Hierarchical Partitions in Cyclic Closed Systems: A Hardware-Oriented Approach

F. Rosemberg* and S. Ruhman†
Weizmann Institute of Science, Rehovot, Israel

The problem of developing real-time embedded computing systems is addressed, and a simpler hierarchically partitioned layered model is suggested. It is recommended for use in the specification of system requirements and for implementation in hardware as well as software. In addition to the theoretical importance of formally defining a class of systems which encompasses industrial control applications as well as avionics, the model has the practical advantage of significantly reducing the effort required to develop specific systems. The configuration considered is a multicomputer common bus. The concept starts from the idea that communication between tasks is prescribed by programs in a high-level language such as Ada or CSP. Results obtained to date in elaborating protocols and conceiving a bus interface architecture are generalized and a five-layer partition is suggested. Each layer is described, then information exchange between layers is formalized. Finally, it is indicated what part of the solution is worth implementing by a dedicated integrated circuit. This work combines two domains not usually studied in conjunction: high-level languages for distributed computing and common-bus communications. This joint consideration leads to more efficient designs, simpler, more elegant, and easier to understand.

Introduction

THE systems considered herein are typically found in industrial control or avionics. The physical environment in which they operate is usually well defined, hence they are described as "closed." Other characteristic features are high robustness and severe limitations in weight and power consumption. Generally known as embedded computer systems, they include sensors connected to each computer which sample the environment at equal time intervals, and actuators —motors or relays—also activated at regular intervals. Hence, operation is largely cyclic.

Timing constraints imposed by sampling of the inputs, activation of the outputs, and information transfer between the various computers cause serious synchronization problems and significant implementation efforts. We argue that this type of system is sufficiently prevalent and important to deserve a dedicated characterization, and suggest naming it: Real-Time Cyclic Closed Systems (RTCCS). Hardware-software interaction has always played an important role in the conception of computing systems. But recent advances in VLSI open new possibilities by allowing silicon implementation of a great number of processes previously done in software. Given that the solution is general enough to justify development of an integrated circuit implementation, savings in specific development efforts are obvious.

The RTCCS model described below is based on a hierarchical partition including five layers, the three lower ones being implemented by a dedicated, largely integrated, architecture. The paper starts with an abstract description of RTCCS based on nine assertions. Next we introduce our proposed RTCCS layered model together with the notion of message alphabets. A description of the two lower layers, communication specific, is then given for two cases, Ethernet[1] and our "Prioritized Dialog with Provision for Message Cancellation."[2] The next two layers, which are language-specific, are then treated with reference to CSP and Ada. The conclusions are followed by an appendix, which defines the intertask communication constructs of CSP and Ada.

Received Aug. 30, 1985; revision received Jan. 27, 1986.

*Department of Applied Mathematics; also, Israel Aircraft Industry MBT, Yahud, Israel.
†Department of Applied Mathematics.

RTCCS Characteristics

An RTCCS is contained within well-defined physical limits, a vehicle, a plant, etc. Its activity consists of sampling the environment, at equal time intervals in most cases, and sending activating signals to the environment as a result of the computations following the sampling. The environment often imposes severe constraints on the weight, volume, or power consumption of the system, influencing in many cases the technical solution adopted. The word "closed" emphasizes the importance of the physical conditions.

Following are assertions characterizing the RTCCS:

1) An RTCCS consists of N processors P_i, $i=1,2,\ldots,N$; each processor P_i being capable of hosting M processes Π_{ij}, $j=1,2,\ldots,M$.

2) Processes Π_{ij} may migrate from one processor P_i to another P_k $(k \neq i)$, provided its instantaneous location is known over the entire system. When executed by P_k, this process is denoted Π_{kj}.

3) Each processor P_i has its own CPU and memory.

4) Each processor P_i is connected to its own input-output (I/O) devices. These may, in turn, be computerized elements.

5) All processors P_i are interconnected through a *common bus*, which is the only common resource in the entire system.

6) Processes Π_{ij} executed by each processor P_i are largely independent, hence, they interchange a relatively low volume of information and may be considered loosely coupled.

7) Processes Π_{ij} are, in general, cyclic, characterized by durations T_{ij} which may be variable in time. In general, T_{ij} represents the minimum time interval between two successive samplings of the input devices controlled by the process Π_{ij}.

8) The algorithms computed by each Π_{ij} are well defined, deterministic, and cyclic. Only two stochastic elements are generally distinguished: the interaction between various processes situated in different processors due to instantaneous changes in relative speed and the occurrence of failures.

9) The computing load may be partitioned so that each process Π_{ij} consists of *sequential* parts associated with *communication* parts.

It will be shown how intertask synchronization in RTCCS systems may be achieved by a "coordination rule" common to both CSP[3] and Ada.[4]‡ "Consider an information transfer between process Π_{ij} and process Π_{kl}, $i \neq k$, and let Π_{kl} reach

‡Ada is a registered trademark of the U.S. Department of Defense.

its communication statement first. Its activity will be suspended until Π_{ij} reaches its corresponding communication statement, at which time the communication will take place, and Π_{kl} will be allowed to resume its activity. In other words, no communication is executed unless both participating processes reach their corresponding communicating statements."

Process migration among processors is not used in avionic systems, its application being difficult due to limitations in bus communication bandwidth and difficulties encountered to program it. High-level languages (HLL) for distributed processing such as Ada, as well as increased speed communications, may ease the task considerably. For example, during maintenance operations, tasks may be transferred, as a result of a previous test, from the checking processor to the device under test, to execute a more detailed diagnosis. Indeed, in all cases, processes related to input and output devices, can migrate only among processors, connected either to the same or identical I/Os.

General Presentation of the Layered Model

The RTCCS model consists of five layers as indicated in Fig. 1. We started from the International Standards Organization (ISO) reference model for Open Systems Interconnections (OSI),[5] which has seven layers below that of the user. The intrinsic characteristics of RTCCS, together with the use of a language designed for interprocess communication, permits reduction to five layers, including the application or user layer. The model suggested consists of five layers including that of the user. The OSI reference model includes seven layers below that of the user.

Message alphabets (MA) are defined as the set of information entities, sent from one layer to an adjacent one with the purpose of requesting or acknowledging a service, or of going through the lower layers to communicate with the peer layer. MAs are always specified in relation to two adjacent layers. Each information entity of a message alphabet consists of an *instruction*, which may be a *service request* or *service status*, and, optionally, an *object* which in most cases is a pointer to a message. The physical layer specification deals with the electrical characteristics of the communication medium—the bus in our case—with signal modulation and demodulation, signal transmission, and reception.

The datalink layer provides and manages transmission and reception of isolated frames of data, destination address recognition, or source address addition. Referring to our RTCCS formalism, one may say that DLL deals with indices *ij* designating processor and process addresses. PHL and DLL are communication-specific layers, whereas KL and IPL are language-specific layers. The interprocess layer (IPL) serves communication between processes Π_{ij} resident in physically separate processors P_i. IPL decides what control messages must be sent to effect the interprocess information transfer expressed by HLL constructs in the application layer (AL). The kernel layer, the only one besides AL resident in P_i, serves the communications between different Π_{ij} resident in the same P_i, after separating them from those between processes situated in different P_is. Clearly, the KL → IPL and IPL → KL message alphabets vary with the language, but are independent of DLL or PHL rules.

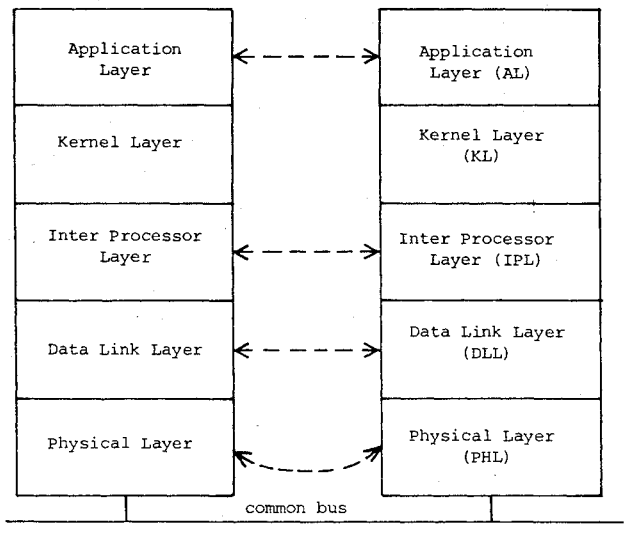

Fig. 1 RTCCS hierarchical layers.

Communication-Specific Layers: PHL → DLL and DLL → PHL

The two lower layers are communication-oriented. For the physical layer (PHL) and the data-link layer (DLL), Ethernet or IEEE Standard 802.3 (proposed) was adopted. The reasons were purely pragmatic. We assumed that with the support of Xerox, DEC, and Intel, Ethernet would quickly become a de facto standard, with components, tools, and software available to support it. The Ethernet probabilistic bus access was also very attractive because of its inherently distributed control. However, it soon proved to have some disadvantages.

In our system-oriented approach, the coordination rule stated above leads us to consider a "dialog" rather than an "isolated message" as the minimum entity for information interchange. A *dialog* is defined as an uninterrupted sequence of messages interchanged by two processes. Only the message initiating the dialog need contend for the bus and acquire it. The dialog is strongly required by Ada as well as CSP: in order to achieve data interchange between two processes resident in separate processors, one always needs more than a single message. In Ethernet, when two processes are ready to communicate, even after one of them succeeds in sending a message to the other, the dialog might well be interrupted by a third party attempting to access the bus. Hence the need for prioritized dialog with provision for message cancellation (PRD-MC) suggested in Ref. 2. It is based on two rules: priority and cancellation. The priority rule causes the bus access to be deterministic after a dialog begins successfully:

> "When a station correctly receives a message, it has first right of access to the bus such that its transmitted message will not be vulnerable to collision."

The rule is ensured by modifying the times each station is required to wait before attempting to access the bus. The second rule is that of canceling:

> "Requests for transmission of nonpriority messages, submitted by IPL to DLL, may be revoked."

This feature, unavailable in the original Ethernet, is required when a control message received from the bus makes some previous request for transmission no longer necessary. In this case, the overall processing duration including the submission of the canceling request should be shorter than the minimum allowed waiting interval before the next transmission. Such a cancellation is unconditionally granted. To gain in efficiency, cancellation is used in a different way in the upper layers as will be shown in the next paragraph.

PRD-MC is explained in detail and is specified formally in Ref. 2, where additional ways of increasing efficiency and reducing response time, such as the use of shorter control messages, are suggested. The message alphabets for IPL → DLL and DLL → IPL, in PRD-MC are given in Table 1.

Comparing with Ethernet, the instructions added here are priority-transmit-request and canceling-request.

The corresponding message alphabets for the unmodified Ethernet are shown in Table 2.

Similar partitioning into physical and datalink layers may be applied, although not as efficiently, to other common bus regimes. We would like to discuss MIL-STD-1553, the stan-

dard in use for communication in avionics systems, without going into its DLL → IPL and IPL → DLL message alphabets.

In the case of avionics, standardization activity started in 1968, the main purpose of this effort being total system integration. The most disturbing factor then was the existence of numerous wire bundles carrying signals between various subassemblies. Often, signals interfered with one another, an additional penalty being the weight of the wires. The first achievement was elimination of the traditional point-to-point wiring scheme.

The decision to replace this scheme by bus techniques, mainly a serial, digital, time-division multiplexed method of data interchange, was summarized in 1973 in MIL-STD-1553 by the Society of Automotive Engineers (SAE). Issued long before OSI-RM, MIL-STD-1553 does not specify it explicitly, but seems to deal with the two lower OSI-RM layers, Initially it was issued under the supervision of the U.S. Air Force. In 1975, it was revised (MIL-STD-1553 A) and appeared as a triservice document (Air Force, Army, and Navy). In 1978, this standard was revised again, augmented by additional details relating to the computer interfacing, and has never been changed since (Version B).[7] The communication is centrally controlled, with a speed of 1 MHz, and the signal over the bus is self-clocked. In recent years, there has been increased support for this standard by integrated circuit (IC) implementation. A typical example may be found in Ref. 8.

Table 1 IPL → DLL and DLL → IPL message alphabets for the PRD-MC network

IPL → DLL	DLL → IPL
Transmit request (message pointer)	Transmit done
	Excessive collision error
	Frame check error
Priority transmit request (message pointer) canceling request	
	Receive OK (message pointer)

Table 2 IPL→DLL and DLL→IPL message alphabets for Ethernet

IPL → DLL	DLL → IPL
Transmit request (message pointer)	Transmit done
	Excessive collision error
	Frame check error
	Receive OK (message pointer)

System sophistication increases continuously and the equipment now includes microprocessors to a greater extent than evisaged in 1978 and nonexistent in 1973. This allows introduction of new functional possibilities. But the deterministic central control of communication in MIL-STD-1553 does not match the stochastic, distributed approach of such HLLs as CSP and Ada. Nevertheless, by concentrating all status information and decisions at the bus controller, it was found possible to construct an elegant and efficient implementation of CSP and Ada protocols at the bus interface. Details of this solution will be presented elsewhere.

We hope to show that higher languages are the best tool for expressing the nondeterminism inherent in multiprocessing. Using intertask communication constructs and grouping elements together for implementation in hardware, a high degree of integration may be obtained.

At the system level, behavior should be deterministic, i.e., the sequence of computations should repeat every one or a multiple, well-defined, number of computing cycles. This is the macroscopic level of behavior. It should not preclude utilization of random access communication regimes, indeed, excluding the case when two tasks are ready to exchange information, which has a deterministic behavior. Random access schemes are simple, elegant, and do not need control messages for bus access.

On the other hand, deterministic system behavior should not disallow programmers to use, for example, Ada select statements which may lead to random selection of an alternative (from many which are ready). The CSP alternative command and Ada select statement allow writing of extremely compact and concise, easy-to-understand code. The conditions causing alternatives to be ready still may lead to deterministic behavior.

Inside a single computing cycle, for one HLL statement at the bus-access level, one has the microscopic, detailed, behavior which can be stochastic.

Microscopic stochastic behavior, under enough spare in bus bandwidth and computing time, may allow the macroscopic level to be deterministic.

Language-Specific Layers: IPL and KL

The IPL receives instructions from KL directly dependent on specific HLL communication constructs. The resulting messages over the bus are of two types: data—including the information exchanged by using Π_{ij}—and control, which request and acknowledge the data. Obviously, the data transfer is executed in conformity to the coordination rule given

Table 3 Message types

Name and mnemonic	Meaning when sent over the bus	Meaning when received from the bus
Input request (IREQ)	The process sending this signal is ready to receive a data message	The process sending the signal requests an input from the receiving process
Output request (OREQ)	The process sending this signal is ready to output a data message	The process sending this signal is ready to output a data message
Input acknowledge (IACK)	Sent over the bus by a process after receiving an IREQ (unmatched case)	Received to acknowledge a previously sent IREQ (unmatched case)
Output acknowledge (OACK)	Sent over the bus to acknowledge receiving an OREQ	Received to acknowledge a previously sent OREQ
Cancel input request (CIREQ)	Transmitted to cancel a previously sent IREQ	Instruction to cancel a previously received IREQ
Cancel output request (COREQ)	Previously sent OREQ	Previously received OREQ
Data message (DMSG)	Sent after receiving a matching IREQ	Received in response to a previous IREQ
Data acknowledge (DACK)	Sent in response to a received DMSG	Received in response to a DMSG

Table 4 Ada-based KL → IPL message alphabet

No.	Instruction	Possible IPL-KL answers[a]	Notes
1	evaluate (accept_statement)	1,3	Task resumed executing the **do**... **end** sequence
2	evaluate (selective_wait (no else, delay or terminate))	1,3,4	
3-5	evaluate (selective_wait (else, delay or terminate))	1,3,4	Answer to host caused by processing of own IPL
6	evaluate (entry_call)	2,3	
7	evaluate (conditional entry_call)	2,3,5	
8	evaluate (timed entry_call)	2,3,6	
9	end_RV	—	After execution of [**do**... **end**] sequence
10	abort (select_statement)	7	

[a]Per Table 5.

Table 5 Ada-based IPL → KL message alphabet

No.	Instruction	Notes
1	Start rendezvous	Received input parameters
2	End entry call	Received output parameters
3	Excessive collision error	
4	Selective wait not executable (else, delay, terminate, select error)	For selective wait with else, delay or terminate Answer provided by own IPL
5	Conditional entry call not executable	Answer provided by own IPL
6	Time entry call not executable	
7	Abort (granted, rejected)	

Table 6 CSP-based KL → IPL message alphabet

No.	Instruction	Possible IPL → KL answer[a]
1	execute (Input Command)	1,4
2	execute (Output Command)	2,4
3	execute (Alternative Command)	1,2,4
4	Cancel (Command)	3,4

[a]Per Table 7.

Table 7 CSP-based IPL → KL message alphabet

No.	Instruction
1	Executed (Input Command)
2	Executed (Output Command)
3	Cancel (granted, rejected)
4	Excessive Collision Error

earlier, which is required implicitly by the CSP and Ada communication constructs.

IPL therefore decides how to continue the protocols for information transfer, i.e., what message types should be requested and sent to DLL or KL. In Ref. 9, a unique set of message types was suggested. It proved to be adequate for CSP as well as Ada, and is reproduced in Table 3. The table summarizes the meaning of each message for two cases: transmission and reception over the bus. Decisions to send these message types are due to IPL internal processing executed by the bus interface.

Tables 4 and 5 are reproduced from Ref. 10 and indicate the MAs for KL → IPL and IPL → KL in the case of Ada. The whole Ada-oriented IPL protocol is given in Ref. 10, but for a complete understanding the reader should refer to Ref. 1 wherein the Ada communication constructs are described in detail.

The list of instructions, possible to be submitted by KL → IPL, is a direct mapping of the Ada constructs for intertask communication. The numbers shown in the third column indicate possible IPL answers for KL, which are shown in Table 5.

Shown in Tables 6 and 7 are the MAs for CSP. Although in this case three protocol variants were issued,[9] the resulting MAs are *unique*. More details on one CSP protocol variant can be found in Ref. 11, which presents its validation by means of temporal logic.

The first column shows commands submitted by the host to the bus interface once a CSP-based instruction is encountered during execution of the program. The first three instructions are a direct mapping of the CSP constructs for interprocess communication. The fourth instruction is for the case of alternative commands, containing requests for exchange of information with processes resident in the same and different hosts.

The first instruction in Table 7 indicates to the host (respectively to KL) that a data message was received. The next instruction relates that a data message was transmitted. The third instruction is the response to a canceling request submitted by KL. The cancel can be *granted* if data transfer related to the corresponding alternative command is not in progress or on the contrary, has been *rejected*.

It can easily be seen[9] that a unique set of control message types (see Table 3) suffices for both languages. What differs is the sequence of control messages interchanged over the bus to execute the rendezvous for Ada or the information transfer in the case of CSP.

For the IPL → DLL and DLL → IPL cases, what differs when executing an Ada or a CSP construct is the *object* of the instruction (in addition to source and destination fields, the specific field in the header indicating message type) and not the instruction itself. Each time a KL instruction is submitted to IPL, its object (a large part of which later makes up the control data message sent over the bus) is carefully analyzed. In most of the cases, the processing consists of searching for matching patterns among similar objects of instructions provided from DLL. Therefore, instruction objects are stored in IPL until the data transfer is executed. An example of processing a KL-submitted instruction is given subsequently in procedural form for the case of CSP under asymmetric protocol.[9]

```
Procedure KL_instructions:
begin
case KL_instruction of:
    execute (Input_Command):
        Transmit_Request (IREQ);
    execute (Output_Command):
        if match then Transmit_Request (DMSG)
        else store_entry
    evaluate (Alternative_Command):
        if matched_output_command
            then Transmit_Request (DMSG)
            else for k = 1 to All_input_Commands
                Transmit_Request (IREQ_k);
    Cancel (command):
        if command_not_in_progress
            then IPL → KL_message (cancel_granted)
            else IPL → KL_messages (cancel_abort);
end  (*case*)
end;
```

It is left as an exercise for the reader to formulate the rest of the IPL internal procedures for CSP in a manner similar to those given in Ref. 10 for Ada. More details regarding CSP protocol variants are presented in Ref. 9.

A Design Methodology

In addition to the simpler hierarchical partition, this paper proposes a new design methodology, which starts with the selection of a communication topology and regime plus a high-level language for distributed processing.

The five-layer partition given previously allows specification of the message alphabets for information exchange between layers. The next step is to decide upon the internal IPL processing given the previously specified DLL → IPL and KL → IPL message alphabets. To keep it simple and efficient, it is sometimes necessary to optimize the DLL rules as was done with Ethernet by creating PRD-MC: for example, the need for canceling a request to transmit a message follows from an understanding of the CSP communication constructs.

The stepwise refinement of internal processing specific to each layer takes into consideration reciprocal influence of other layers. The simpler the internal processing of the lower three layers, the easier it becomes to implement them in silicon.

A strong confirmation of the practicality of the authors' approach can be found in the "transputer,"[12] a microprocessor announced by INMOS Ltd. As a building block for two-dimensional processor arrays, it has four serial interfaces, each executing in hardware/firmware the communication protocols of Occam,§ a high-level language nearly identical to CSP. Thus the work presented herein, which is based on a common-bus configuration, is neatly complemented by that of INMOS Ltd.

Generality of the reference model is expressed by the multiple variants possible to be used for some of the hierarchical layers. IPL (and implicitly KL) may be programmed for Ada, CSP, or Occam. DLL and PHL can be adapted, for example, to MIL-STD-1553, as well as to other communication schemes.

Message alphabets should be specified for each case. However, the architecture proposed for IPL, appended to the bus interface circuits, is unique for all of the cases, and programmable.

Conclusions

It has been shown that Real-Time Cyclic Closed Systems can be characterized by a hierarchically layered concept, and a simple system design methodology is recommended. Our *system-oriented* approach, starting from the requirement to use a high-level language for distributed processing led to a considerably smaller number of layers—five, including that of the application—when compared with the classic *communication-oriented* concept due to Open Systems Interconnections, which has seven layers.

A higher degree of integration may be achieved by encapsulating the interprocess, data-link, and physical layer services into silicon, the activation of the circuit being effected by high-level-language constructs.

In addition to leaving the processors free to execute concrete applications, the approach illustrated herein brings as a major benefit, a considerable saving in design effort. The suggested peripheral integrated circuit should include all of its necessary procedural interfacing up to the kernel layer. Its utilization will be extremely simple, provided it is supported by adequate kernels for the processors in use, and by proper tools for bus communication, monitoring, debugging, and integrating. It appears that this system-oriented approach can be extended to other communication schemes (such as the token), other communication topologies (ring, star), and other (future) high-level languages for distributed processing.

Appendix A

Example: A Cross-Channel Data Link

Assuming a triple redundant computer system as used in flight control, information exchange between computers cannot be implemented by a single central controlled 1553 bus. The bus controller should be either another computer (Fig. A1a), a single-point failure, an unacceptable solution, or should reside in one of the redundant computers (Fig. A1b), again single-point failure, causing also disymmetry in the system. Assuming each computer broadcasts its own information to the other two over a dedicated bus, any computer should include three simple 1553-bus interfaces: two for receiving information and one to send it (Fig. A1c). Traffic over each bus is minimal broadcast of data. The penalty is extra hardware for bus interfacing and, assuming one uses conventional languages, explicit message scheduling for transmission. For a simple cross-channel data-link bus, this is not a complicated task, however, for more sophisticated systems it might be a serious design effort.

The suggested solution has two novel elements: it requires use of distributed processing HLL constructs for exchange of information, and a different bus regime with distributed communication control. Only the bus regime, i.e., the rules of the two lower layers of our hierarchical partition, replaces the 1553 protocol. For example, from a functional point of view, a single bus suffices, no supplementary computer for communication control being necessary (Fig. A1d).

However, buses are triplicated to keep the level of redundancy uniform; in each bus interface only transceivers are being replicated (Fig. A1e). However, the big advantage of the solution suggested is that a great number of operations are carried out under bus interface initiative without necessitating explicit scheduling. In the following lines we introduce Ada and CSP programs for information exchange between these computers. In principle, the program is all that the designer should provide. Once the kernel layer in each computer encounters a CSP or Ada communication construct, it submits it to the bus interface for evaluation—indeed, only if the exchange of information is not with a task in the same computer. Then, until data messages are transferred, all operations are carried on by the bus interface automatically.

n-tuple Computer Redundancy CSP

1) The straightforward solution is a single process in each of the n processors:

```
process(i: j..n) :: PROCESS
```

§Occam is a trademark of INMOS Ltd.

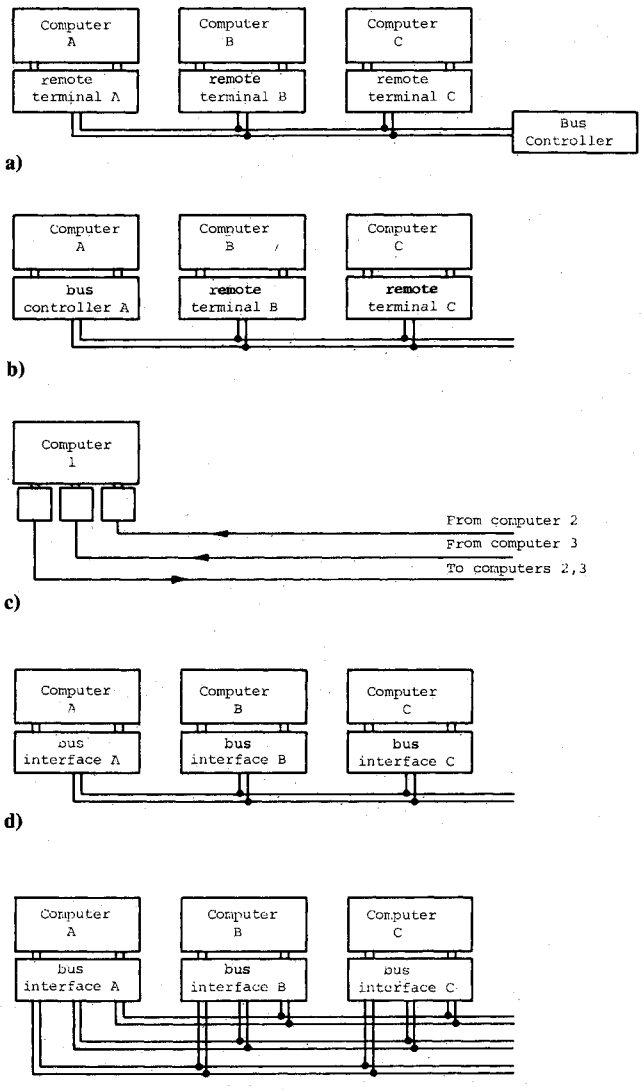

Fig. A1 Triple redundant computer system (typical for flight-control applications): a) 1553 Mux bus communication with a separate bus controller which may be a single-point failure. b) 1553 Mux bus communication with the bus controller resident in one redundant computer. An asymmetric solution. c) Triplicated simple 1553-bus interface. d) Communication scheme with distributed control for exchange of information. The common bus can constitute a single-point failure. e) Triplicating the buses require only three transceivers in each bus interface.

The process resident in the ith processor follows.

PROCESS = COMPUTE; result: (1..n) real;
timeWindow, senTo(k:1..n, $k \neq i$),
recFrom(k:1..n, $k \neq i$):boolean;
timeWindow: = 1; senTo: = 0; recFrom: = 0;
*[(k:1..n)$k \neq i$; senTo(k) = 0; process(k)!result(i) →
senTo(k) : = 1
□(k:1..n)$k \neq i$ recFrom(k) = 0; process(k)?result(k) →
recFrom(k) : = 1
□time, timeLimit:integer; timeWindow; time > timeLimit
→ senTo: = 1; recFrom: = 1; timeWindow: = 0
];VOTE.

2) If communication time is significant and computation time may vary widely among processors, then we may improve speed by adding two processes in each processor: "resCol" for collecting results from the other processors and "resDistr" for distributing the local result. The last processor to complete its computation should find that "resCol" has already collected all of the results from the other processors, and should go on to execute the voting while "resDistr" distributes the local result.

Table A1

	Dialogs	Messages
Single process solution:		
Order of computation completion		
Best	18	42
Worst	26	58
Three-process solution	36	72
Five-process solution	24	48

mainProc(i:1..n)::MAINPROC
||resCol(i:1..n)::RESCOL
||resDistr(i:1..n)::RESDISTR

The processes residing in the ith processor are:

MAINPROC = result:(1..n)real; otheRes:(1..(i − 1)
(i + 1)..n)real;
COMPUTE; resDistr(i)!result(i);
resCol? otheRes; VOTE.

RESCOL = recTimeWindow,
recFrom(k:1..n, $k \neq i$):boolean;
recTimeWindow: = 1; recFrom: = 0;
*[result:(1..n)real;(k:1..n)$k \neq i$; recFrom(k) = 0;
recDistr(k)?result(k) → recFrom(k) : = 1;
□time, recLimit:integer; recTimeWindow; time >
recLimit → recFrom: = 1; recTimeWindow: = 0];
otheRes:(1..(i = 1)(i + 1)..n)real; mainProc(i)!otheRes.

RESDISTR = result:(1..n)real; mainProc(i)? result(i);
distrTimeWindow, senTo(k:1..n, $k \neq i$) boolean;
distrTimeWindow: = 1; senTo: = 0;
*[(k:1..n)$k \neq i$; senTo(k) = 0; resCol(k)! result(i) →
senTo(k) : = 1
□time, distrLimit:integer; distrTimeWindow; time >
distrLimit → senTo: = 1; distrTimeWindow: = 0].

3) To reduce the communication load we might consider partitioning "resCol" into (n − 1) processes, one for each result to be collected. The load imposed on the common bus by each of the above solutions at n = 4 is given in Table A1.

Table A1 is true for staggered completion. But if computation completion does not vary significantly between processors, and result collection is enabled at proper time, communication will tend to a minimum in all solutions, approaching 36 messages in 12 dialogs.

Program Written in Ada

Four tasks are resident in each computer:

—One task to receive data from the two other computers A_RECV_DATA, B_RECV_DATA, and C_RECV_DATA, each including the entry READ.

—Two tasks in each computer to send data to the two others:

A_SEND_DATA_TO_B, A_SEND_DATA_TO_C,
B_SEND_DATA_TO_C, B_SEND_DATA_TO_A,
C_SEND_DATA_TO_A, C_SEND_DATA_TO_B,

each using the entry call, respectively:

B_RECV_DATA.READ,
C_RECV_DATA.READ,
C_RECV_DATA.READ,
A_RECV_DATA.READ,
A_RECV_DATA.READ,
B_RECV_DATA.READ.

```
—One task to execute the vote VOTER including an entry
INFORM_VOTER. This entry is called by A_RECV_DATA,
B_RECV_DATA, and C_RECV_DATA such that
A_RECV_DATA has rendezvous with VOTER resident in A,
B_RECV_DATA has rendezvous with VOTER resident in B,
and C_RECV_DATA has rendezvous with VOTER resident
in C.

Task A_RECV_DATA is                          --resident in A
    --to receive data from B and C
        entry READ (M: in MESSAGE);
end;
Task body A_RECV_DATA is
---
begin
  ---
  loop
    select
      when FROM_B = False or FROM_C = False ⇒
      accept READ (M: MESSAGE)]
        do
          if data_from_B then FROM_B := True; end if;
          if data_from_C then FROM_C := True; end if;
          VOTER.INFORM_VOTER;
          if FROM_B and FROM_C then exit; end if;
        end;
    or
        delay TIMEOUT;
        VOTER.INFORM_VOTER
                                  --send a message to voter
        exit;
    end select
  end loop
  ---
end

Task Voter is                              --resident in A, B and C
    ---
        entry INFORM_VOTER;
end;
Task body VOTER is                           --communicates only
    k := 0;                                        with local tasks
loop
  select
    when k < 2                          --k = 2 after two timeouts or
    accept INFORM_VOTER                 after info is recvd from B,C
        do... k := k + 1 end;
      else exit                                      --end loop
  end select
end loop
VOTING;
end;
```

For data transmission we show the tasks resident in B:

```
Task B_SEND_DATA_TO_A is
    ---
end

Task body B_SEND_DATA_TO_A is
---
  loop
    select
      A——RECV_DATA.READ(M:message);
      exit;
    else
      delay TIMEOUT_B_TO_A;
      exit
    end select
  end loop;          --end the loop after one data transmission
---                                              or after timeout
end

Task B_SEND_DATA_TO_C is
    ---
end

Task body B_SEND_DATA_TO_C is
---
  loop
    select
      C_RECV_DATa.READ(M:message);
      exit
    else
      delay TIMEOUT_C_TO_A;
      exit;
    end select
  end loop;
---
end;
```

For an asymmetric protocol with broadcast of IREQs for each accept statement, the bus interface servicing computer A will transmit over the bus (broadcast) an IREQ once accept READ is submitted.

When B submits to its interface the entry call A_RECV_DATA_READ, match occurs and DMSG is sent from B to A. The two tasks B_SEND_DATA_TO_A in computer B and A_RECV_DATA in computer A resume after data transfer takes place and rendezvous terminates.

Appendix B

High-Level Language Constructs for Intertask Communication

CSP

The CSP language[2] does not allow shared variables. Processes communicate with one another if each names the second one as its partner. The basic constructs are *input* and *output commands* of the form:

$A?x$ receive a value from process A and assign it to variable x

$A!x$ send a value x to process A

Communication takes place when the following conditions hold:

1) An input command in one process specifies the other process as its source. The process suspends its activity and resumes it only after the communication has taken place.

2) An output command in the other process specifies the first process as its destination. The process suspends its activity and resumes it only after the communication has taken place.

3) When the communication takes place the target variable of the input command matches the value denoted by the expression of the output command.

The main construct we deal with is the *alternative command*. It major constituent is the *guarded command*—a variation of that proposed by Dijkstra.[13] It has the form $G \rightarrow C$, G being the *guard*, i.e., a list of Boolean expressions with at most one input command. C is a command list. We assume that guards may include input as well as output commands. An I/O command in a guard is attempted only if all Booleans are true. The alternative command consists of several guarded commands to express the random behavior of distributed processes. When multiple choice is possible, only one command is selected; otherwise, selection is on a first come, first served rule. First, one executes the communication, then the command list associated with the selected guard. If more than one guard is found true, the guard selection may be done at random or according to some predetermined rule. For exam-

ple, a data collection process is expressed as follows:

[$sensor_1$?a → $command\_list_1$

□$sensor_2$?a → $command\_list_2$

];

The symbol □ separates the two alternatives. The process waits until either $sensor_1$ or $sensor_2$ is ready to transfer information; when both are ready, only one is selected at random.

Note that execution of the I/O command requires *explicit naming* of one process by the other.

Intertask Communication in Ada

Entry Mechanism—Accept Statement

> "**Entry**. An entry is used for communication between tasks. Externally, an entry is called just as a subprogram is called; its internal behavior is specified by one or more accept statements specifying the actions to be performed when the entry is called."
>
> Ref. 4, p. 272

The formal definition is given below:

```
entry_declaration :: =
    entry_identifier[(discrete_range)] [formal part];

entry_call_statement :: = entry_name
                          [actual_parameter_part];

accept_statement :: =
    accept entry_simple_name
        [(entry_index) [formal_part] [do
        sequence_of_statements
    end [entry_simple_name]];

entry_index :: = expression
```

The syntax of the sequence of statements included in the **do**...**end** sequence, is similar to that of a procedure. The main difference is that these are executed by the called process, not always immediately after the call. The coordination between an entry call in a calling task and an accept statement in a called task is referred to as *rendezvous*.

During rendezvous, the statements between **do**...**end** are executed by the called task, in mutual exclusion with the caller. Only after this execution is complete may the caller task (T2 in our case) resume its activity. Several tasks may call the same entry. An accept statement may contain other accept statements; nesting is allowed and a talk may include more than a single accept statement for the same or different entries. Each called task is associated with every entry name it owns, a unique queue of waiting entry calls, processed in the order of their arrival.

For the time that may pass between the occurrence of an entry call and the execution of the accept statement, it is required that parameter values of the caller shall not be modified by any other tasks. This condition has the effect of disallowing the use of entry-call parameters as common variables.

If an accept statement is reached for an entry and the queue is empty, the entry status is set to *open* and the task must wait for a call. If the queue is not empty, parameters are transferred to the executing task and rendezvous starts. At the end of the rendezvous, both partners resume their execution, in particular the caller is removed from the queue, and receives the output parameters.

Select Statement

The select statement allows three types of possible executions: selective waiting, conditional entry call, and timed entry call:

```
Select_statement :: = selective_wait
    | conditional_entry_call | timed_entry_call
```

Selective Wait

The formal definition is given below:

```
selective wait :: =
    select
        select_alternative
    {or
        select_alternative}
    {else
        sequence_of_statements}
    end_select;

select_alternative :: =
    [when condition ⇒]
        selective_wait_alternative

selective_wait_alternative :: = accept_alternative
    | delay_alternative | terminate_alternative

accept_alternative :: = accept_statement
        {sequence_of_statements}

delay_alternative :: = delay_statement
        {sequence_of_statements}

terminate_alternative :: = terminate
```

If when entering the select statement no entry has been called, the task waits for the first call. Then, the corresponding accept statement is executed. If one entry has already been called, the call is accepted, but if more than one entry has been called, the selection is random.

Arbitrary selection occurs when either 1) several entries related to the same select statement were called before the statement is encountered, or 2) several open alternatives of the same select statement start with an accept statement for the same entry. Once arbitrary selection occurs, the rest of the pending entry calls are not canceled but will be serviced at the execution of the next loop.

Else, delay, and **terminate** are mutually exclusive and cannot appear simultaneously in the same statement.

An alternative is called *open* if the **when** clause is missing, or if the corresponding condition is true. Otherwise, the alternative is called *closed*. First, the conditions specified after **when** are evaluated. Only open accept alternatives are considered, the selection of one of them taking place immediately, if the rendezvous is possible.

An open alternative with an accept statement is selected only if the corresponding entry call has been issued by another task, i.e., the rendezvous is possible. The **else** part is selected if no accept alternative can be immediately available, particularly, if all alternatives are closed. If the else part is missing, the task waits until an open alternative appears. An open alternative starting with a delay is taken if no other alternative has been selected before the specified time interval elapsed.

Conditional Entry Call

> "Conditional entry call issues an entry call that is then canceled, if the rendezvous is not immediately possible."
>
> Ref. 4, p. 156

```
conditional_entry_call :: =
    select
        entry_call_statement
        {sequence_of_statements}
    else
        sequence_of_statements
    end select;
```

Timed Entry Call

```
timed_entry_call :: =
    select
        entry_call sequence_of_statements
    or
        delay_statement sequence_of_statements
    end_select;
```

The delay acts as time-out: at the end of the time interval if rendezvous has started already it is completed; otherwise, it is aborted and the sequence of statements following the delay statement is executed.

Acknowledgments

The authors are grateful to Professor Amir Pnueli for many thought-provoking discussions. Useful remarks and suggestions were received from Mr. Stephen S. Osder and from the anonymous referees. Special thanks are due to C. Weintraub for T_EXing the paper, and to Y. Barbut for the graphical work.

References

[1]Metcalfe, M.R. and Boggs, D., "Ethernet: Distributed Packet Switching for Local Computer Networks," *Communications of the ACM*, Vol. 19, July 1977, pp. 395-404.

[2]Rosemberg, F. and Ruhman, S., "Prioritized Dialogue in CSMA/CD Networks with Provision for Message Cancellation," *Proceedings of the 8th Conference on Local Computer Networks*, Minneapolis, MN, Oct. 1983, pp. 47-54.

[3]Hoare, C.A.R., "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21, Aug. 1978, pp. 666-677.

[4]"The Programming Language ADA Reference Manual," ANSI/MIL-STD-1815, 1983.

[5]Zimmerman, H., "OSI Reference Model—The ISO Model of Architecture for Open Systems Interconnection," *IEEE Transactions on Communications*, Vol. comm-28, April 1980, pp. 425-431.

[6]"The Ethernet, A Local Area Network: Data Link Layer and Physical Layer Specifications," Digital Equipment Corp., Intel Corp., Xerox Corp., Ver. 1.0, Sept. 30, 1980.

[7]"Aircraft Internal Time Division Command Response Multiplex Data Bus," MIL-STD-1553B, 1978.

[8]Williams, D.G., "Industrial Controller Joins the MIL-STD 1553 Bus," *Electronic Design*, Oct. 14, 1982, pp. 205-211.

[9]Rosemberg, F., Pnueli, A., Ruhman, S., and Ron, D., "CSP and ADA Protocols for Intertask Communication Dedicated to Common Bus Systems," *Proceedings of the 14th Convention of Electrical and Electronics Engineers in Israel*, Tel Aviv, March 1985, 3.2.1-1 to 3.2.1-4.

[10]Rosemberg, F., "An Ada Oriented Protocol for Intertask Communication in Real Time Common Bus Systems," *The International Seminar on Computer Networking and Performance Evaluation*, Tokyo, Sept. 1985, 11.2.1–11.2.17.

[11]Ron, D., Rosemberg, F., and Pnueli, A., "A Hardware Implementation of the CSP Primitives and its Verification," *11th International Colloquium on Automata Languages and Programming (ICALP)*, July 1984, Antwerp, Belgium; also, *Lecture Notes in Computer Science*, No. 172, Springer-Verlag, pp. 423-435.

[12]*IMS T424 Transputer Reference Manual*, INMOS Ltd., Bristol, England, UK, Nov. 1984.

[13]Dijkstra, E.W., "Guarded Commands, Nondeterminancy and Formal Derivations of Programs," *Communications of the ACM*, Vol. 18, Aug. 1975, pp. 453-457.