CPS Runtime Architecture And Automated Transformation of Applications

Lui Sha
Department of Computer Science
University of Illinois
Urbana IL., USA
lrs@illinois.edu

ABSTRACT

National Academy of Science's study on dependable software systems concluded that simplicity is the key. Reducing the complexity of software has been investigated by the FEAST community on automated transformation of application software and by the CPS runtime community on the development of runtime architectures that simply the development of applications. This creates the need of collaboration between these two communities.

The goal of this review paper is to bring this need to the attention of FEAST community.

CCS CONCEPTS

Software system structures→ Software Architecture;

KEYWORDS

CPS runtime architecture, automated software transformation, complexity reduction

1 INTRODUCTION

"*One key to achieving dependability at reasonable cost is a serious and sustained commitment to simplicity, including simplicity of critical functions and simplicity in system interactions. This commitment is often the mark of true expertise. There is no alternative to simplicity. Advances in technology or development methods will not make simplicity redundant; on the contrary, they will give it greater leverage.*"

Software for Dependable Systems: Sufficient Evidence? Committee on Certifiably Dependable Software, National Academy of Science.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
FEAST'17, November 3, 2017, Dallas, TX, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5395-3/17/11…$15.00
https://doi.org/10.1145/3141235.3141238

Reducing the complexity of software has been investigated by different computing communities in parallel. The focus of FEAST has been the automated transformation of application software to make it more efficient. Many assume that runtime architecture remains unchanged. However, runtime architecture itself evolves. The collaboration between these two communities is important.

The goal of this paper is to bring this need of collaboration to the attention of FEAST community, using the example of Physically Asynchronous Logically Synchronous (PALS) architecture developed for networked control systems [3][6][9][11].

2 Physically Asynchronous Logically Synchronous Architecture

2.1 Background

The development of runtime architecture abstractions to simplify application development has played an important role in the advancement of computing systems. For example, the virtual machine abstraction allows engineers to assume they were the only users of the machine and atomic transactions abstraction allows engineers to assume that distributed transactions were executed one at a time.

PALS is a CPS runtime architecture designed to simplify the development of networked control systems such as avionics. Traditionally, to ensure that replicated subsystems are running in lock-steps, custom logics are added to network hardware to synchronize the executions of distributed nodes, e.g., Boeing 777's SafeBus [1] or TTEthernet [2]. Custom network hardware solutions are costly and place severely restrictions on possible network topologies. The PALS [3] approach is a software alternative to custom networks. PALS can use any network technology, provided that it meet the real time and fault tolerance requirements of avionic systems. This reduces cost and increases the flexibility in the development of modern avionics. The correctness of PALS was formally proved at the protocol level by model checking [9] and by theorem prover [11] .

As is, networked control system is a globally asynchronous locally synchronous system, because the skews between distributed clocks can be bounded but not eliminated. When applications on each node are driven by their local clocks,

asynchronous interactions arise. Managing asynchronous interactions is a complex task. Rockwell Collins Inc. implemented a dual redundant flight control system in the lab with and without PALS [6]. The model checking time was 35 hours when logic for asynchronous interactions is embedded in the applications. Running on top of PALS middleware, the model checking time of the applications was dropped to less than 30 seconds [6].

PALS was awarded by American Institute of Aeronautics and Astronautics the David Lubkowski memorial award for the Advancement of Digital Avionics in 2009. PALS middleware, PALSWare, consists about 2000 lines of C code and the source code was formally verified in [14].

PALS is an effective alternative to the custom hardware approach because it allows avionics developers to choice from a larger variety of network technologies. On top of PALSWare, engineers can write application code as if it were running on a perfectly synchronized computer at the fastest rate that can be guaranteed by the network. However, the challenge of transitioning any new runtime architectures into practice is the cost of transforming the legacy applications for the new runtime time system. We wonder if automatic translation of applications designed for legacy runtime system to new runtime systems is possible. In particular, PALSWare APIs are very simple.

2.2 PALSWare

As mentioned before, networked control system is known as globally asynchronous locally synchronous (GALS). Distributed race condition could arise if it is used as is. For example, a command was sent to replica 1 and replica 2. Because of the clock skews, Replica 1 receives the command in period 10 but Replica 2 receives the command in period 9. This leads divergence of actions in the two replicas.

Figure 1: Distributed Race Condition

Figure 2 illustrates that under the PALS protocol the messages exchanges can only occur during the green colored zones and the system will behavior as if it were regulated by a perfect global clock. Using PALSWare, distributed applications can use synchronous design on top of physically asynchronous systems, incurring exponential reduction in verification state space. PALS protocol has minimal communication overhead because no synchronization messages are needed.

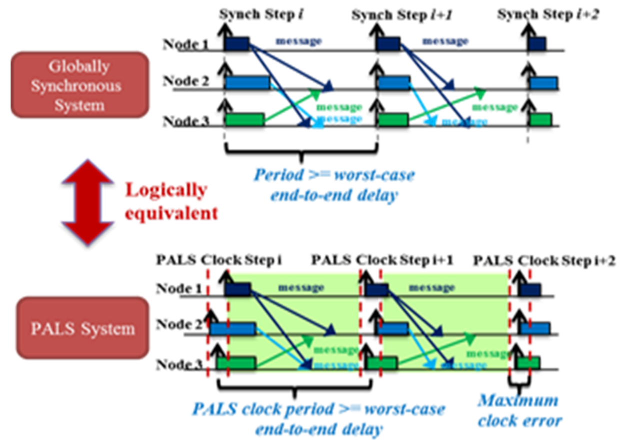

Figure 2: Logical Synchrony

Figure 3 illustrates the role of PALSWare in a networked control system. The PALS architecture pattern defines the following constraints to be satisfied [14]:

1. Distributed applications at each node are triggered by the rising edge of its local PALS clock with period T.

2. PALS Clock period: Distributed computations that require consistent views and actions cannot be achieved faster than the end to end communication delay.

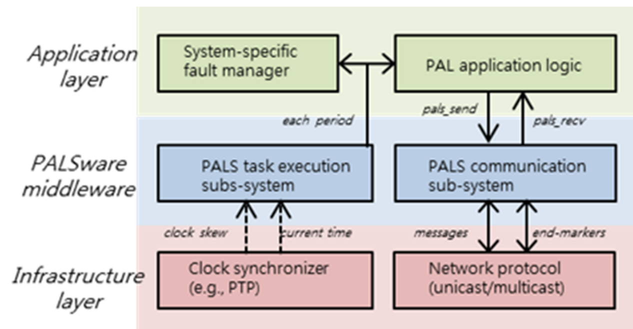

Figure 3: System View

Therefore, the PALS Clock Period T must satisfy the following inequality:

$$T \geq \mu_{max} + 2\mathcal{E} + \max(\alpha_{max}, 2\mathcal{E} - \mu_{min})$$

3. PALS Causality Constraint: Because of the max clock skew ($2\mathcal{E}$) between the sender PALS client and the receiver, when the sender is in its local PALS period i, the receiver may get the message in its local PALS period ($i - 1$) if the network delay is shorter than $2\mathcal{E}$. PALSWare buffers such message and ensure that when a message is sent at sender's local PALS period i, it will physically be received by receiver at its local PALS period i and be ready to be used in receiver's next period ($i + 1$).

PALSWare works under the following 5 conditions:

1. Bounded Clock Error: Each node j has access to an approximation of the true global time t via a local clock c_j, where the maximum error of each local clock is ε, i.e., $|c_j - t| < \varepsilon$. In order words, the difference between any pair of clocks is bounded by 2 ε. This is done by clock synchronization software.

2. Monotonic Local Clocks: the value of each local clock, increases monotonically. This is enforced by real time operating systems.

3. Bounded Computation Time: the computation of a node's new local state and outputs takes time α, where $\alpha_{mi} \leq \alpha \leq \alpha_{max}$. This is done by real-time software design and real time scheduling at each node.

4. Bounded Message Delivery: messages are reliably delivered to their destinations in time μ, where $\mu_{min} \leq \mu \leq \mu_{max}$. This is enforced by the real time network.

5. Fail Silent Nodes: Nodes stop silently upon failure, i.e. nodes do not behave maliciously or randomly on failure. This is enforced by the fault tolerant design needed for networked control systems such as avionics. PALSWare provides two main services for the applications.

1. Application activation service: Implement PALS clocks uniformly at each node.

2. PALS messaging service: When message is sent in step i, it will only be available for read in step $i+1$.

PALSWare main API functions are: *open_tx_port, open_rx_port, timer_create, wait_schedule, send, and recv.*

2.3 Dynamic Measurements: BLS

The active standby application represents a typical distributed fault-tolerant redundant system. There are two physically separate systems (Side 1 and Side 2) which communicate to maintain an active- standby paired state at all times. The 5 essential design requirements are [3]:

1. Both controllers should agree on which controller is active.

2. A controller that is not fully available should not be the active controller if the other controller is fully available.

3. If a controller is failed the other controller should become active.

4. The user can always change the active controller.

5. In other cases, the active controller should not change, unless its availability changes, or the user requests.

Requirements 1-3 are necessary to guarantee both fault-tolerance and the consistent agreement between two controllers. Requirement 4 allows the user to designate the active controller. Requirement 5 is to prevent unnecessary fluctuation of the status.

Figure 4 illustrate a simple active standby system using PALSWare. The use of PALSWare consists of the following steps:

1. Create sending or receiving ports for networking and start a timer with the given PALS period. ⟶ Client Activation Service

2. Start an infinite loop for periodic execution.

3. Wait for timer to expire at next PALS period. ⟶ Client Activation Service

4. Read any incoming message from its receiving ports. ⟶ PALS Messaging Service

5. Perform computation.

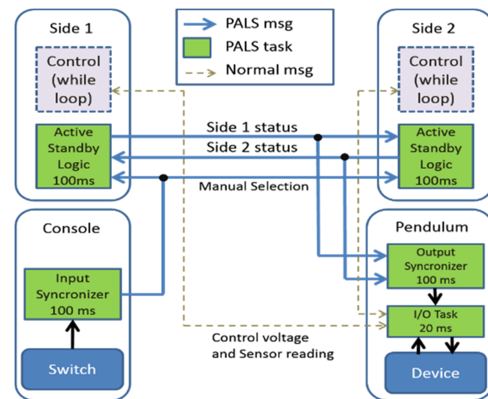

Figure 4: An Example Application of PALSWare

From an application development perspective, PALSWare has the following benefits.

1. Real-time virtual synchrony: Using PALSware, distributed applications can use globally synchronous design on top of typical physically asynchronous systems, incurring exponential reduction in verification state space.

2. Optimal performance and efficiency: PALS protocol has minimal communication overhead.

3. Middleware and design tool support: PALSWare and AADL design tools enable rapid prototyping and design verification of real-time distributed applications [3].

However, a challenge faced by the CPS runtime system community has been the cost of porting legacy application software for an old runtime system to a new runtime system.

We hope that the FEAST community and the CPS runtime architecture community can work together to further the shared goal of developing simpler and more efficient CPS systems.

4 CONCLUSIONS

As noted by National Academy of Science's study on dependable software systems, simplicity is the key.

Reducing the complexity of software has been investigated by different computing communities in parallel. The focus of FEAST has been the automated transformation of application software. The focus of runtime architecture focuses on how to make the application development simpler.

This creates the need of collaboration between these two communities. The goal of this paper is to bring this need to the attention of FEAST community, using the example of Physically Asynchronous Logically Synchronous (PALS) architecture developed for networked control systems. .

ACKNOWLEDGMENTS

This work was partially supported by ONRN00014-17-1-2783. I want to thank Dr. Min-Yong Nam's assistance in the preparation of this review paper.

REFERENCES

[1] Hoyme, Ken, Kevin Driscoll, 1992, SAFEbus, Proc. IEEE/AIAA Digital Avionics Systems Conference (DASC'92), pp. 68-73.

[2] Kopetz, Hermann, G. Bauer, The time-triggered architecture, 2003, Proceedings of the IEEE, Vol. 91, Issue 1, pp. 112-126.

[3] Al-Nayeem, Abdullha, Mu Sun, Xiaokang Qiu, Lui Sha, Steven P. Miller, Darren D. Cofer, 2009, A Formal Architecture Pattern for Real-Time Distributed Systems, IEEE 30th Real-Time Systems Symposium (RTSS), pp. 161-170

[4] Jhala, Ranjit, Rupak Majumdar: Software model checking. ACM Comput. Surv. 41(4) (2009)

[5] Clarke, Edmund, Daniel Kroening, Flavio Lerda: A Tool for Checking ANSI-C Programs. TACAS 2004: 168-176

[6] Miller, Steven P., Darren D. Cofer, Lui Sha, Josė Meseguer, Abdullah Al-Nayeem, 2009, Implementing Logical Synchrony in Integrated Modular Avionics, IEEE/AIAA 28th Proc. of Digital Avionics Systems Conference (DASC), pp. 1.A.3-1 - 1.A.3-12.

[7] Chaki, Sagar, Arie Gurfinkel, Soonho Kong, Ofer Strichman, 2013, Compositional Sequentialization of Periodic Programs, Proc. of VMCAI, Springer.

[8] Feiler, Peter H., David P. Gluch, 2012, Model- Based Engineering with AADL – An Introduction to the SAE Architecture Analysis and Design Language, Addison-Wesley.

[9] Meseguer, Josė, Peter C. Ölveczky, 2010, Formalization and Correctness of the PALS Architectural Pattern for Distributed Real-Time Systems, Proceedings of the 12th International Conference on Formal Engineering Methods and Software Engineering, Pp. 303-320.

[10] Ölveczky, Peter C., Josė Meseguer, 2007, Semantics and Pragmatics of Real-Time Maude, Journal of Higher-Order and Symbolic Computation, Vlusm 20, Issue 1-2, pp. 161-196.

[11] Steiner, Wilfried, John M. Rushby, 2011, TTA and PALS: Formally Verified Design Patterns for Distributed Cyber-Physical Systems, IEEE/AIAA 30th Proc. of Digital Avionics Systems Conference (DASC) pp.7B5-1 - 7B5-15.

[12] Rushby, John M., 1999, Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms, Transactions of Software Engineering, Vol 25, Issue 5, pp. 651-660.

[13] Bae, Kyungmin, Joshua Krisiloff, Peter C. Ölveczky, Josė Meseguer, 2012, PALS-Based Analysis of an Airplane Multirate Control System in Real-Time Maude, International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS).

[14] Min-Young Nam, Lui Sha, Sagar Chaki, and Cheolgi Kim, Applying Software Model Checking to PALS Systems, IEEE/AIAA 33rd Digital Avionics Systems Conference, 2014.