Adaptive Protocols for Negotiating Non-Deterministic Choice over Synchronous Channels

Erik D. Demaine
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
eddemaine@uwaterloo.ca

Abstract

In this paper, we propose several deadlock-free protocols for implementing the *generalized alternative construct*, where a process non-deterministically chooses between sending or receiving among various synchronous channels. We consider general many-to-many channels and examine in detail the special case of *fan* (many-to-one and one-to-many) channels, which are common and can be implemented much more efficiently. We propose a protocol that achieves an optimal number of message cycles per user-level communication, significantly improving on previous results. We propose several other "less aggressive" protocols, which may be more suitable for some applications and networks, and demonstrate how to adaptively switch between them and modify protocol parameters. Finally, we show how to maintain dynamic membership of channels, while avoiding race conditions that would prevent garbage collection of processes.

1 Introduction

CSP (Communicating Sequential Processes) [13, 14] initiated the area of distributed-memory concurrent programming, which is now a large area of research. Many of the ideas have been incorporated into a variety of concurrent-programming languages, including Concurrent ML [18], Facile [12], Fortran-M [8], and occam [21]. Common to these languages are two main concepts, synchronous communication over channels and non-deterministic choice, that provide a powerful message-passing abstraction.

Briefly, message passing is provided by basic send and receive primitives. They are synchronous in that the sender [receiver] waits for a matching receive [send] before continuing. The destination/source that is passed into the send/receive primitive is specified by a *channel*. A one-to-one channel is a uni-directional connection between two processes. Channels effectively correspond to the *port* abstraction (i.e., one of several message queues on a particular process), but are much more convenient

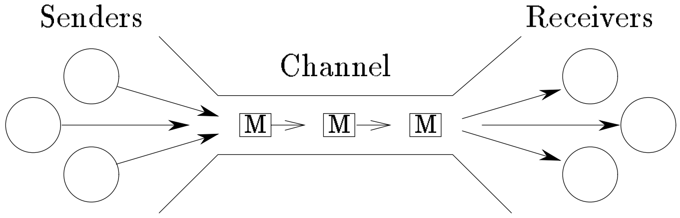

Figure 1: *A many-to-many channel. Attempting to send on a channel will block until one of the receivers attempts to receive a message from the channel, and vice versa. When there are multiple willing senders and/or receivers, the behavior is non-deterministic, although we draw it as a queue here.*

to program with. Many-to-many channels generalize to allow multiple senders and receivers (Figure 1); any send and receive operations on the same channel "match." An important primitive is the non-deterministic choice, originating from sequential languages proposed by Dijkstra [7], which does one out of a list of sends and receives, whichever can complete first by having a matching receive/send.

In this paper, we examine protocols for the efficient implementation of general non-deterministic communication over synchronous channels. This problem is typically ignored by either disallowing distributed systems (e.g., Concurrent ML [18] and Facile [12]) or restricting the non-deterministic-choice construct to allow only receives (e.g., Fortran-M [8] and occam [21]). We clearly do not want to restrict ourselves to sequential systems, since concurrent programs often have a large amount of parallelism that should be exploited using a distributed system. (For this reason, people often use the term "parallel languages" for concurrent languages.)

The so-called "generalized alternative construct," that is, one that supports both send and receive operations, is also important. Hoare [13] noted this as an important feature when he first proposed CSP, because it is useful for describing some applications (e.g., bounded buffers and interleaved conversations [3]) and removes the asymmetry between sends and receives. While the generalization is not known to be required for any applications, it greatly simplifies programming. We believe that the construct is in fact as general as needed, since it can easily be used to implement such high-level communication operations as Reppy's higher-order concurrency [18].

Unfortunately, it is difficult to support send operations in a non-deterministic-choice construct. Silberschatz [22] and Van de Snepscheut [23] examined cases where the construct is easy to implement if only certain processes use it. Inefficient implementations of the general construct include those that use global information (e.g., a central coordinator) [19], require an unbounded amount of time [9], or use an unbounded amount of communication [2, 20]. An overview of these techniques can be found in [4].

Buckley and Silberschatz [4] were the first to propose a protocol that avoids all three of the inefficiencies above. Bagrodia [1] later proposed a protocol that uses fewer messages. Unfortunately, both protocols are prone to deadlock for cyclic communication patterns [16]. Bornat and Knabe independently discovered deadlock-free protocols. Bornat [3] implements the generalized alternative construct using a receive-only construct supporting one-to-one channels. His protocol requires six control messages per user-level communication, whereas our deadlock-free simple protocol (Section 2.4) uses only three for one-to-one channels. Knabe's approach [15] is more similar to ours in that he uses asynchronous (buffered) messages to implement the (synchronous) generalized alternative construct for many-to-many channels. His protocol requires six or seven control messages in the general case, and creates an extra process for each channel, although he avoids multicasting, which is present in some of our approaches.

In the development of our protocols, we focused on efficiency, that is, reducing the number of messages and message cycles. While our deadlock-free simple protocol (Section 2.4) is similar to Knabe's approach [15], we improve on it significantly with the deadlock-free fast protocol (Section 2.6). We also show how to construct a "mega protocol" that adaptively switches between protocols and adjusts performance parameters (Section 3). In Section 4, we show how to maintain the membership of a channel without race conditions. Finally, we conclude in Section 5.

2 Protocols

In this section, we present four protocols to support processes that each attempt multiple send and receive operations over synchronous uni-directional channels and complete exactly one operation, whichever is ready first, breaking ties arbitrarily. Section 2.1 discusses a special class of channels called fan channels, which is what most of our protocols consider. In Section 2.2, we describe the general framework that our protocols are based on. Sections 2.3 and 2.4 discuss basic deadlock-prone and deadlock-free protocols, respectively. Corresponding "fast" protocols are presented in Sections 2.5 and 2.6. Finally, we demonstrate the interoperability of our protocols in Section 2.7.

2.1 Fan Channels

While several concurrent languages (e.g., Concurrent ML [18] and Facile [12]) support many-to-many channels, we will focus on implementing one-to-many and many-to-one channels, which together constitute what we call *fan channels*. In principle, many-to-many channels can also be implemented using our protocols (see Sections 2.4 and 2.5), but they will be inefficient, as is any distributed protocol implementing them. In addition, we feel that fan channels capture the most useful properties of many-to-many channels, and have no trouble restricting ourselves to them in practice. Finally, many-to-many channels can be implemented using a many-to-one channel, a one-to-many channel, and a process in between (Figure 2).

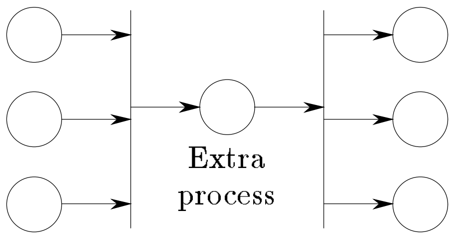

Figure 2: *Building a many-to-many channel using fan channels.*

As far as we know, we are the only ones to have looked at the intermediate restriction of fan channels; most restrict their attention to one-to-one channels. This is unfortunate, since fan channels can be implemented with the same efficiency (for "less aggressive" protocols such as the one described in Section 2.3). The basic property that allows an implementation to be more efficient than many-to-many channels is the notion of a single *owner* involved in all communications, that is, the unique sender [receiver] of a one-to-many [many-to-one] channel. We will call processes on the other end of the channel *members*. By convention, we represent one-to-one channels as many-to-one, that is, choose the owner to be the unique receiver.

2.2 General Framework

All of our protocols can be described as special cases or slight modifications of a general algorithm called Comm. Comm non-deterministically chooses exactly one element from a set $\mathcal{C}$ of synchronous communications (sends and receives). It negotiates this by sending asynchronous (buffered) messages over a reliable communication subsystem. Hence, Comm has the ability to send a message to a specified process or receive a message from an arbitrary process, but has to provide the channel abstraction itself. We shall first present an overview of the protocol used by Comm, and later give a formal description.

Comm uses three kinds of *control messages* to negotiate communication. The first, called notify, informs the recipient of willingness to communicate over the channel. In response, a process may send a request message, which signifies that it is committed to determining whether the communication will succeed (what we call "half-committed"). In response to this, a process can send a cancel or another request message. A cancel message causes the recipient to remove itself from its half-commitment and forget the corresponding notify. A second request message implies full commitment, causing user-level communication.

Comm is parameterized by a set $\mathcal{I} = \mathcal{I}(\mathcal{C}) \subseteq \mathcal{C}$ that specifies the process's *initiative*. In other words, it lists each $c \in \mathcal{C}$ that the process is aggressive on, and will send initial notify messages about.

Before we describe Comm in detail, let us give some terminology. Each $c \in \mathcal{C}$ has a corresponding channel, denoted $\mathsf{channel}(c)$. Each control message m has two fields (other than its type, notify, request, or cancel). The first, denoted $\mathsf{channel}(m)$, is the channel that m refers to. The second, denoted $\mathsf{sender}(m)$, is the name of the process that sent m. We say that m *matches* a communication $c \in \mathcal{C}$ if $\mathsf{channel}(m) = \mathsf{channel}(c)$. Finally, we say that m matches another message m' if $\mathsf{channel}(m) = \mathsf{channel}(m')$ and $\mathsf{sender}(m) = \mathsf{sender}(m')$.

Then Comm works as follows:

1. Set the level ℓ of commitment to 0.

2. For each $c \in \mathcal{I}$, send a notify message m with $\mathsf{channel}(m) = \mathsf{channel}(c)$ to processes on the other end of $\mathsf{channel}(c)$.

3. While $\ell < 1$:

   (a) If $\ell = 0$ and there is a request message m matching some $c \in \mathcal{C}$, send a request message to $\mathsf{sender}(m)$, $c^* \leftarrow c$, $m^* \leftarrow m$, and $\ell \leftarrow 1$.

   (b) If $\ell = 0$ and there is a notify message m matching some $c \in \mathcal{C}$, send a request message to $\mathsf{sender}(m)$, $c^* \leftarrow c$, $m^* \leftarrow m$, and $\ell \leftarrow \frac{1}{2}$.

   (c) If $\ell = \frac{1}{2}$ and there is a request message m matching m^*, $\ell \leftarrow 1$.

   (d) Remove any messages m just matched in the steps above from the lists they were found in.

   (e) If $\ell < 1$, receive a control message m and process it as follows:

      i. If m is a notify or request message, add it to the list of messages of that type (such lists are persistent between calls to Comm).

      ii. If m is a cancel message, remove the corresponding notify message from its list. Furthermore, if $\ell = \frac{1}{2}$ and m matches m^*, $\ell \leftarrow 0$.

4. For each $c \in \mathcal{I} - \{c^*\}$, send a cancel message m with $\mathsf{channel}(m) = \mathsf{channel}(c)$ (similar to Step 2).

5. Send a cancel message for c^* except to $\mathsf{sender}(m^*)$.

6. Execute c^* (either send a user-level message or wait for one to arrive).

To simplify the description of Comm, we have omitted details of sequencing. For each channel x, a process maintains the number of notify messages m it has sent with $\mathsf{channel}(m) = x$. This value is appended to each notify message m, $\mathsf{channel}(m) = x$, and any request message responding to m includes a copy of the value. Hence, Comm can recognize out-of-date request messages, which it can simply discard because a cancel message has already been sent and will be considered a reply.

Note that Comm can be called in between arbitrarily long computation blocks without slowing down other processes, since every notify message sent in a call to

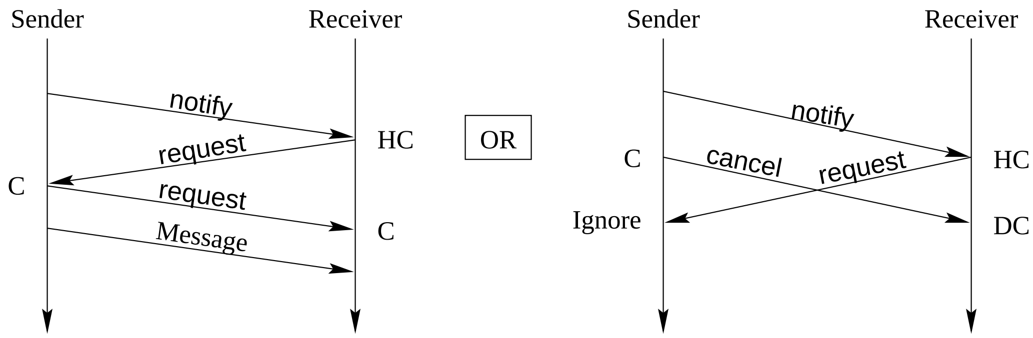

Figure 3: *The successful and unsuccessful scenarios in the simple protocol for a many-to-one channel. HC, C, and DC stand for half-commit, commit, and decommit, respectively.*

Comm is later cancelled or requested in the same call to Comm. This means that any request messages that arrive after that call will effectively be responded to even though the process is computing.

2.3 Simple Protocol

The *simple protocol* is described exactly by Comm, where a channel $x \in \mathcal{I}$ if and only if the process is a member of x, i.e., it is not the owner of x. In this case, control messages (except cancels) between two processes about a particular channel are exchanged sequentially (Figure 3). The main advantage of the simple protocol is that notify messages are only sent to one process (the owner) per channel.

The simple protocol is somewhat slow, requiring three control messages for each user-level (synchronous) communication. The sender can piggyback the user-level message onto its request message, resulting in three message cycles per user-level communication, or 3 MC/UC. For one-to-many channels, this can waste network resources, since the process is only half-committed when it piggybacks. If we are not willing to take this risk, we need an extra message cycle to send the user-level message, resulting in 4 MC/UC. We use the MC/UC measurement because startup overhead and propagation delay typically dominate message-passing time, especially for small (e.g., control) messages.

2.4 Deadlock-Free Simple Protocol

The simple protocol described in the previous section is prone to deadlock for cyclic communication patterns. For example, consider n one-to-one channels connecting n processes in a circle, and suppose each process non-deterministically either sends a message to its clockwise neighbor or receives a message from its counter-clockwise neighbor (Figure 4). Each process will first send a notify message to its clockwise neighbor (Figure 4(a)), receive a notify message from its counter-clockwise neighbor,

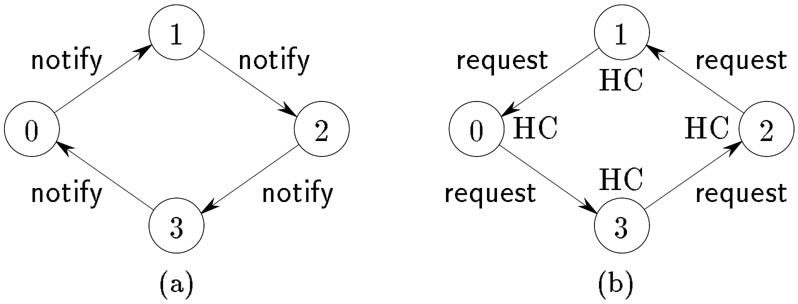

Figure 4: *Deadlock scenario for the simple protocol. The* notify *messages in (a) cause half-commitment and* request *messages in (b). The* request *messages must be ignored because of the half-commitment.*

and send a request message to its counter-clockwise neighbor (Figure 4(b)). At this point, every process will be half-committed to receiving, and so must ignore the request message, resulting in deadlock.

One way to make the simple protocol deadlock-free is to prevent deadlock completely (*deadlock prevention*). To do this, we must first realize that Comm-based protocols can be run in the "reverse direction." The *reverse* of a protocol corresponds to complementing the set $\mathcal{I}$. Hence, in the reverse-simple protocol, owners send a notify message to every member of the channel (unfortunately, this must be done with a reliable multicast operation). If the owner chooses to communicate on another channel, it multicasts a cancel message.

We can combine both protocols to achieve a deadlock-free version of the simple protocol. Basically, a process p uses the simple protocol on processes $< p$ (according to some total order of processes), and uses the reverse-simple protocol for other processes. That is, owners notify lesser processes, and members notify the owner if it is lesser. This can be generalized to many-to-many channels by saying that processes notify all lesser ones on the other end of the channel. This combination disallows cycles of notify messages and hence prevents deadlock.

In Section 2.6, we discuss a technique for recovering from deadlock after it is believed to have occurred. This technique can in fact be used with the simple protocol, and may be more efficient (depending on the application) since it avoids multicasting.

2.5 Fast Protocol

In the previous section, we saw how to combine the simple protocol and the reverse-simple protocol "mutually exclusively," in that each protocol handles transactions between a set of process pairs, and these two sets are disjoint. If we allow deadlock, we can in fact run both protocols simultaneously. In terms of Comm, we simply choose $\mathcal{I} = \mathcal{C}$.

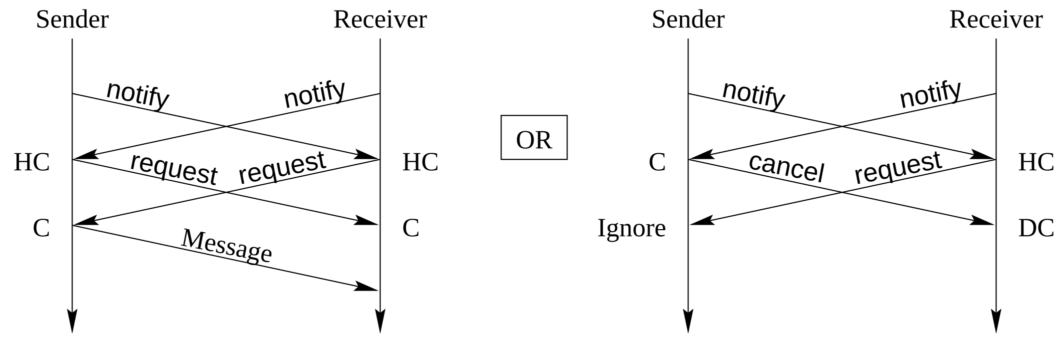

Figure 5: *The successful and unsuccessful scenarios in the fast protocol.*

Although this protocol, which we call the *fast protocol*, has twice as much message volume in the worst case, only 2 MC/UC are required for the control messages (Figure 5). This is clearly optimal for synchronous message-passing. If we choose not to piggyback the message onto the sender's request message, 3 MC/UC are needed.

We have described the fast protocol in a way that assumes nothing about the channel types, so it can in fact be used to implement communication over many-to-many channels.

2.6 Deadlock-Free Fast Protocol

As with the simple protocol, the fast protocol is subject to deadlock for cyclic communication patterns. We cannot use the same deadlock-prevention technique, however, since the reverse of the fast protocol is just the fast protocol itself (that is, it is symmetric), and hence it makes no sense to combine them. We instead adopt a deadlock-avoidance scheme, which is based on a timer.

When a certain amount of time passes in which process p receives no control messages, and p is half-committed, then p suspects that deadlock has occurred. In this case, p sends a relief message to processes that are half-committed to it (that is, p has received request messages from them). A relief message m relieves a process from its half-commitment to sender(m), causing it to half-commit or commit to a different alternative, if one exists; if not, the process is clearly not part of a deadlock, and simply repeats its request message. It should be obvious how to modify Comm to implement this idea; we simply set a timeout on receiving control messages in Step 3(e) (after which we send relief messages), and add a case for relief messages as Step 3(e)(iii).

A process suspecting deadlock cannot send relief messages to every process half-committed to it, because this can simply reverse the cyclic dependence, resulting in another deadlock. Repeating this procedure leads to a livelock. To solve this, processes only send relief messages to lesser processes (according to the total order).

Protocol	Successful		Unsuccessful	
	Cycles	Messages	Cycles	Messages
Simple	4	4	2	3
Simple + piggybacking	3	3	2	3
Fast	3	$2m+3$	2	$2m+2$
Fast + piggybacking	2	$2m+2$	2	$2m+2$

Table 1: *Summary of our protocols.* $im+j$ *denotes* i *multicast and* j *unicast messages.*

This ensures that at least one process in the cycle will not be relieved, but at least one will, so the deadlock will be resolved. Assuming that the timeout is sufficiently long, a deadlock will result in only one set of relief messages, which is a small cost for resolving the unlikely event of deadlock.

2.7 Interoperability

One property of Comm is that $\mathcal{I}$ can be chosen arbitrarily at each process, as long as at least one side of each potentially communicating process-pair sends a notify message. As a result, the simple and fast protocols are clearly interoperable, since they are both special cases of Comm. A process can also choose which protocol to use on a per-channel basis.

One way to switch between the two protocols is for the owner to have "control" over the protocol that is used. By default, everyone uses the simple protocol for that channel. If the owner decides to change to the fast protocol (e.g., the programmer issues a "hint"), then it can start sending notify messages. Comm specifies that members respond, so the entire channel has effectively switched to the fast protocol until the owner switches back and stops sending notify messages. So in fact we can notify only certain members, resulting in a "partial fast" protocol; we will examine this more in Section 3.1.

As noted in Section 2.4, we can use the deadlock-avoidance technique from the previous section with both the simple and fast protocols. Using reasoning similar to the above, these two protocols can be mixed. We can also mix with deadlock-prone techniques by viewing them as having a timeout of ∞ (of course, deadlock freedom is not guaranteed).

In conclusion, the protocols we have discussed are all related and interoperable. They are summarized in Table 1.

3 Adaptive Extensions

If we want a protocol that uses fewer message cycles than the simple protocol, multicasting is necessary. This may or may not improve the communication speed, depending on the application. In this section, we will see how to adaptively choose between the two deadlock-avoidance protocols, as well as computing user-specifiable options.

It will become clear that we need experimentation to evaluate the effectiveness of the various schemes, depending on the application and network. The main point of this section is that there are several methods available for protocol adaptation.

3.1 Simple-Fast Protocol Switch

As in Section 2.7, we adopt the idea that the owner of each channel decides the protocol to be used on that channel. We essentially want to determine whether it is worthwhile for the owner to send the extra notify messages for a channel c.

One way to measure this is as follows: each time the owner has the possibility of communicating on c, make note whether it actually communicates on c. If in the last k trials, communication is often successful, then it seems to make sense for the owner to send extra notify messages to speed up this process. On the other hand, if most communications are unsuccessful, it is pointless to send extra notify messages for c.

The exact cut-off point α in this *binary* approach can be specified by the user. However, it is unlikely to capture the fact that, if c has many members, it is expensive to multicast extra notify messages. If we mainly receive messages from a small subset of the members, it may make sense to send notify messages only to them.

One way to solve this is by sending a notify message with a certain probability. That is, for each member, we flip a pseudo-random coin with a certain weight, and if it comes up tails, we send a notify message to that member. If the owner determines that the probability of successfully communicating with a particular member i is p_i (via a discrete analysis), then it sends a notify message to i with probability $f(p_i)$. For example, f could be a linear function, that is, $f(p) = \max(1, \beta p)$ for some β.

Note that the sum of the p_i's is the probability p_c of successfully receiving on channel c. Hence, this *probabilistic* approach is related to the binary one. In particular, we could choose $f = \lceil p_c - \alpha \rceil$ to exactly mimic the binary approach, or choose $f(p) = \lceil p - \alpha \rceil$ to apply the binary approach on a per-member basis.

3.2 Piggybacking Toggle

Another question is when we should piggyback the user's message on the first request message. Basically, we do not want to append a user message that will not be used.

We can employ a simple binary scheme as follows. A sender (the owner for one-to-many channels, a member for many-to-one channels) makes note of the last k first request messages it sent for channel c, and whether it succeeded (second request message) or was rejected (cancel message). The sender can hence estimate the probability of a request message for c being accepted. If this is sufficiently high, and the message is not too large, the user message should be piggybacked onto the request message. On the other hand, if the probability is low, or the message is large, it is too costly to piggyback.

Owners of one-to-many channels can keep track of the probability on a per-member basis. This allows some members to accept messages frequently even if others do not. Note that it makes sense to send a request message to the process that has the highest probability of accepting, if multiple notify messages accumulate during computation.

We can generalize this idea to a probabilistic scheme as in the previous section. Let p denote the probability that a first request message will be accepted by a particular process under consideration (that is, we are about to send a first request message to that process). Then we piggyback the user's message with probability $g(p)$.

3.3 Deadlock Timeout

It also seems useful to adaptively compute the deadlock-timeout value. If it is too small, processes will falsely predict deadlocks; for example, if we timeout before a control message could actually travel over the network, we will constantly think that deadlock has occurred. To avoid this, we can multiply the timeout value by a constant $k > 1$ whenever a timeout occurs. This exponential growth will quickly leave this situation.

However, the occasional deadlock causes the timeout value to grow and deadlock detection to worsen. For long-running applications, the timeout value will quickly approach infinity (since it follows an exponential growth). To avoid this, we multiply the value by a constant $m < 1$ whenever we successfully complete a user-level communication, providing an exponential decay. Since deadlock is sufficiently rare, this should inhibit any major growth past the cutoff point between too short and too long a timeout value.

One could argue that once we have grown to a sufficiently large value, we can stop the adaptive extension and leave the timeout value fixed. However, this fails to take into account variable network delays. Assuming an appropriate choice of m, the exponential decay should also be slow enough, so we should rarely make the timeout value too small and need to increase it by k.

3.4 Exploiting Common Concurrency Structures

It is important to examine what programmers actually do with channels, to aid in the optimization of the protocols. One common structure is to have a server process that constantly non-deterministically either receives a request (on a many-to-one channel) or sends a reply for a completed operation (if any exist). A call to Comm sends a round of notify messages for the reply channels, and if the server chooses to receive a request, it also sends a round of cancel messages. It may be possible to specify that the server will almost always be executing algorithm Comm, and hence will be responsive to control messages. In this case, we can often avoid the round of cancel messages and shortly after repeating a round of notify messages. In other words, processes will know that they can always send request messages on reply channels, even though the server frequently succeeds in communications.

If we are not careful how we implement this idea, members will half-commit to request messages and block arbitrarily long periods of time (up to the amount of computation the server does). It may be worthwhile to split the server into two threads, which handle control messages and computation respectively, to ensure responsiveness to control messages. This doubling of the number of threads may be too expensive, depending on the system being used. An alternative is available for message-passing systems such as PVM 3.4 [11], where we can interrupt the computation to deal with each control message.

Another common structure is a devoted send or receive, that is, a send to or receive from a single process, executed without other choices. While non-deterministic choice is a powerful construct, it is not always needed. For example, suppose we have the following master-slave model. Slaves repeatedly receive jobs from the master and send their results back to the master, both of which are devoted.

Devoted communication can be exploited as follows. Suppose that member p is guaranteed not to decide to do something else. Then p can state this in the notify message, which will persist at the other end until it is used. If p is sending, the message can be appended to the notify message. Otherwise, the message can be appended to the request message that follows, which signifies a full commit. Hence, we achieve the optimal 2 MC/UC without excess piggybacking.

The user may specify a hint or an adaptive extension may determine that the members (e.g., slaves) are often devoted. In this case, it may not be necessary for the owner (e.g., master) to send the extra notify messages, even if other adaptive extensions say that the fast protocol is useful.

Devoted communications are also useful for the deadlock-avoidance scheme. Basically, processes do not have to send relief messages to processes that are devoted. In fact, it makes sense to choose devoted communications over others to avoid deadlock. In this sense, devoted communications tend to get higher priority, since they are more efficient.

4 Dynamic Membership

In this section, we consider the following dynamic problem: a process can send the name of a channel it owns to another process, the recipient can send this name to another process, and so on. The owner must maintain a list of all processes that hold the name of the channel, except itself (that is, the list of members). The list must be non-empty if and only if a process will hold the name at some time in the future, assuming the owner does not send the name to any more processes. This property can be used to detect whether, during a call to Comm, there will be any processes willing to communicate on a channel. Detecting this is important for garbage collection of processes [5, 6], where we want to detect when a process has lost all connections to other processes and is waiting to communicate, in which case it can be discarded. A preliminary version of this section appeared in [6].

When a member discards the name of a channel, it sends a *remove* message to the owner. However, *add* messages are not as simple.

Suppose that members only send add messages immediately upon receiving a new channel name. Then the following scenario leads to an unfortunate situation: the sole member of the channel sends the channel's name to another process, and immediately deletes its copy. This will cause a remove message to be sent, which will likely arrive before the add message. Thus, for a period of time, the owner believes that there are no members, even though there will be a member soon.

A similar race condition occurs if only the sender issues add messages, allowing arbitrary network delays[1]. Hence, both the sender and the receiver need to send add messages in such a way that the second-received one is ignored. This will remove any possible race conditions.

The owner cannot simply maintain a list of first-received add messages, and remove one from the list when it receives the second add message with the same source and destination. This would not work if a process sent a channel name to another process twice in a row (simply suppose that the sender's add messages both arrived at the owner before the other add messages, and we can arrive at race conditions similar to those above).

To solve this, we split add messages into send-side and receive-side add messages, sent by the sender and receiver of a channel name, respectively. Several send-side [receive-side] add messages with the same source and destination may accumulate in the list, and are individually removed by matching receive-side [send-side] add messages.

We have used this membership protocol in two implementations of higher-order concurrency [17], on top of Java [5] and PVM [6]. In the latter implementation, we exploited the following fact: the membership protocol can be modified to allow processes to "pack" channel names into a buffer, and have this buffer travel through several processes before it is unpacked. This is achieved by storing the original sender and receiver with the channel name, which is used in the receive-side add message, sent during the unpacking phase.

[1] Note that we will assume, for any two processes p and q, the messages from p to q are delivered reliably in the order they were sent. This corresponds exactly to the quality-of-service that PVM [10] provides.

5 Conclusion

In this paper, we have presented several efficient deadlock-free protocols to implement the generalized alternative construct over synchronous channels. We found that *fan* (many-to-one and one-to-many) channels allow much higher efficiency than general many-to-many channels. In particular, the dynamic membership problem has an efficient solution, while avoiding a race condition, thereby allowing garbage collection of processes. The adaptive extensions allow effective choice between our protocols and selection of protocol parameters.

The simple protocol and dynamic-membership protocol were used in two implementations of higher-order concurrency [5, 6]. In the future, we plan to implement the other protocols and evaluate their relative performance. In particular, we are interested in how well the suggested adaptive methods work for various applications and networks.

Acknowledgments

We wish to thank David Taylor for the idea of adaptive extensions, in particular the suggestion of the binary approaches, and for valuable comments on the paper. We also thank Frederick Knabe for providing important references on related work, and Thomas Kunz for initial discussions on the protocols. This work was supported by the Natural Sciences and Engineering Research Council (NSERC).

References

[1] Rajive 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, Massachusetts, May 1986.

[2] Arthur J. Bernstein. Output guards and nondeterminism in “Communicating sequential processes”. *ACM Transactions on Programming Languages and Systems*, 2(2):234–238, April 1980.

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

[4] 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, April 1983.

[5] Erik D. Demaine. Higher-order concurrency in Java. In *Proceedings of the Parallel Programming and Java Conference (WoTUG20)*. IOS Press (Netherlands), April 1997.

[6] Erik D. Demaine. Higher-order concurrency in PVM. In *Proceedings of the Cluster Computing Conference*, Atlanta, Georgia, March 1997. World Wide Web. `http://www.mathcs.emory.edu/~ccc97`.

[7] E. W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(18):453–457, 1975.

[8] Ian Foster and K. M. Chandy. Fortran M: A language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 26(1):21–35, 1995.

[9] N. Francez and M. Rodeh. A distributed abstract data type implemented by a probabilistic communication scheme. In *Proceedings of the 21st Annual Symposium on Foundations of Computer Science*, pages 373–379, Syracuse, New York, October 1980.

[10] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine — A User's Guide and Tutorial for Networked Parallel Computing.* The MIT Press, Cambridge, MA, 1994.

[11] G. A. Geist, J. A. Kohl, P. M. Papadopoulos, and S. L. Scott. Beyond PVM 3.4: What we've learned, what's next, and why. Unpublished manuscript. World Wide Web. `http://www.epm.ornl.gov/pvm/nextGen.ps`.

[12] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.

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

[14] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[15] Frederick Knabe. A distributed protocol for channel-based communication with choice. Technical Report ECRC-92-16, European Computer-Industry Research Centre, München, Germany, 1992.

[16] Frederick Knabe. A distributed protocol for channel-based communication with choice. *Computers and Artificial Intelligence*, 12(5):475–490, 1993.

[17] John H. Reppy. *Higher-order concurrency.* PhD thesis, Dept. of Computer Science, Cornell University, June 1992.

[18] John H. Reppy. Concurrent ML: Design, application, and semantics. In Peter E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *Lecture Notes in Computer Science*, Hamilton, Ontario, Canada, 1993. Springer-Verlag.

[19] F. B. Schneider. Synchronization in distributed programs. *ACM Transactions on Programming Language and Systems*, 4(2):125–148, April 1982.

[20] J. S. Schwarz. Distributed synchronization of communicating sequential processes. Technical report, Dept. of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland, July 1978.

[21] SGS-THOMSON Microelectronics Limited. *occam 2 Reference Manual*. Prentice Hall International Ltd., 1988.

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

[23] J. L. A. Van de Snepscheut. Synchronous communication between asynchronous components. *Information Processing Letters*, 13(3):127–130, December 1981.