Messages vs. Remote Procedures is a False Dichotomy

Michael Lee Scott

Department of Computer Sciences
University of Wisconsin - Madison
1210 W. Dayton
Madison, WI 53706

ABSTRACT

This paper discusses some of the major design decisions that distinguish recent language proposals for concurrent programming. It is argued that the classification of languages as "procedure-based" or "message-based" is misleading, in that it confuses two independent issues. It is further argued that these issues are best left undecided by the language designer.

1. Introduction

The last few years have seen the creation of a large number of high-level languages for concurrent programming. Since the publishing of Lauer and Needham's paper "On the Duality of Operating System Structures"[8] it has been fashionable to divide such languages into two categories: **procedure-based languages** and **message-based languages.** Unfortunately, at least *two* questions must be asked to determine the category in which a particular language belongs:

1) Does the sender of a message continue execution immediately, or does it wait for a reply?
2) Are messages received explicitly by active processes, or does their arrival automatically trigger the execution of some specified body of code?

The first of these questions involves the choice of a synchronization mechanism. The second is really a matter of syntax. I intend to show that the two questions are in fact *independent;* they give rise to *four* combinations of answers, not two. Moreover, there are applications in which each of the combinations is the method of choice, so a reasonable language might need to provide more than one. I will return to these claims in sections 4 and 5 after first discussing the issues of synchronization and message receipt in greater detail.

2. Synchronization

Much recent effort has been devoted to the study of **multi-computers,** in which separate processors have no memory in common. On such machines, all interaction between processes must ultimately be achieved by means of messages. Because of this limitation, synchronization is subsumed in the semantics of the send operation.

This work was supported by NSF grant number MCS-8105904 and by Arpa contract number N0014/82/C/2087.

Liskov[9] has suggested three synchronization mechanisms. These are

1) No-Wait Send: After sending a message, a process continues execution immediately.
2) Synchronization Send: A sender waits until its message has been received.
3) Remote Invocation Send: A sender waits until *it* receives a reply from the *receiver*.

Only remote invocation send deserves the title 'procedure based.' Of Liskov's three alternatives, it is the only one in which the sending of a message is designed to bring about the execution of a remote operation — a procedure — on the sender's behalf. The other two options are clearly 'message-based;' they simply transfer information. Under this nomenclature, Brinch Hansen's Distributed Processes[1] is procedure-based. So are Ada[11], Communication Port[10], and the various languages based on monitors.* CSP[6], on the other hand, is message-based, as are PLITS[4] and Extended CLU[9].

3. Message Receipt

Many concurrent languages provide a receive operation that can be executed like any other instruction. Examples include Ada, Communication Port, CSP, PLITS, and Extended CLU. These languages demonstrate a fairly wide variety of syntax in their respective versions of receive, but they all have one thing in common: Message receipt is an *explicit* operation, performed by an already-active process.

By contrast, Distributed Processes and the various monitor languages have no receive operation at all. Rather, they provide a mechanism for handling incoming messages that looks very much like an ordinary sequential procedure. The arrival of a message triggers the execution of an appropriate body of code, much as an interrupt triggers its handler. With such a mechanism, message receipt is *implicit;* it is not performed by any active process.

It seems reasonable to describe explicit receipt as 'message-based' and implicit receipt as 'procedure-based.' With this classification system, both Communication Port and Ada are message-based. They were procedure-based in section 2.

4. Independence of Synchronization and Message Receipt

Some languages fall neatly into the procedure-based/message-based dichotomy; it makes no difference whether we base the classification on the synchronization mechanism or the form of message receipt. Other languages are not so easy to pin down. Both Ada and Communication Port use remote invocation send, but provide for the *explicit* receipt of messages. I know of no published language that uses only implicit message receipt with one of the simpler synchronization mechanisms. Cook's StarMod[3], however, provides *four different combinations:* both synchronization and remote invocation send paired with both explicit and implicit message receipt.

---

* Some monitor languages (e.g. Modula[12]) allow the unrestricted use of shared data. This can cause problems for implementations on multi-computers. However, if we impose the restriction (as in Concurent Pascal[2] and Mesa[7]) that *all* shared data be protected within monitors, then we can build fairly simple implementations in which monitors receive messages containing in parameters and send messages containing out parameters.

The existence of languages like Ada, StarMod, and Communication Port shows that synchronization and message receipt are independent issues. I suggest that the popular term **remote procedure call** should be used only for the combination of remote invocation send with implicit message receipt. "Message-based language" and "procedure-based language" are ambiguous and should be abandoned altogether.

5. The Need for Variety

Which is the 'best' combination of synchronization and message receipt? For synchronization, one is likely to prefer remote invocation send whenever a reply is expected. It has cleaner syntax than either other method, and it avoids the two implicit acknowledgments needed to achieve similar results with synchronization send. It may not be the best method, however, in situations that require no reply, since it reduces concurrency. Moreover, there are some problems that are nearly impossible to solve cleanly with remote invocation send[5]. These arise when one process requests a service from another (and eventually expects to receive a reply), but may need to provide information to a third process *before* the reply can be made. Clearly, no one synchronization mechanism will be best for all applications.

Now consider the syntax of message receipt. When a server handles requests from a community of clients, I find implicit receipt to be more 'elegant' than the explicit approach. There is something distasteful about a supposedly active process that does nothing but sit in an infinite loop waiting for something to do:

```
module server;
begin
   loop
      wait for a message;
      case request of
         A:
            ...
         B:
            ...
      end case;
   forever;
end server.
```

The implicit approach is more appealing:

```
module server;
entry A:
   ...
entry B:
   ...
end server.
```

For coroutines, on the other hand, implicit receipt is definitely *not* the method of choice. Consider the following:

```
module producer;
var
   mine : buffer;
   done : semaphore := 0;
   putit : semaphore := 1;

entry oldstuff (out his : buffer);  -- called by consumer
   P (done);
   his := mine;
   V (putit);
end oldstuff;

begin
   loop
      P (putit);
      produce;
      V (done);
   repeatedly;
end producer.

module consumer;
begin
   loop
      call producer.oldstuff(buffer);
      consume;
   repeatedly;
end consumer.
```

Equivalently, we may write

```
module consumer;
var
   mine : buffer;
   done : semaphore := 1;
   gotit : semaphore := 0;

entry newstuff (in hers : buffer);  -- called by producer
   P (done);
   mine := his;
   V (gotit);
end newstuff;

begin
   loop
      P (gotit);
      consume;
      V (done);
   for a long time;
end consumer.
```

```
module producer;
begin
   loop
      produce;
      call consumer.newstuff(buffer);
   equally long;
end producer.
```

Neither solution is as attractive as the explicit approach:

```
module producer;
begin
   loop
      produce;
      send buffer to consumer;
   end loop;
end producer.

module consumer;
begin
   loop
      receive buffer from producer;
      consume;
   end loop;
end consumer.
```

As with synchronization, no one form of message receipt will be best for all applications.

6. Conclusion

The discussions in sections 2, 3, and 4 have demonstrated that synchronization is an independent issue from the syntax of message receipt. Since both issues are intuitively appealing bases on which to categorize languages as 'message-based' or 'procedure-based,' this independence casts serious doubt on the validity of the message-based/procedure-based dichotomy. Moreover, the examples of section 5 demonstrate that there are situations in which any one of the possible combinations of synchronization and message receipt may be the method of choice. It seems reasonable to let the programmer decide which mechanism to use for each application. Just as a sequential language benefits from the presence of several similar loop constructs, so might a concurrent language benefit from the presence of several similar constructs for interprocess communication.

Acknowledgments

My thanks to the members of the Charlotte and Crystal research groups. Special thanks to my advisor, Raphael Finkel, and to his colleague, Marvin Solomon.

References

1. Brinch Hansen, P., "Distributed Processes: A Concurrent Programming Concept," *CACM* **21**(11) pp. 934-941 (November 1978).
2. Brinch Hansen, P., "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering* **SE-1**(2) pp. 199-207 (June 1975).
3. Cook, R. P., "*Mod--A Language for Distributed Programming," *IEEE Transactions on Software Engineering* **SE-6**(6) pp. 563-571 (November 1980).
4. Feldman, J. A., "High Level Programming for Distributed Computing," *CACM* **22**(6) pp. 353-368 (June 1979).
5. Finkel, R. A., "Tools for Parallel Programming," Appendix B of Second Report, Wisconsin Parallel Array Computer (WISPAC) Research Project, University of Wisconsin Electrical and Computer Engineering Report #80-27 (August 1980).
6. Hoare, C. A. R., "Communicating Sequential Processes," *CACM* **21**(8) pp. 666-677 (August 1978).
7. Lampson, B. W. and D. D. Redell, "Experience with Processes and Monitors in Mesa," *CACM* **23**(2) pp. 105-117 (February 1980).
8. Lauer, H. C. and R. M. Needham, "On the Duality of Operating System Structures," *ACM Operating Systems Review* **13**(2) pp. 3-19 (April 1979). Originally presented at the Second International Symposium on Operating Systems, October 1978.
9. Liskov, B., "Primitives for Distributed Computing," *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, pp. 33-42 (December 1979).
10. Mao, T. W. and R. T. Yeh, "Communication Port: A Language Concept for Concurrent Programming," *IEEE Trancactions on Software Engineering* **SE-6**(2) pp. 194-204 (March 1980).
11. United States Department of Defense, "Military Standard: Ada Programming Language," (MIL-STD-1815) (10 December 1980).
12. Wirth, N., "Modula: a Language for Modular Multiprogramming," *Software—Practice and Experience* **7** pp. 3-35 (1977).