Real-Time Coordination in Distributed Multimedia Systems

Theophilos A. Limniotes and George A. Papadopoulos

Department of Computer Science
University of Cyprus
75 Kallipoleos Str, P.O.B. 20537
CY-1678 Nicosia
Cyprus
E-mail: {theo,george}@cs.ucy.ac.cy

Abstract. The coordination paradigm has been used extensively as a mechanism for software composition and integration. However, little work has been done for the cases where the software components involved have real-time requirements. The paper presents an extension to a state-of-the-art control- or event-driven coordination language with real-time capabilities. It then shows the capability of the proposed model in modelling distributed multimedia environments

1 Introduction

The concept of coordinating a number of activities, possibly created independently from each other, such that they can run concurrently in a parallel and/or distributed fashion has received wide attention and a number of coordination models and associated languages ([4]) have been developed for many application areas such as high-performance computing or distributed systems.

Nevertheless, most of the proposed coordination frameworks are suited for environments where the sub-components comprising an application are conventional ones in the sense that they do not adhere to any real-time constraints. Those few that are addressing this issue of real-time coordination either rely on the ability of the underlying architecture apparatus to provide real-time support ([3]) and/or are confined to using a specific real-time language ([5]).

In this paper we address the issue of real-time coordination but with a number of self imposed constraints, which we feel, if satisfied, will render the proposed model suitable for a wide variety of applications. These constraints are:

- The coordination model should not rely on any specific architecture configuration supporting real-time response.
- The real-time capabilities of the coordination framework should be able to be met in a variety of systems including distributed ones.
- Language interoperability should not be sacrificed and the real-time framework should not be based on the use of specific language formalisms.

J. Rolim et al. (Eds.): IPDPS 2000 Workshops, LNCS 1800, pp. 685-691, 2000.

We attempt to meet the above-mentioned targets by extending a state-of-the-art coordination language with real-time capabilities. In particular, we concentrate on the so-called control- or event-driven coordination languages ([4]) which we feel they are particularly suited for this purpose, and more to the point the language Manifold ([1]). We show that it is quite natural to extend such a language with primitives enforcing real-time coordination and we apply the proposed model to the area of distributed multimedia systems.

2 The Coordination Language Manifold

Manifold ([1]) is a control- or event-driven coordination language, and is a realisation of a rather recent type of coordination models, namely the Ideal Worker Ideal Manager (IWIM) one. In Manifold there exist two different types of processes: *managers* (or *coordinators*) and *workers*. A manager is responsible for setting up and taking care of the communication needs of the group of worker processes it controls (non-exclusively). A worker on the other hand is completely unaware of who (if anyone) needs the results it computes or from where it itself receives the data to process. Manifold possess the following characteristics:

• *Processes*. A process is a *black box* with well-defined *ports* of connection through which it exchanges *units* of information with the rest of the world.

• *Ports*. These are named openings in the boundary walls of a process through which units of information are exchanged using standard I/O type primitives analogous to read and write. Without loss of generality, we assume that each port is used for the exchange of information in only one direction: either into (*input* port) or out of (*output* port) a process. We use the notation `p.i` to refer to the port `i` of a process instance `p`.

• *Streams*. These are the means by which interconnections between the ports of processes are realised. A stream connects a (port of a) producer (process) to a (port of a) consumer (process). We write `p.o -> q.i` to denote a stream connecting the port `o` of a producer process `p` to the port `i` of a consumer process `q`.

• *Events*. Independent of streams, there is also an event mechanism for information exchange. Events are broadcast by their sources in the environment, yielding *event occurrences*. In principle, any process in the environment can pick up a broadcast event; in practice though, usually only a subset of the potential receivers is interested in an event occurrence. We say that these processes are *tuned in* to the sources of the events they receive. We write `e.p` to refer to the event `e` raised by a source `p`.

Activity in a Manifold configuration is *event driven*. A coordinator process waits to observe an occurrence of some specific event (usually raised by a worker process it coordinates) which triggers it to enter a certain *state* and perform some actions. These actions typically consist of setting up or breaking off connections of ports and streams. It then remains in that state until it observes the occurrence of some other event, which causes the *preemption* of the current state in favour of a new one corresponding to that event. Once an event has been raised, its source generally continues with its activities, while the event occurrence propagates through the

environment independently and is observed (if at all) by the other processes according to each observer's own sense of priorities.

More information on Manifold can be found in [1]; the language has already been implemented on top of PVM and has been successfully ported to a number of platforms including Sun, Silicon Graphics, Linux, and IBM AIX, SP1 and SP2.

3 Extending Manifold with a Real-Time Event Manager

The IWIM coordination model and its associated language Manifold have some inherent characteristics, which are particularly suited to the modelling of real-time software systems. Probably the most important of these is the fact that the coordination formalism has no concern about the *nature* of the data being transmitted between input and output ports since they play no role at all in setting up coordination patterns. More to the point, a stream connection between a pair of input-output ports, simply passes anything that flows within it from the output to the input port. Furthermore, the processes involved in some coordination or cooperation scenario are treated by the coordination formalism (and in return treat each other) as black boxes without any concern being raised as to their very nature or what exactly they do. Thus, for all practical purposes, some of those black boxes may well be devices (rather than software modules) and the information being sent or received by their output and input ports respectively may well be signals (rather than ordinary data). Note also that the notion of stream connections as a communication metaphor, captures both the case of transmitting discrete signals (from some device) but also continuous signals (from, say, a media player). Thus, IWIM and Manifold are ideal starting points for developing a real-time coordination framework.

In fact, a natural way to enhance the model with real-time capabilities is by extending its event manager. More to the point, we enhance the event manager with the ability to express real-time constraints associated with the raising of events but also reacting in bound time to observing them. Thus, while in the ordinary Manifold system the raising of some event `e` by a process `p` and its subsequent observation by some other process `q` are done completely asynchronously, in our extended framework timing constraints can be imposed regarding when `p` will raise `e` but also when `q` should react to observing it. Effectively, an event is not any more a pair `<e,p>`, but a triple `<e,p,t>` where `t` denotes the moment in time at which the event occurs. With events that can be raised and detected respecting timing constraints, we essentially have a real-time coordination framework, since we can now guarantee that changes in the configuration of some system's infrastructure will be done in bounded time. Thus, our real-time Manifold system goes beyond ordinary coordination to providing temporal synchronization.

3.1 Recording Time

A number of primitives exist for capturing the notion of time, either relative to world time, the occurrence of some event, etc. during the execution of a multimedia

application which we refer to below as presentation. These primitives have been implemented as atomic (i.e. not Manifold) processes in C and Unix. In particular:

• **`AP_CurrTime(int timemode)`**
returns the current time according to the parameter timemode. It could be world time or relative.

• **`AP_OccTime(AP_Event anevent, int timemode)`**
returns the time point (in world or relative mode) of an event. Time points represent single instance in time; two time points form a basic interval of time.

• **`AP_PutEventTimeAssociation(AP_Event anevent)`**
creates a record for every event that is to be used in the presentation and inserts it in the events table mentioned above.

• **`AP_PutEventTimeAssociation_W(AP_Event anevent)`**
is a similar primitive which additionally marks the world time when a presentation starts, so that the rest of the events can relate their time points to it.

3.2 Expressing Temporal Relationships

There are two primitives for expressing temporal constraints among events raised and/or observed. The first is used to specify when an event must be triggered while the second is used to specify when the triggering of an event must be delayed for some time period.

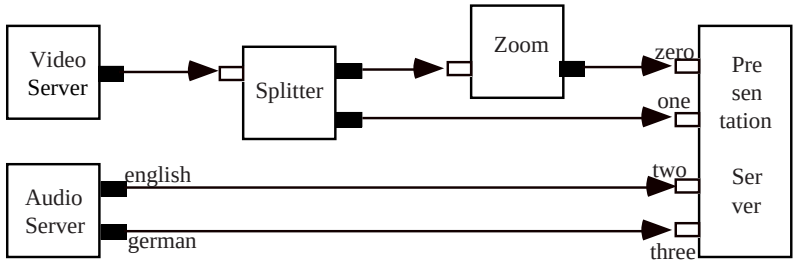
• **`AP_Cause(AP_Event anevent, AP_Event another, AP_Port delay, AP_Port timemode)`**
enables the triggering of the event `another` based on the time point of `anevent`.

• **`AP_Defer(AP_Event eventa, AP_Event eventb, AP_Event eventc, AP_Port delay)`**
inhibits the triggering of the event `eventc` for the time interval specified by the events `eventa` and `eventb`. This inhibition of `eventc` may be delayed for a period of time specified by the parameter `delay`.

4 Coordination of RT Components in a Multimedia Presentation

We show the applicability of our proposed model by modelling an interactive multimedia example with video, sound, and music. A video accompanied by some music is played at the beginning. Then, three successive slides appear with a question. For every slide, if the answer given by the user is correct the next slide appears; otherwise the part of the presentation that contains the correct answer is re-played before the next question is asked. There are two sound streams, one for English and another one for German.

For each such medium, there exists a separate manifold process. Each such manifold process is a “building block”. The coordination set up with the stream connections between the involved processes is shown below (the functionality of some of these boxes is explained later on):



We now show in more detail some of the most important components of our set up. We start with the manifold that coordinates the execution of atomics that take a video from the media object server and transfer it to a presentation server.

```
manifold tv1()
{

begin:(activate(cause1,cause2,mosvideo,splitter,zoom),c
ause1,WAIT).
 start_tv1:(cause2,mosvideo -> ( -> splitter),
            splitter.zoom ->zoom,
            zoom-> (->ps.zero),ps.out1->stdout,WAIT).
 end_tv1:post(end).
 end:(activate(ts1),ts1).
}
```

In addition to the `begin` and `end` states which apply at the beginning and the end of the manifold's execution respectively, two more states are invoked by the `AP_Cause` commands, namely `start_tv1` and `end_tv1`. At the `begin` state the instances of the atomics `cause1`, `cause2`, `mosvideo`, `splitter`, and `zoom` are activated. These activations introduce them as observable sources of events. This state is *synchronized* to preempt to `start_tv1` with the execution of `cause1`. More to the point, the declaration of the instance `cause1`

```
process cause1 is
AP_Cause(eventPS,start_tv1,3,CLOCK_P_REL)
```

indicates that the preemption to `start_tv1` should occur 3 seconds (relative time) after the raise of the presentation start event `eventPS`.

Within `start_tv1` the other three instances, `cause2`, `mosvideo`, and `splitter`, are executed in parallel. `cause2` synchronizes the preemption to `end_tv1` and its declaration

```
process cause2 is
AP_Cause(eventPS,end_tv1,13,CLOCK_P_REL)
```

indicates that the currently running state must execute the other two atomic instances within 13 seconds. So the process for the media object `mosvideo` keeps sending its data to `splitter` until the state is preempted to `end_tv1`. The `mosvideo` coordinating instance supplies the video frames to the `splitter` manifold. The role

of splitter here is to process the video frames in two ways. One with the intention to be magnified (by the zoom manifold) and the other at normal size directly to a presentation port. zoom is an instance of an atomic which takes care of the video magnification and supplies its output to another port of the presentation server. The presentation server instance ps filters out the input from the supplying instances, i.e. it arranges the audio language (English or German) and the video magnification selection. At the end_tv1 state the presentation ceases and control is passed to the end state. Finally at the end state, the tv1 manifold is activated and performs the first question slide manifold ts1. This prompts a question, which if answered correctly prompts in return the next question slide. A wrong answer leads to the replaying of the presentation that relates to the correct answer, before going on with the next question slide. The code for a slide manifold is given below.

```
manifold tslide1()
{
 begin:(activate(cause7),cause7,WAIT).
 start_tslide1:(activate(testslide),testslide,WAIT).
 tslide1_correct: "your answer is correct"->stdout;
                  (activate(cause8),cause8,WAIT).
 tslide1_wrong:"your answer is wrong"->stdout;
               (activate(cause9),cause9,WAIT).
 end_tslide1:(post(end),WAIT).
 start_replay1:
(activate(replay1,cause10),replay1,cause10,WAIT).
 end_replay1: (activate(cause11),cause11,WAIT).
 end:(activate(ts2),ts2).
}
```

The instance cause7 is responsible for invoking the start_tslide state. The declaration for the cause7 instance is

```
process cause7 is
AP_Cause(end_tv1,start_slide1,3,CLOCK_P_REL)
```

Here we specify that start_slide1 will start 3 seconds after the occurrence of end_tv1. Inside that, the testslide instance is activated and eventually causes preemption to either tslide_correct or tslide1_wrong, depending on the reply.

The tslide_wrong instance causes transition to the start_replay1 state which causes the replay of the required part of the presentation and then preempts through cause10, to end_replay1. That in turn preempts through cause11, to end_replay1, after replaying the relevant presentation. The end_replay marks the end of the repeated presentation and preempts to end_tslide1. The tslide_correct state, also causes the end_tslide1 event through the instance cause8. The end_tslide1, simply preempts to the end state that contains the execution of the next slide's instance.

The main program begins with the declaration of the events used in the program.

```
AP_PutEventTimeAssociation_W(eventPS)
```

is the first event of the presentation and puts the current time as its time point. For the rest of the events the function

```
AP_PutEventTimeAssociation(event)
```

is used which leaves the time point empty. Then the implicit instances of the media manifolds, are executed in parallel at the end of the block. These are

```
(tv1,eng_tv1,ger_tv1,music_tv1)
```

`tv1` is the manifold for the video transmission, `eng_tv1` is the manifold for the English narration transmission, `ger_tv1` is the manifold for the German narration transmission and `music_tv1` is the manifold for the music transmission.

5 Conclusions

In this paper we have addressed the issue of real-time coordination in parallel and distributed systems. In particular, we have extended a control- or event-driven coordination language with a real-time event manager that allows expressing timing constraints in the raising, observing, and reacting to events. Thus, state transitions are done in a temporal sequence and affect accordingly the real-time behaviour of the system. We have tested our model with a scenario from the area of multimedia systems where recently issues of coordination and temporal synchronization at the middleware level have been of much interest to researchers ([2]).

References

1. F. Arbab, "The IWIM Model for Coordination of Concurrent Activities", *First International Conference on Coordination Models, Languages and Applications (Coordination'96)*, Cesena, Italy, 15-17 April, 1996, LNCS 1061, Springer Verlag, pp. 34-56.
2. G, Blair, J-B. Stefani, *Open Distributed Processing and Multimedia*, Addison-Wesley, 1998.
3. IEEE Inc., "Another Look at Real-Time Programming", Special Section of the *Proceedings of the IEEE* **79(9)**, September, 1991.
4. G. A. Papadopoulos and F. Arbab, "Coordination Models and Languages", *Advances in Computers*, Marvin V. Zelkowitz (ed.), Academic Press, Vol. 46, August, 1998, 329-400.
5. M. Papathomas. G. S. Blair and G. Coulson, "A Model for Active Object Coordination and its Use for Distributed Multimedia Applications", LNCS, Springer Verlag, 1995, pp. 162-175.
6. S. Ren and G. A. Agha, "RTsynchronizer: Language Support for Real-Time Specifications in Distributed Systems", *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, La Jolla, California, 21-22 June, 1995.