High-Assurance Quasi-Synchronous Systems

Robin Larrieu – École Polytechnique, Palaiseau FRANCE
Natarajan Shankar – SRI International, Menlo Park CA USA

Abstract—The design of a complex cyber-physical system is centered around one or more models of computation (MoCs). These models define the semantic framework within which a network of sensors, controllers, and actuators operate and interact with each other. In this paper, we examine the foundations of a quasi-synchronous model of computation Our version of the quasi-synchronous model is inspired by the Robot Operating System (ROS). It consists of nodes that encapsulate computation and topic channels that are used for communicating between nodes. The nodes execute with a fixed period with possible jitter due to local clock drift and scheduling uncertainties, but are not otherwise synchronized. The channels are implemented through a mailbox semantics. In each execution step, a node reads its incoming mailboxes, applies a next-step operation to update its local state, and writes to all its outgoing mailboxes. The underlying transport mechanism implements the mailbox-to-mailbox data transfer with some bounded latency. Messages can be lost if a mailbox is over-written before it is read. We prove a number of basic theorems that are useful for designing robust high-assurance cyber-physical systems using this simple model of computation. We show that depending on the relative rates of the sender and receiver, there is a bound on the number of consecutive messages that can be lost. By increasing the mailbox queue size to a given bound, message loss can be eliminated. We demonstrate that there is a bound on the age of inputs that are used in any processing step. This in turn can be used to bound the end-to-end sense-control-actuate latency. We illustrate how these theorems are useful in designing and verifying a thermostat-based heating system. Our proofs have been mechanically verified using the Prototype Verification System (PVS).

1 Introduction

Cyber-physical systems are composed of physical and computational components interacting through multiple control loops and at multiple time scales. These systems are realized through a distributed network of sensors, controllers, and actuators. Examples of such systems can range from engine controllers, cars, and robots to factories, buildings, and power grids. Cyber-physical systems are safety critical since failures can have catastrophic physical consequences. Since it is impossible to test these systems in all possible physical conditions, they must be designed to high levels of assurance. We outline an approach to high-assurance systems based on a quasi-synchronous model of computation. We present some basic theorems about this model that can be used in verifying system-level claims as part of an assurance case.

The academic literature on high-assurance distributed control systems has focused on the state machine model where the computation proceeds in consistent sequentially-separable rounds [5]. The separation of a distributed computation into rounds can be implemented through fault-tolerant clock synchronization. A range of clock synchronization algorithms have been developed depending on the computation and fault model [4]. If the rounds occur with sufficient separation as given by a sparse time base, then it is possible to build a synchronous abstraction that offers a simple programming model. These programming models, or models of computation, can then be used to guarantee global system properties. For example, the time-triggered model allows the computation to proceed in synchronous rounds with time-driven multiplexing of shared resources like the bus.

The quasi-synchronous model on the other hand assumes that individual computing nodes operate with independent clocks each with a bounded drift relative to absolute time. The computations local to a node are synchronous but all interaction between computing nodes can only rely on communication channels with known latency bounds and on bounded clock drift. Fault-tolerant clocks synchronization algorithms allow synchronous state machines to be derived from quasi-synchronous interaction. However, many practical systems can be built directly with quasi-synchrony. Caspi makes the point that as control systems evolved from analog systems with fluctuating delays to digital ones, quasi-synchrony was a natural evolution of the analog model. To quote from his "Cooking Book" [1]:

> ... in practice, at least in the domain of critical control systems, the use of clock synchroniza-

Supported by NASA NRA NNA13AC55C, NSF Grant CNS-0917375, and DARPA under agreement number FA8750-12-C-0284. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NASA, NSF, DARPA, or the U.S. Government.

> tion is not so frequent . . . We believe there are historical reasons for this fact, which can be found in the evolution of these systems: control systems formerly used to be implemented with analog techniques, that is to say without any clock at all. Then, these implementations evolved toward discrete digital ones, and then toward computing ones. When a computer replaced an analog board, it was usually based on period sampling according to a real-time clock. For the sake of modularity, when several computers replaced analog boards, each one came with its own clock.

Loosely time-triggered architectures that use communication by sampling (CbS) are an alternative way to implement synchronous models of computations using asynchronous models with bounded execution and communication delays [10].

It is generally believed that quasi-synchronous models are not suitable for high-assurance systems since it is difficult to bound the latencies. For example, Obermaisser [6] notes that

> Since the network load is not completely controlled, the quasi-synchronous channel exhibits a non-zero probability for violations of bounds on communication latencies.

We are using the quasi-synchronous model of computation in a context where the network loads can be bounded. Indeed, we are operating in a context where most of the choices such as threads, schedules, message sizes, and communication channels are statically fixed. In this context, it does become feasible to achieve latency bounds on quasi-synchronous communication. This in turn makes it possible to bound the end-to-end latencies in a sense-control-actuate system. Such a bound is crucial for ensuring the physical properties, as we illustrate in Section 4 with the example of a simple room heating thermostat controller. Also, since synchronous systems can be implemented by quasi-synchronous ones using clock synchronization, any claim that quasi-synchrony necessarily leads to unbounded delays is untenable.

Even with end-to-end latency bounds, there are some challenges for the direct use of quasi-synchrony such as fault tolerance and access to shared resources. We assume that shared resources are handled by the underlying middleware. For fault tolerance, one typically needs deterministic replication and voting as in the state machine model, and this is hard to achieve without clock synchronization. A sparse time discipline can be employed within quasi-synchrony in order to separate the computation into distinct rounds [3], [9]. Sparse time is specifically needed to collect a consistent view of the sensor inputs that can be identified as inputs to a single round. Another option is to use our results to identify bounds on the discrepancy between the computed value and an exact value for a given time. This bound can be used in computing a consensus value for driving the actuators. For discrete values such as Booleans, we can bound the durations of any discrepancy to devise voting schemes. Caspi [1] covers such schemes and provides a framework for safely implementing a restricted class of synchronous programs on a quasi-synchronous platform. We do not explore fault tolerance aspects in this paper since we are mainly interested in showing that we can achieve predictable real-time bounds with the quasi-synchronous model.

Our work is being carried out in the context of architectures that are defined using the Robot Architecture Definition Language (RADL).[1] This language is inspired by the ROS (Robot Operating System) architecture. RADL uses the abstraction of *nodes* and *topics* to describe the different components and communication channels. A topic specifies a message type with a default value. Each topic has a single node that publishes on it and zero or more nodes that subscribe to it. A node is similar to a process, but is specified together with its period, its list of subscribed topics with latency bounds, its list of published topics, and its worst-case execution time.

In each periodic step, a node reads messages from the subscribed topics, executes a computation, and publishes on the topics for which it is the publisher. Topic messages are communicated through mailboxes. Each publisher writes in a non-blocking manner to its outgoing mailbox and contents are then transferred by the underlying middleware to the subscriber mailboxes. If the subscriber mailbox is full, the oldest message is overwritten by the new message. We assume that the underlying architecture provides a bound on the maximum message latency for each publisher-subscriber channel. This is a reasonable expectation since there are a bounded number of message channels, where messages are published on each channel at a specified period.

Each RADL node is set to execute at a given frequency, but the actual rate to publish messages may vary because of clock bias, or execution times that depend on the inputs. Since several nodes may execute physically on the same processor, scheduling randomness adds to this imprecision. When a message is published on a topic, there may be a delay before it is received by each of the subscribers. This delay is caused by the nature of the communication link (network/shared memory) and the underlying mailbox

1. RADL is being developed as part of the DARPA-funded project *High-Assurance Cyber-Military Systems (HACMS)*.

system. Due to these factors contributing to message latency, we cannot ensure a subscriber will have new messages at each step, or that sent messages will be processed by the subscriber (they may be over-written before that). Figure 1 gives examples of these problems.

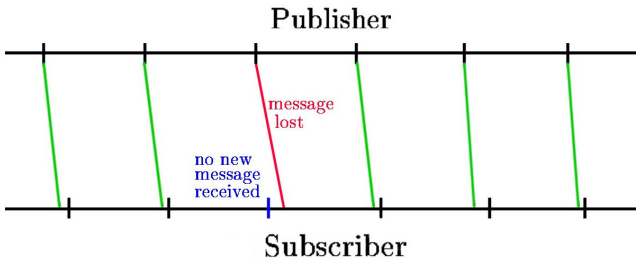

Figure 1. Uncertainty for the messaging system

In this paper, we will establish a worst-case scenario for this quasi-synchronous system. For example, knowing that a node will always have available at least one message with a bounded age can be sufficient to assert that a control loop maintains a state precisely enough around the desired value. This can also be used for monitoring purposes: if a node detects a behavior outside this worst case scenario, it can raise an alert to a supervisor. By collecting these alerts, the supervisor could detect a compromised/crashed node, or a broken link, and execute a mitigation strategy.

In Section 2, we give more details about the ROS architecture and the model used to represent it. In Section 3, we prove low-level properties for the messaging system. In Section 4, we give assurance claims for a simple end-to-end property that a basic control loop maintains a state around a set value. Section 5 presents some observations on this work and concludes with a discussion of future work.

2 The ROS architecture

2.1 Overview of ROS/RADL

The aim here is not to give a complete description of ROS (the interested reader could refer to ROS manuals, e.g. [7]). However, a few notions about ROS abstraction are useful for understanding following sections.

The main purpose of ROS is to provide a framework for distributed computing. Rather than writing one complex program that runs the whole robot, designers prefer writing several simpler programs (potentially running on different computers) in charge of a specific task: one program that uses sensors to scan the environment, one responsible to interact with the operator, another one controlling the wheels, and so on.

Each of these programs is represented by a ROS *node* that can send messages to other nodes. *Topics* are declared to group messages with the same function, and each node can be publisher or subscriber to one or more topics. The ROS master gives information to publishers (IP addresses, shared buffers and so on), so when publishing a message on a topic, they can send messages to all subscribers to this topic. Messages received by a subscriber are stored in a queue until they are computed. If the queue is full when a new message is received, the oldest message is thrown away.

We use the Robot Architecture Definition Language (RADL) to generate ROS deployments where the nodes execute at a fixed period over a fixed topology of topics.

2.2 Modeling assumptions

Topic assumptions: To avoid jamming, we require at most one publisher per topic[2]; this is ensured in our architecture by message authentication and firewall rules. For the purpose of simplicity, we assume in this paper that there is only one subscriber on each topic. This assumption does not lead to a loss of generality: a topic with n subscribers could be seen as n topics with one subscriber. Finally, we assume that the transmission delay for a given message is bounded.

Then, a topic can be characterized by two nodes (the publisher and the subscriber), the maximum transmission delay and the queue size for the subscriber.

Node assumptions: Nodes are discrete simple tasks that are executed regularly. Each node is supposed to execute at a given frequency, but the actual rate may vary. We assume that the time between two consecutive steps (or pseudo-period) is inside some known interval $[minT, maxT]$ with $minT > 0$. The typical example is when the frequency is known with a certain drift ρ, and the actual instantaneous frequency is in $[(1-\rho).f, (1+\rho).f]$.

We also make the simplifying assumption that the task at each step is executed instantaneously. Actually, in the case of a node that is only a publisher (respectively, subscriber)[3], the time for the execution event can be taken as the time when it sends (respectively, reads) the message, which are atomic operations. In the case of an intermediate node that is both subscriber and publisher on different topics, the computed duration between these two operations could just be added to the delay of the published topic.

2. If there is no publisher on a topic, a warning is generated, but this is considered non-critical.

3. For example, a sensor (respectively, actuator).

A node can be characterized by the upper and lower bound for the pseudo-period between consecutive steps.

Notations: Events are described by functions τ from $\mathbb{N}$ (the system runs forever) to $\mathbb{R}^+$ such that $\tau(E)$ is the time when event E occurs. For example, if e is the function used to describe the executions of some node, $e(n)$ is the time when the n'th execution occurs.

Definition 2.1. : *Node execution*

Each node N is defined by its minimal pseudo-period $minT(N)$ and maximal pseudo-period $maxT(N)$. Then an execution of N is any function e such that

$$e(0) = 0 \text{ and } \forall n \in \mathbb{N}, \begin{cases} e(n) + minT(N) \leq e(n+1) \\ e(n+1) \leq e(n) + maxT(N) \end{cases}$$

The assumption $e(0) = 0$ could seem too strong because it implies all nodes start exactly at the same time (which is quite unrealistic). First, having a common origin simplifies inductions, that are widely used in proofs. Second, we could prove that for any initial time shift Δ, with a nontrivial interval $[minT(N), maxT(N)]$ as small as desired, there exist executions for nodes N and N' such that $e(n) - e'(m) = \Delta$ for some n and m. This means that our model can be generalized to allow a time shift by assuming nodes N and N' send their default value (assumed to lead to a safe behavior) before periodic step n and m.

An induction gives the result for more than 2 nodes. The two nodes N and N' can be grouped into a single node $\hat{N}$ with period a common multiple of admissible periods for N and N': $\hat{e}(k) = k.T = e(n+ka) - e(n) = e'(m+kb) - e'(m)$. Assume N'' has an initial time shift of Δ' with N. Apply previous result with $\hat{N}$ and N'' for a time shift of $\Delta' - e(n)$: $e''(l) - \hat{e}(k) = \Delta' - e(n)$. Then, $e''(l) - e(n+ka) = \Delta'$ and $e''(l) - e'(m+kb) = \Delta' - \Delta$ (initial time shift between N' and N'').

Definition 2.2. : *Message reception*

Let T be a topic with a maximum transmission delay of D, and let s be an execution of its publisher S (associated to the event a message is sent*). A reception of these messages is any function r such that*

$$\forall n \in \mathbb{N},\ s(n) \leq r(n) \leq s(n) + D \quad \text{and} \quad r \text{ is injective}$$

The hypothesis that r is injective means that two different messages cannot be received exactly at the same time. This is linked to hardware limitations, and is needed to represent the queue: if messages m and n are received exactly at the same time, which one comes before the other in the queue?

Note that this definition does not assume the channel conserves the order of messages. We could have $r(k) > r(k+1)$ and therefore, the message received just after message n is not necessarily message $n+1$. However, we can define a function $next$ such that there is no message received between $r(n)$ and $r(next(n))$. Then the iterated function Nth_next is such that there is exactly $N-1$ messages received between $r(n)$ and $r(Nth\_next(N,n))$. This means $next(n)$ is the message received just after n is, and $Nth\_next(N,n)$ is the N'th message received after n is.

Definition 2.3. : *Processed message*

Let T a topic with a subscribers queue size of L. Let s and c be an execution of its publisher and subscriber and r be a reception of sent messages.

By definition, we have

$$\left|\left\{k \,|\, r(n) \leq r(k) < r(Nth\_next(L,n))\right\}\right| = L$$

Then, message n is in the queue at time t iff [4] $r(n) \leq t < r(Nth\_next(L,n))$

A message n is said to be processed *if and only if*

$$\exists k \in \mathbb{N}, \quad r(n) \leq c(k) < r(Nth\_next(L,n))$$

The message is said to be *lost* or *dropped* otherwise

3 Low-level messaging system

In this section, we consider a topic T, characterized by a publisher P, a subscriber S, a maximum transmission delay D, and a queue length L.

For $n \in \mathbb{N}$, we note $s(n)$ and $r(n)$ respectively the time when the n'th message is sent and received. We also note $c(n)$ the time when the subscriber executes its n'th computation.

3.1 Latency

For a processed message, we define its latency as the duration between the time it is sent and the time it is computed for the first time (see Figure 2):

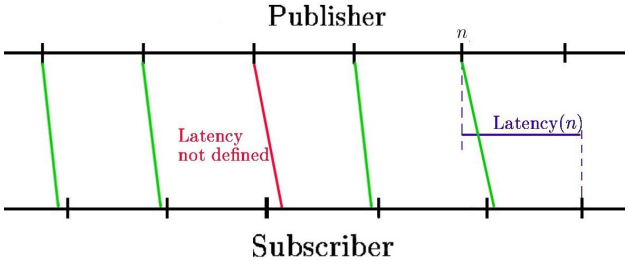

Figure 2. Latency definition

4. Actually, depending on the implementation, the message could be removed from the queue during a computation, but before $r(Nth\_next(L,n))$. This does not affect the definition.

Definition 3.1. : *Latency*

$$Latency(n) = c\Big(min(\{k \in \mathbb{N} | r(n) \leq c(k)\})\Big) - s(n)$$

Since $\forall k, c(k+1) \leq c(k) + maxT(S)$, we have $c\big(min(\{k \in \mathbb{N} | r(n) \leq c(k)\})\big) - r(n) \leq maxT(S)$. Therefore,

Theorem 3.1. : *Latency bound*

$$Latency(n) \leq maxT(S) + D$$

The latency is only defined for a processed message, so this gives little information for assurance properties. However, it can be used to detect a clock failure or a certain type of attack: assume for example that the subscriber reads a new message, but the delay since the messages timestamps exceed the latency bound. It could be that the clock of one of the nodes drifted significantly from real time, or it may mean that the message had been recorded and was resent later by an attacker.

3.2 Overtaking

The model defined in Section 2.2 allows an older message to be delivered after a more recent one. We may want to avoid this even if the mailbox system alone doesn't assert that this property holds. We give here a simple condition that asserts that messages are received in the order they are sent.

By avoiding overtaking, we can derive better bounds in some cases to assert other properties are satisfied. By choosing good node parameters, a developer can also secure the order of received messages and rely on this to prove correctness of an application.

We have $r(n) \leq s(n) + D$, $s(n) + minT(P) \leq s(n+1)$ and $s(n+1) \leq r(n+1)$. Hence the following theorem.

Theorem 3.2. : *Overtaking*

$$D < minT(P) \quad \Longrightarrow \quad \forall n \in \mathbb{N}, \quad r(n) < r(n+1)$$

3.3 Number of lost messages

As we saw before, messages can be overwritten in the queue before they are actually computed by the subscriber, which means these messages are never processed. However, we can ensure the subscriber does not miss too many consecutive messages

3.3.1 Upper bound for the number of consecutive lost messages

Definition 3.2. : *Consecutive lost messages*
Formally, we can define the property "the subscriber never misses N consecutive messages" *by*

$$\forall k \in \mathbb{N}, \exists l < N, \quad \text{message } k+l \text{ is processed}$$

First, we have the following fundamental lemma.

Lemma 3.3.1. *If $c(n) \geq t + D + maxT(P)$, there exists a message k with $t < r(k) \leq c(n)$ that is processed.*

Proof. $r(0) \leq D < c(n)$. Let m be the maximum of the $l \in \mathbb{N}$ such that $r(l) \leq c(n)$. Since $r(m+1) > c(n)$, we have $t < r(m) \leq c(n)$.

The set $S = \{l \in \mathbb{N}, t < r(l) \leq c(n)\}$ is finite and nonempty. There exists a $k \in S$ such that $\forall l \in S, r(l) \leq r(k)$. By construction, such a k solves the problem because no message is received between $r(k)$ and the next execution of the subscriber. □

Basic inequality reasoning gives the following result.

Lemma 3.3.2.

$$r(m) > s(n) + D - minT(P) \implies n \leq m$$

$$r(m) < s(n) + minT(P) \implies m \leq n$$

Theorem 3.3. : *Consecutive lost messages*
If $N.minT(P) > 2.D + maxT(S) + maxT(P) - minT(P)$, then the subscriber never misses N consecutive messages.

Proof. According to definition 3.2, given $k \in \mathbb{N}$, we prove there exists an $l \in [0, N-1]$ such that message $k+l$ is processed.

The subscriber executes at least once in every interval of time of length $maxT(S)$. In particular,

$$\exists n \in \mathbb{N}, \quad \begin{cases} s(k) + 2.D + maxT(P) - minT(P) < c(n) \\ c(n) \leq s(k) + 2.D + maxT(P) \\ \qquad\qquad -minT(P) + maxT(S) \end{cases}$$

Lemma 3.3.1 gives $\exists l \in \mathbb{N}, s(k) + D - minT(P) < r(l) \leq c(n)$ such that the message l is processed. Since $s(k+N-1) \geq (N-1).minT(P)$, with the given condition on N, $c(n) < s(k+N-1) + minT(P)$.

With lemma 3.3.2, we have $k \leq l \leq k+N-1$, and l is processed by construction. □

Case without overtaking: Assume now that $\forall k \in \mathbb{N}, r(k) < r(k+1)$. In that case, with $m = max(k \in nat, r(k) \leq c(n))$ we have $r(k) \leq c(n) \implies r(k) \leq r(m)$ which ensures message m is processed.

With $N.minT(P) > D + maxT(S)$, we have $\exists n \in \mathbb{N}, s(k) + D \leq c(n) < s(k+N)$, then $r(k) \leq c(n) < r(k+N)$. We deduce message m is processed with $k \leq m < k+N$.

Theorem 3.4. : *Consecutive lost messages – no overtaking*
Assume $N.minT(P) > D + maxT(S)$ and $\forall k \in \mathbb{N}, r(k) < r(k+1)$. Then the subscriber never misses N consecutive messages.

3.3.2 Influence of the queue length

As one can expect, increasing the message queue size leads to a lower message loss rate. For more comprehension, we first prove this in the case without overtaking. Then, we give an idea of the proof in the general case.

Assume we have $\forall k \in \mathbb{N},\ r(k) < r(k+1)$ and we never drop N ($N > 1$) messages with a queue length of L. Given $k \in \mathbb{N}$, there exists $l < N$ such that message $k+l$ is processed. If $l < N-1$, the result is proved. If $l = N-1$, it means the message $k+N-1$ was in the queue when some computation $c(n)$ occurred. With a queue length of $L+1$, the message $k+N-2$ was in the queue when $c(n)$ occurred, which means it was processed.

In the general case, we assume the property "never miss N consecutive messages" holds whatever r is as soon as the constraints with s and D are respected (Theorem 3.3 gives this assurance). Like in the previous proof, we assume $k+N-1$ is processed and none of the $k+l$ with $l < N-1$ is. Among non-processed messages received before $r(k+N-1)$, let m be the one received last. With a queue length of $L+1$, m is processed with the same argument as before. The difficult part is to prove that this m exists and must be one of the $k+l$, $l < N-1$. For this, we construct another r', consistent with the constraints on s and D but with N consecutive messages lost (which is a contradiction).

Assume that none of the $k+l$ verifies $r(k+l) < r(k+N-1)$. Let r' be a new reception sequence where reception of message $k+N-1$ is delayed to $r(k)+\varepsilon$ (see Figure 3.a), where ε is small enough to ensure r' verifies constraints of definition 2.2 and that no message is received between $r'(k)$ and $r'(k+N-1)$. In this case, messages k to $k+N-2$ are lost because messages that overwrote them with reception r still do with reception r', and message $k+N-1$ is lost because it is received between message k (that is lost) and the next processed message. This proves m exists and verifies $r(k+l) \leq r(m) < r(k+N-1)$ for some l.

Assume $m > k+N-1$. Let r' be the reception sequence where the order of reception of messages m and $k+N-1$ is switched (no change for other messages; see Figure 3.b). Message m was lost with r means $k+N-1$ is lost with r'. Again, messages k to $k+N-1$ are lost with r'.

Assume $m < k-1$ (case $m = k-1$ is impossible: $k-1$ must be processed since none of the $k, \ldots, k+N-2$ is). If $r(k-1) < r(m)$, switching reception of $k-1$ and m (see Figure 3.c) gives a contradiction with the same argument as previously. Otherwise, we consider the reception sequence r' where reception of message $k-1$ is anticipated to $r(m)-\varepsilon$. Then, messages $k-1$ to $k+N-2$ are lost (see Figure 3.d).

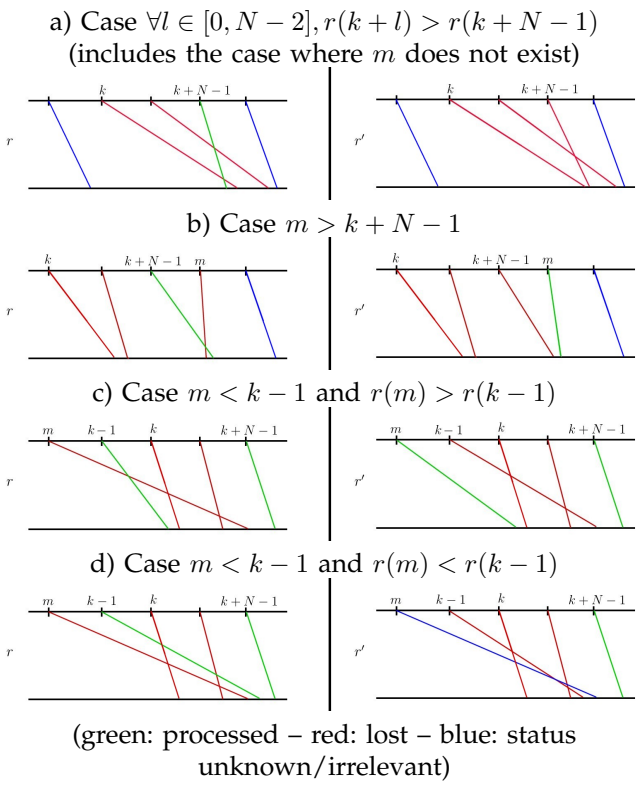

Figure 3. Getting contradictions

By a simple induction, we then get the following result:

Theorem 3.5. *Let N satisfy the conditions of either Theorem 3.3 or Theorem 3.4. Let $m < N$ and assume $L > m$. Then the subscriber never misses $N-m$ consecutive messages.*

In particular, if $L \geq N$, no message is lost.

3.4 Age of processed messages

In this section, we prove that at each computation, the subscriber gets (from a newly received message or with a backup from previous computation) a reasonably recent message.

This is essentially a bound on the age of processed messages that will be used in next section to prove end-to-end properties. It is also helpful to detect errors: when the latest available message to the subscriber is older than the bound, it can raise a timeout flag. By collecting these flags, a monitor could guess whether the publisher node crashed or there is a network failure.

3.4.1 Definition and basic properties

The aim is to model the following behavior (see Figure 4): At each computation, if messages are available

in the queue, the node chooses the most recent one and saves it to give a backup solution for next computation. If not, it uses the backup given by previous step (while no message has been received, and no backup is available, a default value – set at initialization time – is used)

Definition 3.3. : ***Age for the most recent available message***

For $n \neq 0$, we define the (finite, but maybe empty) set of messages computed at step n by

$$PL(n) = \left\{ k \in \mathbb{N} \;\middle|\; \begin{array}{l} c(n-1) < r(k) \leq c(n) \wedge \\ \text{message } k \text{ is processed} \end{array} \right\}$$

Then, the age of the most recent available message can be defined by the recursive function

$$\begin{array}{l} \texttt{Age}(n) \;=\; \{ \\ \quad \texttt{if } n = 0 \texttt{ then } 0 \\ \quad \texttt{else } \{\texttt{if } PL(n) = \emptyset \\ \qquad \texttt{then } \{\texttt{Age}(n-1) + c(n) - c(n-1)\} \\ \qquad \texttt{else } \{c(n) - s(max(PL(n))\} \\ \}\} \end{array}$$

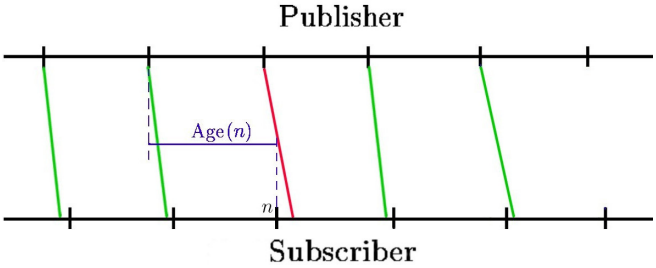

Figure 4. Age definition

Lemma 3.4.1. : ***Basic properties***

A simple induction over n gives $\texttt{Age}\,(n) \leq c(n)$.

Also, with the same argument as in the proof of lemma 3.3.1, we have:

$$PL(n) = \emptyset \iff \{k \in \mathbb{N}, c(n-1) < r(k) \leq c(n)\} = \emptyset$$

3.4.2 Upper bound in worst case scenario

Theorem 3.6. : ***Age bound***

$$\texttt{Age}\,(n) < 2.D + maxT(P)$$

Proof. First, we see by a simple induction over n that:

$$r(k) \leq c(n) \implies \texttt{Age}\,(n) \leq c(n) - r(k) + D$$

(if $PL(n) = \emptyset$, lemma 3.4.1 gives $r(k) \leq c(n-1)$. Otherwise, using the definition of a processed message, we either have $r(k) \leq r(max(PL(n)))$ or $k \leq max(PL(n))$. In both cases, $s(max(PL(n))) \geq r(k) - D$)

In the case $c(n) \leq D + maxT(P)$, lemma 3.4.1 gives the result. Otherwise, by lemma 3.3.1, there exists a message k such that $c(n) - D - maxT(P) < r(k) \leq c(n)$, which proves the theorem with the property above. □

When messages are delivered in the same order they were sent, we get a better bound:

Theorem 3.7. : ***Age bound – no overtaking***

$$\forall k \in \mathbb{N},\; r(k) < r(k+1) \implies \texttt{Age}\,(n) < D + maxT(P)$$

Proof. When no overtaking is possible, the message $m(n) = max(\{k \in \mathbb{N}, r(k) \leq c(n)\})$ (assuming the considered set is non empty) is processed.

Since $r(m(n)+1) \leq s(m(n)) + D + maxT(P)$, we have

$$c(n) - D - maxT(P) < s(m(n)) \leq c(n)$$

$r(m(n)) \leq c(n-1) \implies m(n) = m(n-1)$. Therefore, we get by induction

$$\texttt{Age}\,(n) = \left\{ \begin{array}{ll} c(n) - m(n) & \text{if } r(0) \leq c(n) \\ c(n) & \text{if } r(0) > c(n) \end{array} \right.$$ [5]

□

4 Assurance claim for the plant controller

4.1 Physical model and hypothesis

In this section, we look at a control loop with a plant (external state we wish to maintain), a sensor which measures the plant state, a controller and an actuator. The actuator can be *on* or *off*, which gives two different progression modes for the plant. The controller sets the actuator *on* or *off* according to the input from the sensor and the settings from an operator who can decide to enable or disable control, and can set preferences for the value to maintain the state around. This is summarized in Figure 5. For now, we assume the operator enables the control and doesn't change the settings after it is initially set.

We consider the plant state as a real function of the time. The aim for the plant controller is to bring this state between two bounds and to maintain it there. A first example could be a thermostat: we want to maintain a room temperature around a comfortable value by switching a heater *on/off* when needed. We can also imagine maintaining a correct speed for a vehicle by giving or not power to the engine.

For better comprehension, we will use the vocabulary of the thermostat example in the rest of this section (but the model is actually more general). We name $\theta(t)$ the temperature at time t, and τ and Γ the upper and lower bounds in which we want to maintain the

5. which means $c(n) < D$

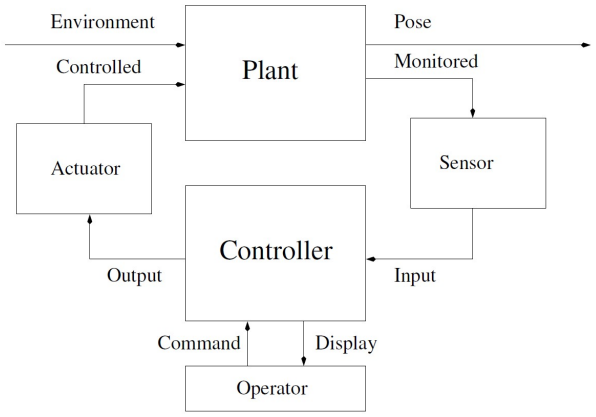

Figure 5. The plant controller model

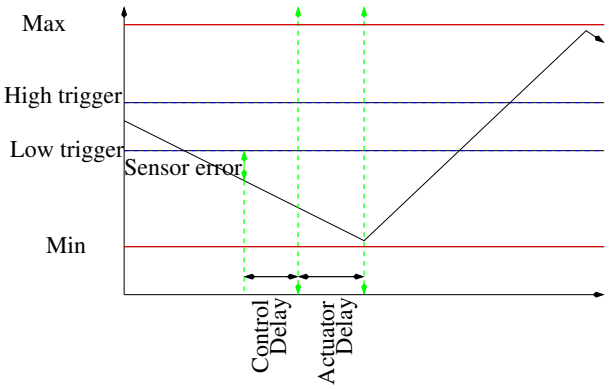

Figure 6. The effect of sensor inaccuracy and control and actuation latencies on the controller.

temperature. We consider that the room leaks energy, and that the heater is powerful enough to increase the room temperature. Typically, while the heater is off, the room temperature decreases with a rate of at most μ, and it increases with a rate of at least $\nu - \mu$ ($\nu > \mu$) while the heater is on. More precisely, considering the temperature is piecewise differentiable,

$$\begin{cases} 0 < \nu - \mu < \dot{\theta} < \Omega & \text{If the heater is } on \\ -\mu < \dot{\theta} < -\omega < 0 & \text{If the heater is } off \end{cases}$$

Assuming the outside temperature is lower than the inside temperature (which explains the use of a heater), this model is quite realistic, at least in a reasonable inside temperature range.

To represent real-life devices, we allow the sensor to be slightly inaccurate. Therefore, if the actual temperature is θ, the sensed temperature will be σ with $\theta - \epsilon \leq \sigma \leq \theta + \epsilon$

4.2 Controller strategy

The intuitive way is to switch the heater *on* when the temperature is too low, and switch it *off* when the temperature is too high. We define two trigger temperatures Δ_1 and Δ_2 with $\tau < \Delta_1 \leq \Delta_2 < \Gamma$. When the measured temperature is below Δ_1, the controller switches the heater to *on*, and when the measured temperature is above Δ_2, the controller switches the heater to *off*. The behavior when the temperature is between Δ_1 and Δ_2 doesn't really matter, so we decide for example to resend previous command.

Of course, we do not use τ and Γ as triggers because of the sensor inaccuracy and because of the potential message latency. Figure 6 shows why the thresholds have to be chosen conservatively to account for the sensor inaccuracy and the control and actuation latencies.

This strategy could be implemented as follows: The controller keeps a binary setting, initialized to some default value. At each step, if a new message have been received, the controller chooses the most recent one and look at the sensed temperature σ inside. If $\sigma < \Delta_1$, the setting is turned to *on*. If $\sigma > \Delta_2$, the setting is set to *off*. Otherwise, no change is applied to the setting. After this test, the current setting is sent to the actuator. (using the previous setting also in case no new message have been received is equivalent to use a backup from the previous computation)

4.3 Correctness proof

To prove correctness of the system, we need to apply this physical model to our logical architecture. We name S, C and A the nodes for the sensor, the controller and the actuator respectively. We note $s(n)$, $c(n)$ and $a(n)$ the time when their respective nth computation occurs. Let $Input$ and $Output$ be the topics for the sensed temperature and control commands respectively (that is, S publishes to $Input$, C subscribes to $Input$ and publish to $Output$, and A subscribes to $Output$). Let finally $r_1(n)$ and $r_2(n)$ be the times when the nth message in these topics is delivered.

Definition 4.1. : *Plant controller characteristic time*

For a topic T, we note $MA(T)$ some bound for the age of latest available message (see definition 3.3) at each step (given for example by theorem 3.6 or theorem 3.7).

The characteristic time for the plant controller is the quantity

$$\lambda = MA(Input) + MA(Output) + maxT(A)$$

Theorem 4.1. : *State stability*

Assume $\theta(t) \geq \Delta_1 - \epsilon \geq \tau + \lambda.\mu$ for some time t. Then, $\theta(t)$ never drops below τ after that: $\forall t' \geq t$, $\theta(t') > \tau$

Symmetrically, assume $\theta(t) \leq \Delta_2 + \epsilon \leq \Gamma - \lambda.\Omega$ for some time t. Then, $\theta(t)$ never becomes above Γ after that: $\forall t' \geq t$, $\theta(t') < \Gamma$

Proof. Let $t' \leq t$ and $g = glb(\{u \in \mathbb{R}, t \leq u \leq t' \text{ and } \theta(u) \leq \Delta_1 - \epsilon\})$. By definition, $\theta(g) = \Delta_1 - \epsilon$ and $g < u \leq t' \implies \theta(u) < \Delta_1 - \epsilon$.

Given $\dot{\theta} > -\mu$, if $t' \leq g + \lambda$, the result is proved. Otherwise, we prove $\theta(t') \geq \theta(g + \lambda) > \tau$, because the actuator is always *on* in the time interval $[g + \lambda, t']$:

Let $u \geq g + \lambda$. The actuator state at time u was set at the latest computation $a(n)$ of A. $a(n + 1) \leq a(n) + maxT(A)$ so we have $a(n) > g + MA(Input) + MA(Output)$. The corresponding command from the thermostat was issued at some $c(k)$ and by definition of $MA(Output)$, we have $c(k) > g + MA(Input)$. This command was computed by comparing a measured temperature with Δ_1 and Δ_2. Let $s(l)$ the execution that measured this temperature. We have $s(l) > g$ (by definition of $MA(Input)$), which means $\theta(s(l)) < \Delta_1 - \epsilon$. Therefore, the sensed temperature was $\sigma(s(l)) < \Delta_1$ □

Theorem 4.2. : *State convergence*

Assume $\theta(t) < \Delta_1 - \epsilon$. Then we have $\theta(t') \geq \Delta_1 - \epsilon$ for some t' with

$$t' < t + \lambda + \frac{\Delta_1 - \epsilon - \theta(t) + \lambda.\mu}{\nu - \mu}$$

Symmetrically, assume $\theta(t) > \Delta_2 + \epsilon$. Then we have $\theta(t') \leq \Delta_2 + \epsilon$ for some t' with

$$t' < t + \lambda + \frac{\theta(t) + \lambda.\Omega - \Delta_2 - \epsilon}{\omega}$$

Proof. Assume for all u with $t \leq u \leq T = t + \lambda + \frac{\Delta_1 - \epsilon - \theta(t) + \lambda.\mu}{\nu - \mu}$, we have $\theta(u) < \Delta_1 - \epsilon$. With the same argument as in previous theorem, the heater is always on in the time interval $[t + \lambda, T]$.

During this interval, the temperature increases at a minimum rate of $\nu - \mu$. Therefore, $\theta(T) > \theta(t + \lambda) + (\Delta_1 - \epsilon - \theta(t) + \lambda.\mu)$. But $\theta(t + \lambda) > \theta(t) - \lambda.\mu$, which leads to a contradiction □

5 Conclusions

Quasi-synchronous systems consist of computing nodes that are locally synchronized using independent local clocks that have a bounded drift relative to absolute time, and that communicate through message channels with bounded delays. These are popular models for the construction of cyber-physical systems as evidenced by the support for them in the ROS middleware. A lot of the academic work has focused on digital models of distributed systems needed for file systems, bank transactions, and stock exchanges. Quasi-synchrony, on the other hand, is well-suited to physical distributed systems where there are physical bounds on the rates at which physical parameters can change. These rate limits can be exploited by bounding the end-to-end latency of the sense-control-actuate loop.

We have a formalized a specific model for quasi-synchrony that is used in the RADL architecture definition language. This model supports mailbox-to-mailbox communication where the input mailboxes are queues of bounded length. In this formalization, we have shown the following

1) Bounded processing latency for a message. Either a message is over-written or it is processed within a bound (Theorem 3.1)
2) No overtaking, with timing assumptions (Theorem 3.2)
3) Bounded consecutive message loss (Theorems 3.3 and 3.4)
4) Bounded queue length to eliminate message loss (Theorem 3.5)
5) Bounded age of messages used by subscriber (Theorems 3.6 and 3.7)

The above theorems are fundamental to proving other metatheorems about the quasi-synchronous models as well as for proving properties of specific systems. We have illustrated this with a correctness proof for a simple room heating thermostat controller. These theorems can guide the design of high-assurance cyber-physical systems. For example, lossless transmission can be achieved either increasing the subscribers queue size or the by increasing the publishers frequency so that each message is repeated $N+1$ times if at most N consecutive messages can be lost. The same holds for messages communicating events like button pushes. These messages should be broadcast in a lossless manner to avoid missed events. We can also use these results to bound the size of the input commands that are being pipelined through the system but ensuring that input commands are generated at a slow enough rate to flush older commands out of the system in order to reduce hysteris. The proofs of the above theorems have been verified in PVS [8] as has the correctness of the room heating thermostat controller.

As future work, we are applying this model of computation within the DARPA HACMS project to verify properties of land-based vehicles such as collision avoidance, constant-speed cruise control, and adaptive cruise control. We plan to integrate these proofs into assurance cases that are built using the Evidential Tool Bus (ETB) [2] for system architectures that are defined using RADL. We also plan to use these theorems to show how we can embed other models of computation into the quasi-synchronous model.

REFERENCES

[1] P. Caspi. The quasi-synchronous approach to distributed control systems. Technical Report CMA/009931, VERIMAG, Crysis Project, May 2000.

[2] Simon Cruanes, Grégoire Hamon, Sam Owre, and Natarajan Shankar. Tool integration with the evidential tool bus. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, volume 7737 of *Lecture Notes in Computer Science*, pages 275–294. Springer, 2013.

[3] Hermann Kopetz. Sparse time versus dense time in distributed real-time systems. In *Distributed Computing Systems, 1992., Proceedings of the 12th International Conference on*, pages 460–467. IEEE, 1992.

[4] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[5] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(2):254–280, 1984.

[6] Roman Obermaisser. *Event-Triggered and Time-Triggered Control Paradigms*. Springer, 2012.

[7] Jason M. O'Kane. *A Gentle Introduction to ROS*. CreateSpace Independent Publishing Platform, 10 2013.

[8] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. 21(2):107–125, February 1995. PVS home page: http://pvs.csl.sri.com.

[9] W Steiner and J Rushby. TTA and PALS: Formally verified design patterns for distributed cyber-physical systems. In *Digital Avionics Systems Conference (DASC), 2011 IEEE/AIAA 30th*, pages 7B5–1. IEEE, 2011.

[10] Stavros Tripakis, Claudio Pinello, Albert Benveniste, Alberto Sangiovanni-Vincent, Paul Caspi, and Marco Di Natale. Implementing synchronous models on loosely time triggered architectures. *Computers, IEEE Transactions on*, 57(10):1300–1314, 2008.