# A Survey of Asynchronous Remote Procedure Calls

A.L. Ananda
B.H. Tay
Department of Information Systems and Computer Science
National University of Singapore
Kent Ridge Crescent
Singapore 0511
Republic of Singapore
BITNET: ananda@nusdiscs.bitnet, taybengh@nusdiscs.bitnet

and

E.K. Koh
Information Technology Institute
71 Science Park Drive
NCB Building
Singapore 0511
Republic of Singapore
BITNET: engkiat@itivax.bitnet

## *Abstract*

Remote Procedure Call (RPC) is a popular paradigm for interprocess communication in distributed systems. It is simple, flexible and powerful. However, most of the RPC systems today are synchronous in nature, and hence fail to exploit fully the parallelism inherent in distributed applications. In view of this, various asynchronous RPC systems have been designed and implemented to achieve higher parallelism while retaining the familiarity and simplicity of synchronous RPC. Asynchronous RPC 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 callee (server) invocation. Asynchronous RPC calls can be classified into two types depending on whether the calls return a value. Most asynchronous RPC systems only support calls that do not return a value, and few support both classes. In this paper, an analysis and comparison of various asynchronous RPC systems are presented.

**Keywords:**
**distributed systems, interprocess communication (IPC), remote procedure call (RPC), synchronous RPC, asynchronous RPC, parallelism, low-latency, high-throughput, transport-independent, intra-machine call.**

# 1. Introduction

Remote Procedure Call (RPC) is a simple, flexible and powerful interprocess communication (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 [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], NCA/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 mechanism (*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. However, both solutions are not attractive to the users. The first solution is unwieldy, hard to debug, and does not scale well in a large distributed environment. The second solution is much more difficult to use than the RPC mechanism. In view of this, various asynchronous RPC systems have been designed and implemented to achieve higher parallelism while retaining the familiarity and simplicity of synchronous RPC. Asynchronous RPC 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.

This paper is a comparative study of a few distinctive asynchronous RPC mechanisms. The asynchronous RPC mechanisms included in the discussion are *Athena Non-blocking RPC, NCA Maybe RPC, Sun Batching RPC, Remote Pipes, Stream (Promises), Future* and *ASTRA*. The comparison is based mainly on the following characteristics of the asynchronous RPC mechanisms:

- Support for the receipt of reply message
- Transport protocols
- Order of delivery of call and reply messages
- Call semantics
- Optimization for low-latency or high-throughput

• Optimization for intra-machine calls

Section 2 examines the motivation for the development of the asynchronous RPC mechanisms, and discusses its design criterion. A detailed analysis of each asynchronous RPC system is presented in section 3. Section 4 presents a scorecard for the asynchronous RPC systems discussed.

# 2. Background

The design of an asynchronous RPC mechanism 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 multiple servers, and still be able to execute in parallel with the servers. The program structure is similar to the *fork/join*, but is unwieldy 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 supported in some of the host operating systems such as Unix System V.

Some systems like Multi-RPC [Satyanarayanan and Siegel 90] attempt to achieve higher parallelism by allowing a client to invoke multiple instances of a procedure 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, one could achieve the desired parallelism through the use of the message-passing inter-machine IPC mechanism. However, the users of such a system have to handle many details which were previously hidden within RPC, including data representation, packaging of the messages, and the pairing of responses with request messages. Asynchronous RPC provides an intermediate abstraction between message passing IPC and synchronous RPC.

## 2.1 Design Criterion

There are several criterion which are desirable in the design of an asynchronous RPC system. Firstly, an asynchronous RPC system must have the look and feel of a synchronous RPC system, except that the client

does not wait for a reply after making an asynchronous call. In this case, the client may or may not be able to defer receipt of return replies. In addition, it is desirable that all the calls are received and executed by the server in the order called by the client to preserve the correct call semantics. Therefore an asynchronous RPC system should retain all the benefits that a conventional synchronous RPC system has to offer, and yet allow parallel execution of the client and the server.

Secondly, an asynchronous RPC system must be 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 occasional request-response (RR) type of calls to the server. By extended exchange we mean the client is either involved in bulk data transfer, or makes many RR type of calls to a particular server. An asynchronous RPC system should ideally incorporate both virtual-circuit and datagram transport protocols, so as to allow the application to choose the best transport that meets its needs. To achieve optimum performance, virtual-circuit could be selected for extended exchange since it provides better flow and error control with negligible processing overhead. On the other hand, datagram is more suitable for intermittent exchange due to its simplicity.

Thirdly, an asynchronous RPC facility must be optimized for intra-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, an asynchronous RPC system should be designed to optimize intra-machine calls by bypassing the expensive data conversion and network communication operations.

Existing synchronous RPC systems are designed for *low-latency* to improve the response time, whereas the asynchronous RPC systems are mostly designed for *high-throughput*. It is desirable to structure an asynchronous RPC system such that either low-latency or high-throughput can be achieved. In this case, the user is able to specify explicitly the optimization needed at run-time, and to mix low-latency calls with high-throughput calls.

# 3. Asynchronous RPC Systems

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. A taxonomy of the asynchronous RPC mechanism discussed in this paper is shown in figure 1.

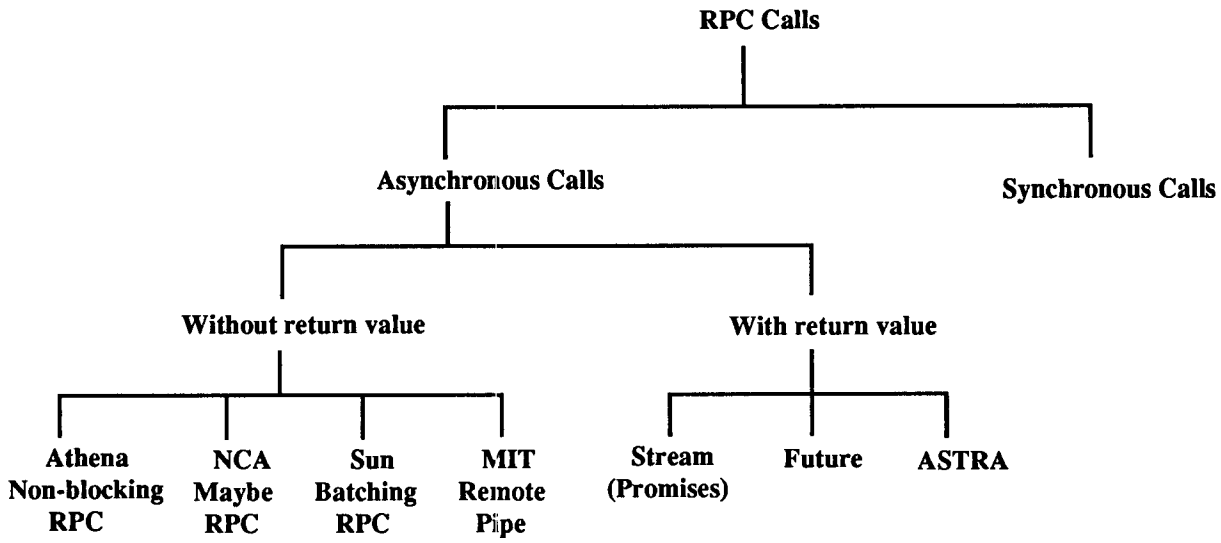


**Figure 1 A Taxonomy of RPC calls**

In this section, we examine a number of asynchronous RPC systems in the chronological order of the development, and highlight their similarities and differences. Section 3.1 describes the asynchronous RPC systems that do not return a value. We will discuss the class of asynchronous RPC systems that can defer receipt of replies in the section 3.2.

## 3.1 Asynchronous RPC without return value

### 3.1.1 MIT Project Athena Non-blocking RPC

The objective of MIT's project Athena [Champine et al. 90] was to integrate various computing and communication resources for educational purposes. Athena RPC [Souza and Miller 86] was developed under the constraints imposed by the coherence model[0] of the Project Athena. Some of the constraints included no modification to the Unix kernel, support of RPC in a heterogeneous environment, and support of multiple language suites. It was implemented as a prototype in the BSD4.2 Unix operating system.

Athena RPC provides both blocking (synchronous) and non-blocking (asynchronous) calls. Athena non-blocking RPC was developed primarily to improve the performance of applications where no information or status need to be returned from the called procedure.

---

[0]Coherence means that the same system interface is provided to all users regardless of the systems deployed from various vendors.

A request-response (UDP-RR) protocol is built on top of UDP for normal blocking RPC. Non-blocking RPC uses UDP as its transport mechanism. As a result, it does not guarantee the delivery of call messages, nor the order of delivery. Consequently, the order of execution in the server may not be the same as the order of call invocation by the client.

Athena non-blocking RPC has may-be call semantics, as defined in Spector's paper [Spector 85], since there is no guarantee of delivery of call messages. The applications have to implement their own end-to-end mechanism if any communication reliability is desired. The may-be semantics renders it unsuitable for any transactional applications.

To reduce latency, Athena non-blocking RPC sends out its call message immediately after each call. In addition, it does not differentiate between inter-machine and intra-machine calls, and hence no optimization is performed for local intra-machine calls.

### 3.1.2 NCA Maybe RPC

NCA/RPC [Zahn et al. 90] was developed by HP/Apollo as part of the Network Computing Architecture (NCA). It provides a rich set of RPC calls for the programmer: a normal blocking RPC which is termed *send-wait-reply*, an asynchronous RPC which is called *maybe RPC*, *broadcast RPC*, and *broadcast/maybe* RPC.

To specify a maybe RPC, the interface definition language (IDL) file must contain a **[maybe]** operation attribute in front of the procedure definition. This is illustrated in the following code segment written in the Network Interface Definition Language (C syntax):

```
[maybe] void simple$op(...);
```

Here *simple$op()* is the name of a remote procedure which does not return a value. The stub generator will produce a stub procedure *simple$op()* as specified in the IDL file. The client can call the the remote procedure *simple$op()* as a local procedure.

In addition to the above call types, NCA/RPC provides functions which are rarely available in other RPC implementations; for example, a client is able to send "ping" packets to inquire an outstanding request and "quit" packets to inform the server that it is to abort processing of the remote call.

NCA/RPC is designed to work on top of a connectionless-oriented transport layer. Currently, this layer supports the Apollo Domain Protocol (DDS) and UDP. NCA maybe RPC is very similar in characteristics

to Athena non-blocking RPC. Both rely on an unreliable datagram for transporting the call messages, and do not expect a reply from the servers. Therefore, it has may-be call semantics and the order of delivery of messages is not guaranteed.

NCA maybe RPC does not attempt to buffer the call messages; the call message is sent immediately to achieve low-latency. In addition, it does not optimize intra-machine calls since it does not distinguish between intra-machine and inter-machine calls.

### 3.1.3 Sun Batching RPC

Sun ONC/RPC [Sun 88] was developed by Sun Microsystems as part of the Open Network Computing (ONC). Sun batching RPC is one of the call types provided by Sun RPC; others are normal synchronous RPC and broadcast RPC. Batching RPC allows a series of calls to be made from the client to the server. Each RPC call in the pipeline requires no reply from the server, and the server can not send a reply message. The last call must be a normal blocking RPC in order to flush out the pipeline of calls.

Sun RPC provides two types of interface for application programmers. One is available as library routines. The other interface uses a RPC specification language (RPCL) and a stub generator (RPCGEN). The RPCL is an extension of the eXternal Data Representation (XDR) [Sun 87] specification. To use batching RPC, one can use RPCGEN or the library routines.

The *clnt_call()* library routine can be used to invoke Sun batching RPC as follows:

**clnt_call(client, PROCNUM, xdr_req, &request, xdr_ret, return, timeout);**

The call is distinguished from the normal blocking RPC by setting **timeout**, **xdr_ret** and **return** to zero (NULL). Here **client** is a handle returned by *clnt_create()* which creates and binds a RPC client handle; **PROCNUM** is the procedure number to be called in the server, and the **xdr_req** is the corresponding *xdr* routine for the data called **request**.

Sun RPC provides both UDP and TCP as its transport communication mechanism. However, batching RPC is built on top of TCP alone. In contrast to Athena non-blocking RPC and NCA maybe RPC, all the messages are reliably delivered. However, the fact that TCP is the only transport mechanism supported makes it un-suitable for request-response type of transactional applications. The call semantics of Sun batching RPC is at-most-once. This is an improvement over the may-be call semantics of Athena non-blocking RPC and NCA maybe RPC.

Sun batching RPC makes use of TCP to buffer call messages, and send them to the server in one Unix *write()* system call. This greatly decreases the system call overhead, thus improving performance and throughput. However, no optimization is done for intra-machine calls in Sun batching RPC.

### 3.1.4 Remote Pipe

Remote pipe [Gifford and Glasser 88] was designed to allow bulk data and incremental results to be efficiently transported in a type-safe manner. These objectives are realized using a communication model called the Channel Model. The Channel Model consists of three basic elements: remote procedure, remote pipe and channel groups.

In the Channel Model, a node[1] is similar to a process. A node may contain a number of channels. A channel, in this context, is either a remote procedure or a pipe. The difference between a remote procedure and a pipe is that the former is a synchronous RPC while the latter is an asynchronous RPC that does not return a value. A node can import any channel exported by any other node or possibly re-export them to other nodes. This makes the channel a first-class value[2] that can be freely exchanged among nodes [Nelson 81]. The importing node can group channels into a set called a channel group. A channel group controls the sequencing of the calls; data sent to a channel within a channel group are received in the order sent.

The remote interface definition can be specified using Modula-2 as follows:

```
DEFINITION MODULE xxx;
        PIPE pipeop(...);
        PROCEDURE rpcop(...);
        ...
        SEQUENCE pipeop, rpcop;
        ...
END xxx.
```

In this example, **xxx** is the name of the module, **pipeop** is the name of the remote pipe, and **rpcop** is the name of the remote procedure. The **SEQUENCE** statement statically groups **pipeop** and **rpcop** into a channel group. Alternatively, the programmer can group a set of channels using the *group()* primitive at run-time. As the definition module suggests, remote procedures or pipes can be called just like any local procedures as follows:

---

[1] A node is defined as a virtual computer with a private address space. A physical computer can have one or many nodes.

[2] A first-class object is a value (including remote procedures) that can be freely stored in memory, passed as a parameter to both local and remote procedures, and returned as a result from both local and remote procedures.

```
{ Import the xxx server }
yyy := xxxClient.Import(...);
{ Call the remote pipe and procedure }
yyy.pipeop(...);
yyy.rpcop(...);
```

Here yyy is an instance of the type xxx.

The Channel Model is transport independent; it can be implemented on top of raw datagrams by providing its own flow control, or on transport protocols that include flow control such as VMTP [Cheriton 86]. In fact, Gifford and Glasser implemented it on top of TCP.

The Channel Model can be implemented using either at-most-once or exactly-once semantics. However, it is believed that most of the implementation of the channel model will have at-most-once semantics due to its simplicity. This is similar to many other RPC systems such as Sun batching RPC.

The performance of the Channel Model can be optimized by buffering and combining pipe calls destined for the same sink node into a single message to reduce the message-handling overhead and hence improving the throughput. This is similar to Sun batching RPC. Other optimizations include combining pipe calls with procedure calls to flush out the buffered pipe calls, combining pipe returns to a single message to reduce message handling and system call overheads, preallocating processes in a process pool to eliminate *fork* overhead, and factoring packages and groups to save space. In the Channel Model, no attempt is made to differentiate between inter-machine and intra-machine calls. As a result, intra-machine calls are not optimized.

## 3.2 Asynchronous RPC with return value

Although the RPC systems discussed provide some form of asynchronism, none of them includes a mechanism to defer receipt of return results. 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 reply from servers is needed, 3) directly program on top of the transport layer. In view of these shortcomings, asynchronous RPC systems that can defer receipt of replies such as *stream (promises)*, *future* and *ASTRA* have been developed.

### 3.2.1 Stream (Promises)

*Stream* in the MIT Mercury system is the first RPC system that combines synchronous and asynchronous calls with return value in a clean and uniform way [Liskov et al. 88]. Stream provides three kind of calls: normal *synchronous RPC calls*, *stream calls* and *send*. Stream calls is a kind of asynchronous RPC call with reply message. Send, on the other hand, is similar to Sun Batching RPC and remote pipe calls, in that the client is not interested in the reply. In addition to the above three calls, stream provides a *"flush"* primitive that can be used to flush out the buffered call or reply messages, and a *"synch"* primitive that will block the caller until the processing of all the earlier calls have been completed.

A stream-based transport protocol such as TCP is used for transporting and sequencing the stream call and reply messages reliably. It simplifies the implementation of stream, and also provides at-most-once call semantics. However, the fact that stream relies solely on a specific reliable stream-based transport makes it more suitable for bulk data transfer rather than low-latency calls. Moreover, the use of TCP leads to higher overheads for most transactional applications where a request-response protocol is more appropriate.

Like Sun batching RPC and remote pipes, it was designed mainly to achieve high-throughput where call messages are buffered and flushed when convenient. This is to reduce the system call overhead. Although low-latency can also be achieved by explicitly flushing out the calls, it is however somewhat inconvenient. Again, no optimization is done for intra-machine calls.

Streams have been integrated into Argus [Liskov 88] as a new data type called *promises* [Liskov and Shrira 88]. A promise is returned to the caller whenever a stream call is performed. It is like a mailbox that holds the result computed by the server, and can be claimed by the client when it is ready. The result of the claim operation reflects the outcome of a stream call. The claim returns the type of the result if the call succeeds, and the names and types of the exceptions if the call fails. A promise can be claimed many times in any convenient order, and the same outcome will be returned each time. Although stream only provides at-most-once call semantics, promises is able to achieve exactly-once call semantics because Argus computations run as atomic transactions.

The declaration, stream call and claim operations for promises can be illustrated as follows:

```
% declaration - this is a comment statement
objtype = promise returns (type) signals (...)
% stream call operation
objtype$operation_name(x, ...)
% claim operation
y: type = objtype$claim(x)
```

```
except when ...
        end
```

Here **x** is a promise of type **objtype**, and **type** is the data type for the return result **y**. The control only goes to the **except** portion when the call terminates with an exception.

### 3.2.2 Future

*Future* [Walker et al. 90] is an asynchronous RPC provided in the CRONUS system [Schantz et al. 86]. A future is an object that is returned after each client invocation. It is an I.O.U. that can be used to claim the result of an invocation at a later stage. Futures are created and claimed by the stub procedures which are automatically generated from a specification of the remote operations.

For each remote operation that is invoked, a pair of stub procedures - *FInvokeXXX()* and *FClaimXXX()* are generated by the stub generator. *FInvokeXXX()* is used to invoke a remote operation and return a future. The calling format of *FInvokeXXX()* is shown as follow in the C language syntax:

```
FUTURE FInvokeXXX(object, Statement, InvokeControl)
```

In this example, **XXX** is the remote operation name exported by the server, **object** specifies the item to be operated on, and **statement** specifies the input data. **InvokeControl** is used to set various handling options for the invocation such as the hostname where a server resides and the absolute time limit for the reply to arrive. The system will signal an error if the absolute time limit is exceeded. On the other hand, *FClaimXXX()* is used to claim a future at a later time. The calling format of *FClaimXXX()* is shown below:

```
int FClaimXXX(future, output)
```

Here **XXX** is the operation name exported by the server, **future** is the unique identifier returned by *FInvokeXXX()*, and **output** is the output arguments returned by remote procedure **XXX**.

In addition, there are three other primitives provided to manipulate futures : *Discard()*, *IsReady()* and *AllowMultipleClaims()*. *Discard()* notifies the system that the future is no longer of interest and should therefore be destroyed. *IsReady()* tests if a future is ready for collection. *AllowMultipleClaims()* allows multiple replies to be claimed for a future. This is a unique primitive provided in future to support asynchronous call with multiple replies.

Future also provides an abstraction called *FutureSet*. This allows multiple futures to be grouped into a set. FutureSet facilitates the management of futures and eliminates the strict ordering of claim operations. For example, the primitive *FuturesetExtractReady()* extracts any one of the futures in a set which is ready. This

particular future retrieved can be claimed subsequently using *FClaimXXX()*. Thus *FuturesetExtractReady()* is analogous to the *select()* primitive in the BSD socket [Sechrest 86] .

In addition, future supports flow control for asynchronous calls through an abstraction called *Funnels*. A Funnel essentially specifies the maximum number of outstanding futures that is allowed at any one time. When a client calls *FInvokeXXX()*, the call will return to its caller immediately. However, the remote operation is invoked only if the number of outstanding futures has not exceeded the maximum. Otherwise the call message is held until the outstanding futures are either claimed or discarded. As a result, overload in server is prevented via funnels. However, in the current implementation, the flow control using funnels is handled at the application level, not at the system level.

Future was implemented on both TCP and UDP. TCP is the main transport protocol supported in future. With TCP, the delivery of the call and reply messages are guaranteed. On the other hand, future do not provide any end-to-end mechanism on top of UDP. Thus UDP-based future calls are not reliable nor dependable. Although TCP is a sequenced transport protocol, future makes no guarantees concerning the order of delivery of the call messages. The call messages may be reordered in the process of buffering before it is transmitted [Walker 90]. This is a serious drawback since the order of execution in the server may be different from the order called by the client.

Unlike most of the asynchronous RPC systems, future was designed mainly for low-latency. The call message is sent immediately for each request made, and the returned results can be claimed in any order. In the current implementation, future does not bypasses the expensive data conversion and network communication for intra-machine calls [Walker 90].

### 3.2.3 ASTRA

ASTRA [Ananda et al. 91] 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 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.

ASTRA calls are similar to stream and future calls in that it is able to defer receipt of results. The client can make the an ASTRA call in the C language using the following primitives:

**RPCXID rpc_clntasycall(clnthandler, service, call_option, ...)**

The **clnthandler** is a handle returned by *rpc_clntinit()* which creates and binds a client handle; **service** is the service name/number to be called, and the **call_option** parameter can be used to specify various options such as low-latency or high-throughput. Each *rpc_clntasycall()* call returns a monotonically increasing *rpcxid* in the case of a successful call or a -1 in the case of an error. Each *rpcxid* is unique within a *clnthandler* and is used for claiming the reply message for a particular invocation at a later stage. To receive a reply for a particular call for a *clnthandler*, the client can use the following primitive:

**int rpc_clntclaim(clnthandler, rpcxid, delay_option, ...)**

Unless the **delay_option** is set to NO_DELAY, this function will be blocked if the reply message for this particular invocation is not available.

In addition to the normal call and claim primitives, ASTRA provides a primitive *rpc_clntwait()* that allows a client to wait for a reply up to a specified time limit. Several other primitives are also provided for handling the abnormal conditions, including *rpc_clntping()*, *rpc_clntretry()*, and *rpc_clntabort()*. The *rpc_clntping()* primitive is used to determine the status of the server process. The *rpc_clntretry()* primitive is used to re-try a particular call without re-executing the operation if it has been performed earlier, and *rpc_clntabort()* aborts a call that is pending or executing in the server.

ASTRA is transport independent in 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 and RDTP/IP. RDTP is a reliable datagram transport protocol that is built on top of UDP. ASTRA sequences the delivery of call and reply messages regardless of the underlying transport protocols. Thus all the calls are received and executed by the server in the order called by the client. Moreover, ASTRA is designed to achieve at-most-once call semantics.

ASTRA integrates both low-latency and high-throughput communication into one single asynchronous RPC model. 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 and future that are designed to achieve only one of them, but not both.

Unlike stream and future, ASTRA provides highly optimized (light-weight) 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 is a unique feature provided by ASTRA. However, ASTRA does not incorporate concepts like FutureSet and Funnel. The flow control in ASTRA is done by the underlying transport protocol.

# 4. Asynchronous Remote Procedure Calls Scorecard

The following table is a scorecard of the asynchronous RPC systems discussed. The call semantics defined here follow the definitions in Spector's paper [Spector 82] closely, except we denote Only-Once-Type-1 as at-most-once.

| | *Athena RPC* | *NCA/RPC* | *Sun Batching RPC* | *Remote Pipes (Channel Model)* | *Stream (Promises)* | *Future* | *ASTRA* |
|---|---|---|---|---|---|---|---|
| **Distributed Computing Environment** | MIT Athena | HP/Apollo NCS | Sun ONC | Mercury | Mercury | CRONUS | SHILPA |
| **Transport Protocol** | UDP | UDP | TCP | Datagram or Stream based (TCP) | TCP | TCP UDP | TCP RDTP (UDP) |
| **Defer Receipt of Reply** | No | No | No | No | Yes | Yes (TCP) No (UDP) | Yes |
| **Call Semantics** | may-be | may-be | at-most-once | at-most-once | at-most-once | at-most-once | at-most-once |
| **Reliable Delivery of Message** | No | No | Yes | Yes | Yes | Yes (TCP) No (UDP) | Yes |
| **Ordered Delivery of Message** | No | No | Yes | Yes | Yes | No | Yes |
| **Low Latency** | Yes | Yes | No | No | Yes | Yes | Yes |
| **High Throughput** | No | No | Yes | Yes | Yes | No | Yes |
| **Suitable for RR transaction application** | No | No | No | No | Yes (high overhead because of TCP) | Yes (high overhead - TCP) No (UDP) | Yes |
| **Light-weight Intra-machine Call** | No | No | No | No | No | No | Yes |

**Table 1 A comparison of the various Asynchronous RPC Systems**

# 5. Conclusion

This paper briefly examined why asynchronous RPC is the most suitable paradigm to achieve higher parallelism in a heterogeneous distributed computing environment. It also discussed the design criterion for

such a mechanism. Lastly, a number of asynchronous RPC systems developed in recent years were analyzed and compared.

This survey revealed a few salient points. Firstly, the present trend is towards the development of asynchronous RPC systems with return values. This is evident from the fact that the latest asynchronous RPC developments such as stream, future and ASTRA are all in this category. Secondly, most of the systems surveyed here did not optimize for intra-machine calls. A probable reason is that the dominance of intra-machine calls is not known until recently when Bershad et al reported their findings [Bershad et al. 89]. Lastly, virtual-circuit (TCP) is a popular transport mechanism for the asynchronous RPC system, since it conveniently provides the reliability and ordering of the asynchronous RPC calls.

Although the importance of asynchronous RPC is highlighted, we expect that normal synchronous RPC calls will dominate. That almost all the asynchronous RPC systems come with a synchronous counterpart is a testimonial to our belief. We are confident that future RPC implementations will integrate both synchronous and asynchronous calls together in order to provide a uniform, complete, and comprehensive remote operation mechanism. Only then can distributed applications be universally and easily supported.

## Acknowledgements

The work described here is supported by the National University of Singapore's Research 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 points in his paper.

# References


[Ananda et al. 91] A.L. Ananda, B.H. Tay, and E.K. Koh, "ASTRA - An Asynchronous Remote Procedure Call Facility", *Proc of the 11th International Conference on Distributed Computing Systems (ICDCS-11)*, IEEE, Arlington, Texas, United States, May 20-24, 1991, pp.172-180.

[Bal et al. 87] Bal, H.E., Renesse, R. van, and Tanenbaum, A.S., "Implementing Distributed Algorithms using Remote Procedure Call", *Proc. National Computer Conference*, AFIPS, pp. 499-505, 1987.

[Bershad et al. 87] Bershad B.N., Ching D.T., Lazowska E.D., Sanislo J., Schwartz M., "A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems", *IEEE Trans. on Software Eng.*, Vol. 13, No. 8, Aug 1987, pp. 880-894.

[Bershad et al. 89] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy, "Lightweight Remote Procedure Call", *Proc. of 12th Symp. on Operating Systems Principles*, Dec 1989, pp. 102-113.

[Birrell and Nelson 84] Birrell A.D. and Nelson B.J., "Implementing Remote Procedure Calls", *ACM Trans. on Computer Systems*, Vol. 2, No. 1, Feb 1984, pp. 39-59.

[Champine et al. 90] George A. Champine, Daniel E. Geer, Jr., and William N. Ruh, "Project Athena as a Distributed Computer System", *Computer*, Sep 1990, pp. 40-51.

[Cheriton 86] Cheriton, D.R., "VMTP: A Transport Protocol for the Next Generation of Communication System", *Proc. of SIGCOMM'86*, Aug 1986, pp. 406-415.

[Dineen et al. 87] Dineen, T. H., Leach, P.J., Mishkin, N.W., Pato, J.N., and Wyant, G.L., "The Network Computing Architecture and System: An Environment for Developing Distributed Applications", In *Proc. of the USENIX Conference (Phoenix, Ariz., June). USENIX Association*, Berkeley, Calif., 1987, pp. 385-398.

[Gifford and Glasser 88] David K. Gifford and Nathan Glasser, "Remote Pipes and Procedures for Efficient Distributed Communication", *ACM Trans. on Computer Syst.*, Vol. 6, No. 3, Aug 1988, pp. 258-283.

[Liskov 88] B. Liskov, "Distributed Programming in Argus", *Comm. of the ACM*, Vol. 31, No. 3, Mar 1988, pp. 300-312.

[Liskov et al. 88] B. Liskov, T. Bloom, D. Gifford, R. Scheifler, and W.E. Weihl, "Communication in the Mercury System", in *Proc. 21st Annu. Hawaii Int. Conf. Syst. Sc.*, Jan 1988.

[Liskov and Shrira 88] B. Liskov and Shrira, "Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems", in *Proc. of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 22-24, 1988, pp. 260-267.

[Mullender et al. 90] Sape J. Mullender, and Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren, "Amoeba: A Distributed Operating Systems for the 1990s", *Computer*, May 1990, pp. 44-53.

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

[Ousterhout et al. 88] John K.Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N.Nelson, and Brent B. Welch, "The Sprite Network Operating System", *Computer*, Feb 1988, pp. 23-36.

[Satyanarayanan and Siegel 90] M. Satyanarayanan and E.H. Siegel, "Parallel Communication in a Large Distributed Environment", *IEEE Trans. on Computers*, Vol. 39, No. 3, Mar 1990, pp. 323-348.

[Satyanarayanan 90] M. Satyanarayanan, "Scalable, Secure, and Highly Available Distributed File Access", *Computer*, May 1990, pp. 9-21.

[Schantz et al. 86] R. Schantz, R. Thomas and G. Bono, "The Architecture of the CRONUS Distributed Operating System", *Proc of 6th International Conference on Distributed Computing System*, Cambridge, Massachusetts, May 19-23, 1986, pp. 250-259.

[Souza and Miller 86] Robert J. Souza and Steven P. Miller, "Unix and Remote Procedure Calls: A Peaceful Coexistence?", *Proc of 6th International Conference on Distributed Computing System*, Cambridge, Massachusetts, May 19-23, 1986, pp. 268-277.

[Sechrest 86] Stuart Sechrest, "An Introductory 4.3BSD Interprocess Communication Tutorial", *Unix Programmer's Supplementary Documents*, Vol. 1 (PS1), 4.3 Berkeley Software Distribution, Computer Systems Research Group, Computer Science Division, Univ. of California, Berkeley, Calif., Apr 1986.

[Spector 82] Spector, A.Z., "Performing Remote Operations Efficiently on a Local Computer Network", *Comm. of the ACM*, Vol. 25, No. 4, Apr 1982, pp. 246-260.

[Sun 87] Sun Microsystems, "XDR: External Data Representation Standard (RFC 1014)", in *Internet Network Working Group Request for Comments, No. 1014*, Network Information Center, SRI International, Jun 1987.

[Sun 88] Sun Microsystems, "RPC: Remote Procedure Call Protocol Specification Version 2 (RFC 1057)", in *Internet Network Working Group Request for Comments, No. 1057*, Network Information Center, SRI International, Jun 1988.

[Tay and Ananda 90] B.H. Tay and A.L. Ananda, "A Survey of Remote Procedure Calls", *ACM Operating Systems Review*, Vol. 24, No. 3, July 1990.

[Walker et al. 90] Edward F. Walker, Richard Floyd, and Paul Neves, "Asynchronous Remote Operation Execution in Distributed Systems", *Proc. 10th Intl. Conf. on Distributed Computing Systems (ICDCS-10)*, IEEE, Paris, France, May 28-June 1, 1990, pp. 253-259.

[Walker 90] Edward F. Walker, *Private Communication*, Oct 1990.

[Wilbur and Bacarisse 87] Steve Wilbur, Ben Bacarisse, "Building Distributed Systems with Remote Procedure Call", *Software Eng. Journal*, Sep 1987, pp. 148-159, also appeared in *UCL-CS TR 141*, Dept. of Comp. Sc., Univ. College London.

[Zahn et al. 90] Lisa Zahn, Terence H. Dineen, Paul J. Leach, Elizabeth A. Martin, Nathaniel W. Mishkin, Joseph N. Pato, and Geoffrey L. Wyant, *Network Computing Architecture*, Prentice-Hall, 1990.