THE RTC - A SOFTWARE SUPPORT SYSTEM FOR THE CONTROL OF THE CERN SPS

W. Herr, R. Lauckner and G. Morpurgo
SPS Division, CERN, Geneva, Switzerland

Abstract

The RTC (Run Time Coordinator) is a software support system designed for the SPS control system to provide a runtime environment for application software. It coordinates the execution of individual programs or processes and supervises the process control, i.e. process synchronisation, inter process communication, data transfer and operator I/O. This supervision includes the control of processes distributed on a UNIX based network. A standard language independent data interface is part of the system. The system includes tools for data presentation, error logging and contention resolution. This strategy of separating system dependent features from the body of the application programs leads to high flexibility and simplifies the software development.

The basic philosophy of the RTC is discussed and its implementation is described.

INTRODUCTION

What we have to deal with The infrastructure in which software for the control system of the SPS has to be developed can be considered as a single computer network. The network is heterogeneous : different types of computers (Apollo Workstations, IBM PC-ATs, IBM PC-RTs, Norsk-Data ND 100s) running different operating systems (AEGIS-UNIX, XENIX-UNIX, Syntron[1]). Some of the computers in the network are also connected to 'front end processors', which establish an interface to the hardware controlling the different parts of the SPS.

Programs running on this network can be written in different languages (typically C, Fortran and NODAL[2]).

What we want to do In order to control the accelerator, or parts of it, people write what we call **applications** : an application can be a software program, or a set of programs, normally made of several parts, running on one or more computers in the network. We want to run **distributed applications** on this network and we also want to be able to run individual parts of an application independently. We want to be able to run several applications at the same time. Finally, we want to make life easier for both the people writing the applications (hiding from them as many problems as possible which are not directly related with the main goal of their application), and the operators, who have to run the applications (by giving them an uniform way of starting and controlling the applications).

[1]Syntron is the operating system running on the SPS Norsk-Data computers

[2]NODAL is an interpretative language running on the SPS Norsk Data computers

THE BASIC PROBLEM

An application, as we said, can be made of several modules, running on different computers on the network. In building an application, several problems have to be dealt with :

- The overall control of the execution flow and the synchronization of the different modules composing the application. We can isolate several aspects of this problem :
  - Starting of the different modules
  - Sequencing
  - Execution of one module conditioned by the status or the execution result of another one
  - Cyclic repetition of a module
  - Informing one module that data generated by another module is ready
- The networking : how to start the modules on different nodes, and how to maintain the control of what is going on.
- The data communication between the different parts of the application : for example, how to make available to a module data generated by another module running on another computer.

All of these problems, and others we will introduce later, are not directly related with the main goal of the application itself, which is to perform some operation on a part of the accelerator; also, they are common to a wide range of applications. We could say that they are *system software* problems.

THE RTC IDEA

It should not be necessary for the application programmer to have to deal with system software problems. Firstly because that is not his job, and he has to divert part of his precious time from his real objectives. The other, deeper, reason is that if the programs he writes contain explicit solutions to the system software problems he meets, these programs will become system dependent.

Consider, for instance, the case in which a part of the application (a program) explicitely starts another part (another program) when a certain condition occurs. In this case, it will no longer be possible to run these programs individually, to reuse them in other environments. Also it will be more difficult to modify one part of the application without creating unforeseen side effects in the overall execution of the application. Finally, it will be impossible to modify the overall execution flow without modifying and recompiling the programs.

In order to overcome all these problems a system has been designed [1],[2] and implemented. This system is called the **Run Time Coordinator (RTC)**[5], and it is composed of a distributed set of supervisor programs, under whose control the applications will be executed.

Main goals of the RTC are to enable the application programmer to

- Split a complex, distributed application into a set of functional units.

CH2669-0/89/0000-1678$01.00©1989 IEEE

- Specify outside the application programs the control flow and the synchronization between the different modules (*functional units*) composing the application.
- Also keep outside the programs computer dependent features (like creation of windows)
- Specify the data flow between different modules outside the application programs.

As a consequence, the functional units composing an application can be strongly decoupled, because they do not need to explicitely refer to each other.

From now on, we will call "**task**" an application, and "**process**" any of the functional units composing an application. From the main goals stated above, it follows that the RTC system

- Must provide the application programmers with an easy way of specifying the external functionality of the task.
- In particular, there should be the possibility of specifying synchronization and data communication between different processes in a task.
- Must be able to execute the task in the way specified by the application programmer.

FORMALIZATION OF THE IDEAS

In order to let the application programmer specifying the interactions between the different processes of a task, a simple language has been defined . This language enables the application programmer to express in a simple and readable way his requirements concerning control flow, data flow , and process synchronization. The application programmer writes a description (the **Task Specification**) in which the global properties of the task, the definition of the processes composing the task, and the control and data flow of the task are specified using this **Task Specification Language (TSL)**[3][5].

The global properties of a task consist of a name via which the RTC system will recognize the task, plus some fields defining the applicability of the task itself. A UNIX file can be specified, containing environment variables to be passed to all the processes of the task.

The definition of a process includes fields like the host where the process has to run, the executable file corresponding to the process, plus some optional fields, e.g. command line arguments which have to be passed to the process, the type of interaction the process will have with the operator, size and position of the process window. An important field specifies what data is generated by the process.

The formalization of the control and data flow of the task, and of the interaction between processes, are defined by a set of statements. These statements link the execution of an **action** to the occurrence of an **event**, or of a Boolean combination of events.

The RTC language supports the following actions

- execute a process
- signal a process
- kill the task

and the events

- a process terminates or starts
- a process exits with a given exit value
- a process reports a given value to the RTC
- data has been generated
- an accelerator timing event occurs
- a time interval is finished
- a process is not allowed to run

The application programmer can combine these primitives to express the desired control flow for his task. Clauses, like *repetitions, preconditions, intervals between the occurrence of an event and the execution of the related action* add flexibility to the scheme. In the appendix some examples of possible statements are shown.

The Task Specification is then passed through the **Task Specification Compiler**. This operation checks that the syntax of the TSL was respected, and produces a set of tables. The RTC will use these tables at run time to perform the execution of the task in the way specified by the programmer.

From what has been said until now, it is not clear how the programmer can specify the data flow : the only two points where reference to data appears in the Task Specification are the *data generated by a process* , in the process block, and the statements of the form *execute action when data has been generated*. The way RTC supports data communication between different processes is based on the UNIX *interprocess shared memory* facility. On top of this UNIX facility, an object-oriented dynamic memory management system, called **MOPS** (*Multiple Object Partitioned Structure*) has been implemented and made usable from different languages. A MOPS [4] is a memory area dynamically partitionable into *objects*. Several programs can share the access to a MOPS, and they can refer by name to the objects it contains. In order to communicate data between RTC processes, these processes will create, write and read objects in MOPS. A special *Mops Server*[6], with the functionality of transferring MOPS data structures over the network, even between different types of computers, has been written.

If the action to be performed as a consequence of a 'data is written' event was the execution of a process, **the RTC will assume that this process needs the data, and it will take care of making the data available to the process before executing it**. In this way, the programmer is relieved from the problem of distributing the data over the network.

THE RUN TIME IMPLEMENTATION

The Run Time Coordinator, as stated before, consists of a set of supervisor programs, under whose control tasks and processes are executed. Our implementation assigns a well defined function to each of these supervisors. We have

- A **Scheduler** program. The Scheduler is responsible for the overall control of the task execution. It receives the request for activating a certain task, and it performs the actions related to the starting of the task. Actions are either performed directly, or by issuing commands to other supervisors.
  In the current implementation, all the events related to the task (termination of a process, data updating, timeouts, etc.) are reported to the Scheduler on the computer from which the task was launched. Every time an event is reported to it, the Scheduler will look into the task tables produced by the Task Specification Compiler, and will perform any action related to that event.
- An **Executor** program. The Executor is responsible for executing the processes belonging to the task. These processes run as *UNIX children* of the Executor. The Executor also deals with assignment of windows, and input/output redirection, and with setting the environment and working directory for the process.
  Every time one of its children terminates, the Executor signals the event to the Scheduler.
  The Executor can also receive from the Scheduler the order to kill all the processes belonging to a given task.
- The **Timer**. All the operations involving time intervals are

delegated to the Timer process. This process receives from the Scheduler the requests for setting up single or repetitive time intervals and it will report an event to the Scheduler every time one of these interval expires.

The execution of a task with processes running on different computers requires communication of information over the network. For instance, the Scheduler controlling the task has to inform the Executor on another computer that a process has to be run, and later on the Executor has to report to the Scheduler that the process is terminated. Two programs, the Sender and the Receiver, deal with this problem.

- The **Sender**. The Sender process receives, from the other supervisors on its node, all the messages to supervisors on other nodes. The Sender transmits the message to the Receiver on the destination node.
- The **Receiver**. The Receiver receives messages from Sender processes on other nodes, and transmits these messages to the supervisors to whom the messages are addressed. It can also receive a message requiring the distribution of a MOPS. In this case the Receiver creates a copy of that MOPS on the node where it runs.

The different supervisors running on the same computer communicate with each other via UNIX messages and queues. Message exchange between the Sender on one node and the Receiver on another one makes use of a Remote Procedure Call utility developed at CERN and called the *Network Compiler*[7].

Our implementation of the RTC is written in the 'C' language for the UNIX environment. The essential parts should therefore be portable to any other computer running the UNIX operating system.

A user friendly interface has been also implemented. From dedicated *consoles* (Apollo workstations) the SPS operators can launch up to 10 different tasks at the same time, follow their executions, and make windows associated with the task processes visible or invisible. All of this just by using a single mouse button.

CONCLUSION

We have described a software system, the RTC, which offers advantages both to those writing the software and to those who run it. By using the RTC, application programmers can split complex distributed applications into small, decoupled functional units. Furthermore, he does not have to deal with system dependent problems. The interaction between these units is specified in a simple language.

The set of functional units constituting an application is then run under the control of a distributed set of supervisor programs (the RTC). Tools are provided to give the operators the possibility of following and also controlling the execution of several applications at the same time.

By modifying the Task Specification it is easy to change the way in which the application is run, and it is straightforward to adapt it to new external requirements.

References

[1] L. R. Evans et al. *Software Managers for Multicycling* CERN/SPS/AOP/86-10

[2] W. Herr, R. J. Lauckner, C. G. Saltmarsh *Design Principles of the High Level Controls Structure for CERN SPS.* CERN/SPS/87-40(AMS)

[3] R. J. Lauckner *The Task Specification Compiler* CERN/SPS/AMS/NOTE/87-5

[4] W. Herr *M.O.P.S. User Guide for "C" Programs.* CERN/SPS/88-43(AMS)

[5] G. Morpurgo *The Run Time Coordinator and Task Specification Compiler User Guide* CERN/SPS/ACC/89

[6] W. Herr and G.Morpurgo *Binary Transfer of MOPS Data Structures* CERN/SPS/ACC/89-1

[7] I. Kristiansen *The Network CompilerUser Guide.* Unpublished.

A.1 Direct process–>RTC communication

A task process can communicate directly with the RTC, either by *reporting a status value*, or by *declaring that a piece of data has been written.* Routines supporting this communication have been written. These routine are only effective if the process is actually running under the RTC control; otherwise they do not have any effect. This preserves the possibility of running any process outside the RTC control.

A.2 Task Specification Language samples

```
/* An example of a task widely used
                    by the SPS operators... */
#define YES 4
#define WHISTLE 16
#define EMPTY -1
#define SHUT_DOWN -1
/*      The task block                          */
task name:        coffee_maker
machine types:    all
beam types:       positron_1, positron_2
permissions:      SPS_coordinator, shift_leader
       ...
/*       followed by process blocks             */
       ...
proc name:        buy_coffee
exec file:        /user/coffee/buy
host:             Migros
p_permissions:    SPS_budget
comlineargs:      Lavazza qualita' oro
int data gen:     coffee_powder
inter type:       full
window:           0 0 10 16
       ...
/*       and some possible statements           */
       ...
%*
exec  light_the_stove          /* immediate action */
exec coffee_box_server
repeat make_coffee when drink_coffee finished
repeat signal WHISTLE coffee_man when coffee arrives
repeat drink_coffee pausing 60 secs
                              when coffee arrives
repeat buy_coffee when coffee_box reports EMPTY
repeat add_sugar when (coffee arrives and
                       ask_sugar reports YES)
                       until sugar_box reports EMPTY
kill coffee_maker  when SPS reports SHUT_DOWN
*%
```

In the example, 'coffee_maker' is the name of the task.
'light_the_stove', 'coffee_box', 'add_sugar', 'buy_coffee', 'sugar_box', 'SPS', 'coffee_man' 'drink_coffee' and 'make_coffee' are processes.
'coffee' is data shared by several processes.