Bandwidth Isolation for Composability in Fixed Priority Real-Time Networks

Daniel Sangorrín
Nagoya University, Japan
dsl@ertl.jp

Michael González Harbour
University of Cantabria, Spain
mgh@unican.es

Abstract

The increasing complexity of real-time systems has motivated the application of component-based software engineering principles during the last few years. Temporal encapsulation is key to smoothing the integration stage of software components in complex distributed hard real-time systems. This paper presents a network scheduling server algorithm to guarantee and at the same time limit the network bandwidth assigned to streams of messages with different real-time requirements in a fixed priority network. The algorithm is based on a recently corrected version of the POSIX sporadic server whose rules, originally intended for scheduling tasks, have been adapted and optimized for the special case of fixed-priority networks. The algorithm is able to provide bounded response times that can be analyzed with off-the-shelf real-time analysis tools and can be used for both synchronous and asynchronous messages. The proposed approach has been implemented and evaluated on real hardware, using the CAN bus. The performance evaluation results show that bandwidth isolation can be achieved with rather low overhead both on the processor and the network resources.

1. Introduction

The complexity of developing large real-time applications can be handled by independently developing components that are later integrated into a physical platform. The success of the integration depends on the ability of the platform to provide the required resource usage guarantees to every component while protecting each component from timing faults in the others. Some compositional frameworks [5] have an integrated view of the different resources involved in a distributed application. In particular, for network resources, application components are able to specify their bandwidth requirements, so that the implementation can make the corresponding guarantees or reservations.

In real-time distributed systems, the communication paradigm (event- or time-triggered [10, 23]) plays an important role in the composability, flexibility and responsiveness of the system. In the *time-triggered* paradigm, messages are sent at predefined time windows according to a global schedule. This approach is well-suited for periodic activities that require very low jitter. Furthermore, it enables composability regarding to the temporal behavior because the access to the bus is predefined and decoupled from the actual network load. The major drawbacks of this approach are the lack of flexibility in the design process, the need of global synchronization between the nodes and its poor support for aperiodic messages. In the *event-triggered* paradigm, messages are sent as a response to the occurrence of an event. This approach is generally more flexible and better suited to support asynchronous traffic together with critical activities that require very short response times. The major disadvantages of this approach are the increased message jitter and the lack of temporal isolation.

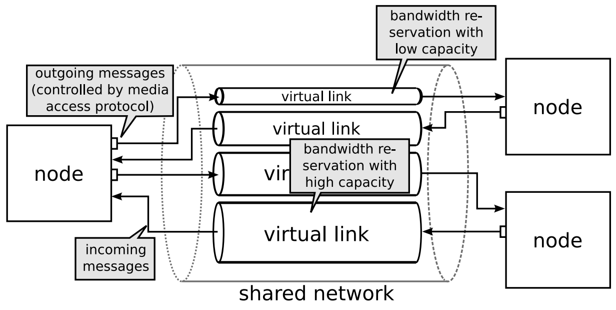

bandwidth reservation with low capacity
outgoing messages (controlled by media access protocol)
virtual link
virtual link
node
node
bandwidth reservation with high capacity
virtual link
node
incoming messages
shared network

Figure 1: Bandwidth reservations

Since both paradigms have strong and weak points, several protocols that combine support for both event- and time-triggered traffic have been proposed. In some protocols (e.g., FlexRay [3] or FTT-CAN [15]), temporal isolation between both types of traffic is enforced by implementing a cyclic sequence that alternates between them. However, the arbitration of the event-triggered phase of these protocols, implemented with different approaches such as fixed priorities (FTT-CAN) or TDMA with minislots (FlexRay), has the same drawbacks that were mentioned before, high jitter and lack of temporal isolation. For instance, in case of a software babbling idiot failure [13], a misbehaving task may affect the bandwidth preallocated to other tasks that are working correctly by transmitting excessive messages at a higher priority.

This paper presents a network scheduling algorithm that follows the event-triggered paradigm and is able to satisfy the requirements for the integration of independently developed components. The algorithm is based on a recently corrected version [25] of the POSIX sporadic server whose rules, originally intended for scheduling tasks, have been adapted and optimized for the special case of fixed-priority networks. The algorithm is able to control the jitter caused by aperiodic messages, provide bounded response times that can be analyzed with off-the-shelf real-time analysis tools and it can be used for both synchronous and asynchronous messages. It enables the creation of bandwidth reservations, which can be thought as unidirectional virtual links between two nodes providing a guaranteed service, as shown Fig. 1.

The paper is organized as follows. After an introduction to server-based scheduling for networks in Sec. 2, Sec. 3 proposes an algorithm for an optimized version of the sporadic server policy for fixed-priority networks. Sec. 4 gives details about the implementation of the algorithm on real hardware, whose overhead is evaluated in Sec. 5. Sec. 6 compares the proposal with previous work and Sec. 7 closes the paper with conclusions.

2. Server-based scheduling in networks

Server-based scheduling techniques have been used for a long time to limit the processor time assigned to a particular computation or set of computations while also guaranteeing some minimum level of service. Servers such as the periodic server [14], the sporadic server [24], or the constant bandwidth server [9] are a few examples.

The concept of server is also applicable to the outgoing direction of a network stack to limit the bandwidth used by message streams. However, scheduling in the networks is somehow different than in the processors. When a server is used to schedule a network the concept of execution time must be mapped into transmission time. In most networks, messages are fragmented in units called packets which are usually non preemptible. Therefore, an easy way to specify budgets in a network server is to measure them in terms of number of packets. The maximum packet size is usually limited by the network, but a smaller limit can also be imposed by the implementation as necessary, for instance as an application-defined parameter. Of course, if the message stream mixes very short messages with longer messages that fit into the maximum packet size, the bandwidth available to the message stream may be suboptimal, since each message consumes one unit of budget regardless of its size. However, it is easy to design a solution to this problem by creating several sporadic servers with different maximum packet sizes, and submitting the messages to the appropriate server based on their size. The non-preemptability of the network packets has bounded delay effects that can be easily modeled through a blocking time term. Other more complex analysis models can also be used to better estimate response times [12].

2.1. Sporadic server

The sporadic server is a bandwidth preserving scheduling algorithm designed for processing aperiodic events in hard real-time systems [24, 18]. It allocates a specific bandwidth for processing aperiodic requests at a given priority level (the normal priority). This bandwidth is provided by allocating a certain execution time capacity for each interval of time called the replenishment period. The scheduling algorithm is defined through a set of rules for consuming this execution capacity when the sporadic server runs, and for later replenishing this capacity. When the capacity is consumed, the sporadic server may still do useful work at a background priority level, to make full use of the resource.

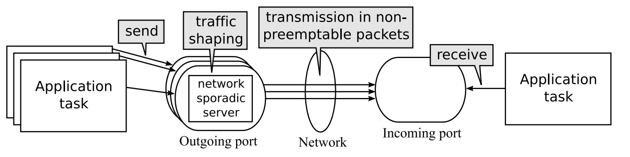


Figure 2: Communication elements

When the sporadic server was standardized in the additional real-time extensions [6] of the POSIX standard for portable operating system interfaces (later included in the unified version of the standard [7]), it was defined with a set of consumption and replenishment rules, intended to allow for a feasible implementation in the context of a real-time operating system (RTOS). Unfortunately, except for some specific cases, the new rules had the same problem of the original sporadic server definition that could cause preemptions to occur too early [18]. Recently, in [25], a new set of rules has been proposed in order to fix the original POSIX sporadic server problem while maintaining its main value, the simplicity of its implementation.

In the next section, the sporadic server proposed in [25] is adapted and optimized for the case of fixed priority networks. To simplify the implementation, the presented approach takes advantage of the discrete nature of the network packets and considers that the capacity chunks used in the sporadic server are always of size one. This allows to create a capacity queue of fixed size, equal to the number of packets represented in the budget of the sporadic server. Each packet in the capacity queue is annotated with its replenishment time. This simplifies the budget arithmetics and eliminates the need to introduce optimizations to limit the fragmentation of the capacity.

3. Network Sporadic Server algorithm

The proposed network sporadic server policy is based primarily on two parameters: the replenishment period and the initial transmission capacity. The replenishment period is called *repl_period* and is measured as an absolute time. The initial transmission capacity is called the *init_budget* and is an integer number of network packets of bounded size. As shown in Fig. 2, the network sporadic server policy is used to schedule a stream of messages that are sent from a specific sender node in the system, through the network. The destination node of these messages is any node that is reachable in a single hop. Messages to be sent are submitted by the application and stored in a transmission queue until they are sent. Messages in this queue fit into one packet, but a fragmentation layer is provided outside the sporadic server implementation if larger messages are required.

Fig. 3 shows the architecture of the network sporadic servers. For each sporadic server the system maintains in the sender node a capacity queue, with transmission capacity chunks. The size of the queue is equal to *init_budget*. Each chunk represents a transmission capacity of one

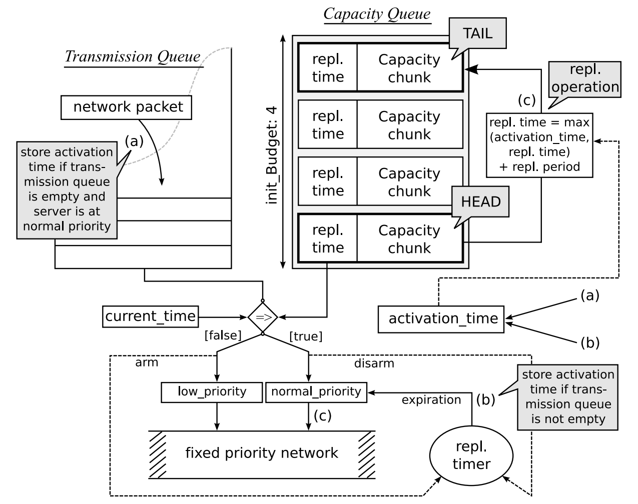
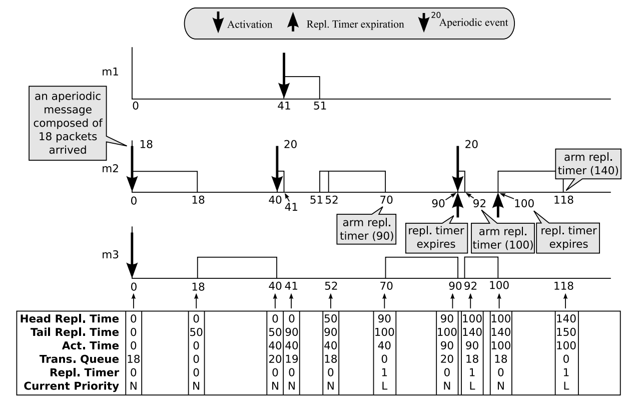


Figure 3: Network Sporadic Server

packet, and contains a replenishment time, which is an absolute time after which the capacity may be consumed. Initially, all the chunks in the queue have a replenishment time equal to the time at which the queue is initialized. In addition, the system keeps one value associated with each sporadic server: an absolute time called the *activation_time*. Finally, the system has a conceptual replenishment timer associated with each sporadic server.

The priority assigned to messages sent though a sporadic server is determined in the following manner: if the replenishment time of the head of the capacity queue is equal to or earlier than the current time, the server is considered to have execution capacity available, so it is assigned the priority specified by *normal_priority*, and its replenishment timer is disarmed; otherwise, the assigned priority shall be *low_priority*, and the replenishment timer is armed to expire at the replenishment time of the head of the capacity queue. The modification of the capacity queue and, consequently of the assigned priority, is done as follows:

1. Each time the server is made ready at the *normal_priority* level, either because a new message arrived at the transmission queue while it was empty (path (a) in Fig. 3) or because the replenishment timer expired and the transmission queue is not empty (path (b)), the time at which this operation is done is stored in the *activation_time*
2. When a message is sent at the *normal_priority* level a replenishment operation is performed (path (c)), as described in 3. Then, if the replenishment time of the new head of the capacity queue is larger than the current time, the server is assigned the *low_priority* and the replenishment timer is armed to expire at the replenishment time of the head of the capacity queue.
3. Each time a replenishment operation is performed the head of the capacity queue is removed from the queue and reinserted at the tail with a replenishment time equal to the maximum of the *activation_time* and its current replenishment time, plus *repl_period* (see path (c) in Fig. 3).



Head Repl. Time	0	0	0	0	50	90	90	100	100	140
Tail Repl. Time	0	50	50	90	90	100	100	140	140	150
Act. Time	0	0	40	40	40	40	90	90	100	100
Trans. Queue	18	0	20	19	18	0	20	18	18	0
Repl. Timer	0	0	0	0	0	1	0	1	0	1
Current Priority	N	N	N	N	N	L	N	L	N	L

Figure 4: Scheduling sequence using the network sporadic server

4. When the replenishment timer expires the server is assigned the *normal_priority* level.

3.1. Example

The following example illustrates the presented network sporadic server algorithm. Consider a system with three periodic message streams with parameters shown in Table 1, with deadline-monotonic priority ordering. Suppose that m_2 is a network sporadic server to transmit aperiodic messages. The sporadic server is given a initial transmission capacity of $C_2 = 20$ packets, and a replenishment period $T_2 = 50$ time units. For simplicity, the transmission time of one packet is supposed to take one time unit.

Table 1: Periodic message streams

Message	C_i	T_i	D_i
m_1	10	200	20
m_2	20	50	50
m_3	50	200	100

Fig. 4 shows a transmission sequence scheduled under the network sporadic server policy defined in this paper. It also shows the evolution of the replenished time value in the head and tail of the capacity queue, the *activation_time* variable associated to the server, the number of packets in the transmission queue, the current priority of the server, normal (N) or low (L) and the status of the replenishment timer which can be armed (1) or disarmed (0). The example is similar to the one used in [25] to illustrate the correction of the premature replenishments defect in the original POSIX sporadic server.

4. Implementation

The network sporadic server policy defined in Sec. 3 has been implemented on real hardware. As a relevant example of fixed priority networks, the Controller Area Network (CAN) [2] was chosen. CAN has been used extensively in the automotive industry to connect Electronic Control Units (ECUs) using a shared bus. CAN features non-preemptive frame transmission and priority-based arbitration through

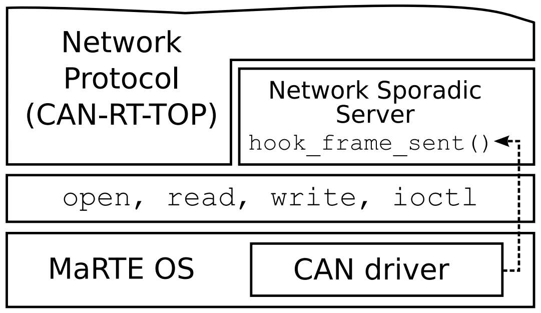


Figure 5: Implementation layers

a bit dominance protocol which enables bounded latencies that can be analyzed through real-time schedulability theory. Popularity of CAN has reached other sectors such as industrial control applications or medical equipment.

Fig. 5 shows the main elements of the implementation. The architecture was implemented on MaRTE OS [11], a hard real-time operating system that follows the Minimal Real-Time POSIX.13 subset and provides an easy-to-use and controlled environment to develope multi-thread real-time applications. The core of MaRTE OS is written in Ada language, and it supports mixed-language applications in Ada, C and C++. In this study, MaRTE OS was extended with a driver that supports the NXP SJA1000 chipset [8], a stand-alone controller for CAN commonly used within automotive and general industrial environments. The driver provides a POSIX character interface (i.e., `open`, `read`, `write`, etc.). In addition, several hooks can be installed inside the driver through the `ioctl` system call.

The implementation of the network sporadic server executes on top of the MaRTE OS interface. The main sources of overhead introduced by the network sporadic servers, compared to using the native CAN protocol, are the following:

- The replenishment operations (see path (c) in Fig. 3) which are executed every time a CAN frame is sent. A hook is installed in the CAN bus driver, through the `ioctl` system call, in order to be notified about the transmission of a CAN frame.
- A replenishment thread, which waits for expirations of the replenishment timers, modifies the priority of the server and updates the activation time.

In addition, as shown in Fig. 5, a previously presented high-level protocol for CAN (CAN-RT-TOP [19]) was adapted to send messages through the developed network sporadic server. Details about the adaptation of the protocol and its source code, distributed under the GNU/GPL v2 license, can be obtained at [4].

5. Evaluation

This section presents evaluation results of the network sporadic server presented in this paper. The evaluation environment consisted of nodes equipped with AMD Duron 800 Mhz processors, 256 MB RAM memory and Adlink PCI-7841 CAN bus cards [1], which are based on the NXP SJA1000 controller. The CAN bus was configured to 1Mbps and CAN 2.0B mode. MaRTE OS version 1.8 was built, using the AdaCore GNAT/GPL 2007 (gcc 4.1.3) compiler, with default options which disable assertions, inlines code and perform some optimizations for targets with Local APIC.

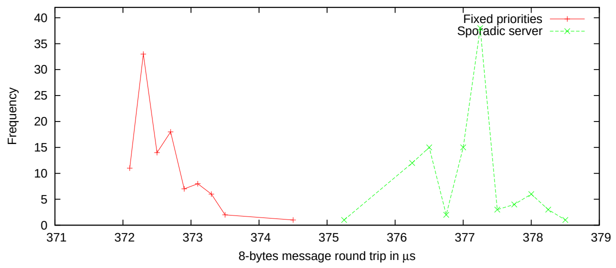


Figure 6: 8-bytes message round-trip measures

5.1. Message round-trip latency measures

In order to measure the influence of the network sporadic server on the end-to-end latency of the message streams, two nodes were connected through the CAN bus and programmed to send query-reply messages continuously under two scenarios: using fixed priorities and using the proposed network sporadic server. Each measure, defined as a round-trip measure, was taken from the instant when the message was sent and the moment when the reply was received. Measures were repeated for 100 times for different message sizes.

Table 2: Maximum round-trip measured values

Bytes	Fixed Priorities	Sporadic Server
8	0.375 ms	0.379 ms
32	1.355 ms	1.372 ms
64	2.643 ms	2.673 ms
512	20.78 ms	21 ms
1488	60.37 ms	61.03 ms

Fig. 6 shows the comparison of the measured values when using 8-bytes messages, which fit in the maximum size of a CAN frame and therefore do not require fragmentation. Table 2 shows the comparison of the maximum measured values for several message sizes. The overhead introduced by the network sporadic server is rather small compared to the transmission times.

5.2. Overhead in the CPU

Table 3 shows execution-time measures of the main sources of processor overhead caused by the sporadic server policy. The first row represents the overhead associated to a replenishment operation. The second row represents the execution time of the body of the replenishment thread which is executed on every replenishment timer expiration.

In order to better evaluate the influence that the measured values, shown in Table 3, represent on the total CPU overhead, simulations of the network sporadic server exe-

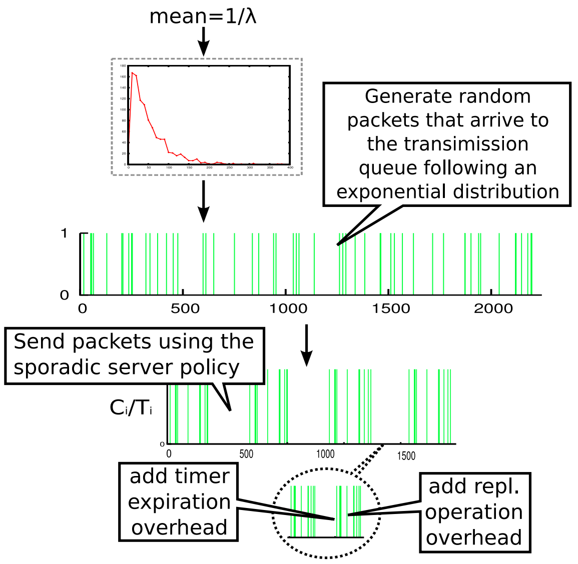

mean=1/λ
Generate random packets that arrive to the transimission queue following an exponential distribution
Send packets using the sporadic server policy
Cᵢ/Tᵢ
add timer expiration overhead
add repl. operation overhead

Figure 7: Simulation procedure for the estimation of the CPU overhead under different configurations

Table 3: Sporadic servers measured CPU execution time (in μs)

Measure	Min	Avg	Max
Repl. Program	0.78	0.81	2.34
Repl. Thread	2.69	2.88	3.52

cution have been performed under different configurations. Fig. 7 depicts the procedure followed during the simulations. First, 1000 random aperiodic events (packets arriving to the transmission queue) are generated according to an exponential distribution. Then, packets are sent using the presented sporadic server policy. Each time a replenishment operation or a timer expiration occurs, the corresponding CPU overhead is accounted (maximum measured overhead values, 2.34μs and 3.52μs, were used). When all packets are sent, the total overhead time is divided by the total time to get the overhead as a percentage. The sporadic server was configured with an utilization equal to the mean of the packet inter-arrival instants. It has the highest priority in the network (to evaluate its performance in isolation) and it never transmits at low_priority (i.e., because there are always lower priority messages being transmitted). For simplicity, each packet is supposed to occupy the bus for a constant time of 1 ms. Simulations were repeated for different inter-arrival rates and different server budget/period configurations.

Table 4 contains the overhead results for several inter-arrival rates (defined by $1/\lambda$) of aperiodic events. The overhead is rather small and can be decreased even more by configuring the sporadic server appropriately. Fig. 8 separates the overhead caused by replenishment operations from the timer expirations. Replenishment operations cause a constant overhead since they appear each time a packet is sent. On the other hand, timer expirations overhead can be reduced considerably by increasing the capacity of the server.

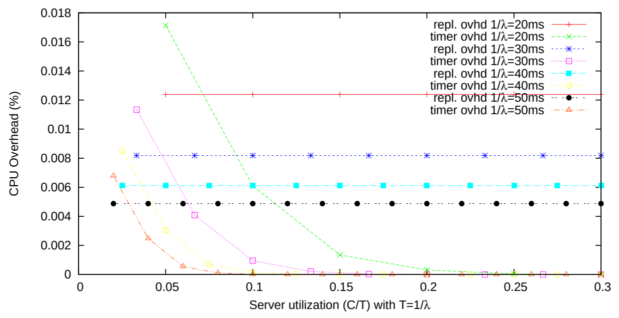


Figure 8: replenishments overhead vs. timer overhead

Table 4: Simulated sporadic server CPU overhead (in %) with parameters $Budget = n$ packets and $Period = n \cdot \frac{1}{\lambda}$

$1/\lambda$	$n = 2$	$n = 4$	$n = 6$	$n = 8$	$n = 10$
20ms	0.0286	0.0284	0.0275	0.0269	0.0266
50ms	0.0114	0.0112	0.0111	0.0110	0.0109
250ms	0.0023	0.0023	0.0022	0.0022	0.0022
1000ms	0.0005	0.0005	0.0005	0.0005	0.0005

6. Related work

The leaky bucket concept used in network traffic shaping [17, 16] is similar to the concept of server-based scheduling. The leaky bucket algorithm is useful to control that the traffic is sent to the network at a constant rate. However, it does not handle efficiently the available bandwidth since the leak rate is a fixed parameter and, there may be instances when the network is unused while there are packets pending to be sent. The benefits of the network sporadic server when compared to the leaky bucket are a higher capacity, a shorter response time and minimal interference on lower priority tasks, because the available execution capacity is usable without delay at the specified priority level, and because the effects on lower priority tasks are no worse than those of an equivalent periodic task with an execution time equal to the execution capacity, and period equal to the replenishment period.

In [21], server-based mechanisms based on dynamic priorities (EDF) were proposed for scheduling the CAN bus [2]. The algorithms proposed in that work are based on a master-slave architecture where nodes are synchronized to a trigger message sent periodically by the master. Although the use of dynamic priorities may allow optimal resource utilization, the overhead generated by the necessary synchronization messages and the scheduling algorithm must be taken into account. The benefits of the network sporadic server when compared to that work, are the ability to provide faster response times while minimizing the overhead and the fact that it does not require a complicated implementation.

In addition, the network sporadic server can be integrated with previously presented protocols that organize the bus time as a sequence of time- and event-triggered windows. Fig. 9 depicts an FTT-CAN [15] cycle, divided into synchronous and asynchronous windows. The use of net-

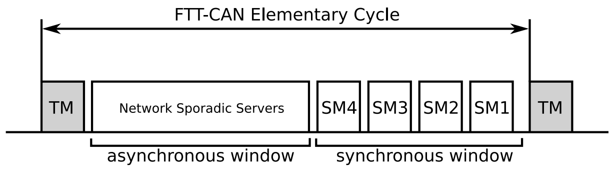


Figure 9: Network sporadic server integrated with FTT-CAN

work sporadic servers in the asynchronous window makes it possible to provide bandwidth isolation between aperiodic message streams. For instance, if a software babbling idiot failure [13] occurs, the messages transmitted by the misbehaving task would be shaped by the server and would not affect the deadlines of other message streams with lower priority.

7. Conclusions

The sporadic server is a very interesting scheduling policy for handling resource reservations, which are key to smoothing the integration stage of software components in complex distributed hard real-time systems. This paper described how to adapt a recently corrected version of the POSIX sporadic server, originally intended for scheduling tasks, to the case of fixed-priority networks. The algorithm was optimized to take into account the discrete nature of the network packets. An implementation on the CAN bus was also described together with its evaluation. The measured performance shows that bandwidth isolation can be achieved at rather low overhead both on the processor and the network resources. The algorithm can be applied to other networks where message streams compete for the media access through fixed priorities. For example, a porting exists to provide reservations on the RTEP protocol [20]. The work presented in this paper has been used in [22, 5] to implement contract-based network bandwidth reservations in the context of a flexible scheduling framework.

References

[1] ADLINK Website. `http://www.adlinktech.com/`.

[2] CAN Specification Version 2.0. 1991, Robert Bosch GmbH, Postfatch 30 02 40, D-70442 Stuttgart.

[3] FlexRay Consortium. `http://www.flexray.com/`.

[4] FRESCOR Fieldbus Systems (D-ND1). `http://www.frescor.org/index.php?page=publications`.

[5] FRESCOR Website. `http://www.frescor.org`.

[6] IEEE Std. 1003.d-1999. Information Technology -Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) Amendment: Additional Realtime Extensions [C Language]. The Institute of Electrical and Electronics Engineers.

[7] ISO/IEC 9945-1:2003. Standard for Information Technology -Portable Operating System Interface (POSIX).

[8] NXP Website. `http://www.nxp.com`.

[9] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *RTSS '98: Proceedings of the IEEE Real-Time Systems Symposium*, pages 4–13, Washington DC, USA, 1998. IEEE Computer Society.

[10] A. Albert. Comparison of Event-Triggered and Time-Triggered Concepts with Regard to Distributed Control Systems. In *Proceedings of Robert Bosch GmbH Embedded World*, pages 235–252, Nuremberg, February 2004. `http://www.semiconductors.bosch.de/pdf/embedded_world_04_albert.pdf`.

[11] M. Aldea and M. González Harbour. MaRTE OS: An Ada Kernel for Real-Time Embedded Applications. In *Proceedings of the International Conference on Reliable Software Technologies*, Ada-Europe-2001, Leuven, Belgium, May 2001. Lecture Notes in Computer Science. `http://marte.unican.es`.

[12] R. J. Bril, J. J. Lukkien, and W. F. J. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited. In *ECRTS '07: Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 269–279, Washington, DC, USA, 2007. IEEE Computer Society.

[13] I. Broster and A. Burns. The Babbling Idiot in Event-triggered Real-time Systems. In *Proceedings of the Work-In-Progress Session, 22nd IEEE Real-Time Systems Symposium*, pages 25–28, 2001.

[14] G. C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 2002.

[15] J. Ferreira, P. Pedreiras, L. Almeida, and J. A. Fonseca. The FTT-CAN Protocol for Flexibility in Safety-Critical Systems. *IEEE Micro*, 22:46–55, 2002.

[16] E. Hernández and J. Vila. A new approach to optimize bandwidth reservation for real-time video transmission with deterministic guarantees. *Real-Time Imaging*, 9(1):11–26, 2003.

[17] C. F. John Evans. Deploying IP and MPLS QoS for Multiservice Networks: Theory and Practice. Morgan Kaufmann Publishers, 2007.

[18] J. Liu. Real-Time Systems. Prentice Hall, 2000.

[19] J. López Campos, J. J. Gutiérrez, and M. González Harbour. CAN-RT-TOP: Real-Time Task-Oriented Protocol over CAN for Analyzable Distributed Applications. In *Proceedings of the 3rd International Workshop on Real-Time Networks (formerly RTLIA)*, Catania, Sicily (Italy), 2004.

[20] J. M. Martínez and M. González Harbour. RT-EP: A Fixed-Priority Real Time Communication Protocol over Standard Ethernet. In *10th International Conference on Reliable Software Technologies, Ada-Europe*, pages 180–195. Springer, June 2005.

[21] T. Nolte, M. Nolin, and H. Hansson. Real-Time Server-Based Communication for CAN. *IEEE Transactions on Industrial Informatics*, 1(3):192–201, August 2005.

[22] D. Sangorrín, M. González Harbour, H. Pérez, and J. Javier Gutiérrez. Managing Transactions in Flexible Distributed Real-Time Systems. In *Proceedings of the 15th International Conference on Reliable Software Technologies*, Ada-Europe 2010, Valencia, Spain, June 2010.

[23] J. Scarlett and R. Brennan. Re-evaluating Event-Triggered and Time-Triggered Systems. In *11th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA '06*, pages 655–661, Sept. 2006.

[24] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic Task Scheduling for Hard-Real-Time Systems. In *The Journal of Real-Time Systems 1(1)*, pages 27–60. Kluwer Academic Publishers, 1989.

[25] M. Stanovich, T. P. Baker, and M. González Harbour. Defects of the POSIX Sporadic Server and How to Correct Them. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2010*, Stockholm, Sweden, April 2010.