SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 10, 21–28 (1980)

A Heterogeneous Dual Processor

T. A. MARSLAND AND S. F. SUTPHEN
Computing Science Department, University of Alberta, Edmonton, Canada

SUMMARY

Details of a computer–computer communication facility are presented. The primary feature of the system is the use of a single terminal to monitor and control processes on different machines simultaneously, working together on the solution of a common problem. Experience with an application, error handling and maintenance of synchronization are described.

KEY WORDS Distributed computing network Concurrent processes

INTRODUCTION

The linking of a local minicomputer system with a remote major computing facility is an increasing widespread practice.[1] In most instances the local machine is used primarily as a file transport and job entry station,[2, 3] with relatively little attention being given to the investigation of the additional potential of such a combination. Our aim in the present paper is to demonstrate, if one is prepared to look beyond the obvious basic functions, one of the possibilities which occur: the design of a system which enables a single terminal to control processes executing concurrently in both local and remote computers. The two machines work together on a single problem, communicating and maintaining synchronization by a direct exchange of messages.

One important aspect of this work is the use of existing computer communication facilities, of a type generally available, to produce a network of diverse computers. Some other studies of interprocessor communication have been based on special networks of identical computers.[4] Although many of the problems discussed there are relevant to our situation, for instance the partitioning of jobs for parallel processing, the use of general networks (i.e. those containing a variety of computers) raises many additional questions. The computers will probably have radically different architectures, so certain design choices must be made, including the selection of an appropriate device for communication.[5] Because the problems we faced are typical of those encountered in the implementation of interprocess-communication, we have discussed our choices in some detail. In recognition of practical realities, we have also placed an extra constraint on our work, by requiring that all support software be installed on only a few of the computers in the network.

Operating systems

For the purposes of this discussion we shall classify our machines as 'local' and 'remote' computers. These words are not necessarily intended to reflect their physical location relative to the user's terminal, but rather to the degree of control and access level to the operating system residing on those machines. On the local machine full

0038–0644/80/0110–0021$01.00 *Received 14 September 1978*

control is assumed, while low-level access to the remote computer may be impractical for technical or priority reasons, since it will generally be part of some service bureau. At other establishments full control may exist over both (all) systems and a distributed-processing operating system may be feasible.[5]

The communication link between the two machines may be indirect through a number of store-and-forward communication processors, which may or may not be transparent to the data being exchanged, a rarely considered factor. The term 'Communication Processor' (CP) is used quite generally to encompass any machine through which data must flow to its destination. This implies that a remote computer may become a CP if it is commanded to make an external link (e.g. via DATAPAC[6] to another machine. More typically a CP is some sort of 'front-end' for a remote computer, and provides special support for the interactive and other i/o devices attached.

Reliability

In the fundamental file transport problem, error-free communication is more important than speed. A user-level protocol, which exchanges packets of information, can be constructed to achieve the necessary reliability. Where hardware support is available a sophisticated protocol such as X.25 could be used, but it is not essential. For specialized applications like ours, simpler protocols have been proposed, of which the variation we are developing[7] is typical. Even so, the level of support is quite substantial. Messages are bound by header and trailer records to form packets. Typically the header includes packet sequence number and length fields, and the trailer contains a redundancy check character for error detection. An acknowledgement scheme is necessary in order that corrupted packets can be re-sent. Futhermore, to guard against lost replies, the sender must receive an acknowledgement within some specified time interval, otherwise the system may deadlock with both sender and receiver expecting the other to respond.

The packet approach is very attractive but the overhead is fairly high because source and destination addresses may be needed, together with some additional signals to frame the packet. Thus in order to transmit 'raw' (binary) data it is better if these packet delimiters and other control signals are non-standard. In bit-serial transmission, for example, unique control signals can be generated by bit stuffing, thus keeping the communication protocol isolated from the data being sent. In store-and-forward networks extra error conditions can arise. A packet acknowledgement may be lost for example, causing retransmission of a previously forwarded packet and the potential for packet duplication. More seriously, safe arrival of a packet at an intervening node may be acknowledged, but the node may be 'overrun' or malfunction before the packet is forwarded. Unless there is also an end-to-end acknowledgement, the sender may erroneously believe his packet reached its destination safely.

MINICOMPUTER SUPPORT

Having surveyed briefly some of the problems that are handled by protocols for packet transmission, in which the network is conceived as a whole and consistent software is developed for all the intervening CPs, let us look again at the more common case in which extensions are made to an existing partial network to provide new capabilities.

Typically, as at the University of Alberta (Figure 1), the existing facility consists of a simple star network of local terminals using asynchronous character transmission, with an external link to a commercial network to support file transfer and job execution functions at other remote centers. With the passage of time, support for local minicomputers has become necessary. This support can be provided in a variety of ways, and at considerable cost depending on the expected quality of service. Ideally one would like to use as much of the network's existing capability as possible, but a protocol providing asynchronous computer to computer communication is needed.

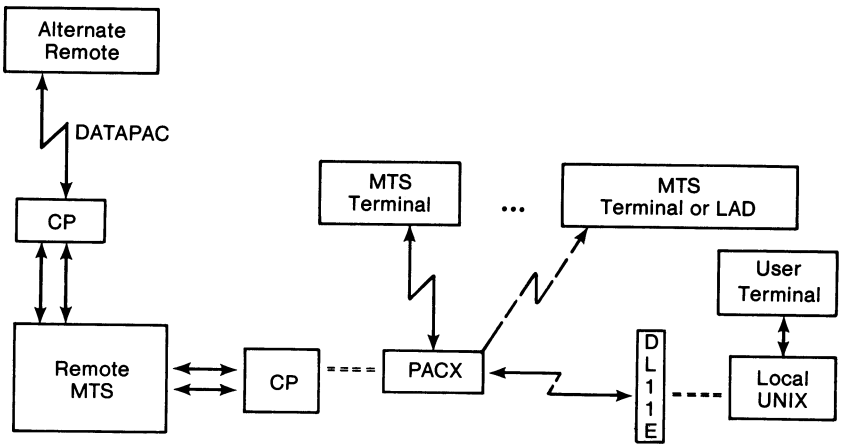

Figure 1. Basic structure of distributed computing system

Despite the fact that packet protocol is well suited to computer–computer communication needs, there is pressure to minimize costs and make use of the existing support for interactive and passive devices, and do without a formal protocol. At our installation two choices are offered. Either make your local computer behave like a simple terminal, or use the support for a 'local attached device' (a LAD), wherein the local mini behaves like a passive sequential device. Neither proposal is attractive, although each has some positive features. The support for LADs, for instance, can be used in such a way that the transmitted data is transparent to the intervening CP. Thus executable code generated by cross compilers on the central computer can be transmitted without incurring any transformations (such as EBCDIC–ASCII conversion), and without the data being interpreted as a CP command. If set up properly the LAD may also be able to accept *all* the data as input. Clearly a precise count must be kept of the data being transferred, for no signal indicating 'end-of-file' can be recognized. Similarly, no 'end-of-line' indicator is possible so fixed size blocks of data must be moved. Despite the positive features of this approach a number of disadvantages occurred to us. Should a transmission error actually occur some manual intervention to reinitialize the LAD will be necessary, before an attempt to retransmit the data is made. This is quite unacceptable. Furthermore, the approach requires two independent communication lines to the remote center, one for the LAD and one for the user's console to initiate the transmissions. Since this approach is not well suited to our dual interacting processor application it was rejected in favour of making the local computer behave like a terminal, and will not be discussed further despite its utility in other applications.

File transport

What then are the problems with making the local computer behave like a terminal? The COMLNK[2] and COM360[3] systems, both of which interconnected PDP11 and S370 computers, avoided this issue in different ways. The former group used a high-speed 16-bit parallel line directly onto the PDP11 unibus to achieve direct memory transfers. The COM360 system, on the other hand, used a standard telecommunications interface and a modem. Both groups employed a kind of packet protocol and relied on user mode programs executing simultaneously in the two machines to process the file transfer, and both assumed that the preferred direction of transfer was from the local PDP11 to the central machine. COMLNK handled everything on a file basis with a rather massive header to open the i/o operation and a trailer to close. The data packets were very large, often big enough to accommodate a whole file. A file basis protocol has many advantages, including low overhead under error-free conditions, but may require long periods of interrupt-free processing. Conversely, the COM360 scheme built user level packets, one for each line of the file, and awaited a positive or negative acknowledgement before deciding what to do next. The protocol consisted of a header specifying the message type, followed by either a function code or the message plus a sequence number and length. A positive acknowledgement was sent for every message, and concurrent send/receive operations were not supported. Their telecommunications access method also posed some problems. Since it would accept only a subset of the ASCII character set, input from the local computer was treated as a continuous bit stream, broken into groups of six bits to form a transmittable pseudo-character. The receiving station rebuilt the original message, as a consequence the effective transfer rate was significantly reduced.

The methods used in these earlier systems are not readily applicable to the current situation in which major computing centers front-end their machines with programmable CPs, to multiplex data from a large number of terminals. Problems with control of and interference from these CPs is made more difficult by our restriction that low-level access exist only at the local computer. The situation is further complicated by the fact that it may not be possible to identify the originator of such low-level directives as *Please signoff, disk failure imminent,* and distinguish them from a variety of advisory broadcasts.

Communication processors

CPs are usually contemporary 16-bit programmable minicomputers whose job it is to keep track of the special characteristics of each individual terminal, such as physical line length, speed of carriage return, line or character mode, tabbing positions, translation required and so on. Our CP serves as a store-and-forward node and supports up to 128 terminals over dial-up and dedicated lines (through a PACX) at speeds to 4,800 baud. At peak traffic times unacceptably high error rates occur. Although these errors are signalled as transmission errors it would seem that their appearance is much more a consequence of overruns in the CP software. This means that even if dedicated microprocessors are attached to both ends of the communication line, to handle a packet protocol, a reduction in apparent transmission errors might not occur. In many respects this is unfortunate since cheap hardware currently exists to support serial-bit transmission with automatic cyclic redundancy check generation, and bit insertion to provide control signals (e.g. Signetics Multi-Protocol Communications Circuit 2652).

Our experience has been that the quality of transmission over dedicated communication lines is excellent. Nevertheless, some flow control protocol to maintain synchronization and prevent one computer from overrunning the other is necessary. In our system the available mechanisms for flow control are rudimentary.[8] The CP normally discards characters when it is not ready for them (as would be the case if the remote host—MTS—does not have a 'read' pending). A paper tape convention is followed in which our CP informs the sender that it is ready to accept a new line of characters by transmitting an XON. Reverse flow control is accomplished by setting page mode on, with the page length set to a single line. In this mode the CP waits, after sending each line, for a character from the receiver before it sends the next line. These flow control mechanisms have been known to fail in our CP during peak periods because the XON was sent prematurely. This has caused us to insert deadlock detection and correction code in our device driver. The situation is not satisfactory as data could be lost, but under our initial restriction that no direct control be exercised over the CP software, we must be content to wait until our needs and the service bureau's priorities are equivalenced.

There are also problems in recognizing the sources of various advisory and warning messages in the system. These include error messages such as *⟨file⟩ does not exist, enter replacement or cancel*, as a consequence of user program actions; CP messages like *Processor failure, DATAPAC reloaded*; and user–user communications. CP error messages are particularly troublesome, since subsequent restart procedures may change the state of the CP and its view of the terminals it manages. Consequently, any computer which is trying to model the responses of a user under these hostile conditions is itself transformed into a questionable state with no clear course of action. If it keeps a table of all the known CP error messages it must also be kept informed of any revisions and extensions. No general solution to this problem is possible so long as sotfware control cannot be exercised at all intervening CPs in the network. Normally such control is infeasible.

DISTRIBUTED COMPUTING APPLICATIONS

With all these considerations in mind, and despite the fact that some of the established techniques and hardware were not available to us, we have built and used a distributed dual processor system which allows processes executing in two different computers to work together on a common problem. The two systems in question are UNIX[9] on a PDP11/45 and MTS[10] on an Amdahl 470 V/6. The primary aim of our work is to provide a system in which a single terminal simultaneously controls processes executing on both local and remote computers. Thus the most desirable features of both operating systems could be used in the solution of a single problem. Since input or output for one computer may reside on the other, our normal mode of operation allows simultaneous access to files on both machines. This achieved through a logical extension of the communication pipe provided under UNIX. Naturally a file transfer capability is available but, because of extreme differences between the UNIX and MTS file systems, some processing conventions have been established. For instance, UNIX maintains only sequential files in which logical records are separated by a 'new line' character, and significant file compression is achieved through the use of TAB characters. In some respects the MTS file system is more elaborate, supporting random access line-oriented files. In such files each logical record is associated with a unique

index and can be accessed independently. Because of our initial constraint that the additional software to support this project be installed on a local computer, all the filters to handle these mappings run under UNIX.

The system which handles the message exchange between the two computers is represented by process 'A' of Figure 2. A detailed description of that process is given in our report.[8] Physically the user's terminal is attached to UNIX and has full access to that machine. Once process 'A' is invoked and the link to the remote computer (MTS) established, then the capabilities of that system can also be used. A special prefix

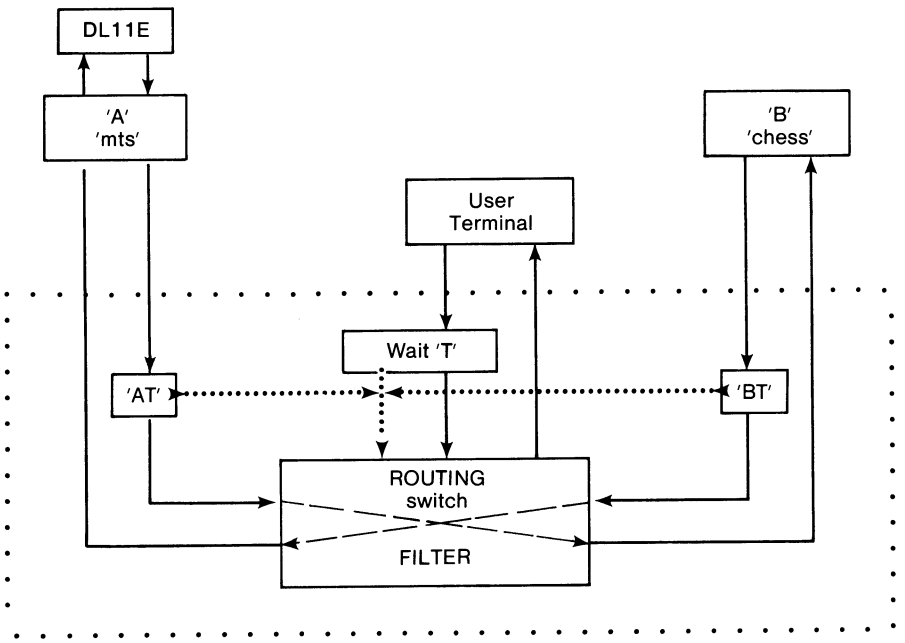

Figure 2. Process and pipe diagram for local computer

character (!) enables an escape to UNIX, thus commands to the two systems can be interlaced and overlapped. Details of our file transport mechanism, line connection routine and device driver are given elsewhere,[8] as are the description of the timeout mechanism to avoid deadlock, and the transmission error recognition scheme. Rather than elaborating these techniques, which do not have broad appeal, experiences with an application of our system are presented. In particular, process 'A' is used to establish communication and exchange messages between processes executing on local and remote computers, so that these two machines can work together on a common problem. Input to, and control of, these machines is achieved from a single terminal, which also receives a summary of the output produced.

Example

The six processes and logical connections necessary to support a sample application—two computers playing chess—are illustrated in Figure 2. Of the four components in the large box, the primary parent process, here designated the 'route-switch-filter', supports three children—the virtual terminals 'AT', 'T' and 'BT'. Processes 'AT' and 'BT' in turn are the parents of the MTS interface process 'A', and the UNIX application program 'chess' (process 'B') respectively. The routing function

redirects the data transfers to the appropriate destination. All three virtual terminal processes communicate with the parent in essentially the same way, by having the parent use the 'ptrace()' function to catch the program interrupts of its children. Typically a message comes from the remote computer via the DL11E interface to process 'A' and its virtual terminal 'AT'. These input characters are placed into a pipe directed to the switch and, upon logical termination of the input stream (carriage return), 'AT' generates a program interrupt signal which awakens its parent process to accept the data. This data is then filtered (reformatted) as required by the application and routed to either the user terminal or forwarded to process 'B'. The virtual terminals are normally waiting on a 'read()' from their child process, while the parent hangs on a 'wait()' for a status change (program interrupt) of its children. By this means the system incurs negligible overhead, is self-synchronizing and deadlock free.

Messages from the user's terminal to the processes executing on MTS and UNIX are distinguished via a prefix character. More importantly *attention interrupts* to these systems must be handled. MTS makes heavy use of this interrupt not only to suspend execution of a task but also to allow one to return to MTS command mode for a while before restarting. Under UNIX this interrupt (DEL) is used more commonly to terminate a task. We have found it quite convenient to use a common key to initiate this interrupt, even though the ultimate destination is apparently not clear. In practice the bulk of the support processes execute with this interrupt disabled, while application programs are specifically programmed to respond appropriately, as illustrated later.

In or chess example the two machines are working alternately, each computing and preparing output destined as input for the other. This process models an execution monitor developed earlier which was used to perform experiments with pairs of chess programs that executed on the same computer.[11] The current approach allows us to generalize the earlier one, providing a facility in which the two machines may work together each handling a different aspect of a common problem. Thus at negligible cost a heterogeneous dual processor system has been produced.

Results

The tests on the system have been so successful that the two machines are routinely controlled from a terminal communicating over a dial-up phone line, and physically remote from both computers. The processes are initialized from the terminal and the line from UNIX is opened automatically and attached to MTS as an i/o device. After the initialization phase the transactions can be echoed at the terminal, and control can be regained through the use of an attention interrupt. Since there is only one real terminal this interrupt is caught by the virtual terminals which in turn alert the application processes on MTS and UNIX. Therefore a simple convention routes messages to either machine. This mechanism is vital in order to recover from situations in which one process enters a CPU bound loop or produces excessive amounts of output, or to re-establish message synchronization should a transient CP malfunction occur.

Re-establishing the flow control protocol in any system is potentially troublesome, and is very installation dependent. For example, the state variables which describe the properties of our CPs may be sent and reset in three distinct ways. The commonest is by issuing 'device commands' from the terminal to the CP; the easiest is indirectly (through the back door) via a command from the remote computer; and lastly the CP may restore itself to some default state after a malfunction or special condition. These

special conditions include, for example, the receipt of an attention interrupt, which causes message echoing to be re-enabled. This occurrence is for historical reasons so that our CP can model an established MTS practice. Our provisional solution is to force the CP into a known state, since the mechanism to interrogate its state is awkward to use. Clearly many of these problems can be handled more easily with a proper host–host protocol.

Conclusion

Aside from experimenting with the computer–computer communication problem, the initial use of our dual processor system is to develop a chess program which is partitioned to reside on both machines. Each part can take advantage of local data bases and work concurrently on different aspects of the problem. For example, one will select goals which may be attainable, while the other will seek paths to attain those goals. However, our distributed computing system is not application dependent, but clearly is only appropriate for those problems which *can* be partitioned into parallel processes. It does, however, serve the useful function of allowing a single terminal direct access to two independent systems, so that one can switch quickly and easily between them.

ACKNOWLEDGEMENT

Financial assistance for the purchase of equipment was obtained from the National Research Council of Canada Grant A7902.

REFERENCES

1. A. G. Fraser, 'SPIDER—A Data Communication Experiment', *Comp. Sci. Tech. Rep.* No. 23, Bell Labs, Murray Hill, N.J., U.S.A.
2. J. van den Bos and H-J. Thomassen, 'COMLNK—a File Transport and Job Entry Utility for a Communication Link', *Software—Practice and Experience,* **7**, 173–177 (1977).
3. K. W. Colby, 'Sharing Minicomputer Data with a Full Size Computer Complex', *MINI 75—Proc. of the Int. Symp. on Mini- and Microcomputers and their Applications,* ACTA Press, Calgary, Canada, 124–127 (1975).
4. E. G. Manning and R. W. Peebles, 'A homogeneous network for data sharing communications', *Tech. Rept. GCNG–E–12, Univ. Waterloo, Canada* (1974).
5. R. Pardo, M. T. Liu and G. A. Babic, 'An N-process Communication Protocol for Distributed Processing', *Proc. Symp. on Computer Network Protocols, Liege, Belgium,* pp. D7–D10 (1978).
6. The Computer Communications Group, *DATAPAC—Standard Network Access Protocol Specification,* Trans-Canada Telephone System, 1976.
7. M. S. Sloman, B. K. Penny and T. A. Marsland. 'A Communication Protocol for Distributed Microprocessors', *IEE Int. Conf. on Distributed Control Systems, Birmingham, UK,* 64–68 (1977).
8. S. F. Sutphen and T. A. Marsland, 'A Heterogeneous Dual Processor Using MTS and UNIX', *Tech. Rep. TR78–2, Computing Science Dept., Univ. Alberta* (1978).
9. D. Ritchie and K. Thompson, 'The UNIX time-sharing system', *Bell Sys. T. J.* **57**, 6, Part 2 1905–1930 (1978).
10. D. W. Boettner and M. A. Alexander, 'The Michigan Terminal System', *Proc. IEEE,* **63**, 912–918 (1975).
11. T. A. Marsland and P. G. Rushton, 'Mechanisms for comparing chess programs', *Proc. ACM73,* 202–205, Atlanta, U.S.A. (1973).