A Deterministic Infrastructure for Real-Time Distributed Systems

Claudiu Farcas
Calit2, University of California
San Diego, USA
cfarcas{at}soe.ucsd.edu

Wolfgang Pree
C. Doppler Lab Embedded Software Systems
University of Salzburg, Austria
wolfgang.pree{at}cs.uni-salzburg.at

ABSTRACT

The development of reliable software for real-time systems is currently a challenge. Moreover, changing the underlying platform for simple purposes such as a processor upgrade may severely affect the behavior of the real-time software. Working with distributed systems is even more difficult, and transitioning from one system to another is typically impossible.

We address these problems through a development framework for deterministic and portable real-time software using the Timing Definition Language (TDL). It enables transparent, yet deterministic distribution of real-time components regardless of the target platform and its deployment architecture. In this paper, we introduce the algorithms and internal mechanisms for transparent real-time distribution, and analyze the interactions between the user-functionality, the virtual machine of the language, the communication subsystem, and the underlying platform.

1. INTRODUCTION

The advances in computational hardware and the corresponding promises for real-time process control make "embedding" computers into many systems a common practice. Complex applications typically require distributed systems for reasons of dependability (fault-tolerance), scalability, localization. A distributed system may be more reliable than a single node system as through replication faults on a node may be corrected by the other nodes; thus, maintaining a high degree of dependability of the overall system. It can also be extended by adding more processing nodes to solve a computational-intensive job, in comparison with the case of a single node system where a more powerful processor may be too costly, require too much power, or simply be unavailable. On the other hand, the complexity of the distributed systems is several orders of magnitude more significant and harder to deal with than a single-node system. Migrating from a single-node solution to a distributed system is hardly possible with the traditional development methodologies for real-time systems. Even simple changes in the topology of a distributed system or addition of new nodes become a challenge in most applications.

To address these problems our approach for real-time distribution relies on a high-level component-oriented language that makes the timing an explicit part of the real-time software design and decouples the timing from the implementation of the computational tasks of an application. The Timing Definition Language (TDL) [17, 5] is a high-level description language for specifying the explicit timing requirements of a time-triggered [11] application, which may be constructed out of several independently developed components. The actual functionality can be implemented in any imperative language available for the target platform, e.g. C, C++, Java, and later linked with the compiled TDL source. TDL relies on the *Logical Execution Time* (LET) abstraction introduced in the Giotto language [7], but goes beyond with a component model, improved syntax and semantics, and full support for distribution.

LET means that the observable temporal behavior of a computational task is independent from its physical execution. The LET of a TDL computation is always equal with its invocation period and we only assume that its physical execution is fast enough to fit somewhere within the logical start and end points. Thus, it is always defined which value is in use at which time instant and there are no race conditions or priority inversions involved. LET introduces a *unit-delay* behavior [7], which may appear as a disadvantage. However, it provides determinism, composition [9], and platform abstraction, which are more relevant for safety-critical systems.

This paper focuses on the algorithms and mechanisms for transparent hard real-time distribution and the interplay of the components of the run-time system. We present the distribution from the logical point of view of the developer and then from the underlying run-time system of TDL. We introduce an algorithm to bridge the gap between the task and network communication scheduling, and detail the interactions between the user-functionality, the virtual machine of the language, the communication subsystem, and the underlying platform. We introduce an abstraction layer for distribution and present the algorithms for data encapsulation and state synchronization across the network.

We begin with an overview of the TDL component model and its capabilities for structuring complex real-time applications. Section 3 describes the notion of transparent distribution available with TDL and briefly presents the development tool-chain. In Section 4, we analyze the internals of the TDL run-time system, and introduce the algorithms that govern the interactions and synchronization of its constituents, namely the TDL scheduler, the virtual-machine (E-Machine) for logical timing, and the communication layer TDL-Comm. An evaluation in Section 5 illustrates our approach through a running example. We complete the article with a section of related work and our conclusions.

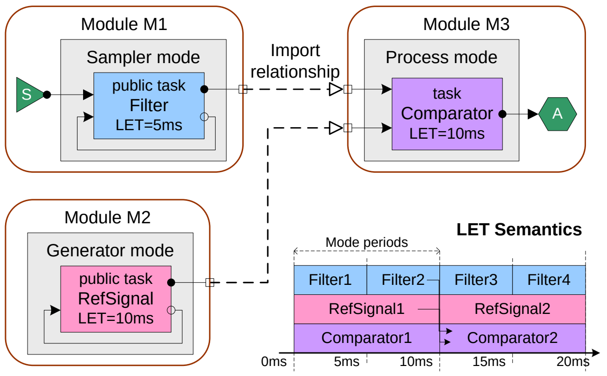


Figure 1: TDL Modules and LET semantics

2. TDL COMPONENT MODEL

The TDL component model relies on the concept of a *module*, which may encapsulate an entire application or parts of a complex application.The TDL modules may work independently or may collaborate to implement a complex behavior. The component model allows for decomposing existing complex applications into smaller, more manageable parts, each with specific timing and functionality, and provides the means for deterministic component interaction. Developers can reuse existing modules to extend the functionality of an application or create new applications.

The TDL module acts as the unit of composition and distribution, and may *import* one or more other modules as depicted in Figure 1. Computation results or environment state can be exported from a module to any other module that imports it. Typically, a developer may import the values of some task output ports by declaring the corresponding task as *public*. This feature introduces data dependencies between modules as one module provides data services to other client modules. It is important to note that LET is always preserved, i.e., adding a new module to an application will never affect the observable temporal behavior of other modules. The TDL compiler performs schedulability analysis using the worst-case execution time for tasks on the target platform and issues an error if LET cannot be maintained (e.g., wcet is too large).

Modes.. The *mode* of a module is a set of periodically executed activities such as task invocations, actuator updates, and mode-changes. The period of an activity within a mode is equal with the mode period divided by the invocation frequency of that activity within the mode. A module has an unique *start* mode.

The mode-change protocol of TDL is different from Giotto, requiring new schedulability analysis [4]. A TDL module changes at run-time its mode independently of other modules. Within a module, TDL enforces *harmonic* mode switches – the mode switch must not break the LET of any task invocation within the current mode of the module. This restriction enables deterministic mode switches in distributed applications. Furthermore, mode switches in a module may break the LET of tasks from other modules, which are not affected by the mode switch. TDL mode changes are regarded as instantaneous.

Tasks are computational units in TDL. A task has a set of input, state, and output ports, along with an external implementation referred through symbolic linking. A *task invocation* within a mode represents the execution of a task instance within the period of that mode. TDL regards the tasks as *scheduled* elements with logical execution time [9]. From the logical point of view, a task reads its inputs at the *release* time, then it runs continuously until its *termination* time, when its computation results are available to the environment and other tasks; whereas, from the platform point of view, the task starts at some point in time after it was released, may be preempted by some other tasks or the RTOS, and completes before the end of its LET. The underlying assumption, which we have to verify [4], is that the run-time system and the scheduling mechanisms used for the physical execution allow each task to complete before its deadline.

Sensors and actuators exchange information between the environment and the tasks of a module. TDL assumes that the external functional implementation of the sensor getters and actuator setters executes in *logical zero time*, i.e., orders of magnitude faster than the smallest task computation. Practical implementations may simply read or write to dedicated memory locations or I/O ports.

Guards are lightweight Boolean functions operating on sensor or task output values. Depending on their result at run time, they condition the execution of corresponding tasks, actuator updates, or mode switches.

Ports interface TDL entities within a module and between modules. There can be input, output, or state ports, each with a distinct type (int, byte, float, etc.). Only tasks can have state ports. To implement the LET mechanism, the TDL tasks have two copies of the output ports: internal and visible. The internal output ports are updated directly by the task functionality code, whereas the visible ports are updated through drivers by the TDL runtime environment at the end of the LET of the task.

The *drivers* as introduced by Giotto are no longer syntactically explicit in TDL. Nevertheless, a TDL driver still performs the port-value copying operations under the LET semantics. The improved syntax of TDL allows the TDL compiler to automatically perform the type checking between ports and then generate the drivers which transport the port values between the interconnected TDL entities.

3. TRANSPARENT DISTRIBUTION

We use the term *transparent distribution* [5] in the context of hard real-time application with respect to two aspects. Firstly, at runtime a TDL application behaves exactly the same, no matter if all modules (i.e., components) are executed on a single node or if they are distributed across multiple nodes. The logical timing is always preserved, only the physical timing, which is not observable from the outside, may be changed. Secondly, for the developer of a TDL module, it does not matter where the module itself and any imported modules are executed. The TDL tool-chain and runtime system frees the developer from the burden of explicitly specifying the communication requirements of modules. It should be noted that in both aspects transparency applies not only to the functional but also to the temporal behavior of an application.

The advantage of transparent distribution for a developer is that the TDL modules can be specified without having the execution on a potentially distributed platform in mind. The only place where distribution is visible is for the system integrator, who must specify the module-to-node assignment.

The development process for TDL relies on the tool-chain from Figure 2. It consists of the following functional components: a TDL compiler, a visual editor fully integrated with the Matlab/ Simulink environment, and a corresponding run-time environment. The TDL compiler has a plug-in architecture, which allows its extension with other tools such as automatic glue-code and bus schedule generators for a target platform. Worst-case execution analysis [20] can be plugged into the visual editor to enable schedulability analysis within the TDL compiler. In this paper, we focus on the TDL runtime environment and briefly mention the relevant aspects of the other tools.

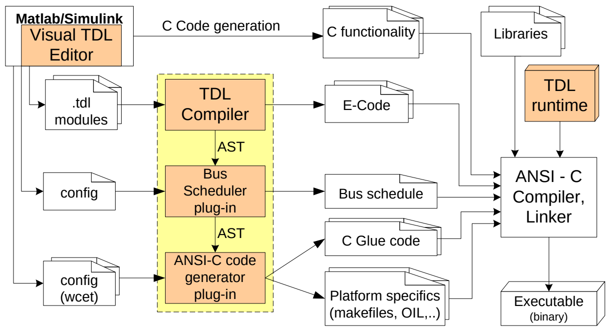


Figure 2: TDL tool-chain for distribution

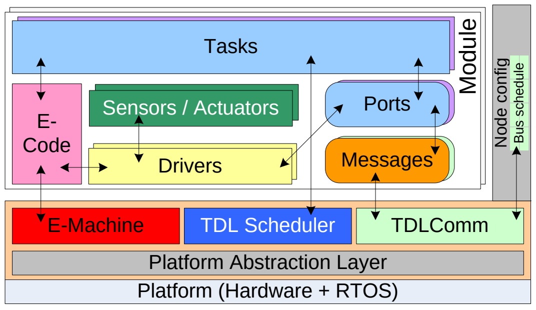


Figure 3: Run-Time Environment

For distributed systems, the Bus-Schedule Generator tool [6] is a compiler plugin that automatically compiles offline the communication schedule. Its configuration file contains the list of computing nodes, the assignment of TDL modules to nodes, and the properties of the communication channels (e.g., bus rate, minimum and maximum packet sizes). The tool analyzes the import relationships in the TDL modules to identify their remote dependencies and the set of messages required for exchanging the information between producer tasks and consumer entities such as tasks, actuators, or guards. It then tries to generate a TDMA communication schedule that satisfies the requirements of the TDL modules and their mode changes. The schedule specifies which node sends information at which time; the structure of the information depends on the current modes at run-time [6].

As a first step towards fault-tolerance TDL supports module replication. The replicas are identified from the module-to-node assignment in the configuration file of the tool. We send the messages produced in all service-provider modules and we process them in all their stubs through majority voting. By scheduling the replicated messages as any other message, the tool also achieves temporal isolation between replicas that improves the recovery chances from transient failures.

4. RUN-TIME MECHANISMS

We introduce in Figure 3 the TDL runtime environment consisting of three logical components deployed on each node: virtual machine, scheduler, and communication layer. The virtual machine supervises the logical behavior of the application and its interaction with the environment. The TDL Scheduler performs the mapping of platform time to logical time, the invocation of the virtual machine and the communication layer, and the preemption and dispatching operations of the user tasks. The communication layer handles the distribution aspects.

4.1 E-Machine

For portability reasons, TDL reuses the approach of a virtual machine, the E-Machine introduced in Giotto [8], to handle the logical aspects of its runtime environment. In addition to Giotto, the TDL E-Machine handles parallel and distributed modules. It executes a small set of instructions (TDL E-Code [17]) related only with the logical aspects of a module: when and which tasks to release, and which drivers to execute for the correct information flow between ports. The functionality of the module runs in the native code of the platform for maximum performance.

In distributed systems, a *service-provider* module and its *client* modules (there can be more than one module importing a service provider module) may be placed on different nodes. The TDL compiler generates a *stub* of the service-provider module on each node that does not contain it but contains one of its clients. A stub module is a logical image of a service-provider module. It does not contain any functionality for tasks, but only their output ports. This concept enables the seamless distribution of modules in the system and improves the performance on the nodes containing the client modules as they do not have to locally execute the service module. Nevertheless, the communication layer must synchronize the state (mode, port values, timing) of a service provider module with all its stubs (see Section 4.3).

From a logical point of view, the E-Machine executes the E-Code in logical zero time, for each module individually, regardless of its type, i.e. ”normal” module or stub module. From the platform point of view, the TDL Scheduler that invokes the E-Machine accounts the time spent interpreting E-Code and adjusts its decisions accordingly.

The TDL compiler generates an E-Code file per module as a compact representation of the activities defined in that module and their timing information. It also includes the dependency information related with the import relationships between modules. The Code-Generator Plugin converts this information into the corresponding drivers and associated glue-code. Thus, the E-Machine can simply execute the E-Code instructions and call the appropriate drivers to ensure the correct intra- and inter-modules data flow.

4.2 TDL Scheduler

The TDL Scheduler is the actual bridge between the TDL semantics and the underlying platform (real-time operating system, computing hardware, and distributed system architecture). Its purpose is to run the E-Machine for each module at the right time as defined by the TDL semantics and then to execute the released tasks according to a specific scheduling policy. In a distributed setup, it furthermore has to coordinate the exchange of data between the nodes via the TDLComm layer.

To support the parallel composition of modules and to execute them on the same node, we have to allocate a fraction of the CPU time to each module of the node. Traditionally, we would solve this problem with clock-driven scheduling [13] via a static time-sharing mechanism, or CPU partitioning. Within a scheduling cycle of the GCD of all activity periods from all modules, we would allocate for each module a time quantum proportional with the maximum load the module generates on the CPU. However, this approach introduces a high context-switch overhead on the running system, because of the infinitesimal time quantum required to implement this mechanism.

As alternative, we could use a on-line scheduling algorithm, such as Rate Monotonic (RM) or Earliest Deadline First (EDF) [12]. In the following, we present a scheduling approach for TDL using EDF on the global set of tasks from all modules. Conceptually, the modules are simply logical constructs; hence, from a scheduling perspective we can treat equally all tasks from all modules. The only property that matters for EDF scheduling is the deadline of the task, regardless of its parent module. In addition, as the application is strictly time-triggered and we know at compile time all task and mode periods, we can benefit from this apriori information to reduce the run-time scheduling overhead by using precompiled tables. The TDL Scheduler avoids unnecessary context switches by running only when it has to invoke the E-Machine/TDLComm or when a task completes. Consequently, it allows a better CPU utilization than the partitioning approach. Note that its sleeping interval is not constant as it changes with every scheduling decision.

Using the EDF algorithm, we build for every module a set of dispatch tables $DT[M]$, one for each of its modes, which captures at compile-time the dispatching order of the tasks within a mode. The dispatch table $DT[M]_m$ of a mode m contains a set of entries, each entry consisting of a task and its relative deadline since the beginning of the mode. On single node systems, these deadlines are simply multiples of the task periods; whereas for distributed systems, the bus schedule reduces the available time for the producer task executions by moving their deadlines sooner, at the start time of the corresponding messages. Note that also other scheduling policies (e.g., power-aware) can be used to build the dispatch tables.

For each module M, we have an associated dispatch table index i_M, which points to a task entry in the dispatch table of the current mode $DT[M]_m$ that has a deadline closest to the current logical time. Each time the E-Machine releases a task, it adds the task to the set of active tasks $\mathsf{Tasks}_a[M]$ of that module. It also resets the index i_M when it performs a mode switch in a module or starts a new cycle of a mode. The set of active tasks in a module is always correlated with the dispatch table of the current mode of that module.

We introduce Algorithm 1 for parallel execution of multiple modules using a lightweight EDF scheduling with precompiled dispatch tables. The complexity of the algorithm is $O(\|\mathsf{Modules}[N]\|)$, where $\|\mathsf{Modules}[N]\|$ represents the number of modules on a node N, because we have to perform EDF scheduling only among a single task per module. Note that it is not possible to create a dispatch table for all tasks from all modules because they can switch their modes independently. Also, creating dispatch tables for all combinations of modes from all modules is highly unpractical.

We note with t the current absolute logical time and with t_m the absolute logical time when the mode m started its current period. We use t_m to convert from the deadlines relative to the beginning of the mode to the absolute deadlines required by the EDF algorithm.

The algorithm proceeds through five steps: update the state of the system (value and time), perform the network communication, perform the logical actions according to the LET semantics, schedule the active user tasks, dispatch one task, and sleep until the task completes or a precomputed timeout expires.

Algorithm 1: Task/Bus scheduling

```
// Step 1 − UPDATE STATE
$t_{begin} \leftarrow$ Get_Current_Time()
Save_Task_Context_and_Preempt($\tau_{old}$)

// account for elapsed time, where t is the logical time
// and $\delta$ is the waiting interval from the previous invocation
if ($t - t_{begin} < \delta$) // a task completed sooner
  // update $\delta$ to reflect elapsed time
  $\delta \leftarrow t - t_{begin}$
end if
Update_Time($t$, $\delta$)
Update_Time($EMachineWait$, $\delta$)

// Step 2 − COMMUNICATION
$NetWait \leftarrow NextPacket$.time $- t$ mod $NetworkPeriod$
if ($NetWait = 0$)
  Invoke(TDLComm) // data exchange required
end if

// Step 3 − LOGICAL ACTIONS
if ($EMachineWait = 0$)
  // at least one module has to perform logical activities
  Invoke(E−Machine) // returns $Time\_to\_Next\_Activity$
  $EMachineWait \leftarrow Time\_to\_Next\_Activity$
end if

// Step 4 − USER−TASKS SCHEDULING
$\delta \leftarrow \infty$ // retains closest task deadline from all modules
foreach $M \in$ Modules // all modules of this node
  // skip past entries
  increment($i_M$) while($t - t_m >= DT[M]_m[i_M]$.dln)
  $i \leftarrow i_M$ // seek the first active task
  while($i < \|DT[M]_m\|$)
    if ($\delta > DT[M]_m[i]$.dln $- t_m$ and $DT_m[i].\tau \in \mathsf{Tasks}_a[M]$)
      $\delta \leftarrow DT[M]_m[i]$.dln $- t_m$
      $\tau_{new} \leftarrow DT[M]_m[i].\tau$
      break
    else
      $i \leftarrow i + 1$
    end if
  end while
end foreach // $\tau_{new}$ has the closest deadline from all modules

// Step 5 − DISPATCHING & WAITING
$\delta \leftarrow$ minimum($\delta$, $EMachineWait$, $NetWait$)
$t_{overhead} \leftarrow$ Get_Current_Time() $- t_{begin}$
$\delta \leftarrow \delta - t_{overhead}$ // account for elapsed time
Set_Alarm_for_Sleep($\delta$) // will sleep after dispatching the task
Dispatch_Task($\tau_{new}$) // start/resume the execution of task $\tau_{new}$
```

In the step 1, we first preempt a previously running task and save its current state. The time interval δ represents the sleeping interval of the TDL Scheduler from its previous invocation. We first verify that the Scheduler slept for the required interval or that a task completed sooner and thus the Scheduler was invoked to dispatch another task. We then update the current logical time t and the time interval until the next logical activity from a module, i.e. the moment when the E-Machine has to execute the E-Code of a module.

The bus scheduler tool provides the communication schedule for each node in the form of a table, which lists the packets and their timing. Thus, at step 2, we lookup in this table the time of the following packet and compare it with the current time correlated with the network cycle time. If they match, we invoke the TDLComm layer to send or receive that packet.

Afterward, at step 3, we evaluate the existence of an immediate logical activity to perform. In the case, when any module has such upcoming logical activity (e.g. beginning or end of a task's LET, actuator updates, mode switches), we invoke the E-Machine to execute the E-Code of the appropriate modules.

Reaching the step 4, we proceed to the actual scheduling of the active tasks from all modules. We process all modules and skip the entries in the dispatch table of their currently executing modes that have the deadlines less than the current logical time relative to the beginning of the corresponding mode. We then select as the next dispatch-able task the first task that is active in the dispatch table, as it would have the closest deadline. We cannot increment the dispatch index at this point as it could be that more tasks have the same deadline but have not been released yet (the alternative of keeping track of both released and running tasks requires twice the amount of memory than the simple set of active tasks $\mathsf{Tasks}_a[M]$).

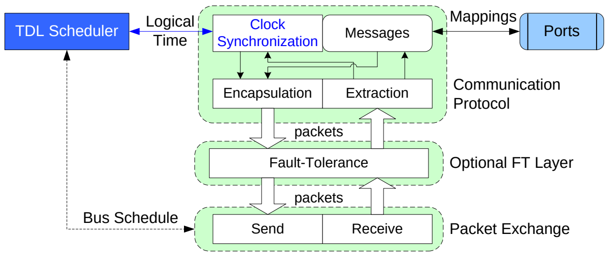


Figure 4: TDLComm services

We update δ with the current closest deadline and the supposed next task to dispatch τ_{new}. After we iterate through all modules, we obtain the smallest sleep/dispatch interval until we have to invoke the scheduler again.

At step 5, we compute the waiting interval as the minimum between the time until the next dispatching, logical, and communication actions. We then subtract from the computed δ the overhead of communication, scheduling, and E-Code execution, to obtain a more accurate estimation of the time allocated to the next dispatching task.

With the information about the next task to dispatch and an estimation of its running time, the Scheduler can prepare its next invocation, dispatch the task and sleep. From the calculation of δ, we can see that the algorithm is invoked repeatedly, but without a fixed period.

We implemented the algorithm on top of existing real-time operating systems such as OSEK [16] or RT-Linux by enforcing the dispatching operations through task/thread priorities. In contrast with classical implementations of EDF on top of fixed-priority operating systems requiring a large number of priority levels [3], our algorithm can work with just three priority levels: high - for the task/thread implementing the TDL runtime (scheduler, E-Machine, TDLComm), medium - for the next/current task to dispatch, and low - all other active tasks. In this way, we still have linear complexity $O(\|\mathsf{Modules}[N]\|)$ because the RTOS scheduler becomes a dispatcher of the first active job (TDL scheduler or dispatched task).

4.3 TDLComm

The TDLComm layer abstracts the physical exchange of information between the nodes of a distributed system. From a logical point of view, using the precompiled scheduling information, the TDLComm layer performs at run-time three steps: the *encapsulation* of port values from service provider modules into packets, the *transmission* of the packets over the communication medium, and the *extraction* of stub-port values from the packets received on the client node (see Figure 4). Each node contains the subset of the global communication schedule relevant for its activities. This subset contains also the failure-management information about the replicated messages how to consolidate them.

For its time-triggered transmission and reception of packets, the TDLComm layer relies on the TDL Scheduler to invoke its functionality at moments of time defined by the communication schedule. On the other hand, it provides the clock-synchronization service on each node of the system, and introduces constraints on the scheduling of the tasks that exchange values over the network. Thus, the close cooperation between the TDLComm layer and the TDL Scheduler is crucial for a successful implementation of the transparent distribution concept of TDL.

The TDLComm layer uses an innovative TDMA protocol [6] that dynamically multiplexes the messages over a static schedule. To handle the independent mode switches of TDL modules, this protocol considers the communication period as the smallest interval where mode switches cannot occur, that is the GCD of the mode-switch periods of all producer modules. The resulting communication period equally divides the period of any mode of a producer module into a fixed number of *phases*. The phases of a mode are mutually exclusive, and any producer module may change its mode only at phase boundaries.

According to this TDMA protocol, any node is allowed to send messages in statically defined slots only. The run-time environment implements a mechanism for global clock synchronization over the network [14]. The data exchange model implemented by the scheduling tool adheres to the Producer-Consumer model. The nodes that generate information (the producers), trigger the sending of information over the network. Contrary to the classical Client-Server model, in the Producer-Consumer model the consumers (the nodes that need the information) do not send any requests to the producers.

Messages, Datagrams, and Packets.

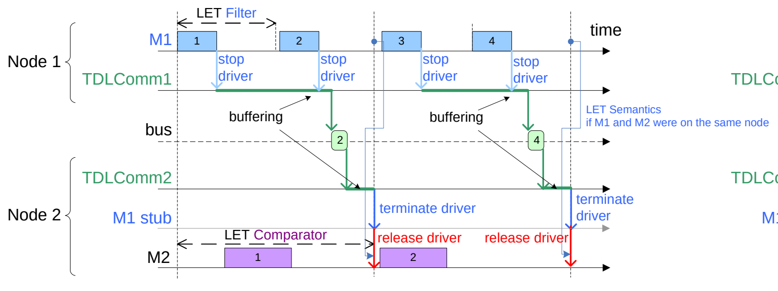
A *message* represents a data exchange between the ports of a pair of TDL entities from two modules located on different nodes in a distributed system. It corresponds to a value exchange operation between sets of ports, discarding the output ports of the producer entity that are not used by a consumer entity. It has a fixed size equal with the sum of the sizes of the producer port types, and two time constraints derived from the availability of the corresponding port values and the latest allowable receive moment (i.e., the end of the LET of the producer task).

The algorithm of the bus scheduler tool identifies the messages from producer tasks per each phase of a mode, and then it associates a message with the phase at the end of the producer's task LET. As the phase and the mode in which a message is produced change at run-time, we also associate to each message a *tag* that encapsulates this information.

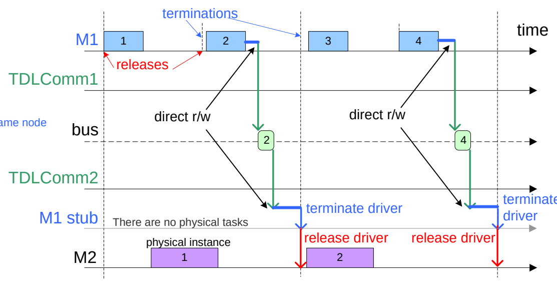
A message belongs to a particular task instance; thus, it is not periodic. The number of messages depends on the periods of the producer tasks, the number and period of mode switches, and possibly on the number and periods of the consumer entities [4].

A *datagram* represents a collection of messages exchanged at the same time instant. It contains one or more messages; thus, it refers indirectly to one or more task entities that provide output values from the same or different modules of a node. A datagram has a fixed size equal with the sum of the sizes of each of its message constituents. During the generation of the communication schedule, the scheduling algorithm may grow or shrink the size of a datagram by adding or removing messages; however, once a feasible communication schedule is found and generated, the allocation of messages to datagrams remains fixed.

We refer to a *packet* as the unit of information to send on the communication channel. A packet has a minimum and maximum size derived from the physical properties of the communication channel and the low-level data-exchange protocol. Any packet may contain one datagram only, but more packets may refer to the same datagram. Thus, we can consider a packet as a physical instance of a datagram on the communication channel. Note that in contrast with a datagram, a packet contains actual port-value information, whereas a datagram acts just as a logical container. The order and timing of the packets is fixed within a communication round and expressed statically in the communication schedule. In addition to the actual message data, a packet may contain control information such as tags, timestamps, or other communication-protocol related information.



a) Stop drivers initiated encapsulation



b) TDLComm initiated encapsulation

Figure 5: Information exchange between nodes

A packets has two important attributes: direction and time. The direction specifies the type of operation the TDLComm layer has to perform with the packet. The time attribute for a sending operation reflects the logical time when the TDLComm layer of a node containing a producer module has to send the packet; whereas, for the receiving operation it reflects the logical time when the packet was already received by the network processor of the node containing the client module, and stored in the processor's local buffers or the main memory. In addition, a packet has an index and a datagram reference.

Sending information

We identified two means for capturing the information from a producer module intended for its stubs. The original approach of TDL relies on the stop drivers of the producer tasks. In this case, after the completion of the tasks, their stop-drivers copy the relevant internal port values to the TDLComm layer as presented in Figure 5 (a). On the client-node side, the terminate driver of the stub of the service-provider module performs the port-value copy operation from the TDLComm layer into its visible ports at the end of the LET of the producer task. The release driver of the client task reads the value from the visible output ports of the stub as if the producer task was running on the same node. With this approach, there are multiple drawbacks from the additional meta-information required at the producer module within the task stop-drivers and at the stub module within the task-termination drivers about the TDLComm data-structures, operational mode of the module, and logical time. This requires different glue-code (containing the drivers) for the provider module in the case of single-node versus distributed systems.

We take a better approach that gives the TDLComm layer direct access to the relevant internal ports at the moments defined by the TDL Scheduler and the communication schedule as depicted in Figure 5 (b). Thus, TDLComm reads directly the internal output ports of the producer task, encapsulates them into a packet and sends the packet to the other nodes. On the client-module side, the corresponding TDLComm layer invoked by the TDL Scheduler at the receiving time extracts the port values from the received packet and stores them directly into the internal output ports of the stub module. Hence, the stop-drivers are no longer needed, the stub task-termination drivers are simply identical with its service-provider module, and the glue-code of the service provider module remains the same regardless of the system architecture. Note that under any circumstances the internal output port values are available only to the TDLComm layer before the end of the LET of the corresponding task. All the user tasks in the system can access only the visible output port values, which retain their previous values until the termination event of the task (when the termination driver updates them from the values of internal output ports).

The overhead of copying the data from the main memory to the network processor memory may be negligible, but on slow systems it has to be evaluated and considered as networking overhead when performing the time-safety checking of the distributed system and at runtime within the TDL Scheduler.

The transmission of the computation results of a producer module to its stubs requires a data encapsulation phase. For this purpose, we introduce the Algorithm 2 that computes first the phase of the current mode of a module and creates a corresponding tag from the module, mode, and phase. It then starts building a new packet by matching the set of possible messages with the previously identified tag. As the tag captures the dynamic state of the module and there may be more than one message with the same tag, it packs the tag and the content of the corresponding ports into the packet. For the case where multiple messages from different modules and phases are merged into a larger frame, we have to repeat the algorithm for each module.

Algorithm 2: Dynamic packet encapsulation

```
// t is the current time
// $m \in$ Modes$[M]$ is the current mode of $M$
// $t_m$ is the time when the current mode period started
// d is the datagram corresponding to the $NextPacket$ to send

$NextPacket$.data $\leftarrow \emptyset$ // first construct a new packet
foreach $M \in$ Modules$[N]$ // all modules on the node $N$
  $\phi_m = (t - t_m)$ mod $NetworkPeriod$ // compute phase
  $tag \leftarrow (M, m, \phi_m)$ // tuple expressing module dynamics
  Store_Tag($NextPacket$.tags, $tag$) // add tag
  foreach $msg \in d$ // process all messages
    if ($msg$.tag = $tag$)
      $v \leftarrow$ InternalPortValue($msg$.PortReference)
      $NextPacket$.data $\leftarrow$ $NextPacket$.data $\cup \{v\}$
    end if
  end foreach
end foreach
// Send $NextPacket$ and advance its index
```

Receiving information

The extraction of port values from the packets relies on the tag information from a received packet. Hence, we introduce the Algorithm 3 that first identifies the tag of the received packet and then decodes the state of the producer module, its mode, and current phase.

When the node containing a client module and a stub of the service provider module receives the packet, the producer module may have changed into a new mode at the beginning of the communication cycle. In this case, we change the mode of the stub module and update its mode start time t'_m and phase ϕ'_m. We proceed to the extraction of the packet's content and store the received values

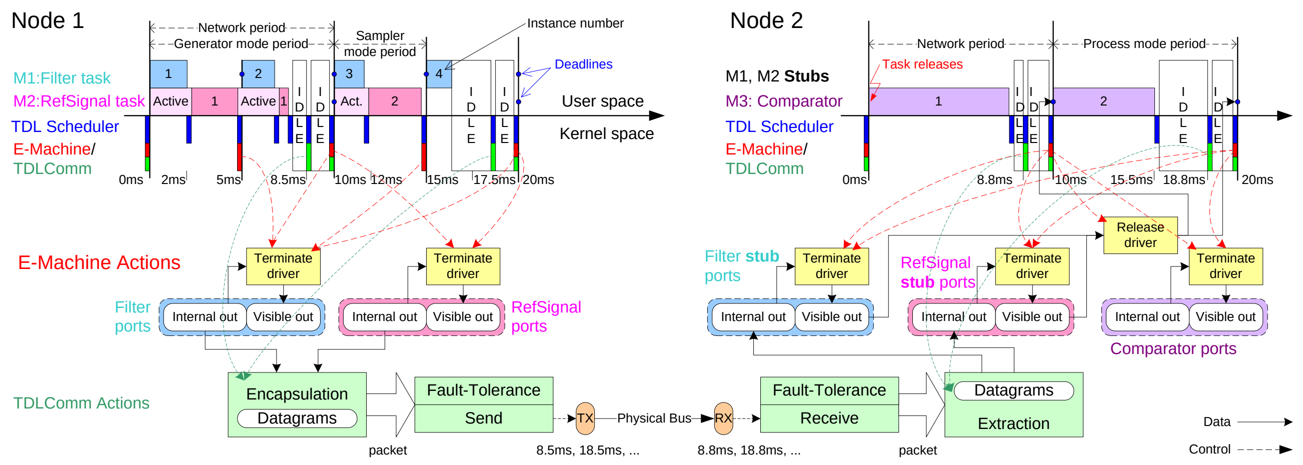


Figure 6: Data flow between modules through TDLComm

into the corresponding internal ports of the stub module.

The protocol implemented by TDLComm brings flexibility in static communication scheduling, by allowing dynamic mapping of messages over the same frames. Nevertheless, it still provides deterministic communication patterns and a solid foundation for transparent distribution.

Algorithm 3: Stub synchronization

```
// t is the current time
// $M_{stub}$ is a stub module for $M$
// $m \in$ Modes$[M]$ is the current mode of $M$
// $m' \in$ Modes$[M_{stub}]$ is the current mode of $M_{stub}$
// $t_m$ is the time when the current mode period started
// $t'_m$ is the time when the current stub mode period started
// $d$ is the datagram corresponding to the $NextPacket$ to receive

foreach $tag \in NextPacket$.tags
  $(M, m, \phi_m) \leftarrow tag$ // decode tag
  $\phi'_m = (t - t'_m)$ mod $NetworkPeriod$ // compute stub phase
  // synchronize timing of producer and stub
  if $(m' \neq m)$ // producer has changed mode during this phase
    $t'_m \leftarrow \lfloor t/NetworkPeriod \rfloor \cdot NetworkPeriod$
    $m' \leftarrow m$ // update mode of stub and its start time
    $\phi'_m \leftarrow \phi_m$ // update mode phase
  end if
  foreach $msg \in d$ // process all messages
    if $(msg$.tag = $tag)$
      $v \leftarrow$ FetchPortValue($NextPacket$.data)
      SetInternalValue($msg$.PortReference, $v$)
    end if
  end foreach
end foreach // internal ports & mode of stub are synchronized
```

5. EVALUATION

We take as example the application containing the three modules from Figure 1. In this simple application, each module has only one operating mode containing one task. The first module has a sensor connected to the input of the Filter task, which is invoked with a period of 5ms. The RefSignal task of the second module has a period of 10ms and only state ports to retain its state between consecutive invocations. The third module has no sensor inputs but imports the other two modules to access the outputs of their tasks. The Comparator task is invoked every 10ms to perform some computations and provide their result to the environment via an actuator.

From a logical point of view, the three module run in parallel regardless of the underlying platform and its capabilities. The LET semantics dictate that the first output of the RefSignal goes to the second invocation of the Comparator (in the second cycle of the Process mode). Similarly, the output of the second instance of the Filter reaches the second instance of the Comparator, then the fourth output of Filter reaches the third output of the Comparator and so on. This is the so-called unit-delay behavior (see Figure 1).

We consider now the case of a distributed system, where the first two modules are located on the node 1 and the third module on the node 2. Figure 6 presents on the upper part the CPU scheduling, and in the lower part the mechanism for transparent distribution (without the clock synchronization service).

On the first node, the TDL Scheduler invokes the TDLComm layer to perform the clock synchronization of the two nodes, and then it invokes the E-Machine to initialize the ports of the tasks and release the tasks of both modules. The deadline of the Filter task is sooner than the deadline of the RefSignal task; therefore, the Scheduler dispatches the Filter task and keeps the RefSignal task into the active state. The run-time environment of node 2 performs the same steps and dispatches the Comparator task.

The two nodes run in parallel, each executing one task until the moment of 2ms, when the Filter task completes its execution. The scheduler of the first node is invoked and dispatches the RefSignal task, whereas the second node continues its computations undisturbed. At 5ms, the scheduler of the first node preempts the RefSignal task to invoke the E-Machine that first executes the termination drivers of the Filter task and then releases a new instance of this task. At this point both tasks on the first node have the same deadline of 10ms. We assume that the scheduler dispatches the Filter task that completes its computations sooner than its first instance (around 6.5ms). The scheduler dispatches the remaining RefSignal task that shortly completes its computations.

Meanwhile, the second node completed the execution of the Comparator task around 7ms and idles. We assume that the communication schedule specifies the sending of a packet from the first node to the second at 8.5ms. As there is no other active task, the node 1 remains idle until this moment. The scheduler on node 1 invokes its TDLComm layer to capture the internal output port values of the two tasks. The results are encapsulated and transmitted over the network in 300us, reaching the second node at the time of 8.8ms. The Scheduler of the first node returns to the idle state, whereas the Scheduler of the second node wakes up and executes its TDLComm layer to extract the data from the received packet and update the internal output ports of the two stub modules. Afterward, it reenters the idle state until the beginning of a new mode cycle.

When the logical time reaches 10ms, on both nodes, the TDL Scheduler invokes the TDLComm layer for clock synchronization and then invokes the E-Machine. The virtual machine executes on the first node the termination drivers of both tasks, thus making their outputs available to the environment. Afterward, it begins a new mode cycle in each module and releases new instances of the Filter and RefSignal tasks. On the second node, the E-Machine executes the stub termination drivers and makes the computation results of the two modules from the first node available to the local environment. Thus, the release drivers of the Comparator task read the correct values as if all modules were executed on the second node. The new cycle of the third module begins with another release and dispatch of the Comparator task.

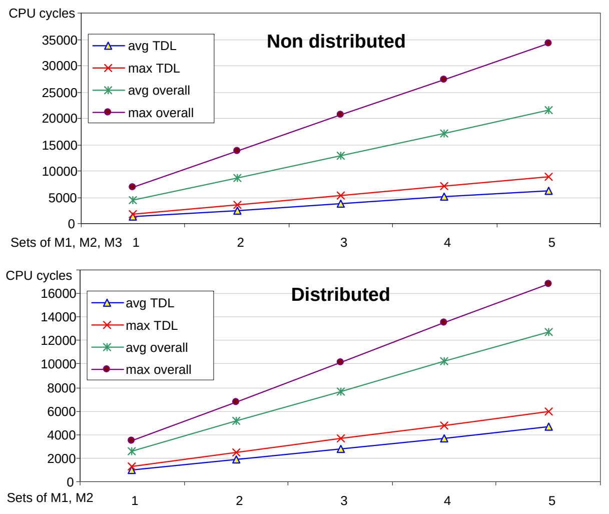


Figure 7: CPU utilization

In the second network cycle, assuming that the sensor values require less filtering, and the comparison takes less time, all tasks from all modules complete their execution sooner than in the first network cycle. Nevertheless, the behavior is still the same and the communication pattern remains unchanged. It also matches the behavior of the modules on a faster single-node system, thus, illustrating the benefits of our approach for distribution.

In Figure 7, we measured using Avrora [18] the number of CPU cycles required for the execution of the TDL runtime environment and the overall utilization. The system under evaluation is an Atmel AVR microcontroller with TDL running on the bare hardware (microkernel design). We start with one set of the three modules on a single node. We continue by adding sets of (M1, M2, M3) up to 15 modules per node and observe that the CPU utilization grows linearly with the number of modules to execute. In correlation with previous research for Giotto [10], our measurements indicate aprox. 1.5% CPU utilization on a similar platform, proving that executing parallel modules increases the development flexibility at no performance penalty. In the second graph, we distribute the module M3 on a second node connected via CAN (with our TDMA protocol on top), and use sets of (M1, M2) for testing. The CPU utilization on the first node drops significantly and the system maintains its observable behavior unchanged. Thus, we can make better use of the available resources of the target platform in a distributed system.

6. RELATED WORK

Giotto [7], the precursor of TDL, is primarily an abstract mathematical concept and there exist only simple prototype implementations, which show some of the potential of LET: static time-safety checks and platform-independent embedded code (E-Code) executed by virtual machines. Its main focus are the single-node systems with limited support for task-level distribution. The developer has to annotate the Giotto source code with network specific parameters such as the hostname and ports; thus, mixing the platform-independent and platform-dependent code. The actual implementation of the communication has to be coded manually with so-called scheduling-code instructions. In other words, no tools automatically generate the message schedules for the bus communication.

Our approach is more realistic and takes into account the complexity of the real-world applications, where entire parts of an application are distributed - modules. The key elements for the transparent distribution in this case are the stub module concept and the TDLComm layer that decouples the distribution aspects from the rest of the run-time environment and frees the developer from the burden of developing individual modules towards a distributed solution. The integrated communication and task scheduling make our approach a feasible solution for distribution, with support for parallel module execution with data dependencies between modules regardless of their placement on the network.

TTP [19] protocol and its related tools provided by the TTTech company approach the problem of building real-time distributed system with proprietary, expensive, high-confidence hardware that features membership services, time-triggered transmission of messages and distributed clock synchronization. In addition, in its current state the developer has to consider in advance the platform topology, the number and type of messages exchanged between the nodes. The schedule itself is then generated with the TTPplan tool, but it cannot support multiple application modes. The possibility of independent mode switches on each node, by arbitrary modules is simply out of the modeling possibility of the current tools.

Our approach abstracts from the communication infrastructure through transparent distribution, which shields the developer from ever defining each message individually or even designing for a particular distributed platform. Modules can be developed independently and later integrated without affecting their timing behavior.

FlexRay [1, 2] is an emerging high-speed fault-tolerant protocol for control applications. A group of companies including BMW, Volkswagen, DaimlerChrysler, Bosch, Philips, and Freescale is actively developing it, with its specifications in the final phase. Apart from time-triggered operations, its focus is flexibility. Thus, it can operate in both active star and passive bus topologies, and can accommodate both static and dynamic parts in its communication rounds. Node may be connected through one or two communication channels (for redundancy), and may transmit the same data on both channels, or different data on both channels, in a given current time slot. However, given its age it is mostly implemented using proprietary ASIC chips and its adoption in the automotive industry has still to gain momentum. The development tools for it have also limited capabilities, comparable with the ones for the TTP protocol. Thus, the potential of the protocol remains still hindered by inappropriate software.

The presented approach is currently under evaluation on this platform and preliminary results show that it is possible to gain the benefits of using this protocol and accompanying hardware in addition to the transparent distribution of the TDL components. A description of the model-driven development process for a FlexRay platform using TDL is available in [15].

7. CONCLUSIONS

The presented development methodology goes significantly beyond Giotto and relives the developer from the burden of explicitly designing the application for a distributed system and allows the integration of an application out of individually developed modules. It also reduces the dependence on proprietary networking technologies for real-time systems, while still benefiting from their capabilities when such networking options are available. Future research, implementation, and testing efforts are required to show the scalability of transparent distribution in complex scenarios. Remaining challenges are better heuristics for generating communication and task execution schedules to account for power/bandwidth usage or other dynamic environments, and strategies for avoiding the re-generation of schedules when components are added or modified.

8. REFERENCES

[1] J. Berwanger, C. Ebner, and et al. FlexRay - The Communication System for Advanced Automotive Control Systems. In *SAE World Congress*, Detroit, MI, Apr. 2001. Society of Automotive Engineers Press. 2001-01-0676.

[2] F. Bogenberger, B. Müller, and T. Führer. Protocol overview. *FlexRay International Workshop: The Communication System for Advanced Automotive Control Applications*, Apr. 2002.

[3] G. C. Buttazzo. Rate monotonic vs. EDF: Judgement day. *Embedded Systems*, pages 67–83, Sept. 2003.

[4] E. Farcas. *Scheduling Multi-Mode Real-Time Distributed Components*. PhD thesis, Department of Computer Science, Univ. of Salzburg, Austria, 2006.

[5] E. Farcas, C. Farcas, W. Pree, and J. Templ. Transparent distribution of real-time components based on logical execution time. In *Proc. of LCTES*. ACM Press, 2005.

[6] E. Farcas, W. Pree, and J. Templ. Bus Scheduling for TDL Components. *LNCS - Dagstuhl Conference on Architecting Systems with Trustworthy Components*, May 2006.

[7] T. Henzinger, B. Horowitz, and C. Kirsch. Giotto: A time-triggered language for embedded programming. In *Proc. of EMSOFT*, LNCS 2211, pages 166–184. Springer, 2001.

[8] T. Henzinger and C. Kirsch. The Embedded Machine: Predictable, portable real-time code. In *Proc. of the PLDI*, pages 315–326. ACM Press, 2002.

[9] C. Kirsch. Principles of real-time programming. In *Proc. International Workshop on Embedded Software (EMSOFT)*, LNCS 2491, pages 61–75. Springer, 2002.

[10] C. Kirsch, M. Sanvido, and T. A. Henzinger. A programmable microkernel for real-time systems. *Proc. of VEE*, 2005.

[11] H. Kopetz. The Time-Triggered Model of Computation. In *Proc. of the IEEE Real-Time Systems Symposium*, 1998.

[12] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in hard real-time environments. *Journal of the ACM*, 20(1):46–61, Jan. 1973.

[13] J. W. S. Liu. *Real-time Systems*. Prentice Hall, 2000.

[14] G. Menkhaus, M. Holzmann, and S. Fischmeister. Time-triggered Communication for Distributed Control Applications in a Timed Computation Model. In *23rd Intl. Digital Avionics Systems Conference*. IEEE Press, 2004.

[15] A. Naderlinger, J. Pletzer, W. Pree, and J. Templ. Model-Driven Development of FlexRay-Based Systems with the Timing Definition Language. Mineapolis, May 2007.

[16] OSEK Group. *OSEK/VDX Operating System v2.2.3, and OSEKtime - Time-Triggered OS v1.0*, 2005.

[17] J. Templ. TDL Specification and Report. Technical report, University of Salzburg, Austria, http://www.softwareresearch.net/site/publications/C059.pdf, 2004.

[18] B. L. Titzer, D. K. Lee, and J. Palsberg. *Avrora: scalable sensor network simulation with precise timing*. IEEE Press, Los Angeles, CA, 2005.

[19] C. G. TTTech. *TTP/C Protocol Specification, Version 1.0*. Wien, Austria, June 2002.

[20] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Muller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. *The Worst-Case Execution Time Problem – Overview of Methods and Survey of Tools*. Tech. Report. Mälardalen University, Sweden, Mar. 2007.