Proceedings of the
47th IEEE Conference on Decision and Control
Cancun, Mexico, Dec. 9-11, 2008

Time-robust discrete control over networked Loosely Time-Triggered Architectures

Paul Caspi and Albert Benveniste

***Abstract*— In this paper we consider Loosely Time-Triggered Architectures (LTTA) as a networked infrastructure for deploying discrete control. LTTA are distributed architectures in which 1/ each computing unit is triggered by its own local clock, 2/ the local clocks are not synchronized, and 3/ communication is by the following principle: each communication link acts as a shared and sustained variable that can be, at will, written by the source node and read by the destination node. The loose communication medium used can cause duplication and/or loss of events, as well as distorsion of the synchronization between events occurring at different nodes of the network. While LTT architectures possess significant advantages, their use for distributed discrete control raises serious difficulties.**

Together with other authors, the authors of this paper have proposed a comprehensive design methodology ensuring the preservation of semantics, from specification to implementation over LTTA. This technique uses sophisticated token based protocols alike so-called elastic circuits recently introduced for asynchronous hardware.

In this paper we propose a completely different approach, with no flow of token, and entirely time based. Our approach relies on upsampling and suitable use of local counters. We prove the preservation of semantics, from specification to implementation on LTTA with this technique, and we study its performance. An extended version of this paper with more results is found in [5].

Keywords: real-time systems, distributed control, time-triggered, loosely time-triggered, MoCC.

I. INTRODUCTION

The advantages of using synchronous models to describe design functionality of embedded systems are well known. However, this comes at a price: it is hard to implement synchrony, especially on distributed execution platforms. A solution for implementing synchronous models on distributed platforms is to use a clock synchronization protocol (e.g., see [10], [11]) to synchronize the clocks of the different execution nodes of the distributed architecture. This approach is followed by the Time Triggered Architecture [8]. Techniques for generating semantic-preserving implementations of synchronous models on TTA have been studied in [6]. However, this approach carries cost and timing penalties that may not be acceptable for some applications. In particular, TTA is not easily implementable for long wires (such as in systems where control intelligence is widely distributed) or for wireless communications.

This research was supported in part by the European Commission under the Networks of Excellence IST-2001-34820 ARTIST and IST-004527 ARTIST2, and by IST STREP 215543 COMBEST.

P. Caspi is with VERIMAG/CNRS, 2 avenue de Vignate, 38610 Gières, France;

A. Benveniste is with INRIA-Rennes/IRISA, Campus de Beaulieu, 35042 Rennes cedex, France albert.benveniste@inria.fr

Hence, there has been growing interest in less constrained architectures, such as the Loosely Time-Triggered Architecture (LTTA) [2]. LTTA is characterized by a communication mechanism, called *Communication by Sampling* (CbS), which assumes that: 1/ writings and readings are performed independently at all nodes connected to the medium, using different local clocks; and 2/ the communication medium behaves like a shared memory. This architecture is very flexible and efficient as it does not require *any* clock synchronization, and it is not blocking both for writes and reads. Consequently, risk of failure propagation throughout the distributed computing system is reduced and latency is also reduced albeit at the price of increased jitter. However, data can be lost due to overwrites or alternatively duplicated because reader and writer are not synchronized. If, as in safety critical applications that involve discrete control for operating modes or handling protection, data loss is not permitted, then special techniques must be developed to preserve the semantics of the specification.

LTT architectures are widely used in embedded systems industries. The authors are personally aware of cases in aeronautics [13], nuclear, automation, and rail industries where the LTTA architecture with limited clock deviations has been used with success. The LTT bus based on CbS was first proposed in [2] and studied for a single writer-reader pair and [12] proposes a variation of LTTA where some master-slave re-synchronization of clocks is performed. More recently, LTT architecture of general topology was studied in [1], [14], using techniques reminiscent from so-called back-pressure [4], [3] and the related elastic circuits [7].

In a different direction, [9] develops an alternative approach where up-sampling is used in combination with "thick" events as a way to preserve semantics. The approach we present in this paper extends and clarifies the above one.

The paper is organized as follows. Artifacts caused by CbS communication are discussed in Section II. The problem is set in Section III and solved in Section IV. An extended version of this paper with more results is found in [5].

II. ARTIFACTS CAUSED BY CBS COMMUNICATION

Throughout this paper, we consider an architecture as shown in Figure 1. In this figure, a distributed architecture for discrete control is shown. Communication medium (figured here as the bus filled in grey) operates by sampling, meaning that:

Assumption 1:

1) the communication medium behaves like a collection of shared memories, one for each variable;

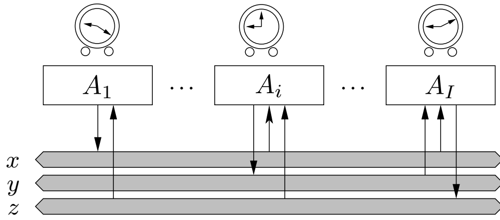

Fig. 1. The considered LTT Architecture. Three shared memories are depicted as shaded buses, for the three variables x, y, and z. For each variable, there is one writer and zero or more readers.

2) updates of every variable are visible to every node;
3) writings and readings are performed independently at all nodes connected to the medium, using different, non synchronized, local clocks.

CbS communication can be formalized as follows, for in the input and out the output:

$$\text{CbS}[\tau] \quad : \quad \left\{ \begin{array}{rcl} m(t) & = & \max\{n \mid s_n + \tau \leq t\} \\ \nu(t) & = & in_{m(t)} \\ out_k & = & \nu(t_k) \end{array} \right. \tag{1}$$

In (1), indexes m and k count the ticks of the input and output clock, respectively, s_m and t_k are the dates of mth and kth corresponding clock ticks, and $\tau \geq 0$ is the communication delay (time needed for fetching data). Say that CbS is zero delay if $\tau = 0$ in (1). Compare (1) with $out_k = in_k$ which corresponds to ideal, zero time communication.

The problem when reading multiple signals is illustrated on Figure 2. We show here the case of A_1 reading two

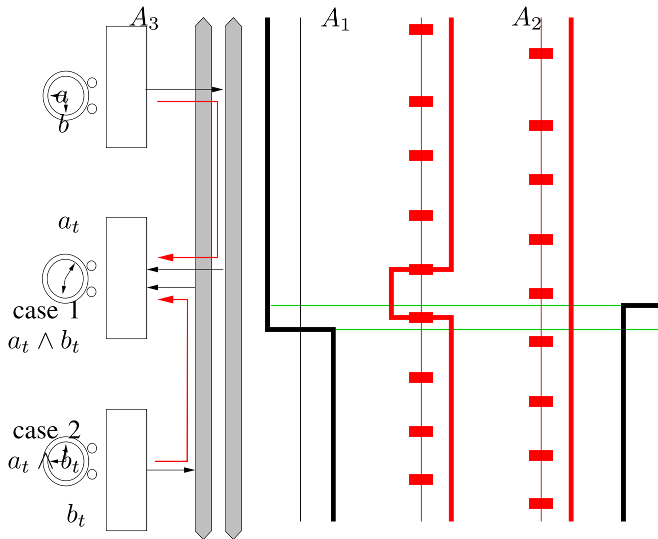

Fig. 2. Sensing multiple signals, for distributed clocks subject to independent drifts and jitters.

boolean inputs a_t and b_t originating from A_2 and A_3, respectively, and computing their conjunction $a_t \wedge b_t$. Cases 1 and 2 correspond to two different outcomes for the local clock of A_1. Observe that the result takes three successive values F, T, and F for case 1, whereas case 2 yields the constant value F. The origin of the problem is that events attached to different signals can be separated by arbitrary small time intervals—in Figure 2 the problem comes from the very close jumps for a_t and b_t. No reasonable assumption can prevent this from happening in a distributed setting.

III. Problem Setting

In this section we set the landscape and explain what the problem is that we want to solve. We first specify the application for deployment. Then we formalize the objective of our study, namely preserving application semantics.

A. The application

We are given an underlying set Z of *variables*. Our specifications for discrete controllers are modeled using dataflow diagrams. That is, they consist of a network of computing nodes of the following form:

$$\text{N} \quad : \quad \left\{ \begin{array}{rcl} X_k & = & f(X_{k-1}, u_k^1, \ldots, u_k^p) \\ y_k & = & g(X_{k-1}, v_k^1, \ldots, v_k^q) \end{array} \right. \tag{2}$$

In (2), k is the discrete time index, $u^1, \ldots, u^p, v^1, \ldots, v^q \in Z$ are the *input variables* of the node, $X \subseteq Z$ is the tuple of *state variables* of the node, $y \in Z$ is the output variable of the node, and f, g are functions.

Our model seems to require that every input and output flow is involved in all transitions of the discrete controller. This may not be always the case, however. In particular, in applications for deployment over LTT Architectures, events of interest are typically represented as changes in some of the variables. Such changes need not to occur at all instants k. For instance, that node N of formula (2) is not active at instant k is captured by the fact that all of its variables remain unchanged at that instant.

Nodes $\text{N}_1, \ldots, \text{N}_n$ can be composed by output-to-input connections to form *systems*, i.e., networks of nodes, denoted by

$$S \quad = \quad \text{N}_1 \parallel \ldots \parallel \text{N}_n \tag{3}$$

Systems can be further composed in the same way, denoted by

$$S_1 \parallel \ldots \parallel S_I. \tag{4}$$

Node (2) is abstracted as the following labeled directed graph:

$$\mathcal{G}(\text{N}) \quad : \quad \left\{ \begin{array}{lllll} X \overset{\text{UD}}{\leftarrow} X & , \; X \leftarrow u^1 & , \; \ldots & , \; X \leftarrow u^p \\ y \overset{\text{UD}}{\leftarrow} X & , \; y \leftarrow v^1 & , \; \ldots & , \; y \leftarrow v^q \end{array} \right. \tag{5}$$

A branch $y \overset{\text{UD}}{\leftarrow} X$ indicates that y depends on X through a Unit Delay, whereas a branch $y \leftarrow v$ indicates a direct dependency. Systems $S = \text{N}_1 \parallel \ldots \parallel \text{N}_n$ are abstracted as the union of the associated graphs $\mathcal{G}(S) = \mathcal{G}(\text{N}_1) \cup \ldots \cup \mathcal{G}(\text{N}_n)$, and the same holds, inductively, for $\mathcal{G}(S)$ when $S = S_1 \parallel \ldots \parallel S_I$.

Assumption 2: We require that, in $\mathcal{G}(S)$, no loop exists involving branches not labeled by delay symbols UD.

Referring to Figure 4, consider $\mathcal{G}(S) = \mathcal{G}(S_1) \cup \ldots \cup \mathcal{G}(S_I)$. Using Assumption 2, erasing, in $\mathcal{G}(S)$, branches labeled by delay symbols UD yields a partial order denoted by $\preceq$; set $\prec = \preceq \cap \neq$. For z a vertex of $\mathcal{G}(S)$, define its *level* $\ell(z)$ as being

$$\begin{array}{l}\text{the largest index } \ell \geq 0 \text{ such that a chain} \\ z_0 \prec z_1 \prec \ldots \prec z_\ell = z \text{ exists in } \mathcal{G}(S).\end{array} \quad (6)$$

The level is illustrated on Figure 3.

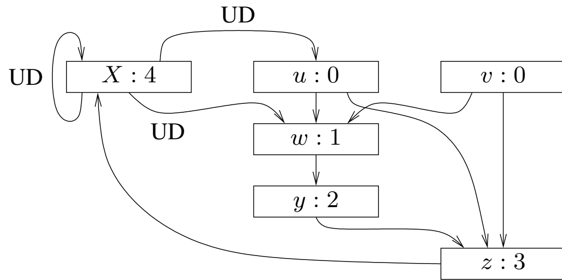

Fig. 3. Illustrating the level of a node.

B. Deployment and semantics preserving

Deploying the application $S_1 \parallel \ldots \parallel S_I$ of formula (4) is performed by mapping each sub-system S_i to the corresponding processor A_i. Consequently, communications between sub-systems are now implemented using CbS communication, as modeled by formulas (1).

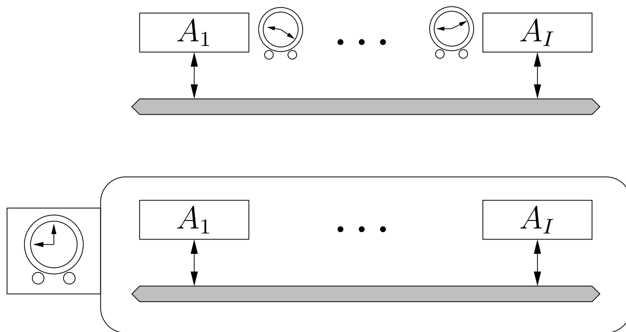

Fig. 4. The problem: showing flow equivalence between LTTA design (top) and strictly synchronous design (bottom). The set of all shared memories is depicted here as a single bus.

The problem we need to solve is explained in Figure 4. In this figure we show 1/ the LTT, CbS-based, Architecture of Figure 1, and 2/ the same architecture in which we assume that all clocks are strictly synchronized with ideal communication taking no time. Deployment of the application over the strictly synchronous architecture of Figure 4, bottom, is straightforward. The different computing units compute in lock steps—we call them *reactions*—according to the global clock. Each node N is assigned some computing unit for its execution. Then, the computation of the different variables is scheduled within each reaction, by respecting partial order $\preceq$. Note that such a scheduling can be statically defined byt need not be so.

In this method, variables can be updated at each reaction. Observe that we can instead update variables at every second (or third, etc.) reaction and keep silent otherwise. We can even cluster successive clock ticks together to form successive *macro-reactions* of the architecture. This can be, e.g., performed by downsampling the clock by a constant factor, so the macro-reactions involve a fixed number of clock ticks in this case. But a variable number of clock ticks can be also considered, for the successive macro-reactions. Now, inside each macro-reaction, one need to schedule the computation of the different variables by respecting partial order $\preceq$. This leaves room for computing different variables at different clock ticks.

All the above designs implement correctly the application function, which consists in mapping input streams to output streams as specified by formulas (2)–(4). We say that *application semantics is preserved.*

In the following section we show that the above notion of macro-reaction can be further adapted to the LTT Architecture of Figure 4, top, in such a way that application semantics is preserved. This will show that the two architectures of Figure 4 can be made equivalent, from the point of view of preserving semantics.

IV. Protocol design

In this section we develop a protocol ensuring that the LTT Architecture of Figure 4, top, preserves application semantics. We first begin by stating assumptions regarding clocks and communication delays.

A. Assumptions regarding clocks and communication delays

We assume a distributed set $(\kappa^i)_{i\in I}$ of quasi-periodic LTTA clocks. Quasi-periodicity consists in assuming the existence of lower and upper bounds $T_{\min}$ and $T_{\max}$ for the interval between any two successive ticks of any clock: $\forall k \in \mathbb{N}$:

$$T_{\min} < \kappa^i_k - \kappa^i_{k-1} < T_{\max} \quad (7)$$

Then, we assume communication delays to be bounded:

$$0 \leq \tau_{\min} \leq \tau \leq \tau_{\max} < +\infty \quad (8)$$

B. A simple protocol

Consider the following additional assumptions:

Assumption 3: Every communication between different sites occurs through state variables and is thus subject to a unit delay.

In the framework of Section III-A, this corresponds to a level 0 for all input nodes of the different systems, see formula (6) and Figure 3.

Assumption 4: For each computing unit, executions take at most one clock cycle and a computing unit which starts executing freezes its input data.

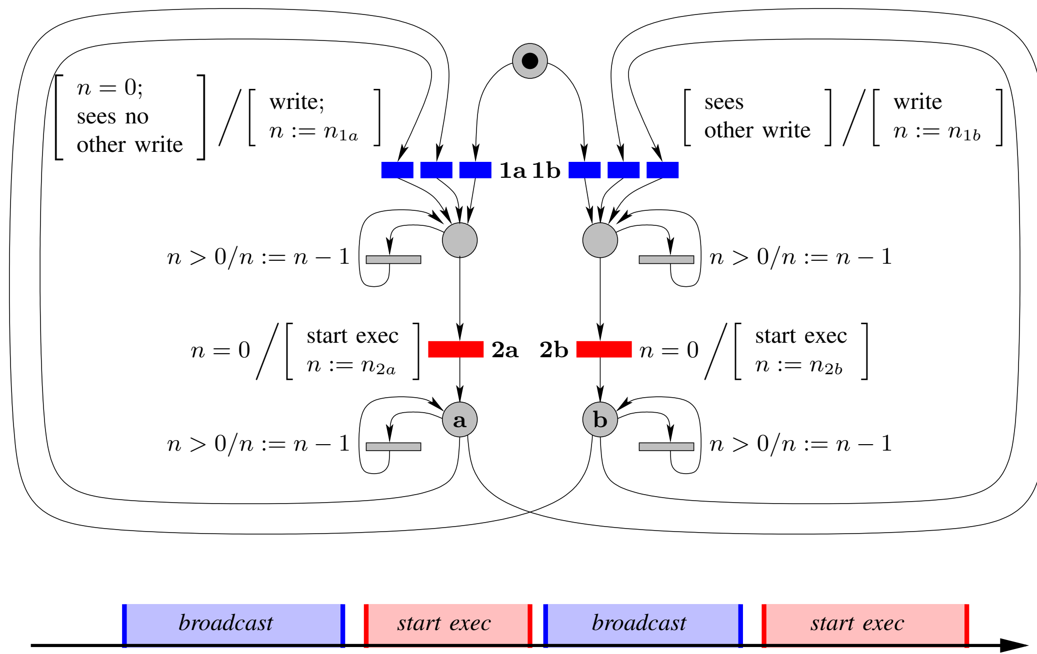

Fig. 5. Showing the protocol for a generic processor A_i (top) and the requirement (bottom). The same label is attached to each of the three blue sibling transitions.

We stress that communicated variables are not updated at each reaction of the application, but only when required by the application. Consequently, a processor does not know a priori which variable update it is supposed to see at a given reaction.

Protocol 1 (simple protocol): Assumptions 1–4 are in force. The protocol is shown on Figure 5.

Referring to figure 5, protocol 1 is modeled as an automaton whose transitions are labeled by actions and are guarded, either by observations from the CbS medium, or by conditions on the counter n. Thus, transitions are labeled by a pair *guard / action*; action is optional, whereas an empty guard means *true*. The net can progress at each tick of the local clock of processor A_i, guards permitting. The requirement is that real-time periods alternate in which *broadcast* events (shown in blue on the net) and *start executing* events (shown in red on the net) occur, for all processors.

Action "start exec" (attached to red transitions) indicates that processor A_i can start executing at the referred clock tick. Action "write" (attached to blue transitions) indicates that processor A_i possibly updates its outputs at the referred clock tick.[1] Guard "sees other write" indicates that, at the considered tick of its own clock, A_i sees an update of some output from some other processor. Guard "sees no other write" indicates that, at the considered tick of its own clock, A_i sees no update of some output from some other processor.

Observe that the guards of transitions outgoing from places (**a**) or (**b**) are not exclusive. Resulting nondeterminism is removed by specifying that the transition labeled with the guard "sees other write" has higher priority.

Proof: For Protocol 1 to be correct we must find conditions on the parameters $n_{1a}, n_{1b}, n_{2a}, n_{2b}$ ensuring that the following two requirements remain satisfied, see Figure 5:

- **Requirement 1 (start execution):** For each computing unit, execution starts after everybody's writes have been all read—this ensures that every computing unit executes with the same data.
- **Requirement 2 (broadcast):** For each computing unit, new writes start after every computing unit has started executing—this ensure that current and next logical reactions do not get mixed.

We prove by induction that the two requirements remain invariant. Our induction hypothesis is that requirements 1 and 2 are satisfied before time t.

a) Requirement 1: suppose the earliest update occurs at real-time t. Our induction hypothesis is that requirements 1 and 2 are satisfied before time t. Then:

- The last readings[2] occur at latest at time
$$t + 2\tau_{\max} + T_{\max}.$$
To see this, pick a computing unit that is "late to awake": this unit just missed the earliest writing, which was made available at latest at $t + \tau_{\max}$. It can notice this writing at latest within one period $T_{\max}$ and then it decides to update its outputs, which takes at most $\tau_{\max}$ before being made available to other units.
- Executions can occur at earliest at time
$$\min(t + n_{1a}T_{\min}\,,\, t + \tau_{\min} + n_{1b}T_{\min}) \qquad (9)$$

[1] Recall that communicated variables are updated only when needed by the application, whence the mention of "possibly".

[2] Observe that only first readings are depicted on the net of Figure 5.

The first case corresponds to a "first to write" unit and the second one corresponds to a "first to awake" unit that could notice earliest possible the first writes.

Thus, Requirement 1 reduces to the following condition:

$$2\tau_{\max} + T_{\max} \quad < \quad \min(n_{1a}T_{\min}\,,\, \tau_{\min} + n_{1b}T_{\min})$$

which can be satisfied by taking

$$n_{1a} \quad = \quad \left\lceil \frac{2\tau_{\max} + T_{\max}}{T_{\min}} \right\rceil \qquad (10)$$

$$n_{1b} \quad = \quad \left\lceil \frac{2\tau_{\max} + T_{\max} - \tau_{\min}}{T_{\min}} \right\rceil \qquad (11)$$

b) Requirement 2:

- The next writings can start not earlier than

$$\min \left\{ \begin{array}{r} t + (n_{1a} + n_{2a})T_{\min} + \tau_{\min} \\ \tau_{\min} + t + (n_{1b} + n_{2b})T_{\min} + \tau_{\min} \end{array} \right.$$

  The first term corresponds to a "first to write" unit — notice the $\tau_{\min}$ which corresponds to the minimal communication delay. The second term corresponds to a "first to awake" unit.

- The latest execution cannot start later than:

$$\max \left\{ \begin{array}{r} t + 2\tau_{\max} + T_{\max} + n_{1b}T_{\max} \\ t + n_{1a}T_{\max} \end{array} \right. \qquad (12)$$

  The first term corresponds to a "last to awake" unit. The second term corresponds to a slow "first to write" unit.

Thus, Requirement 2 reduces to the following condition:

$$\begin{array}{l} \max \left\{ \begin{array}{r} 2\tau_{\max} + T_{\max} + n_{1b}T_{\max} \\ n_{1a}T_{\max} \end{array} \right. \\ < \\ \min \left\{ \begin{array}{r} \tau_{\min} + (n_{1a} + n_{2a})T_{\min} \\ 2\tau_{\min} + (n_{1b} + n_{2b})T_{\min} \end{array} \right. \end{array} \qquad (13)$$

which can be satisfied by taking appropriate lower bounds for n_{2a} and n_{2b}, using the values (10,11) for the pair (n_{1a}, n_{1b}), namely:

$$n_{2a} \quad = \quad \max \left\{ \begin{array}{l} \left\lceil \frac{\tau_{\max} - \tau_{\min} + (1 + n_{1b})T_{\max} - n_{1a}T_{\min}}{T_{\min}} \right\rceil \\ \left\lceil \frac{-\tau_{\min} + n_{1a}T_{\max} - n_{1a}T_{\min}}{T_{\min}} \right\rceil \end{array} \right. \qquad (14)$$

$$n_{2b} \quad = \quad \max \left\{ \begin{array}{l} \left\lceil \frac{\tau_{\max} - 2\tau_{\min} + (1 + n_{1b})T_{\max} - n_{1b}T_{\min}}{T_{\min}} \right\rceil \\ \left\lceil \frac{-2\tau_{\min} + n_{1a}T_{\max} - n_{1b}T_{\min}}{T_{\min}} \right\rceil \end{array} \right. \qquad (15)$$

This finishes the proof of Protocol 1. □

Slow-down caused by the protocol: The above analysis regarding parameters $n_{1a}, n_{1b}, n_{2a}, n_{2b}$ of protocol 1 shows that the maximum number of computing unit clock cycles required in the distributed case to execute a single round of the application is:

$$r = \max\{n_{1a} + n_{2a}, n_{1b} + n_{2b}\}$$

It is noticeable that this slow-down is independent from the number of distributed computing units. Though there can be several motivations for distributing programs (fault tolerance, location of sensors and actuators, etc...) among which performance is not always the major concern, yet we can remark here that distribution in general reduces the task load of each unit allowing them to run faster. Thus this required slow-down can possibly be compensated by faster clock cycles.

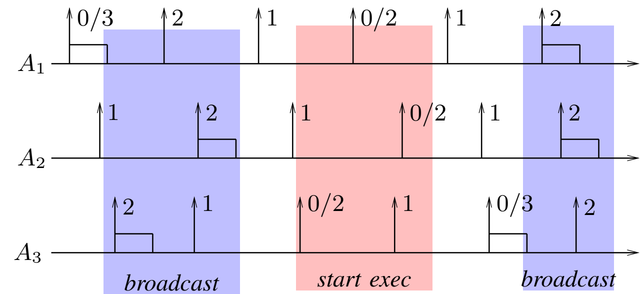

Fig. 6. Illustrating protocol 1. Referring to figure 5, transitions labeled with $0/3$ correspond to firings of **1a**; transitions labeled with $0/2$ correspond to firings of **2a** or **2b**. Small rectangles indicate the delay in broadcasting updates.

Figure 6 illustrates the protocol with: $T_{max} = 1.5, T_{min} = 1, \tau_{max} = \tau_{min} = 0.5$ which yields $n_{1a} = 3$ and $n_{1b} = n_{2a} = n_{2b} = 2$. We can see clearly how synchrony is preserved thanks to a separation in time of alternating broadcast and execution periods. In this case we get $r = 5$ for the slow-down.

C. An illustration example

This example was suggested to one of the authors by Moez Yeddes and René David during the CRISYS project [15]. Consider the following synchronous boolean dynamical system:

$$\left\{ \begin{array}{lclcl} x_k & = & \neg y_{k-1}.u_{k-1} \vee x_{k-1} & , & x_0 = \mathrm{F} \\ y_k & = & \neg x_{k-1}.u_{k-1} \vee y_{k-1} & , & y_0 = \mathrm{F} \end{array} \right. \qquad (16)$$

Only one behaviour is possible for system (16): let $k_* \leq \infty$ be the first time when $u_k = \mathrm{T}$. Then we have $(x_k, y_k) = (\mathrm{F}, \mathrm{F})$ for $k < k_*$ and $(x_k, y_k) = (\mathrm{T}, \mathrm{T})$ for $k \geq k_*$.

We wish to deploy this system on an architecture consisting of three sites S_1, S_2, and S_3, communicating via CbS. Input u is read on site S_1; x is computed on site S_2; and y is computed on site S_3. The needed communications are CbS.

The following scenario can occur if no precaution is taken:

1) At some time t_*, u changes, from F to T. This change is broadcast at the subsequent tick of κ^1, which occurs at real-time $t_1 > t_*$.
2) Due to lack of clock synchronization, this change is first seen by site S_2, say at time $t_2 > t_1$ of its clock κ^2. Site S_2 reacts by emitting immediately, on its output port, the value T for x.
3) Due to the lack of clock synchronization, site S_3 sees, at the same clock tick of its clock κ^3, both the above change in u, and the new change in x. This occurs at time t_3.

4) Following this, the system is stuck at the values (T, F) for the pair (x, y).

This scenario is illustrated on Figure 7. The graph $\mathcal{G}(S)$ is

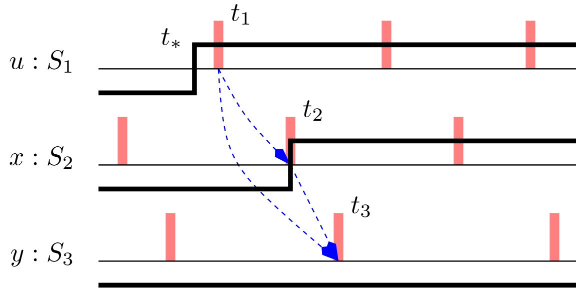

Fig. 7. A wrong scenario for example (16). The values F, T are represented as $-1, +1$. Clock ticks are depicted by pink thick vertical bars. Blues dashed arrows depict how changes in variables propagate in this scenario.

(with obvious abuse of notation):

$$u \to (x, y), (x, y) \stackrel{\text{UD}}{\to} (x, y)$$

Observe that assumption 3 holds. Let the three clocks $\kappa^i, i = 1, 2, 3$, satisfy condition (7), with $T_{\min} = 1$ and $T_{\max} = 2$, as shown on Figure 8. To simplify, we ignore the communication delays: $\tau_{\max} = 0$. Applying formulas (10,11) and (14,15) yields in this case $n_{1a} = n_{1b} = 2$, $n_{2a} = n_{2b} = 2$, which yields an upsampling rate of 4. The result is shown on Figure 8. Note that, at the end of the reaction (marked

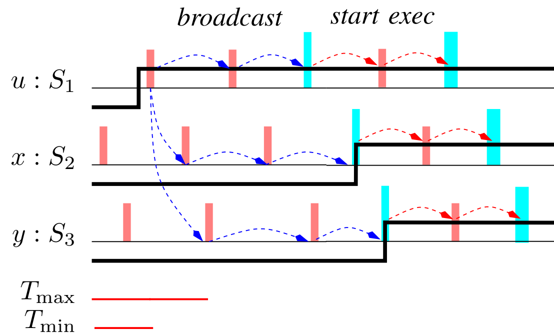

Fig. 8. This figure continues Figure 7, with reduced scale. We see that changes in inputs from site S_1 are taken into account with a delay of 2 clock ticks. Memory updating are postponed by a further amount of 2 clock ticks (depicted with cyan very thick bars).

by the very thick cyan bars), the values for all variables and memories are correct. This ensures preservation of semantics.

D. Relaxing assumption 3: an improved protocol

An improved version of the simple protocol allows handling the case when assumption 3 does not hold, see the full paper [5].

V. Conclusion

We have proposed a simple way of deploying a synchronous spefication over an LTT architecture. Our approach is simple and robust in that it does not require any additional messaging — excess messaging is an additional source of problems when fault tolerance is considered. Instead, our approach entirely relies on a combination of loose synchronization requirements between clocks (limited relative drift between different clocks), upsampling mechanisms, and local counters.

References

[1] A. Benveniste and P. Caspi and M. di Natale and C. Pinello and A. Sangiovanni-Vincentelli and S. Tripakis, "Loosely Time-Triggered Architectures based on Communication-by-Sampling," in *EMSOFT*, 2007, pp. 231–239.

[2] A. Benveniste, P. Caspi, P. L. Guernic, H. Marchand, J.-P. Talpin, and S. Tripakis, "A protocol for loosely time-triggered architectures." in *EMSOFT*, 2002, pp. 252–265.

[3] L. P. Carloni, "The role of back-pressure in implementing latency-insensitive systems," *Electr. Notes Theor. Comput. Sci.*, vol. 146, no. 2, pp. 61–80, 2006.

[4] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–1076, 2001.

[5] P. Caspi and A. Benveniste, "Extended version of this paper," http://www.irisa.fr/distribcom/benveniste/pub/LTTA_CDC08.html, July 2008.

[6] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert, "From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications," in *Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*. ACM, 2003.

[7] J. Cortadella and M. Kishinevsky, "Synchronous elastic circuits with early evaluation and token counterflow," in *DAC*, 2007, pp. 416–419.

[8] H. Kopetz, *Real-Time Systems*. Kluwer Academic Publishers, 1997.

[9] C. Kossentini and P. Caspi, "Approximation, sampling and voting in hybrid computing systems." in *HSCC*, 2006, pp. 363–376.

[10] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[11] N. A. Lynch, *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

[12] J. Romberg and A. Bauer, "Loose synchronization of event-triggered networks for distribution of synchronous programs." in *EMSOFT*, 2004, pp. 193–202.

[13] P. Traverse, I. Lacaze, and J. Souyris, "Airbus fly-by-wire: A total approach to dependability," in *IFIP World Congress, Toulouse*. IFIP, 2004.

[14] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincentelli, P. Caspi, and M. Di Natale, "Implementing Synchronous Models on Loosely Time Triggered Architectures," *IEEE Transactions on Computers*, 2008, to appear.

[15] M. Yeddes, H. Alla, and R. David, "The partition method for the order-insensitivity in a synchronous distributed systems," in *IEEE, ISCC'2000, Antibes (France)*, July 2000.