The Time-Triggered Paradigm

Wilfried Elmenreich, Günther Bauer, and Hermann Kopetz
Institut für Technische Informatik
Vienna University of Technology
Vienna, Austria
{wil,bauer,hk}@vmars.tuwien.ac.at

The time-triggered paradigm encompasses a set of concepts and principles that support the design of highly dependable hard real-time systems.

The concept of a sparse time base enables a consistent view of events and supports the identification of a global system state. As a prerequisite, clock synchronization is necessary across the distributed system. A concise description of interfaces in both, the value and the temporal domain supports a two-level component-based design approach. For dependability, fault-tolerance can be implemented by redundancy and fault containment concepts. The time-triggered communication provides a predictable timing behavior with guaranteed timing assumptions under full load and faulty conditions.

1 Introduction

Hard real-time computer systems are defined by the fact that they must provide a particular result at intended points in real time. The correctness of a result depends on its proper behavior in both, the time and the value domain. It follows that any real-time computer architecture or design methodology must be concerned with the issue of value correctness and the issue of temporal correctness.

There are two major design paradigms for implementing real-time systems, the event-triggered and the time-triggered approach. Simplified, an event triggered system follows the principle of reaction on demand. In such systems the environment enforces temporal control onto the system in an unpredictable manner (interrupts), with all the undesirable problems of jitter, missing precise temporal specification of interfaces and membership, scheduling etc. On the other hand, the event-triggered approach is well-suited for sporadic action/data, low-power sleep modes, and best-effort soft real-time systems with high utilization of resources. Event-triggered systems do not ideally cope with the demands for predictability, determinism, and guaranteed latencies – requirements that must be met in a hard real-time system. Time-triggered systems derive control by the global progression of time, thus use the concept of time in the problem statement as well as in the provided solution. This approach supports a precise temporal specification of interfaces and the implementation of "temporal firewalls" to protect error propagation via control signals. Time-triggered systems support membership identification, interoperability, and replica determinism.

The objective of this paper is to provide an introduction to the time-triggered approach. The remainder of this paper is structured as follows: Section 2 introduces the concept of a sparse time base, the notion of state, real-time entities and real-time images, and discusses the difference between state and event information. Section 3 explains some principles of time-triggered systems such as a temporal firewall, composability, and dependability. Section 4 discusses properties of a time-triggered communication such as synchrony,

common communication schedule, and clock synchronization. The paper is concluded in section 5.

2 Related Concepts

2.1 Sparse Time

For most real time applications it is sufficient to model time according to Newtonian physics [1]. Hence, time progresses along a dense timeline, consisting of an infinite set of *instants* from past to future. Logical clocks, as introduced by Lamport in [2], usually are imprecise whenever physical time is essential. However, when global physical time is used to deduce causality of distributed events, it is necessary to synchronize the local clocks precisely.

Clock synchronization is concerned with bringing the time of clocks in a distributed network into close relation with respect to each other. A measure for the quality of clock synchronization are *precision* and *accuracy*. Precision is defined as the maximum offset between any two clocks in the network during an interval of interest. Accuracy is defined as the maximum offset between any clock and an absolute reference time.

Analysis of the border conditions allows to predict a bounded and known precision of an ensemble of synchronized clocks. However, the finite precision of the global time and the digitalization error make it impossible to guarantee that two observations of the same event will yield the same timestamp. In [3], a solution to this problem is provided by introducing the concept of a *sparse time base*. In this model the timeline is partitioned into an infinite sequence of alternating intervals of *activity* and *silence*. Figure 1 depicts the intervals of silence (s) and activity (a). The duration of the silence intervals depends on the precision of the clock synchronization.

The architecture must ensure that significant events, such as the sending of a message or the observation of an event, occur only during an interval of activity. Events occurring during the same segment of activity are considered to have happened at the same time. Events that are separated by at least one segment of silence can be consistently assigned to different timestamps for all clocks in the system.

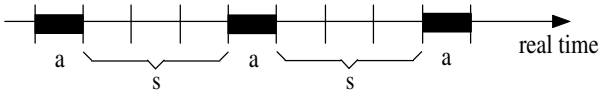

Figure 1: Sparse Time Base

While it is possible to restrict all event occurrences within the sphere of control of the real-time computer system to these activity intervals, the same is not possible for events happening in the environment, as for example, perceived by a sensor. Such events always happen on a dense timebase and must be assigned to an interval of activity by an agreement protocol in order to get a system-wide consistent perception of when an event happened in the environment [1].

2.2 Time and State

The concept of state is introduced in order to decouple the past from the future [4]. Since the state is defined for a given instant, the notion of time and state are inseparably coupled.

In real-time computer systems we distinguish between the *initialization state* (i-state) and the history state (h-state). The i-state encompasses the static data structure of the computer system, i. e., data that is usually located in the static (read-only) memory of the system. The i-state does not change during the execution of a given application. The h-state is the *dynamic data structure* [...] *that undergoes change as the computation progresses* [5]. The size of the h-state depends on the level of abstraction; for example determining the h-state at VLSI level will lead to a huge amount of information to be regarded, while the state of the same application might be also described at high-level language level yielding more compact state information.

The size of the h-state at a given level of abstraction may vary during the execution. A good system design will aim at having a *ground state*, i. e., when the size of the h-state becomes zero. In a distributed system, this usually requires that no task is active and no messages are in transit.

Replicated components usually need to synchronize their state, whereas a ground state enables an easy integration. The sparse time base supports a consistent perception of the h-state among replicated components.

Basically, the behavior of a (deterministic) system can be modeled as a function of its input and its current h-state (Figure 2). Every system can also be made stateless by making the state information explicit as depicted in Figure 3. Thus the system generates its state as part of its output message and receives the state as part of the input data. Such a configuration implies higher communication cost, but eases the integration of replicated components.

2.3 Real-time Entities and Images

The dynamics of a real-time application are modelled by a set of relevant state variables, so-called *real-time entities* that change their state as time progresses. Examples of real-time entities are the flow of a liquid in a pipe, the setpoint of a control loop or the intended position of a control valve. A real-time entity has static attributes that do not change during the lifetime of the real-time entity, and dynamic attributes that change with time. Examples of static attributes are the name, the type, the value domain, and the maximum rate of change. The value set at a particular instant is the most important dynamic attribute. An example of a dynamic attribute would be the rate of change at a chosen instant.

A continuous real-time entity can be observed at any instant while a discrete real-time entity can only be observed in a duration where the state of this real-time is not changing. A *real-time image* is a temporally accurate picture of a real-time entity. The validity of a real-time image is time-dependent and is invalidated by the progression of real-time.

2.4 State/Event Information

Information that is exchanged across an interface is either *state* or *event information*. Any property of a RT entity (i. e., a relevant state variable) that is observed by the real-time computer system is called a *state attribute* and the corresponding data *state information*. A *state observation* records the state of a state variable at a particular point of observation. A state observation can be expressed by the atomic triple

$$< Name, t_{obs}, Value >$$

consisting of the name of the observed state variable, the instant when the observation was made (t_{obs}), and the observed value of the real-time entity.

For example, the following statement contains a state observation: "*The position of control valve A was at 75° at 10:42 a.m.*".

At the sender, state information is not consumed on sending and at the receiver, state information requires an update-in-place and a non-consumable read. State information is transmitted in *state messages*.

A sudden change of state of an RT entity that occurs at an instant is an *event*. Information that describes an event is called event information. Event information contains the *difference* between the state *before* the event and the state *after* the event. An event observation can be expressed by the atomic triple

$$< Name, t_{event}, Value\ difference >$$

consisting of the name of the observed state variable, the instant of the event (t_{event}), and the value difference.

For example, the following statement contains an event observation: "*The position of control valve A changed by +5° at 10:42 a.m.*".

Event observations require exactly-once semantics when transmitted to a consumer. Events must be queued on sending and consumed on reading. Event information is transmitted in *event messages*.

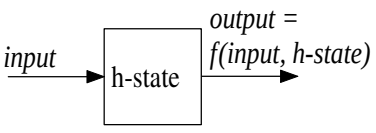

Figure 2: Implicit h-state

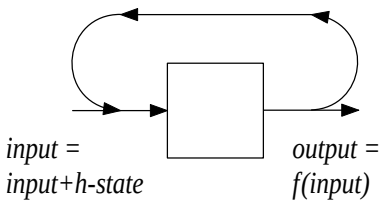

Figure 3: Stateless component with explicit h-state

2.5 Flow Control

Communication between subsystems exchanges information in two distinct domains, the *time domain* and the *value domain*. In the value domain the message data is transmitted, while in the time domain *control information* is exchanged [6]. Control information allows the generating subsystem to influence the *temporal control flow* [5] of the other subsystem.

Commonly a communication between two subsystems is either controlled by the sender's request (*push* style) or by the receiver's request (*pull* style) [7].

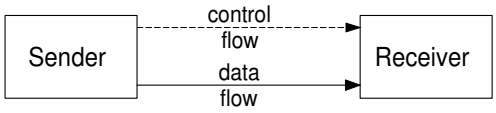

Figure 4: Push Communication Model (implicit flow control)

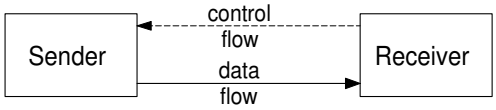

Figure 5: Pull Communication Model (explicit flow control)

For explanation let us assume two components that need to exchange data over a network. Further, without restrictions to generality, we assume the message data to be transmitted from a *producer* component to a *consumer* component.

In order to transfer data between two components, they must agree on the flow control mechanism to use and the direction of the transfer.

Figure 4 shows the push method. The producer is allowed to generate and send its message at any time, thus flow control is managed by the producer. This method is very comfortable for the push producer, but the push consumer has to be watchful for incoming data messages at any time, which may result in high resource costs and difficult scheduling [8]. Popular "push" mechanisms are: messages, interrupts, or writing to a file [9]. The push style communication is the basic mechanism of event-triggered systems.

In Figure 5 the flow control is on the consumer. Whenever the consumer asks to access the message information, the producer has to respond on the request. This facilitates the task for the pull consumer, but the pull producer has now to be watchful for incoming data requests [8]. Popular "pull" mechanisms are: reading a file, polling, state messages, or shared variables [9]. The pull style communication is the basic mechanism of client-server systems. The Time-Triggered Paradigm implements a hybrid approach which is explained in Section 3.1.

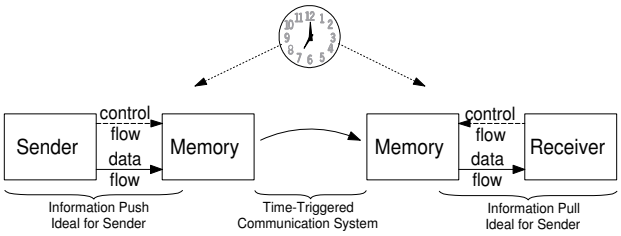

Figure 6: Temporal firewall

3 Design Principles

3.1 Temporal Firewall

A temporal firewall [10] is a fully specified interface for the unidirectional exchange of state information between a sender/receiver over a time-triggered communication system. The basic data and control transfer using a *temporal firewall* interface is depicted in Figure 6. The interface enables different control flow directions between sender and receiver.

This model uses a combination of Push and Pull communication model. Each component possesses a memory object that acts as a data source and sink for communication activities. Components that want to submit data are able to write the data into this memory using a *producer's push* interface. The transmission of data between message data is handled by a time-triggered communication model. After transmission the consumer component accesses the data using a *consumer's pull* interface.

The values in the memory are state messages that keep their content until they are updated and overwritten. The critical ends of the push and the pull communication rely on the memory elements. However, because these elements are usually passive components, the negative effects

of the push and the pull communication do not affect the system performance.

To avoid interference between concurrent read and write operations on the memory element, the task of the communication system is done by a time-triggered protocol as described in Section 4. Since in the time-triggered architecture all nodes have knowledge about transmission schedules and access to a global time base, the instant when the protocol updates a value in the memory element is known to all components.

3.2 Global Time

The global synchronized time is a requirement and a feature in time-triggered systems. The global time must be established by a periodic clock synchronization in order to enable a time-triggered communication and computation.

At the start-up of a cluster, an *initial clock synchronization* is necessary in order to move the system from an asynchronous state with unknown phase shifts between the node's clocks and an unknown number of nodes willing to perform an initial synchronization. The initial clock synchronization depends on asynchronous event-triggered message exchange, but it is possible to give a conditional timeliness guarantee. If there is already an ensemble of nodes that are synchronized to each other an unsynchronized node may join this set. Such an integration procedure can be achieved with guaranteed timeliness. A continuous synchronization of clocks is necessary due to inevitable drifts between the local clock sources of distinct nodes.

The synchronized clocks establish the global time of the cluster. The global time is defined by a granularity of g, whereas the precision of the clock synchronization must less than g in order to have a *reasonable* global time. The global time is used to define the instant of an event or to initiate coordinated actions such as access to a shared resource.

3.3 Composability

In a distributed real-time system the nodes interact via the communication system to distributely execute a global application. This application depends on the timely provision of the real-time information at the real-time interface of the nodes. For an architecture to be composable in the temporal domain, it must adhere to the following four principles with respect to the real-time service interface [11]:

Independent Development of Nodes: Nodes can only be designed independently of each other, if the architecture supports the precise specification of all node services at the level of architecture design. In a real-time system, the real-time service interface must be concisely specified in the value domain and in the temporal domain. Furthermore, the node service, as viewed by the host of the node, has to by described by a proper abstract model. This knowledge about this model is a prerequisite for the independent development of the node software.

Stability of Prior Services: The stability-of-prior-service principle ensures that the validated service of a node – both in the value domain and in the time domain – is not refuted by the integration of the node into a system. For example, the integration of a self-contained node, e.g., an engine controller, into the integrated vehicle control system may require additional computational resources (both in processing time and in memory space) of the node to service this new communication interface. In such a situation, failures in the node's prior services may occur sporadically during and after the integration.

Constructive Integration: The constructive integration principle requires that if n nodes are already integrated, the integration of the $n + 1$. node must not disturb the correct operation of the n already integrated nodes. The constructive-integration principle ensures that the integration activity is linear and not circular.

This constructive integration principle has severe implications for the management of the network resources. If network resources are managed dynamically, it must be ascertained that even at the critical instant, i.e., when all nodes request the network resources

at the same instant, the timeliness of all communication requests can be satisfied. Otherwise sporadic failures will occur with a failure rate that is increasing with the number of integrated nodes.

Replica Determinism: If fault tolerance is implemented by the replication of nodes, then the architecture and the nodes must support replica determinism. A set of replicated nodes is replica determinate [12] if all the members of this set have the same externally visible state, and produce the same output messages at points in time that are at most an interval of d time units apart (as seen by an omniscient outside observer). In a fault-tolerant system, the time interval d determines the time it takes to replace a missing message or an erroneous message from a node by a correct message from redundant replicas. The implementation of replica determinism is simplified if all nodes have access to a globally synchronized sparse time base.

3.4 Dependability

Dependability of a computer system is defined as *the trustworthiness and continuity of computer system service such that reliance can justifiably be placed on this service* [13, page 41]. Dependability is an overall term that includes availability, reliability, safety, maintainability, and security [14].

Requirements for highly dependable systems can only be met, if faults are taken into account. In order to tolerate faults in a distributed system, two design approaches can be identified [15]:

Redundancy: The system contains redundant components that enable the system to provide its service despite the presence of faults. For example, the system could provide several hardware units that offer the same service, thus allowing the provision of a service despite faults.

Recovery: The system's software is designed to be able to detect and recover from faults. Compared to the hardware redundancy approach this approach does not need extra hardware, but the time for recovery has to be taken into account.

In the time-triggered paradigm, both approaches are supported. Hardware redundancy provides a transparent fault-tolerant service, while software recovery is supported for recovery from transient failures. Field studies show that the probability of transient failures to occur is much higher then the probability for permanent failures [16].

The fault-tolerant service can be maintained only if the environment complies with the fault hypothesis, which consists of the assumptions taken on the failure modes and likelihood of faults.

If the environment violates the fault hypothesis – in a properly designed application this must be a rare event – then the system may fall back to a never-give-up strategy. The never-give-up strategy is initiated in combination with the application as soon as it becomes evident that there are not enough resources available any more to provide the minimum required service. The never-give-up strategy is highly application-specific. For example, if the cause of the outage is a massive transient fault, then in some applications the never-give-up strategy may consist of freezing the actuators in their current state until a successful restart of the whole system has been completed.

4 Time-Triggered Communication

The very basic principle of time-triggered communication is that the events of message transmission depend only on the progression of time and not on the availability of new information.

In a time-triggered distributed system, communication takes place according to a common periodic communication schedule. Each node has assigned durations for sending and receiving within the period.

All nodes must agree to the communication schedule and its temporal interpretation and to the beginning of the cluster cycle. Therefore, a time-triggered communication needs a sufficient synchronization among all nodes' clocks.

The instants at which information is delivered or received are a priori defined and known to

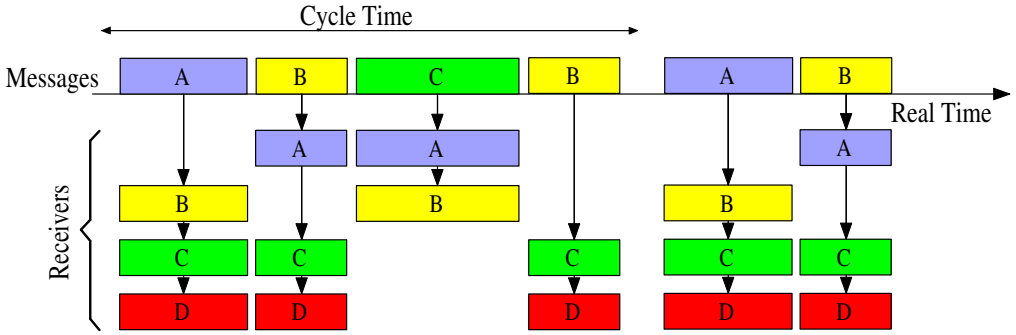

Figure 7: Time-Triggered Communication

all nodes of a cluster. These instants define the deadlines for the application tasks within a host. Knowing these deadlines, it is in the responsibility of the host to produce the required results before the deadline has passed. Any node-local scheduling strategy that will satisfy these known deadlines is "fit for purpose". It is the responsibility of the time-triggered communication service to transport the information from the temporal firewall of the sending node to the temporal firewall of the receiving node within the interval delimited by these a priori known fetch and delivery instants.

4.1 Communication Interface

From the point of view of complexity management and composability, it is useful to distinguish between three different types of interfaces of a node: the real-time service (RS) interface, the diagnostic and management (DM) interface, and the configuration and planning (CP) interface [17]. These interface types serve different functions and have different characteristics. For the temporal composability, the most important interface is the RS interface.

The RS interface provides the timely real-time services to the node environment during the operation of the system. In real-time systems it is a time-critical interface that must meet the temporal specification of the application in all specified load and fault scenarios. The composability of an architecture depends on the proper support of the specified RS interface properties (in the value and in the temporal domain) during operation. From the user's point of view, the internals of the node are not visible at the communication network interface, since they are hidden behind the RS interface.

The DM interface opens a communication channel to the internals of a node. It is used for setting node parameters and for retrieving information about the internals of the node, e.g., for the purpose of internal fault diagnosis. The maintenance engineer that accesses the internals of a node via the DM interface must have detailed knowledge about the internal objects and behavior of the node. The DM interface does not affect temporal composability. Usually, the DM interface is not time-critical.

The CP interface is used to connect a node to other nodes of a system. It is used during the integration phase to generate the "glue" between the nearly autonomous nodes. The use of the CP interface does not require detailed knowledge about the internal operation of a node. The CP interface is not time-critical.

4.2 Time-Triggered Protocols

Two communication protocols that follow the time-triggered paradigm are the fault-tolerant TTP/C [18] protocol and the low-cost fieldbus protocol TTP/A [19].

The TTP/C protocol provides a highly dependable real-time communication service with a fault-tolerant clock synchronization and membership service. TTP/C is suitable for X-by-wire systems in the automotive and avionics domain.

The time-triggered fieldbus TTP/A is intended for the integration of smart transducers in all types of distributed real-time control systems. Although the first target are automotive applications, TTP/A has been designed to meet the

requirements of process control systems as well. TTP/A supports low cost implementations on wide set of available component-off-the-shelf microcontrollers.

5 Conclusion

The time-triggered paradigm is a conglomeration of concepts and methods that have been elaborated in more than twenty years of research in the field of dependable distributed real-time systems. During this period, many ideas have been developed, implemented, evaluated, and often discarded. What survived is a small set of orthogonal concepts that center around the availability of a dependable global time-base. The guiding principle has always been to take maximum advantage of the availability of a global time, which always is part of the world, even if we do not use it.

Today, more and more dependable real-time architectures follow a time-triggered paradigm in order to take advantage of the predictable predictable timing behavior.

Acknowledgements

We would like to acknowledge the contribution of all the many PhD students, master students, sponsors, and industrial partners to the development of the concepts in this paper. At present the research on time-triggered systems at the Vienna University of Technology is supported by the European IST projects NEXT TTA (IST-2001-32111) and HRTC (IST-2001-37652).

References

[1] H. Kopetz and N. Suri. Compositional design of RT systems: A conceptual basis for specification of linking interfaces. Research Report 37/2002, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2002.

[2] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[3] H. Kopetz. Sparse time versus dense time in distributed real-time systems. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, Yokohama, Japan, June 1992.

[4] P. Peti. The concepts behind time, state, component, and interface - a literature survey. Research Report 53/2002, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2002.

[5] H. Kopetz. *Real-Time Systems, Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston, Dordrecht, London, 1997.

[6] A. Krüger. *Interface Design for Time-Triggered Real-Time System Architectures*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, April 1997.

[7] Alcatel Corp., Fujitsu Ltd., IBM, NEC Corp., NTT Corp., and IONA Tech. Management of event domains. *OMG TC Document telecom/2000-01-01*, January 2000. Available at http://www.omg.org.

[8] W. Elmenreich, W. Haidinger, and H. Kopetz. Interface design for smart transducers. In *IEEE Instrumentation and Measurement Technology Conference*, volume 3, pages 1642–1647, Budapest, Hungary, May 2001.

[9] R. DeLine. *Resolving Packaging Mismatch*. PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, June 1999.

[10] H. Kopetz and R. Nossal. Temporal firewalls in large distributed real-time systems. *Proceedings of the 6th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS '97)*, pages 310–315, 1997.

[11] H. Kopetz. The temporal specification of interfaces in distributed real-time systems. In *Proceedings of the EMSOFT 2001*, pages 223–236, Tahoe City, CA, USA, October 2001.

[12] S. Poledna. *Replica Determinism in Fault-Tolerant Real-Time Systems*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 1994.

[13] W. C. Carter. A time for reflection. In *Proceedings of the 12th IEEE Int. Symposium on Fault-Tolerant Computing (FTCS-12)*, page 41, Santa Monica, CA, USA, June 1982.

[14] J. C. Laprie. Dependability: Basic concepts and terminology. In *Dependable Computing and Fault Tolerant Systems*, volume 5, pages 257–282. Springer Verlag, Vienna, 1992.

[15] G. Bauer. *Transparent Fault Tolerance in a Time-Triggered Architecture*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2001.

[16] W. Steiner and W. Elmenreich. Automatic recovery of the TTP/A sensor/actuator network. In *Proceedings of the First Workshop on Intelligent Solutions for Embedded Systems*, pages 25–37, Vienna, Austria, June 2003.

[17] H. Kopetz, M. Holzmann, and W. Elmenreich. A universal smart transducer interface: TTP/A. *International Journal of Computer System Science & Engineering*, 16(2):71–77, March 2001.

[18] H. Kopetz. *Specification of the TTP/C Protocol*. TTTech, Schönbrunner Straße 7, A-1040 Vienna, Austria, July 1999. Available at http://www.ttpforum.org.

[19] H. Kopetz et al. Specification of the TTP/A protocol. Technical report, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, March 2000. Available at http://www.ttpforum.org.