ASTRA - An Asynchronous Remote Procedure Call Facility

A.L. Ananda, B.H. Tay
Department of Information Systems and Computer Science
National University of Singapore
Kent Ridge Crescent
Singapore 0511

E.K. Koh
Information Technology Institute
71 Science Park Drive
NCB Building
Singapore 0511

Abstract

Remote Procedure Call (RPC) is a popular paradigm for interprocess communication (IPC) between processes in distributed systems. It is simple, flexible and powerful. However, most of the RPC mechanisms today are synchronous in nature, and hence fail to exploit fully the parallelism inherent in distributed applications. We have designed and implemented a transport independent asynchronous RPC mechanism (ASTRA) that combines the advantages of both RPC and the message-passing IPC. ASTRA calls do not block the caller (client) and the replies can be received as and when they are needed, thus allowing the client execution to proceed locally in parallel with the server invocation. All the calls are received and executed by the server in the order called by the client. ASTRA is unique among other asynchronous RPC systems in allowing its users to explicitly specify whether low-latency or high-throughput is required for a call, and in providing highly optimized light-weight intra-machine calls. ASTRA is built within the framework of the SHILPA Distributed Computing Environment. This paper describes the motivations and the system architecture of ASTRA in the context of SHILPA, and its design and implementation.

1. Introduction

Remote Procedure Call (RPC) is a simple, flexible and powerful IPC paradigm for developing distributed applications [Wilbur and Bacarisse 87]. It is a widely used communication mechanism in distributed systems and applications such as Amoeba distributed operating system [Mullender et al. 90], Sprite network operating system [Ousterhout et al. 88], and Andrew File System (AFS) [Satyanarayanan 90].

Many RPC systems have been built since Nelson's PhD thesis [Nelson 81]. Notable works include Cedar RPC [Birrell and Nelson 84], Sun RPC [Sun 88], NC/RPC [Dineen et al. 87], and HRPC [Bershad et al. 87]. A survey of some of these works can be found in [Tay and Ananda 90]. However, most of these RPC systems are synchronous in nature, and hence fail to exploit fully the parallelism inherent in distributed applications. This severely limits the kind of interactions the distributed application can have, resulting in lower performance. To achieve concurrency, the user has to resort to other means such as light-weight processes (threads) or the low level inter-machine message-passing IPC (send/receive). If the host operating system does not support thread as in the case of Unix, costly heavy-weight processes have to be used instead.

We have designed and implemented an asynchronous RPC mechanism (ASTRA) that addresses the above problem. ASTRA is built within the framework of SHILPA, a Distributed Computing Environment for the Department of Information Systems and Computer Science (DISCS) at the National University of Singapore (NUS). The departmental network currently consists of three Ethernet segments interconnected through bridges, with a router link to the campus network. The hosts in the network consist of a cluster of VAXes, an AT&T 3B4000, Sun and HP/Apollo workstations, and several Macs and PCs. The operating systems used in our environment include VAX/VMS, Unix System V, Sun OS, Domain OS, MacOS, PC-DOS and OS/2. The main design objective of SHILPA is to provide a generic distributed computing platform for building distributed applications on an interconnection of local area networks in a heterogeneous environment.

There are four distinct features in ASTRA:

- ASTRA calls are similar to any other RPC systems, except it is able to defer receipt of return replies. In addition, all the calls are received and executed by the server in the order called by the client. Therefore, it retains all the benefits that conventional synchronous RPC systems have to offer, and yet allow parallel execution of the client and the server. Every call invoked in ASTRA will have a unique rpc_xid returned to the client. The client can make use of this rpc_xid to claim the reply for that particular invocation at a later stage. If the reply is not available at that time, the client will be blocked. The client can choose to un-block the receive operation by specifying a NO_DELAY option.

- Existing synchronous RPC systems are designed for low-latency to improve the response time, whereas existing asynchronous RPC systems are mostly designed for high-throughput. ASTRA is structured such that either low-latency or high-throughput can be achieved. The user can specify explicitly whether low-latency or high-throughput is the main concern for an invocation, and the system will optimize the call accordingly. It differs from other asynchronous RPC systems such as Stream [Liskov et al. 88] and Future [Walker et al. 90] that are designed to achieve only one of them, but not both.

- ASTRA is transport independent in the sense that it does not rely on any particular communication protocol. Two types of transport services are supported for inter-machine calls: virtual circuit and reliable datagram. Transport protocols currently supported are: TCP/IP [Postel, J. ed. 81] and RDT/IP. RDT is a reliable datagram transport protocol that is built on top of UDP [Postel, J. 80].

- ASTRA provides highly optimized intra-machine calls. For an intra-machine call, ASTRA will bypass the data conversion and network communication, and directly uses the fastest native IPC mechanism provided by the local operating system.

This paper first focuses on the motivation and design considerations of ASTRA, and compares it with other related works. The usage of various call features, and some of the important implementation issues are also discussed. Section 2 presents the rationale and design
considerations for ASTRA, and considers some alternatives to the asynchronous RPC systems. A comparison between our work and other related work is made in Section 5. Section 4 outlines the architecture of SHILPA and its relation to ASTRA. An overview of ASTRA in the context of SHILPA is presented in Section 5. Section 6 discusses some of the implementation issues such as binding, call invocation, failure semantics and RDTP.

2. Background

The design of ASTRA is motivated mainly by the need to achieve high-parallelism while retaining the simplicity and familiarity of the RPC abstraction. Limited degree of parallelism can be achieved by creating multiple light-weight processes (threads) for each RPC call [Bal et al. 87]. This allows the client to make multiple calls to many servers, and still be able to execute in parallel with the server. The program structure is similar to the fork/join, but is unwieldly and hard to debug. Although bundling RPC with threads incurs less overhead, this solution does not scale well. In a large distributed environment where the number of RPC calls grows and shrinks dynamically, using threads is not economical because of the cumulative cost of thread creation, context switching and thread destruction. Moreover, threads are not universally supported in our environment.

On the other hand, in Multi-RPC [Satyanarayanan and Siegel 86] [Satyanarayanan and Siegel 90] a client is allowed to invoke a procedure on many servers concurrently. The client is blocked until all responses are received, or the call is explicitly terminated by the client. While some parallelism is achieved, it is not possible for a client to invoke two different procedures in parallel. Thus Multi-RPC does not fully exploit parallelism in many situations.

Alternatively, through the use of message-passing inter-machine IPC mechanism one could achieve the desired parallelism. However, the users of such a system have to handle many details which were previously hidden by RPC, including data representation, and the pairing up of responses with request messages. ASTRA provides an intermediate abstraction between message passing IPC and synchronous RPC. It combines the advantages of both normal RPC and message-passing IPC by providing high-parallelism while retaining the simplicity and familiarity of the RPC abstraction.

Having selected asynchronous RPC as the underlying distributed communication mechanism in SHILPA, we have several considerations in the design of ASTRA. Firstly, ASTRA is designed to be transport independent to suit different types of application needs. Generally, clients and servers are involved in two kinds of interactions, intermittent exchange and extended exchange. By intermittent exchange we mean the client makes a few intermittent request-response (RR) type calls to the server. By extended exchange we mean the client is either involved in bulk data transfer, or makes many RR type calls to a particular server. In view of these different application needs, we designed ASTRA to be transport independent by incorporating both virtual-circuit (TCP) and datagram (RDTP) transport protocols to allow the application to choose the best transport that meets its needs. To achieve optimum performance, TCP could be selected for extended exchange since it provides better flow and error control with negligible processing overhead. On the other hand, RDTP is more suitable for intermittent exchange due to its simplicity.

Secondly, an asynchronous RPC facility must be optimized for inter-machine calls. According to a survey conducted by Bershad et al. [Bershad et al. 89], less than 10% of the remote activities are cross-machine calls. This is because most of the applications are designed to maximize local processing. In view of this, ASTRA is designed to optimize intra-machine calls by bypassing the data conversion and network communication operations.

Lastly, it must be possible to mix low-latency calls with high-throughput calls. ASTRA allows its user to explicitly specify the optimization needed.

3. Related Works

Asynchronous RPC calls can be classified into two types depending on whether the call returns a value. Most asynchronous RPC systems only support calls that do not return a value, and few support both classes. In this section, we examine a number of asynchronous RPC mechanisms, and highlight their similarities and differences in comparison to ASTRA.

Examples of asynchronous RPCs that do not return values include Non-blocking RPC of project Athena [Souza and Miller 86], MCS May-be RPC [Zahn et al. 90], Sun Batching RPC [Sun 88], and Remote Pipe [Gifford and Glasser 88]. Like ASTRA, they are all designed to increase the degree of parallelism in distributed applications. However, the first two systems do not guarantee the delivery of the call messages. If any communication reliability is desired, the end-application has to implement its own end-to-end mechanism. This is in contrast to ASTRA where all the messages are reliably delivered. On the other hand, in Batching RPC, calls are buffered and transported over a reliable byte-stream transport protocol such as TCP. In this case, a normal blocking RPC is needed in order to flush out the call buffer. Remote Pipe was designed to allow bulk data and incremental results to be efficiently transported in a type-safe manner.

Although all of the above provide some form of asynchronous RPC, none of them includes a mechanism to defer receipt of return results as in ASTRA. This shortcoming limits the design of distributed applications to strictly uni-directional exchange from client to server. There are three choices opened to the application programmer in these systems: 1) program the application using synchronous RPC call and sacrifice concurrency, 2) structure the application in such a way that no replies from servers is needed, 3) directly program on top of the transport layer.

Probably the closest to our work are Stream [Liskov et al. 88] [Liskov and Shriral 88] and Future [Walker et al. 90]. Both of them were developed to provide asynchronous RPC call that returns value, and have the ability to make multiple calls and defer receipt of replies.

Stream in the MIT Mercury system is designed mainly to achieve high-throughput where calls are buffered and flushed when convenient. Low-latency can be achieved by explicitly flushing the calls. This is however somewhat inconvenient. Moreover, Stream relies solely on a specific reliable byte-stream transport such as TCP, making it more suitable for bulk data transfer. The use of TCP leads to higher overheads for most transactional applications where a request-response protocol is more appropriate.

On the other hand, Future in the CRONUS system is designed only for low-latency. However, the order of execution in Future may vary with the order called since Future makes no guarantees concerning order of delivery. In addition, both Stream and Future do not optimize for intra-machine calls.

ASTRA combines the good features of both Stream and Future. A user in ASTRA is able to select between minimizing latency or maximizing throughput for each asynchronous RPC invocation. Moreover, ASTRA is transport independent and orders the sequence of the delivery of call and reply messages. We have also extended the idea of transport independence from inter-machine calls to intra-machine calls by integrating the most efficient native IPC transport mechanism into TCP.
The comparison of ASTRA and other related works is shown in Table 1.

<table>
<thead>
<tr>
<th>Transport Protocol</th>
<th>Remote</th>
<th>Stream</th>
<th>Future</th>
<th>ASTRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDP</td>
<td>TCP</td>
<td>Byte/stream</td>
<td>TCP</td>
<td>UDP,RDT (UDP)</td>
</tr>
<tr>
<td>Del/Syst. of Reply</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No (TCP)</td>
</tr>
<tr>
<td>Call Semantics</td>
<td>May</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Reliable Delivery of Message</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (TCP)</td>
</tr>
<tr>
<td>Ordered Delivery of Message</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Low Latency</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>High Throughput</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Suitable for RR transaction</td>
<td>No</td>
<td>No</td>
<td>Yes (high overhead because of TCP)</td>
<td>Yes (high overhead - TCP)</td>
</tr>
<tr>
<td>Lightweight Intra-machine Call</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 1: Comparison between ASTRA and other Asynchronous RPC Systems

5. Overview of ASTRA

5.1 System Architecture

ASTRA consists of the following components:

- **ASTRA L**: An interface description language (IDL) for defining the remote procedure interfaces. It is used as input to ASTRA_C for generating both RPC client and server stubs.

- **ASTRA_C**: A pre-compiler that takes ASTRA_L and generates the client and server stubs. This tool relieves the pain of coding the stubs manually.

- **ASTRA run-time library**: This is the library that handles all the communication details and exception notification.
In ASTRA, the RPC run-time library interfaces with BMP to achieve transport independence.

- **ASTRA superserver daemon**: It is one of the most important components in the ASTRA architecture. In ASTRA, the superserver must reside on every server machine. It acts as a local binding agent assisting in the dynamic association between client and server once the host is determined explicitly. Another function provided by the superserver is to manage and monitor the servers and auto-activate a server upon a binding request.

In ASTRA, we use client (caller), server (callee), services (called procedures) to denote the sender and receiver relationship. A server may provide many services; for example, a file server may provide open, close, read, and write services. We don’t restrict an application programmer to how a server can be implemented. An easy implementation is to program a server as a single program with services as procedures in the program. On the other hand, a single program can also be written to export many different types of services, with each server offering many services.

A server_ID identifies a server. Every server has a server_ID which is unique in the distributed system. Conflict in server_ID will result in contacting the wrong server. If there are two servers with the same ID, both of them must be a replication of each other, or provide identical functions on different hosts. The services provided within a server are identified via the service_IDs. Each service must have a unique ID within a server, and they must be identical for all the replication of that particular server.

ASTRA provides ordered and exactly once delivery to the server in the absence of site and hard network failure (i.e. network partition or down). The semantics of ASTRA call is exactly-once semantics\(^3\) in the absence of site and hard network failure, and at-most-once semantics otherwise. Section 6.3 contains a detail description of failure semantics of ASTRA.

### 5.2 Call Primitives

A short preview of some of the important primitives provided by ASTRA together with their usage is given in this section. Each primitive is a basic function provided by the run-time library. The user would not interface with these functions directly if they choose to use the stub generator. Figure 2 presents an overview of an ASTRA call with its underlying transport mechanisms.

![Figure 2. The Overview of ASTRA Calls](image)

Before a client can make any call, it must create a clnthandle and import the server interface. This can be done by the primitive:

```c
cln_handle = rpc_clninit(host, transport, server)
```

A clnthandle represents a binding between a client and a server. A client can create many clnhandles in order to contact and bind with many different servers. If the server host address happens to be the same as the local address, all the calls made using this clnthandle are optimized for intra-machine calls. In this case, the specification of the transport protocol is ignored. Currently, if the host is not specified, the creation of clnthandle will fail and an error is returned. However, this function can be enhanced as discussed in section 6.1 along with the binding process.

The client can make the ASTRA call to the server using the following primitives:

```c
rpc_xid = rpc_clntcall(clnthandle, service, call_option, INPUT arguments...)
```

Each rpc_clntcall() call returns a monotonically increasing rpc_xid in the case of a successful call or a -1 in the case of an error. Each rpc_xid is unique within a clnthandle and is used for claiming the reply message for a particular call at a later stage. The call_option parameter can be used to specify various options such as low-latency, high-throughput, and the real time limit. The real time limit is used for indicating the absolute time limit allowed for an execution in a server. If the execution exceeds the time limit, the operation is aborted and an error will be sent to inform the client. A clnthandle can be used for making several ASTRA calls, and all these calls are destined to a particular server specified in the rpc_clninit().

To receive a reply for a particular call or replies for a clnthandle, the client can use the following primitives:

```c
error_code = rpc_clntclaim(clnthandle, rpc_xid, delay_option, OUT arguments...)
```

The rpc_clntclaim() returns 0 in the case of a successful call or a -1 in the case of error. This function will be blocked if the reply message for a previous particular call is not available yet. If the delay_option is set to NO_DELAY, this function will return immediately if the reply message is not available. Further details about the implementation of claim operation are discussed in section 6.2.

A client can wait until there is a reply for any call made in a clnthandle, or until the time limit expires by invoking the following primitive:

```c
rpc_xid = rpc_clntwait(clnthandle, time_limit)
```

If the time_limit is set to -1, the function will block until there is a reply available for the clnthandle. This primitive is analogous to the select() function in the BSD socket.

Several primitives are provided for handling the abnormal conditions:

```c
rpc_clntping(clnthandle)
rpc_clntenq(clnthandle, rpc_xid)
rpc_clntabort(clnthandle, rpc_xid)
```

The rpc_clntping() is analogous to the Internet utility ping. The rpc_clntenq() is used to determine the status of the server process. The rpc_clntabort() is to re-try a particular call if some fatal errors have occurred. If the operation re-tries has been executed earlier, its result will be returned without re-executing the same operation. It is useful when a client is uncertain of the execution status of a server, but would like to re-try the non-idempotent operation again without

\(^3\) Generally, there is no agreed definition on the semantics of RPC [Wilbur and Bacarisse 87]. We follow the semantics defined in Spector’s paper [Spector 82] closely, except we term Only-Once-Type-1 as at-most-once and Only-Once-Type-2 as exactly-once.
invoking the same operation another time if it has already been executed. The `rpc_clntabort()` is to abort a call. The call will be aborted only if the call message is in the execution-queue or if it is being executed by the server. Once a call message has completed execution, it can never be aborted. The `rpc_clntreply()`, and `rpc_clntping()` will be further discussed in section 6.3 along with failure semantics, and the implementation of `rpc_clntabort()` will be described in section 6.4.

A simple usage of Astra calls is illustrated in the following C program segment:

```c
/* initialization of Astra call */
clnhand1 = rpc_clntinit(host1, transport1, server1);
clnhand2 = rpc_clntinit(host2, transport2, server2);

/* start all the Astra calls */
rpc_xid1_1 = rpc_clntasynccall(clnhand1, service1,
call-option1_1, INPUT arguments...);
rpc_xid2_1 = rpc_clntasynccall(clnhand2, service1,
call-option2_1, INPUT arguments...);

.....

/* do some local computation here */

/* start claiming the replies for all the Astra calls */
error_code = rpc_clntclaim(clnhand1, rpc_xid1_1,
call-option2_1, INPUT arguments...);
error_code = rpc_clntclaim(clnhand2, rpc_xid2_1,
delay_option2, OUT arguments...);

6. Implementation of Astra

6.1 Binding

The binding process of Astra is very similar to other conventional RPC systems, particularly Sun RPC and its `portmapper` [Sun 87]. The binding process is as follows: before a server can provide services, it must export its existence (process ID) and location (port number) to the superserver. Every server ID is associated with a unique port number. The superserver is responsible for keeping the server ID, its port number and its process ID. The users of Astra must specify the server's host explicitly, since the superserver only acts as a local binding agent. Once the host is known, the appropriate superserver could be queried for the server port number. This completes the binding process, and the client can now call any service just like a local procedure call.

The superserver is implemented using Astra. Both client and server make the import and export requests respectively to the superserver as RPC calls. Upon receiving an export request from a server, the superserver will check whether there is a conflict in server ID in the local host. If there is a conflict, the superserver will only replace the port number of the old server ID with the new port number if the process IDs are the same. This implies that the same server has changed its port number and is re-registering it. If the process ID of the current exporting server is different from the one kept by the superserver, the export request is refused and an error is returned. This is to prevent an unauthorized server from overwriting an existing server. On the other hand, when there is an import request, the superserver will detect whether a particular server is still alive by sending a NULL signal to it. If the server is still alive, then the superserver will return the server's port number to the client; otherwise an error is returned. The superserver can be considered as an improved and enhanced version of portmapper. The portmapper does not detect server activity, and simply replaces the location information without verifying the ownership when a new export request is received [Tay et al. 90].

There is no global binding agent in Astra in contrast to HP/Apollo NCS/RPC [Dineen et al. 87] [Zahn et al. 90], but Astra can easily be extended to provide a similar feature. This can be achieved by building a global binding agent on top of the superserver which collects all the server information and directs the binding request to the appropriate local superserver. Use of broadcast or well-known address can locate the global binding agent.

Alternatively, in the absence of global binding agent, a client can still locate a server through broadcast. A superserver will reply to the broadcast if it has the particular server registered in it. If there are many replicate servers, the first superserver who replies is probably the least-loaded server and/or it is the nearest to the client. This server will be selected to bind with the client to localize the processing and resources for achieving better performance.

6.2 Call Invocation

The call invocation of Astra is very similar to other RPC systems. However, there are several issues pertaining to the asynchronous nature of Astra. These issues include optimization for high-throughput or low-latency, delivery order of call and reply messages, support for receiving reply messages, and intra-machine calls.

In an Astra call, the user can specify whether low-latency or high-throughput is the main concern. When the high-throughput option is selected, the call message is buffered in the clnhand1's call buffer pool and the control is returned to the user. Subsequently, the user can make more calls. The call buffer pool is flushed when it is full. Buffering the call messages minimizes the system call overheads and therefore improves throughput. On the other hand, if the low-latency option is specified for an invocation, Astra will send its call message immediately after flushing out the call buffer pool. This is necessary to ensure that all the call messages are sent in the order called by its user. Therefore the previously buffered high-throughput calls can be implicitly flushed by a low-latency call.

TCP and RDTP are used in Astra to provide ordered delivery of call and reply messages. In our implementation, a server is always associated with a transport port. If any sub-servers are created by the server program, each of these will be associated with a transport port. For each binding between a client and a server, the delivery order is preserved via underlying transport. If there are multiple clients calling the same server, all the clients' requests to this server are queued sequentially in the server's underlying transport port. The delivery order is preserved with respect to a binding, but is not guaranteed across different bindings.

Replies are claimed explicitly by the client in any order. However, the reply may or may not have arrived when the client issues the claim operation. To further complicate the issue, the order of the arrival is not likely to be the same as the calling order. In our implementation, the system will reserve a result buffer for each call made. The reply messages are queued internally in the BMP transport layer before any claim operations. When a client wants to claim a reply message, Astra will return the reply message if it is available in the result buffer; otherwise Astra will read and scan the reply messages from the transport queue. In the process of scanning, all reply messages are copied into their respective result buffers until the reply message that is to be claimed is read, or the transport queue is empty. The copying is necessary because socket's read operation is destructive and it can only peek on the first message in the transport queue. If the transport queue is empty, the claim operation is blocked and the client is put to sleep until there are more incoming reply messages. The client can avoid blocking by specifying a NO DELAY option. Once the message is available in the result buffer, the unmarshaling of the arguments takes place as in normal synchronous RPC systems.
The optimization of intra-machine calls is done automatically by the system at bind time. The system checks the client and server addresses on binding. If the server's host name/address specified in import process is the same as the local address, then all calls made by this client handle are intra-machine calls and optimized accordingly.

The optimization is done by replacing the default set of routines (for inter-machine call) with the equivalent new set of light-weight routines. This new set of light-weight routines is specially tailored for intra-machine call. It does not contain the expensive marshalling and un-marshalling operations, and it uses the most efficient IPC mechanism provided by the local host operating system for passing the call and reply messages. For Unix System V and its variants [AT&T 86], the IPC is done through two shared memory segments with semaphore control for concurrent access. One shared memory segment is for passing the call messages and the other one is for the reply messages. For the sake of simplicity, we did not combine these two segments. If shared memory is not available, domain socket is used.

6.3 Failure Semantics

When a fatal error occurs, the reliable transport mechanisms such as TCP or RDTP will return an error status to ASTRA after several retries. ASTRA will not call the transport services again, instead it will abort the call and return an error to its user. In this case, the client is responsible for its own error recovery.

There are three kinds of failures that we will discuss here: network failures, client site failures and the server site failures.

Network failures: Network failures include network down or network partition. In either case, the client and server are not able to send or receive messages. If the network failure occurs when the client is sending out the request message using the BMP services, it will get an error returned from TCP or RDTP indicating the network is down. The TCP or RDTP in the BMP layer will return an error status only after several re-tries. If the client is willing to try again, it can use rpc_clntretry() to re-try the call without risking a re-execution of its previous request. Alternatively, it can use rpc_clntping() to detect whether the error is due to server process crash or network/host failure. However, if the client is waiting for a reply message in a claim operation in rpc_clntclaim(), it will be blocked until the reply message is received. The server will keep the reply message in the stable storage if there is fatal error returned from RDTP or TCP when it tries to return a reply message.

Client site failures: Client site failures include client process or the client host crash. The client site failure is detected by the ASTRA run-time system (server side) only when it tries to send a reply message using BMP services. The system on the server cannot differentiate between client site failure and network failure since both TCP or RDTP cannot set them apart. In this case, the server will keep the reply message in the stable storage. It is needed to ensure exactly-once semantics. If the client retries the call using rpc_clntretry() after the network or host system is restored to normal condition, the result will be retrieved from stable storage and returned without another execution. In ASTRA, orphans are detected only after they have completed execution and thus are not killed.

Server site failures: Server site failures include server process or server host crashes. The server process crashes can be detected by the client using rpc_clntping(). In the event of the server host crash, the rpc_clntping() will return an error indicating either the server host or the network is down. In this case, the user can use rpc_clntretry() at a later stage to re-try the call.

In ASTRA, the call and reply messages are reliably delivered. Unlike most other RPC schemes [Birrell and Nelson 84] [Souza and Miller 86], there is no keep-alive messages before and within the execution of the call message. If a server (site or process) crashes after the call message has reached the server and before it can be executed (i.e. it is queued), the client will not have the knowledge of the server status and may wait forever for the reply if the NO_DELAY option is not set in rpc_clntclaim() or the time_limit is set to -1 in rpc_clntwait(). In our actual implementation, we try to trap all the exceptions raised in the client and server computation by capturing all catchable signals, and pass the error status back to the respective client or server. However, there are certain abnormal conditions which can not be captured such as the SIGKILL signal in Unix, or a sudden power failure condition in a system without an Uninterrupted Power Supply (UPS) backup. These may let the client wait forever for claiming the result from a non-existent computation. In this case, it is always nice to use rpc_clntwait() with time_limit set to some positive value before any rpc_clntclaim() operation to prevent the deadlock.

The rationale for omitting keep-alive message is as follow: keep-alive message is similar to the linger option provided by the TCP/IP. It incurs higher overheads and little usage if the message is reliably delivered. Keep-alive messages cannot differentiate a site and network failure. In this case, using keep-alive messages may be misleading if an error occurs due to network failure. Moreover, if there is site or network failure, the subsequent calls made by a client will return an error. In this case, we do not really need keep-alive message to detect the status of a server. Furthermore, according to the implementation experience of RPC in Unix, two threads of control and transport ports are required for each client and server in order to acknowledge the keep-alive messages [Souza and Miller 86]. This particular implementation requirement is not acceptable because most of our host operating systems are Unix which do not support threads. Thus we decided to choose a compromise approach: omitting the keep-alive message to minimize the overheads but allowing a time_limit for rpc_clntwait() and NO_DELAY for rpc_clntclaim(), and provide a function rpc_clntping() to detect the status of a server process.

6.4 RDTP

As shown in table 1, most of the asynchronous RPC systems only depend on TCP to provide reliable transport, and ignore the need for a connectionless-oriented reliable transport mechanism. However, TCP is expensive to use for low-latency call, and as such it is not suitable for request-response type of communication.

The development of RDTP was motivated by the need to provide a low-cost and reliable transport for ASTRA. RDTP is a reliable datagram transport protocol which is built on top of UDP [Postel, J. ed. 811 and SPP in XNS [Xerox 85]]. With RDTP, the reliable delivery of call and reply messages are guaranteed in a connectionless and orderly manner. It is particularly useful for transporting low-latency calls. For simplicity and efficiency reasons, RDTP employs a stop-and-wait protocol with flow and error control. Each RDTP packet contains a unique and increasing sequence number. It is used by the receiving end to detect and discard duplicate or long-delay packets. The maximum datagram size a user can send at one time is 2 Kbytes. One way to improve the performance of RDTP is to use a sliding window mechanism. However, the benefits would probably be insignificant for short messages on a local area network [Lee et al. 86]. In addition, the cost of increased complexity may not justify the benefit gained.

RDTP is different from other reliable datagrams in providing an urgent datagram option. The urgent datagram is similar to the out-of-band data provided by most of the connection-oriented transport such as TCP in TCP/IP protocol suite [Postel, J. ed. 811] and SPP in XNS [Xerox 85]. The maximum size of the urgent datagram is equal to the normal datagram size. Once the user of RDTP specifies the urgent datagram option in sending, RDTP will push the packet out to the
network immediately. On the receiving end, the urgent datagram option is used to send a signal to a server. The user can receive the urgent datagram by specifying the urgent datagram option in receiving. The user can also set to capture the signal and issue the receive operation in order to receive the urgent datagram immediately once it arrives. The urgent datagram option is primarily used by STRA for implementing the routine rpc_closeof() to abort a call sent to a server.

Note that it is possible to implement the end-to-end mechanism within STRA itself without RDTP. This is the approach taken by many other RPC systems such as Athena Project RPC [Souza and Miller 86] and NCS/RPC [Dineen et al. 87]. We did not adopt this approach due to the increased complexity in the RPC protocol, and the need for a generic reliable datagram protocol which can be used by many other applications.

7. Conclusion

The primary contribution of this paper is the integration of both low-latency and high-throughput communication into a single asynchronous RPC model, and its ability to employ different transport mechanisms at bind-time. The transport mechanisms supported include virtual-circuit and datagram services for internode calls, and the most efficient IPC provided by the local host for intra-machine calls. We have shown that the necessity for an asynchronous RPC mechanism, and the importance of transport independence in achieving an efficient and flexible asynchronous RPC under different types of application requirements.

Although we have emphasized the importance of asynchronous RPC, we expect that the normal synchronous RPC calls to be predominant and therefore have included it in the remote operation layer (ROP) for simple one-to-one calls. It has the same interface as STRA. We have provided a complete and comprehensive remote operation mechanism that is powerful and flexible with an uniform access mechanism. We believe ROP (together with STRA) can support a wide range of distributed applications.

The current implementation platforms of ASTRA and other components of SHILPA are on an AT&T 3B4000, Sun SPARC IPC and 386 PC. All of them run Unix System V Release 3 or its variants, except the Sun workstation which runs SunOS 4.1. We have plans to port the entire SHILPA services (including STRA) to the VAX/VMS environment, and build a global binding agent on top of the current system.

Acknowledgements

The work described here is supported by the National University of Singapore Scholarship and Research Grant RP900608. We thank Barbara Liskov, William Weihl for sending us their papers, and Edward F. Walker for clarifying some of the details in his paper.

References


