IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION INTERNE
N° 1289 (revised)

DISTRIBUTING SYNCHRONOUS PROGRAMS ON A *LOOSELY SYNCHRONOUS,* DISTRIBUTED ARCHITECTURE

ALBERT BENVENISTE ,
PAUL CASPI , STAVROS TRIPAKIS

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA
Institut de Recherche en Informatique et Systèmes Aléatoires
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Distributing synchronous programs on a *loosely synchronous,* distributed architecture *

Albert Benveniste ** ,

Paul Caspi , Stavros Tripakis***

Thème 1 — Réseaux et systèmes
Projets Ep-Atr

Publication interne n ° 1289 (revised) — October 5, 2001 — 17 pages

Abstract: In this short note we discuss the distribution of synchronous programs on distributed, asynchronous architecture of the type preferred for embedded real-time control applications. In such architectures, no global clock is broadcast. Instead, the bus, as well as its users are triggered by local periodic clocks. To improve fault-tolerance and robustness, these clocks are not synchronized, neither physically nor logically. Therefore such an architecture has some (restricted) degree of asynchrony, we call it *loosely synchronous.* A typical example of such architecture is the OSEK protocol, used for embedded electronics in automotive industry.

In this note we show how synchronous programs can be safely distributed on loosely synchronous architectures. The method described here may not be novel to some communities, but it aims at ensuring awareness of the community of engineers in the area of real-time, embedded control systems.

Key-words: synchronous languages, desynchronization, distributed code generation, loosely synchronous.

(Résumé : tsvp)

* This work is or has been supported in part by the following projects: Esprit LTR-SYRF (Esprit EP 22703), and Esprit R&D CRISYS EP 25514.
** Irisa/Inria, Campus de Beaulieu, 35042 Rennes cedex, France, Email: firstname.lastname@irisa.fr, Web: http://www.irisa.fr/sigma2/benveniste/home.html
*** Verimag, Centre Equation, 2, rue de Vignate, F-38610 Gieres, Paul.Caspi@imag.fr, http://www.imag.fr/VERIMAG/PEOPLE/Paul.Caspi, Stavros.Tripakis@imag.fr

Centre National de la Recherche Scientifique
(UPRESSA 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Distribution de programmes synchrones sur une architecture *mollement synchrone*

Résumé : Cette note présente une approche très simple pour la répartition de programmes synchrones sur des architectures typiques des bus de terrain en automatique temps-réel (un exemple connu est OSEK dans le domaine de l'électronique embarquée en automobile). De telles architectures peuvent être vues comme un bus doté d'une horloge périodique. Les usagers du bus procèdent également à des accès périodiques. Le point important est que 1/ toutes ces horloges sont purement locales et non synchronisées, et 2/ rien ne doit être bloquant. Il s'agit donc d'architectures réparties qui ne satisfont pas au modèle des langages synchrones, car elle n'assure pas d'horloge logique globale. Nous qualifions ces architectures de *mollement synchrones.*

Dans cette note, nous proposons donc une approche très simple pour répartir, de manière correcte par construction, des programmes synchrones sur une architecture mollement synchrone.

Mots clés : langages synchrones, désynchronisation, génération de code distribuée, synchrone mou.

Contents

1 Introduction

Distributing embedded software for industrial control systems can be performed by resourcing to the following different philosophies:

- Develop system architecture[1] so that it is inherently robust to the uncertainty arising from the distributed nature of its computer architecture. This is what we refer to as the "traditional" approach of engineering – this is by no means a pejorative statement, as this approach has vastly proved its merits.
- Make sure that the actual system deployment conforms to its specified model, i.e., moving from the simulation model to the actual deployment is performed in a "provably correct" way.

Let us first further elaborate on these two approaches.

1.1 The "traditional" approach of engineering

Industrial control systems are often physically distributed systems composed of interconnected subsystems, e.g., intelligent sensors and actuators, functions embedded on computers (typically PC's, or alternatively PLC's — Programmable Logic Controllers — in process control industries, or embedded DSP's or hardware in automobiles or communication handsets).

Thus it is reasonable to develop separate programs for each separate computer and its functions. For control engineering type of functions, tools and techniques of control engineering are typically used. In particular software is mapped directly from block-diagram speficications or sequential function charts. In any case, the design of these sub-systems generally follows the so-called *synchronous* programming methodology (even though synchronous programming tools are not necessarily used). This means that programs obey a simple and strict protocol: programs evolve according to a sequence of reactions, and in each reaction inputs are scanned, computations are performed by reading inputs and memories, and then outputs are emitted and memories are refreshed.

Of course, the problem of assembling the sub-systems into the whole system is faced with the problem of distributed architecture, communications, and fault tolerance. The supporting distributed architecture does not comply with the ideal model of synchronous programming.

The traditional way of coping with this difficulty consists in designing the system so that it is *robust* with respect to the asynchronous nature of the distributed architecture. Also a certain degree of fault tolerance is expected from the design. This involves the system and its functions at an early stage of the design, and the functions themselves need to be specified with this objective in mind.

Clearly this is a difficult art, and very little theory is at hand to support such an approach. In the framework of the Esprit 25514 CRISYS project, Paul Caspi and coworkers have

[1] Here "system" refers to the plant for control, i.e., the car, the aircraft, the power plant, etc., plus its associated computer control system.

proposed a mathematical framework and associated methodology to address this problem in some restricted form [5].

1.2 Desynchronizing synchronous system designs

If one insists on layering the design process by keeping a cleanest possible separation of specifications, software development, and architecture generation, then an alternative approach can be considered. Design the specifications independently from the target architecture[2]. Then, prototype and validate the software independently from the target architecture. This approach is of particular interest for medium and higher levels of the control of embedded systems, it offers an interesting alternative to the previous approach which is barely implementable for such levels.

Then use advanced techniques to distribute the software and generate the architecture in such a way that the behaviour of the actual, distributed, system can be safely predicted from that of the prototype software.

In the framework of automotive industry, the work of Kopetz and coworkers [1, 2] on the TTP (Time Trigerred Protocol) is an attempt to offer a communication medium and associated distributed architecture, which in fact comply with our model of strict synchrony. By providing a fault-tolerant bus, and broadcast on it a fault-tolerant clock, global reactions can be propagated through the bus. Of course, this takes physical time, therefore no instantaneous dialog between two communicating components can be implemented. But, as soon as dialogs involve *logical* delays (in the form of registers), the paradigm of strict synchrony applies. Therefore the mapping from specifications to architecture becomes trivial. But there is a price to pay : guaranteing the synchrony hypothesis amounts to implement systematic, time-triggered, protocols which impairs the performance of the communication medium, and, therefore, of the system.

An alternative approach has been proposed in the framework of the Esprit 20897 SACRES project, based on the desynchronization of synchronous programs [3]. It has been shown that synchronous programs can safely be distributed on asynchronous architectures, provided that fault tolerance is guaranteed by the communication medium. This means that *the design is robust to delay uncertainties* (but not to message loss). This is achieved by automatically generating properly tailored protocols and signalling for correctness of the distributed design. As such protocols can be blocking, this approach may not be suitable for all uses.

1.3 Our objective

Our objective is to show that one can combine the two approaches of subsections 1.1 and 1.2, and take advantage of them in the following way :

(a) Assume a communication medium of *loosely synchronous* type : quasi periodic communication of packets is broadcast through a bus. This bus does not guarantee message

[2] Of course this is too strong a claim but the idea is to avoid taking into account details of the architecture

losslessness. It does not either broadcast a strict synchronisation clock. But it guarantees that packets received will never be outdated beyond a certain limit.

(b) Assume low level control functions are designed according to the approach of subsection 1.1.

(c) We wish to develop medium or higher level functions according to the method of subsection 1.2.

Item **(c)** is somewhat in contradiction with items **(a)** and **(b)**. *We will nevertheless show that we can enhance the desynchronisation method of subsection 1.2 so that deployment on architecture of type* **(a)** *is correct by construction. This allows one to combine the above three approaches, together with their respective advantages, in a smooth way.*

In fact, the ideas we use in this work also belong to the folklore of "distributed computer science" : Lamport revealed, twenty years ago, how resynchronisation could be performed in a robust way, for a distributed architecture [6, 7]. In some sense, what we propose here is a way to strictly tailor such mechanisms to the application at hand, therefore keeping the resulting overhead minimal.

2 Informal description of the approach

In this section we informally draw our approach. Formal presentation is deferred to the next section.

2.1 The distributed architecture

The considered distributed architecture consists of media behaving (roughly) as follows :

- Access to the bus occurs (quasi)periodically, in a non-blocking way. Writings and readings are performed independently at each extremity of the channel in synchrony with each considered local clock.
- Then the bus behaves like a shared memory, i.e., values are sustained by the channel.

Of course, nondeterminism results from the following uncertainties :

- Periodicities do not match exactly, i.e., the local clocks are not strongly synchronized ; residual phase shift or jitter thus follows.
- Transmission delays impair the ideal statement that "the bus behaves like a shared memory".

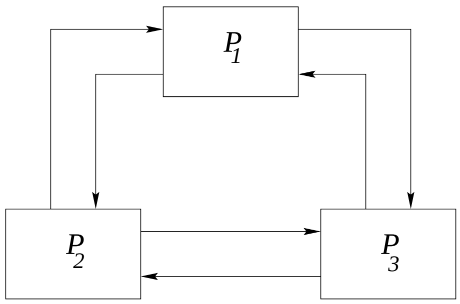

Figure 1: *A synchronous design.*

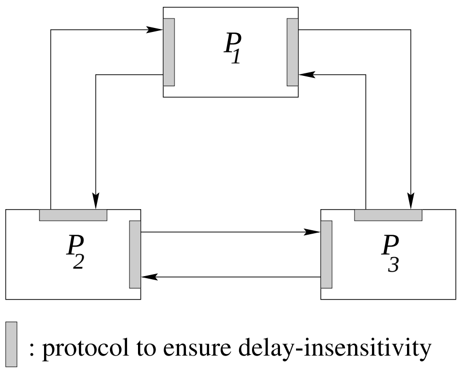

Figure 2: *Securing delay-insensitivity.*

2.2 Adapting code distribution to the architecture

Figure 1 shows a synchronous design, with 3 components. In figure 2, adequate synchronous programs have been added, they are depicted using the light grey rectangles. Based on the theory developed in [3] [4], these "protocols" aim at securing delay-insensitivity of the communications. This means that the semantics of each of the components in the considered system is not modified if the perfectly synchronous communication channels are substituted with asynchronous channels with unknown and unbounded delays. Of course the global timing of the program is lost. This is achieved by ensuring so-called *isochrony* of the synchronous commmunication network, see [3]. At this stage delay-insensitivity is achieved, assuming however that the communication medium is reliable: each flow is individually preserved by the communication (but their global synchronisation is not).

Now we show in figure 3 how to adapt this design for the case of a loosely synchronous architecture such as described in subsection 2.1, i.e., with *non-reliable, but loosely synchronised,* channels. Basically, the idea is to re-encode events using state variable changes, these additional protocols are figured by the dark rectangles.

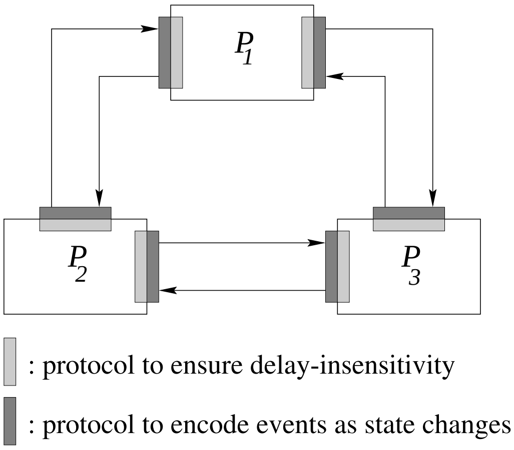

Figure 3: *Encoding events as state changes, and performing upsampling.*

In figure 4, we have redrawn the figure 3 in a different way, for the purpose of the analysis. We combine the dark rectangles together with the "physical" channels, to obtain

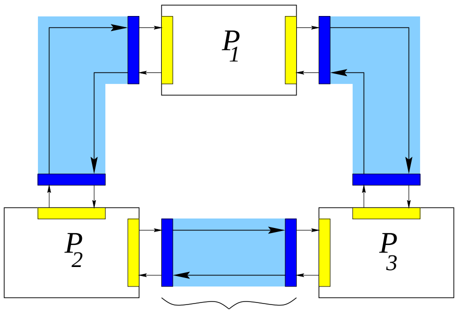

Figure 4: *Redrawing the figure 3 differently, by introducing so-callled "virtual" channels.* Then, prove that the virtual channel conforms the requirements for the design of figure 2 to be valid.

so-called "virtual" channels. Our technique for analysis consists in checking whether the virtual channel satisfies our requirements for delay insensitivity.

3 The protocol

We apply the following protocol for transmitting a given variable y, we first describe it informally:

1. *The emitter can write in the bus periodically, based on its own, non synchronized, local clock.* Therefore writings are activated by a logical clock which we denote by $h_{\mathbf{w}}$. For y a variable to be sent by the emitter, its clock h_y is therefore a subclock of the "writer clock" $h_{\mathbf{w}}$. For y such a variable, attach to y a boolean flag b_y which has same clock as y and switches each time it is delivered, therefore delivering the sequence $\mathrm{T}, \mathrm{F}, \mathrm{T}, \ldots$ simultaneously with each successive y. The pair $\widehat{y} = (b_y, y)$ is the desired coding for the variable y and is submitted to the channel.

2. *The bus scans its environment and is available for reading periodically, based on its own, non synchronized, local clock.* Therefore the bus is activated at its own logical clock which we denote by $h_{\mathbf{c}}$. We assume that $h_{\mathbf{c}}$ is faster than the other periodic clocks, this requirement is formalized later on. The channel sustains a state variable for each of its inputs, and updates it when corresponding input writing occurs. Call Y the state variable associated to an input variable $\widehat{y}$ for the bus. Hence Y sustains the two components (boolean flag, carried value), of the message $\widehat{y} = (b_y, y)$. We call Y a bus variable.

3. *The receiver scans the bus periodically, based on its own, non synchronized, local clock.* Therefore readings are activated by a logical clock which we denote by $h_{\mathbf{r}}$. For Y a bus variable, the receiver decomposes Y into its boolean flag b_{Y} and carried value ν_{Y}, and this latter value is validated if and only if b_{Y} just switched otherwise it is deleted. This results in a received flow we call $y^{\mathbf{rec}}$.

See figure 5 for an illustration of this protocol. The question is: denoting by $(y_n)_{n>0}$ the

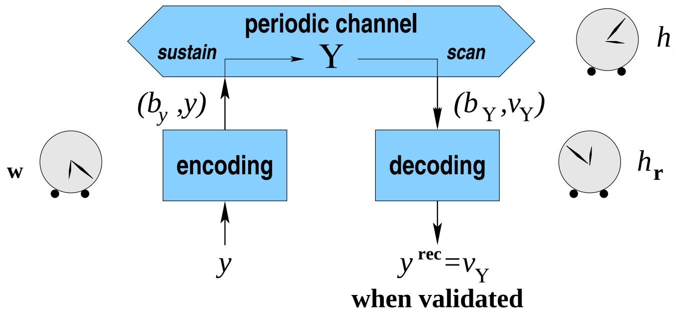

Figure 5: *The protocol.*

successive *present* occurrences of flow y, and similarly for $y^{\mathbf{rec}}$,

$$\text{does} \quad \forall n > 0 : y_n^{\mathbf{rec}} = y_n \quad \text{hold?} \tag{1}$$

Note that $y_n^{\mathbf{rec}}$ is not synchronous with y_n! Also, question (1) involves each individual message, separately. A positive answer to (1) would allow us to safely apply our previous techniques for correct desynchronisation. In the section to follow we prove the following result.

Theorem 1 (sampling theorem) *Referring to the protocol of figure 5, let the writing/channel/reading period be $w/c/r$. If*

$$w \geq c \; , \; \left\lfloor \frac{w}{c} \right\rfloor c \; \geq \; r \; , \tag{2}$$

then property (1) holds true.

In theorem 1, notation $\lfloor x \rfloor$ denotes the largest integer n such that $n \leq x$.

COMMENTS :

1. Assuming that the bus protocol is enhanced with a simple timeout mechanism (telling that the emitter failed sending its message in due time) and corresponding exception handling mechanisms, the whole protocol becomes non blocking and allows fault-tolerance to be implemented. Therefore our approach meets the desired requirements for embedded distributed software for real-time control.

2. In addition, this protocol is easily mixed with the classical protocol of sampled real-time control, in which sampled data from sensors and sampled controls for actuators are communicated periodically via the time-triggered bus : just ignore the validate-and-sustain mechanism and perform careless periodic writings and readings. Of course, by doing this, stuttering can occur, or alternatively data can be lost. But robustness is usually inherently provided by the proper engineering of the control laws (cf. the so-called "phase-margin" requirement in control).

We now provide the necessary formalisation.

4 Formal study

In this section we use the framework of STS introduced in [3, 4] and the notations therein (in fact we could have as well used the SIGNAL language for this purpose, as its expressive power is equivalent). This framework will help us to formalize our approach.

Briefly said, we shall consider symbolic transition systems in which domains of variables are extended with the special symbol $\perp$ to denote absence. Symbols x, b, etc., denote variables, and symbols b, b_x, etc., will by convention refer to variables of boolean type. An STS has a set V of variables. A state for this STS is a valuation of this set of variables, including the distinguished value $\perp$ to mean that a given variable is absent in the considered state. Runs are sequences of states, each state corresponding to a reaction or an "instant" of the STS.

For x a variable, x^- denotes its associated previous variable, which gives the value of x at the previous instant of a run. For x a variable, h_x denotes its *clock* with domain $\{\perp, \mathrm{T}\}$, such that $h_x = \perp$ iff $x = \perp$. Finally, as variable x can be absent at a given instant, it is of interest to sustain previous *present* values at the instants at which x is absent, this is carried by the special variable ξ_x (we use a greek symbol for these specific auxiliary variables, since they are never absent, in contrast to other variables). Hence, for each variable x, ξ_x can be seen as a "piecewise constant, right continuous interpolation" of the history of x (0 is the initial instant, and x_0 an initial condition for x):

$$\begin{array}{rcccccccc} \text{instant}: & 0 & 1 & 2 & 3 & 4 & 5 & 6 & \dots \\ x: & x_0 & \perp & \perp & x_1 & \perp & x_2 & \perp & \dots \\ h_x: & \perp & \perp & \perp & \mathrm{T} & \perp & \mathrm{T} & \perp & \dots \\ \xi_x: & x_0 & x_0 & x_0 & x_1 & x_1 & x_2 & x_2 & \dots \end{array}$$

In the sequel, we shall specify STS as follows. Variables will be declared, possibly with the additional indication that they are inputs or outputs (this is not mandatory, however). Transition relations will be specified as a conjunction of statements involving variables, and, whenever necessary, we shall also specify initial conditions. For each declared variable x we shall freely use in the statements the auxiliary variables x^-, h_x, ξ_x.

NOTA. While the usual interpretation of symbol $\perp$ is *absent,* other interpretations are perfectly valid as well. Assuming some underlying discrete time available, and a boolean signal b indexed by this discrete time, we can convey that b defines the set of instants at which b is true, or alternatively has a raising edge (i.e., has a transition from false to true). We call h_b this clock, and we convey that a signal x with clock h_b has the value $\perp$ outside the instants of h_b. More generally, symbol $\perp$ can be used to denote *don't cares.* In fact, the important point is that clocks and relations on clocks define a *type system* for signals.

4.1 Encoding

This is specified via the following STS:

$$\begin{array}{l} \mathbf{encoding}\{\ \mathbf{in}: y\ ;\ \mathbf{out}: \widehat{y}\ ;\ \mathbf{local}: h_{\mathbf{w}}\} \\ =_{\text{def}} \\ \left[\begin{array}{ll} & \mathbf{if}\ \ h_y = \mathrm{T}\ \ \mathbf{then}\ \ b_y = \mathbf{not}\ \xi_{b_y}^-\ \ \mathbf{else}\ \ b_y = \perp \\ \wedge & \widehat{y} = (b_y, y) \\ \wedge & h_y \leq h_{\mathbf{w}} \end{array}\right] \end{array} \tag{3}$$

and the pair $\widehat{y} = (b_y, y)$ is the desired coding for the variable y. In (3) $\xi_{b_y}^-$ is the previous present value of b_y, hence b_y behaves as a switch: $\dots, \mathrm{T}, \mathrm{F}, \mathrm{T}, \mathrm{F}, \dots$. This formalizes step 1 of the protocol of section 3.

4.2 Decoding

This is specified via the following STS, where $\mathrm{Y} = (b_\mathrm{Y}, \nu_\mathrm{Y})$ decomposes into a boolean flag, and a valued flow:

$$
\begin{array}{rl}
& \mathbf{decoding}\{\ \mathbf{in} : \mathrm{Y}\ ;\ \mathbf{out} : y^{\mathbf{rec}}\ ;\ \mathbf{local} : h_\mathbf{r}, b_r\} \\
=_{\mathrm{def}} & \\
& \left[\begin{array}{ll} & \mathrm{Y} = (b_\mathrm{Y}, \nu_\mathrm{Y}) \\ \wedge & \mathbf{if}\ \ h_\mathbf{r} = \mathrm{T}\ \ \mathbf{then}\ \ b_r = \xi_{b_\mathrm{Y}}\ \ \mathbf{else}\ \ b_r = \perp \\ \wedge & \mathbf{if}\ \ b_r = \mathbf{not}\ \xi^-_{b_r}\ \ \mathbf{then}\ \ y^{\mathbf{rec}} = \xi_{\nu_\mathrm{Y}}\ \mathbf{else}\ \ y^{\mathbf{rec}} = \perp \end{array}\right]
\end{array} \tag{4}
$$

meaning that output $y^{\mathbf{rec}} = \nu_\mathrm{Y}$ is delivered if and only if it is validated by a change of b_Y. Note that the activation clock of this STS is $h_\mathbf{r}$ and that $h_\mathbf{r}$ is *not* bound to any other particular clock: the receiver works asynchronously. Reading from the bus is performed by reading the *sustained* values ξ_{b_Y} and ξ_{ν_Y}. This formalizes step 3 of the protocol.

4.3 Channels

Referring to the discussion in the introduction, writes and reads occur in a quasi periodic manner. If we accept a discretized view of continuous real-time, channels can be modelled as follows.

Each channel $\mathbf{c}$ has an activation clock $h_\mathbf{c}$, and a finite set of variables it can transmit, these are generically written x. Each variable x can be written asynchronously without blocking, this is modelled by $x_\mathbf{w}$, and read asynchronously without blocking, this is modelled by $x_\mathbf{r}$. Finally, we set constraints on the upsampling which occurs between the writes/reads and the activation clock of the channel.

In the following STS, $h_\mathbf{c}$ denotes the activation clock of the channel. Then, for each variable x of the channel, we consider

$$
\begin{array}{rcl}
x_\mathbf{w} & : & \text{a user writing a value in } x \\
x_\mathbf{r} & : & \text{a user reading a value from } x
\end{array}
$$

The channel is then specified as follows:

$$
\begin{array}{rl}
& \mathbf{channel}_x\{\ \mathbf{in} : x_\mathbf{w}\ ;\ \mathbf{out} : x_\mathbf{r}\ ;\ \mathbf{local} : h_\mathbf{c}, x\} \\
=_{\mathrm{def}} & \\
& \left[\begin{array}{ll} & \mathbf{if}\ \ h_\mathbf{c}\ \mathbf{then}\ \ x = \xi_{x_\mathbf{w}}\ \mathbf{else}\ \ x = \perp \\ \wedge & \mathbf{if}\ \ h_{x_\mathbf{r}}\ \mathbf{then}\ \ x_\mathbf{r} = \xi_x \end{array}\right]
\end{array} \tag{5}
$$

This formalizes step 2 of the protocol, by taking into account conditions (2) of theorem 1.

4.4 The virtual channel

The virtual channel as drawn in figure 5 is therefore modelled by the STS :

$$
\begin{array}{rl}
& \mathbf{virtual\_channel}\{\,\mathbf{in}: y\ ;\ \mathbf{out}: y^{\mathbf{rec}}\ ;\ \mathbf{local}: h_{\mathbf{c}}, h_{\mathbf{w}}, h_{\mathbf{r}}\} \\
= & \left[\begin{array}{ll}
 & \mathbf{encoding}\{\,\mathbf{in}: y\ ;\ \mathbf{out}: \widehat{y} = (b_y, y)\ ;\ \mathbf{local}: h_{\mathbf{w}}\} \\
\| & \\
 & \mathbf{channel}_b\{\,\mathbf{in}: b_y\ ;\ \mathbf{out}: b_{\mathrm{Y}}\ ;\ \mathbf{local}: h_{\mathbf{c}}, b\} \\
\| & \\
 & \mathbf{channel}_\nu\{\,\mathbf{in}: y\ ;\ \mathbf{out}: \nu_{\mathrm{Y}}\ ;\ \mathbf{local}: h_{\mathbf{c}}, \nu\} \\
\| & \\
 & \mathbf{decoding}\{\,\mathbf{in}: \mathrm{Y} = (b_{\mathrm{Y}}, \nu_{\mathrm{Y}})\ ;\ \mathbf{out}: y^{\mathbf{rec}}\ ;\ \mathbf{local}: h_{\mathbf{r}}\}
\end{array}\right]
\end{array}
\tag{6}
$$

Note the three, non synchronized, activation clocks $h_{\mathbf{c}}, h_{\mathbf{w}}, h_{\mathbf{r}}$.

4.5 The sampling theorem

For $x = (x_k)_{k\geq 0}$ a sequence with values in a domain extended with the special symbol $\perp$, we denote by

$$x^{\top} \quad =_{\mathrm{def}} \quad \{x_{k_n} \ :\ x_{k_n} \neq \perp\}$$

the sequence of the occurrences of x which differ from $\perp$. By applying this to each history of a flow, we extend the map $x \longmapsto x^{\top}$ to flows. We restate theorem 1 :

Theorem 2 (sampling theorem) *Let the writing/channel/reading period be $w/c/r$. Then The* STS $\mathbf{virtual\_channel}\{\,\mathbf{in}: y\ ;\ \mathbf{out}: y^{\mathbf{rec}}\ ;\ \mathbf{local}: \ldots\}$ *satisfies*

$$y^{\mathbf{rec}\,\top} \quad = \quad y^{\top} \tag{7}$$

iff

$$w \geq c\ ,\ \left\lfloor \frac{w}{c} \right\rfloor c \ \geq\ r\,. \tag{8}$$

holds.

Condition (7) means that the channel satisfies the requirements (1) for correct desynchronisation. The bottom line is that our whole approach for code distribution on the loosely synchronous architecture is proved to be valid by construction.

Note that $w/2 < \lfloor w/c \rfloor\ c \leq w$, and $\lfloor w/c \rfloor\ c \sim w$ for w/c large. Hence, for a fast channel, the condition of theorem 2 reduces to

$$w \ >\ r\,. \tag{9}$$

We illustrate the protocol on the figure 6.

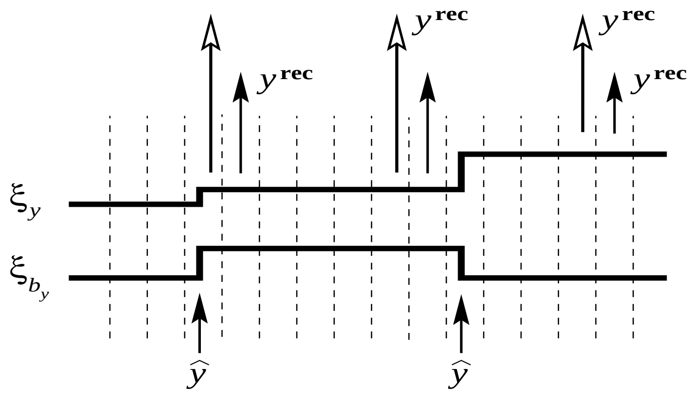

Figure 6: *The sampling theorem.* We illustrate it on two cases (black and white arrows), depending on the relative position of the clocks $h_{\mathbf{c}}, h_{\mathbf{w}}, h_{\mathbf{r}}$. The two thick lines depict the sustained histories (ξ_{b_y}, ξ_y) corresponding to the two components (b_y, y) of $\widehat{y}$. The dashed vertical lines depict the ticks of the channel clock $h_{\mathbf{c}}$. The vertical arrows sitting at the bottom depict the ticks of the writer clock[4]. The vertical arrows sitting at the top depict the ticks of the reader clock. Note the role of the boolean flag for validation: even if the value of y was unchanged at two successive emissions it would hev been received, thanks to the boolean validation flag. The role of this flag is also enlightened by the two cases for the relative position of the clocks (black and white arrows). The case of the black arrows is the "normal" one. Let us focus on the more tricky case of the white arrows. In this case, when the first instant of clock $h_{\mathbf{r}}$ (first white arrow), then the clock $h_{\mathbf{c}}$ has not yet seen the change in ξ_{b_y}, therefore the corresponding value ξ_y is not validated by the reader at this point. But a change in ξ_{b_y} is detected at the second occurrence of $h_{\mathbf{r}}$, and, then, the corresponding value ξ_y is validated. As seen from the diagram, changes in the boolean flows are detected at the receiver with a nondeterministic, but bounded delay. On the other hand, perfect synchrony is lost: reading occurs with a nondeterministic, but bounded, delay according to *physical* real time.

Proof: take $w = 1$. In the following, let $n, m, p = 0, 1, 2, \ldots$ denote integers. Let the writing/reading/channel sampling instants be

$$\begin{aligned} t_w(n) &= n \\ t_{c,\psi}(p) &= pc + \psi c,\ 0 \le \psi < 1 \\ t_{r,\varphi}(m) &= mr + \varphi r,\ 0 \le \varphi < 1 \end{aligned}$$

where the writing/reading/channel period is $1/r/c$. Set

$$\hat{t}_{c,\psi}(n) = \min\{t_{c,\psi}(p) : t_{c,\psi}(p) \ge n\},$$

$\hat{t}_{c,\psi}(n)$ is the first channel instant where the channel sees the nth writing. We have

$$\hat{t}_{c,\psi}(n) = \left(\left\lfloor \frac{n}{c} - \psi \right\rfloor + 1 \right) c + \psi c$$

where $\lfloor . \rfloor$ is the integer part. Similarly, set

$$\hat{t}_{r,\varphi}(n) \quad = \quad \min\{t_{r,\varphi}(m) : t_{r,\varphi}(m) \geq \hat{t}_{c,\psi}(n)\}\,,$$

$\hat{t}_{r,\varphi}(n)$ is the first instant where the reader sees the nth writing. To be sure that we do not miss any writing, we need (and it is enough) that the map

$$n \;\longmapsto\; \hat{t}_{r,\varphi}(n)$$

is *strictly* increasing (then it is invertible). We have

$$\delta\hat{t}_c \;=_{\mathrm{def}}\; \min_{n\geq 0\,,\,0\leq\psi<1} \left(\hat{t}_{c,\psi}(n+1) - \hat{t}_{c,\psi}(n)\right) \;=\; \left\lfloor \frac{1}{c} \right\rfloor c$$

Hence, $n \;\longmapsto\; \hat{t}_{c,\psi}(n)$ is strictly increasing iff

$$\left\lfloor \frac{1}{c} \right\rfloor c > 0 \text{ , i.e. } 1 \geq c\,. \tag{10}$$

Now, in order to guarantee that $n \;\longmapsto\; \hat{t}_{r,\varphi}(n)$ is strictly increasing, our last duty is *not to miss a reading during any period of length* $\delta\hat{t}_c$*,* i.e.:

$$\left\lfloor \frac{1}{c} \right\rfloor c \;\;\geq\;\; r\,. \tag{11}$$

Combining (10) and (11) yields the theorem. ◇

We can generalize theorem 2 to time varying periods. Without loss of generality we can assume that w remains fixed.

Theorem 3 *Assume now that writing/channel/reading are related via the following equations:*

$$\begin{array}{lclcllcl}
t_w(n) & = & wn & & & & & \\
t_{c,\psi}(p) & = & pc(1+\delta^c(p)) & + & \psi c & , \; 0\leq\psi<1 & , & |\delta^c(p)| \leq \delta^c < 1 \\
t_{r,\varphi}(m) & = & mr(1+\delta^r(m)) & + & \varphi r & , \; 0\leq\varphi<1 & , & |\delta^r(m)| \leq \delta^r < 1
\end{array} \tag{12}$$

Then a sufficient condition for the protocol formalized in section 3 to be valid is:

$$w \geq c(1+\delta^c) \quad , \quad F(c,\delta^c) \geq r(1+\delta^r)\,, \tag{13}$$

where

$$F(c,\delta^c) \;=_{\mathrm{def}}\; \min_{\delta\,:\,|\delta|\leq\delta^c} \left\lfloor \frac{1}{c(1+\delta)} \right\rfloor c(1+\delta) \;=\; \left\lfloor \frac{1}{c(1-\delta^c)} \right\rfloor c(1-\delta^c)\,.$$

The proof of this theorem is a straightforward adaptation of that of theorem 2, and we leave it to the reader as an exercise. Of course, the interest of theorem 3 is that it guarantees a good degree of robustness of the protocol with respect to imperfect sampling clocks, since the relative *periods* of the clocks can vary with respect to each other, up to some maximal bound.

Comment. The request that readings are more frequent than writings is not a problem when using the loosely synchronous bus for circular communications. For example, assume two computers C_1, C_2 which communicate in both directions through the bus. Some signals are written by C_1 and read by C_2, and vice-versa. Consider the case of signals written by C_1 and read by C_2. The protocol consists in the following chain :

$$y \mapsto \hat{y} = (b_y, y) \mapsto stut_1(\hat{y})$$
$$\mapsto chan\,(stut_1(\hat{y})) \mapsto$$
$$read_2\,(chan\,(stut_1(\hat{y}))) \mapsto valid_2\,(read_2\,(chan\,(stut_1(\hat{y})))) \mapsto y^{\mathbf{rec}}$$

where :

- $\hat{y} = (b_y, y)$ is the coding of y at the writer, with the added boolean flag b_y ;
- $stut_1$ stands for "stuttering" by C_1, it means that $\hat{y}$ is repeated at each writing instant, by C_1 ;
- *chan* is the function implemented by the channel ;
- $read_2$ denotes reading by machine C_2 ;
- $valid_2$ is the validation mechanism at C_2, it uses the sustained value of the boolean flag, as transferred by the channel.

Condition (7) of theorem 3 is sufficient to validate the whole design, since on the other hand we have *delay insensitive programs* running on each machine (this was a separate issue, solved by endo/isochrony). Also, since (7) is a statement which is, by definition, invariant under any delay, additional latency of the bus is not a "logical" issue any more. We only need to make sure that latency inevitably resulting from our approach (we sustain values and propagate them, and look for changes, this causes anyway latency) does not result in starvation (C_1 waiting for a value from C_2 which takes longer time to come than the valid period he has for emitting his own message, a problem arising when there is mutual feedback between the two machines).

But the nice point is that this timeliness issue is now *orthogonal* to logical correctness, and can therefore be analysed separately, making validation much simpler !

Acknowledgement. Paul Le Guernic is gratefully acknowledged for having pointed out an error in an earlier version of the manuscript.

References

[1] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Kluwer Academic Publishers. 1997. ISBN 0-7923-9894-7.

[2] R. Bannatyne. Time Triggered Protocol: TTP/C, *Embedded Systems Programming,* 9/98, pp. 52-54.

[3] A. Benveniste, B. Caillaud, and P. Le Guernic. From synchrony to asynchrony. In J.C.M. Baeten and S. Mauw, editors, *CONCUR'99, Concurrency Theory, 10th International Conference*, volume 1664 of *Lecture Notes in Computer Science*, pages 162–177. Springer, August 1999.

[4] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: specification & distributed code generation. *Information and Computation*, to appear.

[5] P. Caspi, C. Mazuet, R. Salem, and D. Weber. Formal design of distributed control systems with lustre. In *Proc. Safecomp'99*, September 1999.

[6] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communication of the ACM*, 21:558–565, 1978.

[7] M. Pease, R.E. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–237, 1980.