Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems

Barbara Liskov
Liuba Shrira

MIT Laboratory for Computer Science
Cambridge, MA. 02139

Abstract

This paper deals with the integration of an efficient asynchronous remote procedure call mechanism into a programming language. It describes a new data type called a *promise* that was designed to support asynchronous calls. Promises allow a caller to run in parallel with a call and to pick up the results of the call, including any exceptions it raises, in a convenient and type-safe manner. The paper also discusses efficient composition of sequences of asynchronous calls to different locations in a network.

1. Introduction

This paper describes a new data type called a *promise*. Promises were designed to support an efficient asynchronous remote procedure call mechanism for use by components of a distributed program. A promise is a place holder for a value that will exist in the future. It is created at the time a call is made. The call computes the value of the promise, running in parallel with the program that made the call. When it completes, its results are stored in the promise and can then be "claimed" by the caller.

The development of promises was motivated by a new communication mechanism, the *call-stream*. Call-streams were invented as part of a project in heterogeneous computing [14], in which programs written in different programming languages, and running under different operating systems on different hardware, can use one another as components over a network. Call-streams combine the advantages of remote procedure calls and message passing. Remote procedure calls have come to be the preferred method of communication in a distributed system because programs that use procedures are easier to understand and reason about than those that explicitly send and receive messages. However, remote calls require the caller to wait for a reply before continuing, and therefore can lead to lower performance than explicit message exchange.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-83-K-0125, by the National Science Foundation under grant DCR-8503662, and by the Hebrew Technical Institute Postdoctoral Fellowship.

Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation
Atlanta, Georgia, June 22–24, 1988

Call-streams allow a sender to make a sequence of calls to a receiver without waiting for replies. The stream guarantees that the calls will be delivered to the receiver in the order they were made and that the replies from the receiver will be delivered to the sender in call order. Provided that the receiver executes the calls so that they appear to occur in call order, the effect of making a sequence of calls is the same as if the sender waited for the reply to each call before making the next.

New linguistic mechanisms are needed to make full use of streams. For example, suppose

```
a := p(x)
b := q(y)
```

are two calls on the same stream, and what is wanted is to begin the call of q without waiting for the reply to p. How can this be indicated? How can the results of the two calls be picked up without error or confusion? What happens if one of the calls signals an exception? Finally, suppose a communication problem makes it impossible to complete one of the calls; how is this indicated? Promises were invented to answer these questions in a way that preserves the merits of organizing programs using procedures and procedure calls without sacrificing the performance benefits of streams.

The design of promises was influenced by the *future* mechanism of MultiLisp [5]. Like futures, promises allow the result of a call to be picked up later. However, promises extend futures in several ways: Promises are strongly typed and thus avoid the need for runtime checking to distinguish them from ordinary values. They allow exceptions from the called procedure to be propagated in a convenient manner. Finally, they are integrated with the call-stream mechanism and address problems such as node failures and network partitions that do not arise in a single-machine environment.

Having introduced call-streams into a language, a natural next concern is stream composition. We would like to arrange streams into a pipeline in which the results of calls on one stream are used as the inputs of calls on the next stream. The main concern here is how to do the composition while retaining the performance benefits of the component streams. We investigate some linguistic mechanisms that support such compositions.

The remainder of this paper is organized as follows. In Section 2 we give a brief description of call-streams and describe how streams will be integrated into the Argus programming language, which is the context for our work on promises. Then in Section 3 we define promises and describe how they can be used for making calls over streams. We also discuss how promises can be used with forks of local processes and compare our mechanism to related ones in other languages. In Section 4 we deal with stream composition. We conclude with an evaluation of our mechanism.

2. Call-Streams

Call-streams are a language-independent communication mechanism. This section gives an overview of streams; a more complete description can be found in [14]. We also discuss briefly how streams are embedded into Argus.

We view a distributed program as made up of active *entities* that reside at different nodes of a network. Each entity resides completely at a single node; there may be several entities at a node. Two entities, a *sender* and a *receiver*, can be connected by a stream. The sender can make calls to the receiver over the stream. There are ordinary *RPCs*, in which the sender receives the reply to the call before making another call, and *stream calls*, in which the sender may make more calls before receiving the reply. In addition there are *sends*, which are like stream calls except that the sender is interested in the reply only when the call terminates abnormally. The underlying system takes care of delivering calls to the receiver in call order, and delivering replies to the sender in call order. The application code at the receiver is responsible for executing the calls so that they appear to happen in call order.

There are two reasons for using stream calls instead of RPCs: they allow the caller to run in parallel with the sending and processing of the call, and they reduce the cost of transmitting the call and reply messages. RPCs and their replies are sent over the network immediately, to minimize the delay for a call. Stream calls and their replies, however, are buffered and sent when convenient; in the case of sends, normal replies can be omitted. Buffering allows us to amortize the overhead of kernel calls and the transmission delays for messages over several calls, especially for small calls and replies.

A receiving entity provides one or more *ports*; these identify procedures that can be called from other entities. Each port has a unique name that can be used by the system to locate it when it is called. Typically, a receiving entity will provide many ports, each one corresponding to an operation that can be called by a client. Some ports are created when the entity first comes into existence; others can be created dynamically. Ports may be sent as arguments and results of remote calls.

A port is strongly typed. For example,

port (int) **returns** (real) **signals** (e1(char), e2)

describes a port that takes an integer argument. We are using the termination model of exception handling [11], in which a call can terminate in one of a number of conditions; in each case, results can be returned to the caller. Thus a call on the above port might terminate normally, returning a real, or it might terminate with exception *e1* or *e2*; it returns a character if it terminates with *e1* and returns nothing if it terminates with *e2*. Arguments and results are passed by value as discussed further below.

Ports are grouped together for sequencing purposes: only calls to ports in the same group are sequenced. Groups of ports define the receiving ends of streams. We require that ports in the same group all belong to the same entity because otherwise it would be expensive to control the sequencing of calls to them. Typically an entity determines the grouping of its ports when it creates them.

For example, a window system might provide a *create_window* port that is used to create a new window. When called, this port returns a number of newly-created ports that can be used to interact with the new window, e.g.,

```
create_window: port (...) returns (window)
window = struct [ putc: port (char),
                  putl: port (string),
                  change_color: port (string),
                  ... ]
```

All ports for a particular window might be placed in the same group, but ports of different windows might belong to different groups.

We assume that there may be concurrent activity within an entity. The separate activities should not share the same stream because this can introduce unwanted synchronization and even lead to deadlocks. We use *agents* to identify activities; agents define the sending ends of streams. An agent has a unique name and belongs to a single entity; there can be many agents belonging to the same entity.

An agent and a port group together define a stream: All calls sent by an agent to ports in a port group are sent on the same stream, and thus are sequenced. Calls made by different agents to ports in the same group are sent on different streams, as are calls made by one agent to ports in different groups.

Streams guarantee that messages arrive in good condition. They also guarantee exactly-once, ordered delivery of requests: Each call request or reply is delivered to the user code exactly once, the request for call n + 1 is delivered to the user code at the receiver only after the request for call n has been delivered to it, and the reply to call n + 1 is delivered to the user code at the sender only after the reply to call n has been delivered to it. Of course these semantics cannot be realized completely because of problems such as node crashes and network partitions. If the system is unable to live up to the guarantees, it *breaks* the stream. It does so only if the sender or receiver crashes, or there are serious communication problems. The system tries hard to deliver messages before breaking a stream, so there is no point in the caller repeating a call immediately when a stream breaks.

When the system at the receiving side breaks a stream, this means that further calls on that stream will be discarded at the receiver. Eventually, the system at the sender will also break the stream, either independently, or because communication from the receiver informs it of the break. When the system at the sending side breaks a stream, this means that any calls whose replies have not yet been received will never have replies. We rely on the language in which the calls were made to do something sensible, e.g., cause the calls to terminate with an exception.

A break at the receiver is either synchronous or asynchronous. A synchronous break happens after the reply to a call; that call and all calls before it will be unaffected by the break, but later calls will never receive replies. An asynchronous break happens independently of particular calls, and its effect on the outcome of already-processed calls is nondeterministic. Asynchronous breaks happen when there are communication problems, so replies to earlier calls may have been lost.

The sender can make a broken stream usable again by *restarting* it. A restart is equivalent to a break done by the system at the sender at that moment, followed by the *reincarnation* of the stream so that calls can be made on it in the future.

Two additional primitives are available to the sender. The first is a *flush*, which causes the sending of any buffered call requests on the flushed stream and the flushing back of replies at the other side. (Even without the flush, the system will send these messages eventually; the flush merely speeds this up.) The second is a *synch*. Synching not only does a flush, but it causes the caller to wait until all earlier calls on the stream have completed.

2.1. Argus

To use streams within a programming language, we need to identify the language features that correspond to entities, ports, and so on. In this section we explain briefly how this is done for Argus. The identifications introduced here serve as a basis for the remainder of this paper. The reader is referred to [12, 15, 16] for a complete description of Argus.

Argus provides active entities called *guardians*, each of which resides entirely at a single node of a network. Each guardian provides operations called *handlers* that can be called by other guardians. In creating a handler, the guardian defines two things: a

port that can be used to identify the handler in calls, and a procedure that will run when a call arrives to process it. This procedure is called automatically by the Argus system when it is time to process a call. Ports are grouped by various simple mechanisms, e.g., all ports of handlers created when a guardian is created belong to the same group.

A guardian can have many processes running inside it. Some of these are created when a guardian first starts to run;[1] others are created to run handler calls. Each such process will be associated with a unique agent.

When a handler call arrives at a guardian, the Argus system will delay its execution until all earlier calls on its stream have completed. (Calls on broken streams are discarded automatically, so user code never needs to deal with them.) In this way we make it easy for user code to ensure that calls on the same stream happen in order. (We may provide some explicit overrides to allow more sophisticated programs that process calls on the same stream in parallel.) Note, however, that calls on different streams can be processed in parallel.

For example, consider a mailer guardian with handlers *send_mail* and *read_mail*, both in the same group, and suppose it is being used by two clients, C1 and C2. If C1 calls *send_mail*, this call will start to run immediately. If C2 then calls *read_mail*, this call will also start to run immediately, since it is on a different stream than C1's call. Thus both calls may be running concurrently; each would be run by a different process and agent. If C1 now calls *read_mail* on the same stream as its call of *send_mail*, the processing of this call will wait until C1's *send_mail* call completes; the call can then start running, even though C2's call of *read_mail* may still be running.

Argus already supports RPCs. For example, C2 can make an RPC to the *read_mail* handler of the mailer guardian *g* by executing the statement

```
m: message := g.read_mail(u)
     except when no_such_user: ...
          when others: ...
          end
```

where *read_mail* signals *no_such_user* if *u* is not registered with the system. This call delays the calling process until the reply arrives, or until the system determines that the call cannot be completed. The Argus system terminates the call with the *unavailable* exception if communication is impossible at the moment, and with the *failure* exception if the call is an error, e.g., if guardian *g* no longer exists.

The example illustrates the Argus exception handling mechanism [11]. If a call terminates with an exception, control goes to the nearest **except** statement that contains an arm for the exception; an **others** arm handles all exceptions not named explicitly (*unavailable* and *failure* in the example). The **except** statement can be attached to the call statement as shown, or to any textually including statement.

One point ignored in the above discussion is that Argus computations run as atomic transactions. Atomic transactions allow us to make sense of the above concurrency, e.g., if the calls made by C1 and C2 send and read mail for the same user. They also allow us to make sense of computations in the presence of failures such as node crashes and lost messages. We will discuss transactions briefly later in the paper.

3. Promises

We concentrate now on how to support stream calls. Our solution is intuitive and straightforward. When a stream call is performed, the caller receives a "promise" for a result that will arrive later. A promise is an object that can be used to "claim" the result when it is ready. The type of the promise object reflects the possible results of the call, i.e., the type of the result in the normal case, and the names and types of the possible exceptions.

Associated with each handler type is a related promise type. For example, for

ht = **handlertype** (int) **returns** (real) **signals** (foo)

the related promise type is

pt = **promise returns** (real) **signals** (foo)

A promise type has a results part, listing the type or types of objects returned by the handler call in the normal case, and an exceptions part, listing the exceptions of the handler.

A promise object is in one of two states: blocked or ready. When first created as part of making a stream call, a promise is in the *blocked* state. When the call completes, the promise switches to the *ready* state. In this state, it has a value that indicates the outcome of the call, i.e., whether the call completed normally or with an exception, and the corresponding result in each case. Once a promise is ready it remains ready from then on and its value never changes again.

The *claim* operation waits until the promise is ready. Then it returns normally if the call terminated normally, and otherwise it signals the appropriate exception, e.g.,

```
y: real := pt$claim(x)
          except when foo: ...
               when unavailable(s: string): ...
               when failure(s: string): ...
          end
```

Here *x* is a promise object of type *pt*; the form *pt$claim* illustrates the way Argus identifies an operation of a type by concatenating the type name with the operation name. A promise can be claimed multiple times; the same outcome will occur each time. There is also a *ready* operation, which returns true if the promise is ready and false if it is blocked.

Broken streams are mapped into exceptions and then restarted automatically. As mentioned earlier, when there are communication problems, RPCs in Argus terminate either with the *unavailable* exception or the *failure* exception. *Unavailable* means that the problem is temporary, e.g., communication is impossible right now. It also means, however, that the system has tried hard so that there is no point in the user retrying the call right away. *Failure* means that the problem is permanent, e.g., the handler's guardian does not exist. Thus stream calls (and sends) whose replies are lost because of broken streams will terminate with one of these exceptions. Both exceptions have a string argument that explains the reason for the failure, e.g., *failure*("handler does not exist"), or *unavailable*("cannot communicate"). Since any call can fail, every handler can raise the exceptions *failure* and *unavailable*. We do not bother to list these exceptions explicitly. Thus *ht* and *pt* both have three exceptions, *foo*, *unavailable* and *failure*.

Arguments and results of handler calls in Argus are passed by value [7]. Only certain types of objects are permitted; for example, promises are not legal as arguments or results. Since the caller and the called module may have different representations for the data being communicated, the data are actually sent using an *external representation*. When a call is made, each argument is *encoded* by translating from its representation at the caller to the external representation; when the call message arrives at the receiver, the arguments are *decoded* by translating from the external representation to the internal one. Similarly, results are encoded at the receiver and decoded at the sender. Either encoding or decoding may fail. For example, when an argument or result is an object belonging to some abstract type, encoding and decoding are done by user-provided code, which may contain errors. Such a failure causes the call to terminate with the *failure* exception. In addition, when the problem happens at the receiver, the stream breaks so that further calls on that stream will be discarded.

[1] or recovers from a crash. Guardians can survive crashes as discussed further in [16].

A stream call has the form

x: pt := **stream** h(3)

where h is a handler of type *ht*. The semantics is as follows:

1. The call message is produced by encoding the arguments. If encoding fails, or if the stream being used is already broken, the call fails and signals the appropriate (*failure* or *unavailable*) exception. In this case no promise object is created, and control continues at the appropriate **except** statement.
2. If the call message is produced successfully, a promise object is created in the blocked state and returned to the caller, allowing the caller to continue.
3. Later, when the reply has arrived and it is convenient for the system to decode it, and after all promises for earlier calls on the stream are in the ready state, the reply message is decoded and the promise is changed to the ready state with the appropriate value. This value will be the (normal or exceptional) result returned by the call unless decoding failed, in which case the value will be *failure*("could not decode"). Decoding happens in a process and agent belonging to the system.
4. Alternatively, before the promise changes to the ready state, its stream may break or be restarted. In this case, the system changes the promise to the ready state with an appropriate value, e.g., *unavailable*("could not communicate").

If desired, the program need not create a promise; this is indicated by using **stream** as a statement instead of an expression. In such a situation, the result of the call is still decoded as described above and then discarded. Sends do not show up explicitly in Argus. Instead whenever a stream call is made to a handler with no normal results, the Argus implementation makes the call as a send.

Claims can be done in any convenient order. We do not require that the result of the i^{th} message be claimed before the result of the i + 1^{st}. As noted above, however, if the i + 1^{st} result is ready, then so is the i^{th}.

In addition to making calls, Argus programs can flush and synch streams. The **flush** or **synch** is done on a handler, e.g.,

synch h

The stream is the same one that would have been chosen in a stream call to that handler. In addition to doing a synch on the stream, **synch** allows the program to find out about whether earlier stream calls terminated normally or not. **Synch** returns normally only if all previous stream calls (since the last **synch** or regular RPC on the stream or since the stream incarnation was created) returned normally; otherwise, it signals *exception_reply*. It does not return information about which calls raised exceptions; to discover this, the program must use promises.

3.1. Example

As an example, consider a guardian that stores information about the grades of students and provides a handler, *record_grade*, that records a new grade for a student and returns an updated average for that student. In addition, a second guardian provides printing of grades information via its *print* operation. The program in Figure 3-1 uses one stream to record new grades for students and get their new averages, and a second stream to print an alphabetical list of students with their averages.

The first four lines of this program simply define abbreviations for data types used in the rest of the program. The first loop streams the calls of *record_grade* to the grades database, and stores the promises for the averages returned by these calls in array *a*. It uses the array operation *elements* to obtain the grades information for students in alphabetical order; *elements* is an iterator [10] that yields

```
% define some type abbreviations
sinfo = record [ stu: string, grade: grade ]
info = array [sinfo]
pt = promise returns (real)
averages = array [pt]

grades: info := % this information is pre-recorded and
                % organized alphabetically by student

begin
  a: averages := averages$create(info$low(grades)) % create new,
    % empty array with the same lower bound as the grades array

  % record grades
  for s: sinfo in info$elements(grades) do
    averages$addh(a, stream record_grade (s.stu, s.grade))
    end
  flush record_grade

  % print
  for i: int in averages$indexes(a) do
    stream print(make_string(grades[i].stu, pt$claim(a[i])))
    end
  synch print
  end except ... end
```

Figure 3-1: The Grades Example

the elements of the array from the low bound to the high bound. The elements are produced incrementally; each time an element is produced, the loop body is run with that element stored in variable *x*. The loop uses the array *addh* operation, which extends the array *a* by one and stores the new promise in the new element. When the loop is finished, the program flushes the call-stream to ensure that the last few calls (and replies) are sent out quickly.

The second loop claims the promises in the order they were generated (namely alphabetically by student name) and makes stream calls to the printer. It uses the *indexes* iterator, which produces the legal indices in the array. Since the averages are maintained in alphabetical order in array *a*, the results will be printed in order. Furthermore, the averages will be paired with the proper students because the elements in the two arrays, *a* and *grades*, are paired.

This example uses stream calls both to overlap processing of calls and to obtain the benefits of buffering messages for calls and replies. A considerable amount of overlapping is possible, since once all calls of *record_grade* have been initiated and the replies start to come back, the processing of calls at the grades database can be overlapped with the processing of the print requests. In addition, the example uses promises as a way of organizing replies in a convenient manner (e.g., in an order corresponding to the alphabetical ordering of the students), and it relies on the guarantee of streams that calls and replies are delivered in call order.

However, the example does not have as much overlapping as we would like. We cannot begin printing results until all calls to the grades database have been initiated. A better program would start printing as soon as averages can be claimed. We discuss such a program in Section 4.

3.2. Local Forks

Promises and stream calls allow a client to run in parallel with calls to a server, and pick up the replies in a convenient way. However, there are two parts to our semantics: the deferred result, which allows concurrency between caller and callee, and the ordered processing of the calls. It is clear that concurrency would be useful without the ordering. Therefore, in this section we extend our mechanism to include "forking" of local calls. In addition to creating promises by means of stream calls, we also allow them to be made by means of forks. A **fork** causes a call of a local procedure to run in

parallel with the caller. When the procedure terminates, its results are stored in the promise, which then becomes claimable.

The semantics of forking is as follows. Suppose *foo* is a local procedure,

foo: **proc** (a: array[int]) **returns** (int) **signals** (e)

and *pt* is the associated promise type

pt = **promise returns** (int) **signals** (e)

Then the statement

p: pt := **fork** foo(a)

where *a* is an array of integers leads to the following:

1. A new process (and agent) is created to run the call and the forked procedure is called within that process. The arguments are passed *by sharing* as discussed below; encoding or any kind of copying is not needed.
2. A promise is created in the blocked state and returned to the calling process, which then continues running. At this point both the caller and the called process are running in parallel.
3. When the called procedure terminates, the promise changes to the ready state with the result of the procedure as its value.

There are no lifetime problems caused by **fork**. Argus procedures can share objects but not variables; they have no free variables and Argus does not support call-by-reference. Objects reside in a heap rather than a stack and continue to exist until they are no longer referenced. Arguments (and results) are passed by sharing: a pointer to the argument object (in the heap) is passed to the called procedure.

Forked promises are a useful concurrency mechanism in certain kinds of programs. One place where this occurs is in construction and access of recursive data structures such as lists and trees. For example, promises can be used for parallel insertion and searching of elements in a binary tree in which the nodes of the tree are promises. If a search reaches a node that cannot be claimed yet, it waits until the promise is ready.

3.3. Discussion

In the preceding subsections we described the Argus promise mechanism and showed how promises integrate streams and local forked procedures. Using promises for asynchronous remote calls is entirely new. However, many languages have local concurrency of the "fork" variety. Our mechanism has advantages over others because it is both type safe and provides a convenient way for exceptions to be propagated from the forked process to other processes that need to know what happened. While some other languages provide type-safe mechanisms, e.g., Mesa [17] and Modula-2+ [18], none to our knowledge provides exception propagation.

As discussed earlier, the futures of MultiLisp [5] were an important influence on our work. In MultiLisp, an object of any type can be a *future* for a value that will arrive later. When the value is needed in a computation (e.g., for an addition), it is claimed automatically, and the claiming program waits if necessary.

The uniformity of treating all objects as futures can be convenient. However, futures have two disadvantages. First, they are inefficient to implement unless specialized hardware is available, since every object must be examined each time it is accessed to determine whether or not it is a future. Second, it is difficult to do anything very useful with exceptions. In MultiLisp, exceptions are turned into error values automatically, and information about the error value propagates through the expression that caused the future to be claimed and then through surrounding expressions. Such an approach makes it difficult for a program to determine the reason for the error value. This problem is discussed in [6], which proposes as a solution that programs should claim futures explicitly if necessary to ensure that the error value is discovered in a scope that knows what to do with it. Promises force all programs to be structured like this, so the structure of the program using promises will be identical to one using futures when safe exception handling is a concern.

4. Composing Streams

One kind of program structure that is likely to arise with stream calls is the cascading of results of calls on one stream to the inputs of calls on another stream. We can think of a cascade as composing a number of streams together. A composition can have an arbitrary number of levels; in each case, the output of the i^{th} stream becomes the input of the $i + 1^{st}$ stream, possibly with some local computation done along the way.

The grades example illustrates a two-level composition, with the results of the first stream (to the database) being sent on to the second stream (to the printer). However, as mentioned above, the program shown in Figure 3-1 does not do what we want since it delays streaming to the printer until all calls to the database have been started. Instead, we would like to stream the results from the database to the printer as they become ready, even if some calls to the database have not yet been made. Obviously, this overlapping of recording and printing becomes more important as the number of calls increases.

A further overlapping problem becomes apparent when there are more levels in the cascade. For example, suppose there are three handlers

read = **handler** ( ) **returns** (argtype1) **signals** (e1, e2)
compute = **handler** (argtype1) **returns** (argtype2) **signals** (e3)
write = **handler** (argtype2) **signals** (e4)

each connected to a different stream. The idea is to pipeline the requests for data from the *read* stream into the *compute* stream and then to pipeline the results of *compute* into the *write* stream. However, if we use a program like that in Figure 3-1, then

1. All calls to *read* must start before any calls to *compute* can be made.
2. All results from *read* must be claimed, and all calls to *compute* must be started, before any calls to *write* can be made.

In this section we discuss how to write programs that exhibit the desired kind of data flow. One thing to note, however, is that we are not interested in connecting the output of one stream directly to the input of another, e.g., as is done in pipes in Unix. Instead, we want results to return to the original caller who then sends them on to the next stream. This program form allows arbitrary *filter* computations to be done to "match" the two streams. For example, the result of a call to *record_grade* is only one of several inputs of a call to the printer. Another example concerns exceptions: if a call on the first stream signals an exception, the program might compensate for this either by manufacturing arguments for the call on the next stream or by omitting the call or by terminating the computation.

A good way to get the composition we desire is to introduce concurrency into the controlling program,[2] and run each loop (e.g., in the grades example) in a separate process. The first loop makes stream calls to *record_grade* and fills up a data structure with promises; the second loop consumes these promises and makes the stream calls to the printer. Since the loops are running concurrently, the first one need not run to completion before the second one starts and thus we obtain the desired overlapping.

[2] A bad way is to write a sequential program that explicitly multiplexes the use of the different streams.

4.1. Forks

Forks are a general concurrency mechanism, so they can be used to implement the concurrent loops program structure as shown in Figure 4-1. However, the resulting program is less than ideal, as discussed further below.

The first part of the figure defines two procedures, one to do the first iteration, and the other to do the second. The main program simply forks two processes, one running each of these procedures. The processes communicate by means of a shared queue of promises, *aveq*; the process that records grades stores the promises created for its stream calls into the queue, and the printing process gets the promises from the queue. The queue serves a similar function to the array of averages in Figure 3-1, but in addition it also synchronizes the two processes: when a process attempts to *deq* an element from the queue, it will wait if the queue is empty until an element is enqueued. Queues can be implemented using standard synchronization mechanisms such as semaphores [3] or monitors [8].

The two local forks have associated promises so that the main program can wait until the two processes are done and so that any exceptions raised can be handled in the main program. Note that each procedure does a **synch** on its stream before it returns; therefore, if it returns normally, this means all calls it made completed successfully. (Of course, doing things right when only some of the grades have been recorded requires more sophistication than is shown in this program. Argus provides this ability by allowing computations to run as atomic transactions, as discussed further in the next section.)

As mentioned above, the program is less than ideal: it is awkward, and it has a termination problem. It is awkward to have to define the procedures *use_db* and *do_print*; it would be more convenient to write the code of the procedures right where they are used. Having to create and claim promises when there are no normal results is also awkward. The termination problem arises if the recording process terminates early because of a communication problem; in this case the printing process may hang forever waiting to dequeue the next promise from the queue.

The termination problem will arise in every stream composition; in each case, we will need the ability to treat the processes as a group and to allow a process in the group to cause the termination of the other processes in the group. Furthermore, programming proper termination explicitly can be quite tricky. Therefore, rather then fixing this program, we instead go on in the next section to define a different linguistic mechanism that overcomes both difficulties.

```
pt = promise returns (real)
pt1 = promise signals (cannot_record)
pt2 = promise signals (cannot_print)

use_db = proc (grades: sinfo, aveq: queue[pt]) signals
        (cannot_record)
  for s: sinfo in sinfo$elements(grades) do
    queue[pt]$enq(aveq, stream record_grade (s.stu, s.grade))
    end except when others: signal cannot_record end
  synch record_grade
    except when others: signal cannot_record end
  end use_db

do_print = proc (grades: sinfo, aveq: queue[pt])
        signals (cannot_print)
  for i: int in sinfo$indexes(grades) do
    ave: pt := queue[pt]$deq(aveq)
    stream print(make_string(grades[i].stu, pt$claim(ave)))
    end except when others: signal cannot_print end
  synch print except when others: signal cannot_print end
  end do_print

% composing the streams

  aveq: queue[pt] := queue[pt]$create( )
  p1: pt1 := fork use_db (grades, aveq)
  p2: pt2 := fork do_print (grades, aveq)
  pt1$claim(p1) except when cannot_record: ... end
  pt2$claim(p2) except when cannot_print: ... end
```

Figure 4-1: Using Forks

4.2. Coenters

What we would like is a way to identify the set of processes that implement a stream composition, so that they can be terminated properly when problems arise. In addition, it would be convenient to write the forked code inline. Argus already provides a mechanism that provides these abilities, the **coenter** statement. The **coenter** statement is similar to mechanisms in other languages (e.g., CSP [9]); it differs from these other mechanisms primarily because it offers a complete and sensible treatment of exceptions and early termination.

An example of the **coenter** is shown in Figure 4-2. A **coenter** statement contains a number of *arms*, each defining a computation to be run as a process. Execution of a **coenter** occurs as follows: A process and agent are created to run each arm, and these processes start to run in some undefined order; we will refer to these newly created processes as *subprocesses*. The process executing the **coenter** is halted, and remains halted until all the subprocesses complete. Completion can happen in one of two ways. First, each subprocess may simply finish execution of its arm. In addition, however, a subprocess can cause other subprocesses to terminate early. It does this by causing a control transfer outside of the **coenter**. In this case, any remaining subprocesses that are not yet finished are forced to terminate (as discussed further below) before the "parent" process can continue.

```
pt = promise returns (real)
aveq: queue[pt] := queue[pt]$create( )
coenter
  action % recording grades
    for s: sinfo in sinfo$elements(grades) do
      queue[pt]$enq(aveq, stream record_grade (s.stu, s.grade))
      end
    synch record_grade

  action % printing
    for i: int in sinfo$indexes(grades) do
      ave: pt := queue[pt]$deq(aveq)
      stream print(make_string(grades[i].stu, pt$claim(ave)))
      end
    synch print

  end % coenter
except when others: % in this case some work did not finish
```

Figure 4-2: Using Coenters

The figure implements the grades example using the **coenter**. Recording of grades is done in one arm; printing is done in the other. Each arm is run as an action, as discussed further below. If any stream problems are encountered in either arm, this will cause early termination of the **coenter** because these exceptions are not handled in the arms, but instead go to the **except** statement following the **coenter**. When such an exception is raised, the other arm (the one that did not cause the exception) is also terminated before the **except** statement is executed. For example, if the stream to the grades database breaks and this is discovered in the recording process because the next stream call raises an exception, then both the recording process and the printing process will be terminated immediately and control will continue in the parent process at the **except** statement. As mentioned earlier, without forced termination, the printing process might hang forever waiting to dequeue the next item from the queue.

Early termination of processes raises a question of safety. First, the process might be in the middle of a critical section; stopping it at such a point could leave damaged data. For example, if the printing process were terminated in the middle of dequeuing, this could leave the *aveq* in a damaged state. We solve this problem by delaying termination while a process is in a critical section. The Argus runtime system keeps track of how many critical sections a process is in and delays its termination until the count is zero; the system can do this because Argus provides a built-in critical section mechanism. To encourage a process to leave critical sections rapidly when it should terminate, we "wound" it by greatly restricting what it can do. For example, it cannot make any remote calls at such a point.[3]

Even if we solve the safety problem at this level, however, it recurs at a higher level. For example, recording grades is not something that should be done part way. This level of safety is provided in Argus by means of atomic transactions. An atomic transaction either completes entirely or is guaranteed to have no effect. Thus, running the recording process as an atomic transaction can ensure that if it is not possible to record all grades, none will be recorded. In the example, both processes are running as transactions.

When an action is terminated, we do not wait to terminate any calls that may be running elsewhere. Instead, the Argus system guarantees that it will find these computations and destroy them later [13]. Thus, for example, if the recording action terminates the coenter, we can do the termination immediately without waiting for calls to the grades database or the printer to complete.

Complete discussion of atomicity is beyond the scope of this paper; a full treatment of transactions, including integration into the **coenter**, can be found in [16]. Atomicity can also be used in the program in Figure 4-1; it is not limited to the **coenter**.[4] However, as mentioned earlier, the program with forks would have to indicate somehow what actions to abort in the case of a problem; with the **coenter**, this information is available from the program structure.

A **coenter** is the appropriate mechanism to use when the processes have no results and when the control structure is naturally hierarchical, i.e., it makes sense to delay the parent until the children are finished. When these conditions are not satisfied, a **fork** will be a better mechanism. Note that with hierarchical control there is no variable lifetime problem.

4.3. Discussion

The preceding sections discussed two ways to program stream composition: with forks and with the **coenter**. For stream composition, the **coenter** mechanism is preferable because it allows us to indicate directly what processes are involved in the composition, which in turns allows those processes to be terminated as a group if a problem arises. A secondary advantage of the **coenter** is that the code of the processes can be written in line.

In our discussion of composition, we assumed that there was one module that made the calls and ran the filters. (Recall that filters match results of one call to arguments of the next call.) It would be more efficient to send the filters along with the calls. This would allow a structure more like a real pipeline: results of calls on the first stream would be sent directly in calls on the second stream, and so on, without the need to first send the replies back to the original caller. However, such a structure is not practical in a heterogeneous system, since different programming languages may be in use at the ends of the streams. Sending filters makes more sense in a homogeneous system, but even here it raises difficult issues about how to implement the sending of code in messages.

The programs discussed above were organized around the streams, i.e., each process was in charge of making calls on a single stream. Another way to look at the problem is in terms of what happens to an individual data item, e.g., a grade is first recorded and then printed. With this structure, there would be a process per item. Each process would move its item from one stream to another, thus using all the streams of the cascade; synchronization would be needed to ensure that the calls on each stream were made in order. To implement such a structure, we need a way of spawning a dynamically determined number of processes. Although the concurrency can be obtained by forks, this leads to the same group termination problem discussed above. Instead a mechanism with automatic grouping is preferable. Argus provides such a mechanism, which extends the **coenter** to allow a dynamic number of processes.

Providing concurrency per data item is both an advantage and a problem. The advantage is that the extra concurrency may be useful since it permits us to run the filters in parallel. Clearly, this is of interest only if the filters are lengthy, and only on a multiprocessor. The problem is that there are many more processes to manage than in the process-per-stream case. This can impose a substantial burden on the system, and even slow down the program. Although a good system implementation might be able to do a reasonable job, the process-per-stream structure avoids the whole problem and therefore is better, at least on a sequential machine.

Instead of using **coenters** or forks, another possibility is to provide a construct that supports composition directly. Such a structure could lead both to simpler programs and better performance. However, it is not clear that stream composition is important enough to justify its own linguistic mechanism. At present, we believe that the **coenter** form is adequate for our needs.

5. Conclusions

Streams were invented to improve the performance of distributed programs. In our system we are interested in a very general class of applications written in a wide variety of languages. Our approach is to allow programs to be connected in a flexible and efficient way with streams. At the same time, however, streams support high-level interfaces, each consisting of a set of procedures that clients can call. Even though a client may have several calls to a server in progress simultaneously, the meaning of the calls is the same as if they were made one at a time.

Streams make no statement about what linguistic mechanisms should be used to interact with them. However, in adding streams to a programming language, it is important not to undermine their performance goals. In addition, it is desirable that the use of streams continue to follow the procedure call model. Our main contribution has been the provision of a mechanism that retains the benefits of procedural interaction without sacrificing performance. Reasoning about the correctness of a program that uses promises to make calls on a single stream is no different from reasoning about the same program using traditional RPC. Such reasoning is considerably less complex than reasoning about general message passing.

Most languages for distributed systems provide a procedure-oriented communication mechanism. Examples are Ada [19] and SR [1]. The advantages of this approach are well known: it allows simpler programs to be written that are easier to reason about. However, none of these languages allows the efficiency of streaming. Programs in these languages can be optimized only to reduce the delay of individual calls, not to improve the throughput of groups of calls.

The deficiency of remote calls for high throughput is also well known, and some languages, such as Plits [4] and *MOD [2], have attempted to solve this problem by providing explicit send and receive primitives. Here the sender need wait only until the message

[3] If a wounded process does not terminate quickly enough, we simply crash its guardian. Each Argus guardian is written to survive crashes. More details can be found in [15].

[4] Atomicity cannot prevent partial results from external activities like printing. If a failure prevents an external activity from finishing, there will always be uncertainty as to whether it happened or not. An example of where this would be a problem is when a cash vending machine dispenses money. The best that can be done is to reduce the uncertainty to a very small duration so that the probability of a failure happening in that time is very small.

is produced. Later the outcome of the processing that happens in response to the message is sent to the original caller in a separate message that the caller receives. The send/receive approach can allow programs to achieve high throughput, but it leads to complex and ill-structured programs. The difficulty is that to obtain the efficiency benefits of streaming, it is necessary to have many "calls" in progress at a time, and it is entirely the responsibility of the user code to relate reply messages with the calls that caused them. Promises and streams, however, retain high throughput without imposing this burden.

Note that promises and streams provide almost the full flexibility of send/receive. Send/receive does not require pairing of messages, but as a practical matter pairing is always needed, because ultimately the caller needs to find out what happened. Sometimes, pairing of send/receive takes the form of one reply for many calls; we can accomplish this with sends. Sometimes, however, the reply comes from a third party, a mechanism we do not support.

We also considered the use of promises with forks. Promises for streams have three properties: concurrency of caller and callee, caller control of claiming and putting promises in data structures, and ordering of the processing of a sequence of calls on a stream. Promises for forks have only the first two properties, but are nevertheless very useful. In particular, the ability to propagate exceptions from the forked process to some other process in a convenient manner is extremely useful, and represents a solution to a problem that has been a concern to language designers in this area.

Introduction of streams naturally leads to the problem of how to compose them. A composition allows cascading of information from one stream to the next. Concurrency is a natural way to do stream composition, since we want to start calls on the n^{th} stream in parallel with calls on the $n - 1^{st}$ stream. We investigated what programs that compose streams are like, and what linguistic mechanisms are useful to support this composition.

We conclude that promises are a good way of supporting efficient, asynchronous remote procedure calls in a programming language. The extension to forks allows remote and local concurrency to be provided in a uniform way. Although forks can permit composition of streams, better structured programs result from a mechanism like the **coenter**, which also handles process termination properly.

Acknowledgments

We want to thank Bert Halstead and Bob Scheifler for preliminary discussions that led to the idea of promises, and Dorothy Curtis, Sanjay Ghemawat, Bob Gruber, Paul Johnson, Elliot Kolodner, Juan Loaiza, Sharon Perl, and Bill Weihl for help in developing the mechanism.

References

1. Andrews, G. R. "Synchronizing Resources". *ACM Trans. on Programming Languages and Systems 3*, 4 (October 1981), 405-430.

2. Cook, R. P. "*MOD -- A language for distributed programming". *IEEE Trans. on Software Engineering SE-6* (November 1980).

3. Dijkstra, E. W. "The structure of the 'THE'-multiprogramming system". *Comm. of the ACM 11*, 5 (May 1968), 341-346.

4. Feldman, J. A. "High level programming for distributed computing". *Comm. of the ACM 22*, 6 (June 1979), 353-368.

5. Halstead, R. "Multilisp: A language for concurrent symbolic computation". *ACM Trans. on Programming Languages and Systems 7*, 4 (October 1985).

6. Halstead, R., and Loaiza, J. Exception Handling in Multilisp. International Conference on Parallel Processing, IEEE, August, 1985, pp. 822-829.

7. Herlihy, M. P., and Liskov, B. "A value transmission method for abstract data types". *ACM Trans. on Programming Languages and Systems 4*, 4 (October 1982), 527-551.

8. Hoare, C. A. R. "Monitors: an operating system structuring concept". *Comm. of the ACM 17*, 10 (October 1974), 549-557.

9. Hoare, C. A. R. "Communicating sequential processes". *Comm. of the ACM 21*, 8 (August 1978), 666-677.

10. Liskov, B., Snyder, A., Atkinson, R. R., and Schaffert, J. C. "Abstraction mechanisms in CLU". *Comm. of the ACM 20*, 8 (August 1977), 564-576.

11. Liskov, B., and Snyder, A. "Exception handling in CLU". *IEEE Trans. on Software Engineering SE-5*, 6 (November 1979), 546-558.

12. Liskov, B., and Scheifler, R. W. "Guardians and actions: linguistic support for robust, distributed programs". *ACM Trans. on Programming Languages and Systems 5*, 3 (July 1983), 381-404.

13. Liskov, B., Scheifler, R., Walker, E., and Weihl, W. Orphan Detection. Proceedings of the 17th International Symposium on Fault-Tolerant Computing, IEEE, Pittsburgh, Pa., July, 1987, pp. 2-7. Extended version available as Programming Methodology Group Memo 53, M.I.T. Laboratory for Computer Science, Cambridge, Ma..

14. Liskov, B., Bloom, T., Gifford, D., Scheifler, R., and Weihl, W. Communication in the Mercury System. Programming Methodology Group Memo 59, M.I.T. Laboratory for Computer Science, Cambridge, Ma., 1987. Also in the Proceedings of the 21st Annual Hawaii Conference on System Sciences, January 1988.

15. Liskov, B., et al. *Argus Reference Manual.* Technical Report MIT/LCS/TR-400, M.I.T. Laboratory for Computer Science, Cambridge, Ma., 1987.

16. Liskov, B. "Distributed Programming in Argus". *Comm. of the ACM 31*, 3 (March 1988), 300-312.

17. Mitchell, J. G., Maybury, W., and Sweet, R. Mesa Language Manual Version 5.0. Technical Report CSL-79-3, Xerox Research Center, Palo Alto, Ca., 1979.

18. Rovner, P., Levin, R., and Wick, J. On Extending Modula-2 for Building Large, Integrated Systems. 3, DEC System Research Center, Palo Alto, Ca., January, 1985.

19. U. S. Department of Defense. *Reference manual for the Ada programming language.* 1983. ANSI standard Ada.