# INTEGRATION OF TWO COMPLEMENTARY TIME-TRIGGERED TECHNOLOGIES: TMO AND TTP

R. Obermaisser[1], E. Henrich[2], K.H. Kim[2], H. Kopetz[1], M.H. Kim[3]
[1]*Vienna University of Technology, Vienna, Austria*
[2]*University of California, Irvine, CA, 92697 U.S.A*
[3]*KonKuk University, Seoul, Korea*

Abstract: The TMO model for real-time distributed object-computing supports the specification of temporal constraints with respect to a global time base and provides execution engines for ensuring that these constraints are met at runtime. This paper describes a solution for supporting TMO applications on top of the Time-Triggered Architecture, a system architecture that meets the dependability requirements of safety-critical applications. Thereby, the TMO model advances to the domain of safety-related and safety-critical systems. Due to an ongoing trend in transportation systems (e.g., automotive industry) of increasing functionality and complexity, this domain can benefit from the intuitive and programmer-friendly TMO scheme. We realize TMO on top of the TTA by establishing a Kernel Abstraction Layer (KAL) for the TTP/RTAI-Linux platform. This KAL maps the platform services used by the TMO support middleware to the service of the TTP protocol and the operations of RTAI-Linux. KAL also layers event-triggered communication for remote method invocations and multicast channels on top of the purely time-triggered TTP protocol.

Key words: real-time systems, distributed object-computing, time-triggered control

## 1. INTRODUCTION

Distributed object computing is a widely accepted model for conquering the complexity of large distributed systems. Each object represents a nearly-independent subsystem (Simon, 1996, chap. 8) that is designed without being influenced by the internal details of other objects.

The Time-triggered Message-triggered Object (TMO) model (Kim et al., 1994) has extended distributed object computing to the area of distributed real-time computing in an intuitively appealing form. TMO combines the complexity management benefits of the object-oriented paradigm with the ability to explicitly specify temporal constraints in terms of global time in natural forms. In addition, TMO execution engines based on several major platforms (e.g., Windows NT (Kim and Ishida, 1999), Windows CE (Gimenez and Kim, 2001), and Linux (Kim et al., 2002)) have been developed. An execution engine consists of a commercial operating system (OS) kernel, middleware devised to support TMOs, and libraries of object classes for a particular programming language which wraps the services of the middleware and collectively serve as an approximation of an ideal TMO programming language. An execution engine honors the temporal constraints specifications by managing execution resources judiciously.

With the increasing functionality of electronics in safety-related and safety-critical real-time applications, the need for programming models that facilitate the effective management of growing complexity in this domain has become acute. Currently, in embedded system design such as the design activities in the automotive domain, assembly and C language are still prevalent (VDC, 2003). The distributed object computing paradigm can be exploited to reduce design

costs, time-to-market, and the number of software-related recalls. A prerequisite for applying this technology to safety-related and safety-critical applications is the ability to certify the resulting systems. In this regard, the underlying distributed computing platform must be based on an architecture that supports safety-critical fault-tolerant computing.

This paper describes an approach for distributed real-time object computing on a platform architecture that is appropriate for safety-critical applications. We realize TMO on top of the Time-Triggered Architecture (TTA) (Kopetz and Bauer, 2003). Formal analysis (Rushby, 2002) and experiments (Ademaj et al., 2003) have demonstrated that the TTA with its communication protocol TTP is appropriate for the implementation of applications in the highest criticality class (e.g., class A according to RTCA DO-178B (RTCA, 1992)).

The realization of TMO on TTP platforms is an integration of two complementary time-triggered technologies that work at different levels. The TTP technology offers formally verified architectural services, such as a clock synchronization service with high precision, a communication service with low latency and low jitter, and error containment between components. The TMO technology builds higher-level services by making use of the services of TTP platforms and wrap them onto a uniform Application Programming Interface (API) for real-time distributed computing object programming. It exploits the services of TTP in exchanging messages for remote method invocations and in realizing logical multicast communication channels, as well as in global synchronization of the executions of application and middleware threads.

Furthermore, the integration of TMO and TTP enables distributed real-time object computing for applications with sub-millisecond-level temporal precision requirements. Low transmission latencies and low jitter are important factors for realizing a high degree of quality-of-service in control loops. In addition, a global time base with a precision in the range of a few microseconds enables design of a certain class of real-time applications in the form of distributed object computing, which was not possible before (Kim et al., 2005).

This paper is structured as follows. Section 2 gives an overview of the TMO model and describes the requirements for an underlying platform. The TTA, which is employed as the platform for TMO in this paper, is the focus of Section 3. In Section 4, we discuss the Kernel Abstraction Layer (KAL) for the TTP/RTAI-Linux platform. Section 5 provides results on the temporal performance of the implementation and discusses the merits of integrating TMO and TTP.

## 2. TIME-TRIGGERED MESSAGE-TRIGGERED OBJECTS

TMO is a natural and syntactically minor but semantically powerful extension of conventional objects (Kim, 2000). Significant extensions realized by the TMO scheme are summarized below.

(1) *Globally referenced time base*: All time references in a TMO are references to global time in that their meaning and correctness are unaffected by the location of the TMO.

(2) *Distributed computing component*: TMOs distributed over multiple nodes may interact via remote method calls. TMOs can also interact via logical multicast channels called Real-time Multicast and Memory-replication channels (RMMC).

(3) *Spontaneous method (SpM)*: TMO introduces a new type of methods, spontaneous methods (SpMs) (also called time-triggered methods), which are clearly separated from the conventional service methods (SvMs). The SpM executions are triggered by the real-time clock at points in time specified as constants during design time, whereas the SvM executions are triggered by service request messages from clients.

(4) *Basic concurrency constraint (BCC)*: This rule prevents potential conflicts between SpMs and SvMs and reduces the designer's efforts in guaranteeing timely service capabilities of TMOs. Under BCC, the activation of an SvM triggered by a message from an external client is allowed only to the extent of not disturbing any SpM execution.

(5) *Guaranteed completion time (GCT) and deadline for result arrival:* The TMO incorporates deadlines in the most general form. Basically, for output actions and method completions of a TMO, the designer guarantees and advertises execution time-windows bounded by start times and completion times. In addition, deadlines can be specified in the client's calls for SvMs for the return of the service results.

## 2.1 Basic Structure of TMOSM

TMO Support Middleware (TMOSM) is a TMO execution support middleware model which can be easily adapted to most COTS platforms. Prototype versions of TMOSM currently exist for Windows XP, Windows CE and Linux. Within TMOSM, the innermost core is a super-micro thread called the WTST (Watchdog Timer & Scheduler Thread). It is a "super-thread" in that it runs at the highest possible priority level. It is also a "micro-thread" in that it manages the scheduling / activation of all other threads in TMOSM.

## 2.2 Requirements for an Underlying Platform

TMOSM relies on many system services provided by the underlying OS kernel in order to complete its tasks. The OS-dependent functionality has been grouped into a separate module called Kernel Adaptation Layer (KAL) so that the porting work can be easily achieved by plugging in a new native implementation of KAL built on the host OS.

The KAL specification precisely defines the functionality and timeliness requirements imposed on the underlying platform (i.e., hardware, OS kernel and C++ compiler) by TMOSM. The major components of KAL are described below:

(1) Time services: The scheduler in TMOSM, WTST, relies on a timer to trigger its execution. Since all threading activities in TMOSM are scheduled by WTST, the resolution and the precision of the timer are crucial to the performance of TMOSM. To achieve reasonable performance, TMOSM demands a timer with at least 100 microsecond-level precision. In addition, KAL should support the creation and setting of a timer object representing the current system time. This timer must have a 64-bit horizon and a resolution of one microsecond.

(2) Thread services: TMOSM has two major requirements regarding the threading support by KAL. First, KAL must be able to create a thread with the highest priority in the system. WTST needs to be given this priority so that it can run immediately each time it becomes ready and it may not be preempted by other threads. This is essential to guarantee the timely and predictable schedule of all activities in the TMOSM. Secondly, the thread should be light-weighted, namely, threads should share the same process address space. This is necessary in order to limit the overhead incurred in the context switches that occur with high frequency in TMOSM.

(3) Synchronization services: Events are primarily used to synchronize the activities of different threads in TMOSM. There are two major requirements regarding the event support by KAL. First, the delivery of an event must be reliable. Events cannot be lost even when multiple events are signaled simultaneously. The other requirement is that a thread can wait for multiple events, and the arrival of any of those events should resume the execution of the waiting thread.

(4) Communication services: KAL should support the main standard socket operations, in particular, sending and receiving UDP packets through a socket.

# 3. TIME-TRIGGERED ARCHITECTURE

The Time-Triggered Architecture (TTA) provides a computing infrastructure for the design and implementation of dependable distributed real-time systems (Kopetz and Bauer, 2003). Structuring rules guide the designer in the decomposition of the overall system into components that can fail independently. In addition, the TTA offers to system designers a validated stable baseline of generic architectural services for the development of applications.

## 3.1 System Structure

The basic building block of the TTA is a node computer, which is a self-contained composite hardware/software subsystem (system component). A cluster is a set of nodes that are interconnected by two redundant communication channels. For the interconnection of nodes, the TTA distinguishes between two physical interconnection topologies, namely a TTA-bus and a TTA-star. A TTA-bus consists of replicated passive buses. Every node is connected to two independent guardians, which use the a priori knowledge about the points in time of communication activities to prevent communication outside a node’s time-slot. In the TTA-star topology,

the interconnection of nodes occurs via two replicated central guardians. The star topology has the advantage of a higher level of independence, since guardians are located at a physical distance from nodes (Ademaj et al., 2003).

## 3.2 Architectural Services

The TTA provides four services essential for cost-effective construction of real-time fault-tolerant systems, which are implemented by the Time-Triggered Protocol (TTP):

- **Deterministic and Timely Transport of Messages.** The TTP protocol uses *Time Division Multiple Access* (TDMA) to control the media access to the two independent communication channels. The periodicity of the message transmissions is defined within a cluster cycle. Information about the points in time of all message transmissions during a cluster cycle is contained in a static *message schedule*. The cluster cycle is divided into a sequence of *TDMA rounds* and each TDMA round is partitioned into time slots. Slots are statically assigned to nodes in a way that allows each of them to send a message during the TDMA round.
- **Fault-Tolerant Clock Synchronization.** TTP provides fault-tolerant clock synchronization via the Fault-Tolerant Average (FTA) clock synchronization algorithm (Lundelius and Lynch, 1984). It differs from other algorithms in that no special synchronization messages are used for the exchange of the local clock values of nodes. The difference between the expected and the actual arrival times of an incoming message is used to estimate the deviation between the local clock of the receiver and that of the sender.
- **Strong Fault Isolation.** Although a *Fault Containment Region* (FCR) can demarcate the immediate impact of a fault, fault effects manifested as erroneous data must be prevented from propagating across FCR boundaries (Lala and Harper, 1994). For this reason, the TTA provides *error containment* within an FCR for message timing failures recognized within the fault hypothesis of the TTA (Kopetz, 2004). In addition, the TTA offers a reliable distributed computing platform by ensuring that a message is either consistently received by all correct nodes or detected as being faulty at all correct nodes.
- **Membership service.** The membership service provides nodes with consistent information about the operational state of every node in the cluster.

# 4. IMPLEMENTATION OF TMO ON THE TTA

This section describes the RTAI-Linux / TTP platform employed for the integration of TMO and TTP. In addition, we describe the adaptations of KAL in order to realize the platform services required by TMO. We have mapped the communication and time services of KAL to TTP, while realizing thread and synchronization services with RTAI-Linux operations.

## 4.1 Platform

The basic platform comprises a cluster of TTP monitoring nodes (TTTech, 2002) interconnected by a redundant 25 Mbps TTP network. Each node is equipped with a TTP C2 communication controller and a Motorola embedded PowerPC processor MPC855T. The MPC855T contains a PowerPC core and runs at 80 MHz. It is equipped with 16 MBytes of RAM, and 8 MBytes of flash memory. The TTP monitoring node uses the embedded real-time Linux variant, Real-Time Application Interface (RTAI) (Beal et al., 2000), as its OS kernel. RTAI combines a real-time hardware abstraction layer (RTHAL) with a real-time application interface for making Linux suitable for hard real-time applications.

## 4.2 Kernel Abstraction Layer

The Kernel Abstraction Layer (KAL) provides classes for accessing the OS kernel services from the TMO support middleware. The purpose of KAL is to simplify the porting of the TMO execution engine to different kernel platforms.

### 4.2.1 Time Services

The time services of KAL support the creation and destruction of time objects, the setting of time handlers, and the determination of the current time. KAL timers support a "horizon" of 64 bits and a granularity that depends on the quality of the employed clock synchronization algorithm. For adapting TMO to TTP, we have extended the horizon of the TTP global time (see Section 3.2) from 16 bit to 64 bit at a granularity of 5µs. This horizon corresponds to $2.9 \cdot 10^6$ years, thus preventing a timer overflow within the lifetime of the system.

Another purpose of the time services is the periodic activation of WTST which is in turn responsible for causing the scheduling of the other middleware and application threads. WTST uses the synchronization services of KAL in order to wait for multiple events, including an event triggered by the signaled timer. The TTP handler does the signaling of this event at the beginning of each TDMA round, which has a duration of 3 ms in the TTP communication schedule. This way, the scheduling of WTST is synchronized with the underlying time-triggered TTP communication.

### 4.2.2 Thread Services

The realization of KAL's thread services involves RTAI operations for the creation of fixed-priority, light-weighted processes designated as *real-time tasks*. RTAI priorities for real-time tasks range from 0 (highest), which is assigned to WTST, to 0x3fffFfff (lowest). Linux is assigned priority 0x7fffFfff, which is considered a non-real-time task.

### 4.2.3 Synchronization Services

RTAI does not provide events that can directly meet KAL's requirements regarding synchronization services. Instead, semaphores are used, as illustrated in Figure 1. Setting an event and waiting for an event are easily achieved by signaling and waiting for an associated semaphore. When a thread waits for multiple events, it actually waits for one semaphore shared by multiple events, e.g. multiple events are represented by the same semaphore. In fact, WTST is the only thread in TMOSM that waits for multiple events. Thus, only one shared semaphore needs to be created during the TMO engine start-up. In addition, a 'local' semaphore is created every time a new event is created by the middleware during execution time, and this semaphore represents that event only. The decision of associating an event to its local semaphore or with the shared semaphore is taken when a thread starts waiting for that event. If the event is one of multiple events the thread is waiting for, it will be associated with the shared semaphore. If the thread is waiting for that event only, the event will be associated with its local semaphore. This way, when the middleware sets an event, KAL knows which semaphore to signal.

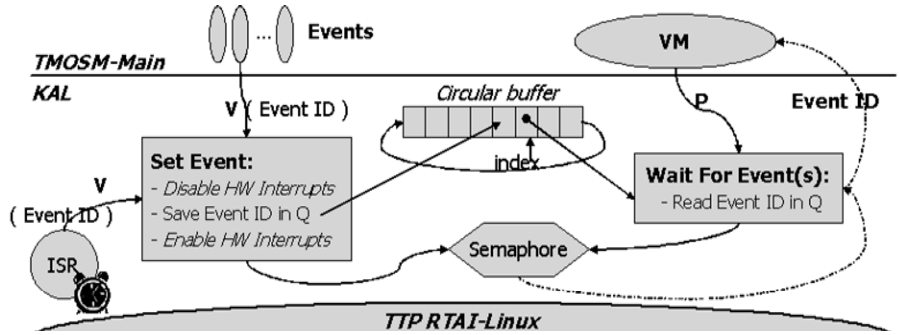


*Figure 1.* Threads Synchronization Using Semaphores

Also, to ensure that no event gets lost when several of them occur at the same time, events are recorded into a circular buffer inside KAL when they are set. Operations on the event buffer are protected by disabling all hardware interrupts.

### 4.2.4 Communication Services

KAL provides to TMOSM-Main a socket interface, which is the de facto standard API for TCP/IP. Sockets can be dynamically created and destroyed and each socket is assigned a local name via a port number. The socket is assumed to queue messages in order to process messages exactly once.

TTP, on the other hand, provides a temporal firewall interface (Kopetz, 1997) for the periodic exchange of state variables. The sender deposits information into the Communication Network Interface (CNI) according to the information push paradigm, while the receiver must pull information out of the CNI. The TTP protocol autonomously carries the state information from the CNI of the sender to the CNI of the receivers at a priori specified global points in time. Since applications are often interested in the most recent value of a real-time entity, old state values are overwritten by newer state values.

The CNI of a TTP node is structured into smaller state variables. For each TDMA round and each node in the cluster, the CNI contains a dedicated state variable. At a particular node n the state variable associated with n at this node's CNI is written by the application, while all other state variables are read by the application. The communication system behaves inversely, reading the state variable associated with n before broadcasting its contents via the network. All other state variables are updated with the contents of messages received from other nodes. The global point in time of a message reception not only denotes the identity of the sending node, but also determines the state variable in the CNI that is to be overwritten.

Since the temporal firewall interface of TTP represents a purely time-triggered communication service, we have implemented event-triggered sockets on top of TTP in KAL. Two queues are associated with each socket, namely an incoming and an outgoing message queue. TMOSM-Main inserts messages that need to be broadcast into the outgoing queue, while KAL removes messages from this queue and puts them into the CNI. The incoming message queue contains messages that have been received from other nodes.

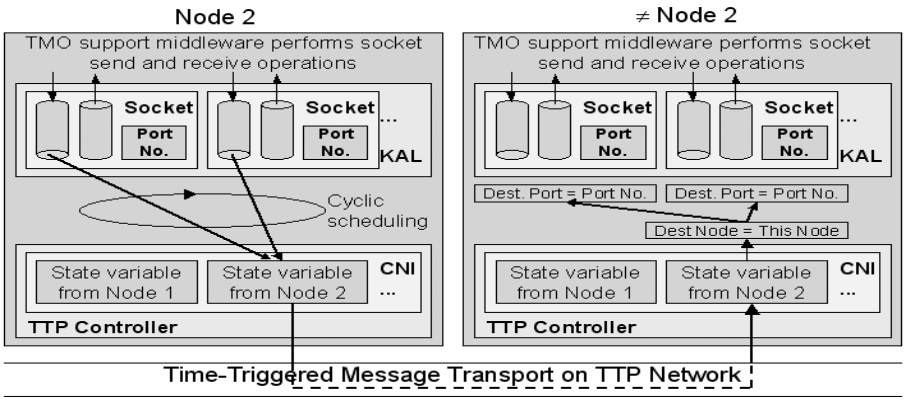


*Figure 2.* Transmission at Node 2 (left) and Reception by Another Node (right)

As depicted in Figure 2, a node's slot in the TDMA scheme is shared among the sockets in that node. For sending messages to other nodes, KAL goes through all sockets in the node in a circular fashion. If the currently checked socket contains a message that needs to be broadcast, then KAL copies the message into the CNI of the TTP controller. Consequently, the message will be autonomously transferred to the CNIs of all other nodes by the underlying TTP protocol.


*Figure 3.* Message Format

Figure 3 depicts the syntactic structure of a message stored in the CNI. In addition to the 16 bit TTP header, each message contains information about its length, a source address, a destination address, and up to 183 data bytes. The destination address comprises the 8 bit identifier of the receiver TTP node in combination with a 16 bit port number. The source address is specified as a 16 bit port number. Since TTP sends all messages of a node exclusively within the slots reserved for the respective node, the identity of the sender is implicitly known.

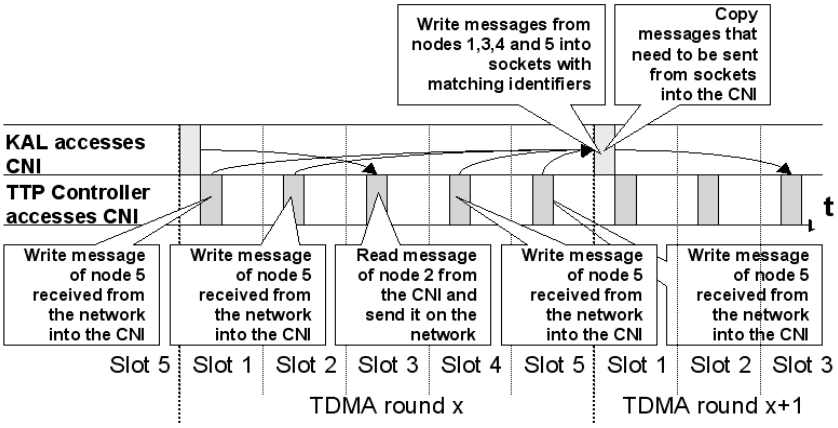


*Figure 4.* Activities at Node 2

In order to copy messages between the event message queues of the sockets and the TTP CNI as described above, KAL is activated at the beginning of each TDMA round. In any particular node n, KAL reads the CNI areas belonging to the nodes other than node n (local node). In addition, KAL writes the CNI area belonging to node n, if node n needs to send during the starting TDMA round. KAL accesses the CNI in time-triggered manners and is thereby implicitly synchronized with the TTP communication controller. Figure 4 illustrates this scheme for

node 2 in an ensemble of five nodes. This design prevents the transmission and reception of partially updated messages.

# 5. DISCUSSION

This section discusses the temporal performance of TMOs on top of the TTA and the benefits with respect to dependability that are gained through the reliable underlying time-triggered architecture.

## 5.1 Temporal Performance

While control algorithms can be designed to compensate a known delay, delay jitter brings an additional uncertainty into a control loop that has an adverse effect on the quality of control (Kopetz, 1997). In case of low jitter or a global time-base with a good precision, state estimation techniques allow to compensate a known delay range between the time of observation and the time of use of the observed image of the real-time entity. State estimation uses a model of a real-time entity to compute the probable state of the real-time entity at a future point in time. In systems of TMOs layered on top of TTA, clock synchronization is performed at the hardware level resulting in a global time base with a precision of 5 μs. Furthermore, communication jitter can be minimized by devising conflict-free time-triggered schedules at design time for both processing activities (e.g., SpM methods) and communication activities (e.g., messages for remote method invocations and those exchanged via multicast channels).

## 5.2 Improved Dependability

The integration of TMO and TTP improves the reliability of application TMOs through the fault-tolerance mechanisms of the underlying TTA services (i.e., fault-tolerant communication service, fault-tolerant clock synchronization, fault isolation between components). The fault isolation capability of the TTA ensures that a fault within the fault hypothesis of the TTA (Kopetz, 2004) does not impact the temporal behavior and the consistency of message exchanges between correct components.

Based on these properties of the architecture, it is possible to realize fault-tolerant TMO applications with active redundancy with relative ease. An example for such a configuration is Triple Modular Redundancy (TMR), which comprises three replica deterministic TMOs that compute the same outputs on different components that fail independently. Due to the consistency guarantees and the determinism of the TTA, such a configuration can mask an arbitrary failure of a single component, when performing an exact voting on the outputs of the three replicas.

# 6. CONCLUSION AND FUTURE WORK

The TMO model is a powerful extension of distributed object-computing for the domain of real-time systems. TMO supports the specification of temporal constraints in a natural form and provides a model for execution engines for ensuring that these constraints are met at runtime.

The establishment of a TMO execution engine on top of a base architecture suited for applications classified to be at the highest criticality levels (e.g., class A according to RTCA DO-178B) is an important step towards enabling economical construction of safety-related and safety-critical applications and effective management of growing complexity in such constructions. The TTA, which has been employed as the basis for the realization of TMO in this paper, is such an architecture. Its constituent communication protocol TTP has been validated through formal analysis and fault injection experiments.

A considerable side effect of the TMO / TTP integration is the availability of a global time base with microsecond granularity to TMO applications, as well as multicast communication channels and remote method invocations with small jitter. Jitter-sensitive applications (e.g., con-

trol loops) and applications with the requirement for high-precision timestamping can benefit from this improvement.

## ACKNOWLEDGEMENTS

This work has been supported in part by the European IST project DECOS under project No. IST-511764 and by the NSF under Grant Number 03-26606 (ITR).

## REFERENCES

Ademaj, A., Sivencrona, H., Bauer, G., and Torin, J., Evaluation of fault handling of the Time-Triggered Architecture with bus and star topology. In *Proc. of the International Conference on Dependable Systems and Networks*, 2003.

Beal, D., et al., RTAI: Real-time application interface. *Linux Journal*, April 2000.

Gimenez, G., and Kim, K.H., A Windows CE implementation of a middleware architecture supporting time-triggered message-triggered objects. In *Proc. of 25th Annual International Computer Software and Applications Conference*. 2001.

Kim, H.-J., Park, S.-H., Kim, J.-G., Kim, M.-H., and Rim, K.W., TMO-Linux: a Linux-based real-time operating system supporting execution of TMOs. In *Proc. of 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. April 2002.

Kim, K.H., "APIs for Real-Time Distributed Object Programming", *IEEE Computer*, June 2000, pp.72-80.

Kim, K.H., and Ishida, M., An Efficient Middleware Architecture Supporting Time- Triggered Message-Triggered Objects and an NT-based Implementation. In *Proc. of 2nd IEEE CS International Symposium on Object-Oriented Real-time Distributed Computing*, 1999.

Kim, K.H. et al., Distributed Computing Based Streaming and Play of Ensemble Music Realized Through TMO Programming. In *Proc. of 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems*. 2005.

Kim, K.H. et al., Distinguishing Features and Potential Roles of the RTO.k Object Model. In *Proc. of IEEE Computer Society Workshop on Object-oriented Real-Time Dependable Systems*. Dana Point, October 1994.

Kopetz, H., and Bauer, G., The time-triggered architecture. *IEEE Special Issue on Modeling and Design of Embedded Software*. 2003.

Kopetz, H., *Real-Time Systems*, Design Principles for Distributed Embedded Applications. Kluwer Academic Publishers. 1997.

Kopetz, H., The Fault Hypothesis for the Time-Triggered Architecture. In *Proc. of the IFIP World Computer Congress*. 2004.

Lala, J.H., and Harper, R.E., Architectural principles for safety-critical real-time applications. *Proc. of the IEEE*, 82:25–40. 1994.

Lundelius, J., and Lynch, N., A new fault-tolerant algorithm for clock synchronization. In *Proc. of the 3d Annual ACM Symposium on Principles of Distributed Computing.* 1984.

RTCA. DO-178B: *Software Considerations in Airborne Systems and Equipment Certification.* Radio Technical Commission for Aeronautics, Inc. 1992.

Rushby, J., An overview of formal verification for the time-triggered architecture. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science, pages 83–105, Springer-Verlag, 2002.

Simon, H., *The Sciences of the Artificial*. MIT Press, 1996.

TTTech Computertechnik AG. TTP Monitoring Node – A TTP Development Board for the Time-Triggered Architecture, March 2002.

Venture Development Corporation (VDC). Current Practices and Emerging Requirements in the Automotive Vertical Market. *Embedded Developers' Demand and Requirements for Commercial OSs and Software Development Tools, Volume I.* Natick, 2003.