PALS: Physically Asynchronous Logically Synchronous Systems

Lui Sha[1], Abdullah Al-Nayeem[1], Mu Sun[1], Jose Meseguer[1], Peter Ölveczky[2],
[1]University of Illinois at Urbana Champaign, [2]University of Oslo

Abstract

In networked cyber physical systems real time global computations, e.g., the supervisory control of a flight control system, require consistent views, consistent actions and synchronized state transitions across network nodes in real time. This paper presents a real time logical synchrony protocol, Physically Asynchronous Logically Synchronous (PALS), to support real time global computation. Under the PALS protocol, engineers design and verify applications as if all the distributed state machines were driven by a single global clock. The PALS protocol is optimal in the sense that 1) the bound on the periods of the real time global computation, such as the supervisory controller, is the shortest possible, and 2) the message overhead in achieving logical synchrony is minimal.

Acknowledgement. Steven P. Miller and Darren Cofer have collaborated with us closely in this work. Min Young Nam, Peter Feiler and Dionisio de Niz have helped us greatly in AADL related challenges. Xiaokang Qiu and Artur Boronat contributed to the translation of AADL to Maude.

This work is sponsored by Rockwell Collins Inc., the Office of Naval Research , the National Science Foundation, Lockheed Martin Corporation, the Research Council of Norway, and the Software Engineering Institute.

1. Introduction

Networked real time control systems have both global and local computations. For example, the supervisory control of a flight control system is a real time hybrid control that 1) periodically adjusts the setpoints of engine speeds and control surface angles, and 2) performs discrete control such as mode changes, e.g., changing the primary controller in a dual redundant flight control system. The supervisory control is a global computation because during mode changes, the views, actions and state transitions of the distributed state machines must be consistent with each other. Local computations may use a combination of local data and/or a subset of globally consistent views and commands provided by the supervisory control. However, local computations do not need to synchronize with the others. For example, in a flight control system, servo control at each node is a local computation, which is solely a function of the local tracking errors with respect to its setpoint. For example, consider a command given to increase the propulsion power of both left and right engines by 10%. If the left engine has a mechanical problem and loses power, the right engine will still execute the command of increasing power by 10%. It is up to the supervisory control to re-adjust the setpoints and regain control.

In the aviation community, networked real time systems are known as Globally Asynchronous Locally Synchronous (GALS) systems, because local tasks are driven by the same clock. As a result, synchronous task interactions within a node can be accomplished easily. On the other hand, skews between different nodes' clocks can only be bounded but not eliminated. When interactions between nodes are directly driven by their local clocks, the resulting interactions become asynchronous. Discrete supervisory controls, such as commands for mode changes, are very sensitive to asynchronous interactions. Suppose that the two clocks of a dual redundant control system have a bounded skew of ϵ with respect to a global clock. As a result, one subsystem can be in state j but the other lags by 2ϵ and remains in state $j-1$. Hence, one could receive the command in state j while the other receiving the command in state $j-1$, leading to potential divergence between the replicated machines. An aircraft has hundreds of computers networked together. Different clocks have somewhat different relative skews, and there can be tens to even hundreds of concurrent discrete commands in the hierarchical control of a complex modern flight control system.

To verify discrete control logics in the presence of asynchronous interactions, a model checker has to examine all possible state transitions under all possible clock skew combinations tick by tick. This creates a combinatorial explosion of interaction state space. Comparing a synchronous system against an asynchronous system for dual redundant flight guidance system, Miller et. al. made the following observation: "*the properties themselves are more difficult to state, were weaker than could be achieved in the synchronous case, and required considerable complexity to be added to the model to ensure that even the weakened properties were true*" [11].

Protocols to provide distributed processes with consistent views in general purpose distributed computing have been an active area of research. Birman and Joseph [1] first introduced the process group abstraction to achieve fault tolerant virtual synchrony for general purpose computation. Guo, Vogels and Renesse introduced a light weight version [2]. They reported that their light weight version achieved a one-way synchronization latency of 100 msec over a group of 500. Pereira, Rodrigues and Olivera [3] exploited application-level semantics to relax some virtual synchrony constraints. Robinson and Schmid [6] developed an asynchronous bounded cycle model to support logically locked step actions in distributed computation. Delporte-Gallet and Fauconnier [4] presented a soft real time atomic broadcast protocol where each message has an explicit termination deadline, so that the atomic broadcast will either be done in time or aborted. Tarek, Shaikh, Jahanian, and Shin [7] developed a light weight fault tolerance multicast and membership service for real time process groups using a logical ring for concurrency control. When an application needs to send real time messages, it presents the message with timing constraints to an admission controller to perform online schedulability analysis. Real time messages that can be scheduled will be admitted. Otherwise, they will be rejected.

In a typical progress group approach, synchrony management mechanism and fault tolerance mechanism are bundled together. This is a good idea for many applications. However, in safety critical CPS applications such as avionics, a system has subsystems with different levels of reliability requirements. For example, avionics certification standard DO178B defines 5 design assurance levels [16]. Hence, we separate our real time synchrony mechanism from fault tolerance mechanisms, so that designers can combine real time synchrony mechanism with preferred fault tolerance mechanisms to meet different reliability requirements.

The objective of PALS (Physically Asynchronous Logically Synchronous) protocol is to provide the optimal real time logical (virtual) synchronization protocol, under which a machine M_i's views, actions, and state transitions driven by local clocks with bounded skews are identical to M_i's views, actions and state transitions driven by perfectly synchronized clocks (synchronized perfectly with an idealized global clock).

Under the PALS design process, engineers first design and verify their application as if it is a synchronous system. Each component is then allocated to distributed nodes driven by clocks with bounded skews. When the PALS protocol is followed, the logical behavior of this physically asynchronous system is identical to that of the synchronous system.

The PALS protocol based design can be specified in a formalized version of AADL [12], an architecture analysis and description language. The AADL description can be automatically translated into Real Time Maude [14] to perform model checking. If the design passes the check, engineers will bind the AADL logical design to an AADL hardware specification of a networked system with bounded clock skews between nodes. The resulting design is then checked if the PALS protocol is followed. Due to the page limitation, the formal model of PALS architecture pattern, and the design tools will be published separately. This paper presents the PALS protocol.

2. The PALS Protocol

We first define the global computation model. Next, we examine the properties of global computations driven by perfectly synchronized clocks with period T. We then show that these behaviors are logically identical to that of the same system driven by clocks with bounded skews and period T under the PALS protocol, in the sense that they have the identical state transitions and identical inputs and outputs.

G1: Local Clocks for Global Computation. Each state machine M_i engaged in global computation is driven by a local clock C_i with the same period T. The global clock time is denoted as t. Clock C_i is said to be at its j^{th} period, denoted as $C_i = j$, if the global time t satisfies the constraint, $\uparrow (C_i = j) \leq t <\uparrow (C_i = j + 1)$; where $\uparrow (C_i = j)$

is the rising edge of clock C_i, when it just enters its j^{th} period.

All the local clocks used for global computation are synchronized with the perfect global clock with clock skews of at most ϵ. If $\epsilon = 0$, all clocks are perfectly synchronized. A system driven by perfectly synchronized clocks and the same system driven by the perfect global clock have the same behaviors. Note that is important to ensure that the clock values are not only monotonic but also avoid large jumps. When a clock is ahead/behind, it should be corrected by decreasing/increasing its rate of progress. A clock value that goes backwards or has large jumps generates serious errors in the computation of velocity and acceleration.

Finally, in addition to global computation, a node can also perform local computations. Local computations are modeled by different state machines. The clocks used for local computations do not need to be synchronized with the clocks for global computation. In practice, it is convenient to synchronize a set of local clocks at some rate and then derive all other clock values. For example, we may choose to synchronize all the 100 Hz real time clocks and then derive all the other clock rates from this 100 Hz clock, including those used for global computation.

G2: Real Time Network. The network has a network queueing (scheduling) delay q bounded by $0 < q_{min} \leq q \leq q_{max}$ and a network transmission delay μ bounded by $0 < \mu_{min} \leq \mu \leq \mu_{max}$.

G3: Real Time Machine. Messages arriving at real time machine M_i during M_i's j^{th} clock period are buffered. At $\uparrow (C_i = j + 1)$, M_i reads the messages from the buffer, carries out the computation, transitions to the next state, and sends output messages. The task completion time α, including real time scheduling, computation, and I/O is bounded by $0 < \alpha_{min} \leq \alpha \leq \alpha_{max}$. Since state transitions are driven by the clock, we say that a state machine M_i is at its j^{th} state, when its clock is at its j^{th} period.

To simplify notation, we assume that a (global computation) state machine sends and receives messages from and to every machine at each state. When there is no physical message, then it is modeled as sending/receiving messages with the "null" value that has no effect on the computation.

A CPS system may interact with the external environment. When the environment sends a message to two replicated state machines, the network delays may be different. This can result in one machine receiving the data at clock period j, while the other receives it at clock period $j + 1$, even if their clocks are perfectly synchronized. The inconsistent views may lead to the divergence of the state machines. To avoid this problem, we need an environment message I/O synchronizer.

G4: Environment Message I/O Synchronizer. Let the input synchronizer, $M^{I\text{-}syn}$, be a real time machine as defined by G3. Messages from the environment are sent to the input synchronizer. Messages arriving at $M^{I\text{-}syn}$ during $\uparrow (C^{I\text{-}syn} = j) \leq t < \uparrow (C^{I\text{-}syn} = j + 1)$ are buffered. At $\uparrow (C^{I\text{-}syn} = j + 1)$, the input synchronizer reads buffered messages and forwards them to their destinations. When machines need to send messages to the external environment, they send them to the output synchronizer. Similarly, the output synchronizer, $M^{O\text{-}syn}$, reads messages at the rising edge of its clock tick and forwards them to the environment. The output synchronizer allows an external observer to have a synchronous view of the distributed states of a global computation.

We note that a networked computer is typically shared by state machines for both global computation and local computation. Local computation state machines can perform their local I/O independently of the PALS protocol. For example, a local servo controller reads the states of the local physical device, compares them to the setpoint provided by the supervisory controller, computes the control commands, and sends them to the device at a rate that is typically higher than the PALS clock rate used for supervisory control.

G5: The Period of Perfectly Synchronized Clocks. Giving a set of perfectly synchronized clocks used to drive global computation, the clock period T should satisfy the constraint $T > \alpha_{max} + q_{max} + \mu_{max}$.

We now state the properties of a distributed real time system driven by perfectly synchronized clocks modeled by G1, G2, G3, G4 and G5.

Fact 1. Under a set of perfectly synchronized clocks, $C_i, 1 \leq i \leq N$, defined by G5, when a machine M_s sends a message during period $C_s = j$, this message will reach all the N receiving machines, $M_r, 1 \leq r \leq N$, before their next clock ticks at $\uparrow (C_r = j + 1), 1 \leq r \leq N$. That is, a message sent during sender's local clock's j^{th} clock period will be received when receiving machines are still in their j^{th} period.

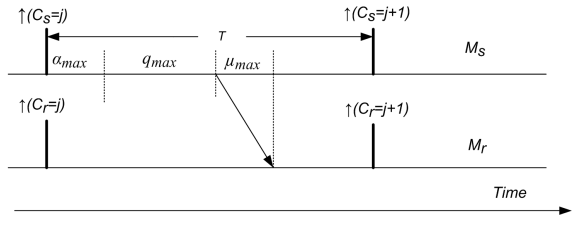

Figure 1: A System Using Perfectly Synchronized Clocks

Proof. As illustrated by Figure 1, by G5 any pair of sender M_s and receiver M_r, the distance between sender's rising clock edge at period j and any receiver's rising clock edge at next period $j+1$ is T. That is, $\uparrow (C_r = j+1) - \uparrow (C_s = j) = T$. Since $T > \alpha_{max} + q_{max} + \mu_{max}$, Fact 1 follows.

Fact 2. Under a set of perfectly synchronized clocks, $C_i, 1 \leq i \leq N$, with their period T defined by G5, any message from external environment received by input synchronizer during j^{th} period, will reach each of the N receiver machines, $M_r, 1 \leq r \leq N$, during $(j+1)^{th}$ period.

Proof. By G4, any message from environment must be sent to the input synchronizer. The messages arriving at time $\uparrow (C^{I\_syn} = j) \leq t < \uparrow (C^{I\_syn} = j+1)$ will be buffered at the synchronizer M^{I_syn}. By G4, these messages will be forwarded at $t = \uparrow (C^{I\_syn} = j+1)$. By Fact 1, the message will reach all the N receiver machines by $\uparrow (C_r = j+2), 1 \leq r \leq N$. Fact 2 follows.

We now examine a networked real time systems where global computations are driven by clocks with bounded skews.

PALS Clocks. All the local clocks used by the PALS protocol for global computation are synchronized with the global clock with skews of at most ϵ.

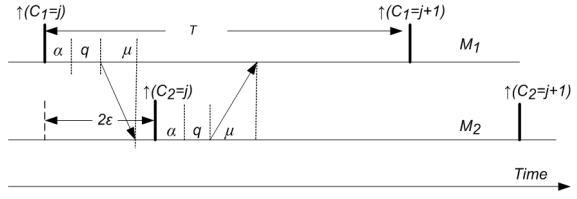

Figure 2: Logical Equivalence to Causality Violation

We now examine the effect of clock skews. In Figure 2, M_1 is a replication of M_2. First, M_1 sends a message to M_2 at time $t = \uparrow (C_1 = j) + \alpha + q$. At global time $t = \uparrow (C_1 = j)$, M_2's local time is at $\uparrow (C_2 = j) - 2\epsilon$. Suppose that the end to end delay from M_1 to M_2 is very short. That is, $\alpha + q + \mu < 2\epsilon$. Under this condition, the M_1's message transmitted during clock period j may reach M_2 when M_2 is still at clock period $j-1$. If the clocks were perfectly synchronized, this could only happen through a violation of causality. Since M_1 is a replication of M_2, this simulates sending a message to one's own past. Finally, note that M_2 also sends a message to M_1 at its j^{th} period at $t = \uparrow (C_2 = j) + \alpha + q$. This message is received by M_1 at its j^{th} period. Inconsistent views between replicated machines may lead to state divergence.

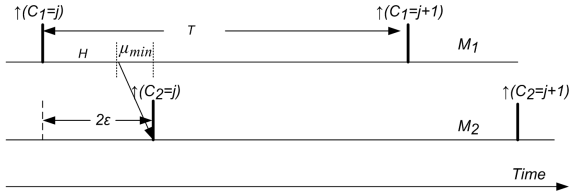

Figure 3: Clock C_1 leads Clock C_2

To make sure that M_1's message sent during its j^{th} period will arrive at M_2 no earlier than $\uparrow (C_2 = j)$, we introduce a minimal network queueing delay threshold H. As we can see in Figure 3, if M_1 sends its message at or after $\uparrow (C_1 = j) + H$, where $H = 2\epsilon - \mu_{min}$, then the message will arrive at M_2 no earlier than $\uparrow (C_2 = j)$. So the first rule of the PALS protocol, called the PALS Causality Rule, is to require that machine M_i at its local PALS clock period j can only transmit a message no earlier than $\uparrow (C_1 = j) + H$. Since the time lag between any pair of machines is less than or equal to 2ϵ, under the PALS Causality Rule when a message sent from a machine M_s during its j^{th} period cannot reach a receiving machine M_r earlier than $\uparrow (C_r = j)$.

G6: PALS Causality Rule. A machine M_i at (PALS) clock period j cannot send a message earlier than $\uparrow (C_i = j) + H$, where $H = 2\epsilon - \mu_{min}$.

We now examine the case where machine M_2 leads machine M_1 by 2ϵ, as illustrated in Figure 4.

Since clock C_1 now lags clock C_2, we need to ensure that a message sent by M_1 at j^{th} period

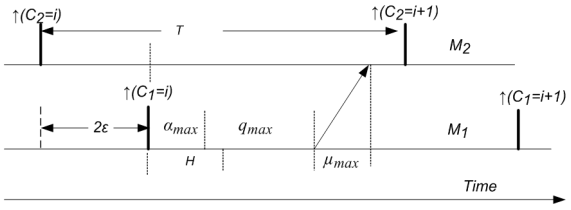

Figure 4: Clock C_2 leads Clock C_1

will reach M_2 before $\uparrow (C_2 = j + 1)$. As illustrated in Figure 4, the latest instant at which M_1 can transmit its messages on the network is $max((\uparrow (C_1 = j)+H), (\uparrow (C_1 = j)+\alpha_{max}+q_{max}))$. The maximal network transmission delay is μ_{max}. Hence, it is necessary that the clock period $T > 2\epsilon + max(\alpha_{max} + q_{max}, H) + \mu_{max}$. We now define the PALS clock period.

G7: PALS Clock Period. PALS clock period $T > 2\epsilon + max(\alpha_{max} + q_{max}, H) + \mu_{max}$

The definfions so far now allow us to define a PALS system.

PALS Definition. A PALS system consists of state machines, environment input synchronizer, environment output synchronizer, and PALS clocks defined by rules G1, G2, G3, G4, G6 and G7.

Fact 3. Under PALS rules, a message sent during sender's j^{th} clock period will be received by machines when they are still in their j^{th} clock period.

Proof. Suppose that Fact 3 is false. There are two possible cases.

Case 1. Assume that there exists a pair of machines, where M_1's message sent during its clock C_1's j^{th} period reaches M_2, during M_2's clock C_2's $(j-1)^{th}$ period.

Proof of Case 1. As illustrated in Figure 3, in order for machine M_2 to receive M_1's period j message at M_2's clock period $j-1$, M_2's clock must lag M_1's clock and the maximal lag is 2ϵ. Due to the PALS Causality Rule (G6), the earliest possible message arrival time at M_2, t_{arr}, is

$$\begin{aligned} t_{arr} &= \uparrow (C_1 = j) + H + \mu_{min} \\ &= \uparrow (C_1 = j) + (2\epsilon - \mu_{min}) + \mu_{min} \\ &= \uparrow (C_1 = j) + 2\epsilon \end{aligned} \quad (1)$$

However, M_2 lags M_1 at most 2ϵ. It follows that

$$(\uparrow (C_2 = j) - \uparrow (C_1 = j)) \leq 2\epsilon \quad (2)$$

Substitute 1 into 2, we have

$$\uparrow (C_2 = j) - \uparrow (C_1 = j) \leq t_{arr} - \uparrow (C_1 = j)$$

Hence, $\uparrow (C_2 = j) \leq t_{arr}$. This contradicts the Case 1's assumption.

Case 2. Assume that there exists a pair of machines, where a message from M_1 sent during its clock C_1's j^{th} period reaches M_2 during its clock C_2's $(j+1)^{th}$ period.

Proof of Case 2. As illustrated in Figure 4, to maximize the chance of machine M_2 receiving M_1's j^{th} message at M_2's clock $(j+1)^{th}$ period, C_1 should lag C_2 by the maximum 2ϵ.

The latest message arrival time at machine M_2, t_{arr} is:

$$\begin{aligned} t_{arr} &= \uparrow (C_1 = j) \\ &\quad + max(\alpha_{max} + q_{max}, H) + \mu_{max} \end{aligned} \quad (3)$$

The starting time of machine M_2's clock period $j+1$ is

$$\begin{aligned} \uparrow (C_2 = j+1) &= \uparrow (C_2 = j) + T \\ &= (\uparrow (C_1 = j) - 2\epsilon) + T \end{aligned}$$

Since $T > max(\alpha_{max} + q_{max}, H) + \mu_{max} + 2\epsilon$

$$\begin{aligned} &\uparrow (C_2 = j+1) \\ &\quad > (\uparrow (C_1 = j) - 2\epsilon) \\ &\qquad + (2\epsilon + max(\alpha_{max} + q_{max}, H) + \mu_{max}) \\ &\qquad = ((\uparrow (C_1 = j) \\ &\qquad\quad + max(\alpha_{max} + q_{max}, H) + \mu_{max}) \end{aligned} \quad (4)$$

Subtract Equation 3 from 4, we have $\uparrow (C_2 = j+1) - t_{arr} > 0$ That is, the message arrives at machine M_2 before $\uparrow (C_2 = j + 1)$. This contradicts the assumption of Case 2. By the proofs of Case 1 and Case 2, Fact 3 follows.

Fact 4. Under the PALS protocol, messages from the environment buffered by the Input Synchronizer during clock period j will reach all N destination machines at time t, $\uparrow (C_r = j + 1) \leq t < \uparrow (C_r = j + 2), 1 \leq r \leq N$.

Proof. Similar to the proof of Fact 3.

3. Proof of PALS Equivalence

In this section, we show that global computations under PALS protocol and under perfectly synchronized clocks are equivalent. Because we need to compare machines' behaviors under perfectly synchronized clocks and under PALS protocol, we add "g"

to the superscripts of variables representing state machines under perfectly synchronized clocks. we add "p" to the superscripts of variables representing state machines under perfectly synchronized clocks.

Let $M_i, 1 \leq i \leq N$ be a group of distributed state machines engaged in global computation. When machine M_i is driven by perfectly synchronized clock C_i, we denote the machine as M_i^g. When machine M_i is driven by PALS clock C_i with bounded skew ϵ, we denote the machine as M_i^p. Machine M_i^p follows the PALS PALS Causality rule.

Let $M_i^g(j)$ be the state of machine M_i under a perfectly synchronized clock at periods $C_i = j,\ j = 0, 1, 2, ..., L$. Let $M_i^p(j)$ be the state of machine M_i under a PALS clock at periods $C_i = j, j = 0, 1, 2, ..., L$. State 0 and state L are the initial state and the last state before termination respectively.

Let $I_i^g(j)$ be the set of messages received by M_i^g from machines other than the input synchronizer during state $M_i^g(j),\ j = 0, ..., L$. Let $I_i^p(j)$ be the set of messages received by M_i^p from machines other than the input synchronizer during state $M_i^p(j),\ j = 0, ..., L$. In addition, let $I_i^{g\text{-}syn}(j)$ be the messages received by M_i^g from the input synchronizer during state $M_i^g(j),\ j = 0, ..., L$. Let $I_i^{p\text{-}syn}(j)$ be the messages received by M_i^p from the input synchronizer during state $M_i^g(j),\ j = 0, ..., L$.

Let $O_i^g(j)$ be the set of messages sent by M_i^g to machines other than the output synchronizer during state $M_i^g(j),\ j = 0, ..., L$. Let $O_i^p(j)$ be the set of messages sent by M_i^p to machines other than the output synchronizer during state $M_i^p(j),\ j = 0, ..., L$. In addition, let $O_i^{g\text{-}syn}(j)$ be the messages sent by M_i^g to the output synchronizer during state $M_i^g(j),\ j = 0, ..., L$. Let $O_i^{p\text{-}syn}(j)$ be the messages sent by M_i^p to the output synchronizer during state $M_i^g(j),\ j = 0, ..., L$.

Assumption 1. At each clock period, identical messages are received by the input synchronizer for perfectly synchronized clocks and by the input synchronizer for PALS clocks. That is, $I^{g\text{-}syn}(j) = I^{p\text{-}syn}(j), j = 0, 1, ..., L$.

Note that the initial messages from the environment are buffered by the input synchronizer and will be delivered at $\uparrow (C^{g\text{-}syn} = 1)$ and $\uparrow (C^{p\text{-}syn} = 1)$ respectively.

Assumption 2. Machine M_i's initial states under a PALS clock and a perfectly synchronized clock are identical, that is, $M_i^g(0) = M_i^p(0), 1 \leq i \leq N$. In addition, initial messages pre-deposited in the input buffers are the same. That is, $I_i^g(0) = I_i^p(0), 1 \leq i \leq N$.

Assumption 3. Each machine, $M_i, 1 \leq i \leq N$, is deterministic and time invariant.

Assumption 4. The perfectly synchronized clocks C_i^g and the PALS clocks C_i^p has the same period T as defined by G7.

Theorem 1. Under Assumptions 1, 2, 3 and 4, a system under the PALS protocol is logically equivalent to the same system driven by perfectly synchronized clocks. That is, for each state $j = 0, 1, ..., L$, we have identical states $M_i^g(j) = M_i^p(j)$; identical inputs from state machines $I_i^g(j) = I_i^p(j)$; identical outputs to state machines $O_i^g(j) = O_i^p(j)$; and identical outputs to output synchronizers $O_i^{g\text{-}syn}(j) = O_i^{p\text{-}syn}(j)$.

Proof.

By Assumption 2, machine M_i under the PALS clock C_i^p and under a perfectly synchronized clock C_i^g has identical initial states and identical initial inputs. That is, identical states $M_i^g(0) = M_i^p(0), 1 \leq i \leq N$; and identical initial (null) inputs, i.e., $I_i^g(0) = I_i^p(0), 1 \leq i \leq N$. By Assumption 3, machine M_i under a PALS clock and under a perfectly synchronized clock makes identical state transitions and generates the identical outputs to state machines and to the output synchronizers. That is, $M_i^g(1) = M_i^p(1), 1 \leq i \leq N$, and $O_i^g(0) = O_i^p(0), O_i^{g\text{-}syn}(0) = O_i^{p\text{-}syn}(0), 1 \leq i \leq N$. Hence, at the initial state, machine M_i under PALS clock and under a perfectly synchronized clock are logically equivalent.

Assume that M_i^g and M_i^p are logically equivalent at state k, where $k > 0$. By this assumption, machine M_i under PALS clock C_i^p and under a perfectly synchronized clock C_i^g has identical states $M_i^g(k) = M_i^p(k), 1 \leq i \leq N$, and has identical inputs, $I_i^g(k) = I_i^p(k), 1 \leq i \leq N$, from the other state machines. In addition, by Assumption 1, the environmental input messages to M_i under PALS clocks and under a perfectly synchronized clocks are the same at state $k-1$. That is, $I_{syn}^g(k-1) = I_{syn}^p(k-1)$. By Fact 2 and Fact 4, the same environmental input messages will be in the buffers of $M_i^g(k) = M_i^p(k), 1 \leq i \leq N$, during their k^{th} state.

Since machine M_i under a PALS clock and under a perfectly synchronized clock at state k has identical states, identical environmental inputs, and identical inputs from state machines, by Assumption 3 machine M_i under a PALS clock and under a perfectly synchronized clock makes the identical

state transitions and generates the identical outputs to state machines and to the output synchronizer. That is, $M_i^g(k+1) = M_i^p(k+1), 1 \leq i \leq N$ and $O_i^g(k) = O_i^p(k), O_i^{g\_syn}(k) = O_i^{p\_syn}(k), 1 \leq i \leq N$. By Fact 1 and Fact 3, these outputs will arrive at their receiving machines before next state. In addition, by Assumption 1, Fact 2 and Fact 4, the environmental inputs to $M_i^g(k+1)$ and $M_i^p(k+1), 1 \leq i \leq N$ are the same.

Since machine M_i under a PALS clock and under a perfectly synchronized clock at state $k+1$ has identical states, identical environmental inputs, and identical inputs from state machines, by Assumption 3 machine M_i under a PALS clock and under a perfectly synchronized clock makes the identical state transitions and generates the identical outputs. That is, $M_i^g(k+2) = M_i^p(k+2), 1 \leq i \leq N$, and $O_i^g(k+1) = O_i^p(k+1), O_i^{g\_syn}(k+1) = O_i^{p\_syn}(k+1),\ 1 \leq i \leq N$. It follows that $M_i, 1 \leq i \leq N$, under a PALS clock C_i^p and under a perfectly synchronized clock C_i^g are logically equivalent at state $k+1$. By induction Theorem 1 follows.

Corollary 1.1. The bound on PALS clock period, $(2\epsilon + max(\alpha_{max} + q_{max}, H) + \mu_{max})$, is tight.

Proof. As illustrated in Figure 3, since the clock skew is bounded by ϵ, a sender M_1 can lead the receiver M_2 by at most 2ϵ. If a sender's output hold time is less than $H = 2\epsilon - \mu_{min}$ as defined by the PALS protocol, a message from sender at time $\uparrow (C_1^p = j)$ can reach the receiver before $\uparrow (C_2^p = j)$. Hence, H cannot be shortened.

As illustrated in Figure 4, sender M_1 may lag the receiver M_2 by 2ϵ. In this case, the latest instance at which the sender M_1 transmits a message at the network is $\uparrow (C_1^p = j) + max(\alpha_{max} + q_{max}, H)$. The longest network transmission delay is μ_{max}. In order to reach M_2 before $\uparrow (C_2^p = j+1)$, the PALS clock period T must satisfy the inequality, $T > 2\epsilon + max(\alpha_{max} + q_{max}, H) + \mu_{max}$. Corollary 1.1 follows.

Corollary 1.2. The PALS protocol uses minimal messages to achieve logical synchronization.

Proof. By the rules of the PALS protocol, PALS does not use any synchronization message at all. Corollary 1.2 follows.

We have shown that global computation under PALS clocks and under perfectly synchronized clocks are logically equivalent. Hence, we say that a networked embedded system under the PALS protocol is logically synchronous. However, logical synchrony is different from physical synchrony. When the clocks have bounded skews, the I/O's performed by distributed nodes have relative I/O jitters. In a system driven by perfectly synchronized clocks, each machine can perform input and output at exactly the same time. The PALS protocol should only be used in global computations, where relative I/O timing jitters are tolerable by the applications in question. If such jitters are not acceptable, then neither PALS nor any other form of networked real time systems can be used. This is because any method M that can reduce I/O jitters can also be used by a PALS based system, provided that method M does not violate PALS Causality rule (G6). Finally, if the I/O jitters occur in continuous control, for example, during the adjustment of setpoints of engine speeds and control surface angles, they translate to control errors. If such errors are modest, they can be compensated by feedback controls.

4. Example Application

Although a study of PALS application has been conducted [19], due to the page limitation, we will illustrate PALS using a greatly simplified example based on [11]. This dual redundant FGS example has two physical sides corresponding to the left and right sides of the aircraft. Each of the FGS is connected to two redundant real time networks. Each message will be sent on both networks. So the failure of a single network will not affect the operations of FGS. The FGS periodically compares the current state of the aircraft (speed, altitude and position) to the desired setpoints and then generates pitch and roll guidance commands. The FGS receives input about the current state of the aircraft from different subsystems, such as the Air Data System (ADS) and Flight Management System (FMS). The FGS supplies the pitch and roll setpoints to the autopilot (AP) and display them on the Primary Flight Display (PFD). The pilot interacts with the FGS through the Flight Control Panel (FCP) to perform discrete control, e.g., changing the modes. The FGS can operate in different modes: dependent and independent mode. In the dependent mode (hot standby mode) only one FGS remains primary (pilot flying side), while the other operates as a hot standby. In the independent mode, both can independently operate as the navigational source.

In the hot standby mode, two sides synchronize their computations with each others. To illustrate the application of PALS, we will focus on the dis-

crete global computation that performs the synchronization logic between these systems. During the operation of FGS, one side may fail and then restarted if it is recoverable. If the primary fails, then the standby will automatically become the primary and the former primary that has failed will restart as the standby. On the other hand, a pilot may issue command to switch the primary and standby. However, the pilot's "switch primary" command will be ignored if it would let the failed side as the primary.

Note that in the hot standby configuration, when the primary fails, it will take one step for standby to detect the failure and to become the primary in next step. From a control perspective, this means that supervisory control of setpoint adjustments could be missing for one period. While this can be compensated in control design, it is still undesirable. This is the reason why PALS provides the lower bound for supervisory controller periods.

We now give a concrete example to illustrate the PALS design vs GALS design under the following failure model.

1. **Assumption 5: Fail-stop.** Machines are fail-stop and may recover.
2. **Assumption 6: No Concurrent Failure.** The probability of concurrent failures of both machines is negligible, so is the probability of concurrent failures of both real time networks.

There are two design requirements.

1. **Unique Primary.** There can be only one primary at any time.
2. **Bounded No Primary Duration.** For each machine failure, there can be at most 1 period during which there is no primary.

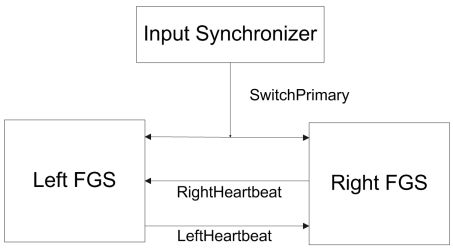

Figure 5: Dual FGS system

Figure 5 shows our simplified model of the dual FGS system. This model consists of three components: left FGS, right FGS and an environment event Input Synchronizer. In this example, the only environment event to these two FGS is the SwitchPrimary. The pilot can send a $SwitchPrimary = 1$ command to change the primary FGS. In the absence of any pilot command, the Input Synchronizer sends a value 0 (zero). We assume that the FGS can fail. We inject failure using two boolean inputs: FailLeft and FailRight. For example, if $FailLeft = 1$, then left FGS will fail in the next state. Additionally, we assume that both FGS cannot fail simultaneously.

In our model, both FGS exchange heartbeats for the failure detection. If $FailLeft = 1$, then at the next state $LeftHeartbeat = 0$. The heartbeat is sent after the FGS has successfully completed its computation and application I/O. We assume that a heartbeat sent in period j will reach the other side in period j and be read at $\uparrow (C = j + 1)$. In our design, the standby also sends heartbeat to the primary. This prevents the pilot switching the primary to a known failed side.

If a FGS is primary, then its status (LeftIsPrimary) is set to 1 (one). Thus, the system must satisfy the predicate that exactly one FGS is primary, which is represented by

$$(LeftIsPrimary = 1 \wedge RightIsPrimary = 0) \vee (LeftIsPrimary = 0 \wedge RightIsPrimary = 1)$$

Analysis of the System with Perfectly Synchronized Clocks

In this case, the distributed state machines of these three components operate at locksteps. Before we can prove whether the system satisfies the aforementioned two requirements for this ideal system, we must define how each FGS determines whether it is primary or standby. We use two state variables, (LeftIsAlive, LeftIsPrimary), to represent if left FGS is alive and whether left FGS is primary or standby, similarly for the right FGS. We initialize both sides alive, left FGS to be primary and right FGS to be standby. That is

$$LeftIsAlive = RightIsAlive = 1;$$
$$(LeftIsPrimary = 1) \wedge (RightIsPrimary = 0)$$

We specify the behaviors of the left FGS using a truth table given in Table 1. The right FGS has identical behaviors, except left FGS is initialized to be the primary and right is initialized as the standby. Note that a failed FGS cannot be the primary FGS immediately after recovering from failure (as shown in Row 2 of Table 1). Otherwise, the

	Input			Current state		Next state	
Row	FailLeft	SwitchPrimary	RightHeartbeat	LeftIsAlive	LeftIsPrimary	LeftIsAlive	LeftIsPrimary
1	1	D	D	D	D	0	0
2	0	D	D	0	D	1	0
3	0	D	0	1	D	1	1
4	0	0	1	1	0	1	0
5	0	0	1	1	1	1	1
6	0	1	1	1	0	1	1
7	0	1	1	1	1	1	0

Table 1: State transitions of left FGS. ("D" represents *don't care*)

unique primary requirement will be violated. For example, at state i right FGS is the primary. Left FGS is the standby and it fails. At state $i+1$, a SwitchPrimary command arrives. In addition, left FGS recovers in time and becomes active during state $i+1$. However, the heartbeat sent by left FGS will be read by right FGS at the beginning of state $i+2$. Hence, during state $i+1$ right FGS ignores SwitchPrimary command, because heartbeat sent by left FGS during state i was missing. If we let the left FGS to become primary, then we have two primaries. If pilot issue another SwitchPrimary command, then both become standby.

We have modeled the design specified by Table 1 using both NuSMV [17] and Maude [18]. We verified that this synchronous design meets both requirements. Both models can be found in `https://agora.cs.illinois.edu/download/attachments/9527/RTSS09_PALS.zip`. In a separate paper, we will give a detailed description on how to mechanically map a synchronous design specified in AADL onto networked system with bounded clocks skews and automatically check if PALS protocols are followed.

Compare to the synchronous design, the GALS solution is more complicated. To illustrate the added complexities, we apply our simple synchronous solution in the GALS environment directly, where clocks have bounded skews. Figure 6 shows a counter example where the design fails.

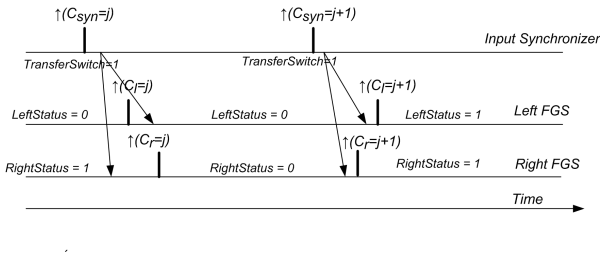

Figure 6: Counter example for the GALS solution

In this case, the Input Synchronizer delivers two SwitchPrimary event at consecutive periods of the Input Synchronizer as a result of the pilot's actions. While the right FGS processes these two events at separate periods, the left FGS receives both SwitchPrimary event at the same period. Forms of inconsistencies, which do not exist in synchronous designs but exist in asynchronous designs, add complexity. For example, as illustrated in Page 12 of [11], a design that works in a synchronous case would fail in the asynchronous case, resulting in each side permanently stuck in the primary state.

With additional hand shaking protocols, the problem was solved. However, "*the properties themselves are more difficult to state, were weaker than could be achieved in the synchronous case, and required considerable complexity to be added to the model to ensure that even the weakened properties were true.*" [11]. Another problem with the asynchronous design is the state explosion problems due to clock asynchrony. In a case study of an active standby system, it was founded that model checking the asynchronous model took over 30 hours, while model checking the PALS design took less than 30 seconds[1] [19]. These problems motivated the PALS research reported in this paper.

5. Summary and Conclusion

In networked cyber physical systems, real time global computations, such as supervisory control, require consistent views, actions and synchronized state transitions. In the aviation community, networked real time systems are known as Globally Asynchronous Locally Synchronous (GALS) systems, because the skews between local clocks can only be bounded but not eliminated. When interactions between nodes are driven by local clocks, the resulting interactions become asynchronous. Discrete supervisory controls, such as commands for

1 See Page 42.

mode changes, are very sensitive to asynchronous interactions. An aircraft has hundreds of computers network together, each with slightly different clock skews, and tens to hundreds of concurrent discrete commands in the hierarchical control of the flight control system.

To verify control logics in presence of asynchronous interactions, a model checker has to examine all possible state transitions of each machine at each clock tick under all possible clock skew combinations. This creates a combinatorial explosion of the global system state space. Furthermore, in asynchronous system designs the properties themselves are more difficult to state, weaker and required considerable complexity to be added to the model to ensure that even the weakened properties.

This paper presents a real time logical synchrony protocol, PALS, for distributed real time global computation. Under the PALS protocol, engineers design and verify applications as if all the distributed state machines were driven by a perfect global clock. The PALS protocol is optimal in the sense that 1) the bound on the periods of real time global computation is shortest possible, and 2) the message overhead in achieving logical synchrony is minimal.

References

[1] Birman, K. and Joseph, T. Exploiting virtual synchrony in distributed systems. SIGOPS Oper. Syst. Rev. 21, 5,Nov. 1987.

[2] Guo, K., Vogels, W., and van Renesse, R. Structured virtual synchrony: exploring the bounds of virtual synchronous group communication. 7th Workshop on ACM SIGOPS European Workshop: Systems Support For Worldwide Applications, Sept. 1996.

[3] Pereira, J., Rodrigues, L., and Oliveira, R. Reducing the Cost of Group Communication with Semantic View Synchrony. international Conference on Dependable Systems and Networks, June 2002.

[4] Delporte-Gallet, C. Fauconnier, C.D.I. Real-time fault-tolerant atomic broadcast. Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems,1999.

[5] Hooman, J. Verification of Distributed Real-Time and Fault-Tolerant Protocols. Proceedings of the 6th international Conference on Algebraic Methodology and Software Technology,Dec. 1997.

[6] Peter Robinson and Ulrich Schmid. The Asynchronous Bounded-Cycle Model. Proceedings of the 10th International Symposium on Stabilization, Safety, and Security of Distributed Systems, November 2008.

[7] Tarek Abdelzaher, Anees Shaikh, Farnam Jahanian, and Kang Shin. RTCast: Lightweight Multicast for Real-Time Process Groups. IEEE Real-Time Technology and Applications Symposium, June 1996.

[8] E. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[9] Ricardo Bedin Franca, Jean-Paul Bodeveix, David Chemouil, Mamoun Filali, Dave Thomas, and Jean-Francois Rolland. The AADL behaviour annex experiments and roadmap. *UML&AADL'2007*, 2007.

[10] Nicolas Halbwachs and Louis Mandel. Simulation and verification of asynchronous systems by means of a synchronous model. In *Proceedings of the 6th International Conference on Application of Concurrency to System Design*, 2006.

[11] Steven P. Miller, Mike W. Whalen, Dan OBrien, Mats P.E. Heimdahl, and Anjali Joshi. A methodology for the design and verification of globally asynchronous/locally synchronous architectures. NASA Contractor Report NASA/CR-2005-213912.

[12] Society of Automotive Engineers. SAE standards: Architecture analysis & design language (AADL). *AS5506*, November 2004.

[13] P. C. Ölveczky and J. Meseguer. Abstraction and completeness for Real-Time Maude. *Electronic Notes in Theoretical Computer Science*, 176(4):5–27, 2007.

[14] P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.

[15] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2nd edition, 2000.

[16] Software considerations in airborne systems and equipment certification, DO-178B. *RTCA Inc: Washington, DC*, December 1992.

[17] http://nusmv.irst.itc.it

[18] http://maude.cs.uiuc.edu

[19] Steven P. Miller and Darren Cofer. *PALS for Asynchrous Design*. Technical Presentation, Rockwell Collins Inc., July 15, 2008.