A Fair Protocol for Non-deterministic Message Passing

Peter Grogono
Computer Science and Software Engineering
Concordia University
Montréal, Québec, Canada
grogono@cse.concordia.ca

Nima Jafroodi
Computer Science and Software Engineering
Concordia University
Montréal, Québec, Canada
n_jafr@cse.concordia.ca

ABSTRACT

Since Hoare introduced Communicating Sequential Processes as a model of distributed computation, there has been much discussion about efficient and flexible implementations. Previous research has led to communication protocols with restrictions: a process may only choose amongst receive operations; only a single pair of processes can be connected to a channel; or the protocol may be subject to deadlock or lack of fairness. We describe a fair protocol that allows arbitrary, non-deterministic communication amongst a set of processes connected by channels.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features.

General Terms

Design, Languages, Communication, Concurrency.

Keywords

Concurrency, message passing, deadlock, fairness.

1. INTRODUCTION

We are developing a language called Erasmus[1] in which the fundamental abstraction is a *process*. The code of an individual process is sequential, but processes execute concurrently and communicate by exchanging data. The rationale and structure of Erasmus have been presented in earlier papers [12, 13].

A *program* in Erasmus is a collection of processes that communicate by sending messages to one another. This style of programming was proposed early in the history of programming, with names such as *Cooperating Sequential Processes* [8] and *Communicating Sequential Processes* [15] but has not been widely used until recently. We refer to this style of programming as "message passing", abbreviated to MP. The MPI is one of the better-known examples of a message-passing infrastructure [14].

[1]The language is named after the Dutch humanist and scholar, Desiderius Erasmus (1466-1536).

C^3S^2E–*10* 2010 Montréal, Québec, Canada. Editor: B.C. Desai

Although MP has a number of advantages that we will discuss below, the reason that it has never become popular is probably because the advantages become significant only in multiprocessor settings. There are two main paradigms of multiprocessor programming: shared memory and MP. For a long time, shared memory systems attracted most of the attention of researchers because they seemed more efficient than MP systems. Recent developments in computer architecture, however, have changed this attitude. With processors that can execute hundreds of instructions per main memory cycle, leading to the need for multiple levels of caching, shared memory becomes less attractive and interest has moved towards MP.

In MP, communication may be either *synchronous* or *asynchronous*. When two processes communicate synchronously, one process waits until the other is ready, the message is transferred from the writer to the reader, and then both processes resume execution. When two processes communicate asynchronously, the writer places the message in a buffer owned by the reader and continues executing. Later on, the reader finds data in its buffer, usually referred to as a *mailbox*, and processes it. CSP [15], Joyce [5], and Ada [1] use synchronous communication; Erlang [2] uses asynchronous communication.

A synchronous system can easily simulate asynchronous communication: it is necessary only to attach to each reader a secondary process that act as its mailbox. The converse is not true: mailboxes or their equivalent are a basic component of asynchronous systems and cannot be ignored. Thus synchronous communication is more fundamental and, for this reason, Erasmus uses it.

2. BACKGROUND

Hoare's original proposal for communicating sequential processes [15] provoked a flurry of research. CSP introduced the concept of *selection*, which allows a process to choose amongst several other processes ready to communicate with it. Although CSP provided such a choice only for receivers, Hoare acknowledged that allowing choice for senders would lead to improved program structure for many applications. Most subsequent proposals for CSP extensions assume selection for senders.

Despite its simple formalism, CSP turned out to be hard to implement efficiently. Algorithms for synchronous communication with selection appeared quickly [3, 6, 11, 18], but suffered from problems such as deadlock and starvation. Bornat [4] published the first deadlock-free algorithm, but it supported only a single sender and receiver. Knabe [16] was the first to demonstrate an efficient, deadlock-free algorithm for selection on channels connected to an arbitrary number of processes.

Synchronous communication with selection was introduced to the functional programming community by Reppy in the form of an abstraction called *event* [17]. This approach has been refined and is currently used in Concurrent ML and Concurrent Haskell [7, 9, 10]. In this paper, we restrict our attention to imperative languages.

Buckley and Silberschatz [6] provide four criteria which can have a significant effect on the efficiency of the **select** statements:

1. the number of processes contributing to a single communication should be small;
2. processes shouldn't have too much information about the system and other processes they wish to communicate with;
3. the number of messages required to make a communication should be small;
4. If two processes in the system have matching send and receive commands, and they are not synchronized with any other processes, they should eventually synchronize.

The first two criteria ensure locality, the third ensures efficiency, and the last ensures progress.

3. SELECTION

A process, P_0 say, may communicate with several other processes, P_1, P_2, ... If P_0 decides to communicate with P_1, but P_2 is ready first, P_2 will have to wait. To avoid such waiting, MP languages provide ways of accepting messages in non-deterministic order. In Hoare's original formulation of CSP [15], the statement

$$[\ X?m \rightarrow S_1 \ \Box \ Y?n \rightarrow S_2\ ]$$

means "either read m from process X and perform sequence S_1 or read n from process Y and perform sequence S_2". Brinch Hansen refers to this kind of statement as a *polling statement*; Joyce [5] provides a statement with the same meaning but slightly different syntax. Ada provides a `select` statement with similar, though more complex, semantics. In Erasmus, port names rather than process names are used for communication, and ports have fields. Assuming that p and q are the appropriate ports, the corresponding statement would be written

$$\begin{array}{l} \textbf{select} \\ \quad m := p.x;\ \ S_1 \\ \quad n := q.y;\ \ S_2 \\ \textbf{end} \end{array}$$

For uniformity, we will refer to statements of this kind as *select* statements and the task they perform as *selection*.

All of the languages mentioned place various restrictions on selection. As mentioned, the original CSP allows selection only for receiving. This restriction prevents a sender and a receiver from polling the same channel simultaneously. However, it is an asymmetry and can lead to awkward code. Subsequent languages allow selection for both sending and receiving and use other means of preventing deadlock.

Another restriction of CSP and its successors is that, if one end of a channel is handled by selection, the other end of the channel must be an unconditional communication [5, 1]. This is a natural restriction because it is difficult to find an efficient implementation that allows selection at both ends of a channel. However, it is a serious restriction for large-scale programming because it prevents independent compilation: in order to compile a process, the compiler must inspect the code of other processes. Knabe's protocol [16] removes this restriction, allowing any number of selecting processes to be connected to a channel.

It is worth noting that "deadlock" has a special meaning in the context of this discussion. It is very easy to write a CSP program that deadlocks: for example, consider a program consisting of two processes, each of which tries to send a message to the other. We assume that programs are free from such trivial errors. For our purposes, an algorithm deadlocks if some subset of its processes ceases to make progress *even though there is a feasible communication between them.*

4. THE DISTRIBUTED PROTOCOL

The protocol described in this section is essentially Knabe's algorithm [16]. We describe it to provide background for our protocol, which is similar but has significant extensions.

Processes communicate with one another via *channels*. Each process is connected to any number of channels and each channel is connected to at least two processes. In order for processes P_1 and P_2 to communicate via channel C, there must be a *match*. A match occurs if either: P_1 is ready to send a message with tag T and P_2 is ready to receive a message with tag T, or *vice versa* (i.e., P_1 receives and P_2 sends). The *tag* of a message specifies its type and possibly other characteristics; the important point is that matched tags ensure meaningful communication. The protocol requires an ordering on processes; we will assume that each process has a unique identifier (UID) and that, if u_1 and u_2 are UIDs of distinct processes, then either $u_1 < u_2$ or $u_1 > u_2$.

A channel is an active process that executes as a single, non-terminating loop. The task of a channel is to find matches between pairs of processes. A process that is ready to communicate sends a *request* to one or more of its channels. The channels maintain two FIFO queues, one for send requests and the other for receive requests from processes. A channel remains idle as long as either queue is empty and becomes active when both of its queues are non-empty, at which point it enters a synchronization sequence with two phases: see Figure 1.

During the first phase, the channel, C say, chooses two complementary processes: that is, a process that wishes to send a message of type T and another processes that wishes to receive a message of type T. First, C picks the process $P_<$ with lower UID, regardless of whether it is the sender or the receiver. It sends the message L to $P_<$, requesting $P_<$ to temporarily commit to this communication and defer other signals it might receive. Then the channel waits for a reply.

If $P_<$ replies with `No`, C returns to its waiting state. If

```
begin
  proc Channel() ≡
    step ← Phase 1;

    Phase 1:
    (P<, P>) ← findMatch(Queue1, Queue2);
    query ← generateQuery(P<, P>);
    query.ReqType ← L;
    send(P<, query);
    reply ← receive();
    if reply = Yes
      then
          step ← Phase 2;
          comment:  A potential match.
       else
          Put P> back in the Queue;
          step ← Phase 1;
    fi

    Phase 2:
    query.ReqType ← H;
    send(P>, query);
    reply ← receive();
    if reply = Yes
      then
          comment: Match found.
          inform(P<, P>);
          step ← Phase 1;
       else
          comment: Match failed.
          query.ReqType = Abort;
          send(P<, query);
          Put P< back in the Queue;
          step ← Phase 1;
    fi
  .
end
```

Figure 1: Pseudocode for Channels

$P_<$ replies with `Yes`, C enters the second phase of the synchronization sequence. It sends the message H to the other process of the pair, $P_>$, requesting it to commit to this communication and reject all other signals. Again, C waits for a reply.

If $P_>$ replies `Yes`, C sends a `Ready` signal to $P_<$ and $P_>$, informing them that they can communicate, removes the corresponding entries from its queues, and returns to its waiting state. If $P_>$ replies `No`, C discards $P_>$'s request and sends a `Release` message to $P_<$, releasing it from its commitment. However, $P_<$'s request remains in C's channel.

A process P must also follow a procedure when it enters a selection. Its first step is to send a send or receive request to each of the channels involved in the selection; then it waits. A channel may reply with either H or L, depending on whether the process has the higher or lower UID of the proposed communication. If P receives H, it replies `Yes` to this channel, sends `No` to all of the other candidates for communication, and starts to transfer data.

If P receives L from C, it replies `Yes` and waits. Any signals that it receives from channels other than C are queued. Eventually, P will receive either a `Ready` message or a `Release` message. If the message is `Ready`, P sends `No` to all the losing channels and communicates. If the message is `Release`, P processes the first message on its queue, if there is one, otherwise waits for a message. The procedure ensures that exactly one communication occurs each time the selection is processed.

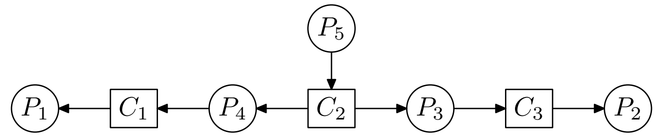

Figure 2: Knabe's algorithm allows P_5 to starve

4.1 Analysis

Knabe proves that the distributed protocol cannot deadlock [16]. The essence of the proof is that deadlock requires symmetry and that the ordered UIDs break the symmetry. However, the protocol does not ensure fairness. Although the whole system cannot become blocked, an individual process may wait indefinitely to communicate. This is called *starvation*.

Consider a group of connected processes Figure 2, in which the P_i are processes and the C_j are channels. A directed edge from a channel to a process denotes an outstanding receive request, and similarly a directed edge from a process to a channel denotes an outstanding send request. With the distributed protocol, it is possible for P_5 to be starved. All three channels will each start by attempting to acquire their low-numbered process followed by the high-numbered process to form matches. Without loss of generality consider a case in which C_1 and C_3 win to acquire processes P_1, P_4 and P_2, P_3 before C_2 can acquire P_3 or P_4. This leads to a situation in which C_1 and C_3 end up finding matches (P_1, P_4), and (P_2, P_3) respectively.

If this cycle happens repeatedly — that is P_1 and P_4 communicate frequently using C_1, and P_3 and P_2 communicate frequently using C_2 — then signals from C_2 will always be discarded, starving P_2.

5. THE FAIR DISTRIBUTED PROTOCOL

We describe an implementation of the **select** statement that provides nondeterministic choice, avoids deadlock, and treats all processes fairly. A **select** statement may have several branches that are a mixture of sends and receives. Channel behaviour is a bit different from the distributed protocol described in the previous. The only difference is that each channel piggybacks the attributes of the other process in the match along with the signal it is about to send to a process. These attributes are shown in Figure 4.

The additional feature of the protocol is a weight attached to each branch of a **select** statement. The weight may be an integer counter or a time-stamp; the important point is that it increases monotonically as the program runs. A counter is easier to implement but may overflow. A timestamp is preferable, but must be fine-grained because communications may occur very frequently.

The pseudocode for processes is given in Figure 3. Each process must follow a procedure when it enters a selection. Its first step is to send a send or receive request to each of

```
begin
  proc Process() ≡
    step ← step 1;

    STEP 1:
    for i := 1 to n do
        sendRequest(C_i, Req);
        comment:  Req ∈ { send, receive }
        step ← step2;
                          od
    STEP 2:
    query ← receive();
    if fair(query) = true
      then
            step ← query.ReqType;
            comment: ReqType ∈ { L, H }
      else
            send(query.QueryingChannel, NO);
            branch[query.branchNO] ← Aborted;
            step ← step 2;
    fi

    STEP L:
    send(query.QueryingChannel, Yes);
    reply ← receive();
    if reply = Ready
      comment: reply ∈ { Ready, Abort }
      then
            Actual Data Communication;
            ++Weight[reply.branchNo];
            abortOtherChannels();
            exit;
      else if reply = Abort
              then
                    branch[reply.branchNO] ← Aborted;
                    if branch[i:0 to n] = Aborted
                      then step ← step 1;
                       else step ← step2;
                    fi
            fi
    fi

    STEP H:
    send(query.channel, Yes);
    send(query.P_<, Ready);
    comment: P_< = process having lower UID
    Actual Data Communication;
    ++Weight[query.branchNO];
    abortOtherChannels();
    exit;
  .
where
proc fair(signal) ≡
  return signal.thisBranchWeight = signal.thisMinWeight
  and
  signal.otherBranchWeight = signal.otherMinWeight;
end
```

Figure 3: Pseudocode for Processes

the channels involved in the selection; then it waits. These requests carry all the attributes of the requesting process such as the weight of the branch, minimum weight of all the branches in the **select** statement, and etc. Unlike Knabe's algorithm in which all processes always respond `Yes` to the very first signal they receive, our implementation takes a different approach.

To clarify this, consider the case in which the process $P_<$ with lower UID has received its first signal from a channel. This signal indicates that if $P_<$ can commit itself temporarily to the signalling channel and delay others. Process $P_<$ examines the weights W of the branches that will be used to communicate; that is if $W_<$ is the lowest weight in the branches of $P_<$'s **select** statement, and the chosen branch of $P_>$'s **select** statement, say $W_>$, also has the lowest weight, then $P_<$ replies `Yes` and waits; otherwise it replies `No`.

If $P_<$ replies `No`, it has nothing further to do: the attempted match has failed, and $P_<$ continues to wait in its **select** statement. However, if $P_<$ replies `Yes`, it waits for another signal. If it receives `Ready`, it should have been sent by the higher UID process, so it sends `No` to any other waiting channels and proceeds with communication. It finally increases the weight of the branch of the **select** statement that was used in the communication.

But if it receives `Abort`, it should have been sent by the channel indicating that the match has failed; $P_<$ remains in its **select** statement, considering requests from other channels. The other case in which $P_>$ has received its first signal from a channel is almost same as above. In this case, process $P_>$ examines the weights W of the branches that will be used to communicate; that is if $W_>$ is the lowest weight in the branches of $P_>$'s **select** statement, and the chosen branch of $P_<$'s **select** statement also has the lowest weight, then $P_>$ replies `Yes` to the signaling channel, it then sends an abort signal to any other pending channels followed by a `Ready` signal to the lower UID process for the actual data communication, and finally it increases the weight of the branch of the **select** statement that was used in the communication. Otherwise it replies `No` to the signaling channel and proceeds with any other signals it receives.

The foregoing discussion has a few gaps in it that we will now fill.

First, each process makes send or receive requests only to channels of those branches having the lowest weights. Each process can have more than one branch having the lowest weight. By doing so each process avoid sending extra requests which are guaranteed to be responded with `No`. This results in receiving less signals which reduces the number of messages needed to find a match. Second, in the case where there is a sender or a receiver process with no **select** statement, the requesting process only sends a committed send or a committed receive request to the channel and waits for the actual data communication. These committed requests inform channels that the requesting process has already precommitted itself and that there is no need for the channel to ask for it.

Third, a process executing a **select** statement may receive `Abort` signals from all of its branches. If this happens, it simply starts everything all over again by sending new requests. Failing to do this could lead to deadlock.

Finally, it is clear that signals contain more information than just the type of the data. In fact, a signal is a fixed-size block of data containing the fields shown in Figure 4. Fields

Field name	Description
`messageId`	Unique identifier for this signal
`messageTag`	The tag/type associated with the message
`thisBranchNum` `otherbranchNum`	The numbers of the branches within the **select** statement
`thisBranchWeight` `otherBranchWeight`	The weights of the branches
`thisMinWeight` `otherMinWeight`	The minimum weights of the branches
`thisProcessId` `otherProcessId`	The UIDs of the processes

Figure 4: Data structure of a signal

after the first two come in pairs with a `this` field referring to the initiating process, and a `other` field referring to the responding process. Using the names in this table, each process needs to check the fairness condition by performing fair(*signal*) before replying to any queries. The body of this function is also given in Figure 3.

5.1 Deadlock

We show that deadlock cannot occur if there is a feasible match. Assume the contrary: there is a feasible match and that deadlock has occurred. This would imply that some processes have received a phase-one signal from one of their channels but have failed to find a match. However, this cannot happen because the process UIDs are ordered and only the low-numbered process of a pair receives a request during the first phase. Therefore, there must be at least one process, the one with the highest UID in the system, that has *not* received a phase-one signal and is available for matching.

5.2 Starvation

There are two situations in which starvation might occur. As an example of the first situation, consider a system in which a channel connects a single server P_0 to multiple clients P_1, $P_2, \ldots,$ P_n, as shown in Figure 5. Client requests are stored in the FIFO queue of the channel. There is a possibility that one or more of the clients might be starved. However, provided that the server continues executing, the protocol ensures that every request from a client will eventually be served. In this example, our protocol behaves in exactly the same way as Knabe's protocol.

Figure 6 shows the other situation. With the distributed protocol, P_2 may be starved. P_0 is the process with lowest UID and the protocol allows it to send `Yes` to all signals from C_1 but none from C_2, thereby starving P_2. With our protocol, the weights on the branches prevent starvation. After P_0 and P_1 have communicated once, $W(P_0, C_1) = 1$ and $W(P_0, C_2) = 0$. This ensures that the next time P_0 communicates, it will be with P_2. Neither P_1 nor P_2 can starve.

In the form described, the fair distributed protocol would have a serious problem: one slow process, treated fairly, could slow down the entire system. The **select** statement in Erasmus, however, provides for the declaration of a *policy*. Communication is implemented as above for **select** state-

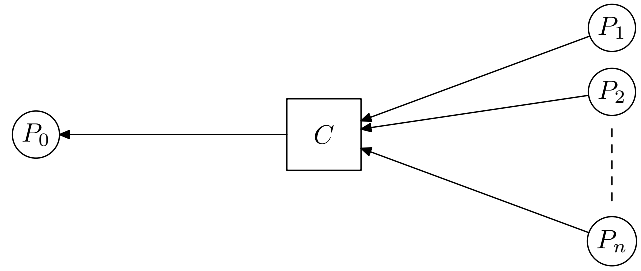

Figure 5: Starvation Avoided

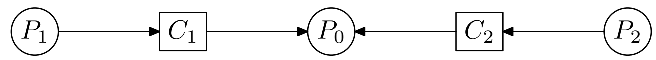

Figure 6: Starvation with the Distributed Protocol

ments that specify the policy **fair**. If **fair** is not specified, the process is not obliged to use branch weights in channel selection.

5.3 Cost

The cost of the algorithm is measured by the number of messages required to establish a communication. These messages include initial send or receive requests by processes, channels signals, responses to the signals, actual data communication, abort messages, and finally the wake-up signal from the sender to the receiver process.

To compute the cost of our algorithm several cases should be considered. The easiest and simplest case is where there are only two processes connecting through a channel, without any `select` statements. The total cost of the algorithm is 5 messages; two committed send and receive requests, a `Ready` message to the sender process, actual data transmission, and a `Ready` message to the receiver process.

The other case is where both processes execute their **select** statements. For this case we can define a lower and a higher bound for the cost. The lower bound is achieved when the process is the one with the highest UID in the system. This process responds `Yes` to the first signal it receives from one of its channels. Without loss of generality assume that this process, the one with the highest UID, has n branches and that the other communicating process has m branches; So, the lower bound on the cost is: $n+m$ requests, two signals, two `Yes` messages, a `Ready` message to the sender process, actual data transmission, and a `Ready` message to the receiver process, followed by $n+m-2$ `Abort` messages to the losing channels leading to the following formula for the lower bound, L:

$$L = 2(n+m) + 5.$$

Deriving an upper bound for the number of messages is difficult. As we have seen, it is possible for a process to continuously receives `Abort` or `No` messages from all of its branches for a while forcing the process to resend all of its requests again. The question is that for how many times would a process have to resend its requests? Equivalently, what is the maximum number of times that a process receives `Abort` messages from all of its branches?

To answer the above questions, consider a process P having the lowest UID in the system. Without loss of generality, assume that P has n branches and $P_1, , P_n$ are n distinct processes which are connected to these branches. If b_i is the number of branches of P_i, and if all processes are executing their **select** statements for the first time then after exactly $B - 1$ times failing where $B = \min\{b_1, \ldots, b_n\}$ the process P eventually communicates with a process.

Therefore, in each round process P makes n requests, followed by n signals, n `Yes` messages, followed by n `Abort` messages. Eventually, in round B, there are n requests, n signals, two `Yes` signals, a `Ready` message to the sender process, actual data transmission, and a `Ready` message to the receiver process. This leads us to the following formula for the upper bound, U:

$$U = 4n(B - 1) + 2n + 5.$$

Finding an upper bound in the case where channels supports many to many communications is similar to the above. The upper bound of messages is calculated separately for each process in the system depending on the number of channels it has, the number of processes sharing channels, and the number of branches its communicating processes have.

6. CONCLUSION

We have described a fair, distributed protocol that allows an arbitrary network of processes linked by channels to communicate fairly and without deadlock. Processes may perform selection on both sends and receives and may be connected to an arbitrary number of channels. Conversely, a channel may be connected to an arbitrary number of processes. The general case requires extensive handshaking, but run-time analysis might allow more specialized and efficient techniques to be used in particular cases. For example, a JIT compiler could determine the number of processes connected to a channel and generate simplified code for the probably common case in which only two processes are involved. Since global static analysis is not required, processes may be compiled independently and linked dynamically. This last feature is essential for the construction of large-scale, distributed systems.

Acknowledgments

Funding for this research was provided by the Natural Sciences and Engineering Research Council of Canada and the Faculty of Engineering and Computer Science at Concordia University.

7. REFERENCES

[1] Ada. Ada 95 Reference Manual. Revised International Standard ISO/IEC 8652:1995, 1995. www.adahome.com/rm95. Accessed 2008/03/12.

[2] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. The Pragmatic Bookshelf, 2007.

[3] Rajive Bagrodia. Synchronization of asynchronous processes in CSP. *ACM Trans. Program. Lang. Syst.*, 11(4):585–597, 1989.

[4] R Bornat. A protocol for generalized occam. *Software—Practice and Experience*, 16(9):783–799, 1986.

[5] Per Brinch Hansen. Joyce—a programming language for distributed systems. *Software—Practice and Experience*, 17(1):29–50, January 1987.

[6] G. N. Buckley and Abraham Silberschatz. An effective implementation for the generalized input-output construct of csp. *ACM Trans. Program. Lang. Syst.*, 5(2):223–235, 1983.

[7] Avik Chaudhuri. A concurrent ML library in concurrent Haskell. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional programming*, pages 269–280, New York, NY, USA, 2009. ACM.

[8] Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[9] Kevin Donnelly and Matthew Fluet. Transactional events. *SIGPLAN Not.*, 41(9):124–135, 2006.

[10] Matthew Flatt and Robert Bruce Findler. Kill-safe synchronization abstractions. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 47–58, New York, NY, USA, 2004. ACM.

[11] Peter Grogono. Aspects of programming language design. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, October 1979.

[12] Peter Grogono and Brian Shearing. Concurrent software engineering: Preparing for paradigm shift. In Bipin C. Desai, editor, *Proceedings of the Canadian Conference on Computer Science & Software Engineering (C^3S^2E'08)*, pages 99–108. ACM Press, May 2008.

[13] Peter Grogono and Brian Shearing. Modular concurrency: a new approach to manageable software. In *3rd International Conference on Software and Data Technologies (ICSOFT 2008)*, pages 47–54, July 2008.

[14] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, second edition, 1999.

[15] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[16] Frederick Knabe. A distributed protocol for channel-based communication with choice. In *PARLE '92: Proceedings of the 4th International Conference on Parallel Architectures and Languages Europe*, pages 947–948, London, UK, 1992. Springer-Verlag.

[17] J. H. Reppy. Synchronous operations as first-class values. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 250–259, New York, NY, USA, 1988. ACM.

[18] A. Silberschatz. Communication and synchronization in distributed systems. *IEEE Transactions on Software Engineering*, SE-5(6):542–546, November 1979.