable of using the original page tables (mapping from *virtual* addresses to physical addresses) in combination with the second level page tables to eventually map between *virtual* addresses to machine addresses without software intervention by the hypervisor.

This way both the computing and memory overhead is much less because this technique needs only one second level page table per virtual machine compared to the case of shadow page tables when as many shadow page tables are needed per virtual machine as many processes are located within the virtual machine in question.

However this technique also has a disadvantage compared to the shadow page tables. This is because speeding things up MMU-s usually have a cache for recently used physical addresses. This cache is called **TLB** (**Translation Lookaside Buffer**). This is a relatively small but fast buffer (compared to the main memory). This way the MMU needs to go through a relatively bigger page table located in a relatively slower memory only if the TLB does not contain the already translated physical address (called a TLB miss). Using the technique of second level address translation has a performance penalty on TLB misses compared to the case of shadow page tables. [3] [8] [9]

2.1.5.2 I/O device virtualization

If there are some I/O devices in the system we might want to virtualize them as well. There are mainly three techniques: **direct I/O** (or **I/O pass-through**), **full emulation** and **paravirtualization** (or **split driver**).

The first method (direct I/O) is not really a virtualization technique. It accomplishes just an allocation task. The only thing the hypervisor is expected to do is to grant exactly one virtual machine direct access to the I/O device in question. After that there is no more intervention by the hypervisor. This technique is by far the most efficient and simplest one as there is no virtualization at all. If at any given time only one virtual machine is expected to handle the I/O device this method is suitable.

But if not we have to virtualize (e.g. duplicate) the device in question. Compared to the one extreme of not virtualizing at all the opposite extreme is the full emulation. This means that the hypervisor is expected to emulate every aspect of the given device for all of the virtual machines that wants to use such a device. A good example can be a virtual disk image file. This technique has the advantage that every virtual machine can possess this kind of a device (not just one compared to the direct I/O case). Another advantage can be that the emulated device is usually a very simple (thus general) implementation. This usually means that more system has support for it compared to a very exotic real device. In the other hand this technique needs the biggest overhead.

The third method is in between the above mentioned two extremes. It is the paravirtualized case. It is also called the split driver technique because the driver is actually composed of two parts. One part (name it the front-end) is located in the virtual machines. These parts

communicate with the other part (name it the back-end) running as part of the hypervisor. The back-end's task is to communicate with the real hardware and to multiplex the device accesses from the various virtual machines. This case is called paravirtualized because a special front-end driver is needed in the virtual machines. And this driver is fully aware that it is not in full control of the given device.

There is a subcase of paravirtualized I/O in some hypervisors where a special virtual machine contains the back-end driver. Having such a special virtual machine has the advantage of simpler hypervisor development as there is no need to make device drivers for all of the exotic devices. An already existing OS with great device support can be used instead. However this advantage comes at the cost of increased code size and memory footprint introduced by an extra virtual machine compared to the case when the device drivers are part of the hypervisor itself.

In the case of I/O virtualization problem can arise with DMA capable devices as these devices are able to directly access memory under normal circumstances. These memory accesses have to be caught by the hypervisor to translate them to machine addresses before actual DMA occurs. Although this task can be accomplished by software it poses a performance penalty.

A so called **IO MMU** device can ease up the burden of DMA address translations for the hypervisor. An IO MMU uses page tables to translate memory accesses from I/O devices in a very similar way as an MMU uses page tables to translate memory accesses from the CPU. (Using an IO MMU has advantages for an operating system too. [10] If the hypervisor allows the guest operating system to use the IO MMU for its own purposes then the task of virtualizing the IO MMU has to be accomplished. This can happen in a very similar way as with MMU virtualization: either maintaining shadow page tables or using second level address translation if the IO MMU supports it.)

2.2 Embedded virtualization

Embedded systems usually differ in some properties from the desktop and enterprise devices. For this reason the virtualization techniques used in the latter case usually cannot be applied without modification. They need to be adapted to the embedded world.

2.2.1 Properties of embedded systems

Traditionally embedded systems are designed to fulfil a very specific task. They have a strong connection to the environment surrounding them: they continuously observe it via their various sensors and if necessary also intervene via their actuators. Although in some cases they can fulfil their task in a pure hardware based way (using ASICs or programmable logic devices like FPGAs or CPLDs) it is far more common to have some kind of a processor and thus embedded software (called **firmware**). A traditional embedded system is a compact and closed device. Its hardware and software components are usually cannot be changed by the user. In most of the cases they have dedicated power supply thus power consumption is a critical aspect. Their task is usually simpler (compared to the desktop or enterprise case). For these two reasons the processing units of traditional embedded systems are usually less powerful (lower clock frequency, fewer pins, lower power consumption) and their operative memory is also smaller. And last but not least it is worth to mention that their task is often safety critical and they often face real-time requirements.

Meanwhile traditional embedded systems continue to exist we can observe the trend that more and more embedded devices are nowadays much liker to a personal computer (just in a miniature form) than to a traditional embedded system. This trend produces a **continuous range** with traditional embedded systems at one end, personal computer like devices at the other end and a lot of devices in between.

If we investigate this range from the virtualization point of view we can state that traditional embedded systems simply do not have enough resources to support virtualization. The other end of the range is able to support virtualization even without modification. The more

interesting domain is the devices in between. A good example can be a smart phone. As it is a cell phone we can think of it as a traditional embedded system. We expect it to operate as a phone with high reliability, long battery life and without the need to periodically update its firmware. For the phone task itself this device does not require powerful resources. However at the same time we think of it as a smart device. For this reason we expect it to be a perfect mini personal computer with rich features, powerful resources and highly customizable software. These devices have enough capabilities to support virtualization however still do not have as powerful resources as their desktop and enterprise relatives. This implies that the techniques used for virtualization need to be customized to fit within the limited capabilities of resources in this segment.

2.2.2 Motivations of embedded virtualization

Why can be virtualization attractive for embedded systems? Let's see a few examples: [7]

- **Multiple concurrent operating systems**: staying with the cell phone example it can be seen that the two-sided expectations of a reliable phone and a versatile mini computer can be fulfilled appropriately only if we choose the right OS for each of the functions. (For the phone function usually a tiny but efficient embedded real-time OS (RTOS) is the right choice meanwhile for the mini computer function a feature rich application OS (like embedded Linux).)
- **Security**: even if we do not want to use different OS-es simultaneously virtualization can be a benefit for security reasons. This is because virtual machines are relatively separated from each other (compared to applications residing on the same OS). Thus if the software within one of the virtual machines fails (either because of a bug or an attack) the failure is more probably can be contained only within the virtual machine in question.
- **License separation**: sometimes licensing issues can arise if we want to use proprietary software along with open-source (like GPL) software. Separating these two components into different virtual machines can be a solution.

2.2.3 Limitations of traditional virtualization

Let's see what are those limitations that make it either impossible or inconvenient to use traditional virtualization in embedded devices located at the middle of the above mentioned range: [7]

- **Granularity vs. performance**: each time we decide to put a software component to a separate virtual machine the hypervisor needs to maintain another copy of the real environment. But what is maybe worse is that most likely our software component needs an OS too which needs to be now duplicated. In an embedded system poor on resources we simply cannot afford too many components to put into separate virtual machines.
- **Integrity**: meanwhile in the enterprise (or even desktop) use case usually independent services are running within the virtual machines, the components of an embedded software are much more linked to each other (they have a much higher level of cooperation):
    - **High-performance communication**: to be effective these components implement high-performance communication between them. Communication between virtual machines are very expensive. If we put these components into separate virtual machines we would get highly degraded communication performance.
    - **Sharing I/O devices**: in embedded software I/O devices often have to be shared between components (either simultaneously or in a time-sharing method). If one virtual machine gets the I/O device exclusively others cannot access it (direct I/O). If the hypervisor is about to fully emulate the device it

has great performance penalty. If the hypervisor uses a paravirtualized approach the hypervisor has to be equipped with a suitable device driver (which is not trivial if we consider the vast variety of embedded systems and their devices). If the paravirtualization puts the back-end driver to a dedicated virtual machine we face the already discussed problem of high-performance communication.

  - **Integrated scheduling**: a traditional hypervisor can schedule only virtual machines and the OS-es running inside the virtual machines have the duty to schedule tasks. This approach is not always sophisticated enough. In some of the cases we have to think about the system as a whole and have to schedule the tasks not depending on which virtual machine they are running.

- **Security policies**: in many cases there are strict rules on communication. Not all components are allowed to communicate with any of the other. Traditional virtualization has nothing to do with such restrictions.
- **Trusted Computing Base (TCB)**: consider a security critical service. The TCB (regarding this service) consists of all the code whose malfunction can compromise the service in question. Besides the code of the service itself what can be counted as part of its TCB? All of the other code on which the service is depending. Despite the very rare cases when a code is mathematically proven to be correct we must assume that a code contains bugs. The easiest way to reduce the potential number of bugs is to reduce the size of the code. Traditional virtualization even increases the TCB by adding the layer of the hypervisor.

2.2.4 Requirements against embedded virtualization

Regarding the above mentioned limitations of traditional virtualization it is needed to be customized for embedded systems considering the following requirements: [11]

- **Efficiency**: as embedded systems have less resources an embedded hypervisor has to be small and efficient (specifically regarding memory usage).
- **Security**: to make it possible for the embedded system to meet security requirements even with virtualization the TCB has to be small.
- **Isolation**: the embedded virtualization technique has to be able to separate software components from each other.
- **Communication**: however between those components that are allowed to communicate with each other the embedded virtualization must provide efficient communication methods.
- **Real-time operation**: the embedded hypervisor has to be able to schedule real-time which implies that the virtual machines cannot be black boxes for the hypervisor (integrated scheduling).

Regarding the efficiency and security requirements type-1 hypervisors are the better solution.

Regarding the isolation and communication requirements our idea is to decompose the embedded software into small components. Between these components isolation and efficient communication has to be provided. Because these components can be relatively small it would be beneficial if the components were able to run even directly on the hypervisor without a complete virtual machine container with an OS.

For the above reasons it is advisable to implement the hypervisor with a microkernel. [7] [11]

Although the above conclusion seems logical it has also disadvantages. Microkernel based hypervisors usually choose paravirtualization which requires the modification of the guest operating system. If we want to implement integrated scheduling even further modifications are required to the kernel of the guest OS-es. And last but not least decomposing the original embedded software to tiny components needs additional development time. For these reasons others rather not advise using a microkernel as the hypervisor. [12]

If someone is interested in this topic more deeply they can observe a scientific battle on this field. [13] [14]

As we can see both the microkernel and the monolithic kernel approach for a type-1 hypervisor can have advantages over the other. For these reasons let's see too examples for each of them: the OKL4 for a microkernel based design and the Xen for a more monolithic approach. After them we also mention QEMU. Although this latter is a hosted hypervisor it has a lot of interesting operating modes among which the capability to execute a binary image complied for a different hardware architecture. This property makes it easy to reuse a component complied for a different architecture.

2.2.5 OKL4 microvisor

This microvisor is developed by Open Kernel Labs[6]. The company was founded in 2006 as a spinout from NICTA[7]. In 2012 it has been acquired by General Dynamics[8]. The company has a great success in embedded virtualization. According their press release from 2012 [15] the OKL4 software has reached 1.5 billion deployments in mobile devices. Even the term microvisor was coined by them. [11]

Unfortunately this hypervisor is not open-source. However it is worth to mention it as it has been designed alongside those clear ideas that concluded the usage of a microkernel as the hypervisor. The structure of OKL4 can be depicted as the following picture.

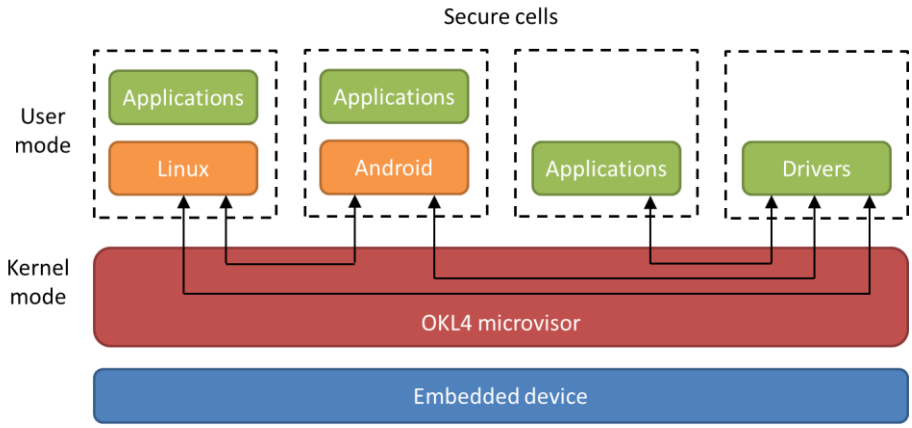


Figure 2-6. The structure of OKL4

As the name implies OKL4 is part of the L4 family of microkernels (which is the work of Joachen Liedtke). The components of an embedded software can be put into so called **secure cell**s. These cells can contain a whole software stack having an OS (like with virtual machines we already know) but can support components (applications or device drivers) even without an underlying guest OS. Within the cells efficient **inter process communication** methods have been developed. Because device drivers can run in separate secure cells these communication methods really have to be efficient as in this case they serve as I/O communication pathways. [11]

The OKL4 microvisor uses paravirtualization. OK Labs has been created the following paravirtualized guest OS-es: OK:Linux, OK:Android and OK:Symbian. [11] The release of OK:Windows is also planned. [16] It is interesting to note that on ARMv5 platform they were able to achieve nearly 50 times performance increase in lmbench context-switching tests

[6] http://www.ok-labs.com/

[7] http://www.nicta.com.au/

[8] http://www.generaldynamics.com/

compared to native Linux. This proves that a relatively small code is much easier to implement efficiently. [7]

The microvisor also has efficient resource sharing mechanisms. It is possible to share memory buffers between the secure-cells (producer-consumer concept) in a way that the consumer has only read access rights. As with this solution there is no need for copy devices

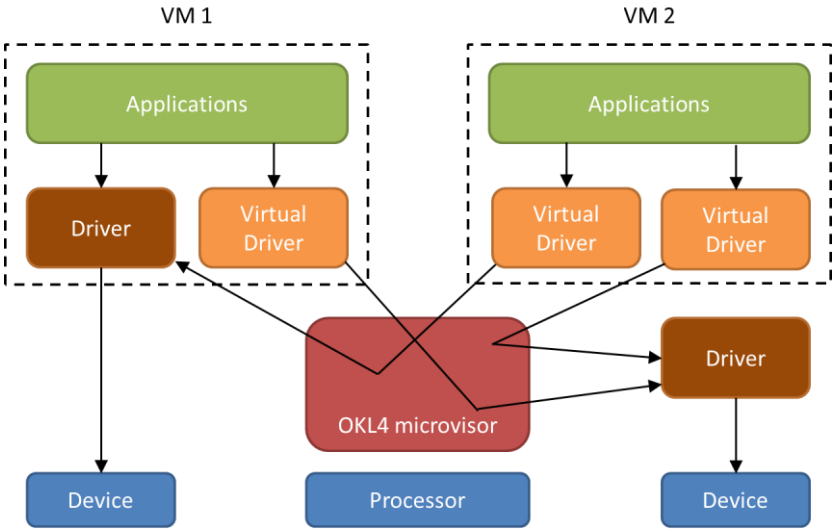


Figure 2-7. OKL4 device driver sharing

with high transfer rates can profit from this concept. Another example for the efficient way of resource sharing is the access to device drivers. These device drivers can be part of an operating system or stand-alone. [7]

The kernel is capable of integrated (real-time) scheduling. Consider the case when there is a feature rich, user friendly application OS (not facing real-time requirements) and another, simpler, lightweight real-time OS for time critical tasks. If we want to assign priorities our first idea may be to assign the virtual machine running RTOS globally a higher priority. However this is not always a good idea. Consider some background task in the RTOS. They supposed to run only when there is no other tasks ready to run in the system. If there is only the RTOS running it works as expected. However if we have another virtual machine with the application OS and think of these two virtual machines together as the system the background task of the RTOS can starve all of the application OS tasks. Furthermore we can think of the case of a soft real-time task (like video playback). This can be accomplished more easily under an application OS.

For these reasons it is important for the microvisor to implement integrated scheduling. An example for assigning priorities for the individual task within the system (rather than assigning priorities to the virtual machines first and then to the task running within them) can be seen on the following picture. [7]

OKL4 supports ARM, MIPS and Intel processor architectures.

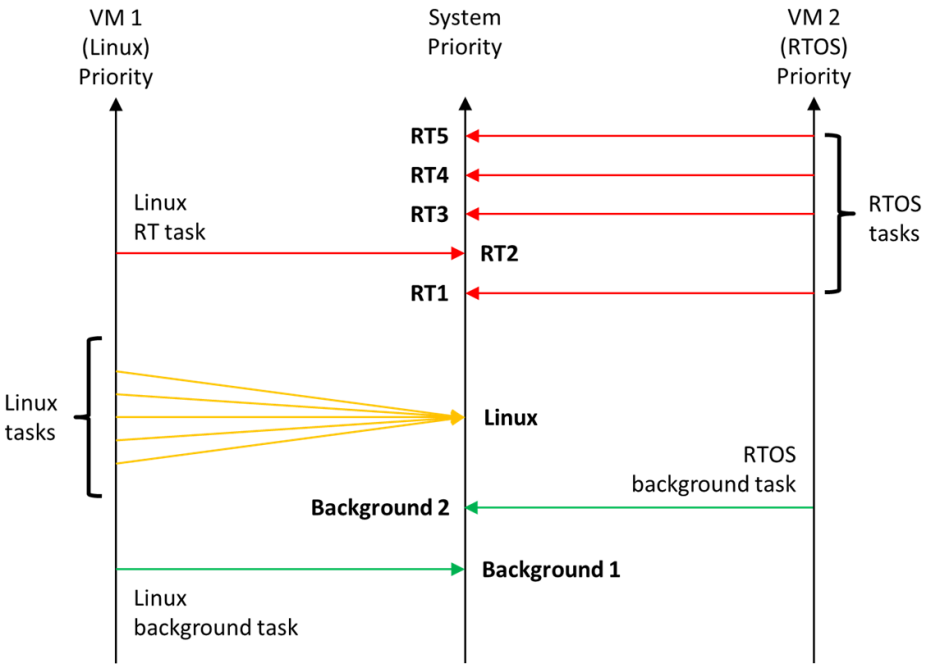
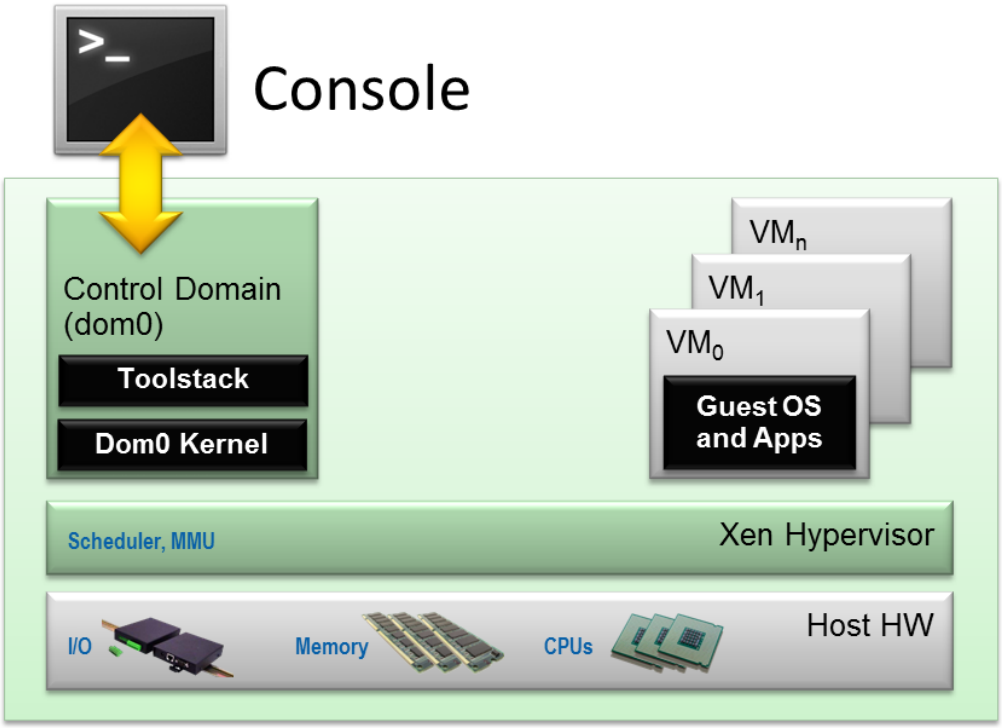


Figure 2-8. Integrated scheduling in OKL4

2.2.6 Xen hypervisor

The Xen hypervisor was originally created for x86 architecture as part of a research project on Cambridge University at the late 90s. In 2002 it became open-source. The first public



2-8. Figure: The structure of Xen Project hypervisor

Author: Lars Kurth; Licence: Creative Commons Attribution ShareAlike 3.0

URL: http://wiki.xenproject.org/mediawiki/images/6/63/Xen_Arch_Diagram.png

release was in 2003 (the developers consider this date as the “birthday” of the hypervisor).

Version 1.0 was released in 2004 and soon after version 2.0. After that one of the founders and other developers have founded XenSource, Inc. as a spinout company to make a commercial product out of the research project. Fortunately the hypervisor remained open-source. Version 3.0 was released in 2005. In 2007 the company has been acquired by Citrix Systems, Inc. A fork project to support the ARM platform (in a paravirtualized way) was started in 2008 by the leadership os Samsung Electronics. Version 4.0 was released in 2010. Out of the box Linux support for the Xen hypervisor has been highly increased in 2011. In the same year an experimental prototype for a hardware assisted ARM hypervisor has been announced. In 2013 this prototype has become functional. Xen Project has become part of the Linux Foundation in the same year. [17]

The structure of the Xen Project can be observed in the picture above. The Xen hypervisor is a type-1 hypervisor. So this software layer handles the CPU, memory and interrupts. The virtual machines are located above this layer. The Xen terminology usually refers them as **domain**s (or guests). There is a special domain named **domain0**. The purpose of this virtual machine is to contain the device drivers and a component named **Toolstack** (or **Control Stack**) to manage the rest of the virtual machines. This management software can be accessed using either command-line interface or GUI (it can also be integrated to some cloud based management service.

The Xen hypervisor can use paravirtualization or classic virtualization (if the architecture supports it). It is even possible to use more virtualization techniques if there are more than one virtual machine in the system. Furthermore even if basically classic virtualization is used for a guest it is allowed to use some paravirtualization methods as well.

The virtualization modes of Xen are namely:

- **HVM** (Hardware Virtual Machine): in this case hardware assisted virtualization is used to virtualize the CPU and the page tables. All other devices are emulated using QEMU [ref].
- **PV on HVM**: in this case hardware assisted virtualization is used to virtualize the CPU and page table (as in the above case). However some of the devices uses paravirtualized drivers.
- **PVH** (PV in an HVM container): in this case hardware assisted virtualization is used to virtualize the CPU and page table (as in the above cases). However all other devices uses paravirtualized drivers.
- **PV** (ParaVirtualization): for every component in the system paravirtualization is used as the virtualization technique.

The following table summarizes the above mentioned:

VS: Virtualized (SW) VH: Virtualized (HW) P: Paravirtualized	**Disk and network**	**ITs, timers and spinlocks**	**Motherboard and system startup**	**Privileged instructions and page tables**
FV | HVM	VS	VS	VS	VH
PV on HVM (Win)	P	VS	VS	VH
PV on HVM	P	P	VS	VH
PVH	P	P	P	VH
PV	P	P	P	P

2-1. Table: Comparison of Xen virtualization techniques.

2.2.7 QEMU

QEMU is the abbreviation for “Quick EMUlator”. Although the name refers to **emulation** QEMU can be a hosted (type-2) hypervisor too (this happens if the target architecture is the

same as the host one). It works on x86 and PowerPC architectures and currently being tested on ARM, HPPA and Sparc platforms. It can emulate x86, ARM, MIPS, PowerPC, Sparc, MicroBlaze and Xtensa architectures with high levels of success (and Alpha, CRIS, M68k and SH4 with varying level of success). [18] It is free and open-source.

It has basically two modes of operation: [18] [19]

- **User-mode emulation:** it is an application level emulation. With the help of QEMU one can run a process compiled for a given architecture on another architecture. Although the processor architecture may differ the used operating system has to be the same. System calls are altered to fix any endianness or 32/64 bit issues.
- **System emulation:** it is a platform level emulation. It means a full system (with peripherals) is emulated. Different operating systems can be booted.

Besides these two main modes of operation QEMU can be integrated into other virtualization software: [19]

- **KVM[9] hosting:** if QEMU is used in conjunction with KVM its purpose is to set up and migrate KVM images. It is involved in emulation of the hardware however the execution of guest code is done by KVM (as requested by QEMU).
- **Xen hosting:** as the previous chapter already mentioned Xen may use QEMU to emulate peripherals. In this case execution of guest code is done by Xen and thus totally hidden from QEMU.

The next chapter describes QEMU user-mode emulation in details (considering emulation speed and the possibilities of seamless execution of foreign binaries).

2.2.8 Using QEMU processor simulator software as a method of computing node virtualization

To achieve architecture independence between the different types of nodes in a sensor network, processor virtualization can also be used alongside or instead of a virtual machine or a scripting language.

Processor virtualization has many benefits compared to the other two mentioned possibilities.

For example, the Java Virtual Machine (not to be confused with the Java-like Dalvik) or the .NET framework are usually not available on embedded platforms. Also, in general, it is hard to write direct hardware accessing code in either Java or a scripting language, which makes in the end reverting to native code under the hood. In some cases, the source code of a legacy embedded software is not accessible, but the processor platform is changed. With virtualization, porting legacy software can be easier. Finally, many embedded programmers still prefer to use C as a primary programming language.

It is somewhat clear that native languages will remain the primary choice on embedded platforms, however the code and binary portability is a problem, especially on distributed systems. A processor simulator which seamlessly integrates into the system can solve portability issues.

2.3 QEMU emulator

For processor virtualization, the QEMU [20] emulator can be used. One of the benefits of QEMU is the great number of host and guest architectures it support via its TCG (Tiny Core Generator) binary translation mechanism. QEMU is also one of the components of the KVM (Kernel-based Virtual Machine) used in server virtualization.

One the greatest advantages of QEMU in embedded systems is the Linux user mode emulation, which allows the user to spare computing power and storage space.

[9] http://www.linux-kvm.org/page/Main_Page

In Linux user mode emulation, QEMU executes only the user program on the guest architecture, however, every Linux system call and IO control is passed to the host Linux kernel, thus allowing greater computation speed and better interaction between the host and guest system. One can use QEMU to access the host hardware IO space without having to create the appropriate virtual hardware for QEMU.

With Linux user mode emulation, application portability simplifies: a Linux application compiled on a given platform can be executed on a different architecture, albeit with a performance loss.

2.3.1 Benchmarking QEMU emulation

To measure the computing capabilities of a QEMU-emulated embedded platform, two different methods were used, reflecting the usual tasks of an embedded system.

The first benchmark measures the computing performance of the QEMU userspace emulation, the second simple benchmark measures the IO performance.

The benchmarked system is a Raspberry Pi board with Raspbian Linux distribution. The QEMU guest architecture emulated on the Raspberry Pi is little endian Xilinx MicroBlaze processor (microblaze architecture).

2.3.2 Benchmark of the computing performance

To benchmark the performance of the emulated architecture, three measurements were done. The first measures the native ARM processor performance, the second measures the emulated Xilinx MicroBlaze processor performance, the third measures the native performance of a 75 MHz Xilinx MicroBlaze processor.

Since the goal of this benchmark is to expose the theoretical upper limit of the computing system, and not the measurement of the performance of a specific task (for example video stream compression) the NBench [21] synthetic computing benchmark program was used.

One of the advantages of the NBench is its simplicity. The tests included in NBench are simple enough to be portable, but complex enough to test the processing power of the system under test. Also, NBench is designed to be scalable: it includes a dynamic workload adjustment, which allows the tests to adjust to the capabilities to the system, while providing consistent results.

The following table shows the benchmark results:

Task	Native (operation/s)	QEMU (operation/s)	MicroBlaze(operation/s)
Numeric sort	237	10	15
String sort	38	1	1
Bitfield manipulation	9.324e7	1.45e6	8.41e5
Floating point	47	2	4
FFT	2612	1	1
IDEA encryption	791	3	12
Huffman coding	487	7	9
LU decomposition	84	1	1

Without any hardware acceleration support, results of QEMU are in the magnitude of the performance of the native MicroBlaze running on 75 MHz.

2.3.3 Benchmark of the IO performance

We have seen that the performance hit of QEMU without any acceleration is high. However, the advantage of the user mode Linux emulation is to enable hardware virtualization without operating system support.

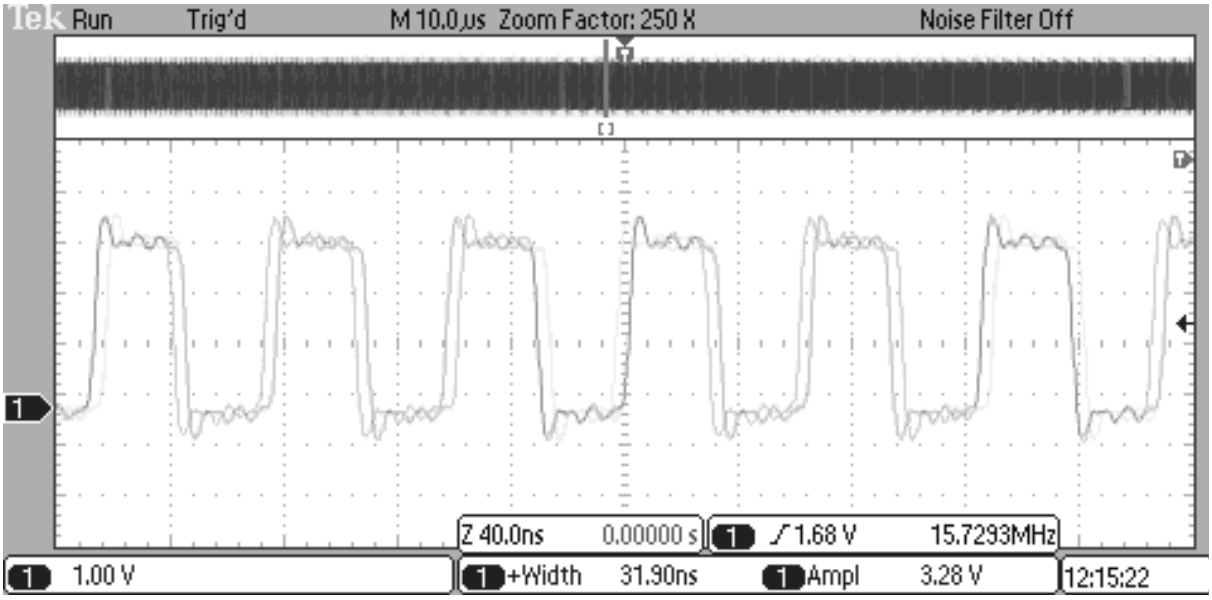
On the Linux operating system, the IO access is done through special device or other system (e.g. SysFS) files. This gives us a great advantage: since every hardware IO from the userspace application is done with IO system calls (*open, close, read, write*), with the QEMU user mode emulation an application executed on a simulated processor can access the real hardware. QEMU translates all system calls from the emulated architecture to the host.

This is the base of the IO benchmark. Since the QEMU Linux user mode emulation passes the guest system calls to the host kernel, software which relies on system calls and host kernel functions (for example an embedded software doing IO) can perform better than a computational software.

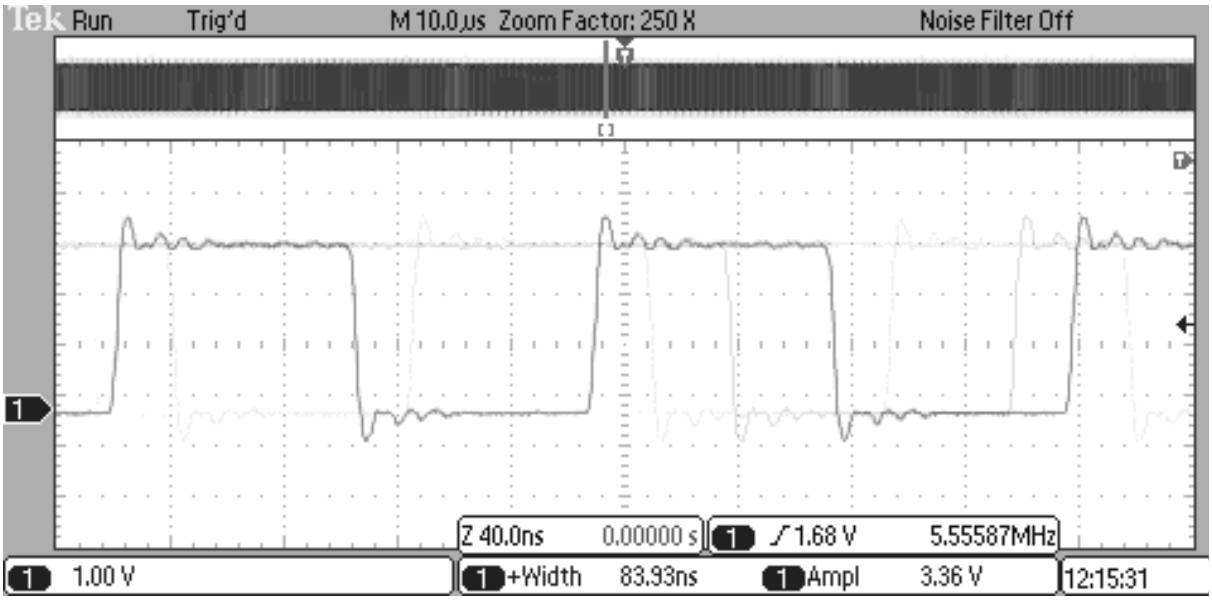
The measurement simply tests the maximum achievable GPIO signal frequency by continously toggling a GPIO bit of the BCM2835 chip used on the Raspberry Pi board.

Problems of this measurement method are that the TCG used in QEMU works especially good with tight loops [22], so we measured something which can be described as the “best-case” scenario. However, since the most common IO tasks are built up from small loops, the measurement reflects the most common use cases.

The result of the measurements are the following: With a native software running on the processor, a frequency of 15-16 MHz can be achieved, in accordance to earlier benchmarks done on the Raspberry Pi by Joonas Pihlajamaa. [23]



1. Illustration: GPIO output frequency of native ARM



2. Illustration: GPIO output frequency of emulated MicroBlaze

On the other hand, the emulated MicroBlaze is capable to generate a 5-6 MHz square wave.

It is worth to be noticed that the relative inaccuracy (~20-25) of the frequency of the square wave generated by the virtualized system is much greater than the relative inaccuracy (~6-8%) of the frequency of the native code generated signal, mainly because of the inaccuracy of the running time of the dynamic recompilation done by QEMU.

In some applications, this inaccuracy is not tolerable.

2.3.4 Integrating foreign architecture binaries to the host

The main advantage of the QEMU Linux user mode emulator is that our native software – although with a performance hit – could run independently on different architectures. This is not unlike Java's "write once, run anywhere" philosophy, with the difference that low level operations are much simpler.

For this approach to work, an elementary operating system support is needed.

An application – especially on UNIX and UNIX-like operating systems – is usually built up from many smaller lesser modules, which use each other. This is usually done from the program with the *exec()*-family of system calls, but executing an application can also be done from a shell script.

Usually, when we try to execute a foreign binary (for the example, we will call it *binfmt*) from the shell, it will not succeed:

```
└──> debug $ >> ./binfmt
bash: ./binfmt: cannot execute binary file
```

To seamlessly integrate a native application coming from a foreign architecture, we have to support its binary format. The support should be done on the kernel-level, since we want to integrate an unmodified native application into the system.

2.3.5 Introduction to Linux foreign binary format support

The UNIX operating system (on which Linux is also based) supports the automatic execution of interpreted languages from the shell. This is done with the support of so-called *shebang lines*. If an interpreted script starts with the special characters *#!* , and afterwards it states the path of the interpreter, it automatically executes it properly:

```
#!/usr/bin/python
print "Hello"
```

For a scripted language, which is a text file, usually having "#" as comment this is a feasible way. However, for a binary executable which has a strict format, insertion of a shebang line is not possible.

The problem could be solved by modifying the Linux kernel to remove the shebang line from the binary before execution. However, if the kernel is to be modified, it is deserving to modify it to enable a higher level of functionality. The support which is already done by kernel developers is called *binfmt_misc.*

2.3.5.1 binfmt_misc

The early versions of Linux used the so-called *a.out* format for its binary executables. However, later versions migrated to the ELF (*Executable and Linkable Format*) binaries. To maintain backward compatibility, support for *a.out* still exists.

Because support for two binary formats already exists in the kernel, it is worth to have a more general implementation, which can handle arbitrary binary formats. This is the *binfmt_misc* module.

Execution of a binary module is built up from two parts: recognition of the binary format, and execution of the specific interpreter or simulator.

binfmt_misc does the recognition part: from a given format description, it recognizes the binary format, and select the specified interpreter for it. Afterwards the Linux kernel executes the interpreter.

The configuration file for *binfmt_misc* resides in the */proc/sys/fs/binfmt_misc* directory. Each file in this directory gives information about the support for a given binary format. For example the support for Python 2.7 binaries is the following:

```
└──> binfmt_misc $ >> cat python2.7
enabled
interpreter /usr/bin/python2.7
flags:
offset 0
magic 03f30d0a
```

This shows that every file which starts with the magic number 0x3f30d0a is launched with the Python interpreter. This means that – in contrary to Windows – can launch executables based on the content of the file, not on the extension of it.

To register a new format, we have to write into the *register* file. The format of the file is the following:

```
:name:type:offset:magic:mask:interpreter:flags
```

1. *name* is the name used for identification.
2. *type* is M or E. M means recognition based on the content of the executable, E means recognition based on the extension of the executable.
3. *offset* gives the number of bytes at the start of the file which should be ignored.
4. *magic* is the magic number we should match
5. *mask* tells which bits in the magic should not be ignored
6. *interpreter* is the executable which will launch the binary
7. *flags* unused values

If we write an entry into the *register* for the foreign ELF format, we can assign the QEMU Linux user mode emulator as interpreter, thus enabling seamless integration.

The specification is available at: http://www.skyfree.org/linux/references/ELF_Format.pdf

The information we use from the ELF is the following:

- Format identifier: 0x7f “E””L””F”
- Word length
- Byte endian
- Executable type
- Architecture information

For a big-endian MicroBlaze the *register* format will be the following:

```
:mbelf:M:0:\x7fELF\x01\x02\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\xbd:\xff\xff\xff\xff\xff\xff\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\xff\xff\xff\xff:/usr/bin/qemu-microblaze:
```

To modify it for a different foreign architecture, the following should be edited:

1. The first \x01 means that the platform is 32 bit
2. The first \x02 means big endian
3. The last \xbd means MicroBlaze
4. The simulator is */usr/bin/qemu-microblaze*

If we load *binfmt_misc*, the execution of a foreign binary will work:

```
└──> debug $ >> ./binfmt
Hello
```

2.3.6 **Conclusion**

Two disadvantages were found using QEMU as a Linux user mode emulator. The first is that the computing performance is much lower than the native performance. For simple control and measurement software this performance limit can be tolerated.

The greatest disadvantage is that the QEMU Linux user mode is somewhat underdeveloped compared to recent Linux kernels, many newly implemented system calls and IOCTL's are yet cannot be passed to the host kernel. For example support to pass the USB IOCTL calls (used by userspace USB drivers) are not yet implemented.

Nonetheless, for simple tasks where portability and direct hardware access is important, QEMU can be an option.

We also explored the possibilities to execute a foreign binary on the host seamlessly. The Linux operating system gives support to execute foreign binary format files.

3 Abstract interfaces for exchanging sensor information

3.1 Semantic web technologies

The continuous advances in computing allow researchers to access an ever increasing pool of data. In order to handle the growing volume and complexity of the available information new approaches were required. The goal is to automate data manipulation so that computer programs can – in a limited way – reason and deduce facts and discover previously unknown relations.

The Semantic Web [24] is an effort coordinated by the World Wide Web (W3C) consortium to create common data formats including semantic content. This inclusion should allow for a structured representation of our knowledge that is self-describing and can be interpreted by computers.

There are a number of concepts and standards that pertain to the Semantic Web initiative, the most notable ones are the XML document formats and languages, Resource Description Frameworks (RDF) and the Web Ontology Language (OWL). These technologies are receiving a lot of support from within and without the W3C consortium, and already have a significant adoption rate in the IT sector.

In this section we will have a special focus on a collection of standards originating from the Open Geospatial Consortium, namely the XML based SensorML language and the Sensor Observation Service standard.

3.1.1 Resource Description Framework

In order to better understand the importance of Resource Description Frameworks [25] and RDF databases it is best to look at the change they represent regarding our approach to collecting new information.

The standard approach was limiting our scope to well-defined, and tightly-categorized information. In order to expand our knowledge, newly acquired data had to conform to our previous definitions and had to be described according to previously agreed categories. Schematic databases such as SQL fit this informational approach well. The basic unit of information is a triple of *entity-attribute-value.* A natural way of storing such information is a *table.*

model	**power**	**torque**	**top speed**
Ferrari F50	513 hp	470 Nm	270 km/h
Fiat Punto	75 hp	86 Nm	172 km/h

Figure 3-1. Schematic table for car

Records of the `car` table correspond to actual entities of cars. The columns of a table correspond to the attributes associated with the entities, and each record has a value corresponding to each entity.

A schematic database is a very efficient tool for a wide range of applications, but has inherent limitations when it comes to representing new information of a different nature. This schema for example does not allow us to store the angle of the rear-spoiler. We could add a new attribute, but that would require modifying the whole table, and not all cars have a rear-spoiler, thus the attribute for the Fiat Punto would be meaningless. A perfect solution differentiating between street and sport cars would require modifying the schemes of multiple tables and even creating new ones.

As network enabled intelligent devices are becoming a cheap commodity a new approach to information management is being adopted by cutting-edge tech companies. This is the *open-ended* informational approach, which handles information knowing, that it is only a small

portion of all available knowledge, and expects that our knowledge will be continuously expanded in an organic way.

The basic unit of information in an RDF is a triple of *subject-predicate-object.* Instead of a table the most natural representation of an RDF database is a directed, labeled graph of such triples.

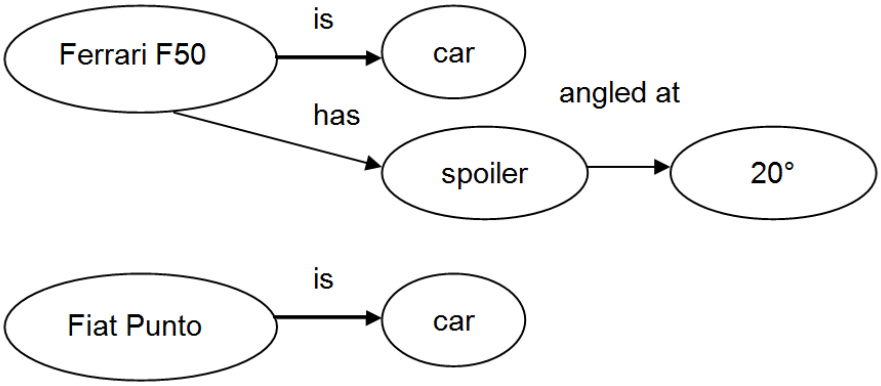


Figure 3-2. A section of an RDF database shown as a directed graph

In this model adding information of a new quality such as a spoiler angled at 20° can be done without affecting the rest of the database.

RDF is meant to be a standard model for data interchange on the web. It offers data merging of data with differing schemas and can naturally support the evolution of schemas. The triples can describe two abstract or concrete entities connected with a relation. All three can described by Internationalized Resource Identifiers (IRI) which are a Unicode generalization of Universal Resource Identifiers (URI). Nodes can be also datatyped literals, such as string, number or date.

The current specification is RDF 1.1 which has been released in 2014. [26] It defines an abstract syntax (a data model) which serves to link all RDF-based languages and specifications. A widely used implementation is the Stardog database, which is free to use under a limited number of connections.

3.1.2 Extensible Markup Language

XML is a text base document format meant to be readable by both humans and machines. [27] It has a strong support for Unicode text. While originally meant for describing documents over time it became a widely used format for the representation of arbitrary data structures. It has been derived from SGML (ISO 8879).

Over time it has become a standard format for data exchange between web services. It is the language for widely used document formats such as RSS, Atom, Soap or XHTML.

XML is capable of conveying syntactic and semantic constraints. The basic rules of XML define syntactic correctness, also referred to as well-formedness. All XML documents or messages must be *well-formed* and this property can be checked according to the general XML rules.

Semantic rules are conveyed through XML schema systems. An XML schema is a description of a type of XML document that define constraints regarding the structure and contents of the document. There are multiple schema languages such as Document Type Definition (DTD) or XML Schema (with a capital S).

The constraints defined by schemas can be lexical (allowed types and literal values at given places), grammatical (defined order of appearance of certain types) and referential (a given value must exist at some other defined location). Hence these schemas define XML based languages that can be adapted to practically any domain.

Verifying that a given XML document conforms to the syntactical and semantic rules of an XML based language is called validation and is carried out by *validating XML parsers.* While

well-formedness can be determined independently from a given language validity can only be confirmed knowing the full ruleset of the language.

The rules and validation process of domain oriented languages make XML a very flexible format, resulting in documents that are particularly suited to automatic parsing and self-describing knowledge representation. It also helps reduce errors by enforcing a set of rules in the form of communication itself. By making inconsistent commands and information inexpressible they help create more robust systems.

3.1.3 Sensor Model Language

SensorML [28] is an XML based language standard created and maintained by the Open Geospatial Consortium (OGC). It is a robust, semantically-tied means for *defining processes and processing components*.

SensorML is intended to be used for expressing and communicating information regarding measurements and post measurement transformation of observations. It can describe sensors, actuators as well as computational processes pertaining to pre- and post-measurement tasks.

The language references properties of various sensors and processes in a self-describing format i.e. using only SensorML documents and messages a knowledge graph of all involved entities can be constructed.

The preferred format of element descriptions is OWL URIs, which stands for Web Ontology Language. [29] OWL descriptors use a computational logic-based language such that knowledge expressed in OWL can be exploited by computer programs. It contains sequences of annotations, axioms and facts. Annotations are meta-data meant for aiding human understanding, while axioms and facts can be classes, properties and individuals identified by further URIs creating a complex knowledge graph. The ontologies pertaining to such descriptions can give us means to discover and categorize sensors and relevant processes.

The current version of the standard is 2.0 published in 2014.

3.1.3.1 Sensor Observation Service

The SensorML language plays an important role in another standard of OGC, that is the Sensor Observation Service (SOS).

The SOS is meant for applications in which sensor data needs to be managed in an interoperable way. It describes a web service interface which allows querying observations and sensor metadata. The preferred format for sensor metadata is a SensorML document.

The SOS acts as a special datastore that can be accessed with self-describing XML messages pertaining to observations of physical quantities, events or computational processes yielding useful information.

3.1.4 Web Services Description Language

Web Services Description Language (WSDL) is an XML based standard for specifying and describing the functionality of a web service. The WSDL also means the concrete description of a specific web service and sometimes called *WSDL file*. This XML based file can be requested from the service and describes the following:

- how the service can be called
- what parameters it expects (in what format)
- what data structures it returns.

These 3 elements can make us associating to the method signature of a typical programming language. The main advantage of using WSDL is that web services can provide information about themselves in a machine readable manner.

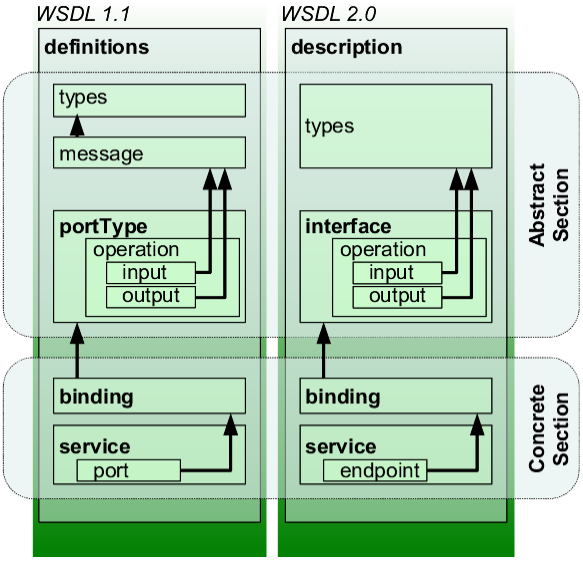


Figure 3-3. Representations of concepts defined by WSDL documents

As we can see on the above figure, WSDL documents can be separated to two parts: a concrete section and an abstract section.

The abstract section declares reusable *interfaces* (port types in 1.1) through which the service can interact with clients. An interface can have one or more *operations* (with input and output). These operations correspond to functions or methods in programming languages, while the interface can be perceived as a function library or class. Both inputs and outputs are constrained by the *types* also specified in the abstract section of the WSDL file. While inputs correspond to the parameter list and outputs to the return value, types can be perceived as the primitive or complex types of a programming language.

The concrete section describes the bindings of the reusable interfaces to concrete service endpoints (with network address). One interface can be bound to more network endpoints.

WSDL is often used in combination with SOAP and an XML Schema to provide machine readable format for service description. Since 2007 the last version (2.0) of WSDL is W3C recommendation.

3.1.4.1 Semantic Annotations for WSDL and XML Schema

Semantic annotations for WSDL and XML Schema (SAWSDL) defines a set of extension attributes for the Web Services Description Language (WSDL) and XML Schema definition language. Application of the attributes shall allow for description of additional semantics of WSDL components. The specification defines how semantic annotation is accomplished using references to conceptual semantic models, e.g. ontologies. SAWSDL does not specify a language for representing the semantic models. Instead it provides mechanisms by which concepts from the semantic models can be referred using annotations.

3.2 Message orientated Middleware for seamless interfacing

3.2.1 Features of Message oriented Middlewares

Robotic systems require significant interaction and coordination of hardware and software elements. In the context of software engineering, the concept of middleware earned a very strong role in the entire software development process.
In robotic applications the use of a middleware can help improving the organization, the maintainability and the efficiency of the code that controls the robot. Robotic systems are usually complex systems built on many different hardware and software components, as sensors and actuators as well as planners and control algorithms. In general, on each robot runs a software that is responsible for reading sensors data, extracting the needed information from them, computing the sequence of actions to accomplish a given task and controlling the actuators to execute the actions.
Using a custom approach, there will be a single monolithic application that will handle all these tasks, making code maintenance hard and preventing every form of code reuse and sharing between different projects. Such a scenario, with many hardware and software components that needs to communicate and collaborate to reach a goal, is exactly where a middleware can help improving the organization, the maintainability and the efficiency of the code. The whole application can be structured into many little concern separated tasks, as ”get a sensor reading”, ”extract features from some data”, ”drive the motors to some speed”. Different components can exchange data using a common communication channel provided by the middleware, using interfaces that are consistent between different applications.
In this way, it becomes really easy to share and reuse code among different projects, or change an algorithm to get some functionality as it is only necessary to keep the same interface. As an example, if you need to switch from a proximity sensor to another, it is possible to write a new component that share the same interface and update it without modifying the rest of the application. This concept can be extended to large and complex applications, in which using a middleware can clearly improve the overall code organization and reduce the programming effort.

3.2.2 Advantages of ROS 2.0

The ROS 2.0 version addresses a set of issues to improve overall performance of the ROS MoM. The major improvements are:

- multiple robots networks
- embedded platforms support
- Real-time features
- Non-ideal networks support (recovery from data loss/data delay)

The additional support of multi-master networks and embedded platforms in ROS 2.0 directly addresses the use of ROS 2.0 for robot swarms or sensor networks due to ability to connect multiple embedded sensors to a combined sensing network.

With the ROS 2.0 improved communication stack and its Data Distribution Service (DDS) as communication layer, the ROS message services will be also able to handle time-critical transactions e.g. sensor messages and it will be tolerant against data loss and data delay to allow communication via non-ideal networks (wireless networks). This will enable enhanced sensor fusion of multiple robots by using wireless data distribution.

3.2.3 ROS 2.0 bridging

The preliminary version of the ROS 2.0 includes bridging features to support communication for ROS-to-ROS2 and ROS2-to-ROS as part of the "ros1_bridge" package. The current simple prototype contains:

- a bidirectional bridge which can only pass along a single message type on a fixed topic.
- two unidirectional bridges which can only pass along a single message type on a fixed topic.

Hence, the basic communication between ROS and ROS2 needed for sensor data transfers is supported.

3.2.4 ROS as communication layer for sensor data distribution

The ROS middleware is a well-known component of robotic systems that allows message based communication between software modules. The comparison of Message orientated Middlewares (MoM) included in [53] gives an overview of available middlewares for robotic systems. As result of the comparison the authors underline the advantages of ROS for the targeted robotic application.

Using of ROS in robotic systems leads to a common message format that is used to exchange data between software components (nodes) inside the robot. As an example for data transfers as part of the MoM ROS, the laserscanner data is transformed to a ROS message type to transfer it from the laserscanner sensor to the SLAM software component.

If we want to improve the sensor data distribution over the ROS communication layer it is beneficial to create sensors that can directly send ROS messages to avoid the transformation step as preprocessing of the sensor data.

As proposed in WP21, the TUBS stereo camera system is able to communicate as ROS master or slave with a connected robotic system. The camera data is transmitted in form of a ROS message, so that other processing entities can directly use the information without previous conversion. Figure 3- shows the architecture of the TUBS stereo camera platform with embedded ARM processor for communication purpose as part of the sensing system.

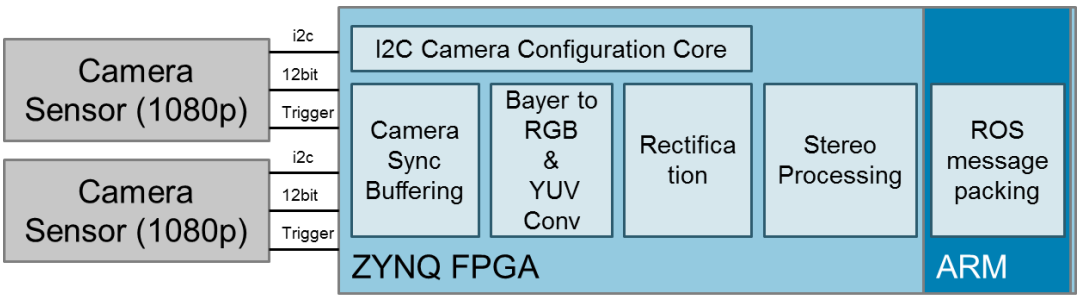


Figure 3-4 - Architecture of the TUBS stereo camera platform (WP21)

The ROS message packing is done by and ROS node running on the ARM processor with an integrated ROS indigo version. This enables the seamless interfacing of the sensor to ROS enabled robotic systems. Moreover the presented sensing system is able to run ROS-bridging features to support ROS 2.0 messages. The Gigabit Ethernet connection of the sensor allows framerates up to 60 frames per second for the transfer of 720p resolution depth images (grayscale, 8bpp).

3.3 Sensors interfaces in security robots

3.3.1 Current state of art

The only commonly recognized standard in software layer is JAUS. Currently there is open implementation of JAUS available called JAUS tool set [8], and there are several offered by software vendors like OpenJAUS [30]. JAUS is standardized as SAE AS-4 JAUS family of standards [31]. This protocol independent, so can be used over various lower layers, Ethernet or serial connection, etc., but most often JAUS systems are Ethernet based.

Designed to be open architecture, communication protocol oriented JAUS employs service oriented architecture. It's services are defined in JAUS Service Interface Definition Language which provides XML schema for describing services. Those descriptions are often used by code generators to create interfaces stubs. Later can also be used for validation purposes.

JAUS standards also define core services[10], that provide following mechanisms:

1. Transport – gateway for messages entering and leaving components
2. Event – event based communication mechanism with requests
3. Access control – implementation of mutual exclusion
4. Management – access to information on components, and life cycle management
5. Time – time synchronization
6. Liveness & discovery – allows discovery and state determination. [32]

One example of European Union funded projects that used JAUS was TALOS, where interfaces between OCU and UGV were defined according to this standard. [33] [34]

[10] SAE AS5710A

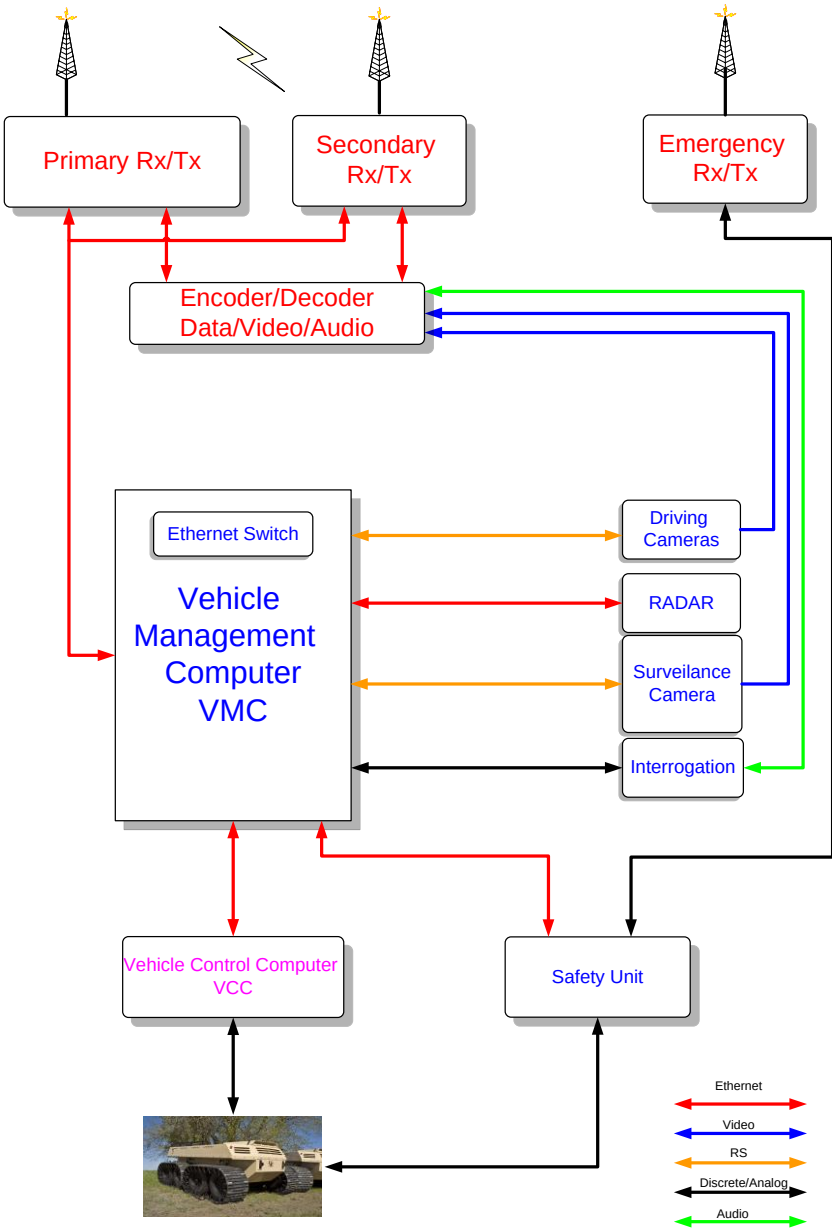


Figure 3-5. Example architecture of JAUS system - TALOS architecture

3.3.2 Interoperability profile

Recently new set of standard was released by United States Military (more exactly by Robotic Systems Joint Project Office) , called Interoperability Profile (v1), based on JAUS (but also defining mechanical and electrical interfaces), and chances are high that it will be enforced, at least in NATO countries (mostly because no competing standard exists). This standard not only specifies communication between UGV and OCU [35] but also communications with various on board sensors. [36]

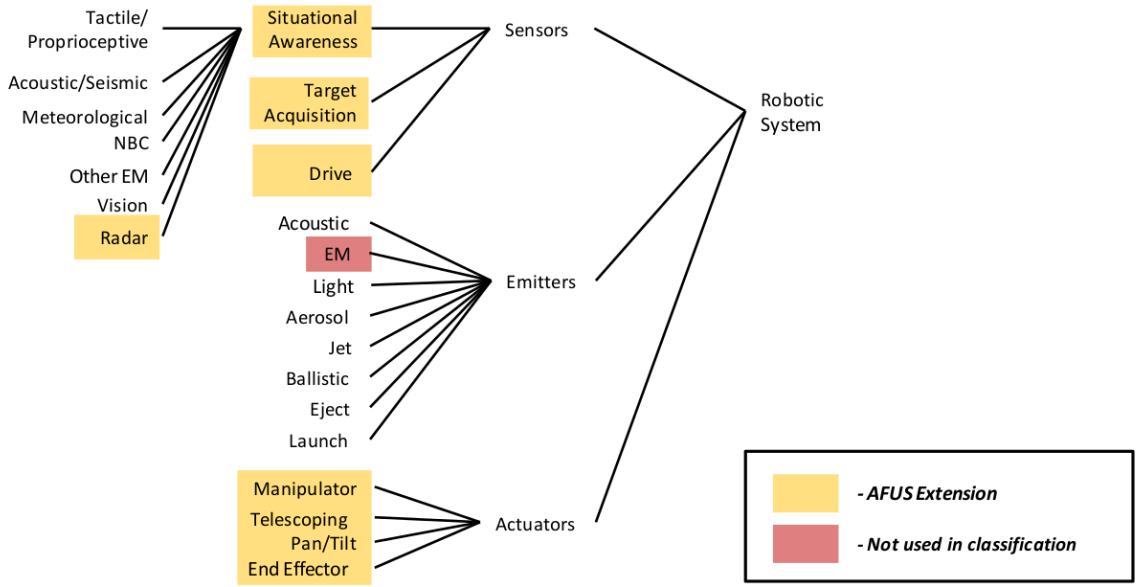


Figure 3-6. Payload taxonomy as defined in IOP [36]

Interoperability Profile mostly relies on existing standards where possible, for example:

1. Video - all IOP compliant digital motion imagery sensors must support H.264 or MPEG2 video standards, it also allows analog video transmission using NTSC standard. Video camera control can be done either using messages defined by JAUS, or by RTSP protocol.
2. Still images - for still images JPEG or JPEG2000 should be used.
3. Audio - for audio transmission Vorbis codes should be used. Transmission can be controlled either using messages defined by JAUS, or by RTSP protocol.

3.4 Sensor interfaces used in development

There are many robotics frameworks that were developed to provide some common ground for various software components. Some more popular frameworks include:

- Microsoft Robotics Studio [37]
- Open Robot Control Software (OROCOS) [38]
- Carnegie Mellon Robot Navigation Toolkit (CARMEN) [39]
- URBI [40]
- Mobile Robot Programming Toolkit [41]
- ORCA [42]
- Cross Platform Software for Robotics Research (MOOS) [43]
- Robot Operating System (ROS) [44]

There is lot of overlap and redundancy between those projects. Also multiple connections between them exist, allowing either component sharing or coexistence of frameworks in same system.

Recently most commonly used software frameworks for developing robots is ROS and OROCOS [38] [44]. Since ROS framework was chosen for R5COP project, so it is very important to discuss information on how sensors data are exchanged there in details.

In ROS for sensor messages specific package (called sensor_msgs) exists [45], and for all basic sensors there is already message specification. [46]

3.4.1 Point clouds

Point clouds are used to represent data coming from various kinds of sensors, like lidars, time of flight cameras, structured light cameras or stereo vision cameras (but for later two depth images are often used).

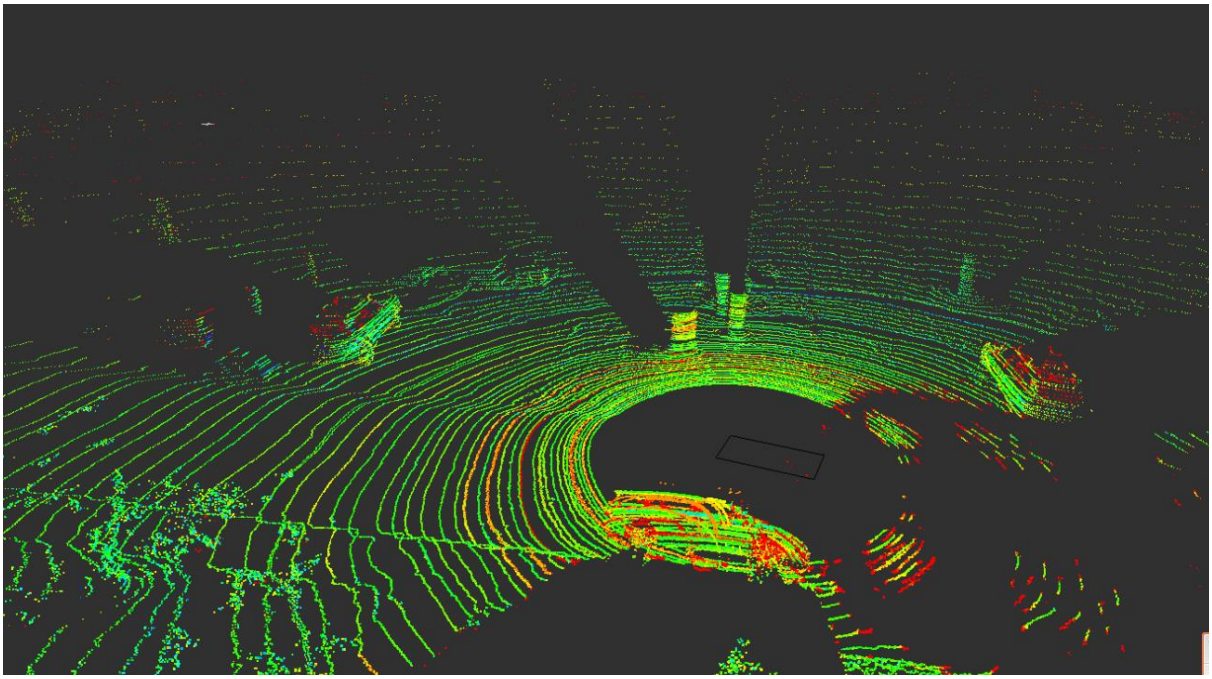


Figure 3-7. Point cloud generated by Velodyne 64E sensor

Popular library for handling point cloud is PCL, that aims to allow large scale 2D or 3D point cloud processing [47], with various algorithms for data processing (eg. filtering, feature estimation). This library is integrated into ROS, and can be used for almost any task related to point cloud handling. This integration allows passing point clouds using standard ROS publish/subscribe mechanisms.

3.4.2 Depth maps

Depth images represent data that are similar in nature to point clouds, are less generic, but also less verbose, and often easier to use. [48]

3.4.3 Laser scans

Laser scan can also be represented by point clouds (or even depth maps), but often is more convenient to use structure that is easier to manipulate with, and less verbose. Normally laser scan is structured as two tables, containing ranges and other containing intensities (lidars often produce not only information about range to obstacle, but also intensity of reflection). This data together with angle range and angle increment information provide full data on single laser scan.

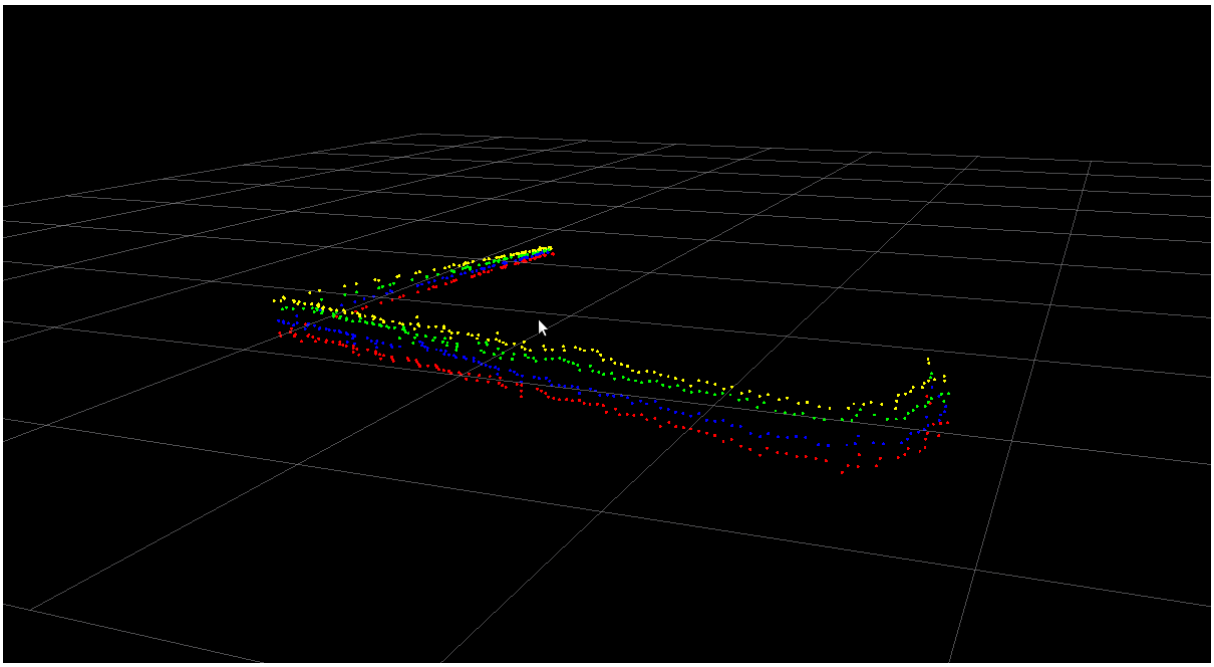


Figure 3-8. Four lasers scans generated by IBEO sensors

3.4.4 Transforms

For exchanging sensor data ROS also utilities transform system. Transform are used for manipulating information about sensor position, and allows translations from one reference frame to another. This is important for doing any kind of sensor fusion for which data from sensors have to be translated do common reference frame.

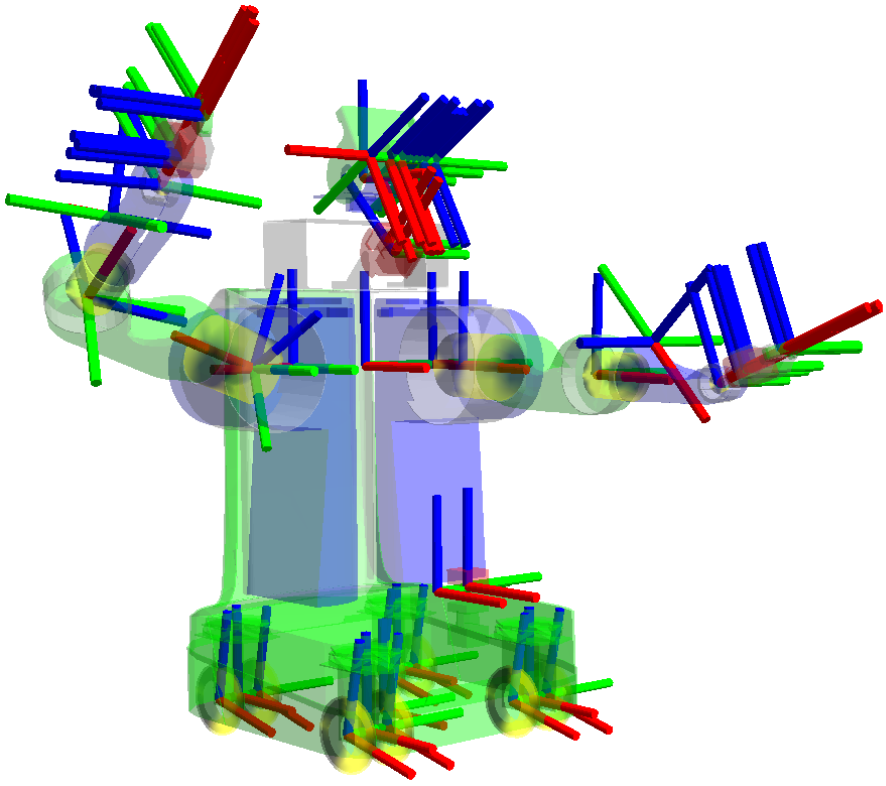


Figure 3-9. PR2 Robot with transforms displayed [57]

3.4.5 ROS Sensor Interfaces

R5-COP is a project based on ROS, therefore we will focus now in several aspects related to sensors, video and interfaces.

In ROS systems, nodes communicate with each other by publishing messages. ROS platform provides several Message Ontologies. *Sensor_msgs* is the one related to messages for representing sensor data [45]. Sensors_msgs package defines messages for commonly used sensors, including cameras and scanning laser rangefinders.

Among others, the package provides the following message types: *Image, CompressedImage, PointField and PointCloud.*

3.4.6 JAUS-ROS Bridge

JAUS is so far the most recognized software platform in the industry. ROS is gaining momentum. In order to integrate modules of both platforms, OpenJAUS has released *jROS*, a software library to bridge JAUS and ROS [49].

jROS allows users to combine the power and flexibility of ROS with the maturity and robustness of the JAUS standard. In such a way, ROS tools for visualization, navigation and perception can be used in JAUS-based system. The library jROS consists of a set of ROS messages and services which are defined with respect to the JAUS message structure.

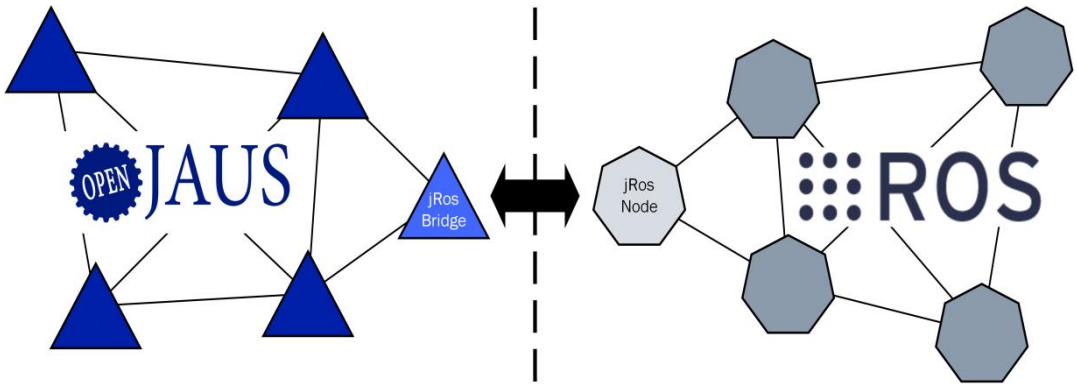


Figure 3-9. JAUS-ROS Bridging

3.4.7 ROS Video Sensors

Some camera vendors have provided a ROS package to control and use their cameras. For instance, Axis has published the software package Axis_camera [50]. The package includes Python ROS drivers for accessing an Axis camera's MJPG stream. It also allows controlling the PTZ parameters of their cameras.

3.4.8 Image Processing in ROS

The images capture by sensors can be processed with the ROS package *image_pipeline stack* [51]. The package processes raw camera images into useful inputs to vision algorithms: rectified mono/color images, stereo disparity images, and stereo point clouds. Components include:

- Camera Calibration for both monocular and stereo cameras.
- Monocular processing: remove camera distortion, color interpolation for Bayer pattern color cameras.
- Stereo processing: to produce disparity images and point clouds.

- Depth processing: for processing depth images (as produced by the Kinect, time-of-flight cameras, etc.), such as producing point clouds.
- Visualization: for viewing an image topic. It also includes a stereo_view tool for viewing stereo pairs and disparity images.

3.4.9 Video Streaming

There are several software packages at ROS.org that provide video streaming functionality.

The *ros_web_streamer* [52] provides a video stream of a ROS image transport topic that can be accessed via HTTP. It opens a local port and waits for incoming HTTP requests. As soon as a video stream of a ROS image topic is requested via HTTP, it subscribes to the corresponding topic and creates an instance of the video encoder. The encoded raw video packets are served to the client. The web streamer tries to minimize internal coding latency by avoiding a B-frame encoding scheme and by forcing the codec to keep its internal network buffer as small as possible.

image_transport [53] provides classes and nodes for transporting images in arbitrary over-the-wire representations, while abstracting this complexity so that the developer only sees sensor_msgs/Image messages

3.4.10 Communications channels bonding

A big portion of the data generated by security robots may be video (e.g. in explorer robots or in robots aimed at deactivating bomb threats) or combined audio & video outputs. Delivering such amount of data via a 3G/4G channel may be problematic: the channel throughput may be insufficient or the cellular network could be down. The solution to such issues is channel bonding in order to get the needed throughput in the uplink as well as resilience, higher availability and sub-second latency.

LiveU/Tellence [54] introduced such a concept in the market. The solutions of this company provide multichannel aggregation of different technologies (3G,4G-LTE, WiFi) over multiple carriers. This creates a reliable, broadband video uplink pipe over multiple narrow-band pipes. The solution uses adaptive algorithms which combine available communications channels and video encoding to achieve high bandwidth and smooth transmission, even as bandwidth and signal levels change across the different connections..

This type of multi-link bonding solutions have been developed for broadcasters and online video professionals and are a viable alternative to costly satellite uplink or portable satellite TV equipment. The challenge in R5-COP would be to integrate such a concept into a security robot for both analog A/V input or digital video sensor interfaces (SDI, HDMI, DVI, etc).

Using channel bonding the video stream is divided among the different channels and reconstructed on destination. The same approach can be used to transmit big amounts of information generated by a large number of (non-video) sensors.

The channel bonding system that is in fact an industry de-facto standard can be packed to be a ROS component in order to make it easy to integrate with other vendors in the robotics industry (ROS and JAUS via the JAUS-ROS bridge).

Security robots may become more autonomous when they don’t depend on the availability of a single channel and/or cellular operator. This might enhance the communications schema of systems like TALOS [33] [34] which uses JAUS standard and whose architecture was introduced in section 3.3.1 .

4 Summary

This document presents an overview of software interfacing technologies and evaluates their potential uses in creating reconfigurable robotic systems.

The techniques introduced range from low level solutions such as virtualization abstracting hardware capabilities, and high-level methods, such as self-describing, ontological data representation for data exchange between processes.

Besides giving a technological overview we also focused on standardization and industrial acceptance. While low level virtualization solutions are not yet present in robotic systems it may be a viable route if we want to significantly increase these systems reconfigurability. The presented semantic technologies already have a very strong industrial adaptation among web technologies, and standards such as SensorML may bring them closer to the field of cyber-physical systems.

5 References

[1] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM,* pp. 412-421, July 1974.

[2] K. Adams (VMware) and O. Agesen (VMware), "A comparison of software and hardware techniques for x86 virtualization," in *ASPLOS XII Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* 2006.

[3] A. S. Tanenbaum, Modern Operating Systems, 3rd ed., 2007.

[4] G. Heiser and B. Leslie, "Convergence Point of Microkernels and Hypervisors," in *APSys '10 Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, 2010.

[5] Z. Amsden, D. Arai, D. Hecht, A. Holler and P. Subrahmanyam, "VMI: An Interface for Paravirtualization," in *Proceedings of the Linux Symposium*, Ottawa, 2006.

[6] Xen Project, "Xen Terminology," [Online]. Available http://wiki.xenproject.org/wiki/XenTerminology. [Accessed 28 January 2015].

[7] G. Heiser, "Virtualization for Embedded Systems WP," 27 November 2007. [Online] Available: http://www.ok-labs.com/_assets/image_library/virtualization-for-embedded-systems1983.pdf. [Accessed 09 June 2014].

[8] O. Agesen (VMware), "Software and Hardware Techniques for x86 Virtualization," 2009 [Online]. Available http://www.vmware.com/files/pdf/software_hardware_tech_x86_virt.pdf. [Accessed 22 June 2014].

[9] W. Xiaolin, "Selective hardware/software memory virtualization," in *VEE '11 Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2011.

[10] "IOMMU," Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/IOMMU. [Accessed 25 June 2014].

[11] M. T. Jones, "Virtualization for embedded systems - The how and why of small-device hypervisors," IBM, 19 April 2011. [Online]. Available http://www.ibm.com/developerworks/library/l-embedded-virtualization/. [Accessed 25 June 2014].

[12] F. Armand and M. Gien, "A Practical Look at Micro-Kernels and Virtual Machine Monitors," in *Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE* 2009.

[13] S. Hand, "Are virtual machine monitors microkernels done right?," in *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, 2005.

[14] G. Heiser, "Are virtual-machine monitors microkernels done right?," in *ACM SIGOPS Operating Systems Review*, 2006.

[15] Open Kernel Labs, "Open Kernel Labs Software Surpasses Milestone of 1.5 Billion Mobile Device Shipments," 19 January 2012. [Online]. Available: http://www.ok-labs.com/releases/release/ok-labs-software-surpasses-milestone-of-1.5-billion-mobile-device-shipments. [Accessed 28 January 2015].

[16] General Dynamics Broadband, "OK:Windows," [Online]. Available: http://www.ok-labs.com/products/ok-windows. [Accessed 28 January 2015].

[17] Xen Project, "History," [Online]. Available: http://www.xenproject.org/about/history.html [Accessed 28 January 2015].

[18] S. Weil, "QEMU Internals," [Online]. Available: http://qemu.weilnetz.de/qemu-tech.html#intro_005fother_005femulation. [Accessed 29 January 2015].

[19] Wikipedia, "QEMU," [Online]. Available: http://en.wikipedia.org/wiki/QEMU#Details [Accessed 29 January 2015].

[20] [Online]. Available https://www.usenix.org/legacy/event/usenix05/tech/freenix/full_papers/bellard/bellard.pdf.

[21] [Online]. Available: http://www.tux.org/~mayer/linux/byte/bdoc.pdf.

[22] [Online]. Available: https://wiki.aalto.fi/download/attachments/41747647/qemu.pdf.

[23] [Online]. Available: http://codeandlife.com/2012/07/03/benchmarking-raspberry-pi-gpio-speed/.

[24] "Semantic Web," [Online]. Available: http://www.w3.org/standards/semanticweb/.

[25] "Resource Description Framework (RDF)," [Online]. Available: http://www.w3.org/RDF/.

[26] "RDF 1.1 specification," [Online]. Available: http://www.w3.org/TR/rdf11-concepts/.

[27] [Online]. Available: http://www.w3.org/XML/.

[28] "SensorML," [Online]. Available: http://www.opengeospatial.org/standards/sensorml.

[29] "OWL Standards Page," [Online]. Available: http://www.w3.org/2001/sw/wiki/OWL.

[30] [Online]. Available: http://openjaus.com/.

[31] [Online]. Available: http://www.sae.org/search/?content-type=%28%22STD%22%29&qt=jaus&x=0&y=0.

[32] [Online]. Available: http://openjaus.com/support/understanding-jaus/.

[33] [Online]. Available: http://cordis.europa.eu/result/rcn/140453_en.html.

[34] [Online]. Available: http://talos-border.eu.

[35] [Online]. Available: UGV IOP V1 Communications Profile.

[36] [Online]. Available: UGV IOP V1 Payloads Profile.

[37] [Online]. Available: https://msdn.microsoft.com/en-us/library/bb648760.aspx.

[38] [Online]. Available: http://www.orocos.org/.

[39] [Online]. Available: http://carmen.sourceforge.net/.

[40] [Online]. Available: http://www.gostai.com/products/jazz/urbi/index.html.

[41] [Online]. Available: http://www.mrpt.org/.

[42] [Online]. Available: http://orca-robotics.sourceforge.net/.

[43] [Online]. Available: http://www.robots.ox.ac.uk/~mobile/MOOS/wiki/pmwiki.php.

[44] [Online]. Available: http://www.ros.org.

[45] [Online]. Available: http://wiki.ros.org/sensor_msgs.

[46] [Online]. Available: http://wiki.ros.org/msg.

[47] [Online]. Available: http://pointclouds.org/.

[48] [Online]. Available: www.ros.org/reps/rep-0118.html.

[49] [Online]. Available: http://openjaus.com/products/jros/.

[50] [Online]. Available: http://wiki.ros.org/axis_camera.

[51] [Online]. Available: http://wiki.ros.org/image_pipeline.

[52] [Online]. Available: http://wiki.ros.org/ros_web_video.

[53] [Online]. Available: http://wiki.ros.org/image_transport.

[54] [Online]. Available: http://www.liveu.tv/.

[55] [Online]. Available: http://www.linux-kvm.org/page/Main_Page.

[56] [Online]. Available: SAE AS5710A.

[57] [Online]. Available: www.ros.org/tf.

[58] [Online]. Available: http://wiki.ros.org/axis_camera.