MIT/LCS/TR-384

Remote Pipe and Procedures for Efficient Distributed Communication

David K. Gifford and Nathan Glasser

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-83-K-0125.

October 1986, revised July 1987

© 1986, 1987 Massachusetts Institute of Technology

Table of Contents

1. The Channel Model for Distributed Communication 2
2. Relation to Previous Work 3
3. Semantics 6
    3.1. Remote Interfaces are Typed 7
    3.2. Channels are Represented by Local Procedures 9
    3.3. Calls on a Channel by a Single Process are Ordered 11
    3.4. Channel Groups Provide Inter-channel Timing Invariants 12
    3.5. Failures Complicate Channel Semantics 14
4. Pragmatics 16
    4.1. Connections are Used to Detect Crashes 16
    4.2. Channel Failure Detection and Recovery 18
    4.3. Sequence Vectors Implement Channel Timing 19
    4.4. Normal Call Processing 20
    4.5. The Performance of the Model Implementation can be Improved 21
5. Practical Experience and Conclusions 22
    5.1. Performance Bounds and Results 22
    5.2. The Elements of the Model have been Proven Practical 26
    5.3. Advantages of the Channel Model 27
6. Appendix: Explicitly Serviced Pipes 29
7. References 30

List of Tables

Table 5-1: Low Latency Experiments 24
Table 5-2: High Latency Experiments 25

Remote Pipes and Procedures for Efficient Distributed Communication

David K. Gifford and Nathan Glasser

A new communication model for distributed systems is described that combines the advantages of remote procedure call with the efficient transfer of bulk data. Three ideas form the basis of this model. First, remote procedures are first-class values which can be freely exchanged among nodes, thus enabling a greater variety of protocols to be directly implemented in a remote procedure call framework. Second, a new type of abstract object, called a *pipe*, allows bulk data and incremental results to be efficiently transported in a type safe manner. Unlike procedure calls, pipe calls do not return values and do not block a caller. Data sent down a pipe is received by the pipe's sink node in the order they are sent. Third, the relative sequencing of pipes and procedures can be controlled by combining them into *channel groups*. Calls on the members of a channel group are guaranteed to be processed in order. Application experience with this model, which we call the *Channel Model*, is reported. Derived performance bounds and experimental measurements demonstrate k pipe calls can perform $min(1 + \frac{r}{p}, k)$ times faster than k procedure calls, where r is the total round-trip remote communication time and p is the procedure execution time.

Keywords: Bulk Data Transfer, Channel Model, Communication Model, Distributed Communication, Pipe, Remote Procedure, Remote Procedure Call

1. The Channel Model for Distributed Communication

We present a new communication model for distributed systems called the *Channel Model* that combines the advantages of bulk data transport and remote procedure call in a single simple framework.

Remote procedure call is now a widely-accepted standard method for communication in distributed computer systems [White76, Gifford81, Nelson81, Liskov83, Birrell84]. This popularity can be attributed to three advantages provided by remote procedure call. First, remote procedure call employs a widely-accepted, used, and understood abstraction, the procedure call, as the sole mechanism for access to remote services. Second, remote procedure call allows remote interfaces to be specified as a set of named operations with certain type signatures. Such specifications enable remote interfaces to be precisely documented, and distributed programs to be statically checked for type errors. Third, since interfaces can be precisely specified, the communication code for an application can be automatically generated, by either a compiler or a specialized stub generator.

The wider use of remote procedure call systems has led to an understanding of their disadvantages, as well as their advantages. Based on our recent application experience [Gifford85], we have have discovered three major problem areas in standard remote procedure call systems: protocol flexibility, incremental results, and bulk data transport.

1. *Protocol Flexibility* Certain communication protocols are impossible to implement if a remote procedure call system does not allow the exchange of remote procedure values between nodes. For example, imagine that a client node wishes to provide a server node with a procedure for use in certain circumstances, and the server node then wishes to pass this procedure on to another server. Unless remote procedures are first-class objects that can be passed from node to node this protocol cannot be expressed in a remote procedure call framework. A first-class object is a value that can be freely stored in memory and passed as a parameter to both local and remote procedures.

2. *Incremental Results* Consider a server, computing a result on behalf of a client, that wishes to communicate incremental results as they are computed to the client. In present remote procedure call systems the client would ask the server to compute the first incremental result, then the second, and so forth until all of the results have been computed. This approach forces a single computation to be decomposed into a series of distinct remote procedure calls. This decomposition reduces the server's performance since

it is inactive between client procedure calls unless the server creates a sophisticated process structure upon the client's first incremental result request. However, sophisticated process structures are undesirable because they substantially complicate a program.

3. *Bulk Data Transport* Remote procedure call mechanisms are optimized for both low call-return latency and the transmission of limited amounts of data (usually less than 10^3 bytes). These optimizations for the normal case seriously affect the ability of remote procedure call mechanisms to transport large amounts of data efficiently. Since in contemporary systems only one remote procedure call at a given time can be in transit between a single process client and a server, the communication bandwidth between them is limited. For example, if we assume that a program transmits 10^3 bytes per remote procedure call and the network has a 50 millisecond round trip latency, the maximum bandwidth that can be achieved is 20 KBytes/second. Furthermore, to achieve even this performance, the client must combine data values as they are produced into 10^3 byte blocks before a remote procedure call is made. If a remote procedure call was made whenever data were available to be sent, e.g. for each character to be displayed on a remote screen, communication performance could drop to 20 bytes/second.

As a direct result of our experience with these limitations we have developed a new communication model called the *Channel Model.* The Channel Model extends remote procedure call in three directions and address the three disadvantages discussed above. First, we permit remote procedures to be first-class values which can be freely passed between nodes. Second, we introduce a new abstraction called a *pipe* that efficiently transports bulk data and incremental results. Third, we introduce *channel groups* which control the relative sequencing of calls on pipes and procedures. These three ideas work together to create a framework that can be used by a variety of applications.

The remainder of this paper is organized into six sections: Previous Work (Section 2), Semantics (Section 3), Pragmatics (Section 4), and Practical Experience and Conclusions (Section 5).

2. Relation to Previous Work

The primary contribution of the present work is the integration of both low-latency and high-throughput communication into a single simple framework for distributed communication via a new technique for sequencing calls. Special purpose protocols have been developed previously that combine low-latency and high-throughput

communication, but these protocols have been highly specific to a particular application. For example:

- The R* System [Lindsay84] uses a bulk data transfer protocol that is built on a request-response communication model. In R* bulk data transport is explicitly programmed at the application level by packing result data into answer buffers. The R* communication model includes the idea of blocking and non-blocking sends.

- The CMU Distributed Log Server [Daniels86] extends remote procedure call with special purpose asynchronous messages. The asynchronous messages transmit streams of data in order to achieve adequate performance.

- The RPC2 System [Satya86] extends remote procedure call by allowing application specific protocols to be included as *side-effects* of remote procedure calls. For example, a file transfer side-effect invokes a file transfer protocol. A special purpose file transfer protocol is used by RPC2 because remote procedure call was not efficient enough for bulk data transport. RPC2 has addressed the limitations of remote procedure call by allowing each application to develop its own special purpose network optimizations.

- The X Window System [Scheifler86] uses a special purpose graphics protocol that combines synchronous calls with asynchronous operations. Asynchronous graphics calls were introduced by the X Window System in order to achieve high throughput operation for display operations.

These special purpose systems demonstrate a common need for a communication model that combines low-latency remote procedure call with a general purpose high-throughput data transport mechanism. The Channel Model is designed to address precisely this need.

The Channel Model can properly be viewed as a new type of sequencing algorithm that permits previous work in remote procedure call protocols to be combined with previous work in asynchronous calls. This previous work is related to the Channel Model in the following respects:

- *First-Class Remote Procedures* The idea of transmitting remote procedure values is discussed by Nelson [Nelson81], and is also present in ARGUS [Liskov83] as handlers. However, neither of these proposals allow remote procedures to be created in nested scopes, limiting the generality of remote procedures.

- *Asynchronous calls* The notion of a pipe is similar to Nelson's immediate

return procedures [Nelson81], and the unidirectional messages of Matchmaker [Jones85]. Nelson, however, rejected immediate return procedures for his communication model because they were inconsistent with procedure call semantics. Our solution to the consistency problem is the creation of a new type of abstract object with well defined properties. In both Nelson's work and Matchmaker, unidirectional message sends are not appropriate for bulk data transport because they are not buffered to improve throughput. Mach [Young87], the operating system underlying Matchmaker, limits efficient bulk transmission of data to data structures that are passed by reference. Thus, Mach cannot transport incremental results efficiently. UNIX pipes [Ritchie74] transport untyped byte-streams, while in our model pipes transport typed values. Because UNIX pipes are not typed stub modules [Nelson81] can not be generated automatically for UNIX pipes.

- *Explicit Process Creation* Conventional `Fork` and `Join` constructs can be used to program asynchronous calls within a conventional remote procedure call package [Birrell84]. This approach has three disadvantages when compared with the Channel Model: it has a higher overhead per asynchronous call; it does not provide any sequencing semantics for a series of forked calls; and calls are not buffered to improve bulk data throughput.

Spector's Remote Reference/Remote Operation Model [Spector82] provides a taxonomy of remote procedure call types, including synchronous and asynchronous calls. Spector points out that asynchronous calls may require flow control. In our framework, flow control ensures that asynchronous calls on pipes are optimized for high throughput. An implementation of the Channel Model can implement flow control on top of raw datagrams, or it can use a transport protocol that includes flow control such as VMTP [Cheriton87].

The ability to sequence procedure and pipe calls with channel groups is unique to the Channel Model. Channel groups are useful because they permit the essential sequencing semantics of certain calls to be preserved, while allowing other calls to run in parallel. Sequencing constraints can be statically specified in interfaces and then enforced using the Channel Model or constraints can be specified dynamically at run-time.

The Channel Model does not directly include a facility for sending a single call to multiple sinks (sometimes called *multicasting*). Such a facility is provided by the MultiRPC facility of RPC2 [Satya86], and the parallel procedure call facility of PARPC [Martin87]. In order to send requests to multiple sinks in parallel within the Channel Model pipes can be used or procedure calls can be forked.

3. Semantics

We discuss in this section

- typed remote interfaces,
- the use and creation of remote pipes and procedures,
- channel call ordering,
- how channel groups provide inter-channel synchronization,
- and failure semantics.

For the purpose of our discussion, we will define a *node* to be a virtual computer with a private address space. A physical computer can implement one or more nodes; the precise size and scope of a node depends on application requirements. We will assume that all of the nodes in a system are interconnected by a network.

Both remote procedures and pipes provide a communication path to a remote node, and thus we call them *channels*. A channel represents the ability to perform a specific remote operation at a remote node. Channels are first-class values. In particular, channels can be passed freely to remote procedures or pipes as parameters, or returned as the result of a remote procedure. Connections are implicitly established as necessary when channels are used as described in Section 4.

A node that exports a set of channels is called a *service* node, while a node that imports a set of channels is called a *client* node. A given node can be a service and a client simultaneously, with respect to different sets of channels.

It is not always the case that a client calls a service; when a service and client are working together on a task sometimes it is most natural for a service to call a client. For example, a client might provide a service with a pipe which allows the service to send a large file to the client. Consequently, we introduce terminology to describe the dynamic relationship between two nodes. We will call the node that processes calls made on a channel the channel's *sink* node, and we call the node that makes calls on a channel the *source* node. Note that a given node can be both a source and a sink.

Remote procedures and pipes are defined as follows:

- *Procedures* A *remote procedure* value is a capability to call a procedure on a remote node. A *remote procedure call* blocks the caller until the remote

procedure has finished execution and a response has been received from the sink node. Since a remote procedure call blocks a client, remote procedure calls are optimized for *minimum latency*. In the event a remote procedure call fails, a distinguished error value is returned as the value of the call. The precise types of failures that can result are described below.

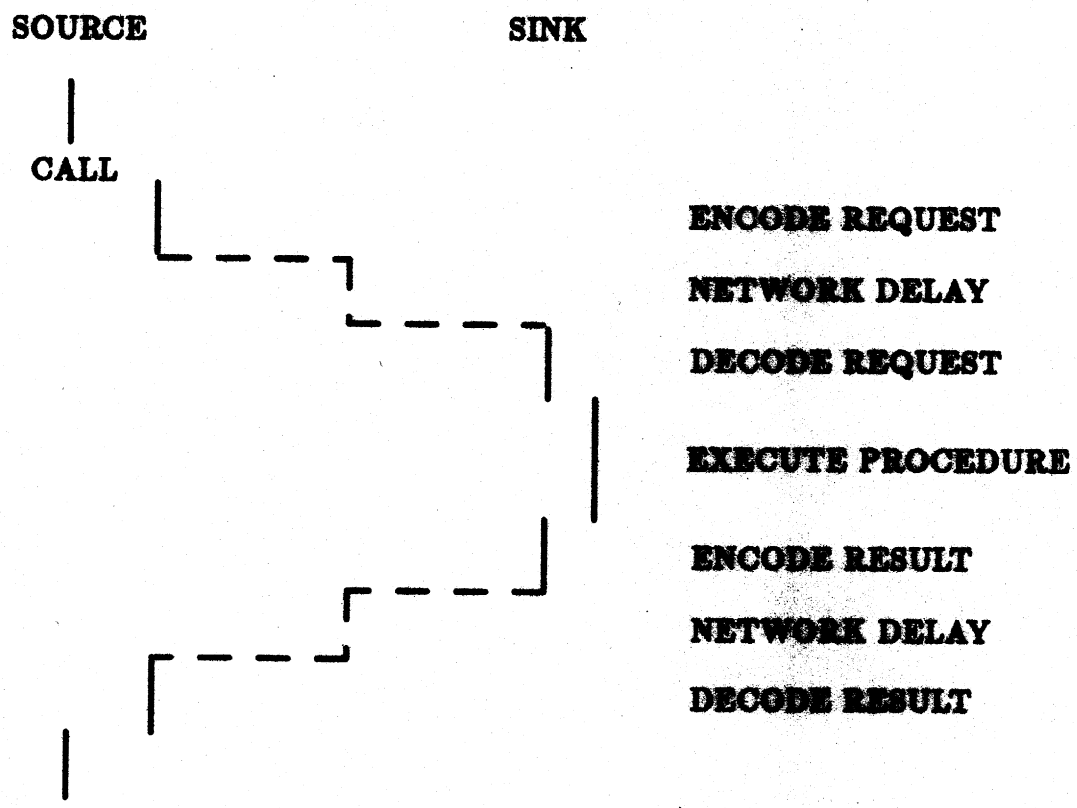
- *Pipes* A *pipe* value is a capability to transfer bulk data to another node. A pipe is used as is a remote procedure. However, unlike a remote procedure call, a *pipe call* does not block the caller and does not return a result. Since a pipe is designed for bulk data transport, it is optimized for *maximum throughput*. A pipe call does not block its caller, therefore a pipe call implicitly initiates concurrent activity at the pipe's sink node. The caller continues execution as soon as the call to the sink is queued for transmission. The pipe's sink node receives the data values sent down a pipe by a given process in the order the pipe calls are made. By "receive" we mean that the sink accepts the data in order, and performs some computation on the data. This computation could process the data to completion or simply schedule the data for later processing. The failure of a pipe call to complete can be detected as described below.

Figures 1 and 2 respectively show the processing of remote procedure and pipe calls. Figure 1 shows that a remote procedure call blocks the source process until the result of the procedure call is received and decoded by the source. As suggested by the figure, we assume that typed values can be converted to byte strings and back again via *encode* and *decode* operations [Herlihy82]. Figure 2 shows that during a pipe call the calling process continues while the pipe call executes asynchronously at the sink.

We designed the Channel Model to adapt readily to a wide range of programming languages. Specific versions of encode and decode must be designed to work with each programming language to properly handle the language's type space and exception handling discipline. For pedagogical purposes, we will write our examples in Modula-II [Wirth85]. Modula-II does not provide full closures, and thus the dynamic creation of procedures is limited in Modula-II. Section 5 discusses a C implementation of remote pipes and procedures.

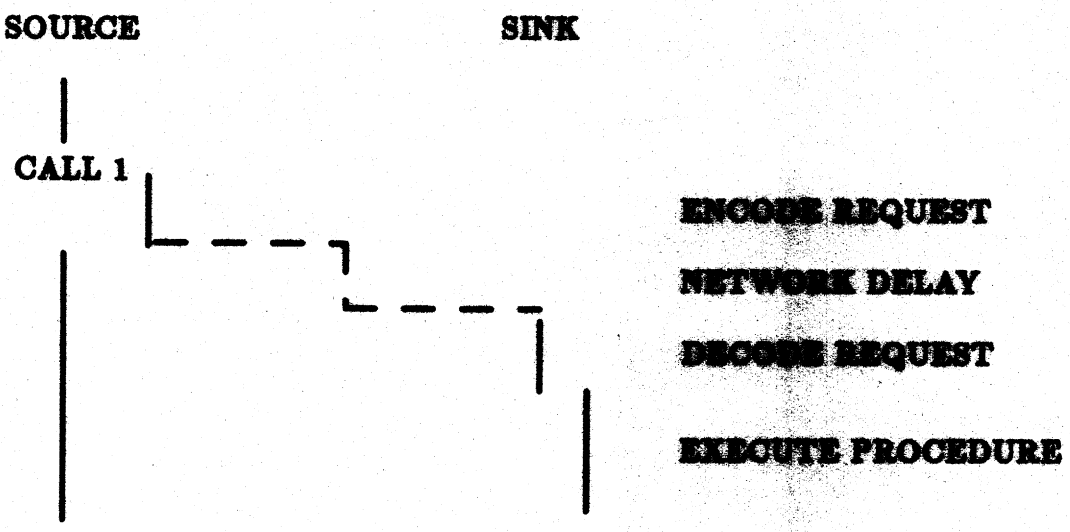
3.1. Remote Interfaces are Typed

A *remote interface* is a set of type definitions used to describe a collection of channels that is exported as a unit by a service. We will use a collection of Modula-II type definitions to describe a remote interface, with the language extended to include types for remote pipes and procedures.



A REMOTE PROCEDURE CALL

FIGURE 1



A PIPE CALL

FIGURE 2

We introduce the types PIPE and PROC in our interface language to describe pipes and remote procedures, respectively. A channel's type further describes the channel's input values and the channel's result values. Note that only procedures have result values. For example a pipe value might have type

```
PIPE PutChar(CHAR);
```

indicating that the pipe is a character pipe, while a remote procedure might have type

```
PROCEDURE GetChar(): CHAR;
```

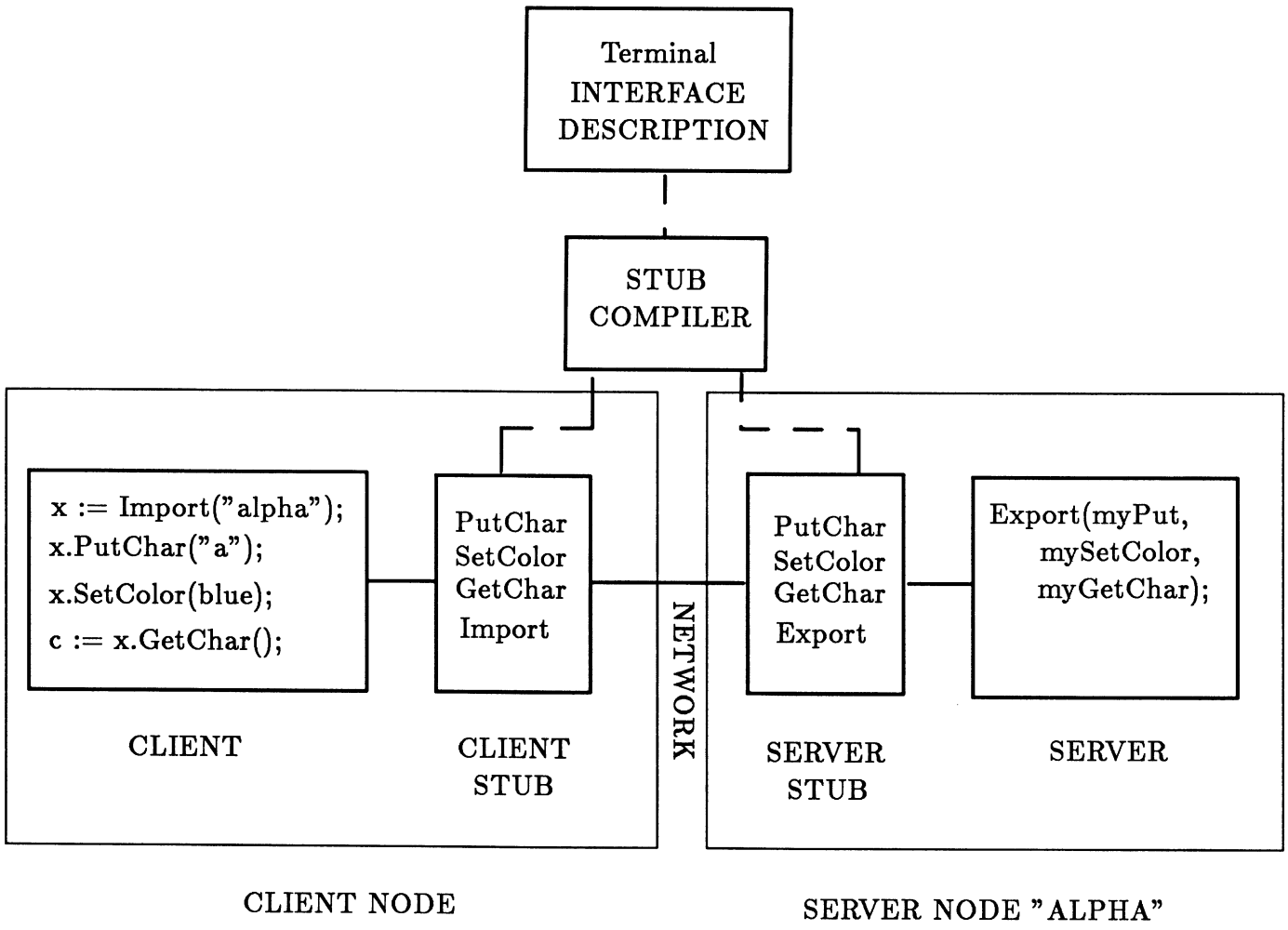
indicating that the remote procedure does not take an argument and it returns a character.

The following is an example of a remote interface that defines three channels. Note that two of the channels are pipes, while one is a remote procedure.

```
REMOTE INTERFACE Terminal;
   TYPE Color = (red, blue, green);
   PIPE PutChar(c: CHAR);
     (* Displays a character on the terminal *)
   PIPE SetColor(c: Color);
     (* Sets the display color for subsequent characters *)
   PROCEDURE GetChar(): CHAR;
     (* Returns the next character.  Blocks if a character is
        not available *)
END Terminal.
```

Interfaces are used to document a service, as well as to allow an implementation of the Channel Model to encode and decode messages of appropriate types. For example, a remote interface, by virtue of its type declarations, contains enough information to permit a stub generator [Nelson81] to automatically generate code to implement the details of a communications protocol. Once an interface is specified, an application programmer can deal with pipes and procedures and not be concerned with the way information is encoded and transmitted over a wire.

Figure 3 shows how the interface above might be used to define a communications protocol. An automatic stub generator transforms the interface into two stubs, a *server* stub and a *client* stub. As shown in Figure 3, the server stub includes the operation `Export`. The server calls `Export` to register local procedures that implement the interface described by `Terminal` for use by client programs. `Export` converts these



REMOTE INTEFACES AND BINDING

FIGURE 3

local procedures into channel values for use by remote clients. In order to use these operations, a client program calls the `Import` operation in the client stub to obtain a specific instance of `Terminal`.

The general problem of providing clients with instances of service interfaces is known as *binding*, and we will assume that an implementation of the Channel Model will use a conventional binding technique. For example, the `Import` and `Export` scheme shown in Figure 3 is used by the Cedar Remote Procedure Call System [Birrell84]. Another technique is to use a distributed database system to record offered services. For example, the Courier remote procedure call system [Xerox81] in the Xerox Star System uses the Clearinghouse distributed database system [White82] for client-server binding.

Binding allows a node to bootstrap itself into a distributed environment. Note that binding establishes initial contact with a set of servers, and this set of initial servers can easily provide a client with a rich set of channels which can be used to contact other nodes that were not provided via the binding mechanism.

3.2. Channels are Represented by Local Procedures

Remote procedures and pipes are used in the same manner as local procedure values. In order to allow the introduction of procedures and pipes without changing Modula-II, we will represent channels with local procedures. Thus, a local procedure must be converted to a channel value for export, and a channel value must be converted to a local procedure on import. If a given channel is first imported as a local procedure, and then that local procedure is exported as a channel, the channel value exported must be the same as the original channel value imported to ensure that extra overhead is not introduced.

The channel type declarations used in programs are automatically produced by the stub generation program from the user's original remote interface definition. For example, the `Terminal` remote interface would result in the following client stub definitions module:

```
DEFINITION MODULE TerminalClient;
 TYPE Color = (red, blue, green);
 Terminal = RECORD
              PutChar: PROCEDURE (CHAR),
              SetColor: PROCEDURE (Color),
              GetChar: PROCEDURE (): CHAR
            END;
 PROCEDURE Import(server: ARRAY OF CHAR): Terminal;
END TerminalClient.
```

As this definition module suggests, remote procedures are used just like local procedures:

```
new-char := term.GetChar();
```

Remote procedures are created by providing a local procedure to **Export** that will receive calls made on the channel.

A pipe value is used in precisely the same way as a procedure value is used except that pipes do not return result values. The following expressions send the values "o" and "k" down **PutChar**:

```
term.PutChar("o");
term.PutChar("k");
```

The values "o" and "k" are guaranteed to be received by the sink of **PutChar** in order (because the two pipe calls shown above are performed by the same process). No reception order is defined for pipe calls made by separate processes.

A pipe is created through the provision of a local procedure called a *pipe sink procedure*, that will process data received over the pipe. As data arrive through a pipe its corresponding sink procedure is applied to each datum in the order the data are received. A pipe's sink procedure must return before it will be applied to the next datum sent down the pipe from the same source process. When a node desires to export a pipe it provides a pipe sink procedure, and this sink procedure is converted into a pipe value by the sink's stub.

When a channel is created by a sink for export it normally will exist until the sink crashes or until the channel is explicitly deleted. Channels that are destroyed by sink crashes are known as *dynamic channels*. Channels which can survive sink crashes are called *stable channels*. Stable channels are useful for stable services that are registered

with a clearinghouse. The state of a stable channel and its associated procedure must be recorded in stable storage to permit recovery of the channel upon sink restart. The details of how stable channels are created will depend on the host language environment.

We now have enough mechanism to introduce a simple file transfer protocol based on channels. Recall that channels can take channels as arguments and return channels as values. Therefore, a simplified file transfer interface could be represented as the two procedures `SendFile` and `GetFile`:

```
REMOTE INTERFACE Ftp;
  PROCEDURE SendFile(name: ARRAY OF CHAR): PIPE (CHAR);
    (* Returns a pipe to receive the contents of name *)
  PROCEDURE GetFile(name: ARRAY OF CHAR, put-here: PIPE(CHAR));
    (* The contents of name are sent down the pipe put-here *)
END Ftp.
```

`GetFile` is passed the name of the file to retrieve and a pipe called `put-here` to receive the file. The symmetric operation `SendFile` returns a pipe to receive the new contents of the file called `name`. The file is then transmitted down the returned pipe, with the end of a file marked by a special character. For example:

```
ftp := FTPClient.Import("zermatt");
ellen := ftp.SendFile("/usr/cassatt/ellen.imp");
ellen("h");
ellen("i");
ellen(EOF);
```

3.3. Calls on a Channel by a Single Process are Ordered

Our communication model guarantees that if a process makes two separate calls on the same channel, then the calls will be processed at the sink in the order they were made by the process. "Processed" means that the second call is not executed at the sink until the procedure invoked by the first call has returned.

The ordering of channel calls not specified by the above invariant is undefined. Thus, a single channel can be invoked simultaneously by different source processes and the corresponding calls will run in parallel at the sink. We assume that each procedure employs monitors [Redell80], or a similar mechanism, to ensure its proper operation in the presence of concurrent invocations.

Since pipe calls do not return values and are processed asynchronously, a `Synchronize` operation is provided. When a source process applies a `Synchronize` operation to a pipe, the pipe's sink is forced to process all outstanding data sent down the channel from the source process, after which the synchronize operation returns. If the pipe has broken for some reason (e.g. the sink node has crashed) then synchronize will return a distinguished error value and reset the pipe so that it can be used again. Errors are described in detail in Section 2.5.

As shown here, once application of `Synchronize` is to ensure that output is complete:

```
term.PutChar("n")
term.PutChar("o")
code := Synchronize(term.PutChar);
```

3.4. Channel Groups Provide Inter-channel Timing Invariants

In our present model the ordering of calls on separate channels is undefined. However at times it is desirable to provide a timing invariant across channels. Consider our example `Terminal` defined above, and the characters that would be displayed by the following statements:

```
term.PutChar("a");
term.SetColor(blue);
term.PutChar("b");
```

Since `SetColor` and `PutChar` are separate pipes, there is no sequencing invariant defined between them. Thus the character `b` may or may not be displayed in blue, depending on the order in which the pipe calls `SetColor` and `PutChar` are processed by their sink node. In this example, we want to specify that calls on `SetColor` and `PutCharacter` must be performed in the order in which the calls were made.

When timing invariants must be preserved between a set of channels, the channels can be collected into a *channel group*. A channel group is a collection of pipes and procedures that share the same sink node and that observe a sequential ordering constraint with respect to a source process. This ordering constraint guarantees that calls made by a single process on the members of a channel group are processed in the order they were made.

In order to construct channel group values we introduce `Group`. `Group` takes a record of channel values as input and returns a new record of channels that are in the same

group. Group relationships that existed in the input channels will be present in the output channels. If all of the channels that are passed to **Group** do not have the same sink node then **Group** will return a distinguished error value. Note that the channels returned by **Group** are copies, and thus the original input record to **Group** will still refer to a set of ungrouped channels after **Group** returns. The channel copies returned by **Group** will each include a new unique *sequence stamp* that has been added by **Group**. The sequence stamp added to each channel identifies each channel's membership in the newly created group. In addition to the sequence stamps obtained by group membership, each channel is assigned a unique sequence stamp upon creation.

We can now solve our earlier problem of synchronizing **SetColor** and **PutChar** calls. For example, the statements

```
seqTerm := Group(term);
seqTerm.SetColor(red);
seqTerm.PutChar("a");
seqTerm.PutChar("b");
```

can be used to create a new sequenced terminal value **seqTerm** and to display the characters "ab" in red.

A channel can be a member of more than one group at once, which allows a wide variety of sequencing semantics to be directly expressed using channel groups. For example, consider the following remote interface:

```
REMOTE INTERFACE ColorDisplay;
   TYPE Color = (red, blue, green);
   PIPE SetFont(font: ARRAY OF CHAR);
      (* Sets the font *)
   PIPE SetColor(c: Color);
      (* Sets the color *)
   PIPE PutChar(c: CHAR);
      (* Displays a character on the screen *)
END ColorDisplay.
```

In order to ensure that characters appear in the proper color and font a **ColorDisplay** server can create two channel groups: {**SetFont**, **PutChar**} and {**SetColor**, **PutChar**}. Note that **SetFont** and **SetColor** do not need to be sequenced with respect to one another. Utilizing two channel groups instead of a single channel group to sequence **ColorDisplay** allows **SetFont** and **SetColor** to execute in parallel.

A programmer can specify sequencing constraints in interfaces instead of calling **Group**

directly. The stub compiler will translate these specifications into server stub calls on `Group`. For example, the desired groups of `ColorDisplay` could be specified as:

```
REMOTE INTERFACE SeqColorDisplay;
   TYPE Color = (red, blue, green);
   PIPE SetFont(font: ARRAY OF CHAR);
      (* Sets the font *)
   PIPE SetColor(c: Color);
      (* Sets the color *)
   PIPE PutChar(c: CHAR);
      (* Displays a character on the screen *)
   SEQUENCE SetFont, PutChar;
   SEQUENCE SetColor, PutChar;
END ColorDisplay.
```

The channel timing invariant provided by the Channel Model can now be succinctly stated:

Channel Timing Invariant If a process makes two separate calls on channels that (1) are at the same sink node, and (2) have a sequence stamp in common, then the calls will be processed at the sink in the order they were made by the process. "Processed" means that the second call is not executed until the procedure invoked by the first call has returned.

The channel timing invariant implies the invariant for calls on a single channel (because a channel will always be at the same sink node as itself and will have a sequence stamp in common with itself). The channel timing invariant embodies all of the ordering semantics provided by the Channel Model. The ordering of channel calls not covered explicitly by the channel timing invariant is undefined.

3.5. Failures Complicate Channel Semantics

Our communication model guarantees that a channel call will be performed precisely once in the absence of failures. In the presence of failures the semantics of remote operations are more complicated. Many kinds of distributed system failures (e.g. node crashes, network partitions) can cause a source node to wait for a reply which will never arrive. In such cases it is impossible to tell if the corresponding remote operation was ever attempted or completed.

In the presence of failures *at-most-once* semantics can be provided for remote calls.

At-most-once semantics guarantees that a remote operation either will be performed exactly once, or will have been performed at most once if a failure has occurred. A failed procedure call returns a distinguished *crash* value as its result. A failed pipe call causes a distinguished *crash* value to be returned as the result of the next procedure call on the same group. When a failure occurs it is impossible to determine whether a remote operation was never started, completed, or only partially completed. Thus *at-most-once* semantics present a serious challenge to the application programmer who wishes to cope gracefully with failure.

One technique used in several practical systems accepts the limitations of at-most-once semantics and insists that procedure calls that mutate stable storage be idempotent. With this restriction a remote procedure call that returns *crash* can be repeated safely until the call completes without failing.

Exactly-once semantics is an alternative to at-most-once semantics. Exactly-once semantics guarantees that a remote operation will be performed exactly once or not at all. Exactly-once semantics protects the actions of a remote operation with a transaction. If a remote operation returns *crash*, the operations corresponding transaction is aborted. The transaction abort will undo the effects of the failed remote operation, and the failed operation will appear to never have happened. The failed operation can then be retried (if desired) with a new transaction.

Exactly-once semantics can be achieved through the combination of communication with transactions in one of two ways. One approach, as suggested by ARGUS's design [Liskov83], is to integrate transactions into the communication model such that each remote operation has an implicit associated transaction. A second approach is to keep the communication model and transactions separate by explicitly specifying transactions where they are required [Brown85].

In addition to success and crash, a third result can be optionally returned from a remote call. If desired, a ping message can proceed a call to ensure that the remote node is available. If the node does not reply to the ping message within a certain amount of time then *unavailable* can be returned as the result of the call. In this case the remote call was not attempted, and no compensation needs to be performed.

In summary, the channel errors that can occur are as follows follows:

1. *Crash* Communication with the remote node was lost during a call, and it is unknown if the operation was performed once, performed partially, or not performed at all.

2. *Unavailable* The sink does not respond to ping messages and thus the requested operation was not attempted.

3. *Destroyed* The channel that was called has been destroyed by its sink node. The operation was not attempted.

Channel errors are always reported to the source node. If a procedure call returns in an error, the error is returned as a distinguished value of the call. If a pipe call results in an error, the error is returned upon the next procedure call to a channel in the same group, or upon the next synchronize operation on a pipe in the group, whichever comes first. When a pipe call results in an error, any subsequent calls on pipes in the same group are discarded until the error is reported. Any calls made before the call that resulted in an error will have been performed exactly once.

4. Pragmatics

We discuss in this section five pragmatic aspects of the Channel Model:

- how connections are used to detect node crashes,
- failure recovery,
- how sequence numbers can be used to implement the channel timing invariant,
- normal call processing,
- and performance elaborations.

For the purpose of this section, we assume that the message system may lose, reorder, and duplicate messages. However, we also assume that messages that are delivered are delivered without error. This ideal can be achieved, with any desired level of reliability, by using larger and larger error detecting codes and discarding messages with detected errors.

4.1. Connections are Used to Detect Crashes

Our crash detection and recovery algorithm is based upon *connections*. Connections provide a simple mechanism for detecting node crashes. A connection is a unique identifier that is shared between a source process and a sink which identifies the incarnation of the source and the sink. In order to implement crash detection, a node

discards its connection state when it crashes. Thus, the connection state of a node can be stored in volatile memory.

Before a source process makes its first call to a remote sink, it must establish a connection with a two packet interchange. The source node first sends the sink the name of the process making the call, the source node identifier, and a proposed connection identifier (one packet). Then the sink acknowledges receipt of the connection identifier and other information (one packet). Each source process keeps track of its outgoing remote connections in a table indexed by remote node identifier. Each sink node keeps track of its incoming remote connections in a table indexed by connection identifier.

Connection state can be garbage collected by both sources and sinks. A source can discard its connection state with a sink when no calls are in progress with the sink by simply forgetting the corresponding connection identifier. The source will have to reestablish a connection before making its next call to the sink. A sink can discard source connection identifiers if no calls are in progress with the corresponding source process. As outlined below, the next time the source makes a call to the sink, the source will discover that the connection has been garbage collected and then create a new connection.

If a sink receives a call message with an unknown connection identifier, the sink sends back a distinguished *unknown connection* message to the source. The sink includes the unique identifier of the call message, the source process identifier, and the unknown connection identifier in the message.

A source will find itself in one of three states upon receipt of an unknown connection message:

- The first case is that the call message received by the sink was sent before the last source crash. In this case, the source process identifier is unknown to the source, and the unknown connection message is ignored.

- The second case is that the unknown connection message is in response to the first transmission of a call message and at the time the call was made the source process had no other outstanding calls to the sink. In this case a new connection is established with the sink and the call is retransmitted with the new connection identifier. Subsequent calls will use the new connection identifier.

- If neither the first or the second case apply, then the source must assume

that the sink has crashed and recovered since the last successful call. Once the *crashed* failure is returned as the result of a procedure call, a new connection with the sink is established which will enable future calls to be processed.

4.2. Channel Failure Detection and Recovery

We guarantee that as soon as a failure occurs on a channel, no further operations on channels in the same group will be performed until the calling process is notified of the error. Here we examine how this guarantee can be implemented. There are two types of failures that we will consider in our discussion: the failure that results when a destroyed channel is called (a *destroyed* failure), and the failure that results when a sink crash occurs during the processing of a call (a *crash* failure).

Procedure call failures are directly reported to the call site by a distinguished return value. Thus, a failure during a procedure call will not cause future operations on the same group to be ignored because the failure is immediately reported.

Pipe call failures cannot be immediately reported to the call site because the calling process does not block and wait for a return value. Therefore, if a pipe call fails, all subsequent pipe calls to channels in the same group will be ignored until a procedure in the same group as the failed pipe call is called. The pipe failure will be immediately returned as a distinguished return value from the procedure call, and then subsequent pipe and procedure calls on the group can be processed. Note that synchronize is a special form of procedure call, and thus synchronize can be used to poll for pipe failures.

In the case of a pipe *destroyed* failure, the sink must ignore future pipe calls on channels in the same group until a procedure call on the same group is made. The sink can of course advise the source of the pipe error, but the source will not be able to report the error to the calling process until it makes a call on a procedure in the same group as the failed pipe.

In the case of a pipe *crash* failure the source must ignore future pipe calls on channels in the same group until a procedure call on the same group is made. If the pipe calls were not ignored, then it would be possible for the sink to recover and process subsequent pipe calls before the pipe failure was reported.

4.3. Sequence Vectors Implement Channel Timing

A mechanism that utilizes vectors of sequence numbers can be used to implement the sequencing semantics for groups. Recall that in the Channel Model a client can dynamically create groups in order to force the sequential processing of calls made on independent channels. A channel value starts with a single sequencing stamp that is unique to the channel, and copies of the channel value can be freely made that include additional sequencing stamps to denote group memberships. Two channel calls will be processed in order if, and only if, the channel values share a sequencing stamp. We call this sequencing property the channel timing invariant.

One way to implement the channel timing invariant is to generate sequence numbers for each sequence stamp on a per-process basis. Using this method, when a call is made on a channel a new sequence number is obtained for each of the channel value's sequence stamps. This set of sequence stamps and sequence numbers is sent in the call message to the sink, along with the process identifier of the calling process. In order to guarantee the channel timing invariant, the sink will only process a call when all of the call's sequence numbers are one greater than the sequence numbers for already processed calls from the process identified in the call message. By "one greater" we mean that for each sequence stamp in the call message, the sequence number associated with the stamp is one greater than the sink's copy of the sequence number for the corresponding stamp. For a given stamp, if the sequence number is 1 and the sink has no previous record of this stamp, the sink initializes its sequence number for the stamp to be 0. At the end of a call, the sink increments the sequence numbers of the stamps sent with the call.

For example, consider the following sequence of channel calls made by a single process. Each call is shown with the list of sequence stamp and sequence number pairs included in the call message:

```
Put("a");              <[1, #1]>
PutSeq("b");           <[1, #2], [2, #1]>
ColorSeq(blue);        <[3, #1], [2, #2]>
Put("c");              <[1, #3]>
PutSeq("d");           <[1, #4], [2, #3]>
```

From the stamps sent in each call messages shown above, we can determine the sequencing semantics of the channel calls. `ColorSeq` and `Put` do not share a sequence stamp, therefore their processing is unordered. `PutSeq` refers to the same underlying

channel as `Put` (identified by sequence stamp `1`), and thus calls on these two channels will be ordered. In addition, `PutSeq` has been extended with the sequence stamp `2` which is used to group `PutSeq` and `ColorSeq` to guarantee that calls on these two channels will be ordered.

In sum, to implement the normal sequencing of calls, the following state must be maintained by a source and a sink:

- The source must keep for each outgoing connection a sequence stamp to sequence number table. When a local process makes a call, appropriate entries in the outgoing sequence table are incremented, and the new sequence numbers are sent with the call to the sink. The first time a sequence stamp is used by a local process, an entry is placed in the process' table with sequence number 1.

- The sink must keep a sequence stamp to sequence number table for each incoming connection. Each request received is checked against the corresponding connection's table, and if the sequence stamps do not match, the request is rejected. After a request is processed, appropriate entries in the connection's table are incremented. When a sink establishes a new connection it creates a fresh table with no sequence stamp entries. If a sequence stamp is not in a table, an entry with a sequence number of 1 will be automatically generated.

It is possible for either a source or a sink node to discard sequence number tables by garbage collecting the corresponding connection. The rules for garbage collecting connections were outlined in the last section.

4.4. Normal Call Processing

Here, we recap the information that is sent in every call message. A call message includes: (1) the sink node address (for the communication system); (2) the connection identifier (to identify the source node and calling process); (3) a unique call identifier, that is different for each retransmission of the call (to determine if a new connection can be opened if the call fails); (4) a sequence vector (to sequence the call), (5) the identifier of the channel being called, and (6) a byte string that represents the data for the channel.

A call message is retransmitted until a corresponding return message is received. A return message includes: (1) the source node address (for the communication system), (2) the connection identifier (to identify the calling process); (3) the unique call

identifier (to identify the call); and (4) a byte string that represents the result data.

4.5. The Performance of the Model Implementation can be Improved

The model implementation we have described is intended only to be suggestive; a practical implementation of the Channel Model would require performance optimizations. Important optimizations include:

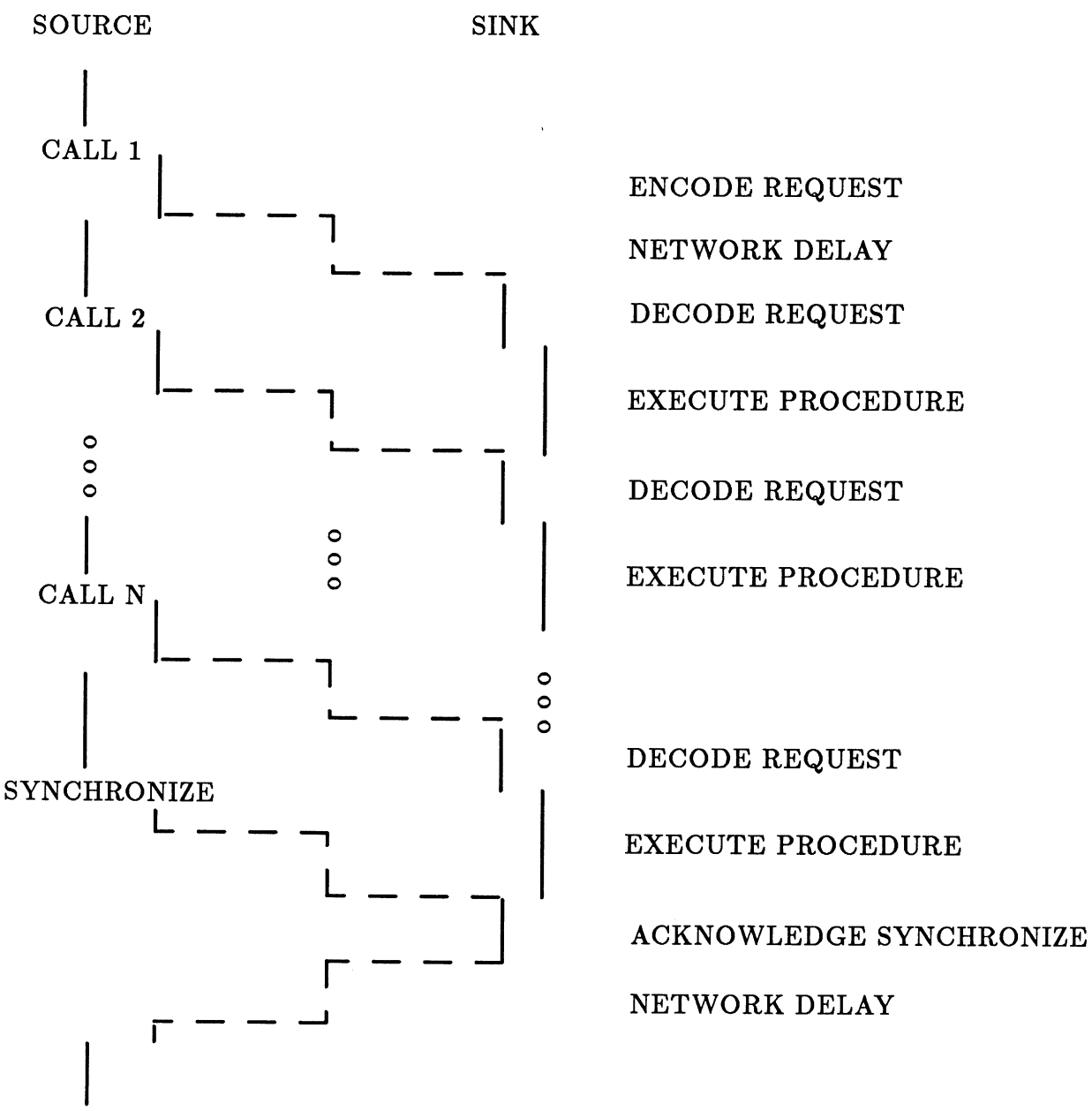
- *Combine pipe calls* Multiple pipe calls destined for the same sink node can be buffered at a source and transmitted as a single message to reduce message handling overhead and improve network throughput. The amount of time a pipe call is buffered before it is sent presents a tradeoff between low pipe latency and efficient bulk communication. A moving window flow control algorithm can be employed [Postel79] to manage the transfer of buffered pipe calls between a source and a sink.

- *Combine pipe calls with procedure calls* A procedure call message will always be transmitted immediately. Therefore, any buffered pipe calls to the same sink should be prepended to the procedure call message whenever possible.

- *Combine pipe returns* Since returns from pipe calls are only used to acknowledge the completion of processing, multiple pipe call returns can be combined into a single message that acknowledges the processing of a set of calls.

- *Preallocate Processes* Processes can be preallocate into a process pool at node startup so that performance of a FORK operation for each incoming remote call is not required. Eliminating FORK overhead is especially important for a collection of pipe calls that arrive in a single message, because the overhead per pipe call is limited to approximately the cost of a procedure call, as opposed to a process creation. A process allocated from a pool would return itself to the pool when the process had finished processing its assigned call message.

- *Explicitly Acknowledge Messages* At times both call and return messages should be explicitly acknowledged in order to improve performance. A call message should be explicitly acknowledged by a sink when the sink has been processing a call for a predetermined interval without producing a result. This acknowledgment informs the source that the call has been successfully received, and the source does not need to retransmit the call message. A procedure return message from a sink should be explicitly acknowledged by a source when the same source process does not make a subsequent procedure call to the sink within a predetermined interval. This informs the



A SERIES OF PIPE CALLS FOLLOWED BY
A SYNCHRONIZE

FIGURE 4

sink that the return message has been received by the source, and that the sink can discard the result contained in the return message.

- *Factor Packages and Groups* In order to save space, information that is common to all of the channels in a package or group value need only be represented once.

5. Practical Experience and Conclusions

We conclude with

- analytical performance bounds and empirical performance results,
- experience with an application of the Channel Model that has proven certain of its elements practical,
- and discussion about general application of the model.

5.1. Performance Bounds and Results

We show here that k pipe calls can be at most k times as fast as k identical procedure calls, and we compare this performance bound to empirical data gathered from an implementation of the Channel Model. Experimental data confirms the analytical model as long as the source does not generate data faster than they can be transmitted to the sink. Performance is degraded from analytical predictions when the source is generating data faster than they can be transmitted to the sink.

First we derive elementary performance bounds on the ratio of time spent processing k procedure calls vs. k pipe calls followed by a `synchronize`. The time spent processing a channel call can be broken down as follows:

- *Procedures* Referring back to Figure 1, if we let r be the time required for remote communication (all encodes and decodes plus the round-trip network latency), and p be the time required to execute the procedure at the sink, then k procedure calls will take $kp + kr$ units of time.
- *Pipes* Figure 4 shows a series of k pipe calls followed by a `synchronize`. The total time required for this set of operations is $kp + r$. This assumes that the time required for the sink encode is less than the time required to execute the procedure and the sink decode. Our analysis is also valid with the opposite assumption, except that costs must be reallocated to p and r.

With these estimates of the cost of pipe and procedure calls we can estimate the time

ratio between k procedure calls and k pipe calls followed by a `synchronize` as:

$$A(k,p,r) = \frac{kp + kr}{kp + r}$$

This equation for $A(k,p,r)$ shows that for fixed r, as the procedure execution time decreases, the performance ratio between pipes and procedures is bounded by

$$\lim_{p \to 0} \quad A(k,p,r) = k$$

Furthermore, for fixed p and r, the performance ratio is bounded by

$$\lim_{k \to \infty} \quad A(k,p,r) = 1 + \frac{r}{p}$$

Combining these two independent bounds yields the single bound

$$A(k,p,r) \leq \min\left(1 + \frac{r}{p},\ k\right)$$

This performance bound cannot be realized when the source generates data faster than they can be transmitted to the sink. This will occur when data are generated at a rate that is greater than the bandwidth of the channel to the sink or when insufficient buffering is provided at the source. In this case additional queuing delays will result that add to the network delay time, and the performance ratio of pipes to procedures will be reduced.

A series of experiments were run on an implementation of the Channel Model to measure the relative performance of pipe and procedure calls. The implementation tested uses TCP as its underlying transport layer, and limits all channels that are exported as a service to be in a single group. Further details of the implementation are given in [Glasser87].

The experiments we ran consisted of making 10, 50, 100, 500, and 1000 channel calls to test the effect of varying k. For each number of calls 0, 100, and 1000 byte arguments were used to test the effect of argument size. Each experiment that tested a combination of number of calls, argument size, and pipe or procedure was repeated 10 times. Experimental values for 10, 50, 100, and 500 procedure call experiments were computed from the corresponding 1000 procedure call experiment.

We first ran these experiments in a low network delay environment. Low-delay was

provided by running the tests on two Mircovax-II workstations on the same 10 MBit/second Ethernet network. In this environment, the average time to perform a 0 byte argument remote procedure call was 13.3 milliseconds, while the average time to perform a 0 byte argument remote pipe call was 4.3 milliseconds.

As shown in the Table 5-1, the measured pipe to procedure performance ratio increases with k as predicated by our simple model. The value for $A(k,p,r)$ was computed by assuming that for 1000 calls that the procedure time was $p + r$ and the pipe time was p. Pipes provide an observed performance improvement, even for only 10 calls, of at least a factor of two.

Table 5-1: Low Latency Experiments

Low Latency Experiments Single Ethernet							
Bytes per Call	Number of Calls k	Procedure Call Time (seconds)	90% Conf. Interval	Pipe Call Time (seconds)	90% Conf. Interval	Pipe/ Procedure Ratio	A(k,p,r)
0	10			0.056	0.006	2.37	2.54
0	50			0.297	0.131	2.24	2.94
0	100			0.439	0.060	3.09	3.00
0	500			2.155	0.010	3.08	3.05
0	1000	13.277	0.034	4.341	0.008	3.06	3.05
100	10			0.062	0.003	2.28	3.23
100	50			0.255	0.004	2.77	4.02
100	100			0.437	0.006	3.23	4.15
100	500			1.178	0.013	4.11	4.26
100	1000	14.136	0.029	3.300	0.010	4.28	4.27
1000	10			0.088	0.005	2.06	2.19
1000	50			0.374	0.005	2.43	2.45
1000	100			0.777	0.065	2.34	2.49
1000	500			3.615	0.076	2.51	2.52
1000	1000	18.153	0.213	7.175	0.015	2.53	2.52

We repeated the same experiments in an environment with a high communication channel latency by employing one node at MIT as the sink and a second node at a distant Arpanet site as the source. The results are shown in Table 5-2. The average time for a null procedure call was 649 milliseconds, which is far longer than the 13

milliseconds measured in the Ethernet case.

As expected, in the high latency case the observed performance improvements are larger because of the increase in r. However, because the source was generating data faster than the Arpanet could absorb them, we did not expect the measured performance ratios to be consistent with our simple performance model. Thus, estimates for $A(k,p,r)$ are not shown in Table 5-2.

One problem we encountered with our high latency experiments is that TCP does not perform well under conditions of high latency, limited bandwidth, and packet loss. In order to improve the performance of the experiments shown in Table 5-2 we adjusted the maximum size of TCP's window size for flow control to be 1100 bytes. Using the standard UNIX 4.2 TCP window size of 16 KBytes, 1000 pipe calls with 1000 byte argument took 3504 seconds. If a rate based flow control algorithm were used instead of TCP's window based approach, such window size adjustments would not be necessary.

Table 5-2: High Latency Experiments

High Latency Experiments Long-Haul Arpanet						
Bytes per Call	Number of Calls k	Procedure Call Time (seconds)	90% Conf. Interval	Pipe Call Time (seconds)	90% Conf. Interval	Pipe/ Procedure Ratio
0	10			3.532	0.167	1.84
0	50			4.770	0.406	6.81
0	100			8.887	0.970	7.31
0	500			15.009	1.640	21.63
0	1000	649.33	51.88	26.717	1.615	24.30
100	10			2.557	0.604	2.74
100	50			14.696	1.371	2.38
100	100			29.649	1.751	2.36
100	500			149.61	7.313	2.34
100	1000	700.82	49.98	305.56	18.60	2.29
1000	10			10.021	1.205	1.12
1000	50			49.477	3.484	1.14
1000	100			80.209	5.267	1.40
1000	500			416.40	23.59	1.35
1000	1000	1123.1	61.2	1414.3	245.5	0.80

In sum, k pipe calls can perform at most $min(1 + \frac{r}{p}, k)$ times better than k procedure calls, where r is the remote communication time and p is procedure execution time. This bound can be approached in practice when sufficient bandwidth is available. Thus, pipes can provide a substantial performance advantage over procedures for many applications.

5.2. The Elements of the Model have been Proven Practical

In order to gain experience with the Channel Model we have used it to implement a distributed database system. The database system we implemented provides query based access to the full-text of documents and newspaper articles, and is presently in use by a community of users. The database system is divided into a user interface portion called Walter that runs on a user's local node, and a confederation of remote database servers which are accessed by Walter via the DARPA Internet. Walter employs a query routing algorithm to determine which server contains the information required for processing of a given user query.

The protocol that Walter uses to communicate with a database server is built using the Channel Model. A stub generator automatically generates both source and sink node stubs from an interface file. This interface file (rewritten from C into Modula-II) is shown below:

```
REMOTE INTERFACE DataBase;
   PIPE EstablishQuery(c: ARRAY OF CHAR);
     (* establishes a new query *)
   PROCEDURE CountMatchingRecords(): INT;
     (* Returns the number of records that have matched so far *)
     (* Number is positive if query processing is complete *)
   PROCEDURE FetchSummary(r: Range, dest: PIPE[Summary]);
     (* Causes summaries in range r to be sent to dest *)
   PROCEDURE FetchRecord(rec: INT, r: Range, dest: PIPE[Line]);
     (* Causes lines in range r of record rec to be sent
        to dest *)
   PIPE Abort();
     (* Causes the server to abort the query and any data that it
        is sending down a pipe *)
   SEQUENCE EstablishQuery, CountMatchingRecords, FetchSummary,
            FetchRecord;
     (* Don't allow the query to change during requests *)
END DataBase.
```

When a user supplies a query the procedure `EstablishQuery` is called. `EstablishQuery` initiates processing of a query at a server. The server procedure `FetchSummaries`, which computes the summaries for a range of articles matching the current query is then called. As the summaries are computed, they are sent down the pipe supplied in the `FetchSummaries` call. The pipe sink procedure that receives the summaries displays them as they arrive. All of the summaries generated by `FetchSummaries` are guaranteed to arrive before `FetchSummaries` returns. In order to view an entire database record, the server procedure `FetchRecord` is used in precisely the same manner as `FetchSummaries` is used. If a user requests that a request be aborted while data are arriving down a pipe the `Abort` pipe is called to abort the `FetchSummaries` or `FetchRecord` operation in progress.

The use of pipes in this database application has provided two distinct advantages over remote procedures. First, pipes permit both `FetchSummaries` and `FetchRecord` to send variable amounts of bulk data to Walter simply. Second, since pipe calls do not block, a server can continue computing after it has sent a datum. If a procedure instead of a pipe were used to return data the server process would suspend processing while waiting for a response from Walter. The concurrency provided by pipes has proven to be important to Walter's performance in practice.

5.3. Advantages of the Channel Model

We have proposed three major ideas:

- *Channel values* Channels should be first-class values which can be freely transmitted between nodes. If a communication model does not not permit channel values to be transmitted between nodes, then its application will be limited to a restricted set of protocols. An application of channel values is the return of incremental results from a service to a client.

- *Pipes* A new abstraction called a pipe should be provided in the communications model. A pipe permits bulk data and incremental results to be transmitted in a type safe manner in a remote procedure call framework. Existing remote procedure call models do not address the requirements of bulk data transfer, or the need to return incremental results.

- *Channel groups* A new sequencing technique, the channel group, is important in order to permit proper sequencing of channel calls. A channel group is used to enforce serial sequencing on its members with respect to a single source process.

As we have explained, these three ideas form the basis for the Channel Model. We expect that this model will find a variety of applications in distributed systems.

Acknowledgments The ideas in this paper benefited from meetings with fellow MIT Common System Project members Toby Bloom, Dave Clark, Joel Emer, Barbara Liskov, Bob Scheifler, Karen Sollins, and Bill Weihl. I am especially indebted to Bob Scheifler for posing a question that resulted in the notion of a channel group. Andrew Birrell, Barbara Liskov, John Lucassen, Bob Scheifler, Mark Sheldon, Bill Weihl, Heidi Wyle, and Kendra Tanacea commented on drafts of the paper.

6. Appendix: Explicitly Serviced Pipes

An alternate model for the sink end of a pipe is to allow a program to create a pipe that is explicitly serviced. This is accomplished by using the procedure `Create` to create a pseudo-pipe sink procedure. However, unlike a pipe sink procedure, this procedure is not called when data arrive down the pipe. Instead, data must be explicitly taken from the pipe with the following procedures:

```
DEFINITIONS MODULE ExplicitPipe;
  TYPE Pipe = PROCEDURE ();
  PROCEDURE Create(): Pipe;
     (* Returns a new local procedure that can be used as
        a pipe sink *)
  PROCEDURE Value(p: Pipe): Any;
     (* Returns the next value from p.  Blocks if no value
        is present until a value arrives.  Successive applications of
        Value will return the same value unless Accept has
        been called  *)
  PROCEDURE Accept(p: Pipe);
     (* Accepts the last datum read with Value, and permits
        Value to get the next value from the pipe.  Blocks its caller
        if no data has arrived for the pipe.  Once Accept discards the
        present value, it does not block its caller waiting for the next
        pipe value *)
  PROCEDURE Ready(p: Pipe): BOOLEAN;
     (* Returns TRUE if there is no data waiting in p, FALSE otherwise *)
END ExplicitPipe.
```

A simple example of how a pipe can be explicitly serviced follows:

```
my-pipe := Create();
ftp.GetFile("fred.txt", my-pipe);
  (* pass the new pipe to remote source *)
c := Value(my-pipe);
Accept(my-pipe);
  (* remote source will terminate with EOF *)
WHILE c#EOF DO
  term.PutChar(c);
  c := Value(my-pipe);
  Accept(my-pipe);
  END;
```

`Accept` defines the sequencing semantics of explicitly serviced pipes. Until `Accept` is called to acknowledge receipt of a call, further calls on the same channel will not be processed.

We call pipes which are connected to a procedure *procedure serviced*, and pipes which

are polled *explicitly serviced.* We expect that both procedure serviced and explicitly serviced pipes will find application.

7. References

[Bershad86] Bershad, B. et. al, A Remote Procedure Call Facility for Heterogenous Computer Systems, Technical Report 86-09-10, Computer Science Department, Univeristy of Washington, Septebmer, 1986.

[Birrell84] Birrell, A., and Nelson, B., Implementing Remote Procedure Calls, ACM Trans. on Computer Systems 2, 1 (February 1984), pp. 39-59.

[Birrell85] Birrell, A., Secure Communication Using Remote Procedure Calls, ACM Trans. on Computer Systems 3, 1 (February 1985), pp. 1-14.

[Brown85] Brown, M., et. al., The Alpine File System, ACM Trans. on Computer Systems 3, 4 (November 1985), pp. 261-293.

[Cheriton87] Cheriton, D., VMTP: Versatile Message Transaction Protocol, Computer Science Department, Stanford University, January 12, 1987.

[Gifford81] Gifford, D., Information Storage in a Decentralized Computer System, Report CSL-81-8, Xerox Palo Alto Research Center, Palo Alto, CA.

[Gifford85] Gifford, D. et. al., An Architecture for Large Scale Information Systems, Proc. of the Tenth ACM Symposium on Operating Systems Principles, ACM Ops. Sys. Review 19, 5, pp. 161-170.

[Gifford86] Gifford, D., Remote Pipes and Procedures for Efficient Distributed Communication, Report MIT/LCS/TR-384, MIT Laboratory for Computer Science, October, 1986.

[Glasser87] Glasser, N., The Remote Channel System, M.S. Thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, May 1987.

[Herlihy82] Herlihy, M., and Liskov, B., A Value Transmission Method for Abstract Data Types, ACM Trans. on Programming Languages and Systems 4, 4 (October 1982), pp. 527-551.

[Jones85] Jones, M., et. al., Matchmaker: An Interface Specification Language for Distributed Processing, Proc. of the 12th Annual ACM Symp. on Princ. of Prog. Languages, January 1985, pp. 225-235.

[Lindsay84] Lindsay, B. et al., Computation and Communication in R*: A Distributed Database Manager, ACM Trans. on Comp. Sys. 2, 1 (February 1984), pp. 24-38.

[Liskov83] Liskov, B., and Scheifler, R., Guardians and Actions: Linguistic Support for Robust, Distributed Programs, ACM Trans. on Prog. Lang. and Sys. 5, 3 (July 1983), pp. 381-404.

[Martin87] Martin, B., PARPC: A System for Parallel Procedure Calls, to appear, Proc. 1987 Int. Conf. on Parallel Processing.

[Needham78] Needham, R., and Schroeder, M., Using Encryption for Authentication in Large Networks of Computers, Comm. ACM 21, 12 (December 1978), pp. 993-998.

[Nelson81] Nelson, B., Remote Procedure Call, Report CSL-81-9, Xerox Palo Alto Research Center, May 1981.

[Postel79] Postel, J., Internetwork Protocols, IEEE Trans. on Comm. COM-28, 4, pp. 604-611.

[Redell80] Experience with Processes and Monitors in Mesa, Comm. ACM 23, 2 (February 1980), pp. 105-117.

[Ritchie74] Ritchie, D.M., and Thompson, K., The UNIX Time-Sharing System, Comm. ACM 17, 7 (July 1974), pp. 365-375.

[Voydock83] Voydock, V., and Kent, S., Security Mechanisms in High-Level Network Protocols, Comp. Surveys 15, 2 (June 1983), pp. 135-171.

[White76] White, J., A high-level framework for network-based resource sharing, Proc. Nat. Comp. Conf. 1976, AFIPS Press, pp. 561-570.

[White82] White, J., and Dalal, Y., Higher-level Protocols Enhance Ethernet, Electronic Design 30, 8 (April 1982), pp. ss33-ss41.

[Xerox81] Courier: The Remote Procedure Call Protocol, Xerox System Integration Standard XSIS 038112, Xerox Corporation, Stamford, Connecticut, December 1981.

[Young87] Young, M., et. al, "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System", Proc. of the 11th ACM Symposium on Operating System Principles, November 1987.

CS-TR Scanning Project
Document Control Form

Date : 3/23/95

Report # LCS-TR-384

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 38(44-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☒ DOD Form ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☒ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
IMAGE MAP (1) UNNUMBERED TITLE PAGE	
(2-3) PAGES #'ED i-ii	
(4-10) PAGES #'ED 1-7,	
(11-12) UN#'ED FIG 1 & FIG 2	
(13-) PAGES #'ED 8-	
(14) UN#'ED FIG 3	
(15-27) PAGES #'ED 9-21	
(28) UN#'ED FIG 4; (29-38) PAGES #22-31	
(39-41) SCANCONTROL, PRINTERS NOTES, DOD, TRGTS (3)	

Scanning Agent Signoff:
Date Received: 3/23/95 Date Scanned: 4/6/95 Date Returned: 4/10/95

Scanning Agent Signature: Michael W. Cook

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR-384	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-83-K-0125
6a. NAME OF PERFORMING ORGANIZATION MIT Laboratory for Computer Science 6b. OFFICE SYMBOL (*If applicable*)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Department of Navy
6c. ADDRESS (*City, State, and ZIP Code*) 545 Technology Square Cambridge, MA 02139	7b. ADDRESS (*City, State, and ZIP Code*) Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (*City, State, and ZIP Code*) 1400 Wilson Blvd. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE (*Include Security Classification*)
Remote Pipes and Procedures for Efficient Distributed Communication

12. PERSONAL AUTHOR(S)
Gifford, David

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
Technical	FROM _____ TO _____	1986 October	24

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	remote-procedure call; flow control; bulk data ransfer

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

The advantages of remote procedure call are combined with the efficient transfer of bulk data and the ability to return incremental results in a new communication model for distributed systems. Three ideas form the basis of this model. First, remote procedures are first-class values which can be freely exchanged among nodes, thus enabling a greater variety of protocols to be directly implemented in a remote procedure call framework. Second, a new type of abstract object called a pipe allows bulk data and incremental results to be efficiently transported in a type safe manner. Unlike procedure calls, pipe calls do not return values and do not block a caller. Data sent down a pipe is received by the pipe's sink node in strict FIFO order. Third, the relative sequencing of pipes and procdures can be controlled by combining them into channel groups. A channel group provides a FIFO sequencing invariant over a collection of channels. Application experience with this model, which we call the "Remote Pipe and Procedure Model", is reported.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Judy Little, Publications Coordinator	22b. TELEPHONE (*Include Area Code*) (617) 253-5894 22c. OFFICE SYMBOL

☆U.S. Government Printing Office: 1985—507-047

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 4/06/1995

M.I.T. Libraries
Document Services