# Replica Determinism and Flexible Scheduling in Hard Real-Time Dependable Systems

Stefan Poledna, Alan Burns, *Member*, *IEEE*, Andy Wellings, and Peter Barrett

**Abstract**—Fault-tolerant real-time systems are typically based on active replication where replicated entities are required to deliver their outputs in an identical order within a given time interval. Distributed scheduling of replicated tasks, however, violates this requirement if on-line scheduling, preemptive scheduling, or scheduling of dissimilar replicated task sets is employed. This problem of inconsistent task outputs has been solved previously by coordinating the decisions of the local schedulers such that replicated tasks are executed in an identical order. Global coordination results either in an extremely high communication effort to agree on each schedule decision or in an overly restrictive execution model where on-line scheduling, arbitrary preemptions, and nonidentically replicated task sets are not allowed. To overcome these restrictions, a new method, called *timed messages*, is introduced. Timed messages guarantee deterministic operation by presenting consistent message versions to the replicated tasks. This approach is based on simulated common knowledge and a sparse time base. Timed messages are very effective since they neither require communication between the local scheduler nor do they restrict usage of on-line flexible scheduling, preemptions and nonidentically replicated task sets.

**Index Terms**—Distributed real-time systems, fault tolerance, distributed operating systems, replica determinism, distributed scheduling, flexible scheduling.

## 1 Introduction

Distributed fault-tolerant real-time systems are typically based on active replication, i.e., critical components in the system are replicated and perform their services in parallel. This replication of components may take place either at the hardware or the software level. At the hardware level, complete processors are replicated in conjunction with the software they are executing. Replication at the software level allows for finer granularity. It is possible to replicate only critical tasks on different processors, while noncritical tasks can be executed without replication. Replication at the software level, therefore, has the advantage of better resource efficiency. It is thus not necessary to replicate a complete processor when only a small fraction of tasks are critical. This leads to a system configuration where processors execute dissimilar sets of tasks (replicated and nonreplicated ones). There are many application areas for distributed fault-tolerant real-time systems where this type of resource efficiency—which, in turn, translates to low system cost—is of utmost importance. One such example is the area of automotive electronics with its high product volumes. However, replication of software tasks is often not cost effective due to the high communication effort required to coordinate the individual replicated tasks in a loosely coupled distributed system.

Safety critical software has traditionally been implemented using a cyclic executive. This uses a regular interrupt to access a table in which sequences of procedure calls are identified. By cycling through the table, periodic jobs are supported. Although conceptually simple, the cyclic executive approach suffers from a number of drawbacks [18]; it uses resources inefficiently, it does not easily cater to sporadic jobs, and it gives poor support to more flexible and adaptive requirements. A more appropriate scheme, using priority-based scheduling of application tasks [4], is currently been used (or evaluated) in a number of application domains, for example, in avionics [2] or automotive electronics [24]. Both preemptive and non-preemptive (cooperative) methods are under consideration.

With cyclic executives, it is relatively easy to coordinate the executions of replicated jobs. With more flexible scheduling, where not all tasks are replicated, this coordination is problematic. In this paper, we propose a scheme for this coordination. The work presented is based on earlier independent treaties by Poledna [25], [26] and Barrett et al. [3]. The paper is organized as follows: In the following section, the issue of replica determinism is introduced. A system model is given in Section 3. Section 4 relates the formal concept of common knowledge in distributed computer systems to the problem of deterministic distributed scheduling. Timed messages are introduced in Section 5. It is shown that timed messages provide a very efficient and flexible means to achieve replica determinism in the presence of on-line scheduling, preemptions, and nonidentically replicated task sets. Finally, Section 6 concludes the paper.

- *S. Poledna is with the Institute for Technical Computer Science, Technical University of Vienna, Vienna A-1040, Austria.
E-mail: stefan@vmars.tuwien.ac.at.*
- *A. Burns and A. Wellings are with the Department of Computer Science, University of York, Heslington, York Y010 5DD, UK.
E-mail: {burns, andy}@cs.york.ac.uk.*
- *P. Barrett is with the Centre of Software Reliability, University of Newcastle upon Tyne, Newcastle NE1 7RU, UK.
E-mail: Peter.Barrett@cs.ncl.ac.uk.*

*Manuscript received 5 Oct. 1998; accepted 26 Feb. 1999.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 107497.*

0018-9340/00/$10.00 © 2000 IEEE

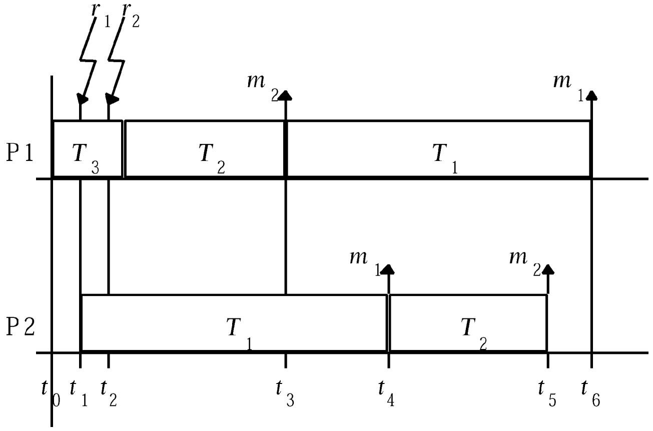

Fig. 1. Inconsistent order through non-identical replicated task sets.

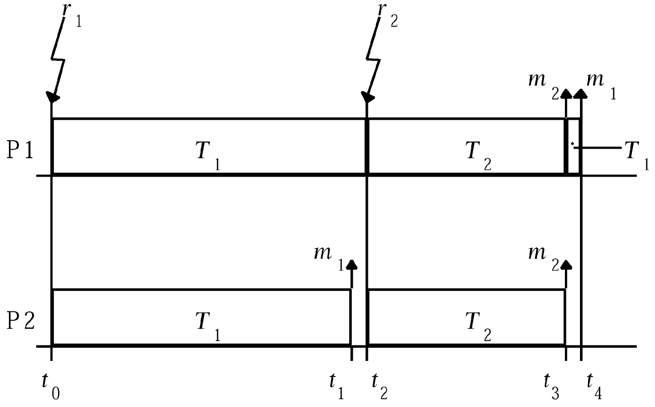

Fig. 2. Inconsistent order through preemptive scheduling.

## 2 Replica Determinism

With active replication, fault-free replicated components are required to exhibit *replica determinism* [32], [29], [28], i.e., they have to deliver identical outputs in an identical order within a specified time interval. If replica determinism is guaranteed, then it is relatively simple to achieve fault tolerance. Failures of components can be detected by carrying out a bit-by-bit comparison of the replicas' results.[1] While it might seem trivial at first that replicated components will produce replica deterministic outputs this is unfortunately not the case. Replica nondeterminism is introduced by 1) the interface to the real world and 2) the system's internal behavior. First, at the interface to the continuous real world, replicated sensors may return slightly different observation results due to their limited accuracy (e.g., 100° C and 99.99° C).[2] Second, system internal nondeterminism is caused by mechanisms such as on-line scheduling, preemptive scheduling or scheduling of nonidentically replicated task sets. This results in different execution order and timing of the replicated tasks, which, in turn, causes inconsistently ordered outputs.

The following two examples illustrate internal replica nondeterminism which results in inconsistent operation of replicated tasks. First, Fig. 1 shows nondeterminism introduced by dissimilar replicated task sets. We assume two identical processors P1 and P2. The tasks $T_1$ and $T_2$ are replicated on these two processors and must produce the same results in the same order. Processor P1 additionally executes task $T_3$ which implements a non-fault-tolerant service that is not replicated on processor P2. At time, $t_1$, task $T_1$ becomes ready. Since P2 is idle, $T_1$ is started immediately on processor P2. $T_1$'s activation is delayed on processor P1 which currently executes task $T_3$. Before task $T_3$ is finished, task $T_2$ becomes ready at time $t_2$. It is assumed that task scheduling is carried out cooperatively (nonpreemptive) and that $T_2$ has higher priority than task $T_1$. Therefore, processor P1 starts with task $T_2$ immediately after finishing $T_3$, while processor P2 has to wait until $T_1$ is finished. Therefore, processor P1 sends message $m_2$ before $m_1$, while P2 sends the messages in the opposite order. Note, in Fig. 1, $r_1$ and $r_2$ represent the event that released the task. They could be clock events, in which case, $T_1$ and $T_2$ are periodic tasks, or irregular events, in which case $T_1$ and $T_2$ are sporadic tasks.

The second example shows that replica nondeterminism may also occur if preemptive scheduling is used instead of cooperative scheduling, even if the task sets are replicated identically. Fig. 2 shows two identical processors P1 and P2. Both processors execute exclusively the replicated task set consisting of $T_1$ and $T_2$. Task $T_1$ starts its execution at time $t_0$ on both processors.

Due to minor differences in the clock speed,[3] $T_1$ finishes at processor P2 slightly earlier than on P1. Message $m_1$, the output of task $T_1$, is therefore sent at time $t_1$. Immediately after the output of $m_1$ on P2, the task activation request for $T_2$ arrives. Again, it is assumed that $T_2$ has higher priority. But, since preemptive scheduling is used, task $T_1$ gets preempted by task $T_2$ at processor P1 before it has sent its message. Task $T_2$ then runs to completion on both processors and both send message $m_2$ by time $t_3$. Finally, at time $t_4$, task $T_1$ finishes on P1 with its remaining instructions and outputs message $m_1$. Again, the order of outputs is inconsistent across both processors.

Both examples show that the requirement for an identical output order is violated. In a plant control system, for example, such a situation might lead to a case where one processor decides to shut down while the other decides to continue the process. Another example would be an alarm monitoring system where, due to different message orders, the replicated processors might evaluate different causes for an alarm. These situations all have in common an inability to resolve the nondeterministic decisions by a voting protocol without extensive application knowledge (which undermines a systematic and, thus, application independent approach to fault tolerance).

In addition to the inconsistent output order, there are large divergences in the timing of the outputs. This nondeterministic behavior is clearly unacceptable for fault-tolerant real-time systems since it would be difficult, if not impossible, to decide whether processors are

[1] This mechanism is based on the assumptions that individual components are affected by faults independently and that the possibility of correlated faults is sufficiently low.

[2] To guarantee replica determinism it is, therefore, necessary to achieve agreement on the replicated sensor observations by using appropriate communication protocols [19].

[3] Fault-tolerant computer systems typically use different clocks for the individual processors to avoid a common source of system failures.

returning inconsistent results due to a fault or due to their nondeterministic behavior.

### 2.1 Shortcomings of Replica Determinism Enforcement Strategies

As the above examples show, it is necessary to enforce replica deterministic behavior of replicated tasks. The related literature describes two different possibilities to ensure deterministic behavior of replicated tasks:

- *On-line agreement:*
  All relevant (preliminary) scheduling decisions are exchanged among the individual schedulers before they are actually taken. Based on the individual scheduling decisions, an agreement protocol is carried out. This guarantees that all schedulers take identical scheduling decisions with respect to the replicated tasks, e.g., [16], [30]. An alternative approach for obtaining on-line agreement is to execute group communication protocols such as those defined by ISIS [31].
- *Off-line agreement:*
  Second, all scheduling decisions are based on a global coordinated time base where the sequence of task activations has been determined completely off-line, e.g., [11]. Agreement on scheduling decisions with respect to the replicated tasks is guaranteed by the off-line scheduler.

The major disadvantage of the first approach is the extremely high communication effort and the latency introduced by the execution of an agreement protocol. For example, in the area of automotive electronics execution of an agreement protocol on the CAN [6] bus has a latency of at least $400\mu s$[4] for a replication degree of only two [27], [20]. A latency of $400\mu s$, however, is unacceptable since up to 10.000 *tasks* per second have to be scheduled in this application. The disadvantage of the second approach is the restrictive execution model, where the task schedule is determined completely off-line and nonidentically replicated task sets, as well as arbitrary preemption, is not allowed.

To overcome these restrictions, a new method is presented which ensures deterministic behavior of replicated tasks even if 1) preemptive scheduling, 2) dissimilar tasks sets, and 3) on-line scheduling is used. This methodology—called *timed messages*—is based on simulated common knowledge which exploits the a priori knowledge available in real-time systems. The basic idea is to present only consistent message versions to the replicated tasks, despite their possibly diverging execution order and timing. Timed messages require no extra communication between the processors and are, therefore, highly efficient.

## 3 System Model

The following assumptions about the distributed fault-tolerant real-time system are made: The system consists of a set of (not necessarily identical) processors denoted $\mathrm{P} = \{\mathrm{P}1, \mathrm{P}2, \ldots, \mathrm{P}p\}$. The processors are connected by means of a message passing facility; they do not share common memory. Each processor has a local clock which is approximately synchronized with the clocks of the other nonfaulty processors in the system. It is assumed that no two nonfaulty clocks differ by more than time $\varepsilon$. The set of tasks which is executed by these processors is denoted $T = \{T_1, T_2, \ldots, T_t\}$. A subset of these tasks is replicated and, thus, executed by more than one processor. Communication between tasks (within the same or different processors) is carried out by sending and receiving messages exclusively. The sending of messages is the only behavior of a task that is visible to the outside and other tasks. It is assumed that no internal operation of a task, other than reading a message, can lead to nondeterminism. The set of messages is denoted $M = \{m_1, m_2, \ldots, m_m\}$. Besides communication, tasks may also have arbitrary precedence and resource constraints.

The semantics of messages are very similar to that of global variables. First, the sending and receiving of a message is nonblocking (and may be buffered). Second, a message does not get consumed by a receive operation. It is, therefore, possible to receive a message more than once. A message reflects the last state of some entity which can be read by receiving a message or updated by sending a message. Between sender and receiver there is no synchronization relation with such messages. Rather, they can be used for an unsynchronized $1 : n$ or $m : n$ information exchange between processors. Messages are typically implemented by pools. The sender puts the recent message in the pool, while receivers get copies of this message from the pool.

Since we consider a multiprocessor system it is possible to have intra and interprocessor communication. For intraprocessor communication, it is assumed that message send and receive operations are atomic, i.e., the data transfer of send and receive operations is indivisible. For interprocessor communication, it is assumed that a reliable broad- or multicast protocol is available, e.g., [7], [8], [35]. This service guarantees that a message, once sent, is delivered to all correct participants within $\delta$ time units. Furthermore, to ensure determinate behavior of the replicated tasks it is necessary that they receive the same sequence of messages. The broad or multicast protocol is therefore required to support suitable order properties such as total and causal order [8], [34].

Each processor has a local scheduler which starts the execution of a ready task and suspends the execution of a running task. It is assumed that the scheduler is activated either by the passage of time (time-triggered activation) or by the arrival of relevant events (event-triggered activation). Thus, both periodic and sporadic task sets can be accommodated. Upon each activation, the scheduler decides whether the currently active task needs to be suspended and another task needs to be started. These scheduler activations are carried out in response to task activation requests which are denoted $r_1, r_2, \ldots, r_r$. The time of occurrence for an activation request $r_i$ is denoted $tr_i$. There are no assumptions on the scheduling strategy, whether scheduling is preemptive or cooperative, nor if scheduling

[4] This latency figure is based on the implementation described in [20]. It is assumed that the CAN is operated at its maximum communication rate of 1 Mbps.

is static or dynamic. The only two necessary assumptions are that 1) the scheduler is able to guarantee hard deadlines and that 2) replicated sets of tasks are started in response to scheduler activations with an agreed upon timestamp. This requires the availability of an approximately synchronized clock service[5] [17], [14]. If time-triggered activation requests are based on approximately synchronized clocks, then the requirement for an agreed timestamp is fulfilled implicitly. In the case of event-triggered activation requests for replicated sporadic tasks, it is necessary that the event carries an agreed upon timestamp with a reference to the approximately synchronized clock. This, in turn, requires that an agreement protocol has to be executed to get an agreed timestamp for these events (see Section 5.5).

There are no assumptions taken concerning fault hypotheses and failure semantics of the system. The reason for this is that timed messages are not a fault tolerance mechanism. They rather guarantee deterministic behavior of all replicated tasks which are not faulty. Timed messages are a building block to implement specific fault tolerance mechanisms. For example, the deterministic output of the replicated tasks can be combined by a consensus protocol to ensure Byzantine resiliency.

## 4 Deterministic Operation and Simulated Common Knowledge

This section gives a brief overview of the concept of common knowledge in distributed computer systems and relates this formal concept to the problem of deterministic operation in fault-tolerant distributed real-time systems. It will be shown that the theoretical results in this area can be used as a starting point for a highly efficient method to manage replicated tasks.

From a formal point of view, there is a correspondence between deterministic operation and common knowledge [10]. If all replicated tasks use common knowledge exclusively and their output becomes available within a specified time interval, then replica determinism can be guaranteed. In other words, it can be guaranteed that the results of replicated tasks are identical and that they are returned in identical order.

### 4.1 Common Knowledge and Simultaneity

A group of processors $P = \{\mathrm{P}1, \mathrm{P}2, \dots, \mathrm{P}p\}$ has *common knowledge* on a true fact[6] denoted $\varphi$ if all processors know that all processors know that all processor know ... that all processors know $\varphi$, where "all processors know" is repeated an infinite number of times. It has been shown that this state of knowledge cannot be attained in practical distributed computer systems (without shared memory) [9]. This observation is based on the impossibility of simultaneous actions which is caused by the finite accuracy of any "real" component.[7] This impossibility result can be translated to the problem of deterministic operation as follows: Since scheduling decisions cannot be carried out simultaneously, it is impossible for replicated tasks to return their results simultaneously. This negative conclusion on the possibility of common knowledge does not say that it is completely impossible to operate replicated tasks deterministically. It rather states that common knowledge cannot be attained and the requirement for simultaneity has to be dropped. Two alternatives have been studied to relax the semantics of common knowledge. First, a relaxed definition of common knowledge and, second, a restricted execution model for tasks.

The first alternative relaxes the definition of common knowledge to cases that are attainable in practical distributed systems. One such case is *epsilon common knowledge* [10], which is defined as: Every processor $\mathrm{P}i$ of the replicated group $P$ knows $\varphi$ within a time interval of epsilon. This weaker variant of common knowledge can be attained in synchronous systems with guaranteed message delivery. In the context of real-time systems, epsilon common knowledge is a "natural" relaxation from common knowledge and corresponds to a system where it is guaranteed that the outputs of replicated tasks are delivered within a known interval of time. This is typically implemented by exchanging messages to agree on the scheduling decisions. Since the communication effort to achieve epsilon common knowledge on all scheduling decisions would be prohibitive, this methodology is not considered any further. However, as epsilon common knowledge about global time is required, it is assumed that the clocks are synchronized within a value of $\varepsilon$ time units.

### 4.2 Simulating Common Knowledge

The second possibility to achieve a relaxed notion of common knowledge is *internal knowledge consistency* [21], [10] or *simulated common knowledge* [22], [23]. The idea behind this concept is to restrict the behavior of the tasks in such a way that they can act as if common knowledge were attainable. This is the case if all replicated tasks never obtain information to contradict the assumption that common knowledge has been attained. This concept of simulating common knowledge has been formalized in [23]. The class of problems where this simulation is possible is defined as *internal specifcation* [22]. These are problems that can be specified without explicit reference to real-time, i.e., it is only allowed to reference internally agreed information like messages or the time of the task activation requests. Based on this restricted class of specifications, it is possible to simulate perfectly synchronized clocks. That is, all tasks cannot detect any behavior which contradicts the assumption that clocks are perfectly synchronized. It is possible to simulate perfectly synchronized clocks either on the basis of logical clocks or on real-time clocks [22]. Simulated perfectly synchronized clocks are the prerequisite requirement for simultaneous actions and, hence, allows simulated common knowledge to be achieved.

A similar approach to simulate perfectly synchronized clocks and common knowledge is the introduction of a *sparse time base* [15]. Although this concept has not been presented in the context of distributed knowledge, it can be understood as simulating common knowledge. This concept is based on a synchronous system model with

[5] That is, clocks are synchronized to within $\varepsilon$ time units.

[6] A formal definition of what it means for a processor $P_i$ to know a given fact $\varphi$ is presented in [10].

[7] For example, no two quartz crystals have exactly the same oscillation speed. This results in slightly diverging execution speeds of processors.

approximately synchronized clocks. However, relevant event occurrences are restricted to the lattice points of the globally synchronized time base. Thus, time can be justifiably treated as a discrete quantity. It therefore becomes possible to reference the approximately synchronized clocks in the problem specification while assuming that the clocks are perfectly synchronized. Hence, it is possible to simulate common knowledge. Again, the class of problems that can be treated by this approach is restricted to internal specifications. In this case, it is possible to reference the approximately synchronized clocks, but it is not allowed to reference arbitrary external events occurring in real-time. For external events, it is necessary to carry out a consensus protocol. But, since it is possible to achieve a clock synchronization accuracy in the range of micro seconds and below [14], [13], this simulation of common knowledge and simultaneous actions comes very close to perfect simultaneity for an outside observer.

### 4.3 Simulated Deterministic Messages

Simulation of common knowledge based on a sparse time base provides the theoretical foundation for a new methodology to achieve deterministic operation in real-time systems. Currently known approaches to deterministic operation are based on solutions which try to coordinate the decisions of the local schedulers, e.g., [16], [30], [11]. It is the goal of these solutions to schedule replicated tasks in identical order. However, a much more efficient and flexible solution can be found by simulated deterministic messages. The idea behind this concept is to drop the requirement that all replicated tasks start their execution in identical order within a known interval of time. As with simulated common knowledge, the requirement for simultaneity is replaced by the weaker requirement that the system *believes* that all replicated tasks are scheduled simultaneously and that the replicated tasks output their messages simultaneously. Furthermore, it has to be guaranteed that the tasks cannot contradict the assumption that common knowledge has been attained in the system. The aim of simulated deterministic messages is to guarantee that replicated tasks behave deterministically, i.e., they produce their outputs in identical order within a defined interval. Simulated deterministic operation thus shifts the problem from scheduling replicated tasks deterministically to the problem of providing deterministic messages to the replicated tasks. While it might seem that shifting the problem does not provide any advantages, it will be shown that the a priori knowledge available in real-time systems allows a highly efficient and flexible solution to the problem.

## 5 Timed Messages

### 5.1 A Priori Knowledge in Real-Time Systems

Real-time systems are characterized by the a priori knowledge of when a task has to finish. The duration between the activation request and the required finish time of a task is called *deadline*. Since deadlines of tasks are a priori known, it follows trivially that individual processors have common knowledge on the finish times of replicated tasks. In the following, we consider a task $T_i$ which is replicated on the processor group $P = \{\mathrm{P1}, \mathrm{P2}, \ldots, \mathrm{P}p\}$ with a deadline denoted $d_i$. The local schedulers within the processor group $P$ are activated in response to a set of task activation requests $R$. Task activation request set $R_i = \{r_1, r_2, \ldots, r_r\}$ requests execution of the replicated tasks $T_i$. Task activation requests are generated either in response to the passage of time (periodic tasks) or the occurrences of external events (sporadic tasks). According to the system model, it is guaranteed for all task activation requests $r_i$ that all local schedulers agree on the timing of these requests (using an approximately synchronized clock). Based on the time of the task activation request $tr_i$ and the tasks deadline $d_i$ the corresponding task finish time $f_i$ can be defined by the sum of the tasks activation request time and its deadline $f_i = tr_i + d_i$. Since the task activation request time and the deadline are agreed upon by the processors, it follows that the finish time is also agreed upon by all processors. Hence, there is implicit agreement on the finish times of replicated tasks. Fig. 3 depicts the execution of task $T_i$ with its activation request time $tr_i$ and its finish time $f_i$. It is assumed that task $T_i$ is delayed after its request and gets preempted three times before it finishes.

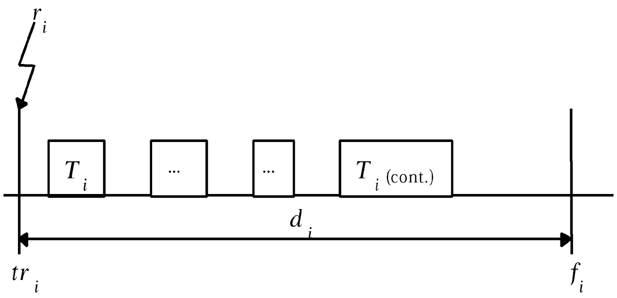

Fig. 3. Task activation request and finish time.

### 5.2 Task Finish Times and Message Validity

In the following, only intraprocessor messages, i.e., messages that are sent locally on one processor, are considered. This assumption will be removed later on.

The agreed upon task finish time can be used as follows to achieve deterministic operation. Each message $m_k$ is sent by one or more tasks. Upon sending a message, the message is associated with the sender task $T_i$'s finish time $f_i$; that is, it is assumed that the message arrives at its destination before the deadline of the task. Since messages are sent only locally within a processor, message arrival is immediate after termination of the send operation. It is therefore guaranteed that every message sent by a task arrives within the task's deadline. The associated task's finish time is called a message's *validity time* which is denoted $m_k(v)$ where $m_k(v) = f_i$. In this context, validity time means *not to use before* rather than *not to use after* time. Messages with validity times are called *timed messages*. This notion of a message's validity time was based on the assumption that messages are only sent locally within one processor. To relax this assumption and to allow intra as well as interprocessor messages, an extension to the message validity time has to be made. For interprocessor messages, the transmission time is not negligible, more specifically, according to the system model, the message transmission

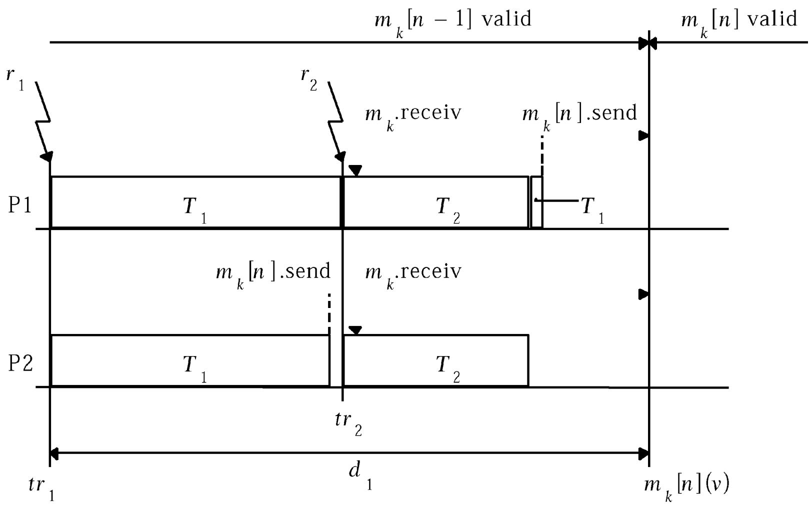

Fig. 4. Timed messages.

time may take up to $\delta$ time units. Since the clocks are only approximately synchronized to $\varepsilon$ time units, a broad or multicast message becomes available after $\delta+\varepsilon$ time units to all correct participants. Therefore, the term $\delta+\varepsilon$ has to be considered for the message validity time of interprocessor messages. This leads to the following definition for a message $m_k$'s validity time:

$$m_k(v) = \begin{cases} f_i & : \text{intraprocessor messages} \\ f_i+\delta+\varepsilon & : \text{interprocessor messages.} \end{cases} \quad (1)$$

Typically, messages are sent more than once during system operation, i.e., possibly different message contents are sent with the same message (identifier) during different invocations. Each send operation generates a new message version. The $n$th message version of $m_k$ and its validity time are denoted $m_k[n]$ and $m_k[n](v)$, respectively. Message receive operations for timed messages—denoted $m.receive()$—are defined as follows: If a task receives a message, it needs to select a message version such that the associated validity time is the largest one that is smaller than the receiver's task activation request time. More formally, if task $T_j$ with the activation request time $tr_j$ receives message $m_k$, then the following message version is returned, where $n$ is the actual number of message versions that have been sent up until now.

$$m_k.receive(tr_j) = \max_{i=0}^{n}\big(m_k : m_k[i](v) \le tr_j\big). \quad (2)$$

Note that all variables in (2) are common knowledge.

If all tasks adhere to this algorithm, then it is guaranteed that they receive the same message contents regardless of when the actual execution of the receive operation takes place. Because the activation request times (as well as the validity times of messages) are agreed upon for replicated tasks, it is guaranteed that all replicas select the same version of a message. Compared to normal message delivery, where messages become available immediately after being sent, timed messages become available at the latest possible (but still correct) moment.

If replicated tasks restrict their implementation to internal specifications, i.e., they make no references to real-time, then timed messages provide a simulation of common knowledge. This does not preclude that any reference is made to real-time at all. It rather restricts the access to timing events which are agreed upon by all replicated tasks. It is then guaranteed that replicated tasks receive the same messages in the same order and produce identical outputs. The replicated tasks *believe* that they are scheduled simultaneously since they have no possibility to prove otherwise. According to the system model, sending of messages is the only behavior of tasks that is visible to other tasks. Timed messages guarantee that messages become available within the time interval that is given by the accuracy of the approximately synchronized clocks. Hence, the execution of replicated tasks becomes independent of the local schedulers' decisions and the resulting execution order among the individual processors as long as all local schedulers are able to meet the deadlines of the tasks.

The importance of timed messages stems from the fact that no additional communication is necessary to achieve simulated common knowledge. This is the reason why shifting the problem from scheduling replicated tasks deterministically to the problem of deterministic messages provides much higher performance and flexibility.

### 5.3 An Example

The principle of operation can be illustrated by the following example (see Fig. 4). This example shows that timed messages guarantee deterministic execution of tasks despite the fact that the execution order and timing of replicated tasks is nondeterministic. Again, there are two processors P1 and P2 which have to execute the replicated set of tasks $T=\{T_1,T_2\}$, where $T_1$ sends a message to $T_2$, but their executions are not synchronized (i.e., there is no precedence relation). At arrival of the task activation request $r_1$, the replicated task $T_1$ is started immediately by the schedulers of the processors P1 and P2. Due to minor differences in the execution speed, task $T_1$ finishes at processor P2 a few instructions earlier than at processor P1. Upon finishing, task $T_1$ sends the $n$th version of message $m_k$. Since $T_1$ finishes earlier on P2, this send operation $m_k[n].send$ is carried out at P2 before the task activation

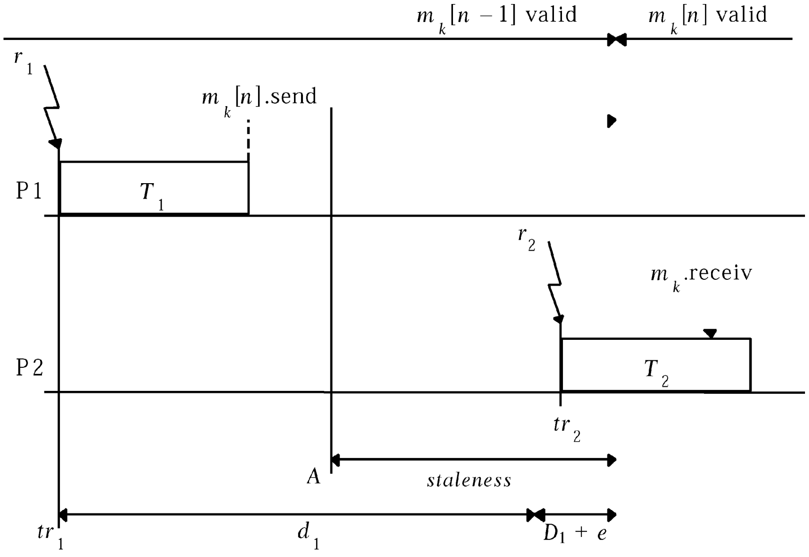

Fig. 5. Data staleness.

request $r_2$ arrives. It is assumed that task $T_2$ has higher priority than task $T_1$. Task $T_2$ is therefore started at both processors immediately after arrival of the task activation request $r_2$. Hence, task $T_1$ is preempted by task $T_2$ at processor P1 before $T_1$ has sent message $m_k$. Without timed messages, this would result in an inconsistent order and timing of message outputs. As with the example in Fig. 2, it is assumed that task $T_2$ receives message $m_k$ when starting its execution. According to the definition of timed messages, upon receiving message $m_k$, the validity time of all message versions is compared against the task activation request time $tr_2$. At time $tr_2$, processor P1 has sent $n-1$ message versions and processor P2 has sent $n$ message versions of message $m_k$. But, since the validity time of message $m_k[n](v) = tr_1 + d_1$ and the task activation request time of $T_2$ is $tr_2$ and $tr_2 < m_k[n](v)$, it follows that the receive operation of task $T_2$ returns message version $m_k[n-1]$ at processor P1 and P2. That is, since the replicated processors agree upon the task activation request time $tr_2$ and the message validity time $m_k(v)$, it is guaranteed that identical message versions are received by the replicated tasks regardless of nondeterministic behavior of the local schedulers.

Timed messages can be implemented at the level of the operating system. This encapsulates implementation details of timed messages such that replicated tasks which send or receive messages do not perceive any difference from the "normal" message semantics. This allows the simulation of common knowledge by timed messages which is transparent to the application program.

### 5.4 Performance Issues

The protocol as it stands is effective in that all processes will select the same version of the message on each receive request, thus preserving replica determinism. It is, however, under certain circumstances, overly pessimistic. It might happen that receiver replicas are forced to use an old message long after the new message has, in reality, become stable. Consider, for example, the scenario shown in Fig. 5.

In the above example, $T_1$ (on processor $P_1$) executes the send fairly early in a relatively long deadline; $T_2$ (on $P_2$) begins execution around the time of $T_1$'s deadline and requests the message some time later. The message itself arrives (on $P_2$) at time $A$.

Under these circumstances, however, $tr_1 + d_1 + \Delta_1 + \varepsilon > tr_2$ and, thus, $T_2$ will be forced to read the old value of the message despite the fact that the new message has, in reality, been available for some time and is available on all nodes with replicated versions of $T_2$. Thus, the data actually received is stale by at least the amount shown in the diagram. Modifications to the protocol can reduce substantially the potential staleness of the data received and a number of such options are now discussed.

#### 5.4.1 Use Common Knowledge about the Worst-Case Response Time of the Task and Message

In the above, it has been assumed that the only common knowledge available is the deadlines of the tasks. However, modern schedulability analysis allows the calculation of task's worst-case response times [1], [12], which, in some cases, will be considerably less than their deadlines. We can also assume that a particular message's worst case delivery time, $\Delta$, can be calculated where $\Delta \leq \delta$. Consider Fig. 6.

In Fig. 6, the validity time for the message is now defined by the worst-case response time for all the replicas of $T_1$. This worst-case response time is denoted $W_1$. Again, for the message validity time, two cases have to be considered: inter and intraprocessor message communication. The resulting equation for the message validity time is given by:

$$m_k(v) = \begin{cases} W_1 & : \text{intraprocessor messages} \\ W_1 + \Delta_i + \varepsilon & : \text{interprocessor messages.} \end{cases} \quad (3)$$

Since all the variables in this equation are common knowledge, correctness of the timed message mechanism is still guaranteed.

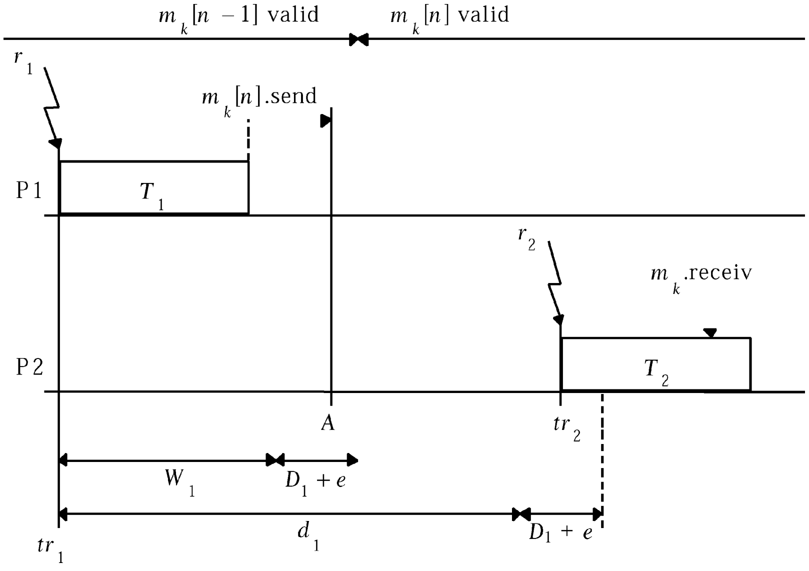

Fig. 6. Timed messages using worst-case response time.

### 5.4.2 Using Response Time for Actual Message Send Operation

Recent schedulability analysis has extended response time analysis to enable the worst case time for any internal computing event to be calculated [5]. Let $S_1$ be the worst-case time for sending the message; as with $W_1$, this is common knowledge. The validity time equation now becomes:

$$m_k(v) = \begin{cases} S_1 & : \text{intraprocessor messages} \\ S_1 + \Delta_i + \varepsilon & : \text{interprocessor messages.} \end{cases} \quad (4)$$

Note that this case only applies to interprocessor communication and that it is therefore not necessary to consider intraprocessor communication in the equation.

### 5.4.3 Using the Best-Case Receive Message Time

Schedulability analysis can also be used to obtain more precise knowledge of when the receiver of the message actually issues the receive-message request. However, in this case, the best-case response time, $B$, must be calculated. This can now be used instead of just $tr$ to select the correct message version. The equation for the message receive operation thus is given by:

$$m_k.receive(tr_j) = \max_{i=0}^{n}\big(m_k : m_k[i](v) \leq tr_j + B_j\big). \quad (5)$$

The overall staleness of any message has now been significantly reduced, as illustrated in Fig. 7.

### 5.4.4 Using Actual Send Times for Single Source Messages

Many messages are sent by only one task (but read by a replicated set of receiver tasks).[8] For these tasks, it is possible to base the message validity time on the actual time of sending. The time of message sending (as measured on the local clock) for a message $m_k$ is denoted $tmsend_k$. The resulting equation for the message validity time is given by:

$$m_k(v) = tmsend_k + \Delta_k + \varepsilon. \quad (6)$$

## 5.5 Agreeing Release Times for Sporadic Tasks

Sporadic tasks are activated in response to external events. This, however, violates the requirement for timed messages that simulated common knowledge and the notion of *internal specifications* is satisfied. According to the definition of internal specifications, it is not possible to reference external events.

To enable sporadic task activations within the framework of timed messages, it is necessary to add an agreement phase before the sporadic task is started. During this phase, agreement on *one* activation request time has to be achieved. After this agreement phase, it is guaranteed that the tasks activation request time have become common knowledge. The external event has thus been *internalized* and conforms now with the requirements of internal specifications and timed messages.

A typical implementation for this agreement phase will use an interrupt service routine to record the timestamp of the sporadic request for task activation. This interrupt routine starts an agreement protocol where the timestamp is used as input. Depending on the fault hypothesis, a suitable agreement protocol has to be selected. For crash and omission failures, it is sufficient that only one interrupt service routine sends its timestamp by means of a reliable broadcast protocol to all the other participants. For a more severe fault hypothesis, all participants have to send their timestamp. Having sent the timestamp, all correct participants wait for the arrival of the communicated timestamp(s). To tolerate crash or omission failures, it is sufficient to take the first arriving timestamp. To tolerate $f$ timing or authentication detectable Byzantine failure, it is sufficient to wait for the arrival of $2f + 1$ timestamps. In the general case, for Byzantine failures, it is necessary to wait for the arrival of $3f + 1$ timestamps. Based on the received timestamps, one common decision for the task's activation

[8] This illustrates the flexibility of the approach since there are no restrictions to mix replicated and nonreplicated tasks on one processor.

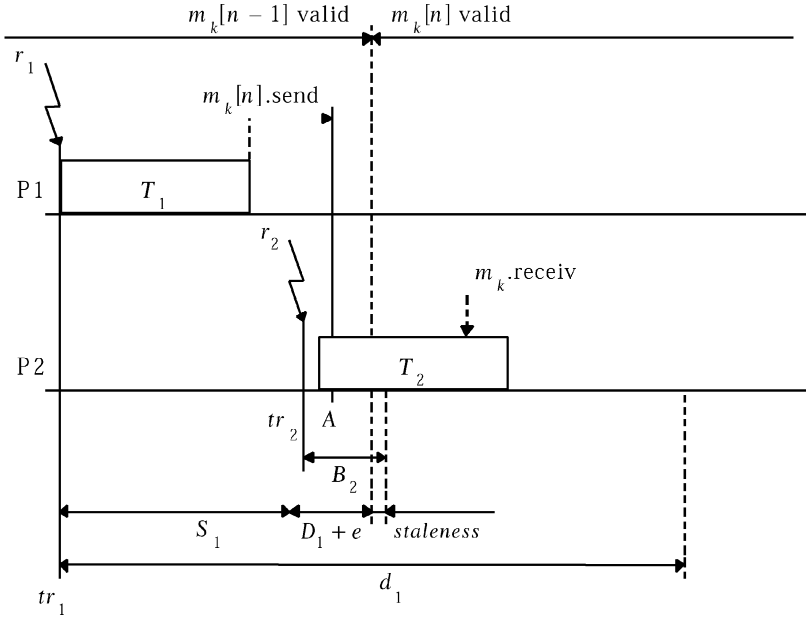

Fig. 7. Reduced staleness.

request time is derived. This information can now, in turn, be used to activate the requested task at one agreed upon point in time.

If sporadic tasks are released by internal events (from either periodic or sporadic tasks), then the release time can be derived from the release time of the "parent" task plus its best-case response time at the point where the event is sent.

### 5.6 Message Versions and Implementation

Until now, the unrealistic assumption has been made that all versions of a message which have been sent during the system operation are available. To drop this assumption, it is necessary to establish criteria to decide how many versions of a message have to be kept in the system simultaneously and how to discard old message versions. Compared to "normal" message semantics, timed messages require that more than one message version has to be kept under certain circumstances. This requires additional memory space for messages, which is the cost for guaranteeing deterministic behavior of tasks.

To give the exact number of versions for timed messages, assume a replicated task $T_i$ with a deadline $d_i$ which receives a message $m_k$. The maximum number of message versions $m_k[n]$ that becomes valid during task $T_i$'s deadline $d_i$ is called the *nondeterministic send rate* of message $m_k$, which is denoted $NDSR(m_k, T_i)$. If $TS(m_k)$ is the set of all tasks that send message $m_k$ and $p_j$ denotes the minimum interarrival period for two task activation requests of task $T_j$, then the nondeterministic send rate $NDSR(m_k, T_i)$ can be defined formally as:

$$NDSR(m_k, T_i) = \sum_{T_j \in TS(m_k)} \left( \left\lceil \frac{d_i}{p_j} \right\rceil + 1 \right). \tag{7}$$

If $TR(m_k)$ is the set of all replicated tasks that receives message $m_k$, then the following number of message versions $MVersions(m_k)$ for timed state-messages is necessary:

$$MVersions(m_k) = \max_{T_i \in TR(m_k)} (NDSR(m_k, T_i)). \tag{8}$$

This sufficient, but not necessary, condition guarantees that *enough* message versions are kept such that all replicas of a task can access one consistent message version. An informal proof can be given as follows: If the replicated task $T_i$ is activated by task activation request $r_i$, then all replicas agree on the timing of the task activation request $tr_i$ and the task finish time $f_i = tr_i + d_i$. It is furthermore guaranteed that all task replicas of $T_i$ perform their receive operation during the time interval $[tr_i, f_i]$ (since they have to finish within their deadline). It is therefore guaranteed that all message versions are kept which are valid during this time interval. Particularly, this implies that the message version which is valid at time $tr_i$ is available for all replicated tasks. This proves that *enough* message versions are kept since, according to the definition of timed messages, a receive operation would return the message version which is valid at time $tr_i$.

For timed messages, it is therefore possible to derive the number of message versions for a given system by means of static analysis. The only two necessary preconditions are 1) that the set of sender and receiver tasks are known for each message and 2) that the deadline and the minimum interarrival period are known for all tasks. Since both conditions are satisfied for real-time systems, allocation of message versions can be made safely. Indeed, as was utilized in Section 5.4, if worst case response times $W_i$ or $S_i$ are known, they can be substituted for $d_i$ in (7).

An important issue concerning timed messages is how to manage message versions with different validity times. Depending on the maximum number of message versions that have to be stored simultaneously, different implementations have to be selected. Typically, messages are sent by only one task and the minimum interarrival period of tasks is larger than the deadlines. According to (8), this results in only two message versions. Under this assumption it is sufficient to store an old version of the message, a new

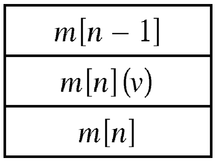

Fig. 8. Timed message data structure.

version, and the validity time of the new message version,[9] see Fig. 8.

The corresponding message operations, **send** and **receive**, are shown in Fig. 9. The caller tasks activation request time $tr_i$ and its deadline $d_i$ are known by the operating system and, therefore, do not have to be passed as function arguments to the operations send and receive.

Further optimizations are possible when it is noted that most intertask communication is used to implement precedence relations. Two common forms of such operations are *event-based transactions* and *time-based transactions*. With event-based, sporadic tasks are used, with the release of a task being caused by an event message sent by the previous task in the transaction; therefore, no messages need to be stored.

Time-based transactions use offsets in time to ensure that a task is not released until any messages it requires are available locally. Consider the validity time for a typical message $m_k[n](v)$. If $tr_2$ is set to this value, then a timed message is not needed; neither are additional message versions. An ordinary message will suffice (although the use of a timed message will enable the released task to know if a fault has occurred further up the transaction—the validity time would be "out of date").

### 5.7 Evaluation

Figs. 8 and 9 show that the overhead for timed messages in terms of memory space and runtime is very small. An experimental implementation of the message operations **send** and **receive** has shown an overhead of 0.625 $\mu s$ and 0.750 $\mu s$, respectively. This implementation has been carried out in the framework of ERCOS [24] under the assumption that the message length is equal to the word length of the processor.

With the proposed method, no more messages are sent than would be required for the support process group communication. Buffer sizes for timed message pools are increased, but by manageable amounts. Message lengths are increased, but again only by a small amount—the size of a timestamp. In effect, replica determinacy is implemented at virtually no extra cost. This compares very favorably with the overheads that occur when attempts are made to coordinate the behavior of the individual schedulers.

It was also noted in the previous section that, in many common software structures, an implementation can significantly simplify the buffering—or eliminate it altogether. There are also situations in which timed messages can be safely replaced by ordinary (untimed) messages.

```
Message::Send(value)
  begin atomic;
    $m[n-1] := m[n]$
    $m[n] :=$ value
    $m[n](v) := ...$ [10]
  end atomic;
```

```
Message::Receive()
  begin atomic;
    if ($tr_i < m[n](v)$)
      return $m[n-1]$
    else
      return $m[n]$
  end atomic;
```

Fig. 9. Send and receive timed message. ([10]The message's validity has to be set according to the selected optimization strategy.)

## 6 Conclusion

To achieve fault tolerance, real-time systems typically replicate tasks over a number of distributed processors. These replicated tasks are required to deliver identical outputs in an identical order within a specified time interval. This requirement, called replica determinism, is violated by distributed scheduling algorithms if on-line scheduling, preemptive scheduling, and nonidentically replicated task sets are used. To avoid the inconsistent order and timing of replicated tasks, global coordination of the local schedulers decisions have been employed previously. This approach, however, results in a very high communication overhead to agree on each scheduling decision or it restricts scheduling such that no on-line scheduling, no preemptions, and only identically replicated task sets may be used.

To overcomes these problems a new method—called timed messages—has been introduced. It has been shown that deterministic operation of replicated tasks corresponds to the problem of common knowledge. Since common knowledge cannot be attained in practical systems, a weaker notion, called simulated common knowledge, is used to achieve deterministic operation. The idea behind this approach is that the system *believes* that all replicated tasks are executing simultaneously, while the tasks have no possibility to prove the opposite. This simulation is implemented by means of timed message. With timed messages, each message version is associated with a validity time. If a task receives a timed message, then the message version with the appropriate validity time is returned. This guarantees that replicated tasks receive identical message versions, regardless of their actual execution timing. Furthermore, since the validity time of messages is based on approximately synchronized clocks, it is guaranteed that timed messages sent by replicated tasks become valid in identical order and within an appropriate time interval.

The major advantage of timed messages is its efficiency and flexibility while guaranteeing deterministic operation of replicated tasks. It has been shown that no extra communication is necessary to establish agreed upon validity times. Additionally, by using timed messages, there are no restrictions on the local schedulers. They can use on-line scheduling, preemptions, and nonidentically replicated task sets. The price of timed messages is that, in some cases, more than one message version has to be kept. For timed messages, it has been shown that the number of message versions can be determined by static system analysis. This allows safe system construction since it is

[9] The old message version does not need a validity time since it is guaranteed by static analysis that the old version is valid whenever the new version is not valid.

guaranteed that the system cannot run out of message space during peak load scenarios. Timed messages can be implemented at the operating system level, which provides transparency to the application program. A number of optimizations have also been considered.

The use of timed messages facilitates the application of a wide range of fault tolerance measures within the context of flexible scheduling. This, in turn, opens up the possibilities for adaptive dependable hard real-time systems.

## Acknowledgments

The authors would like to thank H. Kopetz and H. Appoyer for their valuable comments and suggestions on earlier versions of the paper. This work has, in part, been supported by the ESPRIT projects DeVa, *X*-By-Wire and GUARDS, and by the UK EPSRC.

## References

[1] N.C. Audsley, A. Burns, M. Richardson, K. Tindell, and A.J. Wellings, "Applying New Scheduling Theory to Static Priority Preemptive Scheduling," *Software Eng. J.*, vol. 8, no. 5, pp. 284-292, 1993.

[2] I.J. Bate, A. Burns, and N.C. Audsley, "Putting Fixed Priority Scheduling Theory into Engineering Practice," *Proc. Second IEEE Real-Time Applications Symp.*, 1996.

[3] P.A. Barrett, A. Burns, and A.J. Wellings, "Models of Replication for Safety Critical Hard Real-Time Systems," *Proc. 20th IFAC/IFIP Workshop Real-Time Programming (WRTP '95)*, 1995.

[4] A. Burns, "Preemptive Priority Based Scheduling: An Engineering Approach," *Advances in Real-Time Systems*, S.H. Son, ed., pp. 225-248, Prentice Hall, 1994.

[5] A. Burns, K. Tindell, and A.J. Wellings, "Fixed Priority Scheduling with Deadlines Prior to Completion," *Proc. Sixth Euromicro Workshop Real-Time Systems*, pp. 138-142, 1994.

[6] German patent application DE 35 06 118 A1. Verfahren zum Betreiben einer Datenverarbeitungsanlage für Kraftfahrzeuge. (CAN), filed by Robert Bosch GmbH, 22 Feb. 985.

[7] F. Cristian, H. Aghili, R. Strong, and D. Dolev, "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement," *Proc. 15th Int'l Symp. Fault-Tolerant Computing (FTCS-15)*, pp. 200-206, June 1985.

[8] F. Cristian, "Synchronous Atomic Broadcast for Redundant Broadcast Channels," *J. Real-Time Systems*, vol. 2, no.. 3, pp. 195-212, Sept. 1990.

[9] J. Halpern and Y. Moses, "Knowledge and Common Knowledge in a Distributed Environment," *Proc. ACM Symp. Principles of Distributed Computing*, pp. 50-61, Aug. 1984 (revised version dated Nov. 1985 ).

[10] J. Halpern and Y. Moses, "Knowledge and Common Knowledge in a Distributed Environment," *J. ACM*, vol. 37, no.. 3, pp. 549-587, July 1990.

[11] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, C. Senft, and R. Zainlinger, "The MARS Approach," *IEEE Micro*, vol. 9, no. 1, pp. 25-40, Feb. 1989.

[12] M.H. Klein et al., *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems.* Kluwer Academic, 1993.

[13] H. Kopetz, A. Krüger, D. Millinger, and A. Schedl, "A Synchronization Strategy for a Time-Triggered Multicluster Real-Time System," *Proc. 14th Symp. Reliable Distributed Systems*, Sept. 1995.

[14] H. Kopetz and W. Ochsenreiter, "Clock Synchronization in Distributed Real-Time Systems," *IEEE Trans. Computers*, vol. 36, no. 8, pp. 933-940, Aug. 1987.

[15] H. Kopetz, "Sparse Time versus Dense Time in Distributed Real-Time Systems," *Proc. 12th Int'l Conf.n Distributed Computing Systems*, pp. 460-467, June 1992.

[16] R.M. Kieckhafer, P.M. Thambidurai, C.J. Walter, and A.M. Finn, "The MAFT Architecture for Distributed Fault-Tolerance," *IEEE Trans. Computers*, vol. 37, no. 4, pp. 394-405, 1988.

[17] L. Lamport and P.M. Melliar-Smith, "Synchronizing Clocks in the Presence of Faults," *J. ACM*, vol. 32, no. 1, pp. 52-78, Jan. 1985.

[18] D. Locke, "Software Architectures for Hard Real-Time Applications: Cyclic Executives vs Fixed Priority Executives," *Real-Time Systems*, vol. 4, no. 1, pp. 37-53, 1992.

[19] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *Trans. Programming Languages and Systems*, vol. 4, no. 3, pp. 382-401, 1982.

[20] H.-J. Mathony and S. Poledna, "Real-Time Software for In-Vehicle Communication," *Proc. SAE Int'l Congress and Exposition*, pp. 1-9, Feb. 1996.

[21] G. Neiger, "Knowledge Consistency: A Useful Suspension of Disbelief," *Proc. Second Conf. Theoretical Aspects of Reasoning about Knowledge*, pp. 295-308, 1988.

[22] G. Neiger and S. Toueg, "Substituting for Real Time and Common Knowledge in Asynchronous Distributed Systems," *Proc. Sixth ACM Symp. Principles of Distributed Computing*, pp. 281-293, Aug. 1987.

[23] G. Neiger and S. Toueg, "Substituting for Real Time and Common Knowledge in Asynchronous Distributed Systems," *J. ACM*, vol. 40, no. 3, pp. 334-367, Apr. 1993.

[24] S. Poledna, T. Mocken, J. Schiemann, and T. Beck, "ERCOS—An Operating System for Automotive Applications," *Proc. SAE Int'l Congress and Exposition*, pp. 55-65, 1996.

[25] S. Poledna, "Replica Determinism in Fault-Tolerant Real-Time Systems," PhD thesis, Technical Univ. of Vienna, Institut für Technische Informatik, 1994.

[26] S. Poledna, "Deterministic Operation in Fault-Tolerant Distributed Real-Time Systems," Research Report 28/95, Technical Univ. of Vienna, Institut für Technische Informatik, 1995, *Proc. Sixth IFIP Int'l Working Conf. Dependable Computing for Critical Applications*, 1997.

[27] S. Poledna, "Fault-Tolerance in Safety Critical Automotive Applications: Cost of Agreement as a Limiting Factor," *Proc. 25th Int'l Symp. Fault-Tolerant Computing*, pp. 73-82, June 1995.

[28] S. Poledna, *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism.* Kluwer Academic, 1995.

[29] "Replicated Software Components," *Delta-4: A Generic Architecture for Dependable Computing*, D. Powell, ed., vol. 1 of ESPRIT Research Reports, chapter 6.4, pp. 100-104,Vienna, New York: Springer Verlag, 1991.

[30] "Semi-Active Replication," *Delta-4: A Generic Architecture for Dependable Computing*, D. Powell, ed., vol. 1 of ESPRIT Research Reports, chapter 6.7, pp. 116-120, Vienna, New York: Springer Verlag, 1991.

[31] M.K. Reiter and K.P. Birman, "How to Securely Replicate Services," *ACM Trans. Programming Languages and Systems*, vol. 16, no. 3, pp. 986-1,009, 1994.

[32] F.B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 229-319, 1990.

[33] Siemens *Microcomputer Components SAB 80C167 16-Bit CMOS Single-Chip Microcontrollers for Embedded Control Applications, User's Manual*, 1993.

[34] P. Veríssimo, "Causal Delivery Protocols in Real-Time Systems: A Generic Model," *J. Real-Time Systems*, vol. 10, no. 1, pp. 45-73, Jan. 1996.

[35] P. Veríssimo, L. Rodrigues, and M. Baptista, "AMp: A Highly Parallel Atomic Multicast Protocol," *Proc. SIGCOMM Symp.*, pp. 83-93, Sept. 1989.

**Stefan Poledna** received the MSc and PhD degrees in computer science from the Technical University Vienna, Austria, in 1991 and 1994, respectively. Since 1982, he has been active in industry, focused on automotive electronics. He is the cofounder and CEO of TTTech Time-Triggered Technology, a high-tech spin-off from the Technical University of Vienna providing development products and support for TTP (Time-Triggered communication Protocol). At the Technical University of Vienna, he lectures on fault-tolerant computing. Dr. Poledna is a member of the IEEE Computer Society and the ACM.

**Alan Burns** graduated in mathematics from the University of Sheffield in 1974 and undertook his PhD at the University of York before taking up a tenured post at the University of Bradford. He is a professor of real-time systems in the Department of Computer Science, University of York, United Kingdom, which he rejoined in 1990 and where he was promoted to a personal chair in 1994. Together with Professor Andy Wellings, he heads the Real-Time Systems Research Group at the university, a group that has currently four faculty members, nine postdoctoral researchers, and seven PhD students. He has served on many program committees and has been program committee chair and general chair for RTSS. He has published widely and works on a number of research areas within the real-time field. He is a member of the IEEE.

**Andy Wellings** is a professor of real-time systems at the University of York, United Kingdom in the Computer Science Department. He is interested in most aspects of the design and implementation of real-time dependable computer systems. He has authored/coauthored more than 150 papers/reports. He is European editor-in-chief for the computer science journal *Software-Practice and Experience*. Together with Professor Alan Burns, he heads the Real-Time Systems Research Group at the University of York. He is a member of the ACM.

**Peter Barrett** is Research and Development Manager for Tradezone International Limited, a United Kingdom company developing applications to support highly secure business-to-business electronic commerce over the Internet. Prior to this, at the time the work reported here was undertaken, he was a member of the Centre for Software Reliability at the University of Newcastle upon Tyne, United Kingdom, where he specialized in dependable architectures for safety-critical applications such as aircraft flight control and process control. He has nearly 20 years of experience in both academic and industrial environments, much of which has been related to the field of fault tolerance and systems dependability.