Time-triggered Communication for Distributed Control Applications in a Timed Computation Model

Guido Menkhaus, Michael Holzmann and Sebastian Fischmeister, University of Salzburg, Austria

Abstract

Distributed real-time control applications consist of sets of tasks that interact with the physical world through sensors and actuators and are executed on a dispersed set of locations that are interconnected by a communication subsystem. Timeliness and safety requirements of the application demand deterministic execution of tasks and predictive communication. Deterministic and predictable systems can be build, if upper bounds for processing and communication latencies are known and event arrivals have deterministic distributions.
In this paper we describe the timing definition language (TDL) system architecture implementing time-triggered computation and time-triggered communication. The TDL system implements the timed computation model and its architecture consists of two parts: TDL-Exe (for time and value deterministic execution of tasks) and TDL-Com (for predictive communication of values). The paper presents TDL-Exe and describes implementation details of TDL-Com.
Keywords: real-time control systems, timed computation model, time-triggered communication, system architecture

1 Introduction

Most modern control applications are implemented in software, dedicated to perform specific tasks and interacting with the physical world through sensors and actuators. Physical and software processes differ conceptually in their treatment of the role of time [1]. Physical processes evolve in real-time and software processes evolve in so-called soft-time. Soft-time coincides with real-time only at the instances of input and output activities. Timed real-time programming deals with the activities of mapping soft-time to real-time [2].
Real-time control systems are often implemented as distributed systems where a set of computational nodes is interconnected by a communication system. The dispersal of application resources, organizational issues, and fault-tolerance mechanisms are design rationales for distributed systems. In such a system, each node executes a set of tasks contributing a specific functionality to the overall control system. Individual tasks cooperate and results of one task may require to be communicated to tasks located at different nodes. The design of a real-time distributed control system can broadly be divided into global and local design issues [3]:

- *Global design decisions.* Global design decisions deal with activities that are relevant to more than one computational node (e.g., communication concerns). These activities must be coordinated among the computational nodes and require to work consistently together towards the goal of the control system.

- *Local design decisions.* Local design decisions are concerned with activities within a single computational node (e.g., the set of tasks released on this node need to be invoked and executed in a consistent and synchronized way).

The correctness of a real-time control system depends on the computed values and on the point in time, at which these values are available. A key problem of distributed real-time systems is the timely interaction between the system and the environment while maintaining consistency and correctness of data. To achieve consistency in

distributed systems in the time and value domain, the system must be made deterministic. Being deterministic is mandatory for software systems that control physical systems that are ruled by deterministic physical laws [4].

- *Non-determinism.* A non-deterministic system does not have a unique output sequence to a given input sequence. An external observer cannot consistently predict the behavior of the system. In a deterministic (i.e. predictable) system the development of future states of the system can be predicted.
- *Value determinism.* A systems is said to be value deterministic, if the same sequence of inputs produces the same sequence of outputs.
- *Time determinism.* If the system produces for the same sequence of inputs the same sequence of outputs at always the same time, it is time-deterministic.

Determinism in time-triggered systems is accomplished by introducing a logical execution time (LET) and a logical computation time (LCT) for each task as well as a logical transmission time (LTT) for communicating messages between computational nodes.

- *Logical execution time.* Input and output ports define logical points of interaction between tasks. The start of the LET marks the point in time when the values from input ports to a task are read. The end of the LET marks the point in time when the results of the computation of a task become available at the output ports to other tasks or actuators. Even if the output of a task became available prior the end of the LET, the output values will not be released prior to the expiration of the LET.
- *Logical computation time.* The LCT specifies a fixed time interval in real-time in which the task is active. The task is schedulable, if there is enough soft-time available within the interval to execute the task. According to a scheduling scheme, the task starts after it has been released, may be preempted, but resumes and completes its execution before the LCT has elapsed. The release and the termination of a task are time-triggered events emitted at the start and the end of the LCT. The LCT of a task is always greater or equal to its worst case execution time (WCET).
- *Logical transmission time.* The LTT is determined by a time-triggered communication schedule. The schedule defines when the communication system transfers values from one computational node to another and when the next transmission will take place. The LTT of transferring a value from one node to another is always greater or equal to the worst case communication time (WCCT) [5].

The LTT is important for the inter-node communication and modeled in the global design decisions. The LCT is determined during the local design decisions and deals with computational resources on a single node.

- *Centralized Application.* For control application executed on a single computational node, the length of the LET equals the length of LCT (see Figure 1). Local intra-node communication is instantaneous, which means that reading inputs and writing output happens conceptually in zero time.
- *Decentralized Application.* To sustain the principle of instantaneous communication in a distributed control application, the LET consists of the LCT and the LTT (see Figure 2).

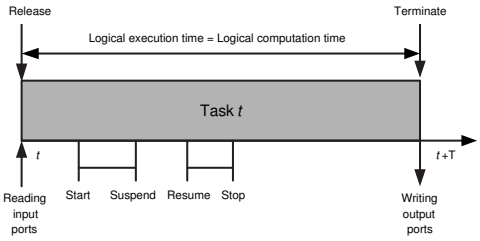

Figure 1. Logical execution time in a centralized application.

Information necessary to determine the LET of a task is the invocation frequency that the control law of the controlled object requires. The length of the

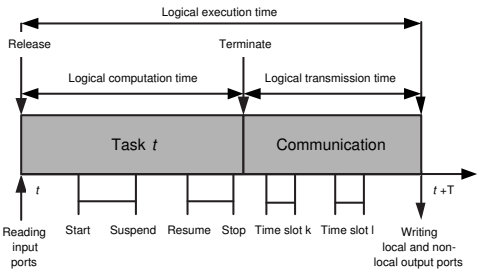

Figure 2. Logical execution time in a decentralized application.

LCT must be larger than the WCET. The schedulability test of the TDL compiler verifies this property. The length of the LTT is determined by the WCCT, which represents the upper bound required to transmit a message from the sender to the receiver over the network.
In this paper we present the TDL system architecture that allows for the design of control applications using time-triggered computation and time-triggered communication. The TDL [6] bases on the concept of a fixed LET of tasks. The communication system of TDL provides the programmer with timing abstraction for implementing distributed control systems with hard real-time constraints. For an end-to-end time-triggered approach, it is necessary to provide time and value determinism for the global and local design.
The remaining of the paper is structured as follows: Section 2 presents the motivation of the work. Section 3 discusses related work such as TTP and FlexRay. Section 4 presents the TDL system architecture for centralized and decentralized system. Section 5 provides details about the implementation and characteristics of the decentralized system are discussed in Section 6. Section 7 concludes the paper.

2 Motivation

The software implementation of distributed real-time control applications must be predictable yet flexible. Predictable, because hard real-time applications are time and safety critical. Flexible, because all tasks do not have to be specified fully pre run-time. Non-deterministic system require over-sizing of computing and communication resources to avoid time delays in worst load case situations: for example in dynamic control systems, in which the upper limit of processing and communication latencies is unknown and the event or task arrival have non-deterministic distributions [7].
Event-triggered systems offer more choices for scheduling task computation and communication. However, it is difficult to build deterministic systems utilizing event-triggered communication together with jitter introduced by communication-error correction.
In time-triggered systems, communication and computation of tasks are predictable. Predictability implies deterministic temporal system behavior under the imposed timing and functional constraints. For those systems the functional timing requirements are always met. Static schedules are used to plan task executions on each computational node. But for this, the release time of tasks must be known a priori. Time-triggered communication provides predictable message transmission. Static schedules drive the communication system and determine the timely transmission of messages.

3 Related Work

A number of time-triggered systems have been devised in the context of distributed control applications for safety critical hard real-time systems.
The time-triggered protocol (TTP) [8] provides time-triggered communication of messages and static cyclic scheduling of application tasks. A member of the TTP family is the time-triggered protocol TTP/C, intended for safety critical hard real-time applications, and TTP/A, intended for low-cost field bus applications. TTP/C provides distributed fault tolerant clock synchronization, error detection, membership service, and redundancy management. Cyclic scheduling of application tasks is described with the task descriptor lists (TADL). The TADL specifies the time of starting and stopping a task and the WCET of a task. It describes the temporal behavior of the system before the systems starts. The finishing time of a task is determined by the number and length of

preemptions. Results are immediately available to other tasks on the same node after the finishing time.
A consortium of companies supports and promotes the FlexRay Communications System [9, 10], which is a communication infrastructure for high-speed control systems targeting the automotive domain. The communication cycle of FlexRay consists of a static and a dynamic segment. Each communication cycle starts with the static segment. Similar to the time-triggered protocol, all communication is divided into slots, which the developer assigns to individual nodes. Following the static segment, the dynamic segment is intended for aperiodic messages such as burst transmissions or diagnosis information. The dynamic segment utilizes the flexible time-division multiple access (FTMA) protocol ByteFlight [11] that uses message identifiers as means for messages scheduling. Applications using FlexRay can be implemented with OSEKtime [12]. OSEKtime is responsible for starting tasks according to a periodic task execution scheme (similar to the TADL) and it monitors the task deadlines. The time-triggered tasks can preempt each other.
TTP and Flexray provide time-triggered communication that ensures time determinism on a global level. However, on a local level, i.e., on a single computational node, time determinism and value determinism cannot be guaranteed. We present the TDL system architecture for time-triggered computation and communication that aims at value and time determinism on a global and local level.

4 TDL System Architecture

The design process for a TDL control application can be split into a local design process, targeting a centralized single processor solution, and a local and global design processes for distributed control applications. The run-time environment for a centralized processor TDL control application is the TDL runtime environment (TDL-Exe), consisting of the E machine [13]. The TDL-Communication environment (TDL-Com) complements it for distributed control application.
We first present the activities of the local design process and the local runtime environment before discussing the global design process and the TDL-Com.

4.1 Centralized System

The following steps lead to a TDL application for the centralized, single processor solution (see Figure 3).

1. *Task-Set Declaration.* A TDL program can be modeled using a visual task modeling tool [14]. The essential idea of the timed computation model is time-triggered cyclic computation in which the LET, the LCT, and the WCET describe the timing behavior of a task.

   The most important programming abstractions of TDL are modules, modes, tasks, and ports: The highest-level programming construct is a module. A module declares a set of modes, which specify sets of tasks and other activities that are executed periodically and in parallel. A TDL module can only be in one mode at a time, but can change from one mode to another at the end of a period. A module may have a start mode. If a module has a start mode it is an executable TDL program and the application described by the TDL program starts executing in this mode. A task has a set of input ports, output ports and a set of drivers that handles the data for the ports. Ports are logical points of interconnection between tasks and modes. Task drivers copy values from an output task port to an input task port, and there are drivers for sensor readings and actuator updates. Mode drivers read sensors and update mode ports, which are a subset of the task output ports.

2. *Task Timing Definition.* A TDL mode specifies the invocation period, i.e., the length of one computation cycle. The LET of a task is determined with respect to the invocation period of the mode to which the task is assigned. It is calculated by dividing the invocation period of the mode by the tasks frequency.

   Task drivers, sensor drivers, and actuator drivers differ in the fact that task and sensor drivers are called at the beginning and the end

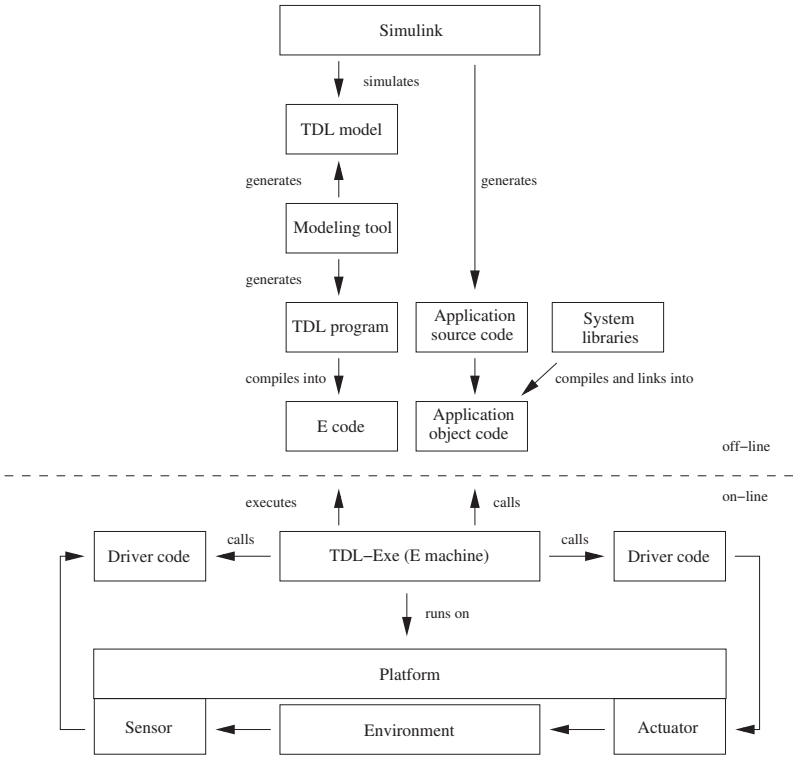

Figure 3. Centralized TDL system architecture for a single processor application.

of the LET, whereas actuator drivers are treated like tasks having their own invocation period.

If the set of tasks is schedulable, the TDL program is time-safe. However, time safety depends on the correct analysis and calculation of the WCET of each task for the given platform.

3. *TDL Program.* The visual modeling tool generates a TDL program.

4. *TDL Model.* On the basis of a TDL program, a TDL model generator produces a Simulink model [15]. Simulink is designed for the simulation and modeling of control laws. For modeling, Simulink provides a graphical user interface for building models as block diagrams. It includes a comprehensive block library of sinks, sources, components, and connectors. The Simulink model that results from the TDL program (the TDL model) can be simulated to validate the timing and functional behavior. This is especially helpful, if the application code has been modeled in Simulink.

5. *E code.* The TDL compiler compiles a TDL program into E code for the E machine [16]. The E code is a platform-independent assembler language that targets the E machine. The E machine executes E code that ensures the timing consistency of the task and driver executions. The E code consists of a small set of instructions for basic control flow and processing, that allows for synchronous driver calls, task scheduling and initializing the execution of a set of E code instructions at some point in time in the future.

6. *TDL-Exe (E machine).* The TDL runtime part of the architecture is represented by the TDL-Exe consisting of the E machine. The E machine is a virtual machine that executes the platform-independent E code and calls the platform-dependent application code.

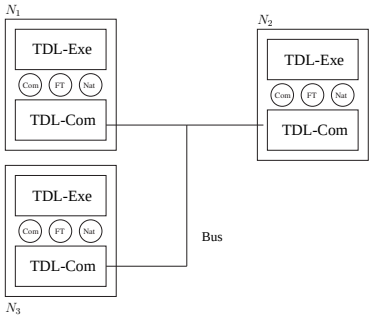

Figure 4. Distributed TDL System architecture.

4.2 Decentralized System

The TDL system architecture for a decentralized (or equivalently distributed) real-time control system is modeled by a set of TDL nodes that are interconnected by a real-time communication system. Each node consists of a TDL-Exe and a TDL-Com part (see Figure 4).
The TDL-Exe is not concerned with the global design of the communication between nodes, because the synchronization is done implicitly by the timing definitions of the tasks on each node. Each task provides the data items that need to be transmitted to a different node. However, the timing definitions of tasks on each node are designed in mutual agreement with tasks running on other nodes. TDL-Com supplies a data communication system that allows for transmitting values from task output ports of a TDL program to input ports of a task running on a different node. TDL-Com uses a time-triggered communication subsystem to transmit data. It works autonomously: sending and receiving of messages happens without any interaction from the application program.

4.3 TDL-Com Interfaces

TDL-Com exposes three interfaces to TDL-Exe (see Figure 4):

1. *Com Interface.* The Com interface mediates between TDL-Exe and the communication subsystem of TDL-Com. It allows the TDL-Exe to submit and retrieve data values via drivers that are connected to input and output ports. Input and output ports define logical points of interaction between tasks and the Com interface. The drivers copy values of output ports of a task to the input ports of the Com interface, which forwards them to the communication network interface. The Com interface has a set of output ports, whose values are destined for input ports of tasks, which are then retrieved via tasks drivers.
2. *FT Interface.* The FT interface allows for access to information related to fault tolerance. Redundantly produced and communicated values have status fields, which provide information, for example, on the number of active replicas, the status of fault-tolerant communication, or the confidence values of fusion algorithms.
3. *Native Interface.* The native interface allows for access to platform specific services, such as the error counter of the CAN controller.

4.4 TDL-Com Architecture

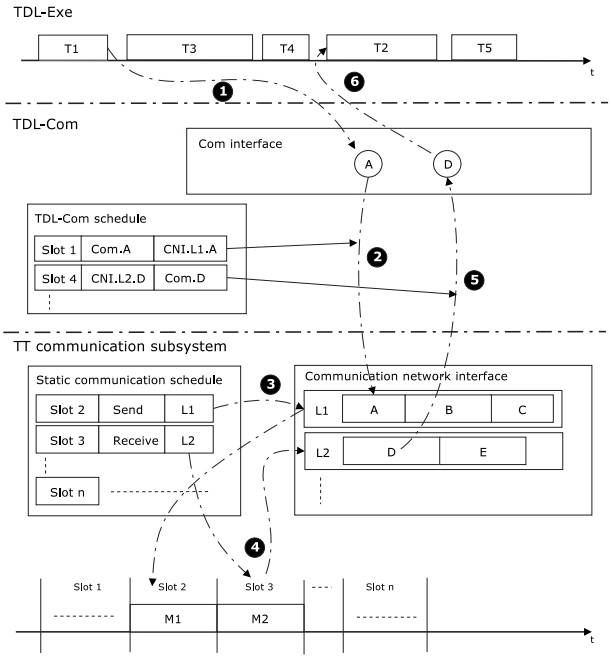

Figure 5. Distributed TDL Architecture

Figure 5 illustrates the TDL system architecture of a network node of a distributed TDL application. TDL-Exe provides the runtime environment which

executes TDL applications. TDL-Com builds on a time-triggered communication subsystem. The communication subsystem of each network node processes autonomously all communication activities. The communication network interface is the interface between the communication controller and the TDL-Com layer. The communication network interface contains messages, which are sent and received by the communication subsystem. TDL-Com reads data from the communication network interface and passes it to the TDL-Exe layer. TDL-Com writes data to the communication network interface to send data submitted by the TDL-Exe layer. The interface between TDL-Exe and TDL-Com is the Com interface that allows for data exchange via drivers and ports.

The communication schedule list determines the temporal behavior of the communication subsystem. Each node stores such a list. This list specifies at which point in time (time slot) a node is allowed to send messages and at which point in time it will receive messages.

Figure 5 shows an example of sending and receiving a message. Task `T1` produces a value A and submits it to TDL-Com in step 1. In Step 2, TDL-Com copies value `A` to location `L1` within the communication network interface (it will be communicated in time slot 2 with message `M1`). When sending message `M1` in Step 3, the communication subsystem reads the data from the indicated location `L1` in the communication network interface, generates the message `M1` and transmits it. In Step 4, the communication system receives message `M2` during time slot 3. It stores its data at the indicated location `L2`. TDL-Com copies the data item `D` from the communication network interface to the value *D* within the Com interface (Step 5). Finally, task `T2` consumes the value in Step 6.

Temporal Synchronization of TDL-Com. As described above, there are temporal dependencies between the TDL-Exe, TDL-Com, and its subsystems. For example, a value has to be produced, copied, and marshaled before its transmission. The dependencies result from the fact that the time-triggered communication subsystem runs autonomously and the progression of time drives the timing of the whole node. The activities of the TDL-Com and the TDL-Exe layer need to be synchronized to the activities of the communication subsystem.

From the point of view of TDL-Exe, communicating values between tasks (reading input and writing to output ports) is transparent in centralized as well as in the decentralized applications. However, in centralized applications, the LCT determines the LET of a task (i.e., LCT = LET). In decentralized applications, if values of output ports of this tasks need to be transmitted to a task on a different node, the LET consists of the LCT plus the LTT (i.e., LET = LCT + LTT). The LTT always succeeds the LCT. LET includes the transmission time of a message. Consequently, from the point of view of TDL-Exe, inter-node communication happens conceptually in zero time (i.e., transparent to the single-node computation model).

TDL-Com and communication subsystem synchronize their timing via a communication schedule list. TDL-Exe is implicitly coordinated with the communication subsystem by the timing definitions of the tasks on each node. However, from the communication schedule list and the local design of timing definitions of the tasks may raise global design restrictions (scheduling restrictions), that need to be resolved.

4.5 TDL-Com Toolchain

The construction of the communication schedule list of the communication subsystem is an off-line activity. The schedule is then used on-line (during runtime) to ensure predictive communication. Figure 6 illustrates the off-line activities of the TDL-Com toolchain.

- *Off-line.* The off-line part of the toolchain determines the communication requirements for the distributed application and generates a global communication schedule list and the TDL-Com schedules for each network node.

  To generate the global communication schedule list, the TDL-Com Compiler determines the communication requirements of the whole decentralized application. It scans TDL programs and modules and detects accesses to

non-local data ports. Access to remote ports results in communication requirements. Optimization of communication can by achieved, for example, by packaging several data items within the same invocation period into a single message.

A TDL-Com Compiler plug-in maps the communication requirements into the format of a vendor-specific bus scheduling tool, which generates the network schedule for a specific communication platform (e.g., TTP, FlexRay, TTCAN). The plug-in reads and analyzes the generated network schedule and provides the TDL-Com Compiler with information to generate the TDL-Com schedule lists for each node.

- *On-line.* The TDL-Com layer provides interfaces to TDL-Exe and to the communication subsystem (see Figure 5). It copies values to be transmitted between the Com interface and the communication controller interface and vice versa, marshals values into communication frames and supports voting for fault tolerance. The behavior of the TDL-Com layer is statically pre-defined before runtime. The activities are cyclically repeated.

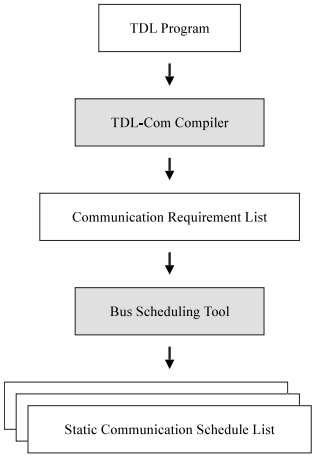

Figure 6. Overview TDL-Com tool chain

5 TDL-Com Prototype Implementation on Top of CAN

The prototype implementation of the TDL-Com uses time-triggered CAN (TTCAN) [17] as communication protocol. TTCAN extends the CAN protocol by providing time-triggered communication via the standard physical CAN link. CAN [18] is a mature standard and widespread in the field of automation as well as in the automotive field.

We implemented a proprietary version of the TTCAN protocol in software which we call software TTCAN (sTTCAN). The reasons were as follows: (1) It allows for adaptation to the needs of the TDL-Com prototype implementation, (2) there were no embedded boards equipped with TTCAN controllers available, (3) the TTCAN chip was still in evaluation status, (4) lack of a supporting tool chain. A benefit of the TTCAN software implementation is that it can be run on standard embedded boards and ECUs without the need to change or modify the hardware to profit from reliable, time-triggered communication. Like the TTCAN implementation by Bosch [19], we support transmitting sporadic messages within dedicated time slots. The objective of our implementation is to provide an implementation for the OSEK/VDX operating system.

The TDL-Com implementation bases on the Motorola MPC 555 Power PC derivate and the OSEK/VDX operating system. The processor chip already integrates a variety of common I/O (e.g., two CAN controllers). KANIS OAK EMUF boards, hosting the Motorola MPC 555 processor, are the target hardware for our implementation. Each network node consists of one KANIS OAK EMUF board, the boards are interconnected via a CAN bus link. Each board features two physical CAN links including bus drivers and on-board connectors.

The prototype is implemented under OSEK/VDX using OSEKWorks, an OSEK implementation by WindRiver and the development environment supplied by WindRiver [20].

5.1 Clock Synchronization

Time-triggered communication requires a synchronized time base among the participating nodes to provide a time division multiple access (TDMA) bus arbitration scheme. TDMA allows for a number of nodes to access a single transmission channel without interference by allocating unique time slots to each node within each channel.
The sTTCAN implementation of clock synchronization is inspired by TTCAN [17]. TTCAN uses a master-slave clock synchronization scheme based on the idea of the TTP/A fireworks protocol [21]. A dedicated station, the master, periodically sends a synchronization frame s, which other nodes use to synchronize their local clocks to the clock of the master.
We describe the clock synchronization algorithm using the notation of event-recording automata [22]. Timed automata have been introduced to model the behavior of real-time systems. They augment finite state automata with a set of clocks. Event-clock automata are timed automata that correlate the value of clocks and the occurrence of events and maintain their correspondence.
An automaton consists of a set of locations V and a finite set of event-recording clocks X. We write x^l to denote that the clock $x \in X$ is assigned to a location $l \in V$. The infinite word $w = (s, t_0)(s, t_1)\ldots$ is emitted by the master node and is the input to the automaton that describes the clock synchronization system. $\frac{1}{p_i}$, with an equidistant period $p_i = t_i - t_{i-1}$ for all i is the frequency of the occurrence of s. The content of message s is the value of the clock of the master node at times $t_0, t_1, \ldots$. We write x_s^l to denote the time of the clock x at location l at reception of synchronization message s of the master node. We write x_s^m to denote the value of the clock of the master node at the time when sending the message s. At reception of message s at location l, we compute the deviation d_s^l between the value of the master clock x_s^m and the value of the clock at location l (we assume an instantaneous transmission of messages):

$$d_s^l = v(x_s^l) - v(x_s^m).$$

Clock values are measured in ticks. They are split into a macro tick and a micro tick part. We write $\langle v(x) \rangle$ to denote the micro tick part of a clock and $\lfloor v(x) \rfloor$ to denote the macro tick part of $v(x)$, such that $v(x) = \lfloor v(x) \rfloor + \langle v(x) \rangle$. Macro ticks t_M are counted in our implementation in OSEK system timer ticks. A macro tick is composed of a specific number m of micro ticks t_m (CPU timer ticks), such that $t_M = m_0 \cdot t_m$ at time t_0.
To synchronize a clock x_s^l, the nominal number of micro ticks for every macro tick is increased to slow down the clock and it is decreased to speed it up. To compensate for a clock deviation of d_s^l in period p_i in the next synchronization period, the number of micro ticks $t_{m_i,corr}$ need to be added to the nominal number of micro ticks that make up a macro tick during the next period p_{i+1}. $t_{m_i,corr}$ is computed by dividing the deviation by the length of period p_i. The $sign(d_s^l)$ indicates a positive or a negative clock deviation.

$$t_{m_i,corr} = sign(d_s^l) \frac{d_s^l}{\lfloor p_i \rfloor}.$$

The number of micro ticks that make up a macro tick for the next period p_{i+1} is computed as

$$t_M = (m_{i-1} + t_{m_i,corr})\, t_m.$$

with $m_i = m_{i-1} + t_{m_i,corr}$. $t_{m_i,corr}$ is usually a fraction number and the value is split into an integral and a fractional part. The integer part is immediately corrected as shown above. The fraction part is accumulated and corrected within the current period by adding or subtracting one more t_m as soon as the absolute sum exceeds 1.
Valuation of clocks uses the time stamping mechanism of the MPC555 CAN controller. It automatically generates a time stamp at the time of start of frame [18] of an incoming message. The usual method of generating time stamps with interrupts is imprecise because of interrupt latencies. Furthermore, using interrupt service routines to generate the time stamps causes delay of running tasks and increases CPU load, which might negatively influence the temporal predictability of the system.
With the current implementation of the clock synchronization algorithm we achieve an accuracy of 20 μs, i.e., a maximum deviation of +/- 10 μs of the slave clocks from the master clock. The CAN

bus is currently run at 100kBit. Increasing the transmission speed of the CAN bus from 100kBit to 1MBit and thus decreasing the length of the bit cell on the bus and improving the synchronization algorithm by adding rate correction of the local clocks we expect to improve the overall accuracy down to 5 μs.

5.2 Time-Triggered Communication

The clock synchronization algorithm of sTTCAN coordinates the OSEK system timer of the OSEK/VDX operating system of the involved nodes, which allows for the implementation of time-triggered communication based on the CAN bus using solely OSEK API functions.
Time-triggered communication is done in a periodically, recurring pattern, the so called communication round [21]. The synchronization message that is periodically sent by the master node, indicates the start of a new communication round. A communication round is subdivided into time slots, in which only one dedicated node is allowed to access the network and to transmit messages. The length of time slots is fixed and is measured in OSEK system timer ticks (macro ticks, t_M) and denotes the LTT for the message sent in the time slot. After reception of the synchronization message (at the beginning of a communication round), the nodes start transmitting messages according to statically predefined communication schedules that assigns communication time slots to nodes (see Figure 7).

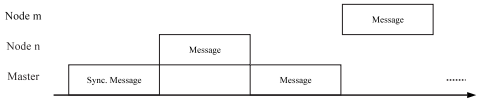

Figure 7. A communication round.

The structure of the static communication schedule list is inspired by the message descriptor list (MEDL) introduced by Kopetz [8]. The schedule is locally stored at each node and it defines for the local node the time slots that it may use to access the network to transmit messages.
Sending messages is implemented as follows: OSEK alarms are successively set up for each entry in the communication schedule list indicating the time slots for the local node. Whenever an alarm fires, the alarm-callback routine triggers the CAN controller to send the message buffer denoted in the current schedule entry. The next alarm is set-up according to the next entry in the communication schedule list until the list is completely processed. With the beginning of the next communication round the processing of the communication schedule list restarts from the beginning.
Table 1 shows the structure of the communication schedule for one node with two entries: The node sends a four byte message contained in buffer one in the time slot which starts at t_M = 2 (time slots are counted always relative to the start of the communication round). It then sends a two byte message contained in buffer two in the time slot which begins at t_M = 15.

Pos.	Alarm	#Buffer	Length
1	2	1	4
2	15	2	2

Table 1. Structure of the communication schedule.

Receiving messages is done automatically by the CAN controller, no software interaction is required. The received data is stored within one of the 16 buffers of the CAN controller. The TDL-Com stack fetches the data from the message buffers and copies it to the Com interface where TDL tasks can access it.
Verification of the reception time of messages, done by protocols for safety-critical applications such as TTP or Flexray, has been omitted in the current implementation for performance reason.

5.3 TDL-Com Com-layer

The prototype implementation of TDL-Com consists of a set of specific drivers and supports only the Com interface yet. The message buffers of the CAN controller are directly used as communication network interface (see Figure 5). For communicating data to a receiving node, TDL-Exe executes a set of specific drivers that are declared in the E code of the TDL program that is executed on

the sending node. These drivers copy the values of output ports of a tasks directly to CAN controller message buffers. Drivers declared in the E-code of the TDL program on the receiving nodes get these messages, extract the values, and copy the value to the correct task input ports.
The timing of the communication subsystem and the E machine rely both on the local OSEK system timer and on the fact that the OSEK system timer of all involved nodes are synchronized. If an input port of a task receives a value from a different node timely arrival of the value at TDL-Com is ensured.

6 Discussion

The TDL system architecture allows for the design of deterministic control applications using time-triggered computation and time-triggered communication. It provides the programmer with timing abstraction for implementing distributed control systems with hard real-time constraints. However, the current approach of the TDL-Com implementation has a drawback: It leads to a runtime behavior of the application which exhibits unbalanced CPU and network load. The TDL semantic defines, that the first period of all tasks within a mode period (hyper period) starts simultaneously at the beginning of the mode period [5]. In a centralized TDL application, the period of all tasks equals the LET and computation of the tasks can be equally distributed over the hyper period of the application. In a decentralized system, the LET model defines for each task that the interval of computation (defined by the LCT) coincides at the beginning of the task period whereas communication activities (defined by the LTT) succeeds computation and thus occurs towards the end of the period. Because the periods of all tasks of a TDL application start simultaneously at the beginning of the hyper period, there is a peak in computation (higher CPU load) at the beginning of the hyper period, whereas communication (network traffic) dominates at the end of the period. To balance the CPU and the network load and to distribute it equally over the hyper period, we currently work on new approaches, which overcome this drawback of the current model on which TDL-Com is based on.

7 Conclusion

The timing definition language (TDL) provides programming abstraction for the implementation of control systems. The TDL system architecture consists of a runtime environment for deterministic execution of tasks and predictive communication of messages for distributed control applications. Determinism and predictability are requirements for safety-critical and fault-tolerant systems. The timed computation and communication model bases on these principles.
In this paper, we introduced TDL-Com, a time-triggered communication system. The TDL-Com toolchain produces static scheduling lists, which describe the communication behavior of the whole distributed application. A globally synchronized time base is the basis for interconnected tasks. The tasks that run on different computational nodes need to be coordinated in the time domain to operate in a consistent and highly synchronized manner.

8 Acknowledgments

We thank the team of the Software Research Lab of the University of Salzburg for their help in developing and implementing the ideas presented in this paper.

References

[1] E.A. Lee, S. Neuendorfer, and M.J. Wirthlin, 2003, Actor-Oriented Design of Embedded Hardware and Software Systems. *Journal of Circuits, Systems, and Computers*, 12(3):231 – 260.

[2] C.M. Kirsch, 2002, Principles of Real-Time Programming. *LNCS*, 2491.

[3] E. Fuchs and D. Millinger, 1998, Task set design tools for an embedded distributed control system. In *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, pages 182–188. ACM Press.

[4] N. Halbwachs, 1997, *Synchronous Programming of Reactive Systems*. Kluwer.

[5] B. Horowitz T. A. Henzinger, C. Kirsch, 2001, Giotto: A Time-triggered Language for Embedded Programming. In *Proceedings of EMSOFT 2001*, LNCS. Springer.

[6] J. Templ, 2003, TDL Specification and Report. Technical Report C059, Computer Science, University of Salzburg, Mar.

[7] B. Ravindran, 2002, Engineering Dynamic Real-Time Distributed Systems: Architecture, System Description Language, and Middleware. *IEEE Trans. Softw. Eng.*, 28(1):30–57.

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

[9] R. Belschner, J. Berwanger, C. Ebner, H. Eisele, S. Fluhrer, T. Forest, T. Führer, F. Hartwich, B. Hedenetz, R. Hugel, A. Knapp, J. Krammer, A. Millsap, B. Müller, M. Peller, and A. Schedl, 2002, *FlexRay Communications System — Requirements Specification*. BMW AG, DaimlerChrysler AG, Robert Bosch GmbH, General Motors/Opel AG4, April. Version 2.0.2.

[10] Flexray website, 2004. `www.flexray.com`.

[11] BMW AG. *ByteFlight Specification*. available at `www.byteflight.com`.

[12] OSEK Group, 2001, *OSEK/VDX Time-triggered Operating System Specification, Version 1.0*, July.

[13] T.A. Henzinger and C.M. Kirsch, 2002, The Embedded Machine: predictable, portable real-time code. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 315–326.

[14] G. Stiegelbauer and A. Werner, 2004, Integration von TDL und Simulink: Komplexitätsreduktion beim Entwurf von eingebetteter Software. In *Proceedings of the 2nd Workshop: Automotive Software Engineering*, Ulm, Germany.

[15] MathWorks, www.mathworks.com. *Simulink*.

[16] C. Kirsch T. A. Henzinger, 2002, The Embedded Machine: Predictable, Portable Real-Time Code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[17] T. Führer, B. Müller, W. Dieterle, F. Hartwich, R. Hugel, and M. Walther, 2000, Time Triggered Communications on CAN (Time Triggered CAN - TTCAN). In *Proceedings 7th International CAN Conference*, Amsterdam, Netherlands.

[18] Bosch, 1991, *CAN Specification, Version 2*. Robert Bosch GmbH, September.

[19] F. Hartwich, B. Müller, T. Führer, and R. Hugel, 2000, CAN Network with Time Triggered Communication. In *Proceedings 7th International CAN Conference*, Amsterdam, Netherlands.

[20] WindRiver web site. `http://www.windriver.com/`, 2004.

[21] H. Kopetz, 1995, TTP/A The fireworks protocol. In *SAE International Congress and Exposition, Detroit, Michigan*, February.

[22] R. Alur, L. Fix, and T.A. Henzinger, 1999, Event-clock automata: a determinizable class of timed automata. *Theor. Comput. Sci.*, 211(1-2):253–273.