D-RES: Correct Transitive Distributed Service Sharing

Augusto Born de Oliveira, Akramul Azim, Sebastian Fischmeister
Electrical and Computer Engineering
University of Waterloo
Waterloo, ON, Canada
{*a3olivei,aazim,sfischme*}*@uwaterloo.ca*

Ricardo Marau
University of Aveiro
Aveiro, Portugal
marau@ua.pt

Luis Almeida
IT / Faculdade de Engenharia
University of Porto
Porto, Portugal
lda@fe.up.pt

***Abstract*—With the growth of complexity in the embedded domain, the use of distributed systems to support multiple real-time applications has become commonplace. These applications may share processor and network resources, and real-time scheduling policies can guarantee that these applications do not interfere with each other's ability to meet their temporal constraints. We believe that these applications should also be able to transparently share *services* and *chains of services*, without the coupling that such sharing typically implies. To solve this problem, we propose D-RES, a resource management system that guarantees temporal isolation between service-sharing applications in a distributed system. D-RES transparently tracks which application uses which service, billing the correct application even in case of nested service calls. We implemented D-RES, and demonstrate its ability to isolate service-sharing applications even in case of overload.**

***Keywords*-Real-Time, Distributed, Service Partitioning**

I. Introduction

Modern distributed embedded real-time systems, such as avionics or automotive systems, have been subject to an increasing pressure towards reducing costs acting, among other aspects, on weight, cabling complexity, number of processing units and power consumption. Harmonizing all these aspects with an ever growing functionality can only be achieved by means of sharing resources, making different functionality co-exist in processor units and networks, even if they are of different criticality. However, this sharing has a high risk of creating interference across different criticality levels, whereby best effort applications cause critical ones to miss their deadlines.

Resource partitioning techniques were devised [1] to mitigate the mutual interference problem arising from resource sharing, allowing the temporally isolated execution of multiple applications on the same processor, or the temporally isolated transmission of multiple streams over the same communication link. Consequently, it also allows deriving timing guarantees based on temporal partitioning properties, independently of the actual behavior of the applications/streams that run/flow inside. This makes resource partitioning a key instrument to support mixed criticality systems, open systems, or even adaptive systems with real-time requirements.

In spite of the large body of work on partitioning or reservations on individual resources, such as processor [2], memory [3], disk [4] or network [5], few works addressed the whole distributed system such as [6] and [7]. To enable system-wide reservations, a management mechanism must associate the resource consumption of *each application* to partitions in all the different shared resources, and guarantee temporal isolation between those applications.

However, such mechanisms do not cope well with typical programming patterns such as remote procedure calls in client-server[1] applications or service invocations in service-oriented applications. In fact, as discussed in Section VII, existing distributed resource management approaches do not provide temporal guarantees when sharing services such as video servers, sensor servers, filters or even databases.

In this paper we propose D-RES to address the sharing of *services* as resources, meaning that multiple applications can use the same service with guarantees of temporal isolation. Moreover, our mechanism has the specific feature of correctly handling transitivity in nested service invocations so that their execution can be billed to the appropriate reservations. Our framework has been partially developed within the iLAND project [8] targeting reconfigurable real-time service-oriented applications.

The contributions of this paper include: (1) The D-RES global resource management mechanism, which enables temporally isolated sharing of services, with support for transitivity handling in nested service executions, and (2) an end-to-end response time analysis based on the properties of our implementation of D-RES.

II. The Problem of Sharing Services

Consider a distributed system comprising multiple real-time applications that share resources (e.g., computation time on the processors, communication bandwidth on network links). Traditionally, *reservation policies* (also called quality-of-service (QoS) policies in distributed systems) have been used to partition shared resources to ensure that applications do not interfere with the timely execution of each other. This is especially important for hard real-time applications, since unaccounted temporal interference may cause disastrous consequences.

If applications need to share *services* in addition to regular resources, simple reservations may be insufficient to guarantee the specified QoS. For example, consider the system in Figure 1, composed of a low-priority video-player application and a high-priority telemetry application, running on Node A, which share a remote file service, running on Node B. The requests from the applications are

[1]Unless written otherwise, we use *service* to refer to a server entity in a client-server relationship, and *server* to refer to a partition manager as typical in hierarchical scheduling.

made in a client-server manner and cause the caller to block while retaining the processor. The video application has a WCET of 2 time units, and the telemetry application has a WCET of 1 time unit. Both applications have a period of 3, and are therefore locally schedulable via both EDF and RM. However, since both applications make requests to the remote file service, they must account for its execution time when considering schedulability. If the service responds in a timely manner, such as shown in the first two periods, all deadlines are met.

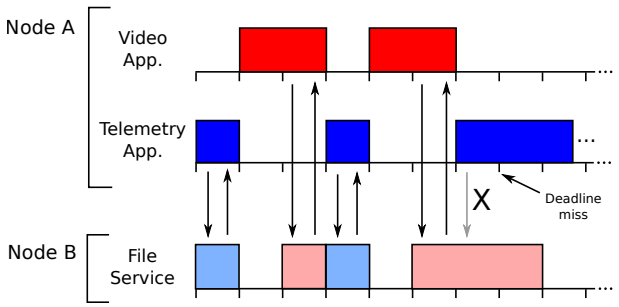

Figure 1. Example of a shared service causing deadline misses.

If the video player enters a faulty state and makes an erroneous request to the file service, causing it to go into a faulty state (as shown in its second instance), the critical application may suffer unbounded delays in a classic priority inversion scenario *even though temporal isolation is upheld at Node A*. This inversion occurs because the reservation policy does not account for the *service* they share. Therefore, it will be unclear at run time which application to charge for the file service's resource consumption, and whether the request from the video application should be preempted to allow the request from the telemetry application to execute. Furthermore, if the video player saturates the network, the telemetry application's messages may fail to even reach the service; therefore communication delays must also be deterministic, and the network must be included in the timing analysis.

This problem is even further complicated in case of transitive invocations. For example, if the file service shown in Figure 1 is used as a disk service in the application's behalf. That disk service's resource consumption must be "charged" to the correct application despite being invoked by the file service. In a distributed system, this transitivity requirement can even span across network links, for example when the disk service is hosted on a different node than the file service. In this paper we address the general problem of resource reservation in distributed systems with D-RES, focusing on managing services as resources, particularly adding protection against service overloading. Moreover, we also address the specific situation of nested service invocations of arbitrary depth.

III. D-RES: Distributed Service Sharing

The D-RES system model is composed of periodic active tasks, called applications, and passive tasks, called services. At the beginning of each period of an application, a new job is released. These application jobs can invoke services, causing service jobs to be created. Services can also invoke other services transitively. This model allows for services that handle concurrent, independent calls by multiple clients, but we do not allow shared state between service instances for deterministic timing analysis.

When a service invocation happens remotely, a network message is created. We use Adaptive Partitioning Scheduler (APS [9]) for computation and Flexible Time-Triggered Switched Ethernet (FTT-SE [10]) for communication. Both APS and FTT-SE enforce periodic reservations in a form similar to sporadic servers such that overloads in one application do not affect the timeliness of others. When services invoke other services transitively, we use mTags [11] for correct billing, which are metadata extensions to microkernel inter-process communication (IPC).

The goal of D-RES is to permit applications to make reservations in shared services, so that, if any client exceeds their reservations (causing an *overload*), then the other clients' quality of service will suffer minimal and bounded QoS degradation. D-RES accomplishes this by isolating each service's resource usage according to its originating client, and accurately enforcing every reservation over a bounded period of time.

D-RES assumes that: (1) resource reservation mechanisms are available for sharing active resources, e.g., processors and networks; (2) all resource usage is tracked at all times, i.e., all resource usage occurs within reserved partitions; and (3) that the resource usage of a service can be dynamically billed to different applications. A service, then, uses such a resource reservation to bill its clients for any work it does on their behalf. If two requests arrive concurrently at a service, then that service will serve those requests concurrently in isolation, and their individual resource usage is billed to the originating client. Services must, therefore, support accurate isolation of concurrent work. We implement this by having a lightweight dispatcher that ensures work is done on behalf of the correct client, and since a service activation can be interrupted by another activation of the same service, service code must be reentrant.

Shared services may still have internal critical sections, causing blocking among instances of the same service when concurrent requests arrive. These critical sections must be strictly bounded to support schedulability analysis. The maximum blocking that any service can suffer must then be taken into account when reserving resources by the application. For simplicity, D-RES assumes no abuse of critical sections.

In D-RES, applications must make reservations for themselves *and* all services they intend to use. While handling a request from a client, a service will bill its resource usage to that client's reservation. D-RES provides transitive resource tracking: if a service makes use of other services on behalf of an application (e.g., a real-time file server using a memory file system), then the originating application will be billed for all resource usage calculated at run time along this service chain.

IV. Components of D-RES

D-RES requires three basic functionalities: resource partitioning, per-client service isolation, and transitive client tracking. The components of D-RES are APS, FTT-SE, and

mTags, which implement the functionalities. The Adaptive Partitioning Scheduler (APS) [9] is a reservation-based scheduler that limits mutual interference among applications running on one processor. A system administrator is able to create *partitions*, each with a budget of time that it is to receive every *averaging window*. This is a sliding window that approximately bounds the longest period of time a partition may be starved of computation time, as well as the maximum interference that a partition can incur on other partitions. Flexible Time-Triggered Switched Ethernet [12] [10] is a real-time communication protocol that exploits the master/slave paradigm to control the traffic submitted to the network. The traffic scheduling is carried out by the master which grants the flexibility of using any desired policy, including hierarchical reservation-based [13]. mTags [11] are label-like pieces of meta-data that can be attached to applications and were designed for use in microkernel environments where all services communicate via message-passing. mTags are associated to applications. Whenever a tagged application executes a job, its tags are passed on to message receivers who inherit the tags.

V. End to End Timing Analysis of D-RES

In a client-server architecture, it is usual that the system can be modelled with applications and services composed of several segments [14], where each segment requires a certain amount of execution time. For example, consider that a client application issues service requests to several servers, possibly executing on different nodes. The client application can be modelled as several segments that either terminate making requests or are released by completed requests to services. Services, in turn, can also be modeled as sequences of segments that invoke other services. In this work, we adopt this approach of decomposing applications and services into segments.

Having introduced components in the previous section, we now analyse worst-case timing requirements of D-RES. Each application $i = 1 \dots n$ running on a set of $\aleph$ nodes characterized by the following parameters:

- $\{\tau_i^u\}$ = set of segments of application i where $u = 1 \dots m_i$ and $m_i = |\tau_i^u|$.
- ν_i = node where the application i is allocated.
- tag_i = tag of application i and its associated segments.
- $C_i = \sum_{x \in \{\tau_i^u\}} c_i^x$ WCET of all application i segments, excluding services.
- $S_i = \{\sigma_j | j = j(u), u = 1 \dots m_i - 1\}$ set of services invoked by the first $m_i - 1$ segments in application i.
- $\{T_i^u\}$ = set of application segments' minimum inter-arrival time.
- T_i = application minimum inter-arrival time and $T_i = \min_u(T_i^u)$.
- $\{D_i^u\}$ = set of application segments' deadline and we consider $D_i^u \leq T_i^u$.
- D_i = application deadline and $D_i = \max_u(D_i^u)$.
- $\{P_i^u\}$ = set of application segments' priorities.
- P_i = application priority and $P_i = \min_u(P_i^u)$.

and a service σ_j is characterized by:

- $\{\sigma_j^v\}$ = set of segments of service j where $v = 1 \dots m_j$ and $m_j = |\sigma_j^v|$

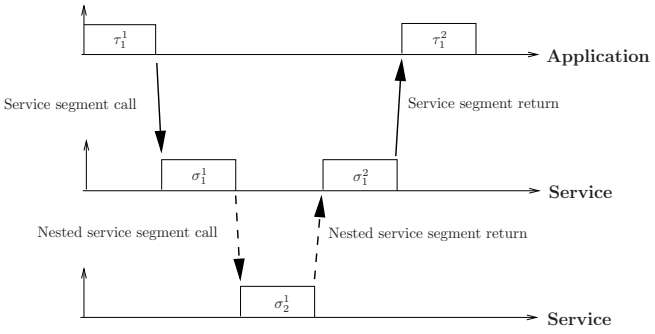

Figure 2. An example of an application chained execution

- ν_j = node where the service (i.e., segments) is allocated.
- $\Gamma_j = \{tag_k\}$ where k is an index to all the applications that invoke this service directly or indirectly.
- $C_j^s = \sum_{y \in \{\sigma_j^v\}} c_j^{s,y}$ WCET of all service j segments, excluding chained services.
- $S_j^s = \{\sigma_h | h = h(v), v = 1 \dots m_j - 1\}$ set of chained services invoked by the $m_j - 1$ segments in service j.

To illustrate this model, consider an application with two segments τ_1^1 and τ_1^2 as shown in Figure 2. The application segment τ_1^1 calls service σ_1 which in turn has two segments and invokes service σ_2. Finally τ_1^2 executes triggered by the termination of service σ_1.

Finally, whenever the caller of a service and the service itself reside in different nodes, two messages are created in the network, for invocation (μ_j^{in}) and response (μ_j^{re}), respectively, which receive and convey the respective application tag.

Following the model above, we set a number of reservations in the system resources to accommodate the execution requirements of the application and all the services it invokes, directly and indirectly, and the transmission of the required messages. These reservations are then scheduled in the respective resources according to a fixed priority scheduling policy.

The reservations are also set differently in the nodes and network. For application i, a computing reservation is made in all the nodes where the segments of an application c_i^x and its invoked segments of services $c_j^{s,y}$ execute. Thus, in each such node r, application tag tag_i has an execution requirement of

$$C_i^r = \sum_{\forall j : \nu_j = r \wedge tag_i \in \Gamma_j} \; \sum_{y \in \{\sigma_j^v\}} c_j^{s,y} + \sum_{x \in \{\tau_i^u\} \wedge (\nu_i = r)} c_i^x$$

The corresponding tag_i APS partition in node r is set with a bandwidth at least C_i^r / T_i. Note that all APS partitions in each node also share a *period* parameter that sets the temporal resolution in scheduling the partitions. The actual length of the APS partition can be set using the server design technique in [15] or equivalent, that considers both needed bandwidth and server activation delay. If occasional deadline misses are tolerable, then a slightly more efficient

partition can be set to $C_i^r/T_i * period$, considering bandwidth requirements, only. Finally, the priority of the partition is set to P_i.

In the network resource, we make use of FTT-SE asynchronous messages which are automatically executed within a simplified sporadic server per message that enforces a desired minimum inter-arrival time together with a maximum message size. These servers are reservations that we set implicitly in the network when creating each message, with capacity equal to the message transmission time. There is also a fixed priority parameter that we set to the application priority P_i. The minimum inter-arrival time is also inherited from the corresponding application parameter T_i.

In this work, we derive the worst-case timing requirement using response-time analysis (RTA). If we consider that Rwc_i is the worst-case response time of application i, including the execution of all the services it invokes and the transmission of all its messages, then the schedulability test reduces to checking condition as in Equation 1.

$$\forall_{i=1..n} Rwc_i \leq D_i \tag{1}$$

In order to compute Rwc_i, we sum all the worst-case response times [16] by invoking $\mathrm{WRT}(c_i^x, , tag_i)$ and $\mathrm{WRT}(c_j^{s,y}, tag_i)$ for application tag tag_i plus the worst-case response time for the messages that are used in the communication with remote services carrying the same tag_i. These are FTT-SE asynchronous messages whose temporal analysis was discussed in [10]. We represent with R_j^{in} and R_j^{re} the worst-case response time of messages μ_j^{in} and μ_j^{re}, respectively. Equation 2 gives us an upper bound on Rwc_i where A is used for simplicity, being true whenever the invoker of the service j and the service itself reside in different nodes.

$$Rwc_i \leq \sum_{r \in \aleph} \Big(\sum_{\forall j : \nu_j = r \wedge tag_i \in \Gamma_j} \sum_{y \in \{\sigma_j^v\}} \mathrm{WRT}(c_j^{s,y}, tag_i) + \sum_{x \in \{\tau_i^u\}} \mathrm{WRT}(c_i^x, tag_i) \Big) + \sum_{\forall j : tag_i \in \Gamma_j \& A} \left(R_j^{in} + R_j^{re} \right)$$

VI. Experimental Evaluation

To validate the feasibility and usefulness of D-RES in a real setting, we implemented a prototype mixed-criticality real-time system based on the computer systems of a modern car. Figure 3 shows the topology of the system: in it, a hard real-time application — engine, steering and breaking control — uses a sensor service. A best-effort logging application stores data about the car on a file server. The file server, in turn, uses a file system to store data. A media player streams video data from the same file server the logging application uses. These applications and services execute in four nodes: a sensor node, a control node, a media node, and a file node. This experiment stresses all main aspects of D-RES: isolation of computation and network resources in the presence of hard and soft real-time applications, the sharing of services, and transitive client tracking.

The experimental setup consists of four application nodes and one FTT-SE master node. All machines are off-the-shelf Pentium 3 or 4 systems with CPUs ranging from 1 to 3GHz, and with memory ranging from 512MB to 1GB. All nodes execute an mTags-enabled version of QNX Neutrino 6.5.0 and FTT-SE 2.5.1. The FTT-SE elementary cycle length and each processors' APS window sizes were configured to 8 and 10ms respectively, allowing for short response times.

Table I details the period, the empirical execution time, and the analytical WCRT of each component. Jobs for each of the three applications consist of: (1) computation on the application side, (2) a request message from the application to the service it utilizes, (3) computation on the service side, and (4) a reply message from the service to the application. Due to this structure, execution times will be no shorter than three FTT-SE periods, as a total of two messages are being exchanged (one period in each way plus one for the turnaround).

Each application's reservation sizes was inspired by industrial common practice. For example, sensoring and control software execute in isolation in low-power hardware, and therefore reserve most of those platforms' computation power. The video player executes on more powerful hardware, and requires a significant amount of network bandwidth.

While the period for the control application was set to meet the worst-case response time provided by the analysis, the periods for the media and logging applications were set after extensive experimental exploration of their worst-case response time on the distributed platform and were set slightly below the analytical worst-case, for the sake of bandwidth efficiency. Moreover, the periods used were deliberately chosen to be close to the *observed* end-to-end worst-case response times to stress the system. The computation load numbers presented include only useful application load, i.e., it excludes overhead from network protocols, scheduling, interrupts, or any other source.

All applications initialize concurrently at boot time. All of the reservations are made statically, as the current prototype does not support admission control. After service and application threads begin execution, the system enters a steady state where all reservations are being honored, and, consequently, all deadlines should be met. Table II summarizes the data from all three experiments, showing the minimum, the maximum, the mean, and the standard deviation of the execution times of each application. We will refer to this table as needed in the analysis that follows.

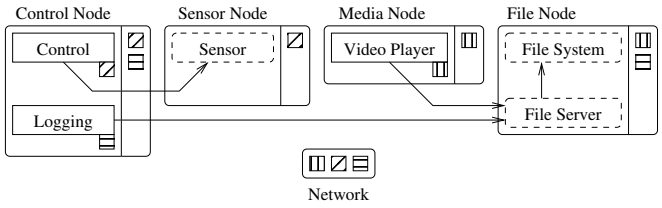

Figure 3. Topology of the Case Study System

A. Experiment 1: Proof of Concept

The first experiment consists of an execution of the system under normal conditions. That is, all applications behave as

App.	Experiment 1				Experiment 2				Experiment 3			
	Best	Mean	S.D.	Worst	Best	Mean	S.D.	Worst	Best	Mean	S.D.	Worst
Control	39.56	44.32	2.881	72.19	39.61	44.21	2.800	60.37	39.21	43.80	2.824	59.98
Log	37.28	44.57	4.748	61.76	37.33	85.75	133.203	1074.00	37.23	82.08	125.014	489.20
Media	52.71	57.80	3.414	76.89	52.76	56.88	2.321	76.64	52.91	82.24	100.245	497.70

Table II
SUMMARY OF DATA FROM EXPERIMENTS 1-3. ALL TIMES ARE IN MILLISECONDS.

Process	Period/Deadline	Comp. Time	WCRT
Control	80ms	15ms	76ms
Log	150ms	5ms	194ms
Sensor	N/A	1ms	N/A
Media	100ms	20ms	110.5ms
File Service	N/A	2.5ms	N/A
File System	N/A	7.5ms	N/A

Table I
CAR SYSTEM PERIODS AND COMPUTATION LOADS

defined in Table I. The experiment executed until the control application reached 1,000,000 executions (approx. 22 hours).

Execution time measurements include the following: a blocking message sent from each application to the service it uses, the computation time of that service (and any services they, in turn, use), the return of the data from the service to the application, and the computation time of each application (as per Table I). The leftmost column of the figure shows that the execution time distribution has a long tail, which is expected in this type of system, where the worst case execution time is often multiple times longer than the average case. The "Experiment 1" column of Table II confirms that all jobs executed within their expected periods, and no deadlines were missed.

This proof-of-concept experiment shows that D-RES is capable of scheduling all the applications in the network without missing the deadlines we attributed to them. Two properties were confirmed in this experiment: the response time for the control task is within the expected bound, and the time of the shared file server and file system services was correctly accounted for.

B. Experiment 2: Overload with D-RES

To illustrate how D-RES is capable of isolating resource consumption, we modified the proof-of-concept experiment to include a "buggy" logging application. This buggy version sets its own priority to be higher than all other user-level tasks, and, once every 10 jobs, generates a file server request that takes approximately 400ms to execute. Even though the hard real-time control application shares CPU with the buggy logging application, CPU partitioning should guarantee its timely execution. Similarly, even though the media application shares the file service with the buggy logging application, service partitioning should keep it from missing any deadlines. This experiment was executed until the control application reached 30,000 jobs, approximately 40 minutes.

Figure 4 shows standard boxplots for the execution time of the media application. The y-axis shows the execution time in a logarithmic scale, and the x-axis denotes the different

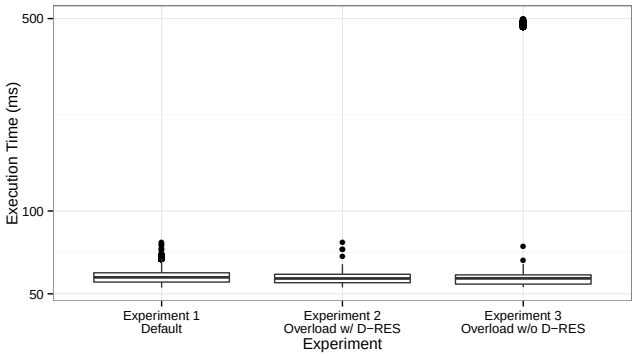

Figure 4. Execution Times for the Media Application

experiments. By comparing Experiment 1 with Experiment 2, we see that the media application was largely unaffected by a buggy application sharing a service it uses; the median and quartiles are nearly the same, with a smaller number of outliers for Experiment 2 (Table II).

C. Experiment 3: Unprotected Service Sharing

To illustrate how the service partitioning functionality of D-RES is necessary to enable shared services, we expand on Experiment 2. By modifying the file service to treat all requests in a non-isolated fashion, requests from both the media and the now buggy logging application were treated without partitioning. Therefore, there will be *indirect* interference between the media and the logging application at the file server, a remote service they share. Like Experiment 2, this experiment was executed until the control application reached 30,000 jobs, approximately 40 minutes.

We refer again to Figure 4, where comparing the boxplot for Experiment 2 with the one for Experiment 3 shows that there is a large spike (remember, the y-axis follows a log scale) in the worst case execution time of the media application. Table II shows that these large spikes cause the media application to miss deadlines, making evident the need for proper partitioning of shared services. Also of note is that allowing the logging application to interfere with the media application causes the overload to be "shared", that is, the worst case execution time for the logging application is reduced in comparison with Experiment 2, where it suffered the overload in isolation.

VII. RELATED WORK

Over the years, a number of distributed resource management strategies have been proposed. However the sharing of services and the transitivity of their invocations seems to have been overlooked. For example, the Globus Architecture

for Reservation and Allocation (GARA) [17] addresses large-scale reservations on heterogeneous resources by constructing application-level co-reservations for collections of resources. It defines an architecture and a set of instructions to handle the reservation procedure but there is no service partitioning.

Another case is that of Q-RAM (QoS-based Resource allocation Model) [18], a framework for global management of distributed heterogeneous resources developed in the late 90s that maximizes an utility parameter of the tasks while meeting the resources constraints. This framework, further improved in [19] for systems with rapidly changing resources usage, provides a sub-optimal distribution of the resources capacities among the tasks that compose the system but again, it does not address service sharing.

Recent works in this direction include Distributed RK [7] and FRESCOR [6]. The former presents an extension of the Linux/RK real-time kernel to handle distributed resources consistently across the distributed system. This framework supports workflows described as directed acyclic graphs of subtasks but the proposed version supports CPU reserves, only, despite accounting for communication latencies when verifying the end-to-end deadlines. FRESCOR, on the other hand, provides an integrated resource management framework through contracts established and enforced on each resource. These contracts are called Virtual Resources (Vres), they can refer to processors, networks, memory, shared resources, disk bandwidth, and energy, and they are managed by a distributed application called Distributed Transaction Manager (DTM) [20].

VIII. Conclusion

In this paper we presented D-RES, a resouce management mechanism that allows the partitioning of services with the guarantee of temporal isolation. The core idea of D-RES is partitioning the resource usage of a service between its client applications' resource reservations, and performing correct client tracking even in case of nested service invocations. We presented an initial step to the response-time analysis for D-RES, based on its components. Future work we will carry out an extensive validation of the analysis. Finally, we demonstrated the usefulness and necessity of D-RES in practice by implementing a fully functioning D-RES system and analyzing the performance of a distributed real-time application based on the control and entertainment systems in a modern car.

Acknowledgments

This research was supported in part by projects NSERC DG 357121-2008, ORF-RE03-045, ORF-RE04-036, ORF-RE04-039, APCPJ 386797-09, CFI 20314, CMC Microsystems, and the industrial partners associated.

References

[1] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource Kernels: a Resource-Centric Approach to Real-Time and Multimedia Systems," in *Readings in multimedia computing and networking*, K. Jeffay and H. Zhang, Eds., 2001, pp. 476–490.

[2] E. Bini, G. Buttazzo, J. Eker, S. Schorr, R. Guerra, G. Fohler, K.-E. Arzen, V. Romero, and C. Scordino, "Resource Management on Multicore Systems: The ACTORS Approach," *Micro, IEEE*, vol. 31, no. 3, pp. 72–81, may-june 2011.

[3] A. Eswaran and R. Rajkumar, "Energy-Aware Memory Firewalling for QoS-Sensitive Applications," in *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, July 2005, pp. 11–20.

[4] S. Saewong and R. Rajkumar, "Cooperative Scheduling of Multiple Resources," in *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, 1999, pp. 90–101.

[5] R. Santos, M. Behnam, T. Nolte, P. Pedreiras, and L. Almeida, "Multi-Level Hierarchical Scheduling in Ethernet Switches," in *Proceedings of the ninth ACM international conference on Embedded software (EMSOFT'11)*, 2011, pp. 185–194.

[6] FRESCOR, "FRESCOR: Framework for Real-time Embedded Systems based on COntRacts," http://www.frescor.org/, accessed Oct. 2012.

[7] K. Lakshmanan and R. Rajkumar, "Distributed Resource Kernels: OS Support for End-To-End Resource Isolation," in *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE*, april 2008, pp. 195–204.

[8] "iLAND project (middleware for deterministic dynamically reconfigurable networked embedded systems)," http://en.wikipedia.org/wiki/ILAND_project, accessed Apr. 2014.

[9] D. Dodge, A. Danko, S. Marineau-Mes, P. V. D. Veen, C. Burgess, T. Fletcher, and B. Stecher, "Process scheduler employing adaptive partitioning of process threads," US Patent US 2007/0 226 739 A1, 2005.

[10] R. Marau, P. Pedreiras, and L. Almeida, "Asynchronous Traffic Signaling over Master-Slave Switched Ethernet protocols," in *Proc. on the 6th Int. Workshop on Real Time Networks (RTN'07)*, Pisa, Italy, Jul. 2007.

[11] A. B. de Oliveira, A. Saif Ur Rehman, and S. Fischmeister, "mTags: Augmenting Microkernel Messages with Lightweight Metadata," *SIGOPS Oper. Syst. Rev.*, vol. 46, no. 2, pp. 67–79, Jul. 2012.

[12] R. Marau, L. Almeida, and P. Pedreiras, "Enhancing Real-Time Communication over COTS Ethernet Switches," in *Factory Communication Systems, 2006 IEEE International Workshop on*, 2006, pp. 295–302.

[13] Z. Iqbal, L. Almeida, R. Marau, M. Behnam, and T. Nolte, "Implementing Hierarchical Scheduling on COTS Ethernet Switches Using a Master/Slave Approach," in *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, June 2012.

[14] M. Korsgaard and S. Hendseth, "Design patterns for communicating systems with deadline propagation," in *CPA*, 2009, pp. 349–361.

[15] L. Almeida and P. Pedreiras, "Scheduling within temporal partitions: response-time analysis and server design," in *Proceedings of the 4th ACM international conference on Embedded software*, 2004, pp. 95–103.

[16] A. B. de Oliveira, A. Azim, S. Fischmeister, R. Marau, and L. Almeida, "Adaptive worst case response-time analysis," https://bitbucket.org/aazim/aps/downloads, accessed July. 29th, 2014.

[17] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy, "A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation," in *Quality of Service, 1999. IWQoS '99. 1999 Seventh International Workshop on*, 1999, pp. 27–36.

[18] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek, "A Resource Allocation Model for QoS Management," in *Proceedings of the 18th IEEE Real-Time Systems Symposium*, dec 1997, pp. 298 –307.

[19] S. Ghosh, J. Hansen, R. Rajkumar, and J. Lehoczky, "Integrated Resource Management and Scheduling with Multi-Resource Constraints," in *Proceedings of the 25th Real-Time Systems Symposium*, 2004, pp. 12 – 22.

[20] M. G. Harbour, D. Sangorrn, and M. Sojka, "Distributed Transaction Manager - Proof of Concepts," Universidad de Cantabria, Tech. Rep. FP6/2005/IST/5-034026, Dec. 2007.