Project N°: 610456

D4.4 User-space applications runtime support specifications

September 30, 2015

Abstract:

This deliverable defines the user-space runtime support specifications for the following user-space applications: CloudRAN, MQTT server and COMPSs support for web services. It describes the current status and future plans of the work in T4.3.3, T4.3.4, and T4.3.5.

Document Manager – Paul Carpenter (BSC)	
Author	**Affiliation**
Stefano Adami	Eurotech
Jorge Ejarque	BSC
Yue Zheng Wen	TUD
Emil Matus	TUD
John Thomson	OnApp
Reviewers – For draft version	
Julian Chesterfield	OnApp
John Thomson	OnApp
Fabien Chaix	FORTH
Manolis Marazakis	FORTH
Reviewers – For final version	
John Thomson	OnApp
Fabien Chaix	FORTH

Document Id N°:		**Version:**	2.0	**Date:**	30/9/2015

Filename:	EUROSERVER_D4.4_v2.0.docx

Confidentiality

The EUROSERVER Consortium consists of the following partners:

Participant no.	Participant organisation names	Short name	Country
1	Commissariat à l'énergie atomique et aux énergies alternatives	CEA	France
2	STMicroelectronics Grenoble 2 SAS	STGNB 2 SAS	France
3	STMicroelectronics Crolles 2 SAS	STM CROLLES	France
4	STMicroelectronics S.A	STMICROELECTRONICS	France
5	ARM Limited	ARM	United Kingdom
6	Eurotech SPA	EUROTECH	Italy
7	Technische Universitaet Dresden	TUD	Germany
8	Barcelona Supercomputing Center	BSC	Spain
9	Foundation for Research and Technology Hellas	FORTH	Greece
10	Chalmers Tekniska Hoegskola AB	CHALMERS	Sweden
11	ONAPP Limited	ONAPP LIMITED	Gibraltar
12	NEAT Srl	NEAT	Italy

Revision history

Version	Author	Notes
0.1	Paul Carpenter	Version for internal review
0.2	Paul Carpenter	Review comments from Stefano Adami and Fabien Chaix. Updates from Jorge Ejarque.
0.3	Paul Carpenter and Jorge Ejarque	Responded to internal review comments
1.0	Paul Carpenter	Draft sent to EC, 19 June 2015
1.1	Paul Carpenter	Revised for final internal review, 20 September 2015
2.0	Paul Carpenter	Sent to EC, 30 September 2015

Contents

List of Abbreviations

Term	Definition
ACC	Accelerator
API	Application Programming Interface
Cloud RAN / CRAN	Cloud—Radio Access Network
COMPSs	COMP Superscalar: BSC framework and programming model for distributed computing
DAG	Directed Acyclic Graph
DB	Database
DFE	Data-flow Runtime Engine
DIP	COMPSs Data Info Provider
DNA	Deoxyribonucleic acid
EC	Eurotech's Everyware Cloud
EDC	Everyware Device Cloud
IoT	Internet of Things
JM	COMPSs Job Manager
KVM	Kernel Virtual Machine
LAMP	Linux—Apache—MySQL—PHP/Python
LNS	Largest Number of Successors Scheduler
LPT	Longest Processing Time Scheduler
M2M	Machine-to-machine
MQTT	Message Queue Telemetry Transport
MVC	Model—View—Controller pattern
OASIS	Organization for the Advancement of Structured Information Standards
OS	Operating System
PE	Processing Element
POSIX	Portable Operating System Interface
QoS	Quality of Service
RAN	Radio Access Network
REST	Representational State Transfer
RM	COMPSs Resource Manager
SoC	System on Chip
SP	System Processor
SPT	Shortest Processing Time scheduler
STTF	Setup Transfer Time First
TA	COMPSs Task Analyzer
TCO	Total Cost of Ownership
TS	COMPSs Task Scheduler
VB	Virtual Broker (MQTT)
VM	Virtual Machine
WCET	Worst Case Execution Time

List of Figures

Executive Summary

The EUROSERVER project is paving the way for a novel hyperconverged, next-generation, microserver architecture that will advance the state of the art for both software and hardware in the data center. The efforts are focused around energy efficient platforms that produce more useful computation for less energy. The efforts of Work Package 4 are concentrated on improving the software stack for microserver platforms. In particular, the software looks to create a holistic software stack using low resource, densely integrated nodes and benefitting from the unique aspects of the EUROSERVER architecture such as share-everything that set it apart from existing platforms.

This deliverable describes the user-space applications that take advantage of the EUROSERVER architecture in WP4. Cloud-RAN (CRAN) is a system for virtualization of radio access protocols, network functions, edge services and CRAN management on a common hardware platform, which represents the telecommunications use case. MQTT is a lightweight publish/subscribe messaging protocol designed for machine-to-machine (M2M) communication, specifically for constrained devices and low-bandwidth, high-latency or unreliable networks, which represents the (deprecated) transportation use case. COMPSs is a framework and programming model for coarse-grain task parallelism, designed for distributed platforms such as clusters, grids and clouds.

The work on Cloud-RAN, performed by TUD in T4.3.3, addresses the design and implementation of a data-flow runtime engine (DFE) that enables parallel execution of Cloud–RAN-specific workloads on the EUROSERVER multi-core platform. It focuses on the definition and specification of the system and software environment, development and optimization of DFE library elements for specific needs of telecom applications, and installation and demonstration of the DFE on testing platforms, a 20-core Intel Xeon x86 and an eight-core SoC with ARM Cortex-A15 and Cortex-A7 in a big.LITTLE configuration.

MQTT is an enabling technology that is specifically related to the Eurotech transportation use-case. This work was performed by Eurotech in T4.3.4, and concerned the definition and design for integration of the MQTT on microservers. Now that Eurotech has left the consortium, this work on MQTT has been deprecated, and removed in favour of a data-centre-specific use-case.

The work on COMPSs was performed by BSC in T4.3.5, and mainly focused on integrating COMPSs with web application frameworks and providing an energy-aware scheduler for COMPSs. Regarding the integration with Web application framework, BSC has proposed the usage of COMPSs to accelerate the web applications requests. On the other hand, BSC has proposed a model to estimate the energy consumed by the execution of an application. This model is used by the runtime scheduler in order to take into account the expected energy consumed when scheduling the application tasks.

These three applications will help validate the proposed EUROSERVER architecture and software stack. The work being done in WP4 is being coordinated into a single software stack to enable integration into the complete EUROSERVER prototype in WP6.

Regarding deviations from planned work:

- MQTT has been deprecated in favour of a data centre use case.
- As described in Section 4.1, the work in T4.3.5 targeted web applications in general rather than LAMP stack, as reflected in the revised Description of Work.

1. Introduction

EUROSERVER is a holistic project aimed at providing Europe with leading technology for capitalising on the new opportunities required by the new markets of cloud computing. Benefitting from Europe's leading position in low-power, energy efficient, embedded hardware designs and implementations, EUROSERVER provides the next generation of software and hardware required for next-generation data centers. The efforts of Work Package 4 are focused on improving the software stack for microserver platforms. In particular, the software looks to create a holistic software stack using low resource nodes and benefitting from the unique aspects of the EUROSERVER architecture. Instead of utilising a small number of 'fat' cores the EUROSERVER software platform will support multiple 'thin' cores that scale our and share resources. To enable support of such an architecture we propose fundamentally different concepts, tools and techniques. These go some way towards addressing some of the scalability issues that arise as the number of nodes in the platform increases. Virtualisation has led to massive consolidation of software workloads on servers. As this consolidation continues, the overheads of existing solutions need to be mitigated through optimised resource sharing.

Another part of the share-all vision of EUROSERVER is that all the resources should be shared between all microservers that might need those resources, subject to fair access control mechanisms and accounting of resource usage.

As shown in Figure 1, components and techniques such as UNIMEM, the MicroVisor, shared and optimised I/O, optimised virtualisation techniques and orchestration and energy aware orchestration tools address some of the scalability issues that arise as the number of nodes in the platform increases. The work being done in WP4 is being coordinated into a single software stack to enable integration into the complete EUROSERVER prototype in WP6.

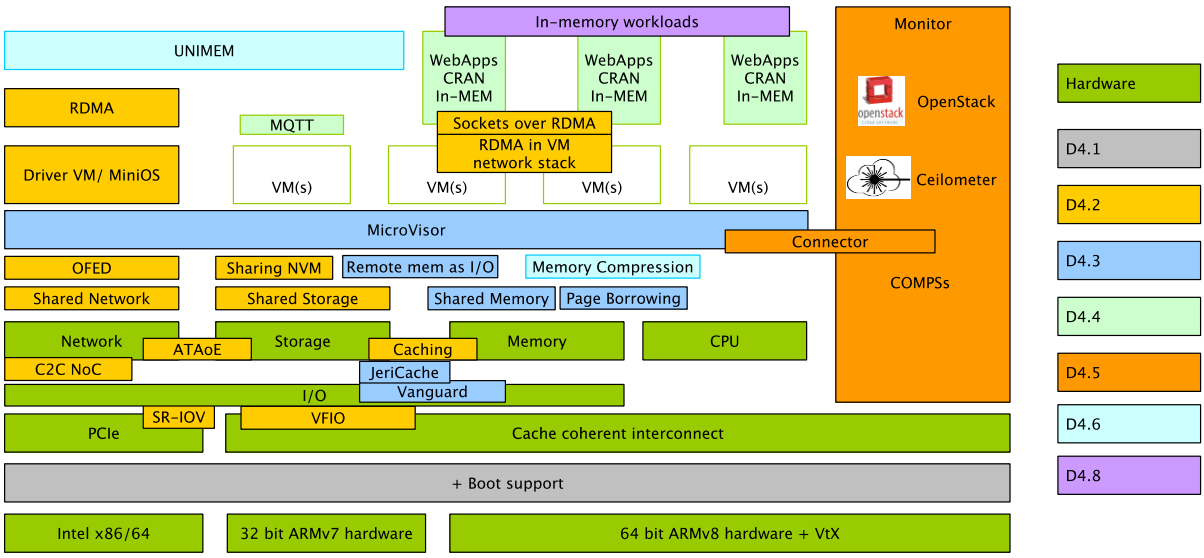

Figure 1: Diagram showing the areas being worked on in EUROSERVER in WP4

This deliverable describes the work on the following user-space applications:

(T4.3.3) Cloud-RAN, representing the telecom use case, described in Section 2.

(T4.3.4) MQTT server, representing the transportation use case, detailed in Section 3.

(T4.3.5) COMPSs runtime for web applications, described in Section 4.

The work on Cloud-RAN and COMPSs is ongoing. On account of Eurotech leaving the consortium, the work on MQTT has been deprecated. MQTT is specifically related to the transportation use-case which was defined by Eurotech, and therefore it also is planned to be removed in favour of a data-centre-specific use-case.

2. Runtime resource management for Telecom servers (TUD)

2.1. Introduction

This section describes the work that has been done by TUD in the context of Task 4.3 (Subtask 4.3.3) during the M7–M21 period. In this part of work, TUD addresses the design and implementation of the data-flow runtime engine (DFE), enabling parallel execution of Cloud-RAN (CRAN)-specific workloads on the EUROSERVER multi-core platform. CRAN is a system for virtualization of radio access protocols, network functions, edge services on a common hardware platform, and it represents the telecommunications use case. The development activities were coordinated according to the concept and requirements analysis within T2.2 and the results of this work have been complemented with activities in T3.1 (CRAN benchmark), as well as in T4.2 (Subtask 4.2.3 – Programming interface). The work and the contributions of TUD within reported project period (task 4.3.3) are summarized in the following items:

- Definition of system model and the specification of DFE (based on T2.2, T3.1 in conjunction with T4.2).
- Specification of POSIX-based software environment (tools, APIs and libraries) for DFE implementation enabling cross-platform development and simulation, as well as seamless portability from development platform to discrete/full EUROSERVER prototypes.
- Development of the concept of modular hierarchical DFE allowing exploration and optimization of different DFE arrangements for specific hardware configurations and application requirements (in conjunction with T4.2.3).
- Development and optimization of DFE library elements for specific needs of telecom applications (including thread-safety, low overhead and high responsive task/system description data structures, containers, queues, controller and manager components)
- Selection and installation of DFE development, simulation and demonstration platforms (20-core Intel Xeon x86 and 8-core big.LITTLE ARM Cortex-A15 and Cortex-A7).
- Development of pipelined multi-threaded DFE and verification and analysis on simulation platforms)
- Test benchmarks definition and implementation for purpose of DFE analysis.
- Concept and implementation of initial version of hierarchical list-based scheduling algorithms.

2.1. System Concept

As illustrated in Figure 2, in general, the CRAN computing system allows the virtualization of radio access protocols, network functions, edge services and CRAN management on a common hardware platform. While the virtualization of RAN management, edge services, some network functions and higher layers of access protocols is the state-of-the-art approach in RAN/CRAN design, virtualization of lower protocol layers suffers from the low computation capabilities of underlying general purpose hardware. As the EUROSERVER platform provides scalable computation capacity, the motivation of this part of work is to develop runtime environment and management techniques enabling efficient and simultaneous execution of multiple protocol instances on common hardware. Particularly, the focus is on protocol data plane processing that represents the most challenging part of radio access due to the data-intensive computations and strict real-time requirements.

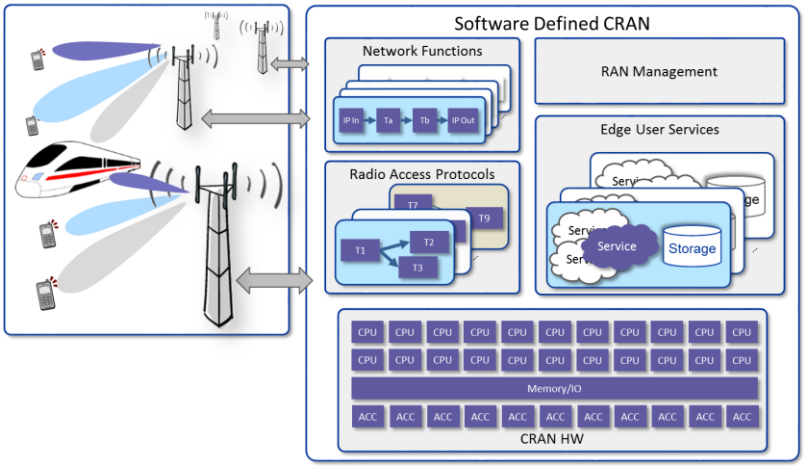

Figure 2: Concept of Cloud-RAN computing architecture

2.2. Concept of dataflow engine

In order to well represent and implement the data plane part of radio access protocol applications, a dataflow computation model has been selected. More specifically, a *hierarchical* dataflow model has been adopted (proposed in T4.2.3 and reported in D4.2), which better reflects the multi-protocol requirements. In this case, the protocol application comprises coarse-grain "system" elements typically associated with control functions and fine-grain "task" elements representing atomic computation kernels. The system is a composition of task and system elements. Precedence constraints can be defined for system and task elements reflecting data flow dependencies. Moreover, a set of parameters can be associated with task and system elements; e.g. I/O arguments, Worst Case Execution Time (WCET), task type, and logical core affinity. Based on this, the general concept of the CRAN system has been defined according to Figure 3.

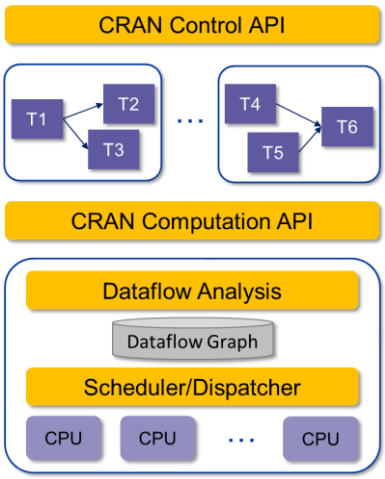

Figure 3: General concept of Cloud-RAN computing architecture

The CRAN Control API control multiple virtualised systems associated with CRAN data plane applications while the Computation API allows instantiation, configuration and execution of dataflow applications (both developed within T4.2.3 and reported in D4.2). The requests from Computation API are translated to system calls of the underlying DFE. The DFE schedules and dispatches tasks/systems to available computing resources, according to a specific scheduling policy and precedence constraints. In order to differentiate system/task managing strategies, the hierarchical DFE architecture has been proposed (Figure 4).

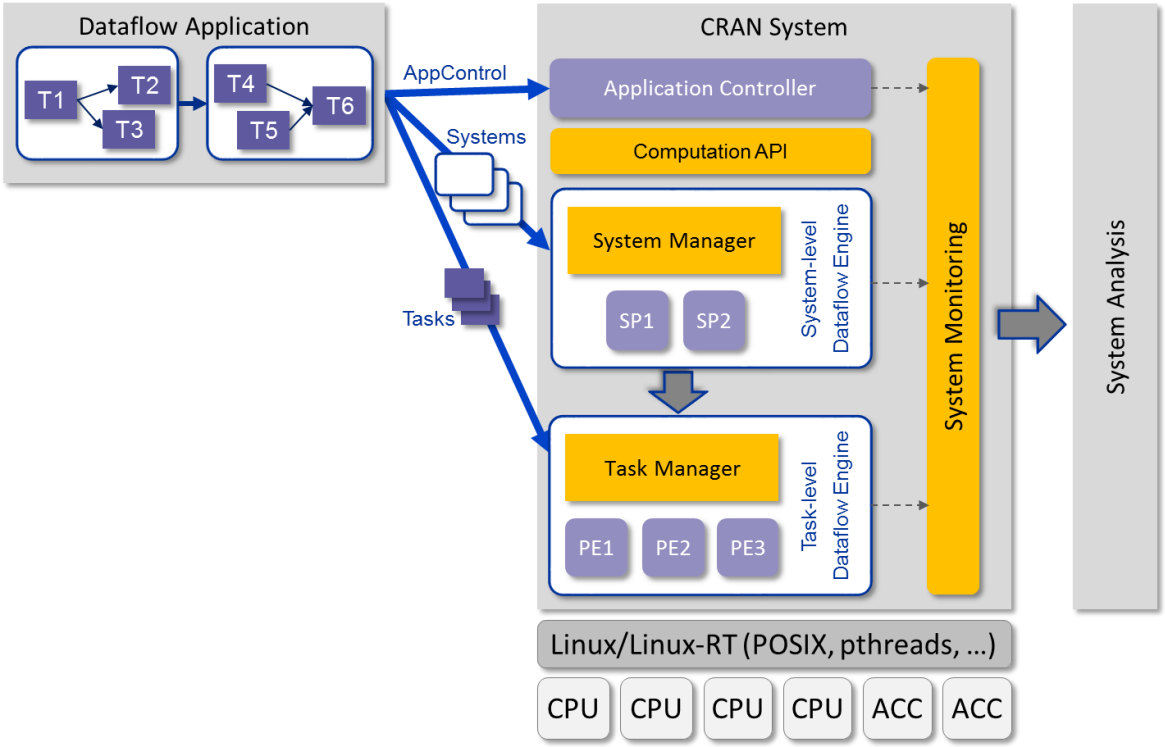

Figure 4: Principle of hierarchical runtime dataflow engine

The engine comprises dedicated system and task managers that schedule the systems to system processors (SP) and tasks to processing elements (PE), respectively. SPs and PEs represent logical (virtual) computing resources that are mapped to physical resources using DFE and operating system configuration parameters. The number and type of SPs and PEs is variable and can be configured either statically (at design time) or dynamically (at runtime). Figure 5 shows an example of mapping DFE elements to physical resources in a two-node hardware configuration comprising both CPU threads and accelerators (ACC). Note that the assignment of logical to physical resources is highly dependent on the hardware configuration, application properties and optimization criteria, and will therefore be a subject of a specific optimization subtask in a later phase of the project.

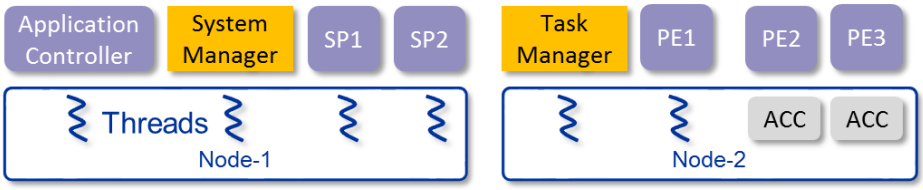

Figure 5: Example mapping DFE elements to physical resources (two nodes with CPU and 2 accelerators)

DFE employs the pipelined queuing architecture illustrated in Figure 6. Engine functional units (controller, system manager, task manager, SPs, PEs) are associated with OS threads. Units are connected via queues enabling message passing. This distributed pipelined architecture enables fully asynchronous operation of all units and it enables OS level system optimization by managing access of particular threads to shared physical computing resources (prioritization, core affinity). Moreover, the pipelined nature of the architecture will allow to scale-out performance in the future by splitting DFEs among multiple nodes of the EUROSERVER platform at any queue boundary. On the other hand, asynchronous operation of distributed units poses challenges for system debugging and monitoring.

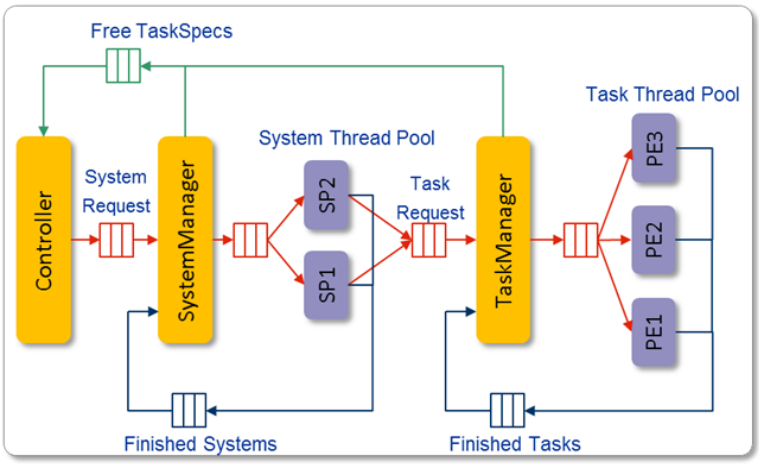

Figure 6: DFE architecture

In DFE, the units of computation – systems and tasks – are specified using a dedicated data structure called TaskSpecification. The TaskSpecification data structure contains all relevant parameters and state associated with specific task or system. Messages comprising TaskSpecification structures are passed between DFE units in order to process the task or system in a subsequent stage of the engine. The life cycle of task/system within DFE is illustrated in the state diagram in Figure 7.

Figure 7: State diagram illustrating the life cycle of a task/system within DFE

2.3. DFE Implementation

For the purpose of portability, the DFE has been developed on top of a subset of the POSIX API (providing pthreads, real-time features, and so on) within a Linux environment. The following tool chain has been used for this implementation:

- Compiler: GNU C/C++, libc, glibc
- Libraries: FFTW
- Analysis tool: Octave/Matlab
- Graphics: Gnuplot

Using a standard Linux environment simplified the development effort since the DFE could be completely implemented on a host Xeon x86 platform (Figure 8), and later ported to an ARM Cortex-A15/Cortex-A7 system (Fig. 2.7b), before the EUROSERVER platform comes available. This allowed early software validation, demonstration and analysis. Compatibility with the POSIX specification guarantees portability both to the EUROSERVER discrete prototypes and to the full prototype (Fig. 2.7c), as well as to real-time OS powered systems.

After initial development of DFE on an x86 system, in M16, TUD purchased an Odroid XU3 big.LITTLE ARM system comprising four Cortex-A15 and four Cortex-A7 cores, and successfully ported DFE to this platform. The initial tests demonstrated portability, functionality and stability of DFE as well as the data-flow API (from T4.2) and the dataflow benchmark (from T3.1). It is expected that it will be possible to seamlessly port the developed software stack to the EUROSERVER platform in a later phase of project (in cooperation with FORTH, CEA/ST and OnApp).

Figure 8: Development, simulation and demonstration platforms for Cloud-RAN system implementation

2.4. Ongoing and future work

In the current phase of the project, the DFE is extensively analyzed. The bottlenecks have been identified within DFE (e.g. thread pool queues and synchronization, …) as well as the CRAN benchmark application (e.g. fine-grain partitioning at some places, …). The bottlenecks have been analyzed and classified, and the priority for optimization has been defined. In the next phase of the project, we will optimize DFE according to priority. Optimization will be done at multiple levels of the software stack:

- Dataflow application:
  - Improve application partitioning; e.g. reduce granularity in critical sections.
  - Exploit a given ARM instruction set; e.g. leverage ARM Cortex-A53 Advanced SIMD / NEON instructions in critical tasks (in cooperation with ARM).
  - Implement additional benchmarks for analyzing specific parts of DFE.
- DFE:
  - Modify the DFE architecture (queues, thread synchronization mechanism, reduce contention at shared objects, …).
  - Optimize structure of DFE.
  - Optimize size and alignment of critical data structures.
  - Reduce dynamic allocation of data structures by using pre-allocated containers.
  - Consider lock-free techniques in critical sections using EUROSERVER-specific Cortex-A53 instruction set (in cooperation with ARM).
- OS:
  - Tune the Linux system in order to improve responsiveness (e.g. minimize services, optimize page size, optimize I/O interrupts, …)
  - Install real time kernel and analysis of DFE determinism.
  - Explore the thread core affinity and prioritization strategies for system performance optimization.

After the DFE optimization phase, additional tasks according to the project plan will be addressed within the next phase of the project:

- Development of hierarchical scheduling for efficient use of resources,
- Explicit data management in order to improve responsiveness and locality.
- Acceleration of critical tasks.
- Validation of DFE on EUROSERVER discrete and full prototype (in cooperation with FORTH, CEA, ST).
- Scale-out solution of DFE architecture for multi-node (chiplet) operation in EUROSERVER platform using the UNIMEM approach (in cooperation with FORTH),
- Instantiation and operation of DFE within virtual machine (in cooperation with ONAPP, BSC)

3. MQTT server for M2M cloud computing applications (ETH+ONAPP)

Note: Now that Eurotech has left the consortium, the work on MQTT has been deprecated. MQTT is an enabling technology that is specifically related to the Eurotech transportation use-case, which has been removed from the revised Description of Work in favour of a data-centre-specific use-case.

3.1. Introduction

Microservers, including those proposed by EUROSERVER, offer a low power alternative to traditional servers and can be used in embedded system applications. As described in D2.3 “Transportation server requirements report”, rolling stock (trains, metros, trams), buses, public and commercial vehicles can benefit from the adoption of the EUROSERVER architecture due, in part, to:

- Multiple high-level services on constrained size and power;
- High Integration and modularity for demanding environments
- Low Total Cost of Ownership (TCO) for both its initial adoption and for the whole solution lifecycle

In particular it was highlighted in D2.3 that there will be the need for a solution to handle a combination of multiple I/O sources; Both low-level sources and high-level sources that integrate with a backend system for further data collection and processing. The data collection system would be a remote M2M cloud platform running on a EUROSERVER data centre server or on a private on-premises EUROSERVER server.

To handle the I/O from multiple sources it is important to demonstrate the capability of the EUROSERVER platform to handle multiple input sources in a timely manner and be able to react to some of those inputs, whilst still conforming to power standards and requirements as set out by the transportation industry, including but not limited to;

1. EN50155: Railway applications – Electonic equipment used on rolling stock
2. EN61373: Railway applications. Rolling stock equipment. Shock and vibration tests
3. IPxx: Ingress Protect Level

On the backend, a EUROSERVER system executing the M2M platform should comply with the data centre requirements collected in D2.1.

- Scale-out for multiple customers, multiple geographically co-located set of sensors

- I/O (storage) virtualization, for high speed, low latency access to shared No-SQL database
- Resource isolation, for security and reliability solution and legal requirements

3.2. MQTT: MQ Telemetry Transport

MQ Telemetry Transport (MQTT) is an open, extremely simple and lightweight publish/subscribe messaging protocol designed for constrained devices and low-bandwidth, high-latency or unreliable networks [10]. It is an OASIS standard [11], designed for machine-to-machine (M2M) communication.

The protocol was designed for the types of resource-constrained system being proposed by EUROSERVER. Furthermore, the transportation platform can also benefit from the EUROSERVER scalability design in order to scale up to the expected demand.

The original architecture of the Eurotech MQTT-based messaging server is shown in Figure 9. It is integrated in the Eurotech commercial M2M application platform called Everyware Cloud (EC)[1] which is currently designed to work on an x86 system/platform. For the current breed of transportation systems the number of I/O sensor devices is typically on the order of hundreds of devices, but given that some embedded applications for M2M and Internet of Things (IoT) have between a thousand and ten thousand devices and are expected to exceed a million for large deployments in the near future, the scalability of the system must significantly improve.

Each sensor/device in the diagram will typically send a telemetry message (informational only) and will receive command/management messages (information + actuation);

- A telemetry style of message contains possibly multiple metrics (types of information). Some of these sensors will be event-driven and/or combine with continuous acquisition systems, where information rates may change depending on changes in the device's contextual environment.
  The period in which messages may be sent after a particular event is typically between one and thirty seconds but it can also commonly last up to five minutes. The sensor data rate may also vary among types of devices, depending on the type of information being detected from the operational field and the data processing and transmission policies. For each of these messages that are sent from a device, the common message size is *N* x 100 bytes (*N* x 64 bytes of which are the payload) where *N* is the number of metrics. Depending on the encoding system used and also the particular level of detail, or quality of service (QoS), there may be requirements to send extra redundant copies of the information, in order to increase the likelihood of an accurate transmission.

[1] http://www.eurotech.com/en/products/software+services/everyware+cloud+m2m+platform

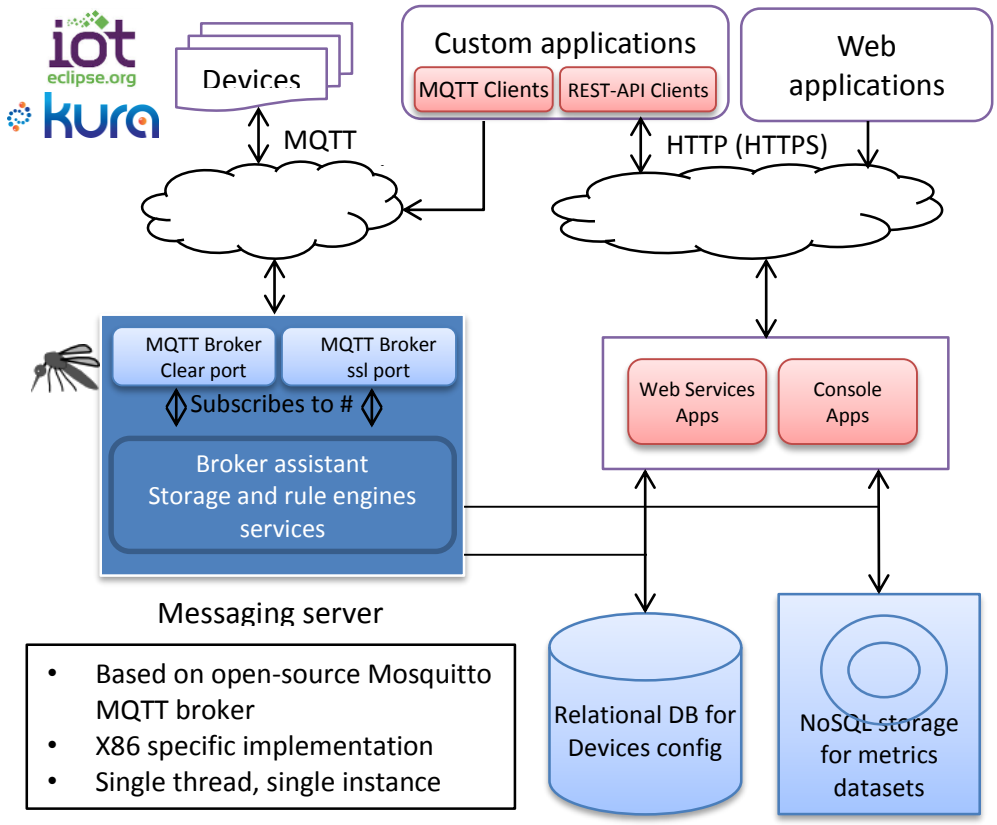

Figure 9: Original MQTT architecture

- A command/management message will be sent by the M2M platform to devices, in particular with actuators, and may have many different types of payload depending on whether the device is in a state of configuration, management, provisioning and/or updates. The payload size varies depending on the type of message but it can be much larger than standard telemetry messages, being on the order of several kilobytes or megabytes. In the case of ensuring QoS rather than having to repeat the message several times there will generally be a single re-transmission that is acknowledged by the receiving device.

3.3. Implementation for EUROSERVER architecture

A single messaging server will need to handle this wide range of types of messages from/to various types of devices, with different messages having different lengths and processing times associated with them. To be well engineered a messaging server will also need to be able to respond to these messages in a timely manner, prioritizing the most important actions and information. The information that is gathered from these sensors will then need to be stored in such a way that it can be accessed and managed with real-time constraints, either from the messaging server, or from remote concurrent customer applications. For post-processing, the requirements will likely be less constrained. The messaging server will also need to handle updates for the configuration of devices and will either maintain state transitional information locally or request it periodically to ensure that it is synchronized. To be able to handle the worst case situation, the server will have to be able to handle the peak workload that it has been designed for, and scale in accordance with the number of

requests it temporarily receives. Having an adaptable platform such as EUROSERVER allows the hardware to be scaled back when fewer messages are being received.

Figure 10 shows the envisioned target architecture, in which the original architecture of Figure 9 has been adapted to better suit the EUROSERVER hardware architecture. The envisaged scale-out design of EUROSERVER allows for core functionality to be maintained by always-on EUROSERVER device cores, which is important for certain reliability and legal purposes. This core unit provides services to other M2M platform components. It monitors the security and health of each server subsystem and implements a load balancer to manage the handling of device requests, in order to be able to scale on demand. Having a shared memory system such as UNIMEM with support in the storage I/O sub-system will mean that resources for large datasets can be shared between cores without the penalties normally associated with remote accesses. From the core unit, the other nodes can connect securely either with direct hardware access and/or other remote secured channels to allow for further extensions as the number of devices in the system increases and more nodes are needed to handle the load. Intelligent placement of resources, as also proposed in EUROSERVER, will allow for resources that are being regularly accessed to be cached in memory using systems such as memcache [9] that could be used for the configuration of local devices, to avoid a large number of concurrent accesses to a centralized database. Finally, segregation and isolation of resources as proposed by EUROSERVER provides improved security and reliability for multiple deployments[2] hosted by the same M2M Integrated platform.

[2] Typically multiple customers with multiple geographically co-located group of sensors

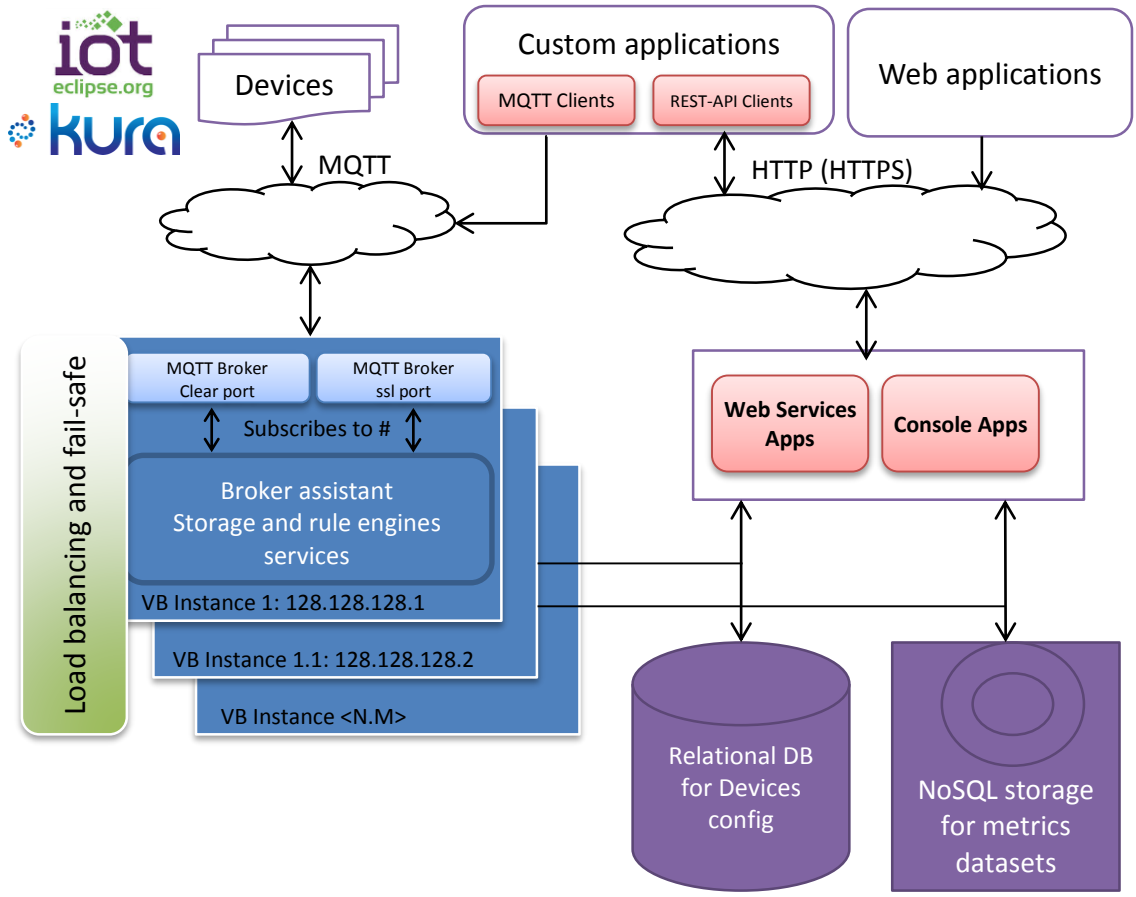

Figure 10: Target MQTT base messaging server architecture

For the implementation of the target architecture shown in Figure 10, work has been performed to design and implement a new Messaging server architecture that is natively cross-platform (supporting both x86 and ARM architectures) and to provide load balancing that exploits the EUROSERVER architecture. For the core functionalities of MQTT broker, results from initial analysis target our effort towards existing alternative broker implementation instead of extending the capabilities of the original one. The now mature ActiveMQ [1] MQTT broker has been tested and selected. It is an open-source Java-based application and has native support for clustering (network of brokers) as well as failover, which is important for this type of use-case. The server architecture has also been ported to work on top of an OSGI container that uses the open-source Karaf project [2], which implements an ActiveMQ bundle. Original messaging server components, implementing storage and rule engine services has also been adapted and ported on the OSGI container messaging server, and ported to an ARM 32-bit platform. This allows testing and investigating the suitability of the final EUROSERVER Silicon Prototype and allowing development to proceed before the full hardware becomes available.

As a result of this work the implementation of the MQTT messaging server was revised for EUROSERVER, as shown in Figure 11. This shows how the new MQTT architecture has been revised to take advantage of some of the features that are available in the EUROSERVER platform that were not envisaged when the original specifications of MQTT were drawn up. In particular, the handling of resources via virtual brokers (VBs) allows for the platform to scale in a way that was not previously

planned. Through intelligent placement of resources and continual monitoring of the data coming from the devices, it is expected that when running on the EUROSERVER platform, the MQTT server will be able to power down Virtual Brokers when they are not required and thus save energy relative to an over-provisioned system that must run continually without any power control options. As the number of devices increases, the proportion of peak to expected use will likely rise, making the potential savings even more important when considered in the long-term, which will greatly reduce the TCO of the platform.

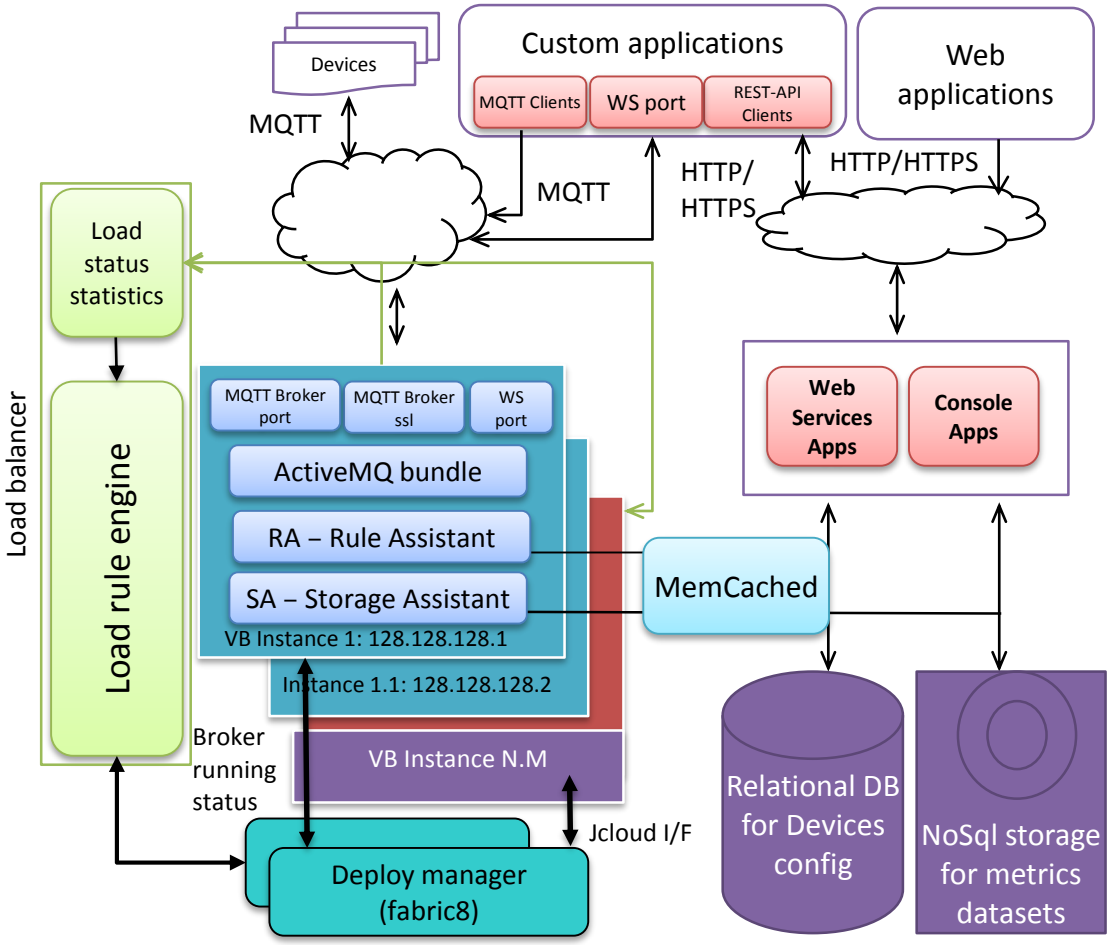

Figure 11: Implementation of the new MQTT messaging server architecture for EUROSERVER

To understand how effective the hardware platform would be in the real world, a hardware and software simulation using ARM 32-bit embedded boards was created and demonstrated at the M18 review meeting. The demonstration was set up as shown in Figure 12. Two Raspberry Pi2 Model B[3] boards, which are cheap, popular, low-power 32-bit ARM development boards, were used to show a broker–client system. One of the Raspberry Pi boards was set up to emulate the kind of telemetry information that would be expected from heater sensors, communicating using the Kura-based MQTT protocol to an M2M integration platform enabled with the newly developed Messaging Server. These sensors were programmed to simulate four metrics, internal temperature, text, alarm,

[3] https://www.raspberrypi.org/products/raspberry-pi-2-model-b/

and error, with a message size of 100 bytes (65% payload). To allow for stability in a demonstration environment, the messaging rate was turned down to two messages per second, whereas in real life each sensor may transmit as many as sixteen messages per second. The M2M integration platform is composed of the messaging server, also running on a Raspberry Pi2 Model B, and a laptop computer providing the storage subsystem, web services and console interface to interact with the platform. These three physical devices are interconnected on a local Ethernet 10/100 Mb/s network.

The sensors subscribe to one information topic on the Messaging Broker and publish their metrics samples. The server manages the subscriptions by actively listening to these published messages, parsing their content and storing the metrics values in datasets of No-SQL database Cassandra[4] on the storage subsystem on the laptop computer. Other information, including control and configuration of the messaging server and for the registered sensors are stored in a dedicated database, which is also on the Laptop, and are available to both the messaging server and the M2M platform web services.

Each Raspberry Pi2 Model B device contains a 900 MHz quad-core ARM Cortex-A7 SoC (ARMv7 ISA) with 1 GB of local RAM. The resulting demonstration platform is an order of magnitude less powerful than the EUROSERVER prototype, but it was sufficient to demonstrate the concepts, test and validate the porting and the developed modules, and perform initial calculations, as well as showing functional viability.

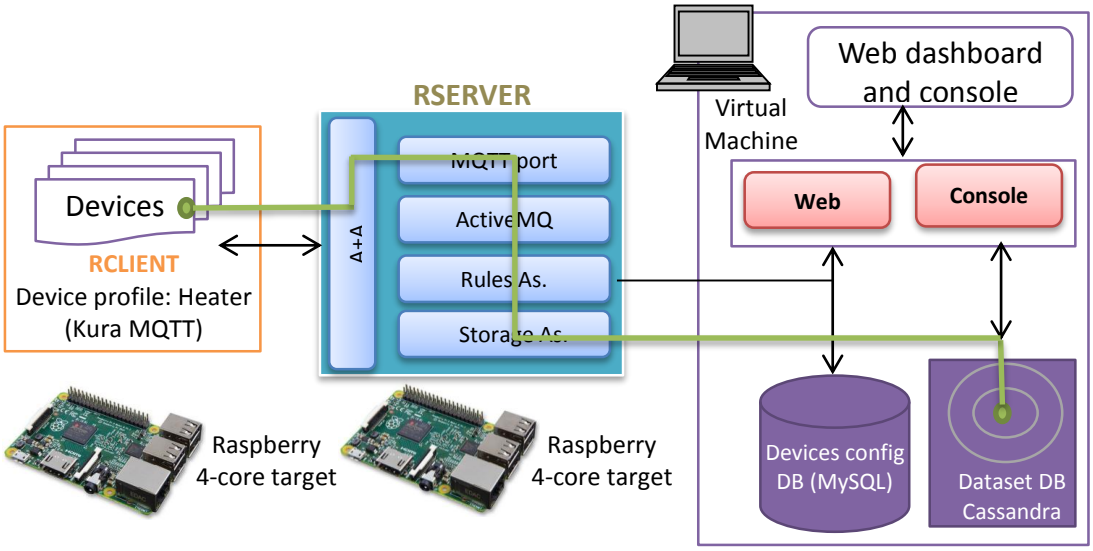

Figure 12: Real-world simulation of EUROSERVER implementation using ARM 32-bit hardware

The Eurotech Everyware Device Cloud (EDC) [4] is an "end-to-end solution that includes purpose-built hardware, connectivity and embedded device management[...]". The **Everyware Device Cloud Client** and Machine-to-Machine (M2M) cloud-based services deliver actionable data from the field to downstream applications and business processes, dashboards, and reports. As a commercial offering, it is important that device connection and set up is clearly visible to the customer's administrator, and this is done through the EDC web-dashboard. In the demonstration setup, the dashboard also

[4] http://cassandra.apache.org/

executes on the laptop computer, together with a client browser to access the user interface. Figure 13 is a screenshot of the dashboard for the demonstration environment, populated with the simulated sensors (the *Devices* tab). From here, each device has an entry that can be configured using an extensible framework. The dashboard shows historical information, device and broker configuration, and various other features that would be useful for an administrator using the EUROSERVER hardware interacting with a variety of devices. Simple and complex rules logic (also real-time) can be defined here, leveraging the Rule Assistant capabilities on the Messaging Server. Post-processing systems could be connected via APIs to the dashboard to remotely process the data, asynchronously from the EUROSERVER hardware platform and be used for business intelligence and other more complicated systems and processes that cannot be directly run on the hardware.

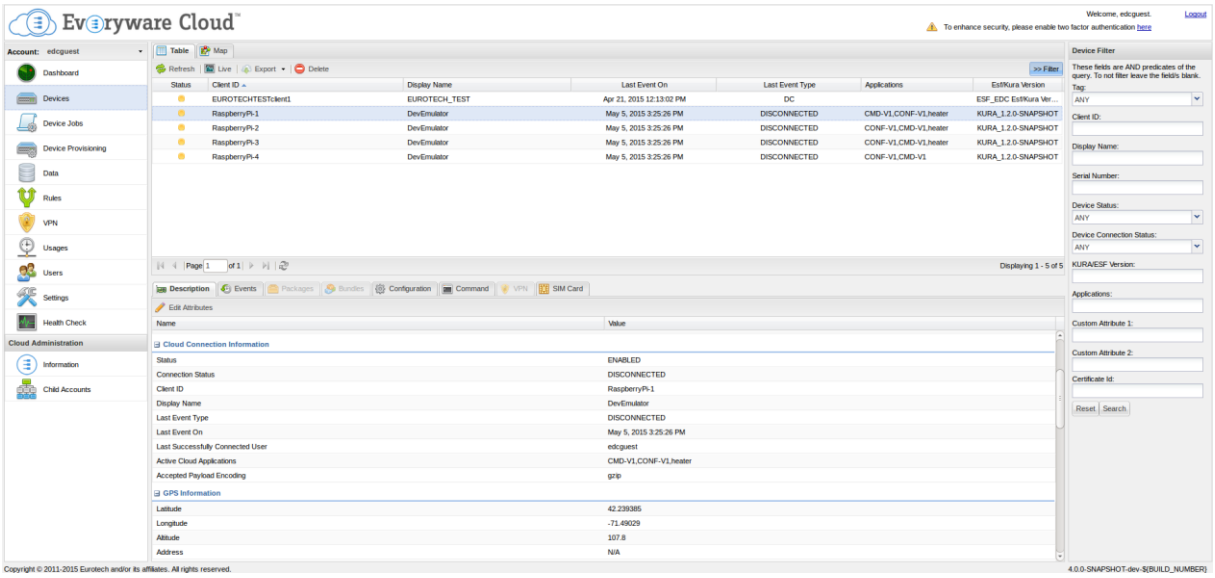

Figure 13: Eurotech EDC dashboard, populated with four heater sensors of the demonstration setup

4. Runtime resource management for web services (BSC)

4.1. Deviation from LAMP stack to web applications

The EUROSERVER Description of Work specifically mentions use of the LAMP stack in Task 4.3.5. LAMP is an acronym referring to a commonly-used software stack for programming web sites, in which the Operating system is Linux, the HTTP server is Apache, the database is MySQL and the scripting language, which processes HTML document requests, is PHP or Python. Although the LAMP stack is still widely used, other software stacks have recently appeared for the development of next-generation web applications, such as Spring[5], Django[6], Ruby on Rails[7], etc. Most of these stacks are based on either the Model—View—Controller (MVC) or the *n*-tier pattern, where the highest layer manages presentation to the user using scripting languages such as PHP or JavaScript to generate HTML documents, and the lowest layer stores data using whichever engine is most appropriate for

[5] https://spring.io/
[6] https://www.djangoproject.com/
[7] http://rubyonrails.org/

the application data (SQL, NoSQL, flat files, etc.). The middle layers implement the application logic to carry out the user requests using the stored data.

In order to avoid losing focus on the other layers of the stack and thereby limit the impact of the results, it was decided to not work specifically on the LAMP stack. Consequently, research was conducted in the context of web applications in general, and, to be more specific, effort focused on research on the middle layers where the COMP Superscalar programming model can produce more benefit. Regarding the other layers, the presentation layer has low computation requirements on the server side, so efforts would grant little benefit within COMPSs. The storage layer is tightly coupled to the used framework, such as MySQL[8] for SQL databases, Cassandra[9] or MongoDB[10] for No-SQL databases and Key-value stores or Hadoop[11] File System for distributed file systems, and the research on this topic is already being adequately addressed by respective communities.

4.2. Overview of COMPSs

COMP Superscalar (COMPSs) [3], developed at BSC, is a framework and programming model that enables straightforward development of applications to be executed in distributed platforms such as clusters, grids and clouds. The main idea is that the COMPSs framework schedules coarse-grain work in a similar way to how a superscalar processor executes out-of-order instructions. COMPSs uses a task-based programming model, in which developers mark as tasks those parts of the code that could potentially be executed remotely in the distributed platform. Tasks are defined using an annotated interface, which indicates which methods implement tasks, together with each task's input and output data. The application code is implemented in a sequential fashion, where task methods are invoked in the same way as standard method calls. When the application executes, the runtime system detects when a task method is invoked, and it performs a data dependency analysis to build a directed acyclic graph (DAG), in which the vertices are tasks and the arcs are dependencies between tasks. Task invocations that are free of dependencies can be executed concurrently on the available resources. Whenever a task has finished executing, the runtime system removes any dependencies on it, releasing any tasks that were waiting for the data generated by that task. The task lifecycle is depicted in Figure 14.

[8] https://www.mysql.com/
[9] http://cassandra.apache.org/
[10] https://www.mongodb.org/
[11] https://hadoop.apache.org/

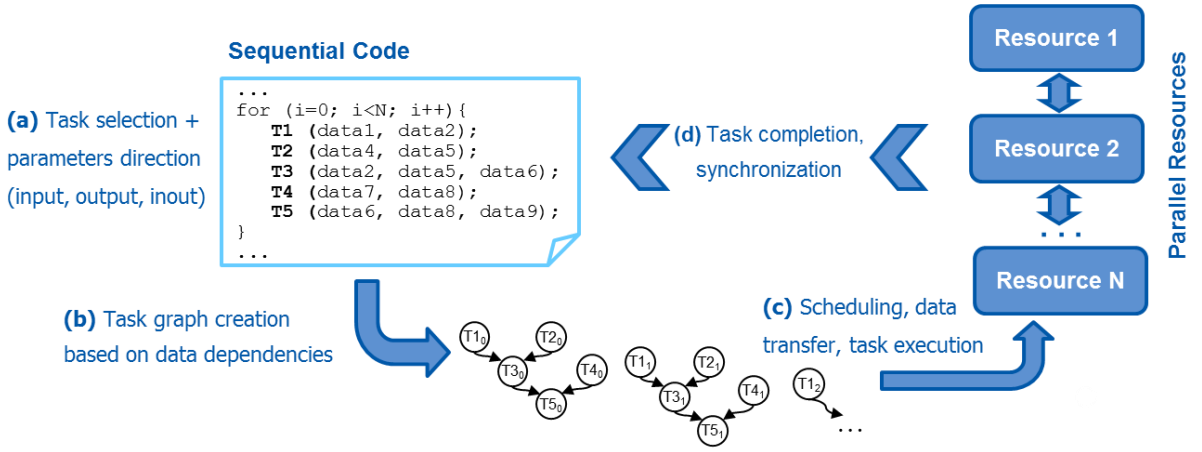

Figure 14: COMPSs application execution lifecycle

Figure 15 shows a detailed view of the COMPSs runtime system, including the various runtime components. The Task Analyzer (TA) analyzes the application code to detect data dependencies between tasks and the Data Info Provider (DIP) provides a registry of the data used by tasks, storing the location of multiple versions in order to eliminate false dependencies, in analogy to register renaming in a superscalar processor, as well as replicas, which are generated in order to improve data locality. After the analysis, the Task Scheduler (TS) plans the execution of the DAG using the information about the available resources, managed by the Resource Manager (RM), and the data location provided by the DIP. Once a task has been scheduled, it can be executed on the resources using the Job Manager (JM) component.

COMPSs can be configured to run on clusters, grids and clouds. In the first two options, the number of resources is static during the application execution, whereas in the case of a cloud infrastructure, the runtime can employ an elastic number of resources. In the latter situation, the Task Scheduler detects when there is a need for additional resources or excess of resources and, if necessary, it informs the Resource Manager to contact the Cloud Middleware to create or destroy resources.

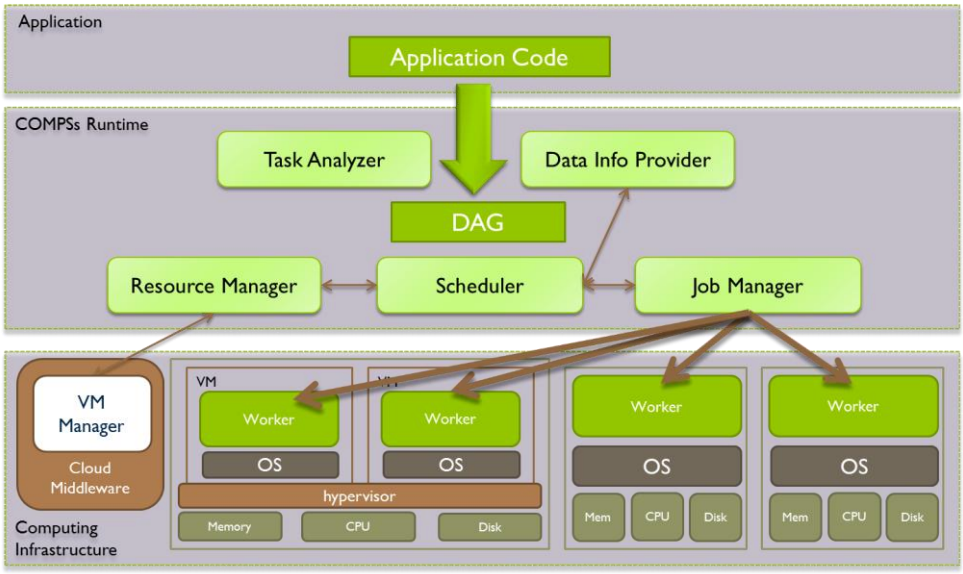

Figure 15: COMPSs runtime components

4.3. COMPSs for web applications

As introduced in Section 4.1, most web application software stacks are based on the Model—View—Controller architectural pattern. Using this pattern, depicted in Figure 16, web applications are composed of three parts: the model, which is in charge of managing the application data, the controller, which is charge of implementing the logic to process the data according to the user's request, and the view, which is in charge of visualization of the request forms and their results.

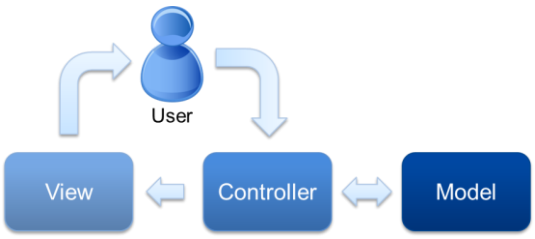

Figure 16: Model-View-Controller pattern

Figure 17 shows how an MVC application is usually deployed. The model (application data) is normally deployed in a database/storage server, and the controller and part of the view are normally deployed in an application server. The part of the view that is embedded in HTML pages, using PHP, JavaScript or similar, is executed in the user's web browser.

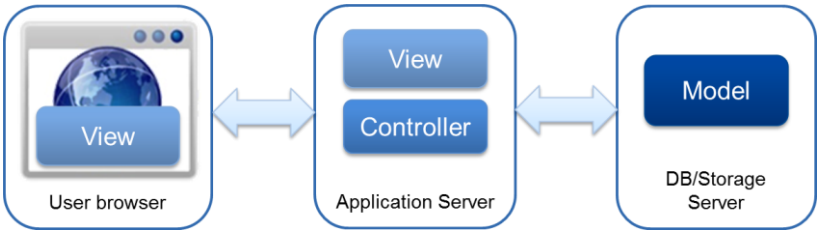

Figure 17: Deployment of the MVC components

The application server dedicates a pool of threads for the execution of the user requests. When the number of requests is greater than the number of threads, some of requests must be queued until threads have been released, which increases the response time. The response time can be reduced by increasing the number of threads in the pool, but the maximum number of simultaneously executing threads is limited by the server hardware. In situations where the request demand is larger than is supported by the hardware, web applications can be scaled by deploying multiple application servers, as depicted in Figure 18. In this case, multiple application servers are coordinated using a Load Balancer, which distributes the users' requests across multiple servers, and storage servers are coordinated in various ways depending on the data and the underlying technology (e.g. a master/slave mode is common in SQL databases and a peer-to-peer mode is common in NoSQL databases and key–value stores).

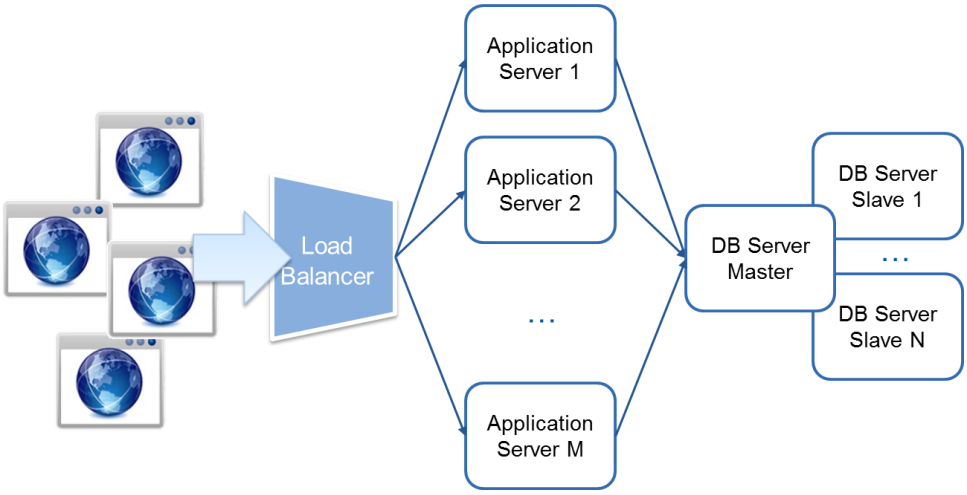

Figure 18: Scalable web-application deployment

With this scalability option, the response time could be reduced as much as required until it reaches a lower limit fixed by the execution time of a single request. When the computation performed by each request is small and fast, this response time could be low enough to provide a good Quality of Service, but in the case of requests with larger computations, finer grain scalability could be needed, and this is the case where COMPSs can produce a benefit.

As introduced in Section 4.2, the COMPSs runtime is able to detect the parallelism in the application code by analyzing data dependencies between tasks and execute them according to the detected task parallelism and the available resources. So, implementing the web application requests with COMPSs superscalar, the runtime could detect the parallelism inside the request and decide how many threads are needed to have an efficient request execution.

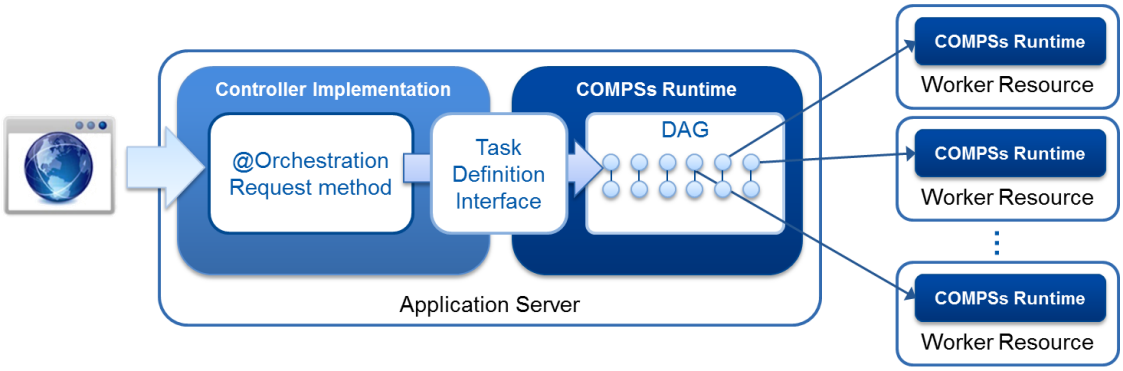

Figure 19: Controller implementation with COMPSs

Figure 19 shows how the web application controller is implemented using COMPSs. Each application request is represented by a method that implements the required functionality using the COMPSs programming model. In order to employ the COMPSs programming model, (1) the *@Orchestration* decorator must be added to request methods that should be instrumented by the COMPSs runtime and (2) these tasks must be defined in the Task Definition Interface. Methods defined as tasks should be invoked several times during the request workflow and they should have a sufficient computation time to compensate for the overhead of remote invocation. The controller code, Task Definition Interface and COMPSs runtime libraries are deployed together in the application server. At runtime, each time that a user invokes a request method, the COMPSs runtime creates a DAG for the request, and its tasks are scheduled and executed on the available worker resources. Regarding scalability, resources can be replicated as desired to execute an appropriate number of tasks in parallel. Since resources are assigned to request tasks, instead of the whole request, this allows response time and scalability to be improved.

Assigning a set of worker resources per request is most efficient if the generated DAG is embarrassingly parallel, since the task scheduler can balance the load to get good performance and resource usage. In contrast, if the levels of the DAG have varying degrees of parallelism, there is a trade-off between the best response time with wasted resources, and the best resource usage but with a larger response time. One solution to mitigate this issue is depicted in Figure 20. Instead of generating one DAG per request, multiple request methods are aggregated into a single DAG. Doing so shares the available execution resources among all requests, leading to more efficient use of the available resources. While the ready tasks from one request provide a low level of parallelism, the ready tasks from another request may expose a high level of parallelism, and overall there is a better resource usage.

Moreover, if web applications are deployed in a cloud infrastructure, the cloud's elastic properties allow the number of resources to be adjusted according to the task load in the aggregated DAG. Doing so improves the response time without increasing the average resource usage. So, when the COMPSs runtime detects that there are too many pending tasks in the aggregated DAG and that the expected execution time with current resources is large enough to compensate the time to deploy a VM, the COMPSs Resource Manager contacts the VM Manager in the Cloud middleware to request the creation of a new VM. In contrast, when the COMPSs runtime predicts that executing the pending tasks on the current resources will likely cause too many resources to be idle (i.e. that there

will be a waste of resources), the COMPSs Resource Manager contacts the VM Manager in the Cloud middleware to request the destruction of VMs.

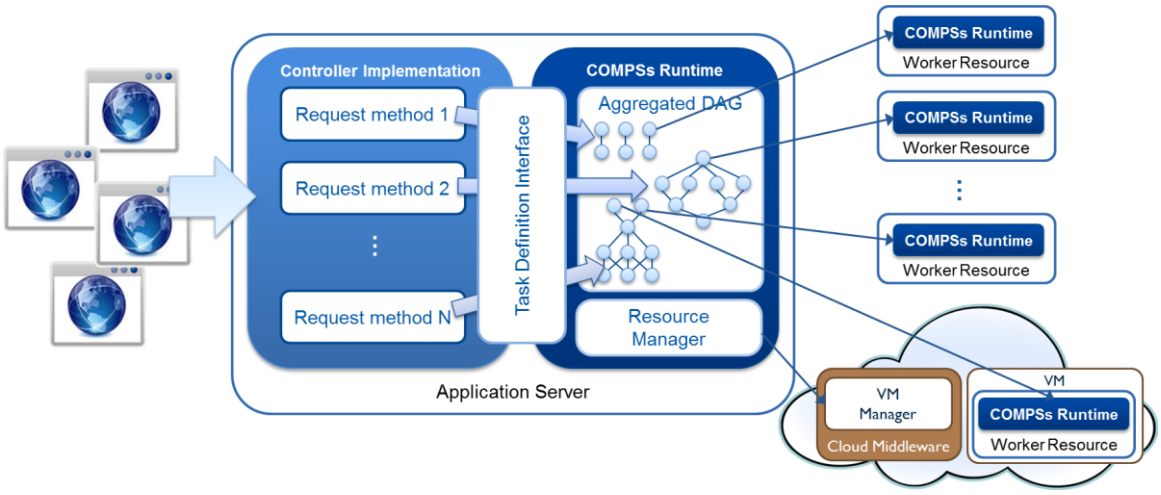

Figure 20: Aggregated DAG solution

Finally, the scalability provided by COMPSs can be combined with traditional web-application scalability. Although most of the request code is executed on the worker resources, there will still be some code executed on application server threads, which consumes application server resources. As the number of requests grows, this resource can be exhausted, requiring replication of the application server. A first approach for integrating both scalability methods would be to duplicate the solution depicted on Figure 20 for each of the application server replicas, but a more resource-efficient solution is to have the workload of multiple application servers managed by a common COMPSs scheduler.

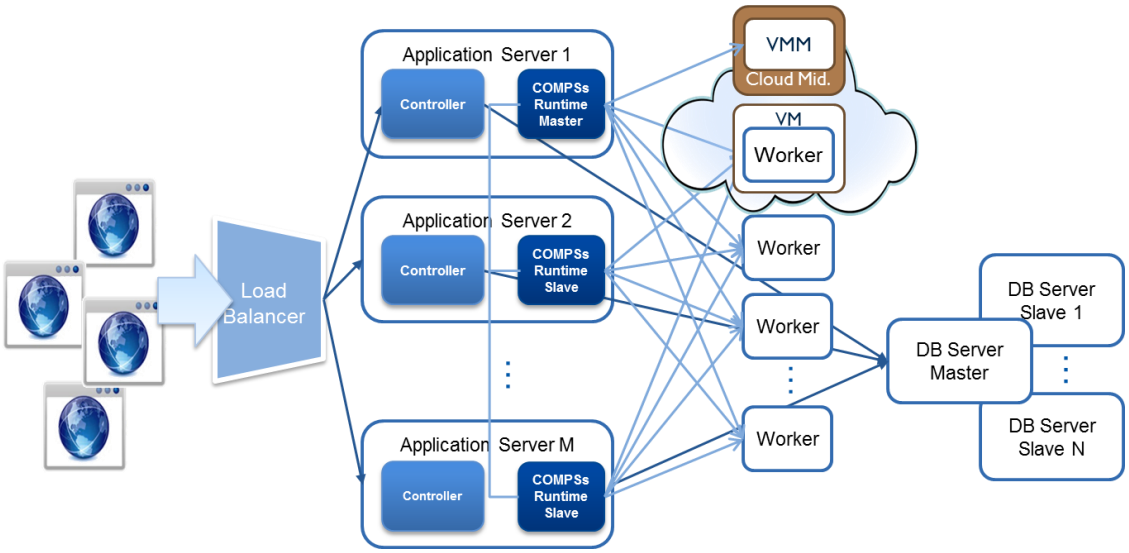

Figure 21: Combination of traditional web application scalability with COMPS

An initial implementation of the latter solution is depicted in Figure 21. Every replica of the Application Server deploys a controller implemented with COMPSs and the corresponding runtime. All the runtime instances contain the Task Analyzer component to generate the request DAGs. Then,

instead of calling their own scheduler to plan the task execution, the generated tasks are submitted to a master runtime instance which contains a global Task Scheduler and Resource Manager in order to make a global scheduling and resource management. Finally, once the execution is planned, the Job Manager of the different slave instances performs the task execution.

To validate the integration of COMPSs with web applications, we have implemented a Gene Detection web application. Each request of this application performs an automatic search and analysis of the relevant genes in a selected DNA sequence. The COMPSs runtime and the web application have been deployed on a cluster of boards based on ARM Cortex-A15, virtualized with OpenStack and KVM. The COMPSs runtime has been configured to use two VMs as worker resources and a resource manager connector to enable the dynamic creation and destruction of VMs by contacting the VM Manager implemented in Task 4.4.2.

To demonstrate and test this integration, we logged in as the web application user and performed the following requests. First we requested a small-scale search and analysis. As expected, the runtime detected the relatively low level of parallelism among the tasks, and it executed them on the available worker resources. Then we executed a request for a larger-scale analysis. In this case, the COMPSs runtime detected a greater number of parallel tasks, so it requested two additional VMs to speed up the request execution. Shortly after the analysis had finished, the extra VMs were destroyed. Finally, we performed several small requests in order to validate that the request DAGs were aggregated and the resources were scaled up and down as expected.

4.4. Energy-aware scheduling

Recent studies [6][7] have estimated that around 1.5 percent of the total electricity consumption is consumed by data centers and this energy demand is growing extremely fast due to the popularization of Internet services and distributed computing platforms such as grids and clouds. Regarding the efficiency of data centers, studies have concluded that, in addition to the energy consumed by the computation system, a considerable part of the energy is consumed by support systems such as cooling, Uninterrupted Power Supplies (UPSs), etc. For that reason, saving 1 W in computation becomes a total saving of 2.8 W saving due to this cascade effect [8].

There are several complementary ways to reduce the energy consumed by an application. One way is to use low-power processor architectures and technologies such as FD-SOI and Silicon-in-Package integration, as pursued by EUROSERVER WP3 and WP5, which leads to an upgrade in the computation infrastructure. Another way is to redesign the application algorithms using energy efficient patterns [5] [12], whose implementation overheads can be unaffordable. A final way is to change the policies that define how the parts of an application are scheduled on the resources.

Traditionally, scheduling policies have tried to minimize the application's total execution time without caring about the energy consumption. In some occasions, however, the scheduling solution that minimizes the execution time may consume more energy than a different solution that has a longer execution time. In these occasions, there is a trade-off between energy consumption and execution time.

The following paragraphs present an energy-aware scheduling system for task-based applications, which takes into account the application's estimated energy consumption, in order to determine

which scheduling solution is best, depending on the user's preferences. First, we give an overview of how the proposed energy-aware scheduler works; then, we present how the energy and time consumptions are estimated, and finally, we conclude the section with the system evaluation.

Overview

The main objective of the energy-aware scheduler is to introduce the energy consumed by an application as another metric for the scheduling algorithm to take into account, but there are other requirements and constraints that must be considered when designing and implementing the scheduler.

First, the relative importance of energy consumption versus performance, the latter being represented by the total execution time in our case, varies depending on the type of application and the user's preferences. So, the scheduler has to let users indicate their preferences indicating which is more important: performance or energy.

The energy-aware scheduler has to act as a runtime scheduler, so the overhead to find the best solution should be low compared with the task and application duration. The computational complexity of finding an optimal solution is high and could require too many resources. For that reason, the scheduler should use heuristic rules to find near-optimal solutions, which are fast and consume few resources.

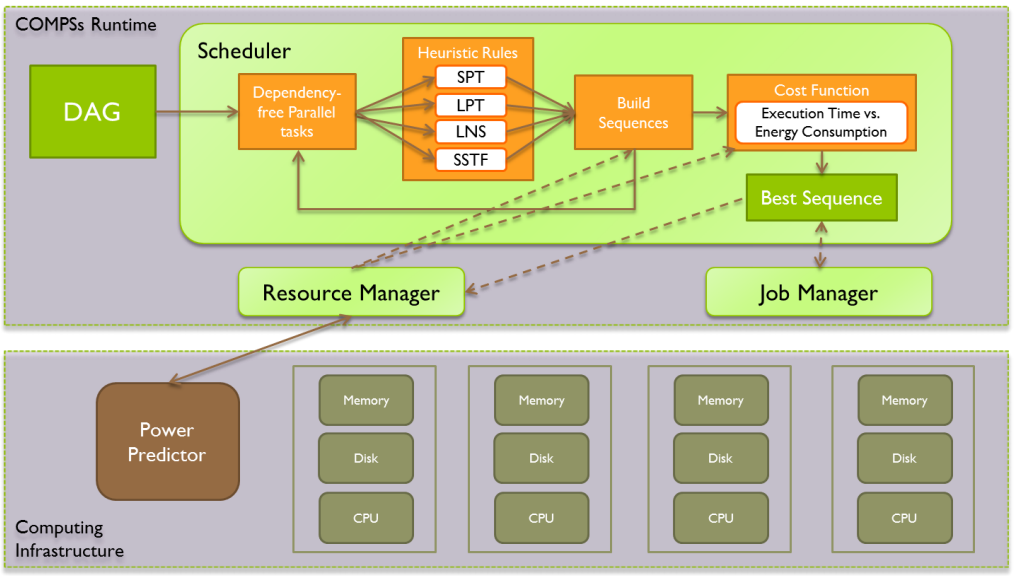

Figure 22: Runtime energy-aware scheduler

Based on the aforementioned requirements, we have designed the energy-aware scheduler depicted in Figure 22. The scheduler has a DAG as input and, it creates a set of scheduling solutions, by applying multiple heuristics, including Shortest Processing Time (SPT), Longest Processing Time (LPT), Largest Number of Successors (LNS) and Setup Transfer Time First (STTF), the last of which tries to schedule first the tasks with larger transfers. For each solution, it estimates the total energy consumption and execution time, and using these metrics it calculates the cost of the solution according to a cost function. Finally, the scheduler selects the solution corresponding to the minimum cost.

The cost function used by the scheduler is shown in the below equation. It is a normalized bi-objective function that combines the estimated execution time and the estimated energy consumption, using an importance factor, α, where 0 ≤ α ≤ 1. The value of α indicates which is most important to the user: energy (α close to zero) or performance (α close to one).

$$Cost = \frac{T_{max}}{T_{NF}}\alpha + \frac{E_{max}}{E_{NF}}(1-\alpha)$$

Energy and Time Consumption Estimation

Figure 23 shows how the application's execution time and energy consumption are estimated. Given the DAG that represents the application to execute, the scheduler generates a scheduling solution applying a heuristic rule. The estimated total execution time is the difference in wallclock time between the end of the final action (end of VM1 destruction in the figure case) and the beginning of the first action (start of VM1 creation in the figure case). The estimated energy consumption for a given solution is the sum of the partial energy consumptions of the various actions performed during the application's execution. These actions include task executions (blue), data transfers (red), idle states (orange) and, in case of cloud computing, the hypervisor creating and destroying Virtual Machines (green). The sum of all these contributions gives the application's total energy consumption.

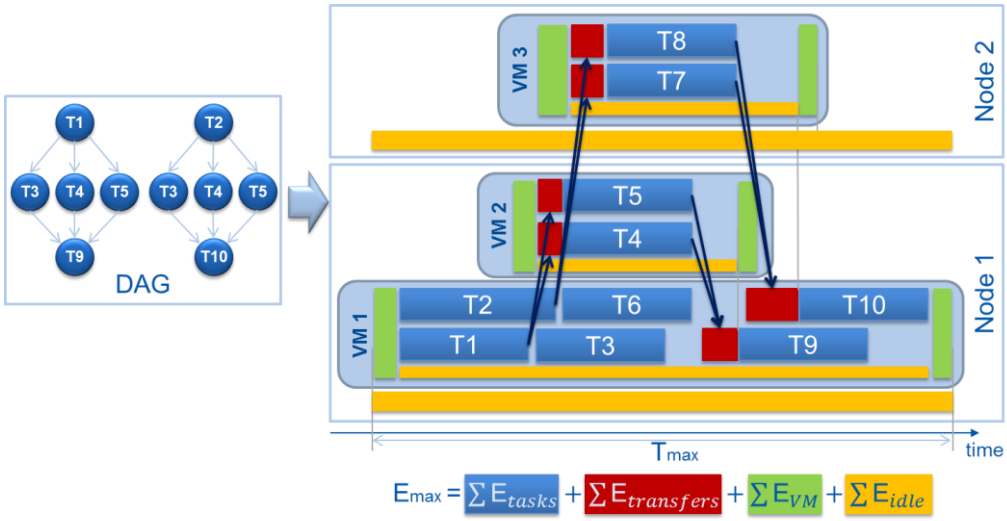

Figure 23: Execution Time and Energy Consumption Estimation

The different partial energy consumptions are estimated by multiplying the mean power consumed during an action execution (tasks, data transfers, etc.) by the expected duration of those actions. The expected duration is calculated by the runtime using statistics from previous runs, and the calculation of the mean power is explained in next subsection.

How to get Mean Power estimation

To calculate the mean power, we partially use the results of Task 4.4.2 where a power-monitoring environment has been set up such as the one depicted in Figure 24. They have integrated the metrics provided by the Power Distribution Units with the Ganglia Monitoring tool in order to measure the power consumed by cluster nodes with different computational loads. To estimate the different

mean power values, we can run a set of benchmarks and calculate the mean of the power measurements during the benchmark execution. So, to calculate the mean power when idle, we just need to calculate the mean of the measured power during idle intervals. The same can be done for file transfers and for hypervisor tasks (deploying and destroying VMs).

In the case of the task execution, calculating the energy consumption is more complex because it depends on the resources utilized by the tasks. To get the resource usage, the runtime periodically profiles the executions of the different type of tasks and sends it to the Power Predictor, which provides an estimation of the mean power consumed by tasks applying a power model. The work to build the power model is under development at T4.4.2 and more details about the model will be provided in D4.5 (note that a draft version of Deliverable D4.5 covering this material was provided in M22).

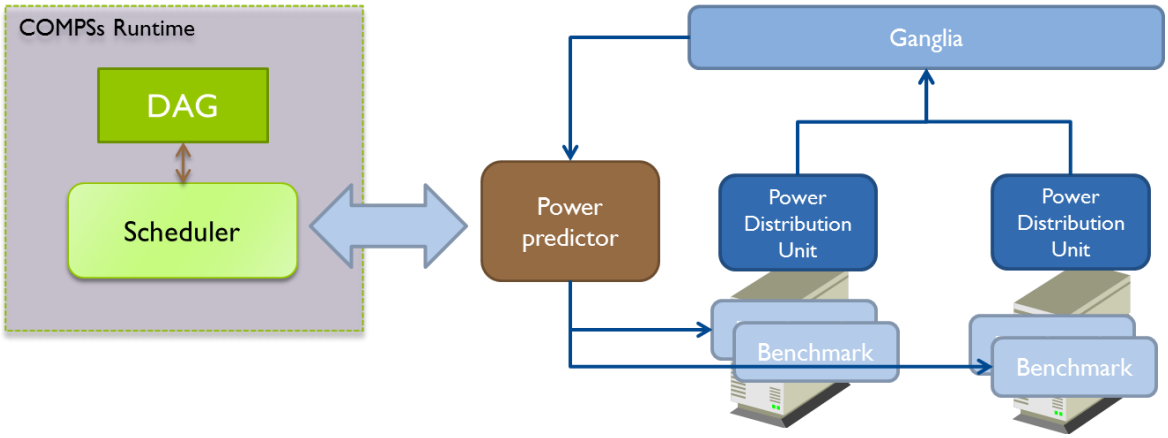

Figure 24: Mean Power Estimation

4.5. Evaluation

A prototype of the energy scheduler has been implemented with a set of existing heuristic rules, including SPT, LPT, LNS and STTF, as defined above. We have generated different DAGs to evaluate how the scheduler behaves in different situations. The generated graphs include a pipeline graph, a parallel graph, the Matrix Multiplication graph, a gather graph and scatter graph. We have simulated the scheduling of these graphs in a heterogeneous cluster hosted by BSC, which is composed by 4-core AMD Opteron and 6-core Intel Xeon servers, with each VM containing two cores. Monitoring the energy consumption, we measured that the Intel servers are more efficient than the AMD in idle times and data transfers, but that the increment per active core is quite similar.

For each of the graphs, we have studied the effect of the importance factor, α, identified which heuristic rule provides the best solutions, and evaluated the expected energy savings. In addition to these, we also evaluated the introduced overhead by measuring the time for finding the scheduling solutions, as a function of the number of tasks in the DAG.

Overhead

To evaluate the overhead introduced by the scheduler, we measured the time to determine the scheduling solution for different numbers of tasks. The results of these measurements are shown on Figure 25. For instance, in the case of ten target cluster nodes (60 cores) and an execution of 1,000

tasks whose mean duration is 10 seconds per task, the total application execution time is around 170 seconds and the scheduling time is about 190 milliseconds. This gives a scheduling overhead of 0.11%. Even for smaller tasks (~1 sec.) the scheduling overhead will be just 1.1%, which can be considered suitable for a runtime scheduler.

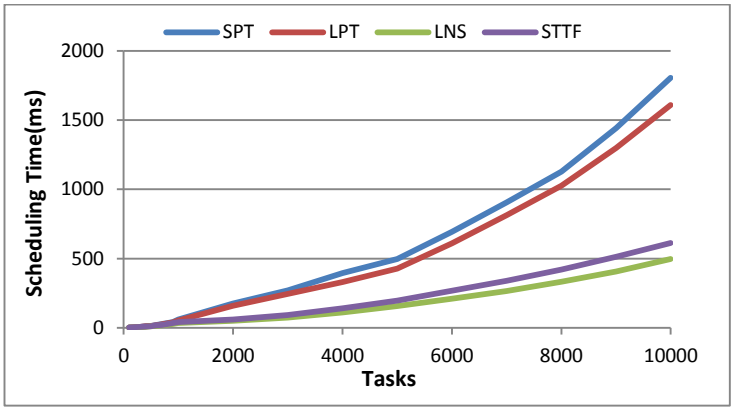

Figure 25: Scheduling overhead measurements

Importance Factor

Regarding the effect of the importance factor, we have inspected the estimated execution time and energy consumption estimated by the scheduler for the different graphs, heuristic rules and importance factors. Figure 26 shows the results obtained for the Parallel graph with 500 tasks, where we can observe the trade-off between the energy consumption for the different factors. Comparing the extremes (α=1 to α=0), we see a reduction in energy consumption of 20% but a considerable increase in execution time. However, in the case of an intermediate value (STTF rule and α=0.4), we can achieve a 10% saving while only increasing the execution time by 3%.

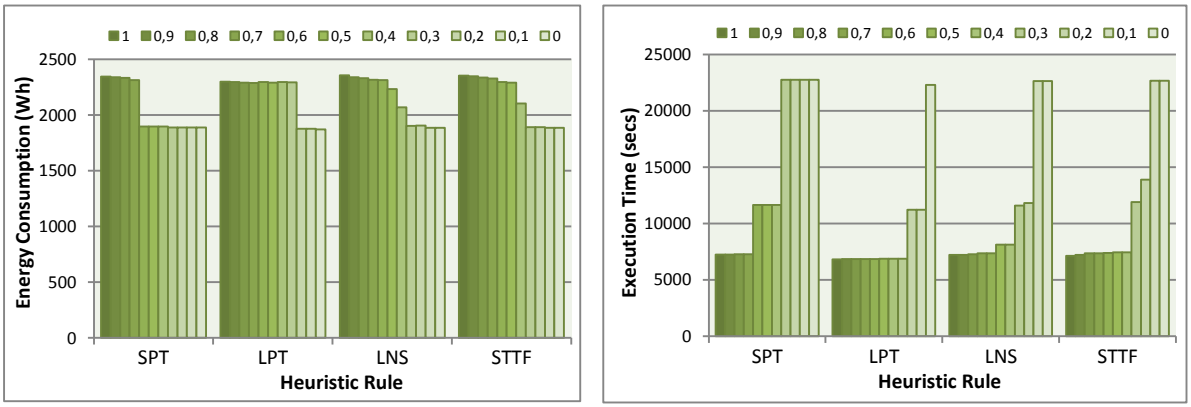

Figure 26: Estimated energy consumption and execution time for different heuristics and importance factors

Elasticity

Another important issue we observed is related to the trade-off between the importance factor and the elasticity. Figure 27 shows a timeline with the VM deployments decided by the scheduler, for the Scatter-Gather graph with different importance factors. The Scatter-Gather graph is an interesting case that shows the effects of application elasticity, because it starts with a single task, then its parallelism increases progressively until a certain number of parallel tasks is reached, and then the

parallelism reduces until the application ends with a single task. The figure shows that when the energy is the most important, the scheduler tries to reduce the number of VMs in order to have fewer transfers and hypervisor management tasks, and consequently less energy consumed. In contrast, when the performance is more important, the number of VMs is increased.

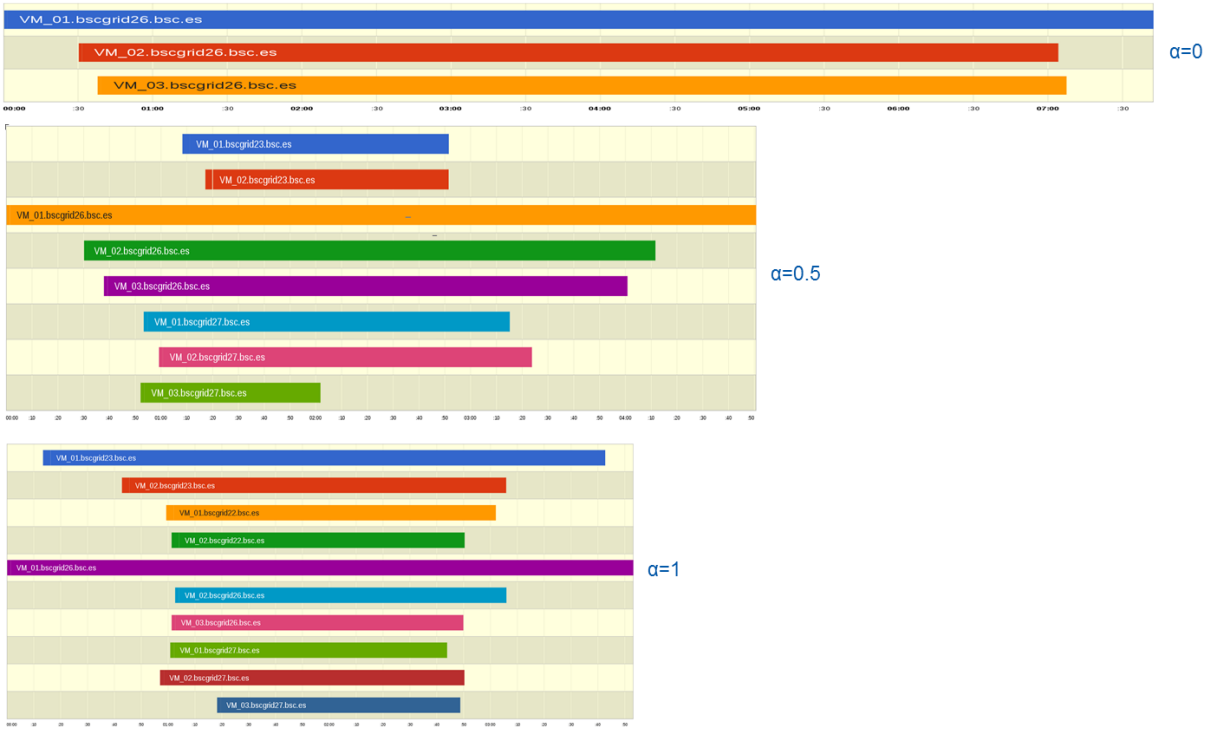

Figure 27: Trace of VMs created for a scatter-gather graph depending on the importance factor (α)

4.6. Ongoing and future work

COMPSs for web applications

Section 4.3 has presented how COMPSs can be used to implement scalable web applications. In ideal conditions, this solution could be used for any web application whose requests contain some implicit parallelism. However, in the real case, a considerable amount of computational work is required per request, in order to compensate for the overhead introduced by remote invocation protocols and data transfers. COMPSs was initially designed for grid computing, where applications are composed of compute-intensive batch tasks with soft deadline constraints. In this scenario, the submission overhead is not important. However, the most important metric in web applications is the response time, so in this case, overheads are important and, the lower it is, the smaller the response time will be.

For that reason, we are currently implementing a new submission and data transfer protocol, based on the idea of persistent workers and non-blocking I/O. With this implementation, we are trying to reduce the overhead in order to have an efficient execution for web applications with lower computational demands.

Energy-aware scheduling

For simplicity, we have ignored interference between tasks. For CPU-intensive tasks, this interference is not very important, because CPU energy consumption can be usually approximated using linear regression. However, it could be important for memory-intensive tasks, where interference increases the number of cache misses, increasing main memory accesses, which increases the energy

consumption in a potentially non-linear manner. One of the next steps will be to identify the intervals where tasks are overlapped and create a model to estimate the resource consumption of overlapped tasks in order to obtain more accurate power estimations.

The implemented greedy algorithm is fast but sometimes far from optimal. Moreover, once the runtime has submitted the first bunch of tasks it remains idle until their computation finishes. This idle time could be used to perform backtracking or backjumping techniques in order to improve the scheduling solution.

Finally, the prototype scheduler has been implemented in a prototyping environment outside the COMPSs runtime but taking into account the expected input and output information. So, part of the ongoing work is dedicated to integrate the energy-aware scheduler in the COMPSs runtime.

Integration with EUROSERVER software stack and prototypes

The current implementation of the COMPSs runtime has been deployed on a cluster of boards based on ARM Cortex-A15. This allowed us to validate that the programming model and runtime function correctly on ARM boards. As future work, we will port the current COMPSs framework and associated implementations to the 64-bit EUROSERVER prototypes. First, we will deploy the COMPSs runtime and testing applications on the 64-bit discrete prototype in order to validate the proper behavior of the COMPSs runtime on chips based on ARM Cortex-A53 chips so as to detect potential problems in advance. Once the silicon prototype is available, we will migrate all the work done in the 64-bit discrete prototype to the silicon prototype.

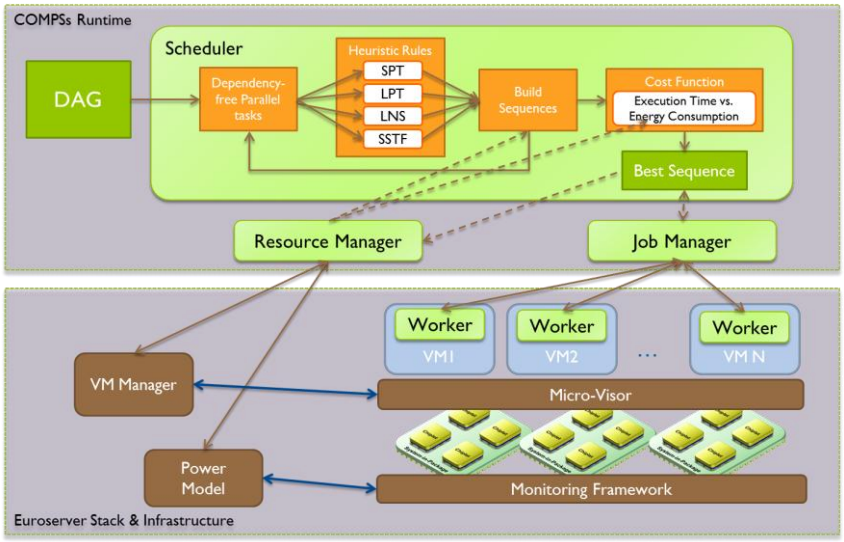

Figure 28: Integration of the COMPSs Runtime with the rest of the EUROSERVER software stack

We will also integrate the runtime with the rest of the EUROSERVER WP4 software stack, in the manner indicated in Figure 28. The COMPSs runtime will interface on top of the VM Manager that is being developed in T4.4.2 and reported in D4.5.[12] This will be the main interface used by COMPSs to create and remove the VMs used to execute the application. This integration has been implemented during the first part of the project and it was demonstrated as part of the review demonstrations. In terms of energy scheduling, we will integrate the power model implemented in WP4 to estimate the

[12] Task 4.4.2 will be reported in D4.5, due in M36. A draft of this D4.5 was delivered to the EC on 19 June 2015, in advance of the 2nd Intermediate Review.

energy consumed by the execution of tasks in the EUROSERVER infrastructure. As future work, we plan to demonstrate the runtime capabilities on top of a virtualized layer managed the VM Manager, OpenStack and the MicroVisor developed by OnApp.

5. Conclusions

This document described the current status and future plans of the ongoing work in WP4 on Cloud-RAN (T4.3.3), MQTT (T4.3.4), and COMPSs (T4.3.5). Cloud-RAN (CRAN) is a system for virtualization of radio access protocols, network functions, edge services and management on a common hardware platform, which represents the telecommunications use case. MQTT is a lightweight publish/subscribe messaging protocol designed for machine-to-machine (M2M) communication, specifically for constrained devices and low-bandwidth, high-latency or unreliable networks, which represents the transportation use case. COMPSs is a framework and programming model for coarse-grain task parallelism, designed for distributed platforms such as clusters, grids and clouds.

Work on Cloud-RAN and COMPSs is ongoing, but MQTT has been deprecated now that Eurotech has left the project consortium. Future work on Cloud-RAN will improve resource utilization and evaluate the use of the UNIMEM approach. Future work on COMPSs will include integration with the VM Manager. Both will be evaluated on the 32-/64-bit discrete prototype and the silicon prototype, and are to be integrated with the rest of the EUROSERVER software stack.

References

[1] ActiveMQ, activemq.apache.org.

[2] Apache Karaf, karaf.apache.org.

[3] Enric Tejedor and Rosa M. Badia. "COMP Superscalar: Bringing GRID superscalar and GCM Together" *in Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid*, 2008

[4] Eurotech Everyware Device Cloud. http://www.eurotech.com/en/products/software+services/everyware+device+cloud

[5] Jason Flinn and M. Satyanarayanan. "Energy-aware adaptation for mobile applications." *In Proceedings of the seventeenth ACM symposium on Operating systems principles*. pp 48-63, 1999.

[6] Jonathan Koomey, *Growth in Data Center Electricity Use 2005 to 2010*, Analytics Press, 2011.

[7] Natural Resources Defense Council, *Data Center Efficiency Assessment. Scaling up energy efficiency across the Data Center Industry: Evaluating Key Drivers and Barriers*, 2014

[8] Emerson Network Power, "Energy Logic: Reducing Data Center Energy Consumption by Creating Savings that Cascade Across Systems" *in A White Paper from the Experts in Business-Critical Continuity*, 2009.

[9] Memcached, memcached.org

[10]www.mqtt.org/faq

[11]OASIS MQTT Version 3.1.1, 29 October 2014. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

[12]Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. "EnerJ: approximate data types for safe and general low-power computation". *In Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation,* pp. 164-174, 2011.