



**Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom**

TELEPHONE:
INTERNATIONAL:
FAX:
E-MAIL:

**Cambridge (01223) 515010
+44 1223 515010
+44 1223 359779
apm@ansa.co.uk**

Training

ANSAwise - Remote Procedure Call in Distributed Systems

Chris Mayers

Abstract

Distributed systems usually use remote procedure call (RPC) as a fundamental building block for implementing remote operations.

However, these systems have detailed but important differences in the way RPC operates, and applications programmers need to be aware of these differences.

This module of the ANSAwise training programme discusses simple and advanced use of RPC. It covers RPC semantics, idempotence, failure handling policies, and asynchronous RPC. It discusses the differences between RPC operation in CORBA, DCE, and ANSAware.

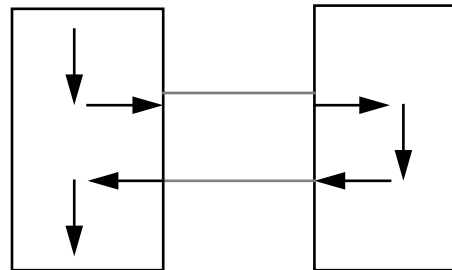
APM.1344.01

Approved
Briefing Note

25th November 1994

Distribution:
Supersedes:
Superseded by:

Remote Procedure Call in Distributed Systems



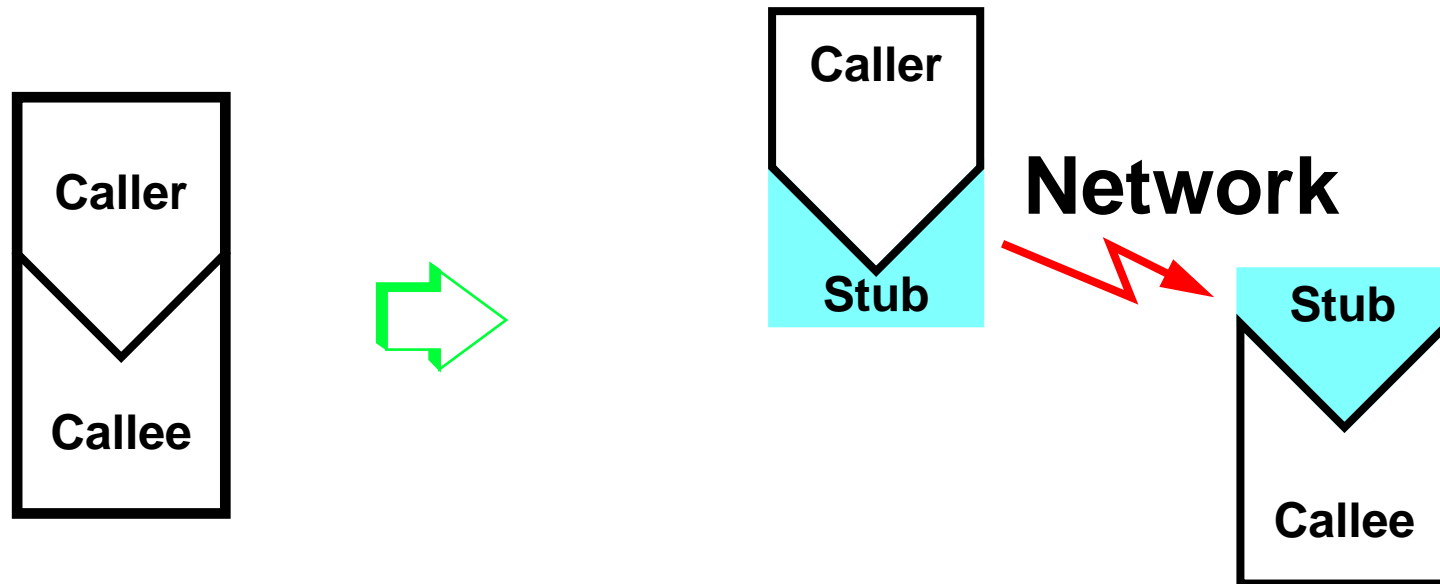


In this session

- *Explain the function of remote procedure call (RPC) in distributed systems*
- *Explain the importance of understanding RPC execution semantics*
- *Examine more sophisticated uses of RPC*
- *Explore the differences between RPC in various distributed environments*
 - **CORBA, DCE, and ANSAware**

Remote Procedure Call (RPC)

- *Local procedure call can be transformed into a remote procedure call*



- *The caller is the client, the callee is the server*



Stubs in RPC

- *The stubs in RPC are responsible for packing and unpacking the call parameters, and the call results*
 - this is called marshalling/unmarshalling
- *Stubs must allow for the fact that client and server may be machines of different types*
 - for example, integers may be represented differently (byte-ordering)



RPC execution semantics

- *In an ordinary local procedure call you need not be concerned about independent failure of client and server*
- *... in a remote procedure call you must be able to handle this*
- *This is reflected in the RPC semantics, which may be*
 - *at-least-once*
 - *at-most-once: the realistic case*
 - *exactly-once: the ideal case*
- *An RPC product may offer a choice of the above*
 - *different RPC products offer different choices with different defaults*
 - *.... this affects portability between RPC products*



At-least-once RPC semantics

- ***At-least-once semantics are appropriate for operations that have the same effect when invoked more than once***
 - these are called *idempotent*
- ***For example***
 - “add 50 units to stock level” is not idempotent...
 - ... “set stock level to 100 units” is idempotent
- ***DCE allows you to specify an operation as idempotent***
 - it will be executed more efficiently
 - ... but DCE does *not* support at-least-once semantics at all!
- ***At-least-once has a straightforward implementation***
 - ‘retransmit until acknowledged’



At-most-once RPC semantics

- *At-most-once RPC semantics are appropriate for non-idempotent operations*
- *Clients must allow for operation not having occurred*
- *Sometimes described as best-effort*
- *Trivial implementation ('fire and forget')*
 - *in practice, retransmit until acknowledged, with duplicate suppression*



Exactly-once RPC semantics

- *At-least-once + At-most-once = Exactly-once*
- *Straightforward implementation under normal conditions...*
 - 'wait for acknowledgment'
 - ... in practice, periodic retransmission and duplicate suppression
- *... much harder under failure conditions*
 - requires server and message replication, and special group RPC execution and recovery protocols to reduce failure probability
 - still an active area of research; some commercial solutions are available



RPC semantics and failures

- *There are two cases to consider*
 - **successful case**
 - **failure (exception) case**
- *In CORBA, DCE, and ANSAware the default is*
 - **if successful, exactly-once**
 - **if failure (exception), at-most-once**

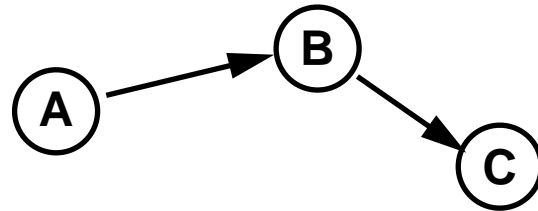


Handling RPC failures - policy and transparency

- *This is the application's responsibility...*
 - ... which it may choose to delegate to the infrastructure in the form of standard policies
 - ... for example, to retry a certain number of times for certain failures
- *Failures can be made almost transparent to the application programmer*
 - for example, when contacting a service, ANSAware will automatically invoke a relocater if the service cannot be found...
 - ... but this default behaviour can always be overridden
- *Failures can never be eliminated*

The nested timeout problem

- *Suppose there is a chain of invocations...*



- *... A client, B client and server, C server*
- *The A->B timeout must be no less than the B->C timeout*
 - *or A may incorrectly believe that B has failed*
- *B must also allow time to retry C, if that is its chosen policy*
- *This may force timeouts to be unduly large*
 - *making failure detection slow*



Transforming a non-idempotent operation

- *Consider the non-idempotent operation...*
 - “Add 50 units to stock level”
- *...transform this into the pair of idempotent operations*
 - “Read old stock level”
 - “Set stock level to old stock level plus 50 units”
- *Why is such a transformation almost never practical?*
 -
 -
 -
 -

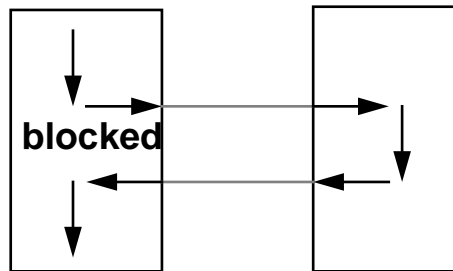


Invocation versus Initiate and Redeem

- *There are two basic ways of using an operation*
 - invocation (synchronous)
 - initiate and redeem (asynchronous)

Invoking an operation

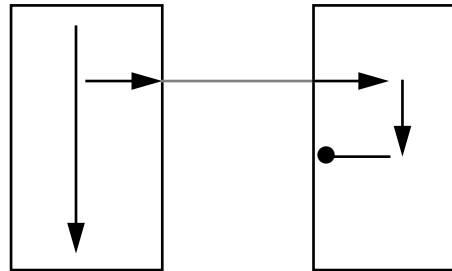
- *Invoking an operation is synchronous...*



- *...the invoker blocks until the operation is complete, and the result is available*
- *This mimics the behaviour of an ordinary local procedure call*

Initiating an operation

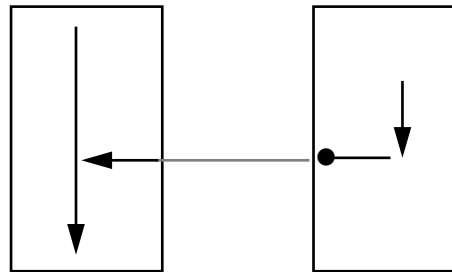
- *Initiating an operation is asynchronous...*



- *...the initiator continues concurrently*

Redeeming an initiate operation

- *The initiator is given a voucher*
- *When the initiator is ready for the result of the operation, it redeems it*



- *If the operation is not yet complete, the redeem will block*
 - *just as for an invocation*
- *Initiate/Redeem is also known as follow-up RPC (FRPC)*



Using follow-up RPC

- *Use follow-up RPC only if the client can do useful work in the meanwhile*
 - local I/O and computation
 - invoking other remote operations in parallel
- *Treat follow-up RPC as an engineering optimization*



Simple use of follow-up RPC

- *Many operations are naturally implemented as a series of read operations followed by a series of write operations*
 - sometimes, these write operations are independent of each other
- *For example a customer change-of-address might involve updates to*
 - customer details
 - account manager details
- *These updates could be executed as separate follow-up RPCs*
 - in parallel with each other

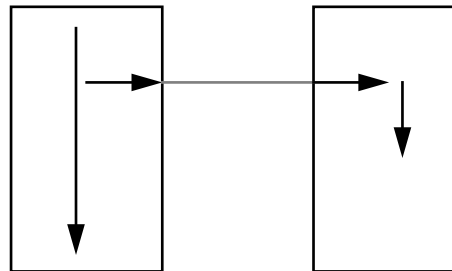


Difficulties with follow-up RPC

- *Error handling is bound to be more complex*
 - suppose you have initiated 3 operations, and the first of them fails?
 - you also need to be careful about race conditions and timeouts
- *Some implementations do not preserve the order of follow-up RPC correctly*

One-way operations

- *One-way operations are like an initiate with no redeem...*



- *...obviously, no results can be returned*
- *Semantics may be restricted*
 - for example, at-most-once only
- *Some infrastructure failures may still be detected on the client side*
 - for example, a local transmission failure
 - the client application must still cope with these errors



Specifying one-way operations

- *One-way operations are specified in the interface definition*
- *In CORBA this is specified using the oneway operation attribute*
 - the semantics are at-most-once
- *In DCE this is specified using the maybe IDL attribute*
 - the operation is implicitly idempotent
 - the semantics are no-guarantee (0, 1 or more times)
- *In ANSAware they are specified as ANNOUNCEMENTS*



One-way operations as building blocks

- *One-way operations are best regarded as a communications mechanism for building application-specific interaction types*
 - for example, a 3-phase data commit handshake, or bulk data transfer
- *One-way operations give an application 'raw' one-way communications*
 - ... with access and location transparency
 - ... without the handshake overhead required by stricter RPC semantics
- *One-way operations are for the systems programmer*
 - not the applications developer
- *Streams fulfil the real need*

Follow-up RPC versus one-way operations



- *Both offer potentially faster performance*
- *Both are described as ‘asynchronous’*
 - *but are quite different*



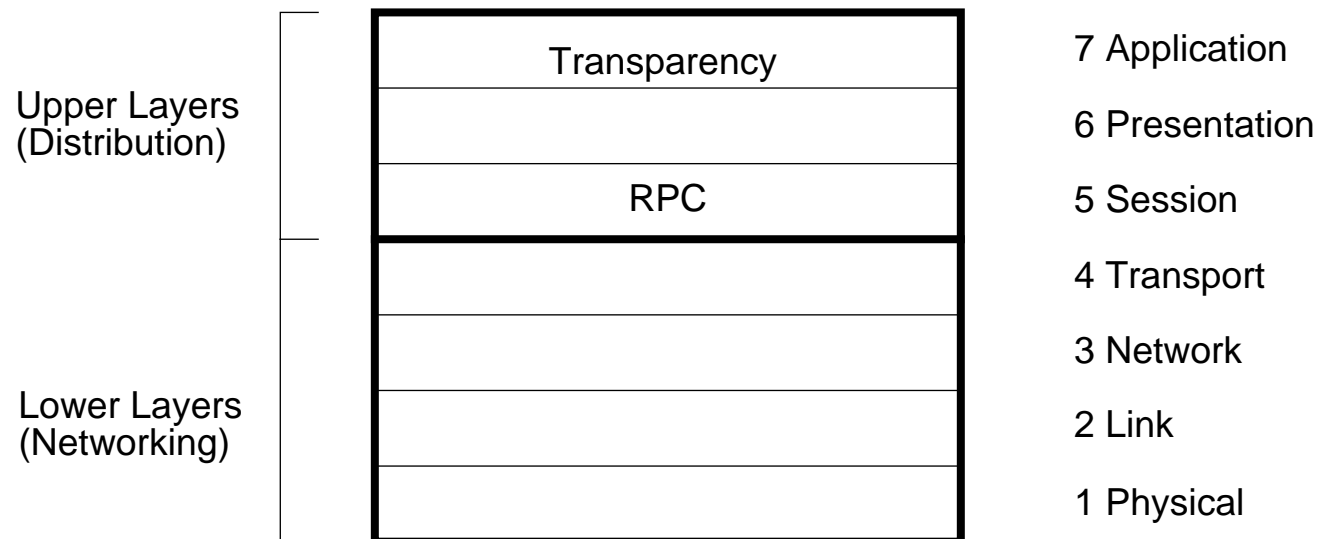
Follow RPC compared with one-way operations

- *Follow-up RPC*
 - always gets a response (when you redeem)
 - can have results
 - can be used on any operation; it is an implementation choice for the client
- *One-way operation*
 - never gets a response (although failure is still possible)
 - cannot have results
 - are specified for the operation; affects all clients



RPC in the communications protocol stack

- RPC is the lowest level building block of a distributed system*





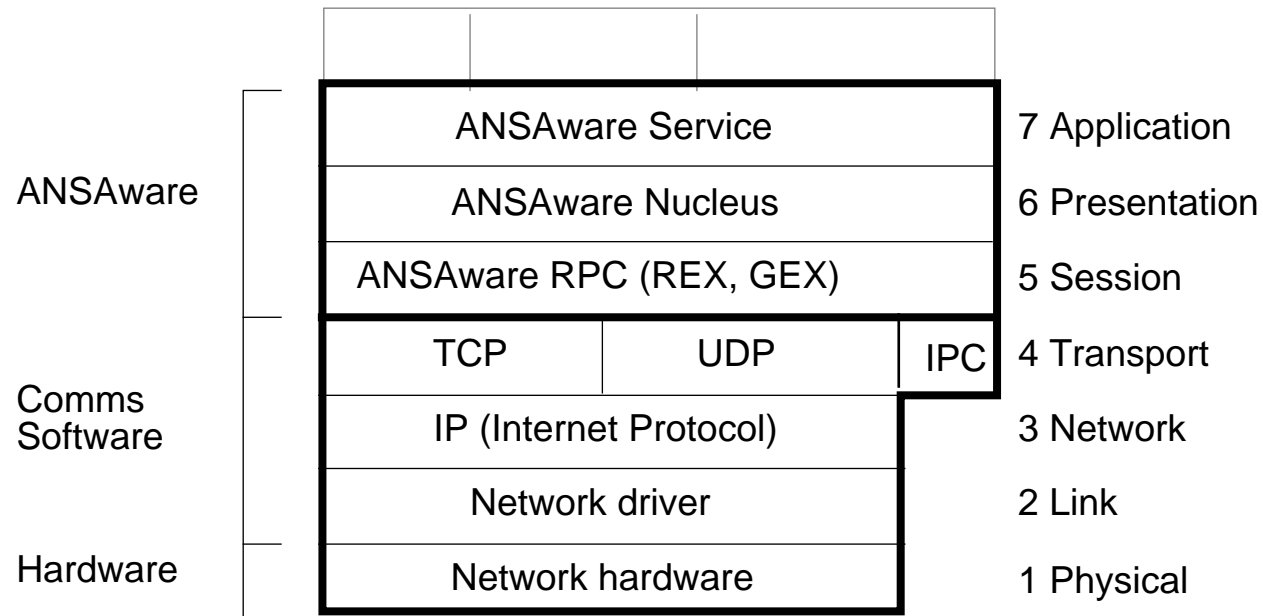
RPC and transport protocols

- *RPC can be implemented over*
 - connection-oriented transport protocols (e.g TCP)
 - connectionless transport protocols (e.g UDP)
- *RPC is a more natural fit to connectionless protocols*
 - request/response maps onto a pair of outgoing/incoming datagrams
- *But connectionless protocols often*
 - are “unreliable” (delivery not guaranteed)
 - have a maximum size (determined by the underlying network)
- *The RPC layer must then provide its own delivery guarantees, and handle its own fragmentation*
 - even so, performance may well be better over a connectionless protocol



Multiple transport protocols

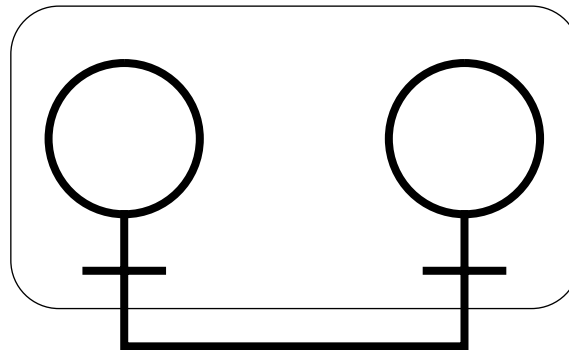
- *Systems can support multiple transport protocols simultaneously...*



- *... including a lightweight local RPC protocol (here, IPC)*

Lightweight local RPC

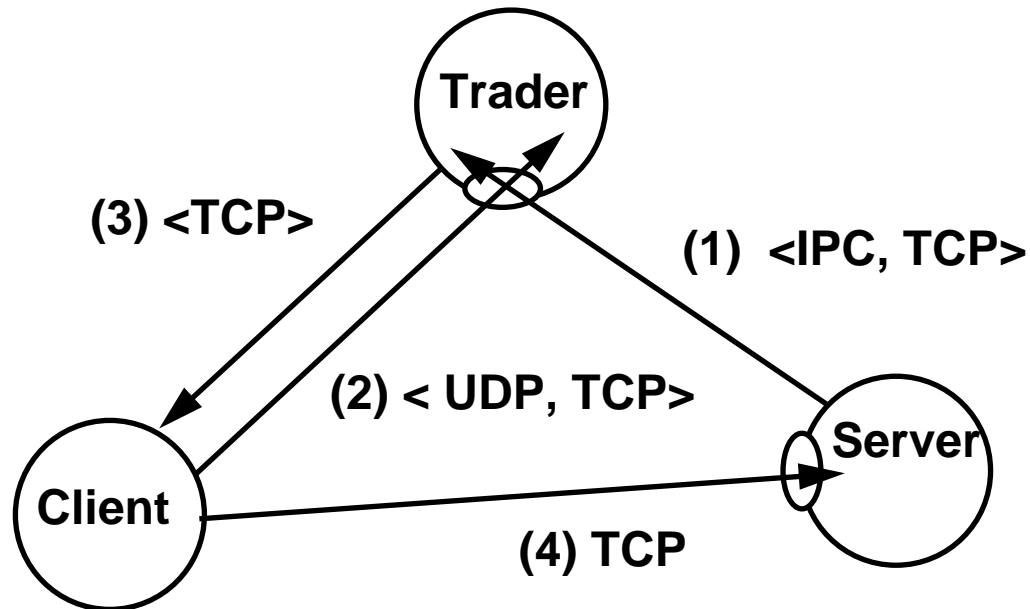
- *Calls between co-located objects (on the same machine) can be optimized*



- *... avoiding network traffic*
- *... avoiding data copying*
- *... avoiding marshalling overheads*

Trading and multiple transport protocols

- *A service can state which transport protocols apply to an offer...*



- *... allowing a common protocol to be selected (here, TCP)*



Multiple RPC protocols - interoperability

- *Suppose you have both CORBA and DCE...*
 - ...how can a CORBA client access a DCE server and vice-versa?
- *It would be necessary to federate their RPC protocols*
 - ... but RPC interoperability alone is not enough for service interoperability
 - ... many other issues have to be resolved



Basic performance of RPC

- *With the configuration...*
 - Ethernet (10 Mbit/s)
 - 10 MIPS CPU
- *... 1992 measurements indicate for a Null RPC (no arguments, no results, no computation at server)*
 - 2 to 4 milliseconds (typical implementation)
 - 0.3 milliseconds (best known implementation)
- *... this is how long it takes to 'do nothing'!*



Realistic performance of RPC

- *Instead of a null RPC, assume a more realistic...*
 - 100 bytes request packet size
 - 2 milliseconds server computation
 - 100 bytes response packet size
- *... this equates to*
 - 6 milliseconds (typical implementation)
 - 2.5 milliseconds (best known implementation)
- *... or, put another way*
 - 150 RPCs/second (typical implementation, single-threaded client)
 - 400 RPCs/second (best known implementation)
 - 30 kbytes/second (2% of Ethernet) (typical)
 - 80 kbytes/second (6% of Ethernet) (best)



Comparison with file transfer

- *The same machine on the same network performing file transfer of the same packet size...*
 - using the standard Novell NetWare PERFORM3 benchmark program
 - 200 kbytes/second (best)
- *... with bigger packet size (2 kbytes)*
 - 500 kbytes/second (best)
 - two clients can (*and do*) saturate Ethernet (1000 kbytes/second)



Why is RPC performance limited?

- ***File transfer is faster because it typically uses***
 - **specific file-transfer protocols**
 - **windowed network protocols**
 - **larger packet size matching network packet size**
 - **protocol stack optimized for LAN file transfer performance**
 - **hand-optimized Ethernet drivers**
- ***RPC is slower because it uses***
 - **general-purpose RPC protocols**
 - **call/response interactions**
 - **packet sizes typically much smaller than optimal network packet sizes**



How important is RPC performance?

- *Look again at the figures for a typical implementation...*
 - 6 milliseconds for a remote procedure call
 - 2 milliseconds for the equivalent local procedure call (pure computation)
- *... is a factor of 3 significant in your application?*



Real-life factors in RPC performance

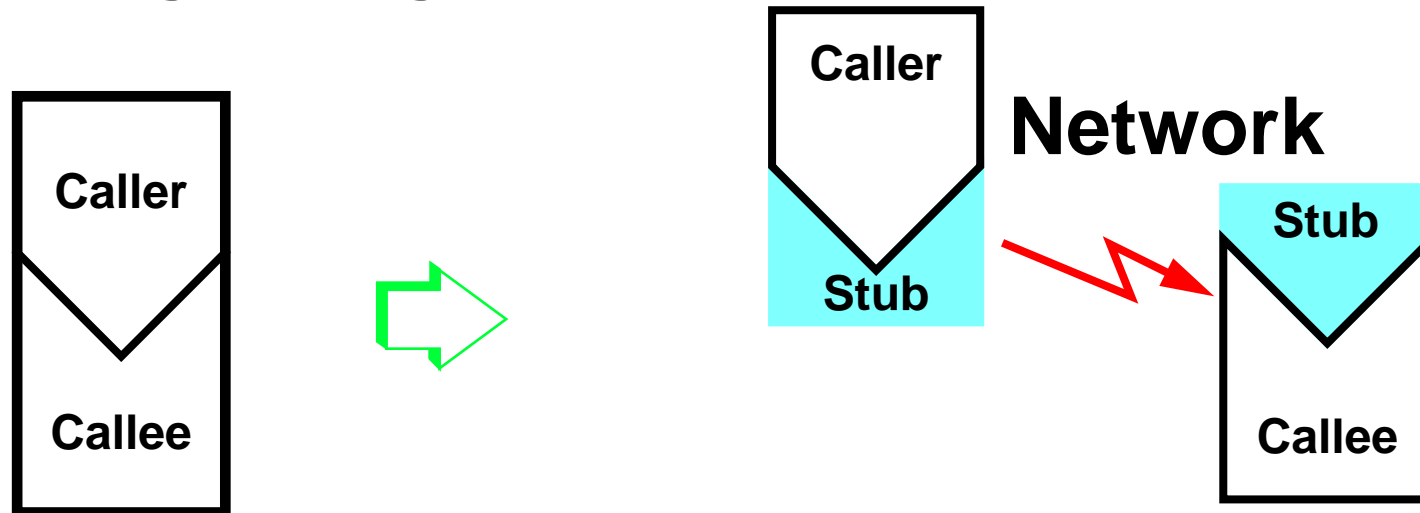
- *If you wish to make your own measurements, all these need to be considered*
 - Size of request (marshalling and buffering overhead)
 - Network bandwidth (unlikely to be a limiting factor for LANs)
 - Network occupancy
 - Speed of local and remote machines
 - Concurrency and other load on remote machines
 - Transport protocol selected
 - Other overheads (e.g. authentication)
 - **Actually servicing the request!**



Engineering control for ordinary procedure calls

- *Suppose you are writing an ordinary non-distributed program...*
- *...for a conventional procedure call, your compiler may offer you*
 - a choice of calling conventions, for compatibility with other programming languages
 - the option of generating inline code
- *Carefully used, valuable techniques for performance engineering*

Engineering control for remote procedure calls



- *For a remote procedure call you may require engineering control over the stubs, for example*
 - to control marshalling of parameters and results
 - to control implicit binding
 - to control error handling policy



Engineering control over stubs

- *In DCE, this is done using the Attribute Configuration Language*
 - for example, the *nocode* attribute eliminates client-side stub code for operations that the client does not use
- *In CORBA, this is implementation-dependent*
- *In ANSAware, various ad-hoc facilities are provided*
 - for example, to control which transport protocols will be offered and used



Other advanced RPC features

- *Group RPC for replication*
 - often using multicast protocols
- *Secure RPC for integrity and confidentiality*
 - supporting authentication and authorization...
 - ... a particular strength of DCE
- *Transactional RPC for dependability*
- *...support for all these features is patchy*



General guidance - specifying interfaces

- *When specifying an operation, determine whether it is logically idempotent or not*
 - *but do not alter the interface to make it idempotent*
- *Avoid using one-way operations, unless you have a very clear reason*
- *Use the default RPC semantics (at-most-once under failure)*
 - *and have a standard application policy for handling failures/exceptions*



General guidance - configuration

- *Choose one RPC protocol for all your applications*
- *Ensure that appropriate transport protocols are available*
 - including a transparent optimized local RPC for the co-located case
- *Ensure that the programming language mappings you need are available*
 - check both client and server machines
 - also check compiler and OS versions
- *Beware of OS-specific restrictions on RPC*
 - for example in non-multi-tasking systems like DOS or Macintosh
 - ... ANSAware has no such restrictions; other products do!



General guidance - performance

- *Decide whether performance is an issue for your application*
- *Consider benchmarking a simple RPC-based application in your own configuration*
 - holding constant as many factors as possible
- *Do not expect RPC performance to match measured file transfer performance in the same configuration*
- *Consider selective use of follow-up RPC*
 - once the application is working!
- *Aim for fast-enough performance for successful invocations*
 - reliable-enough handling for failed invocations



Summary

- ***For more information on RPC***
 - ***see Chapter 5 of *Distributed Systems Concepts and Design* by Coulouris, Dollimore, and Kindberg (Addison-Wesley)...***
 - ***... this has a detailed comparison of Sun RPC and a previous version of ANSAware***
 - ***also see *Power Programming with RPC* by John Bloomer (O'Reilly and Associates)***