In Proceedings of 2006 IEEE 1588 Conference, October 2-4, 2006, Gaithersburg, MD

# Application of Programming Temporally Integrated Distributed Embedded Systems

Yang Zhao*, Edward A. Lee*, Jie Liu†,
*EECS Department, University of California at Berkeley,
Berkeley, CA 94720 USA, Email: {ellen_zh, eal}@eecs.berkeley.edu
†Microsoft Research, Redmond, WA 98052, USA. Email: liuj@microsoft.com

**Abstract**

The introduction of network time protocols such as NTP (at a coarse granularity) and IEEE 1588 (at a fine granularity) gives a relatively consistent global notion of time that has the potential to significantly change how we design distributed real-time systems. In [4], we present a programming model called PTIDES (Programming Temporally Integrated Distributed Embedded Systems) that uses discrete-event (DE) models as programming specifications for distributed real-time systems and describe an execution model that permits out of order processing of events without sacrificing determinacy and without requiring backtracking. In this paper, we present an interesting networked camera application programmed using PTIDES and show how the execution model in [4] can be used to meet real-time constrains in the system.

## I. Introduction

The introduction of network time protocols such as NTP (at a coarse granularity) and IEEE 1588 (at a fine granularity) gives a relatively consistent global notion of time that has the potential to significantly change how we design distributed real-time systems. Time synchronization over standard networks, such as provided by NTP [3], can achieve timing precision within ten milliseconds, which is sufficient for many interactive distributed systems, such as computer games, where human-scale time precision is adequate. The recent standardization (IEEE 1588[1]) of high-precision timing synchronization over ethernet provides much higher timing precision that is essential for many embedded systems. Implementations of IEEE 1588 have demonstrated time synchronization as precise as tens of nanoseconds over networks that cover hundreds of meters, more than adequate for many manufacturing, instrumentation, and vehicular control systems.

In [4], we present a programming model called PTIDES (Programming Temporally Integrated Distributed Embedded Systems) that leverages time synchronization over distributed platforms. PTIDES uses discrete-event models as programming specifications for distributed real-time systems and extends discrete-event models with the capability of mapping certain events to physical time. We use model time to define execution semantics and add constraints that bind certain model time events to physical time. We limit the relationship of model time to physical time to only those circumstances where this relationship is needed. A correct execution will simply obey the ordering constraints implied by model time and meet the constraints on events that are bound to physical time. We then seek execution strategies that can preserve the deterministic behaviors specified in DE models and also provide efficient real-time executions without paying the penalty of totally ordered executions. The key idea is that events only need to be processed in time-stamp order when they are causally related. An execution model that permits out of order processing of events without sacrificing determinacy and without requiring backtracking is described in [4]. The formal foundation is based on the concepts of relevant dependency and relevant order. The results are particularly valuable in time-synchronized distributed systems, since we can take advantage of the globally consistent notion of time as a coordination channel. Based on relevant orders, we can statically analyze

[1] http://ieee1588.nist.gov

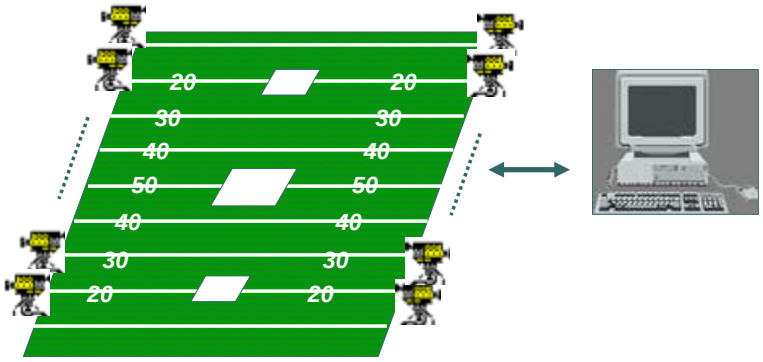


Fig. 1. Networked camera application.

whether a given model is deployable on a network of nodes, assuming that we have upper bounds on the network delays.

In this paper, we describe an application of networked cameras and discuss the implementation and execution of the application as a PTIDES model. This paper is organized as follows. Section II introduces the application. Section III shows the implementation of the application. In section IV, we discuss the challenges in executing the system and show how the execution strategy based on the relevant order [4] can be used to execute the system efficiently to meet the real-time constraints.

## II. Application

Consider that we have $N$ cameras distributed over a football field as shown in figure 1. All the cameras have computer-controlled picture and zoom capabilities. Each camera only has a partial view of the field. The images produced by each camera are transfered over the network to the central computer, where the images get processed to produce an entire view of the field or a sequence of views for some interesting moment. On the other hand, a user sitting in front of the central computer may issue commands to the cameras to zoom or change the frequency of taking images. For example, when there is a touchdown, the user may want to control a set of cameras to zoom in or take pictures at a higher frequency.

Suppose that the clocks at all the cameras and the central computer are precisely synchronized, and each camera can generate precisely timed physical events under the control of software, i.e. we can control a camera to take a picture or zoom precisely at some physical time. Zooming takes time $\kappa$ to set up, and during this period we do not want the camera to take fuzzy pictures. Given that the commands controlling the cameras to zoom or change frequency are transmitted over the network with some delay bounded, the challenges here are how to make sure that all the cameras adjust at the same time and how to coordinate the zoom action and the taking picture action properly on each camera. We discuss our design for this application in the next section.

## III. Implementation

Figure 2 shows the implementation of the networked camera application as a PTIDES program. A PIDES program is given as a composition of actors, by which we mean a set of actors and connectors linking their ports. The dashed boxes divide the model into two parts, the top one to be executed on each camera and the bottom one to be executed on the central computer. The parts communicate via signal $s_1$ and $s_2$. We assume that events in these signals are sent over the network as time-stamped values.

The Command actor is a software component that wraps interactions with the user input device. When a user input comes in, the Command actor checks with its synchronized clock for the current time, uses the returned time value to time stamp the input message and sends the time stamped message, called an *event*, to all the cameras. The right part of the model on the central computer processes the images taken at each camera and displays the result.

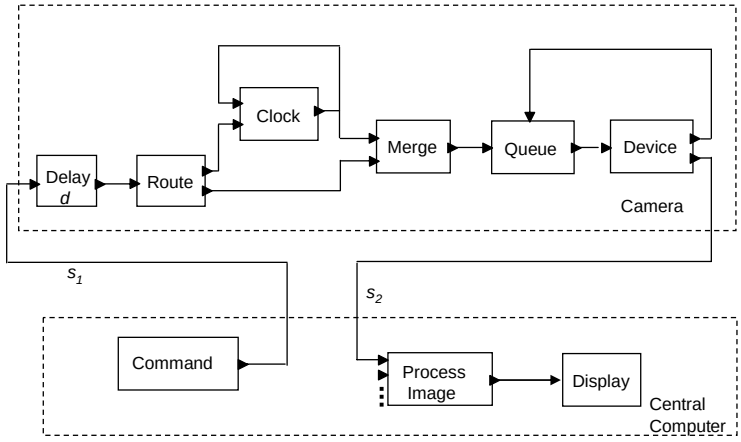


Fig. 2. Specification of the Networked camera application.

The Device actor in the top part of the model shown in figure 2 wraps interactions with the camera driver. At its input port, it receives a potentially infinite sequence of time-stamped values, or *events*, in chronological order. This actor takes the time stamp of an input event as specification of when an action, determined by the value of the event, happens at physical time. The first output port produces a time-stamped value for each input event, where the time stamp is strictly greater than that of the input event. The second output port produces the time-stamped image and sends it to the central computer.

The Queue actor buffers its input event until an event is received at the trigger port, which is the one at the top of the actor. We assume there is an initial event on the trigger port at the beginning. The feedback loop around the Queue and Device actor ensures that the Device does not get overwhelmed with future requests. It may not be able to buffer those requests, or it may have a finite buffer.

The Clock actor produces time-stamped outputs where the time stamp is some integer multiple of a period $p$. The time stamps are used to control when the camera takes pictures. The period can be different for each clock and can be changed during run time upon receiving an input on the second input port. If there is an event with value $v$ and time stamp $t$ at the second input, the clock actor will change its period from $p$ to $p' = p * v$ and produce an output with time stamp $t_0 + np'$ where $t_0$ is the time stamp of the last output and $n$ is the smallest integer so that $t_0 + np' \geq t$. The feedback loop around the clock actor is used to trigger the next output, and we assume there is an initial event on the first input at the beginning.

The Delay actor with a delay parameter $d$ will produce an event with time stamp $t + d$ at its output given an event with time stamp $t$ at its input.

Two kinds of user commands are received on each camera, the change frequency command and the adjust zoom command. The Router actor separates these events and sends the change frequency events to the Clock actor and the adjust zoom events to the Merge actor.

The Merge actor merges the events on the two input ports in chronological order. It gives priority to the second input port if input events have identical time stamps. That is, we give higher priority to the user to control a camera.

In the above discussion, the time stamps are values of model time. Some actors in the model bind model time to physical time. The Command actor binds model time to physical time by producing an event with model time corresponding to the physical time when the user input happens. The Device actor binds model time to physical time by producing some physical action at the real-time corresponding to the model time of each input event. The Device actor also impose real-time constraints to the model. The input events must be made available for the Device actor to process them at a physical time strictly earlier than the time stamp. Otherwise, the component would not be able to produce the physical action at the designated time. We limit the relationship of model time to physical time to only those circumstances where this relationship is needed. For other actors in the model, there is no real-time constraints and

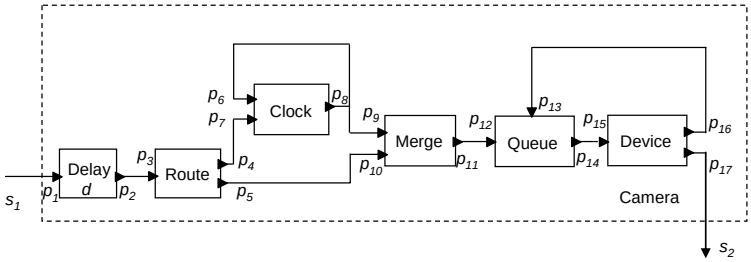


Fig. 3. The program on the camera.

model time is used to define execution semantics.

## IV. Execution

How to build a run-time environment to execute the distributed model shown in figure 2 to deliver the correct behavior and meet the real-time constrains in the system is a challenging problem. A brute-force implementation of a conservative distributed DE execution of this model would stall execution in a camera at some time stamp $t$ until an event with time stamp $t$ or larger has been seen on signal $s_1$. Were we to use the Chandy and Misra approach [1], we would insert null events into $s_1$ to minimize the real-time delay of these stalls. However, we have real-time constraints at the Device actors that will not be met if we use this brute-force technique. The so-called "optimistic" techniques for distributed DE execution will also not work in our context. Optimistic approaches perform speculative execution and backtrack if and when the speculation was incorrect [2]. Since we have physical interactions in the system, backtracking is not possible.

An execution model that permits out of order processing of events without sacrificing determinacy and without requiring backtracking is described in [4]. Here, we only give the result after applying dependency analysis to show which events in the system can be processed in a different order than their time stamp order and why this can help to meet real-time constants. For details about dependency analysis, please refer to [4].

Figure 3 shows the program running on each camera and names the ports. The dependencies between the input and output ports of each actor are listed below:

$$\begin{aligned} &\delta(p_1, p_2) = d, \quad \delta(p_3, p_4) = 0, \quad \delta(p_3, p_5) = 0 \\ &\delta(p_6, p_8) = P_{min}, \quad \delta(p_7, p_8) = 0, \quad \delta(p_9, p_{11}) = 0, \quad \delta(p_{10}, p_{11}) = 0 \\ &\delta(p_{12}, p_{14}) = 0, \quad \delta(p_{13}, p_{14}) = 0, \quad \delta(p_{15}, p_{16}) = \alpha, \quad \delta(p_{15}, p_{17}) = 0 \end{aligned} \tag{1}$$

where $P_{min}$ is the minimum time interval between two consecutive taking picture action on the camera and $\alpha > 0$.

Based on the dependencies specified for each actor, we can calculate the relevant dependencies between any pair of input ports. As an example, the relevant dependency $d(p_1, p_9)$ is $d$, which means any event with time stamp $t$ at port $p_9$ can be processed when all events at port $p_1$ are known up to time stamp $t - d$. Assume the network delay is bounded by $D$, at physical time $t - d + D$ we are sure that we have seen all events with time stamps smaller than $t - d$ at $p_1$. Hence, an event $e$ at $p_9$ with time stamp $t$ can be processed at physical time $t - d + D$ or later. Note that although the Delay actor has no real-time properties at all (it simply manipulates model time), its presence loosens the constraints on the execution. By choosing $d$ properly, i.e. $d > D$, we can deliver $e$ to $p_{15}$ before physical time reaches $t$ and thus satisfy the real-time constraint on $p_{15}$.

What we gain from the dependency analysis is that we can specify which events can be processed out of order, and which events have to be processed in order. Please refer to [4] to see how this information can be used to define a partial order, called the *relevant order*, on events and how to design execution strategies based on the relevant order.

## V. CONCLUSION

We describe the use of DE models as programming specifications for time-synchronized distributed systems. A networked camera application and its implementation as a DE model are studied. The challenges in executing the specification over a distributed platform are discussed, and we then show how an execution model that permits out of order processing of events without sacrificing determinacy and without requiring backtracking can be used to improve the executability (i.e., to meet real-time constrains) of the application.

## VI. ACKNOWLEDGMENTS

This paper describes work that is part of the Ptolemy project, which is supported by the National Science Foundation (NSF award number CCR-00225610), and Chess (the Center for Hybrid and Embedded Software Systems), which receives support from NSF, the State of California Micro Program, and the following companies: Agilent, DGIST, General Motors, Hewlett Packard, Microsoft, National Instruments, and Toyota.

## REFERENCES

[1] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. on Software Engineering*, 5(5), 1979.

[2] D. Jefferson. Virtual time. *ACM Trans. Programming Languages and Systems*, 7(3):404–425, 1985.

[3] D. Mills. A brief history of ntp time: confessions of an internet timekeeper. *ACM Computer Communications Review 33*, April 2003.

[4] Y. Zhao, E. A. Lee, and J. Liu. Programming temporally integrated distributed embedded systems. Technical Report UCB/EECS-2006-82, EECS Department, University of California, Berkeley, May 28 2006.