Programming Execution-Time Servers in Ada 2005

A. Burns and A.J. Wellings
Real-Time Systems Research Group
Department of Computer Science
University of York, UK

Abstract

Much of the research on scheduling schemes is prevented from being used in practice by the lack of implementations that provide the necessary abstractions. An example of this lack of provision is the support of execution-time servers, these important building blocks are not generally available to the system developer. In this paper, we show how new Ada 2005 mechanisms can be used to construct various execution-time servers. We also outline the different server types that could form part of a library of real-time utilities for Ada.

1. Introduction

One of the key building blocks for delivering flexible scheduling is the use of *execution-time servers* [10, 11, 9, 2, 1] – we will refer to these as just *servers* in the following discussions. Server are in some senses virtual processors, they provide their clients with a budget that has a defined 'shelf life' and a means of replenishing the budget in a predictable and analysable way. The servers collectively must manage their budgets (i.e. allow the budget to be available to clients), whilst the clients must ensure that the budget is sufficient for their needs. A number of papers have addressed the scheduling issues associated with server-based systems. In this paper we are concerned with programming servers.

The paper is organised as follows. First two examples of the expressive power of Ada 2005 are illustrated: the programming of the Deferrable and then the Sporadic server. The code in these examples comes from a recent publication [6]. This is followed in Section 4 by a discussion of the various properties servers can have. This discussion is intended to lead to the definition of reusable real-time utilities that could form part of a library or secondary standard [12].

2. Deferrable Server

In this section we illustrate how the Deferrable Server can be constructed. Space considerations mean that the Ada 2005 facilities are not described. Here, the server has a fixed priority (i.e. all client tasks execute with the same priority), and when the budget is exhausted the tasks are moved to a background priority. A client task first registers with its server, after that, in this simple example, it has no direct interactions with the server. The server itself needs a high interrupt priority level as it handles timing events.

```
-- with clauses are omitted for brevity
package Deferrable is
  type Deferrable_Server_Parameters is
  record
    Period : Time_Span;
    Budget : Time_Span;
    Foreground_Pri : Priority;
    Background_Pri : Priority;
  end record;

  protected type Deferrable_Server(
            Params : access Deferrable_Server_Parameters)
    procedure Register(T : Task_Id := Current_Task);
    pragma Priority(System.Priority'Last);
  private
    procedure Timer_Handler(E : in out Timing_Event);
      -- fired when replenishment is due
    procedure Group_Handler(G : in out Group_Budget);
      -- fired when budget exhausted
    T_Event : Timing_Event;
    G_Budget : Group_Budget;
    First : Boolean := True;
  end Deferrable_Server;
end Deferrable;
```

A single timing event is used to replenish the budget at regular timing intervals. A group budget is employed to take the necessary actions when the current budget is exhausted. In the following code the first client task to register sets up the server; this could have been done by a separate routine called when the server is created.

```
package body Deferrable is
  protected body Deferrable_Server is
    procedure Register(T : Task_Id := Current_Task) is
    begin
      if First then
```

```
        First := False;
        G_Budget.Add(Params.Budget);
        T_Event.Set_Handler(Params.Period,
                            Timer_Handler'Access);
        G_Budget.Set_Handler(Group_Handler'Access);
      end if;
      Add_Task(G_Budget, T);
      if G_Budget.Budget_Has_Expired then
        Set_Priority(Params.Background_Pri);
        -- sets client task to background priority
      else
        Set_Priority(Params.Foreground_Pri);
        -- sets client task to servers 'priority'
      end if;
    end Register;

    procedure Timer_Handler(E : in out Timing_Event) is
      T_Array : Task_Array := G_Budget.Members;
    begin
      G_Budget.Replenish(Params.Budget);
      for ID in T_Array'Range loop
        Set_Priority(Params.Foreground_Pri,T_Array(ID));
      end loop;
      E.Set_Handler(Params.Period,Timer_Handler'Access);
    end Timer_Handler;

    procedure Group_Handler(G : in out Group_Budget) is
      T_Array : Task_Array := G_Budget.Members;
    begin
      if G_Budget.Budget_Has_Expired then
        -- test needed to cover race condition
        -- with timer handler
        for ID in T_Array'Range loop
          Set_Priority(Params.Background_Pri,
                       T_Array(ID));
        end loop;
      end if;
    end Group_Handler;
  end Deferrable_Server;
end Deferrable;
```

3. Banded Sporadic Server

To programme the Sporadic Server also requires the use of timing events and a group budget. In the following, tasks are suspended if there is no budget, and a finite set of reused timing budgets is employed. If this number if exhausted, the POSIX scheme of concatenating two replenishments into one is employed[8].

The client is assumed to request the use of the budget by bracketing (non-blocking) application code in the following way:

```
BSS.Register;
...
loop
  BSS.Start_Session;
  -- where BSS is of the server type below

  -- non-blocking code
  BSS.Complete_Session;
end loop;
```

The Sporadic Server allows its clients to preempt each other by supporting a range of priority levels (hence the term *banded sporadic server*). All clients must have a priority within this range. With the sporadic server a client that arrives at time t and uses budget b, results in the replenishment of b at time t+T –where T is the period of the server. To program this, the timing event must know how much budget to return. This is accommodated by extending the timing event type.

```
package Banded_Sporadic is
  Priority_Out_Of_Range : exception;
  Already_Member_Of_A_Group_Budget: exception;

  type Banded_Sporadic_Server_Parameters is record
    Period : Time_Span;
    Budget : Time_Span;
    Low_Priority : Priority;  -- of the band
    High_Priority : Priority; -- of the band
  end record;

  type Budget_Event is new Timing_Event with record
    Bud : Time_Span;
  end record;

  type Bud_Event is access Budget_Event;
  type Bud_Events is array(Natural range <>)
                                  of Budget_Event;

  protected type Banded_Sporadic_Server
     (Params : access Banded_Sporadic_Server_Parameters;
      No_Timing_Events : Positive) is
    pragma Interrupt_Priority (Interrupt_Priority'Last);
    procedure Start_Session(T : Task_Id := Current_Task);
    procedure Complete_Session(T : Task_Id
                               := Current_Task);
    procedure Register(T : Task_Id := Current_Task);
  private
    procedure Timer_Handler(E : in out Timing_Event);
    procedure Group_Handler(G : in out Group_Budget);
    G_Budget : Group_Budget;
    B_Events : Bud_Events(1 .. No_Timing_Events);
    Next : Natural := No_Timing_Events;
    Number : Natural := 0;
    Start_Budget : Time_Span;
    Release_Time : Time;
    Tasks_Executing : Natural := 0;
    First : Boolean := True;
  end Banded_Sporadic_Server;
end Execution_Servers.Banded_Sporadic;
```

The only real complexity in this code comes from the reuse of a finite collection of timing events. The procedure `Set_Timing_Events` is used to manage this.

```
with Ada.Asynchronous_Task_Control;
use Ada.Asynchronous_Task_Control;
package body Execution_Servers.Banded_Sporadic is
  protected body Banded_Sporadic_Server is
    procedure Set_Timing_Events(B : Time_Span;
                                T : Time) is
    begin
      if Number < No_Timing_Events then
        Next := Next mod No_Timing_Events + 1;
        B_Events(Next).Bud := B;
        B_Events(Next).Set_Handler(T,
                              Timer_Handler'Access);
        Number := Number + 1;
      else
        B_Events(Next).Bud := B_Events(Next).Bud + B;
        B_Events(Next).Set_Handler(T,
                              Timer_Handler'Access);
      end if;
    end Set_Timing_Events;
```

```
procedure Register(T : Task_Id := Current_Task) is
begin
  if not (Get_Priority(T) in Params.Low_Priority ..
          Params.High_Priority) then
    raise Priority_Out_Of_Range;
  end if;
  if First then
    First := False;
    G_Budget.Add(Params.Budget);
    G_Budget.Set_Handler(Group_Handler'Access);
  end if;
  G_Budget.Add_Task(T);
exception
  when Group_Budget_Error =>
    raise Already_Member_Of_A_Group_Budget;
end Register;

procedure Start_Session(T : Task_Id)  is
begin
  if Tasks_Executing = 0 then
    Release_Time := Clock;
    Start_Budget := G_Budget.Budget_Remaining;
  end if;
  Tasks_Executing := Tasks_Executing + 1;
  if G_Budget.Budget_Has_Expired then
    Hold(T);
  end if;
end Start_Session;

procedure Complete_Session(T : Task_Id) is
begin
  -- work out how much budget used, construct
  -- timing event and set the handler
  Tasks_Executing := Tasks_Executing - 1;
  if Tasks_Executing = 0 then
    Set_Timing_Events(
        Start_Budget - G_Budget.Budget_Remaining,
        Release_Time + Params.Period);
  end if;
end Complete_Session;

procedure Timer_Handler(E : in out Timing_Event) is
  Bud : Time_Span;
  T_Array : Task_Array := G_Budget.Members;
begin
  Number := Number - 1;
  Bud := Budget_Event(Timing_Event'Class(E)).Bud;
  if G_Budget.Budget_Has_Expired then
    G_Budget.Replenish(Bud);
    for I in T_Array'range loop
      Continue(T_Array(I));
    end loop;
    Release_Time := Clock;
    Start_Budget := Bud;
  else
    G_Budget.Add(Bud);
    Start_Budget := Start_Budget+Bud;
  end if;
end Timer_Handler;

procedure Group_Handler(G : in out Group_Budget) is
  T_Array : Task_Array := G_Budget.Members;
begin
  if G_Budget.Budget_Has_Expired then
    -- a replenish event required for the
    -- budget used so far
    Set_Timing_Events(Start_Budget, Release_Time +
                      Params.Period);
    for I in T_Array'range loop
      Hold(T_Array(I));
    end loop;
  end if;
 end Group_Handler;
end Banded_Sporadic_Server;
end Execution_Servers.Banded_Sporadic;
```

A replenishment event is set up either when the current budget is exhausted or, at the end of a client's session, if there are no longer any active clients.

4. Different Server Characteristics

In this section a number of server characteristics are considered. In many instances these characteristics are orthogonal and hence gives rise to a wide range of possible structures. In all of this discussion fixed priority scheduling is assumed. The use of servers and other scheduling policies such as EDF is not considered in this paper.

4.1. Dispatching

Here two characteristics are identified: concurrency within the server, and the behaviour of the tasks when the server capacity is exhausted. With the former there are three possibilities:

1. All client tasks have the same priority and hence their use of the server's budget is serialised.

2. Client task have distinct priorities but the range of priorities for each server is disjoint. Client tasks can now preempt each other and thereby exhibit a more responsive behaviour.

3. Client task have distinct priorities, and there are no constraints on the priorities. This is the general model supported by Java's (RTSJ) processing groups [4] – but the resulting system is not easily amenable to scheduling analysis [5].

The code for the Deferrable and Sporadic servers have illustrated the first two of these schemes. They have also used alternative policies for dealing with tasks when there is no budget available:

1. Run the client tasks at a background (low) priority.

2. Suspend the tasks.

The suspension of the tasks can make certain aspects of the implementation easier (as the client tasks cannot have executed during the time the server has no capacity). Also with a large system with many servers, there may be little likelihood of spare capacity been available at the background level, and hence the use of suspension is not as inefficient as it might seem.

4.2. Binding

Here a task coordinates its release with the replenishment of the server (either a Deferrable or Periodic server – see below). The task is described as being *bound* to the server. The advantage of this scheme is that it makes the scheduling analysis of these bound tasks less pessimistic [7]. One means of achieving this binding is to introduce into the server a start session entry that the client task calls as their 'wait for next invocation' event.

```
entry Start_Session(T : Task_Id := Current_Task)
                                    when Released is
begin
  Released := Start_Session'Count > 0;
end Start_Session;
```

With the boolean barrier being set in the timing event handler:

```
Released := Start_Session'Count > 0;
```

4.3. Stop on Exhaustion of the Budget

The above illustrates a coordination between the replenishment of a budget and the release of a task. Another form of coordination is between the exhaustion of the budget and the performance of some 'non-terminating' algorithm that is 'stopped' when there is no longer any budget. To implement this, the server would have an entry (e.g. `Exhausted`) that has a barrier variable that is set to true when the group budget event is fired. The application code will call this entry from an ATC structure:

```
loop
  Some_Server.Start_Session;
  select
    Some_Server.Exhausted;
    -- Place result in some appropriate object
  then abort
    code
  end select;
end loop;
```

This server does not change the priority of the client when the budget is exhausted but a `Start_Session` entry is used that is only open when there is budget available. The two entries, `Exhausted` and `Start_Session` could easily be added to the Deferrable Server's specification.

4.4. Budget Sharing

There are a number of schemes described in the literature (eg. [3, 1]) that allow the budget in one server to be passed to another server if there are no local clients requiring service. A collection of fixed priority Deferrable Servers (for example) could be constructed so that each server always knows its 'neighbour'. This is the server with the next highest priority. If this neighbour had active clients they would be executing; as it isn't executing it must have either no such clients or have no current budget. The following scheme is based on the premise that when a budget is exhausted (i.e. the group budget event is fired) an attempt is made to pull capacity down from its neighbour. Only if this fails are the client tasks demoted. The neighbour will pass on a gift of its current budget if it has one – if not, it will attempt to pull down budget from its neighbour.

To implement this scheme, a new function is added to the interface of the Deferrable Server (`Extract`). The group handler can now be executed when either one of its own clients was active and the budget was exhausted or the remaining budget was gifted away. In the latter case no attempt is made to extract extra budget from the servers' neighbour. A new boolean variable `Gift_Aid` is used to distinguish between these two cases. The overall approach is sketched below.

```
package body Deferrable is
  protected body Deferrable_Server is
    procedure Register(T : Task_Id := Current_Task)
                                                 is ...
    procedure Timer_Handler(E : in out Timing_Event)
                                                 is ...
    procedure Group_Handler(G : in out Group_Budget) is
      T_Array : Task_Array := G_Budget.Members;
      Gift : Time_Span := Time_Span_Zero;
    begin
      if G_Budget.Budget_Has_Expired then
        if Gift_Aid then
          Gift := Time_Span_Zero;
        else
          Gift := Neighbour.Extract;
        end if;
        if Gift = Time_Span_Zero then
          for ID in T_Array'Range loop
            Set_Priority(Params.Background_Pri,
                         T_Array(ID));
          end loop;
        else
          G_Budget.Replenish(Gift);
        end if;
        Gift_Aid := False;
      end if;
    end Group_Handler;

    function Extract return Time_Span is
      Gift : Time_Span;
    begin
      if G.Budget.Budget_Has_Expired then
        return Neighbour.Extract;
        -- if highest priority server then no neighbour
        -- so return Time_Span_Zero
      end if;
      Gift_Aid := True;
      Gift := G_Budget.Budget_Remaining;
      G_Budget.Replenish{Time_Span_Zero);
      return Gift;
    end Extract;
  end Deferrable_Server;
end Deferrable;
```

Obviously the use of capacity sharing must meet the re-

quirements of the application. A client may have to wait until the next replenishment in order to execute – as its server's capacity has been given away. Hence, it is best to use this technique with servers that have bound tasks (se section 4.2. If the technique is too extreme, the scheme can be restricted so that only a proportion of the server's capacity is gifted away.

Other kinds of capacity sharing are also possible. There is a need to characterise these and show how they can be implemented in Ada 2005.

4.5. Server Types

In addition to Deferrable and Sporadic servers there are a number of other schemes discussed in the literature. The main additional one is the Periodic Server. This behaves like a Deferrable Server except that its capacity is not preserved during the server's period; it is only available to clients who are ready to execute at the time the budget is replenished. This has the advantage that the Periodic Server behaves just like a periodic task - in terms of its impact on lower priority tasks and servers. This is not the case for a Deferrable Server that can have an increased impact due to its clients arriving late one period and early the next.

There are two distinct means of providing the required behaviour for a Periodic Server:

1. Each server has a looping client that has the lowest priority of all users of the server but which executes a non-blocking busy loop. It will always be available to execute and will use up all the available capacity.

2. When there are no current clients left to execute, reduce the budget to zero. If budget sharing is to be employed then the remaining budget could be passed on by adding it to the available budget of another server.

The latter clearly is less wasteful and can be easily introduced by again using a start and end session interface. The server keeps a count of the number of active agents; when this value goes to zero the remaining budget is removed (or re-assigned).

5. Conclusion

The focus of this paper has been on the construction of server abstractions using the new facilities of Ada 2005. Simple servers such as the Deferrable Server are straightforward and need just a simple timing event and a group budget. The Sporadic Server by contrast is quite complicated and its implementation is a testament to the expressive power of the language.

One of the motivations of this paper is to start to define a collections of useful server abstractions, and to facilitate the use of these abstractions via a library of tested components, and/or a secondary standard.

Acknowledgements

The work reported in the paper is funded, in part, by the EU project ARTIST.

References

[1] G. Bernat, I. Broster, and A. Burns. Rewriting history to exploit gain time. In *Proceedings Real-time Systems Symposium*, pages 328–335, Lisbon, Portugal, 2004. Computer Society, IEEE.

[2] G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *Proceedings 20th IEEE Real-Time Systems Symposium*, pages 68–78, 1999.

[3] G. Bernat and A. Burns. Multiple servers and capacity sharing for implementing flexible scheduling. *Real-Time Systems Journal*, 22:49–75, 2002.

[4] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[5] A. Burns and A. J. Wellings. Processing group parameters and the real-time specification for java. In *On the Move to Meaningfull Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems*, volume LNCS 2889, pages 360–370. Springer, 2003.

[6] A. Burns and A.J. Wellings. Programming execution-time servers in ada 2005. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 47–56, 2006.

[7] R. Davis and A. Burns. Hierarchical fixed priority preemptive scheduling. In *IEEE Real-Time Systems Symposium*, pages 389–398, 2005.

[8] IEEE Std.1003.1c-1995. Information Technology - Portable Operating System Interface (POSIX): Part 1 : System Application program interface (API) – Amendment 2: Threads Extension [CLanguage], 1995.

[9] J.P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks fixed-priority preemptive systems. In *Proceedings 13th IEEE Real-Time Systems Symposium*, pages 110–123, 1992.

[10] J.P. Lehoczky, L. Sha, and J.K. Strosnider. Enhanced aperiodic responsiveness in a hard real-time environment. In *Proceedings 8th IEEE Real-Time Systems Symposium*, pages 261–270, 1987.

[11] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1:27–69, 1989.

[12] A.J. Wellings and A. Burns. A framework for real-time utilities for Ada 2005. In *Offered to the 13th International Real-Time Ada Workshop*. 2007.