A Distributed Protocol for Channel-Based Communication with Choice

Frederick Knabe
European Computer-Industry Research Centre GmbH
Munich, Germany
knabe@ecrc.de

Abstract

Recent attempts at incorporating concurrency into functional languages have identified synchronous communication via shared channels as a promising primitive. An additional useful feature found in many proposals is a nondeterministic choice operator. Similar in nature to the CSP alternative command, this operator allows different possible actions to be guarded by sends or receives. Choice is difficult to implement in a distributed environment because it requires offering many potential communications but closing only one. In this paper we present the first distributed, deadlock-free algorithm for choice.

Keywords: Distributed protocols, channels, synchronous communication, choice operator, CSP alternative command.

1. Introduction

In 1978, C.A.R. Hoare introduced a now-classic paradigm for parallel programming called communicating sequential processes, or CSP [12]. CSP presented a new approach to concurrent programming, and its legacy has continued to influence well-founded proposals for concurrent languages, particularly those relying on synchronous message-passing.

In CSP, processes are allowed to communicate with one another via synchronous send and receive operations. When one process attempts to send data to some other process, it blocks until there is a simultaneous matching receive by that process (and vice versa). In each case, the send and receive operations explicitly name their destination or source process.

Hoare also provided CSP with an "alternative" construct, a generalization of the familiar **if-then-else**. At an alternative a process nondeterministically chooses between several different actions. However, an action can only be chosen if its associated guard, a boolean expression, is true.

A key feature of CSP is that the guards for the different branches of an alternative can also be receive operations. A receive operation is "true" if another process is attempting a matching send. This feature allows a process to wait for input from several other processes simultaneously. The process can take different actions depending on which receive operation is chosen, and perhaps later loop back into the same alternative construct.

Hoare noted the desirability of also allowing send commands as guards. As others also examined this expanded construct, its effective implementation began to draw interest. Buckley and Silberschatz [7] presented an overview of several early attempts, as well as detailing their own algorithm. They also made a notable contribution in outlining four criteria for evaluating different implementation approaches:

1. The number of processes involved in a single synchronization event (between matching send and receive commands) should be small.

2. In order to synchronize, a process should not be required to have too much information about the system and environment.

3. If two processes in the system have matching input and output commands, and they do not synchronize with any other processes, then they should eventually synchronize with one another.

4. The number of messages required to establish communication should be small.

The first two criteria stress locality, while the third ensures progress and the last concerns efficiency.

A natural extension of this generalized alternative construct is to allow a synchronization to involve multiple processes, not just two. Chandy and Misra characterized this as the committee coordination problem [9]: Professors in a university are assigned to various committees. Occasionally a professor will decide to attend a meeting of any of her committees, and will wait until that is possible. Meetings may begin only if there is full attendance. The task is to ensure that if all the members of a committee are waiting, then at least one of them will attend some meeting.

The crux of this problem is that two or more committees might share a professor. When that professor becomes available, she can only choose one of the meetings, while the others continue to wait. Coordinating this is non-trivial, and various solutions have appeared [9, 17, 22, 23].

Several proposals for concurrency primitives [5, 11, 19, 20, 24] further extend the CSP model in a direction briefly mentioned in Hoare's original paper. These introduce communication entities known as channels. A process no longer explicitly names other processes in its send and receive operations but instead names a channel. Synchronous transfer of data from one process to another occurs when the two attempt a send and a receive at the same channel.

Multiple processes may make simultaneous sends and receives to one channel. This in turn implies that several different matchings of processes with one another may be possible, and these are chosen nondeterministically. When we include the level of indirection introduced by channels, implementing the alternative construct (which we call choice) becomes even more complex.

This communication model has drawn strong interest because it is both semantically clean and a natural paradigm for functional programming. Early shared memory implementations such as PFL [13] and Amber [8] led to Reppy's Concurrent ML [25], which is now being applied to systems programming (e.g. eXene [26], a multi-threaded X window system toolkit). Reppy's primitives have also been implemented by Morrisett [21].

The next significant step is to extend the application domain of channels with choice to distributed systems. These platforms should prove a strong test of this model's utility. However, a distributed implementation of these communication primitives requires a protocol to manage the complex interactions between channels and processes. To our knowledge, such a protocol has not appeared in the literature. Joung and Smolka's work [14] is somewhat related, but their algorithm concerns a fixed pool of processes engaging in statically defined "interactions" (which are roughly analogous to rendezvous at channels). Poly/ML has a distributed extension [18] which provides a form of the choice operator, though it is a much simpler construct than the one we have described. Bornat's protocol for a generalized choice operator in occam [6] has the closest similarity to our own work. Processes communicate via channels, but these channels are restricted to one-to-one process connections rather than the any-to-any which we are considering. The underlying communication model is assumed to be synchronous, a restriction we waive (see the next section). Nevertheless, Bornat's protocol shares many similarities in approach to the one we developed independently.

In this paper we present a distributed protocol for choice which satisfies Buckley and Silberschatz's criteria. After describing the algorithm, we discuss the issues of deadlock, fairness, fault tolerance, and cost.

A note on terminology: The term "process" is often used to refer to a heavyweight, operating system process. The processes we refer to are generally assumed to be much more lightweight. In practice, a single heavyweight process can usually support many of these lightweight processes using separate threads of control.

2. The distributed protocol

Let us begin by clarifying our assumptions. In most of the communication models cited previously, channels are themselves values which may be created and transmitted from one process to another. Also, new processes may be forked off at runtime, implying that an implementation cannot rely on some fixed and known process set. Since a process does not leave a choice until at least one communication is possible, it is possible for processes to deadlock or starve depending on how the programmer has structured use of choice. A fair scheduler might be able to prevent some cases of starvation, but this is not a requirement.

Since we are interested in the interprocess communication details arising from choice, we will assume that the branches are always guarded by sends or receives. Furthermore, the underlying network must permit processes to communicate via point-to-point asynchronous messages. The messages are delivered reliably, but their relative ordering is not necessarily preserved.

Channels are usually viewed as mere data objects which serve as rendezvous points, but this can be overly restrictive. We have found it useful in designing our protocol to consider channels as threads of control which can mediate between multiple IO requests. This allows both channels and processes to take an active role in evaluating a choice and simplifies the coordination problem.

Elevating the notion of channels in this way imposes no serious implementation difficulties. A channel's thread of control may be forked off when it is created, and

the data value that the programmer manipulates can be just the internal name of this thread. (In an actual implementation it may be useful to have just one thread of control manage multiple channels.)

Our algorithm also relies on the existence of a global ordering for processes, an idea which goes back to Bernstein [4]. Each process has a unique id, and any two ids may be compared against one another. This is not an unreasonable assumption, for ids are necessary to route messages unambiguously.

We now present the structure of the protocol. We show first the actions associated with the channels, and then turn to the role of the processes.

2.1. The channel side

We implement a channel as an infinite service loop which proceeds through various states depending on the messages it receives. Processes may make either send or receive requests to a channel, depending on which action they wish to accomplish. The purpose of the channel is to recognize when it has received two complementary requests and to attempt to match them with one another. This is not guaranteed to succeed, since a process may send requests to several channels at once and only waits until at least one request leads to a match.

Because any number of requests can flood into a channel, each channel maintains two FIFO queues: one for send and one for receive requests. While one of the queues is empty a channel can do nothing, and it remains idle. Upon detecting a match, however, the channel enters a two-phase procedure.

For any two complementary requests from different processes, there is one process whose id is smaller or lower than the other. Determining which is smaller can be easily imagined: first the names of the processes' hosts are compared, then their process ids, and then perhaps some finer identification. The channel's first step is to pick the request associated with the lower process; it doesn't matter whether that request is a send or a receive.

The channel now sends a query to this process, asking if the process will temporarily commit itself to this channel's match and defer other queries it might be receiving. Once this query message has been sent, the channel waits for a reply.

The protocol allows only two responses: "yes" or "no." If the channel receives a "no," the queried process has already been successfully matched by some other channel. In this case, the channel discards that process's request and once again becomes idle, checking its queues to see if it can start the matching procedure anew.

If instead the reply is a "yes," the channel has captured one of the processes necessary to make a successful match. It can now enter the second phase. The channel queries the other, higher process. This query requests that the second process commit itself to this match and reject all other queries. Once again the channel waits for a reply.

Upon receiving a "yes," the channel's match has been successful, and it removes both processes' requests and returns to its idle state. But if it receives a "no," the channel must abandon this match, since the higher process has chosen some other match. The channel discards the higher process's request, and must now free the lower process from its temporary commitment. This will allow the lower process to consider queries from other channels. To this end, the channels sends a release message to

the lower process (but retains its request), and goes back to its idle state to start from scratch.

2.2. The process side

The process portion of the protocol is somewhat different in structure from the channel side. Whereas a channel always remains in a message server loop, a process will normally be executing user code. When a choice is encountered, special protocol code must be entered.

The choice construct consists of several branches, each guarded by a send or receive command to some channel. The process begins by sending send or receive requests to each of the channels. Then it enters a loop to handle the queries which come back.

Perhaps the simplest case is when the process is the higher-numbered of the two in a match, in which case it receives the second query from a channel. (Processes can distinguish between the two types of queries.) Upon receipt of this message, the process has now been synchronized through the channel with the lower process. It sends a "yes" back to the channel, and then sends a "no" to all the other channels to which it sent requests.

All that remains now is the actual data communication. If the chosen request is a send, the process transmits the data directly to the lower process. Otherwise, the request was a receive, and the process sends a "ready" message to the other process. When the lower process gets this "ready," it sends the data. After the data transmission (either send or receive), the process returns to user code.

Matters are somewhat different if the process is the lower partner in the match. In this case the process simply sends back a "yes" to the querying channel and waits. While it is waiting, the process might receive queries from other channels it sent requests to. These it saves on a queue.

Eventually the process receives one of three messages. If either a data or a "ready" message arrives, it must have been sent by the higher partner, and thus the process has been successfully matched. It sends out a "no" to all the losing channels, and then either processes the data or sends its own in response to the "ready." The process can then exit the protocol. The other possibility is a release message. In this case the process stops waiting and processes the next query on its queue as if it were the first. If the queue is empty, the process must wait for a query to arrive.

2.3. Additional details

In the preceding description of the protocol we have glossed over several important details in the interest of clarity. We mention some of these now. (The complete algorithm may be found in [16].)

After a process sends out several requests, it must be able to distinguish which incoming query matches which request. To this end the process assigns each request a local id that is sent along to the channel. The channels send back these ids in their queries.

The ids are also necessary to resolve the following scenario. A process could send out requests, successfully match, send out "no" messages, and leave the protocol before some of its request messages even reach their destination. The channels receiving these

requests might send queries in response before the "no" messages arrive. Now, what happens? When the process next enters the protocol, it will receive these out-of-date queries. Since the channels involved have already been informed that this process had a successful match, the process doesn't need to do anything with these queries and can just discard them. In order to do this (and also not accidentally throw away any new, valid queries pertaining to its current requests), it relies on the request ids.

It is possible for request messages to arrive at a channel at any time. If the channel is not idle, it just queues these requests appropriately and remains in the same state. A more intriguing circumstance occurs when a "no" message from a process overtakes the original request from that process. (Recall that we allow the network to alter the message ordering.) To handle this properly, the receiving channel stores such "no" messages in a table. Later, as the channel considers each request in its idle state, it first checks to see whether it should cancel the request against an already-received "no."

The notion of a match between two requests is actually narrower than just whether one is a send and the other is a receive. An important feature of the choice construct is that a process can attempt both a send and a receive to the same channel in different arms of a choice without matching itself. When matching two requests at a channel, the algorithm ensures that they are not both from the same process.

Finally, when a channel has received a negative response to its second query, we said it sends a release to its already-acquired low process. However, it might be the case that the channel has another complementary request from a higher-numbered process already in its request queue. If it does, instead of performing the release the channel sends a second-phase query to this process and remains in the same state. Although this is not necessary for the protocol, it can help reduce the message overhead by taking advantage of some work already done.

As suggested in the above, protocol messages do carry some data. Typically this consists of process or channel addresses, request ids, and the like. The amount of information transmitted is fixed and small.

3. Discussion

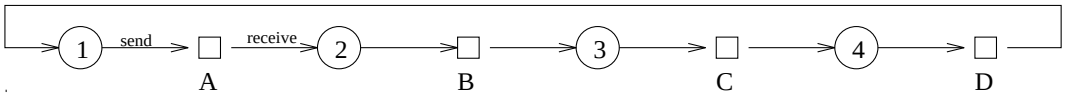
3.1. Deadlock

We consider it essential that our protocol be free of deadlock and livelock. This may seem at first impossible, since no matter what the nature of a protocol, the user can create deadlock using the choice construct. User-created deadlock, however, is distinguished by the fact that no matches exist at any channels. We can do nothing about this. But if matches do exist, then the protocol should never deadlock in the process of establishing communication between the matched processes.

In order to discuss deadlock and related issues, it is useful to look at various "connected groups." A connected group is a set of processes and channels which could all potentially influence one another. More precisely:

DEFINITION: Let processes and channels be vertices of a directed graph where an edge from a process to a channel denotes an outstanding send request and an edge from a channel to a process denotes an outstanding receive request. Let a root node be one which has no ancestors which are not also descendants of it. Then all the descendants of any root node (including the root) form a *connected group*.

As an example, consider the following connected group. Circles represent processes, while squares are channels. The directed edges have the meaning described above (two are labeled for clarity). Note that any of the nodes can serve as a root for the connected group.



We can easily imagine a protocol which could lead to deadlock in this group. Suppose each channel acquires exclusive access to the process which has a send request to it. Now each channel tries to acquire exclusive access to the process which has the receive request. Deadlock results. (This example should be recognizable as the Dining Philosophers problem.) Can our protocol handle this situation?

Let us define another concept. In any connected group there is a set of processes (the "active set") which have requests currently being matched into pairs by various channels in the group. More formally:

DEFINITION: The *active set* of a connected group consists of all processes with edges to channels which have at least one incoming edge and one outgoing edge (that is, channels which have outstanding complementary requests).

In the above example, the active set happens to include all the processes.

CLAIM: In any connected group with a non-empty active set, our protocol guarantees at least one match must succeed.

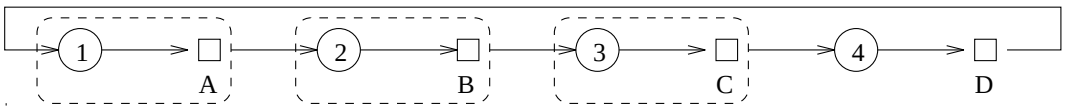
The proof of this hinges on the fact that every process has a unique global identifier. In any non-empty active set, there is one process with the highest identifier in the set. As a result, this process will never be sent a first-phase query by any of the channels matching with it. The process can therefore respond to the first second-phase query it receives, and this will be a match.

COROLLARY: Matching continues in a connected group until no matches remain.

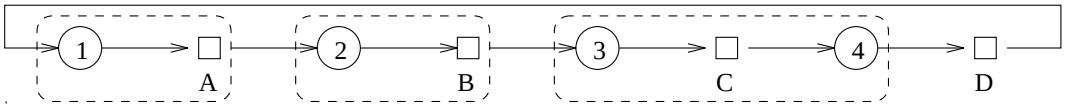
Once a pair of processes have been matched, they drop out of the connected group along with their requests. The preceding claim now applies to the remaining connected groups.

To be complete, we should also point out that matches can only occur within connected groups. This follows from the definition. Thus, we have shown that deadlock cannot occur in our protocol.

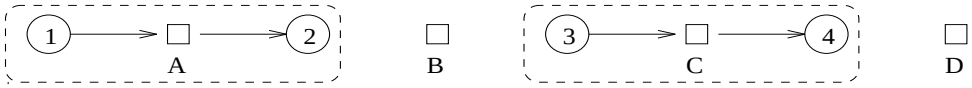
As an illustration, consider one possible course of events with our protocol and the preceding four-processes example. The four channels will each attempt to acquire their low-numbered process. Notice that A and D will contend for the same process; assume A wins (in which case D's query will remain queued at process 1). We thus have this situation:



The dashed boxes indicate which processes have been acquired by which channels. Now channels A, B, and C will attempt to acquire their high-numbered processes. Since processes 2 and 3 have already been acquired, the queries from A and B will be queued, and only C will succeed.



At this point B will receive a “no” from process 3, and it will in turn release process 2. This allows A’s queued query to be accepted, and another match is formed. B also gets a “no” from thread 2 cancelling 2’s request. D is sent “no” messages by both processes 1 and 4, indicating that those processes have been matched.

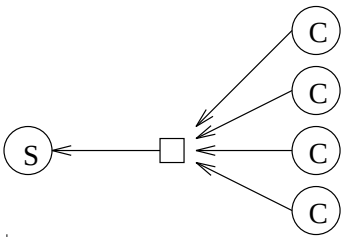


At this point there are no more processes to be matched, and the connected group dissolves.

3.2. Starvation

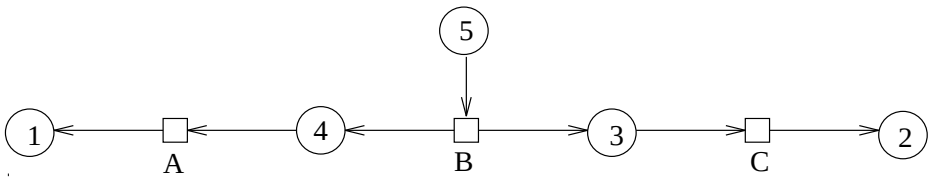
Starvation is a much more difficult problem than deadlock. To prevent a process from being starved appears to require a central fair scheduler or much greater communication between channels and processes (more than we consider reasonable). In fact, since problems may be constructed with unbounded nondeterminism (Hoare [12] gives an example in the CSP domain), no implementation can be guaranteed to be fair.

Our protocol makes a modest effort at preventing starvation, which is enough to support the most common programming paradigms. Consider, for example, a server process which continuously receives from a particular channel. Meanwhile, there are many client processes which make send requests to the channel. We thus have the following connected group:



Our protocol guarantees that none of the client processes will starve. The reason is simple enough. Each client’s send request is queued at the channel in the order it is received. As long as the server keeps receiving, each client will eventually be served.

We cannot guarantee against starvation in the following example, however:



Assume that all the processes are in loops so that this connected group is repeatedly established. Each time, channels A and C could form pairs before channel B acquires access to processes 2 or 3, and as a result process 5 can starve. Our protocol only prevents starvation with requests at a single channel (where we have enough centralized control to do fair scheduling), but not in this multi-channel case.

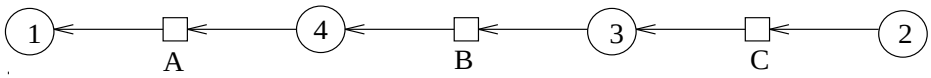
When using choice, the programmer must recognize situations such as the above example. We can hope that in an actual implementation the randomness of the system would allow channel B to gain access to one of the crucial processes before the other channels. Consider, though, that channel A and processes 1 and 2 could be local to one heavyweight process, and likewise for channel C and processes 3 and 4. If channel B is located at a remote machine, network delays may always cause its requests to be too late.

3.3. Modifications to the protocol

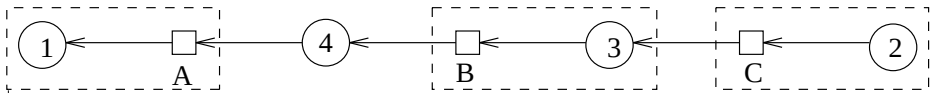
The observant reader may have noticed that there are points in this protocol where both channels and processes can block waiting for other matches to succeed or fail. Although one can design scenarios where a great deal of blocking takes place, in normal practice these cases should be rare. It would still be desirable to avoid them altogether. But as we shall see, most of the "obvious" fixes have unpleasant consequences.

Whenever a channel sends a query to a process, it risks being placed on that process's query queue and being delayed. The channel is unable to proceed until the process either responds positively to its query or sends back a "no" (indicating it successfully matched through some other query). If the channel's query is from the second phase, blocking the channel also implies that its previously acquired low-numbered process is also blocked. As this double blocking seems bad, we might try to avoid it by modifying the protocol. We can require the channel to give up, release its low-numbered process, and start over after being sent some sort of "busy" message from the queried process. This could allow the low-numbered process to participate in other potential matches.

At first this seems like it might improve the throughput of the system. In fact, however, it can lead to a much more insidious form of starvation than that already noted. Consider the following example:



Let us assume that all the processes are looping through their choice constructs, so that this connected group will be repeatedly established. Initially, the channels will acquire their low-numbered processes, leading to the following:



Suppose that A succeeds in obtaining process 4. Meanwhile, B is also attempting to obtain process 4, and C is attempting to obtain process 3. Both B and C will fail, so they will release their low-numbered processes.

B and C will now attempt to form matches again. But let us assume that processes 1 and 4 have now finished their communication, looped back into their choice constructs, and the connected group is once again as in the first diagram. Now the same course of events can be repeated.

The key problem is that one pair of processes can prevent matches elsewhere in the system from succeeding, since the modified protocol allows those other matches to be abandoned halfway (and all progress is lost). Moreover, the example above can be made arbitrarily large. We conclude that this modification is dangerous.

Perhaps a better method would be to increase the likelihood that a channel will have a successful query. Instead of sending just one query to a low process, a channel could simultaneously send queries to all requesting processes (save for the highest-numbered). As soon as one positive response is received, releases could be sent to cancel all the other queries.

Unfortunately, this probably increases the message traffic without improving the overall throughput of the system. Processes will now be more likely to block needlessly, since they will respond to queries which are then cancelled. When blocked they are unable to respond to queries from other channels. Although measurements are needed for a certain determination, the overall effect is likely to be negative.

In another attempt at increasing throughput, we could modify the protocol so that processes favor second-phase queries over first-phase on their query queues. The reasoning is that responding positively to the former always guarantees an immediate match, whereas responding to a first-phase query may entail a wait.

This approach suffers from favoring lower-numbered processes over higher-numbered processes in matches. Consider a process which sends requests to several channels and then receives queries from these channels. With this modification, the process will always select a second-phase query, which in turn must always come from a channel connected to a process with a lower id. In particular, consider the highest-numbered process in the system. No process will match with it, if there is a choice. Aside from this, the modification adds nothing if query queues are generally very short (perhaps only one member), and this has been the case in our experiments.

Finally, there is one optimization we can make which is simple and useful. A fair amount of communication by processes is via simple sends or receives which are not part of choice constructs (or, equivalently, via choice constructs which consist of just one request). When a process makes an isolated request, it can immediately commit, since it can't do anything else until the request is matched. A channel receiving such a request can treat it as an already-acquired process and simply send a second-phase query to its match (there is no fear of deadlock here regardless of which process has the higher number). If the match is also an isolated request, the channel simply informs the matching process of its mate. This can greatly decrease the number of messages necessary to establish a match. (The complete algorithm in [16] includes this modification.)

3.4. Cost

Establishing a lower bound on the number of messages required to establish communication with our protocol is straightforward. The simplest case occurs with two processes and one channel. With the optimization mentioned above, only three messages are required: two requests and the match message from the channel.

Of course, it is somewhat more interesting to consider the case when both processes are in choice constructs and the full protocol must be followed. Between the two processes and the channel, either six or seven messages are required: two requests, two queries, two "yes" messages, and a possible "ready" if the high-numbered process is the receiving partner. This does not account for all messages sent; each process will also send out a request and a "no" for the other branches of its choice.

Establishing an upper bound is much trickier. As we have seen, it is possible for a process to starve for an arbitrarily long period. Should this process suddenly establish communication, how should we measure the number of messages involved? The channel it is connected to might have sent many queries and been denied each time. If instead we examine a given connected group, we can compute the worst case number of messages before no matches remain. We have done this, but the results are not particularly illuminating. The best we can say is that in any connected group with a non-empty active set, it is certain that one channel and two processes will have established communication after sending six or seven messages between one another.

3.5. Fault tolerance

The ability to build fault tolerance into a distributed protocol such as ours is in some measure dependent on the facilities supplied by the transport system. Modern operating systems such as Mach [1] generally supply sufficient information about unreachable message recipients to enable recovery actions, so we will assume this is a given.

It is relatively straightforward how to proceed in our protocol if a message can no longer be sent. For example, suppose a process has received a first-phase query and wishes to reply with a "yes," but the channel is now unreachable. The process could place the query at the end of its query queue and instead take the next waiting query. Or, it could give up on that query altogether and simply delete it. The choice of actions depends on the particular semantics being supported.

Perhaps the most difficult case occurs when a channel is in the process of acquiring its high-numbered process and it suddenly loses its low-numbered process. In such a case both the channel and its high-numbered process must independently take corrective action. But again, we see no real difficulty in implementing this. The characteristics of our protocol which lend themselves to adding fault tolerant features are the small number of participants involved in establishing any particular match and the clearly delineated state transitions.

An additional issue which must be considered in an implementation is how to handle the death of a heavyweight process which is also the location of one or more channels. This location (along with the state of the channels) must be migrated to another process in some manner which allows processes throughout the system to find it. Again, one must rely on operating system services to accomplish the migration.

Before fault tolerance can be implemented, though, the semantics of a fault must

be incorporated into the already existing semantics for channels with choice. This is still an area of research.

3.6. Implementation

We have successfully implemented our protocol in Standard ML of New Jersey [2] using Cooper and Morrisett's ML+threads [10] and Knabe's Mach IPC package for ML [15]. We relied on the Facile definition of choice [11], and our (abbreviated) user signature looks like this:

```
signature FACILE =
  sig
    type channel
    type stop

    datatype request =
        SEND of channel * data * (unit -> unit)
      | RECV of channel * (data -> unit)

    val channel   : unit -> channel
    val spawn     : (unit -> stop) -> unit
    val terminate : unit -> stop
    val parallel  : (unit -> unit) list -> unit
    val select    : request list -> stop        (* Choice *)
    val send      : channel * data -> unit
    val receive   : channel -> data
  end
```

For our initial implementation, `data` is restricted to strings, integers, and channels.

Each heavyweight process in our system has associated with it a channel manager, a listener, and multiple lightweight processes. The channel manager handles all messages directed to channels created at the process. This central management allows channels to be garbage collected away when not in use, and their representation in the rest of the system is simply a location and an identifier. The listener monitors the process's operating system ports and routes remote messages to their destinations. Communication between lightweight processes and channels on the same heavyweight process is significantly faster than between two heavyweight processes, since only memory manipulations (as opposed to actual IPC) are involved.

Ideally, we would like to structure an implementation so that the internal communications could bypass the protocol altogether, while still relying on it for the remote case. When programs begin to use thousands of processes and channels, the overhead for communication can easily dominate other costs. We plan to incorporate this into a future version.

4. Conclusion

It is time to return to Buckley and Silberschatz's four criteria. Since our protocol only requires three threads of control to be involved in a synchronization (the two lightweight processes and the thread managing the channel), it satisfies the first criterion. Each process has no information about the rest of the system other than the channels with

which it communicates directly, thus meeting the second locality requirement. We have shown that the protocol does not deadlock, thus fulfilling the third. Finally, only a small number of messages are required by the protocol in order to establish communication (although due to the semantics of choice, the caveat remains that a process can starve no matter how many times its channels attempt to match it).

Even though we have met the criteria, there is still room for improvement. In particular, it may be desirable to relax the restriction on deadlock. By using a probabilistic algorithm which only makes guarantees about the likelihood of deadlock, it might be possible to reduce the average number of messages necessary for a synchronization. Such an approach might also eliminate the blocking by processes and channels which is currently part of our protocol.

References

[1] M. J. Accetta, R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. T. Jr., and M. W. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–113, June 1986.

[2] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In J. Maluszyński and M. Wirsing, editors, *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming (PLILP)*, number 528 in Lecture Notes in Computer Science, pages 1–13, Passau, Germany, Aug. 1991. Springer-Verlag.

[3] R. Bagrodia. A distributed algorithm to implement the generalized alternative command of CSP. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 422–427, Cambridge, MA, May 1986.

[4] A. Bernstein. Output guards and nondeterminism in "communicating sequential processes". *ACM Transactions on Programming Languages and Systems*, 2(2):234–238, Apr. 1980.

[5] D. Berry, R. Milner, and D. N. Turner. A semantics for ML concurrency primitives. In *ACM Symposium on Principles of Programming Languages*, 1992.

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

[7] G. N. Buckley and A. Silberschatz. An effective implementation for the generalized input-output construct of CSP. *ACM Transactions on Programming Languages and Systems*, 5(2):223–235, Apr. 1983.

[8] L. Cardelli. Amber. In *Combinators and Functional Programming Languages*, number 242 in Lecture Notes in Computer Science, pages 21–47. Springer-Verlag, July 1986.

[9] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, 1988.

[10] E. C. Cooper. A thread interface for Standard ML. School of Computer Science, Carnegie Mellon University, Apr. 1990.

[11] A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, Apr. 1989.

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

[13] S. Holström. PFL: A functional language for parallel programming and its implementation. Technical Report 83.03 R, Department of Computer Science, Chalmers University of Technology, 1983.

[14] Y.-J. Joung and S. A. Smolka. Coordinating first-order multiparty interactions. In *18th ACM Symposium on Principles of Programming Languages*, pages 209–220, Orlando, FL, Jan. 1990.

[15] F. Knabe. A Mach IPC facility for SML. Fox note 5, School of Computer Science, Carnegie Mellon University, 1991. Internal working paper.

[16] F. Knabe. A distributed protocol for channel-based communication with choice. Technical Report ECRC-92-16, European Computer-Industry Research Centre, Arabellastr. 17, W-8000 Munich 81, Germany, June 1992.

[17] D. Kumar. An implementation of N-party synchronization using tokens. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 320–327, Paris, France, May 1990.

[18] D. C. J. Matthews. A distributed concurrent implementation of Standard ML. LFCS Report Series ECS-LFCS-91-174, LFCS, University of Edinburgh, Aug. 1991.

[19] R. Milner. *A Calculus of Communicating Systems*. Number 92 in Lecture Notes in Computer Science. Springer-Verlag, 1980.

[20] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1989.

[21] J. G. Morrisett. Implementing events in ML+threads. Venari Note 5, School of Computer Science, Carnegie Mellon University, 1990. Internal working paper.

[22] M. H. Park and M. Kim. A distributed synchronization scheme for fair multiprocess handshakes. *Information Processing Letters*, 34:131–138, Apr. 1990.

[23] S. Ramesh. A new and efficient implementation of multiprocess synchronization. In *Proceedings of PARLE '87 (Parallel Architectures and Languages Europe)*, number 259 in Lecture Notes in Computer Science, pages 387–401 (vol.2). Springer-Verlag, June 1987.

[24] J. H. Reppy. Synchronous operations as first-class values. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 250–259, June 1988.

[25] J. H. Reppy. An operational semantics of first-class synchronous operations. Draft., Aug. 1991.

[26] J. H. Reppy and E. R. Gansner. eXene: A multi-threaded X window system toolkit. Technical report, Department of Computer Science, Cornell University, Ithaca, NY 14853, 1991. In preparation.