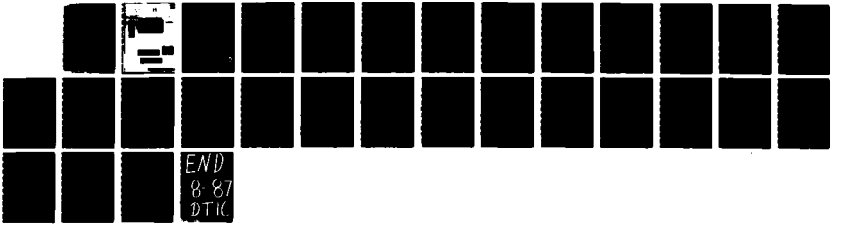
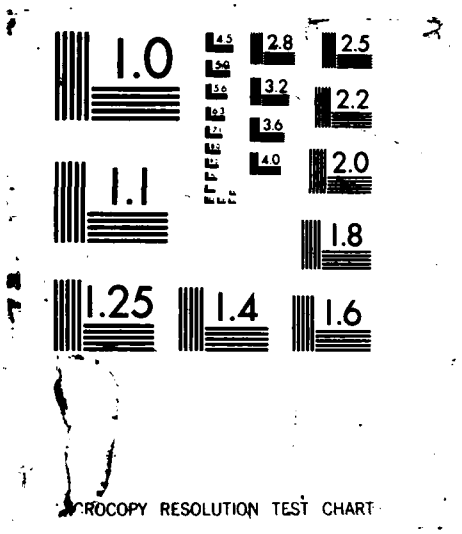
AD-A182 176 REMOTE PIPES AND PROCEDURES FOR EFFICIENT DISTRIBUTED COMMUNICATION(U) MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE D K GIFFORD OCT 86 1/1

UNCLASSIFIED MIT/LCS/TR-384 N00014-83-K-0125 F/G 12/7 NL




1.0
1.1
1.25
1.4
1.6
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.8
ROCOPY RESOLUTION TEST CHART

LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TR-384

Remote Pipes and Procedures for Efficient Distributed Communication

David K. Gifford

MIT/LCS/TR-384

Remote Pipes and Procedures for Efficient Distributed Communication

David K. Gifford

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

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE A182176

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

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

GUB Government Printing Office: 1985—607-047

Remote Pipes and Procedures for Efficient Distributed Communication

David K. Gifford

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 procedures 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.

Table of Contents

1. The Remote Pipe and Procedure Communication Model

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 uses 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 remote procedure values to be exchanged freely 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 can not be expressed in a remote procedure call framework.

2. *Incremental Results* Consider a server that is computing a result on behalf of a client and wishes to communicate incremental results to the client as they are computed. In present remote procedure call systems this would be accomplished by having the client ask the server to compute the first incremental result, then the second, and so forth until all of the results have been computed. The problem with this approach is that it forces a single computation to be decomposed into a series of distinct remote procedure calls. This decomposition reduces the performance of the server since it is inactive between client procedure calls unless it creates a sophisticated process structure upon the client's first incremental result request. 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 only one remote procedure call can be in transit at a given time 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 was 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 *Remote Pipe and Procedure Model* that 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 allows bulk data and incremental results to be efficiently transported. Third, we introduce *channel groups* which control the relative sequencing of calls on pipes and procedures.

Elements of the Remote Pipe and Procedure Model have been present in previous work although these elements have not been combined into a single consistent framework. 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 which limits the generality of remote procedures. The notion of a pipe is similar in some respects 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 Matchmaker remote procedures are not first-class values and unidirectional message sends are not optimized for bulk data transmission. None of the above systems include an idea similar to a channel group.

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

2. Semantics

We discuss in this section

- how remote pipes and procedures are typed,
- how remote pipes and procedures are created and used,
- stable channels that survive crashes,
- channel call ordering,
- how channel groups provide inter-channel synchronization,
- secure channels for secrecy and authentication,
- and failure semantics.

For the purposes 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 will depend on application requirements. All of the nodes in a system are interconnected by a network.

Remote procedures and pipes are defined as follows:

- *Procedures* A *remote procedure value* provides the capability to call a procedure on a remote node, and is used as is a local procedure value. A *remote procedure call* blocks the caller until the remote procedure has finished execution and a response has been received from the remote node. In the event that a remote procedure call fails, a distinguished error value is returned as the value of the call.

- *Pipes* A *pipe value* provides the capability to send bulk data to a remote 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 call does not block its caller a pipe call implicitly initiates concurrent activity at the sink's remote *sink* node. The caller continues execution as soon as the call to the sink is queued for transmission. The data values sent down a pipe by a given process are received by the pipe's sink node in strict FIFO order. Received means that the sink receives 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.

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 freely passed 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 3.

2.1. Channels and interfaces have static types

Every channel has a statically known type. A channel's type describes the channel's input values and the channel's result values. For example a pipe value might have type

```
pipe-a: pipe[string]
```

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

```
proc-a: proc[string] returns [sequence[byte]]
```

indicating that the remote procedure takes a string and returns a sequence of bytes.

Local procedures have type **subr** instead of type **proc**. We restrict **pipe** and **proc** *types so that they can not include* **subr** *types as parameters or results.* Thus, a local procedure must be converted to a remote procedure before it can be used with a channel. This conversion is implicitly performed as we will describe in a moment.

We assume in our model that nodes have disjoint address spaces and thus call by value semantics must be used for channel calls. One consequence of the lack of a shared address space is that remote and local procedure calls can not provide the same semantics for mutable objects.

The semantics of remote and local calls are rationalized in our model by insisting that channel parameters and results be instances of *immutable types*. Instances of immutable types cannot be modified; this allows remote and local calls to have consistent semantics. In order to ease the burden on the programmer (who will often be working with mutable types) implicit conversions between mutable and immutable types are performed at a remote procedure or pipe call site as necessary.

A *remote interface* consists of a set of named channels and can be represented in our model by combining a set of channels into an aggregate structure such as a *package*. The following is an example of a package containing two channels.

```
r: package[print: pipe[string],
           read: proc[string] returns[sequence[byte]]]
```

Channel values can be selected from packages in the same manner as values are selected from records. For example, `r.print` could be used to select the print pipe from the package `r`. New packages can be composed by assembling an arbitrary set of channels, including channels which have been selected from existing packages.

Other aggregate structures (such as records or arrays) can also be used to combine channels. However packages are attractive because their type compatibility rules present the programmer with more flexibility than standard aggregate types. One package is compatible with another package even if the first package contains extra fields or a set of fields which is not in the same order.

Packages by virtue of their type declarations contain enough information to permit a stub generator 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 how information is encoded and transmitted over a wire.

2.2. Channels are used and created like procedures

A remote procedure value is used in the same manner as is a local procedure value. Thus to call the procedure `proc-a` declared above, the source expression

```
my-sequence := proc-a("myfile.txt");
```

could be used. This expression would result in a call message to the node that implements `proc-a`, the execution of `proc-a` with the string parameter "myfile.txt", a reply message containing the sequence of bytes computed by `proc-a`, and the assignment of this sequence of bytes to `my-sequence`.

A new remote procedure value is created by providing a local procedure (a `subr`) where a remote procedure (a `proc`) is expected. In the following example a remote procedure value will be implicitly created from `local-proc-a`:

```
t: int := 0;
local-proc-a = subr[x: int]
   t := x + t;
   end;
% remote-proc-a (which is at a remote node)
% can call local-proc-a during its execution
result := remote-proc-a(local-proc-a);
```

Note that **local-proc-a** is a closure and is able to access variables in its environment. All local and remote calls to **local-proc-a** will update the same variable **t**.

A remote procedure value can also be created by returning a local procedure as the value of a remote procedure call.

```
remote-proc-b = subr[] returns [proc[int]]
  return(local-proc-a);
  end;
```

In this example a program that calls **remote-proc-b** will receive a remote procedure value which will enable the program to call **local-proc-a**.

The actual creation of the remote procedure value that corresponds to **local-proc-a** in both of these examples is performed implicitly as a conversion between objects of type **subr** (local procedures) and objects of type **proc** (remote procedures). No conversion needs to be performed in the opposite direction. Remote procedures are compatible with local procedures, and remote procedures can be provided where a local procedure is expected.

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 "first" and "second" down **pipe-a**:

```
pipe-a("first");
pipe-a("second");
```

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

Since pipe calls do not return values and are processed asynchronously, a *synchronize* operation is provided. When a *synchronize* operation is applied by a source process to a pipe, the pipe's sink is forced to process all outstanding data sent down the pipe 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.

```
pipe-a("mit-cls:/usr/smith/rpp.imp")
pipe-a("mit-db:/usr/gifford/545.tmp")
code := synchronize(pipe-a);
```

A pipe value is created through the provision of a local procedure called a *pipe sink procedure*, that will process data received over the pipe. As data arrives through a pipe its corresponding sink procedure is applied to each datum in FIFO order. 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. A pipe can be declared as follows:

```
total: int := 0;
pipe-b = pipe[val: int]
  total := total + val;
  end;
remote-proc-d(pipe-b);
% remote-proc-d synchronizes pipe-a before it returns
% thus, we know that total is the sum of all pipe calls
% made by remote-proc-d
```

We have assumed that remote procedures will be implicitly created from local procedures when necessary, and that language support is provided for pipe declaration. The Remote Pipe and Procedure Model can be used in a language environment without making these assumptions by introducing the primitives **make-pipe** and **make-proc**. The primitive **make-pipe** creates a new pipe value when supplied with a local pipe sink procedure, and the primitive **make-proc** creates a remote procedure when provided with a local procedure for remote export. In the example below **make-pipe** and **make-proc** are used to create a remote procedure and a pipe.

```
proc-b := make-proc(local-proc-b);
pipe-c := make-pipe(local-proc-c);
```

An alternate model for the sink end of a pipe is to allow a program to create a pipe that has no sink procedure. The following primitives are used by a sink node to obtain values which have been sent down a pipe that has no sink procedure. The following primitives will return a distinguished error value if they are not applied at a pipe's sink node.

- **pipe-value** returns the next value from a pipe. If no value is available **pipe-value** blocks until a value arrives. **pipe-value** does not remove a value from a pipe. Successive applications of **pipe-value** to a pipe will return the same result unless **pipe-accept** has been called.

- **pipe-accept** states that the last datum read with **pipe-value** has been accepted and the datum may be removed from the pipe. **pipe-accept** blocks its caller if no data has arrived for the pipe. Once **pipe-accept** discards the present value of a pipe, it does not block its caller waiting for the next pipe value.

- **pipe-get** gets and accepts the next value from a pipe. Thus this operation is equivalent to combining **pipe-value** with **pipe-accept**.

- **pipe-ready** is a predicate that returns true if a pipe has data available and false if no data has arrived down the pipe.

A simple example of how a pipe can be serviced follows.

```
total: int := 0;
pipe-d: pipe[int] := pipe-create();
remote-proc-e(pipe-d);     % pass the new pipe to remote source
val: int := pipe-get(pipe-d);
while val#0 do             % remote source will terminate with 0
  total := val + total;
  val := pipe-get(pipe-d);
  endloop;
```

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.

When program starts it will need to obtain appropriate channels in order to communicate with other nodes. This is accomplished by providing a program with a system supplied channel to a clearinghouse service. The program can use the clearinghouse service to obtain other channels of interest and to make channels that it creates known to other nodes.

2.3. Stable channels survive crashes

The above examples have shown how both remote procedures and pipes can be dynamically created, but their lifetimes have not been discussed. The desired lifetime of a channel depends upon its application. Thus in our model a *dynamic channel* will exist

until it is explicitly destroyed by a program or until the channel's sink node crashes. An attempt to call a remote procedure which has been destroyed will result in a distinguished error value, and an attempt to call a pipe which has been destroyed will result in a distinguished error value from `synchronize`.

Channels which can survive node failures are useful for stable services that are registered with a clearinghouse. We call a channel that can survive a node failure a *stable channel.* The state of a stable channel and its associated procedure must be recorded in stable storage to permit recovery of the channel upon node restart. The details of how stable channels are created will depend on the host language environment.

2.4. 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 in which they were made by the process. Processed means that the second call is not processed at the sink until the procedure invoked by the first call has returned. In the case of explicitly serviced pipes processed means that the second call will not be processed until a process has accepted the first call's datum by executing `pipe-accept`.

The ordering of channel calls not covered by the above invariant is undefined. Thus, a single channel can be invoked simultaneously by different source processes. We assume that monitors [Redell80] or a similar mechanism is used to ensure the proper operation of remote procedures and pipes in the presence of concurrent invocations.

2.5. Channel Groups provide inter-channel FIFO 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. For example, imagine that we model a remote color display as a package with two channels: `set-color` and `put-character`. `set-color` sets the color of subsequent characters and returns the previous color setting of the display, while `put-character` is used to write characters on the display.

```
display-a: package[set-color: proc[Color] returns[Color],
                   put-character: pipe[char]]
```

In our present model there is no way to specify that calls on `set-color` and

put-character must be performed in the order in which they were made. Thus if we used **display-a** characters could be displayed in the wrong color.

When timing invariants must be preserved between a set of channels the channels can be collected into a *channel group*. A channel group value is a collection of pipes and procedures that all have the same sink node and that observe FIFO ordering constraints with respect to a source process. A channel group value has a distinguished type. For example the following group is identical to **display-a**, except that the group guarantees that calls made by a single process will be performed in the order in which they were made:

```
display-b: group[set-color: proc[Color] returns[Color],
                put-character: pipe[char]]
```

The type compatibility rules for groups permit extra fields, out of order fields, and for a group value to be provided where a package value is expected. Thus, a package may in fact be a channel group. The type **group** is provided to allow the static enforcement of FIFO sequencing where desired.

A group constructor is used to create a channel group value. A group constructor assembles a set of procedure and/or pipe values into a single composite channel group. All of the members of a group must reside at the same sink node. If they do not the group constructor will return a distinguished error value. We could have used the expression

```
display-b := group[set-color: display-a.set-color,
                   put-character: display-a.put-character];
```

to create the group value **display-b**.

A group constructor copies the values of its component channels, creates a new unique *sequencing stamp*, adds this new stamp to each of the channel copies, and constructs a group value out of the stamped copies. The sequencing stamp added to each channel is used to identify its membership in the newly created group. In addition to the sequencing stamps obtained by group membership, upon creation each channel is assigned a unique sequencing stamp.

The individual components of a channel group can be selected in the same manner as can the components of a record. Once a channel is selected from a group the channel can be used. For example, in order to display a character one could issue the call:

```
display-b.put-character["d"];
```

It is possible that a channel which is selected from a channel group will be included in another group. In this case the resulting channel will have more than two sequencing stamps (one unique stamp, and two or more group stamps). If desired, one could prohibit selection on group values in order to limit a channel to a single group sequencing stamp. We will not place this restriction on group values.

Because channels can be selected from groups it is possible to create a package that includes channels that are members of a group. Consider the following package that implements a terminal consisting of a color display and a keyboard:

```
terminal := package[set-color: display-b.set-color,
                    put-character: display-b.put-character,
                    get-character: get-char],
```

set-color and **put-character** calls will be performed in the order in which they are made, but **get-character** calls are not ordered with respect to the display calls. This allows characters to be independently displayed and read on a remote terminal.

The channel timing invariant provided by the communication 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 sequencing stamp in common, then the calls will be processed at the sink in the order in which they were made by the process.

Processed means that the second call is not processed at the sink until the procedure invoked by the first call has returned. In the case of explicitly serviced pipes, processed means that the second call will not be processed until a process has accepted the first call's datum by executing **pipe-accept**.

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 sequencing stamp in common with itself). The channel timing invariant describes all of the ordering provided by the Remote Pipe and Procedure Model. The ordering of channel calls not covered explicitly by the channel timing invariant is undefined. Table 1 shows some of the implications of the channel timing invariant.

2.6. Secure Channels for Secrecy and Authentication

In order to provide secrecy and authentication in our communication model we introduce the idea of a *secure channel*. A secure channel has the same type as does a normal channel and is used in the same manner. Secure channels however, provide additional secrecy and authentication guarantees. These guarantees are provided with cryptographic techniques.

Secure channels can be created only with the assistance of an *authentication service*. An authentication service is a trusted entity which is charged with establishing secure conversations between *principals* in a system [Voydock83]. Principals are the unit of authorization in our model of communication. Each principal has an associated secret key. The keys that are possessed by a node define its principals.

The first step in establishing a two-way authenticated secure conversation is to obtain a *conversation* [Birrell85] from an authentication server. A conversation can be obtained from an authentication server by using an unprotected remote procedure call. For example:

```
conversation  = as two-way[source-principal, sink-principal].
```

A conversation includes a conversation key that is encrypted under the source principal's secret key. Once a conversation is obtained no further interaction with an authentication server is necessary.

One or more secure channels can be created from a single conversation. **secure** creates a secure channel given a channel. For example:

```
secure-chan  = secure(proc-a, conversation).
```

secure can be used to secure a single channel, a package of channels, or a channel group.

The following three invariants are guaranteed for secure channels which have been created with a two-way authenticated conversation.

- *Secrecy* Information sent over **secure-chan** will be kept secret from all principals except **source-principal** and **sink-principal**.
- *Source Authentication* During a call on **secure-chan** the sink can use the primitive **SourcePrincipal** to obtain the authenticated principal identifier

of the process that made the call. Channel calls from a given principal are protected against modification and replay.

- *Sink Authentication* Results that are returned from **secure-chan** are guaranteed to have come from a node that is authenticated as **sink-principal**.

One-way authenticated conversations are also possible within this framework. Needham and Schroeder [Needham78] discuss one way authenticated conversations and other possible extensions.

2.7. Failures complicate channel semantics

Our communication model guarantees that a remote operation (a pipe or a procedure 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. For procedure calls the occurrence of a failure causes a distinguished *failure* value to be returned as the result of the call. For pipe calls the occurrence of a failure causes a distinguished *failure* value to be returned as the result of **synchronize**. When a failure occurs it is impossible to determine whether a remote operation was completed, never started, 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 *failure* 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 is implemented by protecting the actions of a remote

operation with a transaction. If a remote operation returns *failure* the operation's 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 innovative 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 failure, 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 thus no compensation needs to be performed.

3. Pragmatics

We discuss in this section

- a model implementation,
- and performance elaborations.

3.1. A model implementation is possible

In order to demonstrate that our communication model can be effectively realized we present a model implementation. The model implementation describes the key algorithms that are necessary to implement the Remote Pipe and Procedure Model. The implementation describes the format of messages and channel values, how calls are generated and sequenced, message retransmission and timeout, incoming call processing, and crash recovery. The section following the model implementation considers performance optimizations.

We assume that the message system may lose, reorder, and duplicate messages. In addition when a message is delivered we assume that it has not been damaged in transmission. This ideal can be realized with high probability in practice if messages

with incorrect checksums are discarded.

We also assume that typed values can be converted to byte strings and back again via encode and decode operations [Herlihy82]. We consider neither a particular type space or exception handling.

Finally we assume that when a node crashes it loses all of its volatile state. The entire state of the following implementation is stored in volatile storage, except for the state necessary for global unique id generation.

A channel value consists of the channel's sink node address, the channel's unique *channel id*, the type of the channel (pipe or procedure), and the channel's set of sequencing stamps. A channel has one unique sequencing stamp for each group membership, along with a sequencing stamp which is a copy of the channel id.

```
channel = record[sink  Address,
                 channel-id  GlobalUniqueID,
                 type  (pipe, proc),
                 stmps  Stamps],
```

When a channel is called the procedure **SourceCall** is invoked. **SourceCall** is passed the channel being called and its encoded parameters. We assume that an encoding algorithm such as Herlihy's [Herlihy82] is used to convert the actual parameter values to a format for transmission in a message.

SourceCall constructs a call message that includes the sink and source addresses, the *generation id* of the source and sink, a unique request id, the process id of the calling process, a *sequence vector*, the channel called, and the data for the channel. A node's generation id is the value of the node's unique id generator at the time of the node's last recovery from a crash.

```
message = record[sink, source  Address,
                 sink-generation  GlobalUniqueID,
                 source generation  GlobalUniqueID,
                 type  (call, return),
                 request-id, process-id  GlobalUniqueID,
                 sequence  SequenceVector,
                 channel-id  GlobalUniqueID, data  any],
```

Channel calls are ordered with sequence vectors. A sequence vector is a set of pairs of the form <sequence stamp, integer>. Two sequence vectors are ordered if and only if they have a sequence stamp in common. The comparison of two ordered sequence vectors is accomplished by comparing the integers associated with common sequencing

stamps. The larger sequence vector will have larger integers associated with all of the common stamps (because of the way sequence vectors are generated).

SourceCall generates a unique sequence vector for each call by calling the procedure **GetNextSequence**. Each process has its own independent counter for each sequence stamp. **GetNextSequence** takes as input both the id of the process making the call and the sequence stamps that are to be used for this call. **GetNextSequence** increments the per-process counters associated with the input sequence stamps, and returns the new counter values along with their stamps as the next sequence vector.

```
SourceCall = subr[c  channel, data  any] returns [any]
  m  = message$create[sink  c sink,
                      source  GetMyAddress[],
                      source-generation  GetMyGeneration[],
                      sink-generation  GetGeneration[c sink],
                      request-id  GetMonotoneUniqueID[],
                      process-id  GetMyProcessID[],
                      sequence  GetNextSequence[c stamps,
                                                GetMyProcessID[]],
                      channel-id  c channel-id,
                      data  data],
  RegisterOutgoing[m, c type],
  Send[m],
  if c type=pipe then return[nil]
    else return[WaitForResult[m]],
  end,
```

Each request message includes both source and sink generation ids. A sink must reject any message that specifies an obsolete sink generation (because the message was destined for a previous incarnation of the sink) and must also reject any message that specifies an obsolete source generation (because the message originated from a previous incarnation of the source node).

The set of remote generation id's held by a node comprises its connection state. As shown **GetGeneration** returns a node's cached copy of a remote node's generation id. If the generation id of a remote node is not cached **GetGeneration** must send a message to the remote node requesting its generation id.

The source repeats a call message if the call's corresponding reply has not been received after a predetermined interval. This is accomplished by registering all outgoing call messages with the procedure **RegisterOutgoing**. A retransmission process periodically retransmits the set of registered calls. Note that both pipe calls and procedure calls are retransmitted.

Incoming messages to a node are demultiplexed into call and reply messages. This is accomplished by the following loop that receives incoming messages and dispatches call messages to **SinkCall**, and reply messages to **ProcessReply**.

```
while true do
  m := receive[];
  if m.type = call
    then SinkCall[m]
    else ProcessReply[m];
  end;
```

ProcessReply looks up the **request-id** of an incoming message in the set of outstanding calls maintained by **RegisterOutgoing**. If the outstanding call corresponding to an incoming message is not found the incoming message is discarded. If a corresponding outstanding call is found the call is taken out of the outstanding call set. If the incoming message is a reply to a procedure call **m.data** is returned to the calling process that is waiting inside of **WaitForResult**. If the incoming message is a reply to a pipe call, all other outstanding calls with smaller request ids, which are in the outgoing call set for this pipe and process, are removed. These calls are removed because we know, based on the channel timing invariant maintained by the server, that the calls must have been processed previously.

If a reply to a call in the outstanding call set is not received within a predefined interval, the process that retransmits messages will eventually force the call to terminate. The retransmission process accomplishes this by providing **ProcessReply** with a fake reply message for calls with a distinguished *failure* value for **m.data**.

After a sink crash the source must resynchronize with the sink node. This is accomplished by querying the sink for its generation id when the sink fails to respond to a call message. If the sink replies with a generation id value different from that cached by the source, then the sink has crashed and restarted. In this case all of the source's sequence counters for sequence stamps associated with the sink are reset to 0, and the cached copy of the sink's generation id is updated.

SinkCall processes incoming call request messages. **SinkCall** first determines if the received message was generated by a previous incarnation of the source node. The message is ignored if the message's source generation field is less than the sink's copy of the source's generation. If the message's source generation field is greater than the sink's copy of the source's generation then the source has crashed, and the sink must update its copy of the source's generation id, abort any calls in progress from the

previous generation of the source, as well as garbage collect sequencing state and held results associated with the previous generation of the source node.

```
SinkCall = subr[m: message]
  g := GetGeneration[m.source];
  if m.source-generation < g then return[];
  if m.source-generation > g then
    UpdateGeneration[m.source, m.source-generation];
  % invariant: m sent from current incarnation of source
  if m.sink-generation # GetMyGeneration[] then return[];
  % invariant: m sent to current incarnation of sink
  channel := LookupSinkChannel[m.channel];
  if channel=nil then begin
     UnknownChannelReply[m];
     return[];
     end;
  % see if we have the result
  if HaveResult[m] then begin
     SendResult[m, GetResult[m], channel];
     return[];
     end;
  if not NextInSequence[channel, m]
    then return[];
  FORK DoCall[channel, m];
  end;
```

After **SinkCall** ensures that **m** is a contemporary message **m.channel-id** is looked up by **LookupSinkChannel**. If the channel does not exist a distinguished response is returned to the source. If the channel does exist the sink checks to see if it has already computed the result for **m**. The sink will have a held result if **m** is the last procedure call processed from the source process, or if **m** is a pipe call with an old sequence vector. Pipe calls with obsolete sequence vectors have already been processed by the sink and thus can simply be acknowledged. If **SinkCall** can return a response immediately it does so.

If **SinkCall** cannot immediately create a response for the incoming message, **NextInSequence** checks the per-process sequence counters to ensure that **m.sequence** is the next operation to be scheduled for the channel, and that the operation is not already in progress. If either of these conditions is not met **NextInSequence** returns false and the message is ignored. Otherwise **NextInSequence** increments the per-process counters, notes that the operation is in progress, and returns true.

DoCall is forked by **SinkCall** to actually perform the computation requested by the source node. After the processing is complete **Finished** is called to update the per-process sequence stamp counters for **m.process-id**, and indicate that the current

call is finished. **DoCall** then uses **SendResult** to send a result message back to the source. **SendResult** also remembers the most recent procedure call result computed by each process.

```
DoCall = subr[c: sink-channel, m: message]
   result  := c.procedure[m.data];
   Finished[m.process-id, m.sequence];
   SendResult[m, result, c];
end;
```

synchronize can be added to our model implementation by sending a distinguished synchronize operation to a pipe and waiting for the sink to respond. The synchronize operation will be processed after all other outstanding pipe calls have been processed by virtue of the sink's normal message sequencing mechanism. When the sink acknowledges the synchronize operation **synchronize** returns a value signifying normal completion. If the synchronize operation is not acknowledged within a certain time **synchronize** returns a distinguished error value.

The **secure** primitive for creating secure channels creates a channel value that includes both a regular channel value and a conversation. A detailed treatment of the implementation of secure one-way and two-way authenticated communication is discussed in [Needham78]. A complete system that uses this information for a two-way authenticated remote procedure call mechanism is described in [Birrell85].

3.2. 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 Remote Pipe and Procedure Model would require performance optimizations. Important optimizations include:

- *Buffer pipe calls* Multiple pipe calls destined for the same sink node can be buffered at a source and transmitted as a single message in order to reduce message handling overhead and improve network throughput. The amount of time that 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, and any buffered pipe calls to the same sink should be prepended to the procedure call message whenever possible.

- *Preallocate Processes* Processes can be preallocated 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 on 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 a result having been produced. This acknowledgment informs the source that the call has been successfully received, and that 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 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.

4. Practical Experience and Conclusions

We conclude with

- experience with an application of the Remote Pipe and Procedure Model that has proven certain of its elements practical,

- and discussion about general application of the model.

4.1. The elements of the model have been proven practical

In order to gain experience with the Remote Pipe and Procedure 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 can be abstracted as follows:

```
server: group[establish-query: proc[string],
              count-matching-records: proc[] returns [int, bool],
              fetch-summaries: proc[Range, pipe[Summary]],
              fetch-record: proc[Range, pipe[Line]]]
```

When a user supplies a query the procedure **establish-query** is called. **establish-query** initiates processing of a query at a server and then returns immediately to Walter. The server procedure **fetch-summaries**, 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 **fetch-summaries** call. The pipe sink procedure that receives the summaries displays them as they arrive. All of the summaries generated by **fetch-summaries** are guaranteed to arrive before **fetch-summaries** returns. In order to view an entire database record the server procedure **fetch-record** is used in precisely the same manner as **fetch-summaries** is used.

A second process is conceptually running concurrently with the information which is arriving down a pipe and being displayed. This process checks for the abort user request, which aborts the query in progress. If such a keyboard request is received, a primitive is used to abort the **fetch-summaries** or **fetch-record** operation in progress.

The use of pipes in this database application has provided two distinct advantages over remote procedures. First, pipes permit both **fetch-summaries** and **fetch-record** 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.

4.2. The Remote Pipe and Procedure Model has many advantages

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 Remote Pipe and Procedure Model. We expect that this model will find a wide 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. Barbara Liskov, John Lucassen, Bob Scheifler, Mark Sheldon, Bill Weihl, and Heidi Wyle commented on drafts of the paper.

5. References

[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.

[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.

[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.

[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.

[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.

[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.

OFFICIAL DISTRIBUTION LIST

Director
Information Processing Techniques Office
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209

2 Copies

Office of Naval Research
800 North Quincy Street
Arlington, VA 22217
Attn: Dr. R. Grafton, Code 433

2 Copies

Director, Code 2627
Naval Research Laboratory
Washington, DC 20375

6 Copies

Defense Technical Information Center
Cameron Station
Alexandria, VA 22314

12 Copies

National Science Foundation
Office of Computing Activities
1800 G. Street, N.W.
Washington, DC 20550
Attn: Program Director

2 Copies

Dr. E.B. Royce, Code 38
Head, Research Department
Naval Weapons Center
China Lake, CA 93555

1 Copy

Dr. G. Hopper, USNR
NAVDAC-OOH
Department of the Navy
Washington, DC 20374

1 Copy

END

8-87

DTIC