In Proc. of the 10th IEEE Symposium on Computers and Communications (ISCC'05), 2005.

Describing Multidimensional Schedules for Media-Access Control in Time-Triggered Communication

Sebastian Fischmeister
University of Pennsylvania
sfischme@seas.upenn.edu

Abstract

A shared communication medium is characterized by multiple entities that use this medium by reading and writing from and to it. Write operations on the shared communication medium must be coordinated and collision-avoidance schemes are one technique to achieve this; for example time-division multiple access (TDMA). Common solutions for TDMA include descriptive tables or algorithm-based client/server mechanisms. Yet, they are all limited in their expressiveness: at the beginning of the communication period at most one write operation can be scheduled for a specific time slot.

In this work, we propose a system that allows for scheduling several write operations for the same time slot but guarantee that at most one will be performed though. It does not deal with scheduling algorithms per se, it deals with describing and implementing a computed schedule. The consequences of this added expressiveness allow for parallel and stateful communication schedules merged and serialized in an ad-hoc way.

The contribution is the proposed more-expressive yet still value and time-deterministic way of describing communication schedules for time-triggered communication plus a description of its implementation in an interpreter implemented as infrastructure in RTLinuxPro.

1 Introduction

A shared communication medium (SCM) is characterized by multiple entities that use this medium by reading and writing from and to it. Data writers perform write operations on the SCM. Data readers perform read operations and retrieve values written by data writers. An example of such a SCM is the network in a distributed system such as Ethernet in a distributed application or a CAN bus in an automotive application.

If multiple data writers are present, then collision will occur on the SCM in case at least two data writers have performed write operations at the same time. A collision scrambles data written by the data writers and data readers will not receive a correct message. Several different strategies exist to implement error-free communication on a SCM [1]. Collision-detection schemes detect a collision on the SCM and require the data writers to retransmit the message. Collision-avoidance schemes constrain write access for data writers and these constraints guarantee collision-free communication. SCM are widespreadly used in non–real-time and real-time applications.

In real-time applications, timeliness of tasks is a first-class citizen. In hard real-time applications, if a deadline is missed, a catastrophic failure with possibly human loss may be a consequence [2]. Timeliness of tasks implies deterministic communication. This involves communication via SCM, especially in distributed real-time applications [2]. To guarantee deterministic communication, collision-detection schemes must guarantee deadlines until what time a specific message will have been transmitted in the worst case possible. Collision-avoidance schemes must also guarantee deadlines, however, they do not have to worry about scrambled messages as they prevent them implicitly. One well understood collision-avoidance scheme is time-division multiple access (TDMA). It divides time into slots and assigns single slots to at most one data writer. Within the specified slot, the data writer can perform write operations on the SCM without interference from any other data writer. A scheduling algorithm assigns the individual slots to data writers and thereby generates a communication schedule (in contrast to an execution schedule that describes when to dispatch tasks to the CPU).

Currently, to provide guarantees, the TDMA-based communication schedule is calculated offline or at least at the beginning of the communication cycle. So the communication cycle is static. In this work, we propose a mechanism to define more dynamic schedules that can change within the communication cycle, but still provide guarantees. This work solves problems in the context of flexible distributed real-time systems, where meeting communication deadlines is essential, however, static and a-priori systems are not viable due to the demands of the dynamic, chancing environment.

In this work, we first describe a network-code machine (NCM) in Section 2 that can dispatch arbitrary TDMA-based communication schedules (Section 3). Then we introduce the concept of multidimensional schedules (Section 4) and stateful schedules that provide verifiable and flexible communication and increase the expressiveness of TDMA-based communication schedules. Finally, we provide an overview of our toolchain to generate code for the nc-machine (Section 5), discuss related work 6, and close the paper (Section 7).

2 Network-Code Machine

The network-code machine (NCM) conceptually splits communication into two parts: producing values and communicating values (writing them on the SCM). The application on top of the NCM produces values by running application-specific functionality. The NCM is only responsible for communicating values. Communicating values includes reading and writing values from and to a SCM such as a computer network.

The NCM is an interpreter that executes TDMA-based communication schedules in form of network-communication code (n-code). Such a schedule specifies for each participating node: (1) when it is allowed to perform write operations on the SCM, (2) when it should perform read operations on the SCM, and (3) which value (variable) it should use for the write or read operation. Unlike other approaches (see Section 6), n-code does not necessarily have to be a linear communication schedule encoded in a communication scheduling table. N-code is equivalent to a program that is interpreted and executed by the NCM.

The NCM and the application do not necessarily depend on each other. The application and the NCM may either run independently from each other or they use a combined execution schedule. The NCM and the application interact via a predefined interface that encapsulates a memory region (further denoted as *communication buffer)*. The application performs write and read operations on this communication buffer. Depending on the type of application (real time or non real time), the interface of the communication buffer may differ. For example, when implementing an application for the domain of real-time systems, the communication-buffer layout is predefined and the application and the NCM perform read and write operations with specific addresses. Predefined communication-buffer layouts are important for real-time applications because the execution requirements of tasks may result in transferring a specific value (e.g., a task output) at a specific moment in time. Non–real-time applications do not have such requirements. When implementing a non–real-time application, the communication buffer acts as mere queue using the first-in–first-out mechanism.

2.1 Instruction Set

Communication requires the participants to perform different steps such as communicating a value or listening for a value, or waiting. The n-code instructs the NCM what action it should perform on the SCM and the communication buffer.

- **Send.** The instruction $send(loc, id, dl, act)$ tells the NCM to write a value on the SCM. The location loc denotes a location in the communication buffer and specifies which value to be communicated. The id identifies the communication to which this value belongs to. The deadline dl specifies the point in time until which the NCM must have communicated the value. The time value act defines, the point in time at which this value will be accessible to other tasks[1].

- **Receive.** The instruction $receive(loc, id)$ tells the NCM to read a value from the SCM. The location loc specifies the address within the communication buffer, at which the value will be written. The value id identifies the communication to which the received value belongs to.

- **Future.** The instruction $future(dl, jmp)$ schedules a wait operation that will halt the NCM until the deadline dl has passed and then jumps to the position given by jmp.

- **Signal.** The instruction $signal(dl, s)$ is a special form of the instruction $send$ and $receive$. It does not transmit/receive an application-specific value, but it transmits/receives a protocol-specific symbol. The NCM must have transmitted/received the signal before the deadline dl has passed. The symbol s denotes the symbol that is put on or read from the SCM.

  The instruction $signal$ is executed by sending and receiving nodes. The current communication master transmits the symbol and all other member nodes read it.

- **Return.** The instruction $return()$ suspends the NCM until it is resumed by a $future$ instruction.

- **If.** The instruction $if(g, jmp)$ implements a conditional jump. If the guard g evaluates to true, the program counter of the NCM is set to the address jmp. Otherwise, the program counter is increased by one.

[1]Necessary only for time-triggered computation.

2.2 Example Communication

Figure 1 provides an example of a time-triggered communication which involves three nodes (n_1, n_2, and n_3). The communication round has a length of three slots in which at most one node can communicate at the same time and the communication cycle consists of one communication round. All communication on the SCM is broadcast (i.e., a message transmitted by n_1 can be received by n_2 and n_3, however, they may discard the message). The communication graph shows that n_2 receives messages sent by n_1 and n_1 and n_3 receive messages sent by n_2.

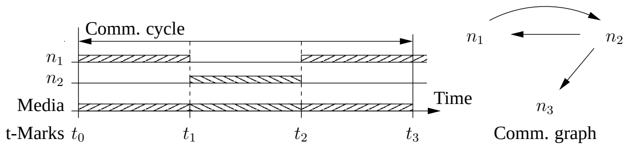

Figure 1. TDMA communication example.

Node n_1 performs three operations on the SCM: First, it sends data, then it receives data from n_2, and finally it sends data again. Node n_2 also performs three operations: First, it receives data from n_1, then it sends data, and finally it receives data from n_1 again. Node n_3 performs only one operation: it receives data from n_2. The following listing shows the corresponding n-code for n_2.

```
Node n_1:
L0: future t_0 , L1
    return
L1: send t_1 , A, t_x
    future t_2 , L2
    return
L2: receive B
    future t_2 , L3
    return
L3: send t_3 , A, t_z
    future t_3 , L0
    return
```

```
Node n_2:
L0: future t_1 , L1
    return
L1: receive A
    future t_1 , L2
    return
L2: send t_3 , B, t_y
    future t_3 , L3
    return
L3: receive t_3 , A
    future t_3 , L0
    return
```

3 Arbitrary TDMA

This section provides the basic formalism required for the following proof and the concepts presented in the following sections. We claim that the NCM and its instruction set is sufficient to express arbitrary TDMA-based communication schedules. This means, the NCM is equivalent expressive to schedules described by tables but also schedules generated at run-time by complex scheduling algorithms. Proofing the former is trivial, proofing the latter requires some effort.

The set N includes all computation nodes n of the system that use the SCM. The set $C := N \times N := \{(n_i, n_j)|n_i \in N, n_j \in N\}$ contains all communication of the distributed application and the tuple (n_i, n_j) describes a point-to-point communication between n_i and n_j in which n_i is the sender and n_j is the receiver. The set B is a subset of C^* and describes all valid broadcast communication by $B := \{b|b \in C^*, \forall c \in C, c \in b : n_{0,i} = n_{0,j}\}$. TDMA-based communication schedules involve time and timestamps. The set T refers to the time base and elements $t \in T$ are timestamps. We use the mapping $TS : N \times T \to T$ with $TS(n, t_l) = t_g$ where $TS(n, t_l)$ relates a time t_l at node n to a global time t_g. For slotted communication, we need to introduce slots to the model. First, we introduce $\chi \subseteq T$ with $\chi := \{(t_i, t_j)|t_i \leq t_j; t_i, t_j \in T\}$ and ς as slot length. The mapping $S : \mathbb{N} \to \chi$ with $S(n) = \{(t_i, t_j)|t_i = n * \varsigma + \delta(n), t_j = t_i * \varsigma\}$ where $S(n)$ defines a time frame $[t_i, t_j)$ in which t_i is starting moment and t_j the excluded ending moment of slot number n. The function $\delta(n)$ can introduce gaps in the schedule between single slots. The mapping $slotting : S(N) \to B^*$ with $slotting(S(n)) = \Psi$ provides the broadcast communication that happens in a specific slot with $\Psi \in B$ that satisfies the communication requirements specified by the application. The scheduler determines the slot assignment and thus calculates the result of $slotting(S(N))$.

In the following proof, we show examples that base on the communication depicted in Figure 1 with $Z \subseteq C$, $Z := \{(n_1, n_2), (n_2, n_1), (n_2, n_3)\}$, $S := \{(t_0, t_1), (t_1, t_2), (t_2, t_3)\}$.

Theorem 3.1 *The NCM can implement arbitrary collision-free TDMA-based communication patterns, i.e., for any C if $\exists\Psi$ that satisfies the communication requirements of the application for all B, we can generate code for the NCM.*

The goal of TDMA is to prevent collision on the communication channel. Consequently it is intended that at most one node performs a write operation on the channel at point t_x. Consequently, $\forall n : 0 \leq |\Psi| \leq 1$. The set B follows this definition: Given the communication pattern in Figure 1, $B := \{\emptyset, \{(n_1, n_2)\}, \{(n_2, n_1)\}, \{(n_2, n_3)\}, \{(n_2, n_1), (n_2, n_3)\}\}$.

All nodes utilizing the SCM use a globally synchronized clock. We use $TS^{-1} : T \to N \times T$ with $TS^{-1}(t) = \tau$ where $TS^{-1}(t)$ denotes the set of timestamps at each node that relate to the global time t. Given the global clock $\forall a \in \tau : t_a = t_{global}$. Given the non-discrete form of time, there exists always a t_y that satisfies $t_x < t_y < t_z$ iff $t_x \neq t_z$. See Kopetz [2] for further details about this lemma. We split the proof of Theorem 3.1 it into three separate parts: proof sending, proof receiving, and proof any combination of sending and receiving. Yet, all three proofs use the same basic idea. Σ includes letters and each letter represents an individual node (e.g., node $n_1 \equiv a$). Each communication slot is assigned an element from $\Sigma \cup \epsilon$. The character ϵ represents an empty communication slot. The language $L = \Sigma^*$ then provides all possible communication patterns on the SCM.

Lemma 3.2 *We can represent all TDMA-based communication patterns in words of the alphabeth Σ.*

Proof Since time monotonically passes, we can assume, that no two letters x and y must be placed on top of each other, except the communication takes place at exactly the same moment (i.e., $t_x = t_y$). Two nodes can either intentionally or unintentionally communicate at the same point in time. No two nodes communicate intentionally at the same point in time (by definition of TDMA). As all nodes know the global time and all nodes participate in the TDMA-based communication (by definition of N), unintentional communication cannot happen, unless one participating node failed or it is not a member of the application. As no intentional and no unintentional communication can happen at the same point in time, no two letters must be placed on top of each other. $\square$

From 3.2 and the language L, we have to transform any encoded word to n-code. To do so, the word w (representing the communication pattern) and the length t_s of the time slot is sufficient. The sequence of characters represents the order in which the communication takes place and it can be translated into n-code.

Mapping communication patterns onto words, the language $L = \Sigma^*$ includes all possible collision free communication patterns for the sending party. One word represents one possible communication pattern. Any communication pattern is thus part of the list of all possible words. The number of possible words is the power #time-slots of #hosts and comprises all possible communication patterns.

Receiving packets from a stream of communication is similar to filtering out single letters from words. For example, given the communication word $aabbcc$[2], n_3 is only interested in data packets from n_2. Thus it filters for all letters b. Node n_1 is interested in all data packets, except its own. Thus, it filters for all letters except a. Node n_2 is only interested in the first packet sent by n_1 and in the last packet sent by n_3. Thus it filters for "$.\epsilon\epsilon\epsilon\epsilon.$".

Local data transfers, i.e., communication tuples (n_x, n_x), need not be communicated via the SCM and we assume that it must not be communicated.

Lemma 3.3 *Sending and receiving can be combined in any form. I.e., there is no t_x at which N_x is sender and receiver.*

Proof No node is sending and receiving its own data and at most one node is sending data (by definition of TDMA). Consequently there is at most one packet on the bus and at most one n-code for any point in time per node. Consequently, sending and receiving data can be combined in any form. $\square$

[2]Three nodes (n_1 to n_3) and six time-slots per period.

4 Multidimensional Schedules

Communication on a SCM is one-dimensional. Such one-dimensional schedules are often written in tables (see Section 6) and read linearly. Such schedules define $\forall\Psi \in B : 0 \leq |\Psi| \leq 1$ for which $|\Psi| = n \in \mathbb{N}$ defines the number of elements of Ψ.

The NCM allows for multidimensional schedules. We define a multidimensional schedule as the tuple $(C, N, slotting, \Theta)$ of a SCM as one for which $\exists\Psi \in B : |\Psi| > 1$ and Θ selects at most one $b \in \Psi$.

We use the mapping $\Theta : \Psi \rightarrow C$ with $\Theta(\Psi) = \Psi'$ where $\Theta(\Psi)$ is a decision algorithm that selects one broadcast communication Ψ' of the current set of communications Ψ with $\Psi' \subseteq \Psi$. The decision algorithm uses message guards as means for selection. The set G contains all guards of the distributed application. The set G^0 contains G and the guard that always evaluates to false denoted by g_0. The mapping $g : N \times S \rightarrow G^0$ with $g(n, s) = g$ where $g(n, s)$ defines the guard of s at n. We use $\pi := G \rightarrow S$ with $\pi(g) = s$ where $\pi(g)$ denotes the slot that g guards. With $\pi^{-1}(S) := \{g|\pi(g) = s\}$ where $\pi^{-1}(s)$ denotes the set of guards that guard slot s. To evaluate the guard we use the mapping $eval : G \times T \rightarrow Bool$ with $eval(g, t) = bool$ where $eval(g, t)$ is an evaluation of g at t. The evaluation of g_0 is always false, i.e., $\forall t \in T : eval(g_0, t) = false$.

Guards can use static and dynamic data of local or global scope. Static data whether global or local, is known a priori and does not require special handling. Such data elements are, for instance, the node number, total number of slots, or the cycle time. Dynamic global data also does not require special treatment. Example data elements are, for instance, the communication round or the current slot number. Dynamic local data (e.g., task output values) requires special treatment. Such data has to be propagated to $\pi^{-1}(s) := \Gamma$, if $\exists g \in \Gamma$ that uses this data. Such propagation requires communication and extends C by $\Upsilon := N \times N := \{(n_i, n_j)|n_i \in N, n_j \in N, (n_i, n_j) \notin C\}$ and increases the number of messages to be scheduled.

We have to guarantee collision-free communication on the SCM where $\exists!b \in \Psi$. In case of $|\Psi| > 1$, Θ must evaluate at most one guard to $true$ while all others evaluate to $false$ with $\exists!g \in \Gamma : eval(g, t) = true$. In other words, Θ evaluates $\bigotimes_{g\in\Gamma} g$ where $\otimes$ is the logical XOR.

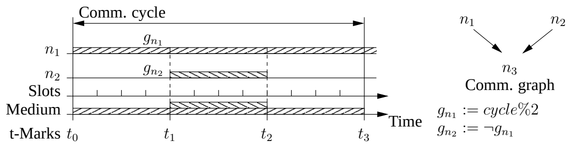

Figure 2. Multidim. schedule example.

Figure 2 shows an example of such a multidimensional schedule with guards that use dynamic global data, only. Θ evaluates to $true$ alternatively between $g(n_1, t_1)$ and $g(n_2, t_1)$. The axis *Medium* at interval $[t_1, t_2)$ shows $\Psi := \{\{(n_1, n_3)\}, \{(n_2, n_3)\}\}$. This means that at this interval two nodes are scheduled. We use this kind of visualization in the following figures.

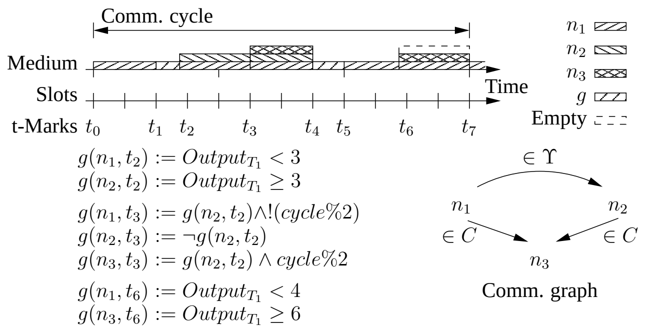
Comm. cycle
Medium
Slots
t-Marks
Time
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7
n_1
n_2
n_3
g
Empty
$g(n_1, t_2) := Output_{T_1} < 3$
$g(n_2, t_2) := Output_{T_1} \geq 3$
$g(n_1, t_3) := g(n_2, t_2) \wedge !(cycle\%2)$
$g(n_2, t_3) := \neg g(n_2, t_2)$
$g(n_3, t_3) := g(n_2, t_2) \wedge cycle\%2$
$g(n_1, t_6) := Output_{T_1} < 4$
$g(n_3, t_6) := Output_{T_1} \geq 6$
$\in \Upsilon$
n_1
n_2
$\in C$
$\in C$
n_3
Comm. graph

Figure 3. Example with guards and local data.

Figure 3 shows an example of a multidimensional schedule that uses guards with static, dynamic global, and dynamic local data. The communication cycle contains twelve slots. The distributed application consists of three nodes (n_1, n_2, and n_3) and at least one task T_1. For sake of brevity, the example does not specify task configurations except T_1 to run at n_1.

The communication graph in right lower corner of Figure 3 shows the producer and consumer relationship between the nodes with $C := \{(n_1, n_3), (n_2, n_3)\}$. Node n_1 communicates some values to n_3 and so does n_2. The lower left corner of the figure shows the guard configuration that provides the necessary information to serialize the schedule at runtime. Guard $g(n_1, t_2)$ uses local dynamic data at t_1 and t_6, consequently $\Upsilon := \{(n_1, n_1), (n_1, n_2)\}$ as $present(t_2) = \{n_1, n_2\}$. In the upper left, the media shows the resulting multidimensional schedule with for example $|\Psi| = 3$ at $n = 6$. Each node uses a different hatching and the corresponding hatching for each node is shown next to the schedule in Figure 3.

For all communication, we have to prove collision-free communication holding ($\bigotimes_{g \in \Gamma} g$). The n-code generator uses an algorithm that generates decidable n-code that holds this equation. The n-code generator adds the complement of predicate p to each $g \in \Gamma$ except the target one. The following algorithm shows this mechanism. Consequently, although the target language may be undecidable, the generated n-code is not.

```
for each $p_{g_y}$ do
begin
  for each $g_x$ in $\Gamma, g_x \neq g_y$ do
  begin
    $g_x := g_x \oplus \neg p$
  end
  $g_y := g_y \oplus p$
end
```

The multidimensional schedule shown in Figure 3 is a simple representation of an eventually serialized one. Figure 3 does not show the full semantics and dependencies between the individual messages. To specify the semantics and dependencies, we use timed automata [3, 4]. The automata shown in Figure 4 shows the exact specification of this figure. Each location contains an ϵ-switch labeled with $t = t + \delta$ and $\delta > 0$ but $t < t^{S(n+1)}$ and $t >= t^{S(n-1)}$. So, one location can switch to itself and continue communicating until it can switch to a different location.

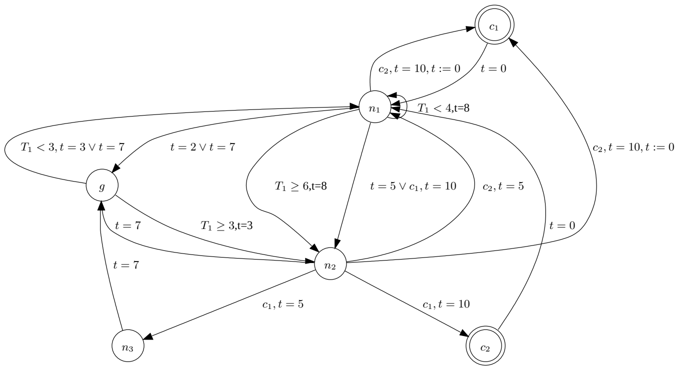

Figure 4. The schedule as timed automata.

5 Tool chain & Implementation

To prove technical feasibility, we implemented a tool chain to generate n-code and implemented a NCM on top of the commercial real-time system RTLinuxPro. The producing application is implemented by the timing definition language (TDL). TDL is a software description language evolved from Giotto [5] and is intended for timed computation. It allows for defining the timing behavior of a set of tasks. It separates the timing constraints of an applications from the functional implementation, which must be provided separately, for example, using an imperative programming language such as C.

We extract all timing and communication information from a TDL source file [6]. It provides the basic information such as tasks, task frequencies, and inter-task communication. The necessary information is converted from TDL input files into one file that includes all communication-related information. This file uses XML format and the developer can annotate it and add, for instance, additional guards and more complex constructs that cannot be expressed in TDL source.

This communication source is parsed into internal data structures and the tool finds valid communication schedules from the communication specifications using a simple branch and search algorithm. To increase options for the scheduler, the communication schedule may add further constraints for the execution schedule such as one specific task has to complete computation even before its deadline to communicate its values within this one specific slot. As a consequence of this optimization, although the communication schedule may be serializable for a SCM, it does not automatically lead to a valid execution schedule at each individual node. Thus, we perform a schedulability check for each node. As schedulability on the node is not the main concern of our work, we use earliest deadline first (EDF) [7] to verify schedulability. If the communication schedule implies a non-schedulable task execution, the tool will generate the next valid schedule and perform a schedulability check again. Eventually a serializable schedule may be found that is also schedulable at each node. Then the n-code generator outputs C-header files for each node that are compiled into the NCM. These files include the n-code. If no communication schedule can be identified or if all communication schedules are not schedulable at the nodes, the tool will prompt a message.

The NCM is implemented in RTLinuxPro (see www.fsmlabs.com). RTLinuxPro 2.1 is a hard real-time, POSIX-compatible operating system. The real-time kernel as the heart of RTLinux is built on top of the interrupt-control hardware and is responsible for the execution of real-time tasks. It treats the Linux kernel as a low priority task and implements real-time applications via kernel modules. These modules augment the kernel and the kernel executes these modules according to a selected scheduling scheme. If processing time is left (i.e., all real-time threads are idle), then the kernel will execute non–real-time tasks in the Linux environment.

The generated n-code is a program header file for the programming language C. The instructions are encapsulated in one data structure and each instruction is a tuple of four values with *(opcode, arg1, arg2, arg3)*. The n-code header file is included in the NCM and when compiled, they are linked together to the platform-specific NCM. Once, the NCM runs, it interprets the n-code data structure.

The current implementation of the NCM runs on RTLinuxPro by FSMLabs and uses the real-time communication stack called LNet. The time synchronization is implemented via the $signal(SYNC)$ command in the NCM and uses a simple time-offset correction algorithm.

6 Related Work

Several related approaches use table-driven communication. For example the time-triggered protocols developed by Kopetz et al. [8, 9, 2] use a so called message descriptor list (in TTP/C) and a round descriptor list (in TTP/A). Other systems use a similar approach (see [10, 10]). Implemented as message descriptor list, the schedule specifies exactly when a node has to send a certain message and when messages from the other nodes have to be received. In combination with this, a task descriptor list describes the cyclic scheduling of application tasks. This list specifies the instances of time of starting and stopping tasks. At runtime the dispatcher reads the table structure and executes one row after the other.

TTP allows for assigning slots to virtual nodes and several physical nodes can be the same virtual node. This bypasses the guard system and allows $\forall \Psi \in B : 0 \leq |\Psi|$. However, the function that evaluates, which of the physical nodes is allowed to send does not satisfy equation (**??**). This allows for programming faulty schedules where Θ evaluates to $true$ for more than at most one guard.

Other protocols aim at more flexibility and communicate their schedules at the beginning of each communication cycle. For example, flexible time-triggered (FTT) Ethernet [11] aims at time-triggered communication with operational flexibility and features centralized scheduling and master/multi-slave transmission control. The central master calculates the schedule for the next communication round and the master/multi-slave transmission control allows for communicating this schedule to several nodes at once and provides better efficiency. Another similar approach is Ethernet PowerLink[3] [12] which is an isochronous protocol based on standard Ethernet. It aims at deterministic communication, low cycle times, and asynchronous communication and uses a master/slave transmission control. Within each period, the master prompts nodes and they respond only when they are prompted. Additionally, the master can invite one node to communicate after the cyclic period has elapsed and before the communication cycle starts.

Other protocols are token driven such as real-time Ethernet (RETHER [13]). RETHER regulates access to the network via a token. The token circulates between the member nodes of the real-time set and the non–real-time set. Each node is allowed to hold the token for an interval of time during which it can access the communication medium. Each real-time process specifies its required bandwidth and the sum of the requirements of all real-time processes is the bases to compute the token behavior (e.g., the token rotation time). The token rotation time implicitly contains all communication constraints and as it affects all nodes of the real-time set, it cannot changed in an arbitrary way.

Much work has been accomplished in making TDMA schemes more flexible. However, they all are limited to predefined one-dimensional schedules. At the beginning of

[3]The following applies to version one of the protocol, since version two is not available for public discussion.

the communication round, the controller knows the schedule and will execute it without variation. Multidimensional schedules introduced in this paper are not.

7 Conclusion

Shared communication media (SCM) introduce the problem of collision when multiple participants perform a write operation at the same time. Common solutions in the time-triggered domain provide linear and one-dimensional schedules that specifies when which participant is allowed to perform write operations on the SCM. Such schedules are usually static in case they need to be verified offline (e.g., for real-time applications) or they are at least known a priori (e.g., at the beginning of the communication cycle) and for each moment within this cycle at most one participant is scheduled for a write operation.

In our work, we introduce a description method for multidimensional schedules that break with traditional approaches and allow for several writers to be scheduled for a single communication slot at the SCM and use an evaluation function Θ to determine the actual writer while the communication schedule is executed.

The proposed system is more expressive than common ones (see Section 6) but it can run on arbitrary communication hardware. It provides more flexibility for the scheduler as it allows in-cycle changes of the communication pattern.

One area of application of multi-dimensional schedules is flexibility in real-time systems for which messages of nodes can be re-scheduled at runtime or communication of shadow nodes can be omitted in favor of lower priority messages such as status reports. We implemented and tested the system on RTLinuxPro using Ethernet as SCM.

8 Acknowledgments

We would like to thank Josef Templ, Johann Edtmayr, and Gregor König for their valuable discussions on this topic.

References

[1] G. Coulouris, J. Dollimore, and T. Kingberg, *Distributed Systems: Concepts and Design*. Queen Mary and Westfield College, University of London, 1996.

[2] H. Kopetz, *Real-time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[3] R. Alur and D.L.Dill, "A Theory of Timed Automata," *Theoretical Computer Science*, vol. 126, pp. 183 – 235, 1994.

[4] R. Alur and P. Madhusudan, "Decision problems for timed automata: A survey," in *Formal Methods for the Design of Real-Time Systems* (M. Bernardo and F. Corradini, eds.), no. 3185 in LNCS, Springer, 2004.

[5] T. Henzinger, C. Kirsch, M. Sanvido, and W. Pree, "From control models to real-time code using Giotto," *IEEE Control Systems Magazine*, Feb. 2003.

[6] J. Templ, "TDL Specification and Report," Tech. Rep. T002, Computer Science, University of Salzburg, 2004.

[7] G. Buttazzo, *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 2000.

[8] W. Elmenreich, W. Haidinger, R. Kirner, T. Losert, R. Obermaisser, and C. Trödhandl, "TTP/A smart transducer programming — a beginner's guide," Research Report 33/2002, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.

[9] S. Eberle, C. Ebner, W. Elmenreich, G. Färber, P. Göhner, W. Haidinger, M. Holzmann, R. Huber, R. Schlatterbeck, H. Kopetz, and A. Stothert, "Specification of the TTP/A protocol v2.00," Research Report 61/2001, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2001.

[10] G. Menkhaus, M. Holzmann, and S. Fischmeister, "Time-triggered Communication for Distributed Control Applications in a Timed Computation Model," in *23rd International Digital Avionics Systems Conference (DASC'04)*, IEEE Press, 2004.

[11] P. Pedreiras, L. Almeida, and P. Gai, "The FTT-Ethernet protocol: merging flexibility, timeliness and efficiency," in *Proc. of the 14th Euromicro Conference on Real-Time Systems*, pp. 134 –142, IEEE Press, June 2002.

[12] BERNECKER + RAINER Industrie-Elektronik Ges.m.b.H., *Ethernet Powerlink: Data Transport Services*, 5 ed., Sept. 2002. White-Paper.

[13] C. Venkatramani and T. Chiueh, "Design, implementation, and evaluation of a software-based real-time ethernet protocol," in *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pp. 27–37, ACM Press, 1995.