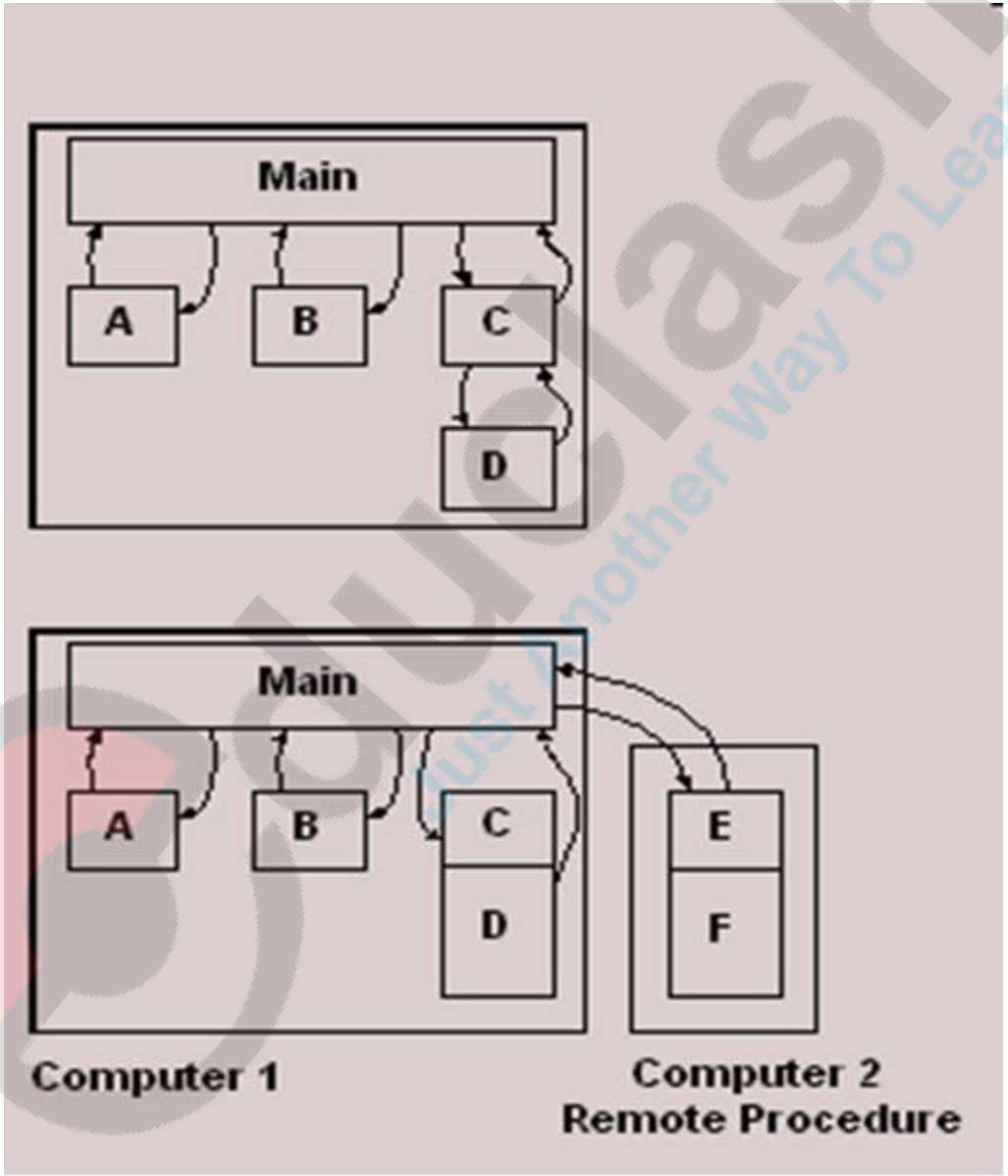
Remote Procedure Call

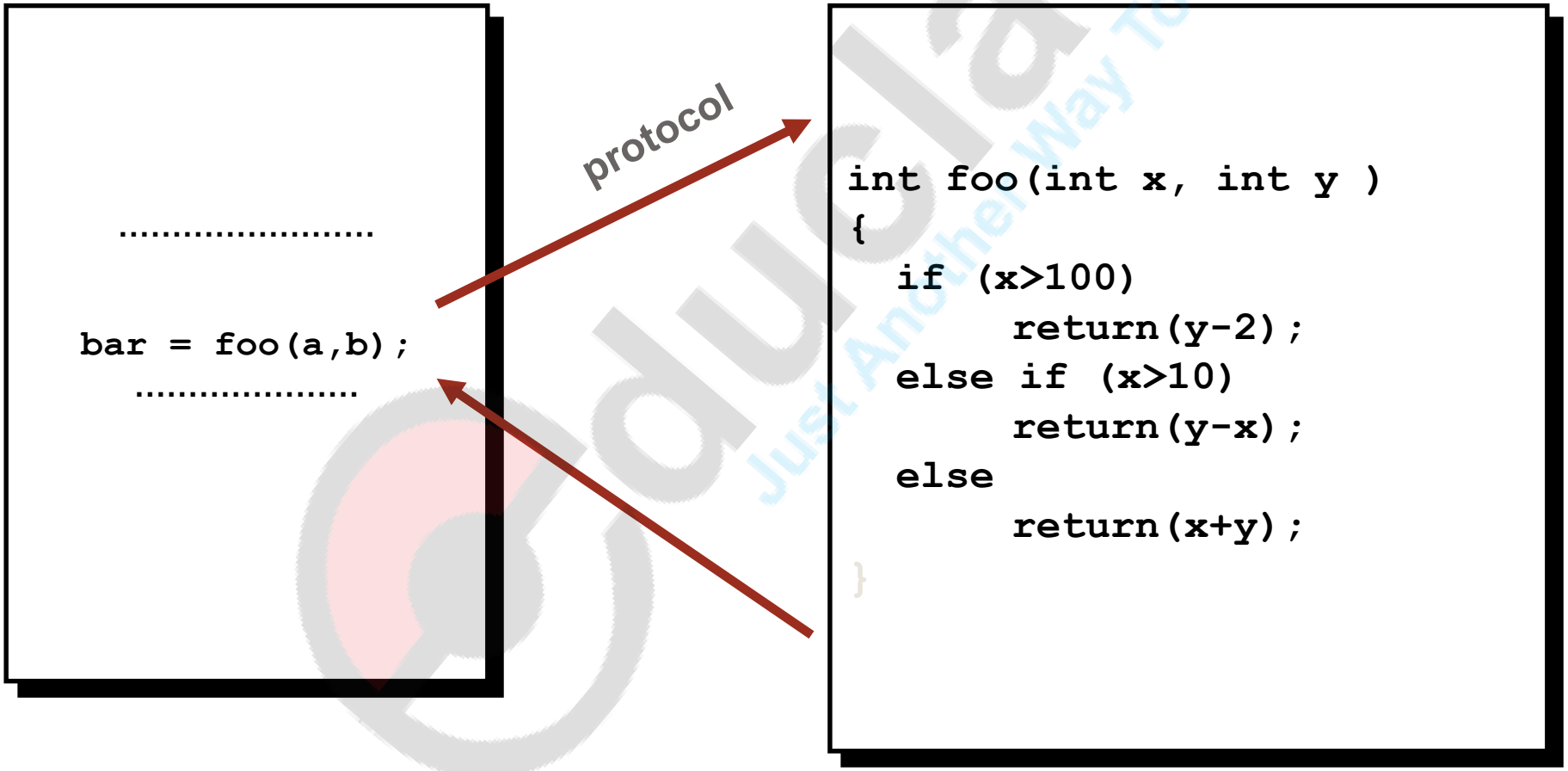
The Programming Model



Remote Subroutine

Client

Server



RPC

- Remote Procedure Call (RPC) is a powerful, robust, efficient, and secure inter process communication technique for creating distributed client/ server programs
- It makes client/server interaction easier and safer by handling common tasks, such as security, synchronization, data flow etc.
- It enables data exchange and invocation of functionality residing in a different process. That different process can be on the same machine, on the local area network, or across the Internet
- Though RPC is not a universal answer for all types of distributed applications, it does provide a valuable communication mechanism that is suitable for building a fairly large number of distributed applications
- RPC(Remote Procedure Call) is a way to
  - Hide communication details behind a procedural call so user does not have to understand the underlying network technologies
  - Bridge heterogeneous environments

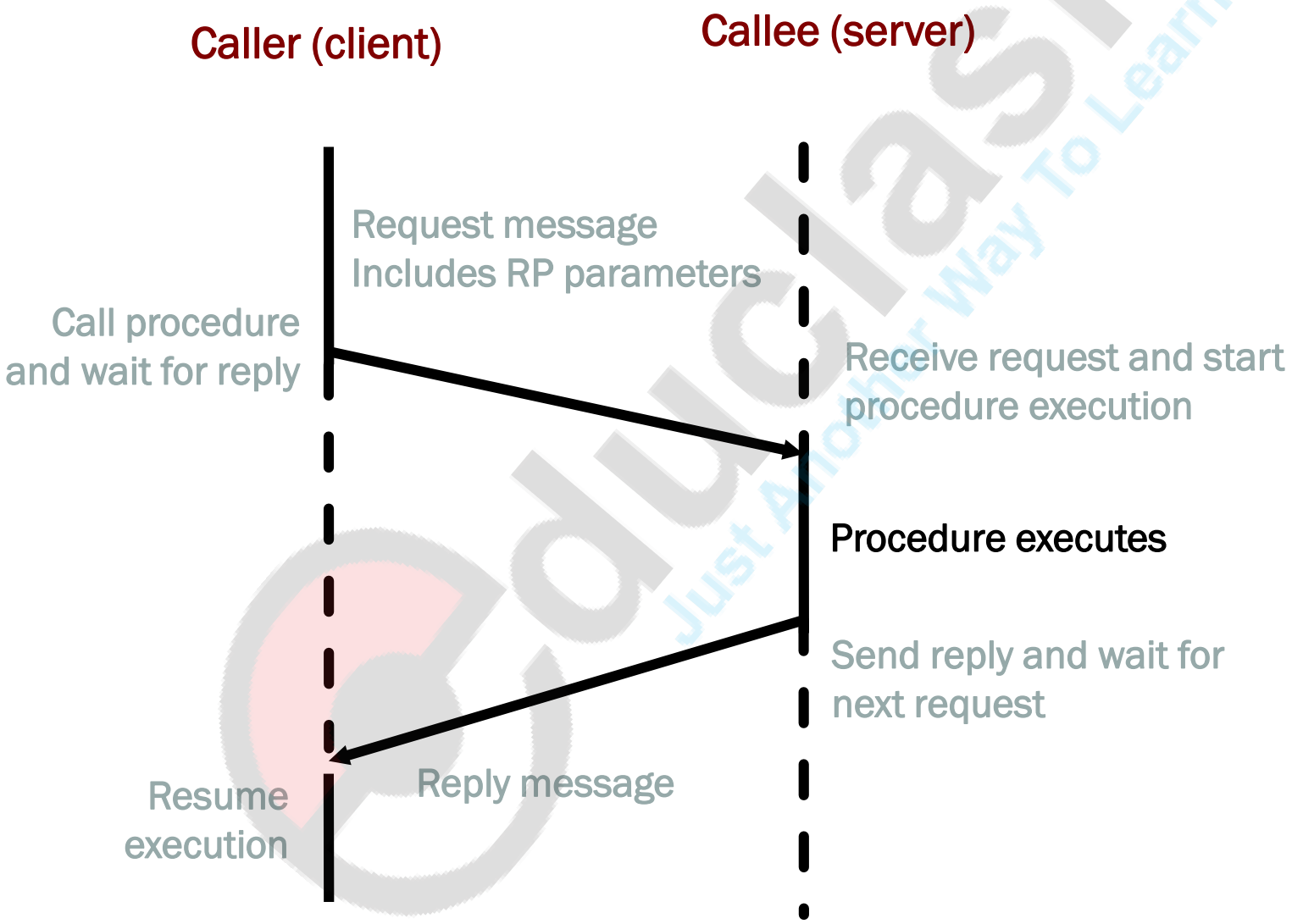
RPC (Cont'd)

- RPC has become a widely accepted IPC mechanism in DCS
- Its popularity is mainly because of the following features:
  1. Simple call syntax
  2. Familiar semantics (similar to local procedure calls)
  3. Its specifications of a well defined interface. It is used to support compile time checking and automated interface generation
  4. Its ease of use: clean and simple semantics of a procedure call makes it simpler to build distributed applications right
  5. Its generality between single machine and multi machine applications
  6. Its efficiency. Procedure calls are simple enough for communication to be rapid
  7. It can be used as an IPC mechanism to communicate between processes on different machines as well as between processes on the same machine

Issues Handled by RPC

- How to invoke service in a more or less transparent manner
- How to exchange data between machines that might use different representations for different data types.
  - data type formats (e.g., byte orders in different architectures)
  - data structures (need to be flattened and reconstructed)
- How to find the service, one actually wants, among a potentially large collection of services and servers
  - Client should not know where the server resides or even which server provides the service
- How to deal with errors in the service invocation:
  - Server is down, communication link broken, server busy, duplicated requests etc

RPC Model



RPC Model (Cont'd)

- The caller normally known as client process sends a call request message to callee (called server process) and waits (blocks) for a reply message
- The request message contains the remote procedures parameters among other things
- The server process executes the procedures and then returns the result of the procedure execution in a reply message to the client process
- Once the reply message is received, the result of the execution is extracted and the caller's execution is resumed
- The server process is normally dormant, awaiting the arrival of a request message

RPC Model (Cont'd)

- When one arrives, , it extracts procedure's parameters, computes the result, sends reply message and then awaits the next message
- Sometimes, RPC implementation may be asynchronous so that caller can do some useful work while waiting for the reply
- Another possibility is to have the server create a thread to process the incoming request, so that server can be free to process other requests

Transparency of RPC

- A major issue in the design of RPC is its transparency property
- Local procedures & remote procedures are indistinguishable to programmers. This requires:
  - Syntactic Transparency: RPC should have same syntax as local PC
  - Semantic Transparency: Semantics of RPC is identical to those of LPC
- Syntactic Transparency is very easy to achieve
- Semantic Transparency almost impossible to achieve because:
  - **Address space of calling process & called process is disjoint**
  - The remote (called) procedure can not have access to any variables or data values in the calling program's environment

Transparency of RPC (Cont'd)

- Hence in the absence of shared memory, it is meaningless to pass addresses in the arguments, pointers to the structure
- Hence parameter passing (pointer and structures) is going to be different
- **Remote procedure calls need the ability to take care of possible processor crashes & communication problems of network and hence vulnerable to failure**
- **Remote procedure calls take more time due to involvement of communication network (network congestion)**
- Because of these difficulties in achieving normal call semantics for remote procedures, many feel that RPC should be nontransparent
- Hence in most environments, total semantic transparency is impossible, enough can be done to ensure the programmers are comfortable

Implementing RPC Mechanism

- RPC implementation is based on the concept of stubs, which provide a perfectly normal LPC abstraction by concealing from the programs the interface to RPC system
- To conceal the interface of the RPC system from both the client and server processes, a separate stub procedure is associated with both processes
- To hide intricacies and function details of underlying network, an RPC communication package (known as RPC Runtime) is used on both client and server side
- Hence implementation of RPC involves following five elements
  - The client
  - The client stub
  - The RPC Runtime
  - The server stub
  - The server

Elements of RPC (Cont'd)

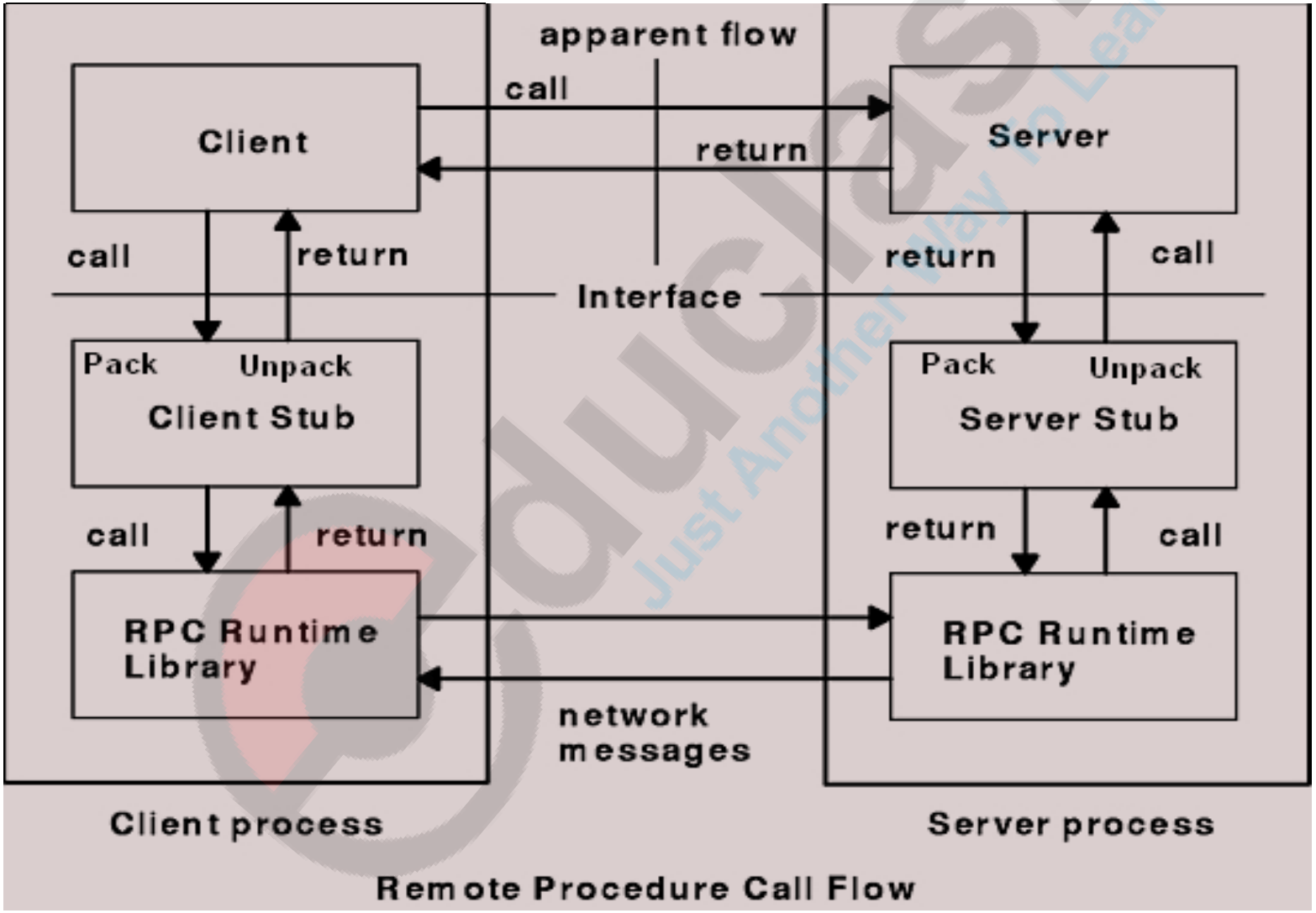
- Client
  - A process, such as a program or task, that requests a service provided by another program
  - The client process uses the requested service without having to deal with many working details about the other program or the service
- Client Stub
  - Module within a client application containing all of the functions necessary for the client to make remote procedure calls and performs following functions :–
    - On a request from client, packs a specification of target procedure & arguments into a message & asks local RPC Runtime to send it to server stub
    - On receipt of procedure execution, it unpacks result & passes it to client

Elements of RPC (Cont'd)

- RPC Runtime
  - Handles transmission across the network between client and server machine.
  - It is responsible for retransmissions, acknowledgment, packet routing and encryption.
- Server Stub
  - Module within a server application or service that contains all of the functions necessary for the server to handle remote requests using local procedure calls.
  - On receipt of call request, it unpacks it and makes a normal call to invoke the procedure in the server.
  - On receipt of the result of procedure execution, it packs the result into a message and ask the RPC Runtime to send it to the client stub.
- Server
  - A process, such as a program or task, that responds to requests from a client.

How RPC Works

- Interaction between them is shown below



RPC Procedure

1. Client makes a procedure call
2. Client stub retrieves the required parameters from the client address space
3. Translates the parameters as needed into a standard format (NDR Network Data Representation) for transmission over the network
4. Calls functions in the RPC client run-time library to send the request and its parameters to the server
5. The server RPC run-time library functions accept the request and call the server stub procedure
6. The server stub retrieves the parameters from the network buffer and converts them from the network transmission format to the format the server needs
7. The server stub calls the actual procedure on the server

RPC Procedure

8. The remote procedure then runs, possibly generating output parameters and a return value. When the remote procedure is complete, a similar sequence of steps returns the data to the client
9. The remote procedure returns its data to the server stub
10. The server stub converts output parameters to the format required for transmission over the network and returns them to the RPC run-time library functions
11. The server RPC run-time library functions transmit the data on the network to the client computer
12. The client RPC run-time library receives the remote-procedure return values and returns them to the client stub
13. The client stub converts the data from standard format to the format used by the client computer. The stub writes data into the client memory and returns the result to the calling program on the client
14. The calling procedure continues as if the procedure had been called on the same computer

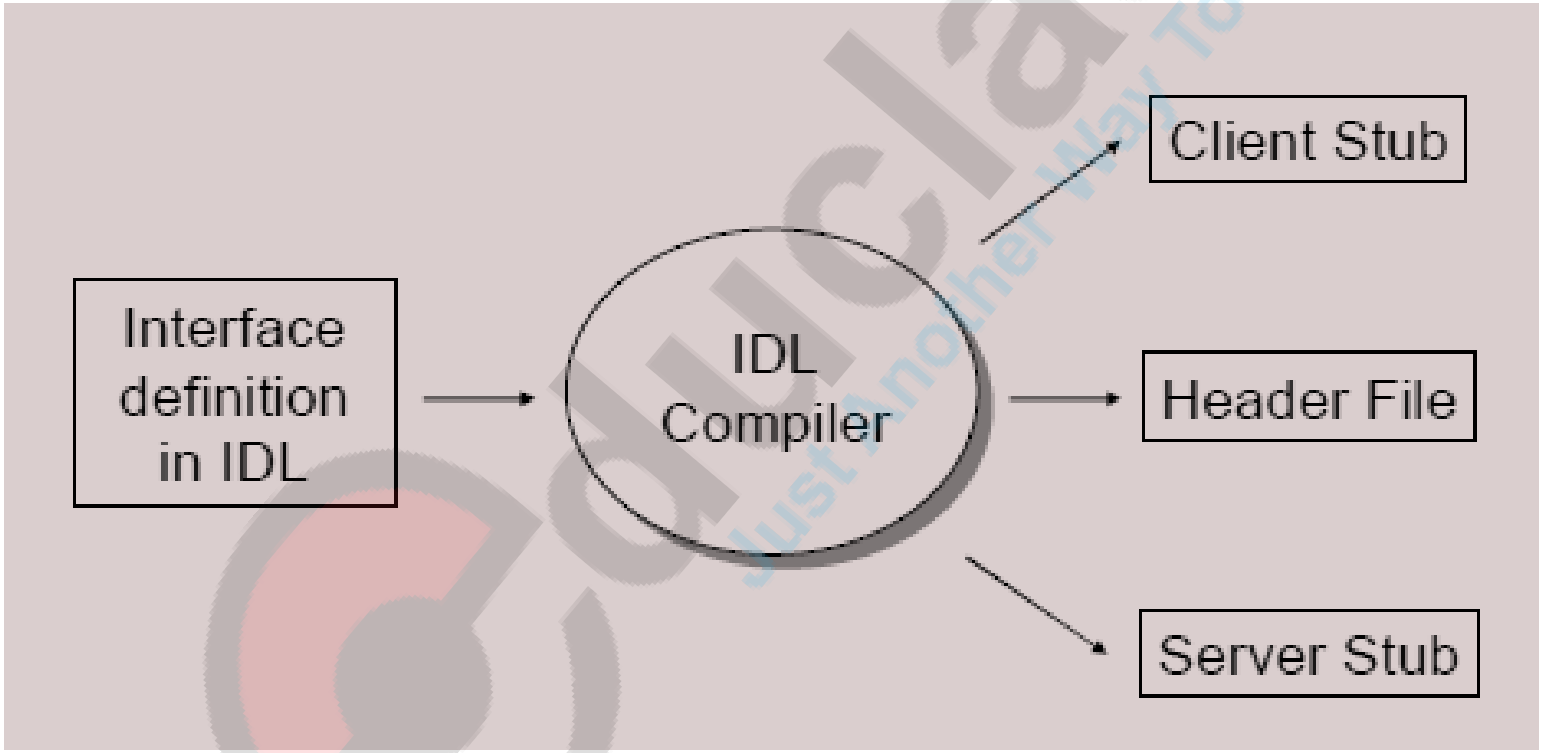
Stub Generation

- Manually
  - A set of translation functions are provided from which user can construct his own stub. The method is simple and handle very complex parameter types
- Automatically
  - It uses Interface Definition Language (IDL) to define interface between client & server
  - This is more commonly used method for stub generation
  - A set of procedure names supported by the interface, with the types of their arguments and results are devised.

Interface Definition Language

- IDLs describe an interface in a language-neutral way, enabling communication between software components that do not share a language
- For example, between components written in C++ and components written in Java
- An interface compiler is then used to generate the stubs for clients and servers (appropriate marshalling/ unmarshalling operations & header file that supports data type in interface definition)
- IDL has header file which is included in both client and server programs and client stub procedures are compiled and linked with client programs, while server stub procedures are compiled and linked with server programs

Where Stubs Come From



RPC Messages

- Any RPC involves a client process and a server process that are located on different machines
- This requires some messages to be exchanged between them
- There are two types of messages
  - *Call Messages* that are sent from the client to the server for requesting execution of a particular remote procedure
  - *Reply messages* that are sent by server to the client for returning the result of the remote procedure execution
- The protocol of the RPC system defines the format of these two types of messages, and they are independent of the transport protocols
- i.e., RPC does not care how the messages are passed between two systems

Call Messages

- This makes the messages to be simple and easy to handle
- The call message must have basic components which are necessary
  - Identification information of the Remote procedure to be executed
  - Arguments necessary for the execution of the procedure
- In addition normally it contains the following fields
  - Message identification field that contain a sequence number
    - It is needed for identifying lost or duplicate messages in case of system failure and for properly matching reply messages with outstanding call messages, especially when several outstanding call messages arrive out of order
  - Message type field that specify whether it is a call or reply message

Call Messages (Cont'd)

- A Client identification field: used for identifying the client to whom reply message has to be sent as well as for checking the authenticity of the client process for executing a particular procedure

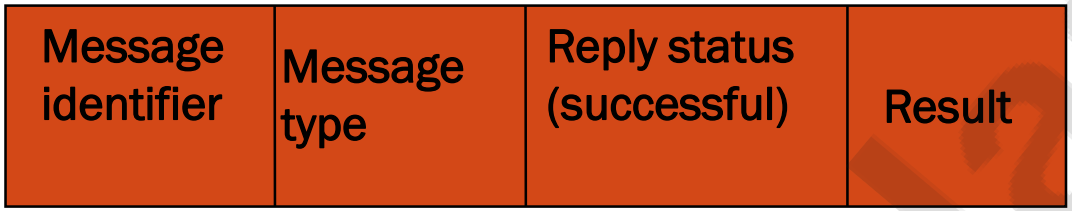
- **Hence a typical call message format may be as given below**

Message identifier	Message type	Client identifier	Remote procedure identifier			Arguments
			Program number	Version number	Procedure number	

Reply Messages

- When a server receives a call message from a client, it will be faced with one of following conditions
  - Call message is not intelligible or it violates RPC protocol. The Server rejects such calls
  - Client is not authorized. Server returns unsuccessful reply.
  - Remote program, version or procedure no. specified is not available with server. Unsuccessful reply
  - If this stage is reached, an attempt will be made by the server to execute the remote procedure specified in the call message. If it is unable to decode supplied arguments. This may be due to incompatible RPC interface used by client and server
  - An Exception (such as division by zero) occurs while executing.
  - Procedure is executed successfully & reply sent back

Reply Messages (Cont'd)



Successful Reply

Unsuccessful Reply

- Obviously in the first five cases, an unsuccessful reply has to be sent with the reason for failure in processing the request
- Successful reply has to be sent in the last case with the result of the procedure execution
- A typical RPC reply messages are shown above

Reply Messages (Cont'd)

- Message identifier is same as in the call message so that the client can properly match them
- Type field is set to reply message
- A successful reply, has status field as zero followed by the result of the procedure execution
- A non-zero status field indicates unsuccessful and its value specifies type of error. In any case a short message is placed in the last field

Marshaling Arguments and Results

- Implementation of RPC involves transfer of arguments from client to server and transfer results from server to client
- These are basically language level data structures (program objects)
- These are transferred in the form of message data between the two computers involved in the call
- In IPC we have seen encoding/decoding of data for transfer of message data between two computers
- In RPC this operation is called as Marshalling
- It is the packing of procedure parameters into a message packet(Encoding/ Decoding). It involves:
  - Taking arguments of a client process or result of server process that will form the message data to be sent to the remote process

Marshaling Arguments and Results (Cont'd)

- Conversion of language level data structures and program objects into stream form that is suitable for transmission and placing them into a message buffer
- Decoding of the message data on the receiver computer and reconstruction of program objects from message data stream

- In order that marshaling can be performed successfully, the order and the representation method (tagged or untagged) used to marshal arguments and results must be known to both the client and server or the RPC

Marshaling Arguments and Results (Cont'd)

- Marshaling procedures are of two types
  - Those provided as a part of RPC software
    - Used for scalar & compound data types built from the scalar ones
  - Defined by users of RPC system
    - Used for user-defined data types & pointer data types
  - A good RPC system should always generate in line marshaling code for every remote call, so that the users are relieved of the burden of writing their own marshaling procedures

Server Management

- In RPC based applications, two important issues to be considered for server management are server implementation and server creation
- Based on the style of implementation used, servers may be of two types: stateful or stateless
- Stateful Server
  - Maintains client's state information from one Remote Procedure Call to the next
  - Client's state information is subsequently used at time of execution of second call
  - Server gives the client a connection identifier unique to the client
  - Identifier is used for subsequent accesses until the session ends

Stateful Server

- Server reclaims main-memory space used by clients that are no longer active

Open (filename ,mode):This operation is used to open a file identified by *filename* in the specified *mode*. When the server executes this operations, it creates an entry for this file in the open file state information table, resets read and write pointers to zero and returns file identifier *fid* to client, which is used by the client for subsequent access to the file

Read(fid, n , buffer): Reads *n* bytes of the file *fid* into buffer named *buffer* starting from the current pointer (maintained by the server). When the server executes this operation, it returns to the client n bytes of the file data starting from the byte currently addressed by the *read-write pointer* and then increments the *read-write pointer* by *n*

Stateful server (Cont'd)

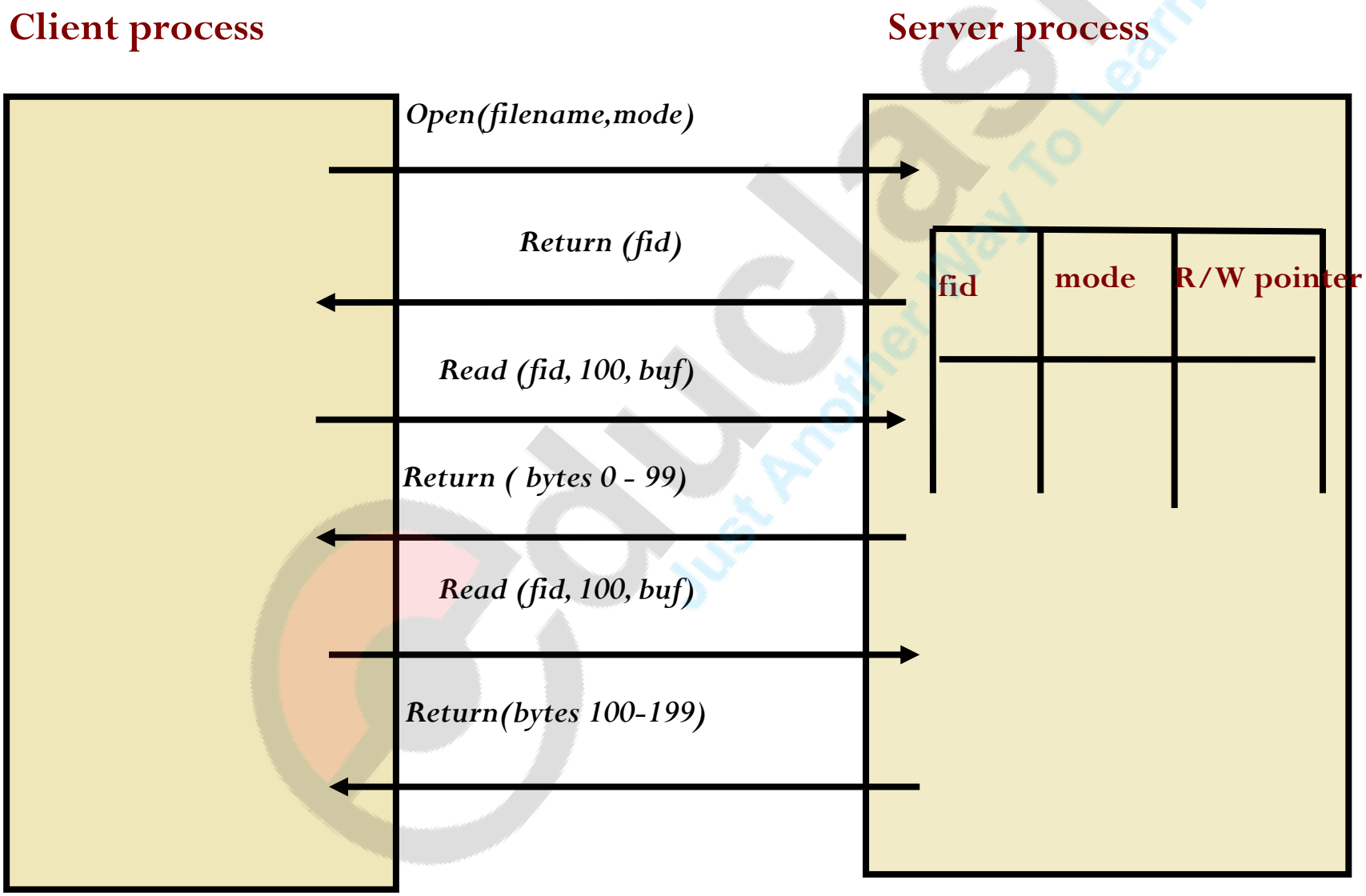
Write (fid, n ,buffer): On execution of this operation, the server takes *n* bytes of data from specified *buffer*, writes it into the file identified by *fid* at the byte position currently addressed by the *read-write pointer* and then increments *read-write pointer* by *n*

Seek (fid , position): This operation causes the server change the value of the *read-write pointer* of the file identified by *fid* to a new value specified by the *position*

Close (fid): Causes the server to delete from its *file-table* the file state information of file identified by *fid*

- The server discussed above is stateful server as it maintains the current state information for a file that has been opened for use by a client

Example of Stateful File Server



Stateless Server

- A stateless server does not maintain any client information
- Every request from a client must be accompanied with all necessary parameters to successfully carry out the desired operation, i.e. every request is self-contained
- No need to establish and terminate a connection by open and close operations

Read (filename, position, n, buffer): On execution of this operation, the server returns to the client *n* bytes of data from the file with name *filename*, from *position* specified

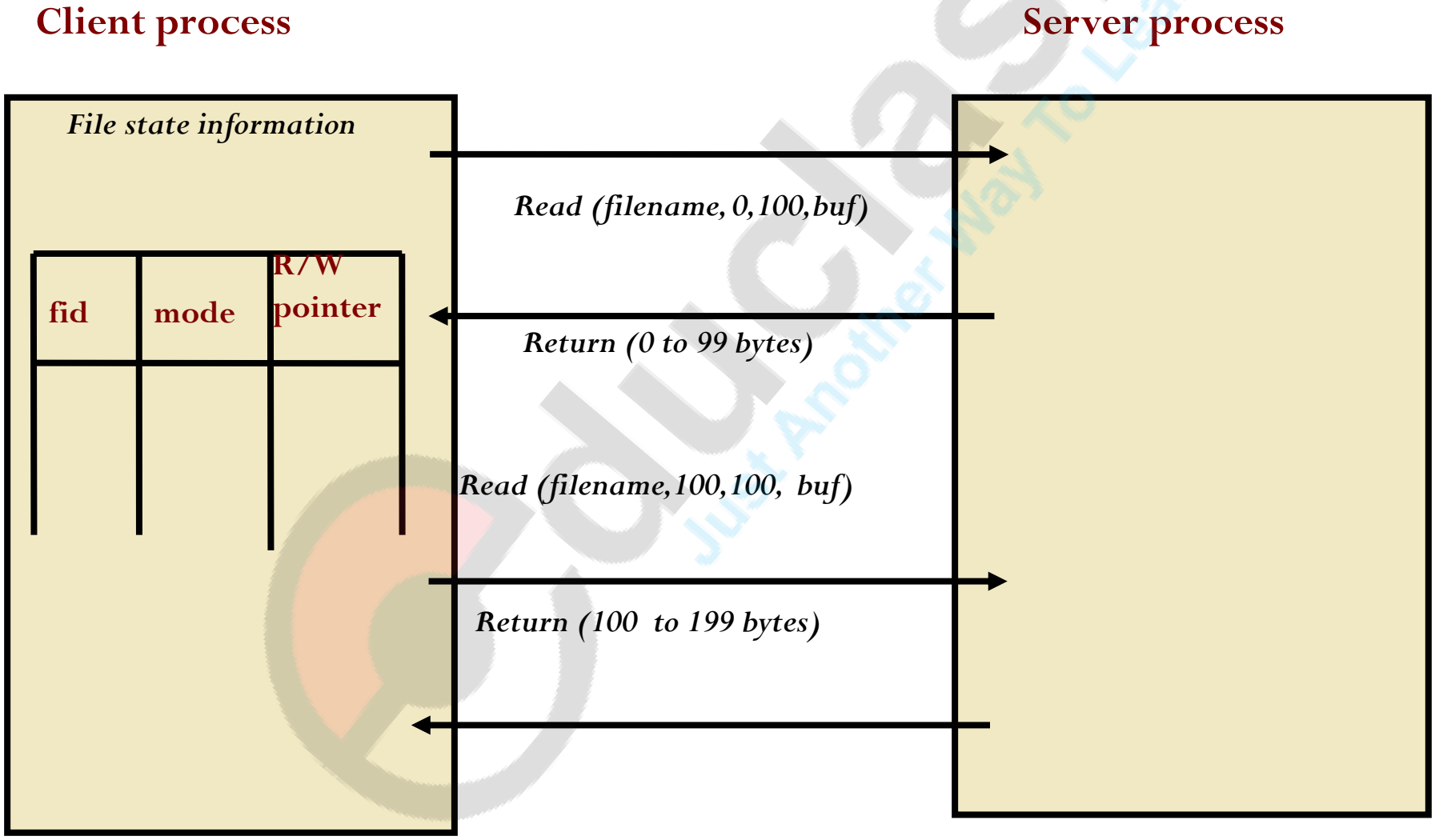
- The returned data is written into the buffer named *buffer*
- The actual no. of bytes read is also returned to client which will be equal to or less than n (i.e. when encountering end of file)

Stateless Server (Cont'd)

Write (filename, position, n, buffer): When server executes this operation, it takes n bytes of data from specified *buffer* and writes it into the file identified by *filename* starting at *position* from and returns actual no of bytes written

- As shown in the figure this file server does not keep track of any information resulting from previous operation
- The figure in the next slide shows two read operations
- Notice that in this case it is the client which maintains file state information

Stateless File Server



Stateful vs Stateless Server

- Stateful Server
  - Provides easier program paradigm
  - Relieves client from keeping track of state information
  - Fewer disk accesses
  - If a file is opened for sequential access, can read ahead the next blocks
  - In general stateful servers are more efficient
- **Then why stateless server ??**
- Stateful Server
  - If server crashes, client can get inconsistent results
  - If client crashes, its information held by server may no longer be valid.
- Stateless Server
  - Better equipped to handle failure: Client only has to retry a request until the server responds and hence crash recovery is very easy
- Both Stateless as well stateful have advantages
- Selection of the right model depends purely on the application

Server Creation Semantics (Cont'd)

- Based on time duration for which RPC servers survive, RPC servers can be classified as
  - Instance-per-call servers
  - Instance-per-transaction / session servers
  - Persistent servers

Instance-per-call servers

- This category of servers exist only for duration of single call
- They are created by RPC Runtime on the server machine only when a call message arrives.
- The server process is deleted after the call execution
- Server is exclusively used by a single client
- Disadvantages
  - They are stateless. Thus intercall state information has to be maintained by either client process or the supporting OS and passed for each call and hence expensive
  - This also leads to loss of data abstraction
  - Overhead in server creation / deletion if same type of service invoked several times successively
  - Not attractive to RPC and hence not commonly used

Instance-per-Session Servers

- Servers of this category exist for the entire session for which a client and a server interact
- Hence the server can maintain intercall state information
- Overhead of server creation / deletion for a client – server session involving a large number of calls is minimized
- In this method, normally there are server managers for each type of service and are registered with the binding agent(binding a client and a server for a type of service, this will be discussed later)
- The client specifies the type of service required to binding agent, which returns address of appropriate server manager to the client
- Client contacts the concerned server manager, which spawns a new server and passes back the address to the client

Instance-per-Session Servers (Cont'd)

- The client now directly interacts with the server for the entire session
- This server is exclusively used by the client for which it is created and is destroyed when the client informs the server manager of the corresponding type that it no longer needs that server
- Server is exclusively used by a single client and hence has to only maintain a single set of state information

Persistent Server

- Persistent server generally remain in existence indefinitely
- Server of earlier two types address only single client, while this can be shared by many clients
- Created and installed before the clients that use them
- Each server registers its service with the binding agent
- When client contacts binding agent for a particular type of service, it returns address of appropriate server to the client
- Binding agent selects a server of that type either arbitrarily or based on some built in policy (like minimum no of clients currently bound to it) and returns address of the selected server to the client
- The client then directly interacts with the server

Persistent Server (Cont'd)

- Note that a persistent server may be simultaneously bound to several clients
- Server interleaves requests from a number of clients, so has to concurrently manage several sets of state information
- If a persistent server is shared by multiple clients, the remote procedure that it offers must be designed so that interleaved or concurrent requests from different clients do not interfere with each other
- Persistent servers may be used for improving the overall performance and reliability of the system
- For this several persistent servers that provide the same type of service may be installed on different machines, to provide either load balancing or some measure of resilience to failure

Parameter-Passing Semantics

- The choice of parameter–passing is crucial to the design of an RPC mechanism. The two choices are call-by-value and call-by-reference
- Call-by-value
  - All parameters are copied into a message that is transmitted from the client to the server through the intervening network
  - This can easily handle integers, counters, small arrays, etc. without any problem
  - Passing larger data types like multidimensional arrays, trees, etc, can consume much time for transmission of data that may not be fully used
  - Hence not suitable for passing parameters involving voluminous data

Parameter-Passing Semantics(cont'd)

- Call by reference
  - Most RPC mechanisms use the call-by-value semantics for passing because the client and the server exist in different address spaces, even different type of machines
  - Hence, passing pointers or passing parameters by *reference* is meaningless
  - Few RPC mechanisms allow passing of parameters by reference in this case the pointer
  - These are usually closed systems, where a single address space is shared by all processes in the system
  - For example distributed systems having distributed shared memory mechanisms can allow passing of parameters by reference

Parameter-Passing Semantics(cont'd)

- In object based system that uses the RPC mechanism for object invocation, the call-by-reference is known as Call-by-object-reference
- In an object-based system, the value of the variable is a reference to an object, so it is this reference(the object name) that is passed in an invocation
- The designers have observed that the use of object-reference mechanism in distributed systems present a potentially serious performance problem because on a remote invocation access by the remote operation to an argument is likely to cause an additional remote invocation

Parameter-Passing Semantics(cont'd)

- Therefore to avoid many remote references, a parameter passing mode called Call-by-move
- In call-by-move, a parameter is passed by reference, as in the call-by-object-reference, but at the time of the call, the parameter object is moved to the destination node (site of callee)
- Following the call, the argument object is either returned to the caller's node or remain at the callee node depending on the state of the argument (these two nodes are called as call-by-visit and call-by-reference respectively)
- Of course the use of call-by-move mode for parameter passing requires that underlying system supports mobile objects that can be moved from one node to another

Call semantics

- In RPC, the caller and the callee processes are possibly located on different nodes
- Hence either one of them might fail independently and later restarted
- Or a failure of communication links between the caller and the callee is also possible
- Hence RPC functioning may be disrupted due to one of the following reasons
  - The call Message gets lost
  - The response message gets lost
  - The caller node crashes and is restarted
  - The callee node crashes and is restarted

Call semantics (Cont'd)

- The call semantics of an RPC system determines how often the remote procedure may be executed in case of failure and under fault conditions
- Possibly or May-Be call Semantics
  - Weakest semantics – no fault tolerance measures
  - Not really appropriate to RPC
  - In order to prevent the caller from waiting indefinitely for a response from the callee, a time out mechanism is used
  - Caller waits for predetermined timeout period & then continues with its execution
  - The semantics does not guarantee anything about the receipt of the message or the procedure execution by the callee
  - This semantics is adequate in a LAN with high probability of successful transmission of messages or where response message is not important

Call semantics (Cont'd)

- Last-One call Semantics
  - This semantics is similar to one discussed in Failure handling class
  - Client retransmits requests repeatedly based on timeouts until a response is received by it
  - Calling of the Remote Procedure by the client, the execution of the procedure by the server, return of the result of the procedure to the client will eventually be repeated until the result is received by the client
  - Hence, result of last executed call is used by the caller
  - Very easily achievable in two node situation
  - Difficult to achieve when nested RPCs that involve more than 2 processors involved

Call semantics (Cont'd)

- e.g., Suppose process P_1 of node N_1 calls procedure F_1 on node N_2, which in turn calls procedure F_2 on node N_3
- While F_2 is being run on N_3, let us say N_1 crashes
- Node N_1 is then restarted with all its processes, and P_1 call to F_1 will be repeated. This will invoke second F_2 on N_3. N_3 is completely unaware of N_1 crash
- N_3 will return the value of two F_2 executions may be out of sequence and hence violating last-one semantics
- Hence, Orphan calls (whose parent (caller) has expired due to node crash) causes problem
- The orphan calls must be terminated before restarting the crashed processes

Call semantics (Cont'd)

- This is done either by waiting for them to complete or by tracking them down and killing them ("orphan extermination")
- It will require manual intervention or special routines in the RPC systems
- Hence other weaker semantics have been proposed for RPC

- Last-of-Many call Semantics
  - Similar to the last-one Semantics except that orphan calls are neglected
  - A simple way to do this is to use call identifiers to uniquely identify each call
  - When a call is repeated, it is assigned a new call identifier

Call semantics (Cont'd)

- Each response message has the corresponding call identifier associated with it
- Caller accepts a response only if its call identifier matches with identifier of most recently repeated call
- Otherwise it ignores the response message

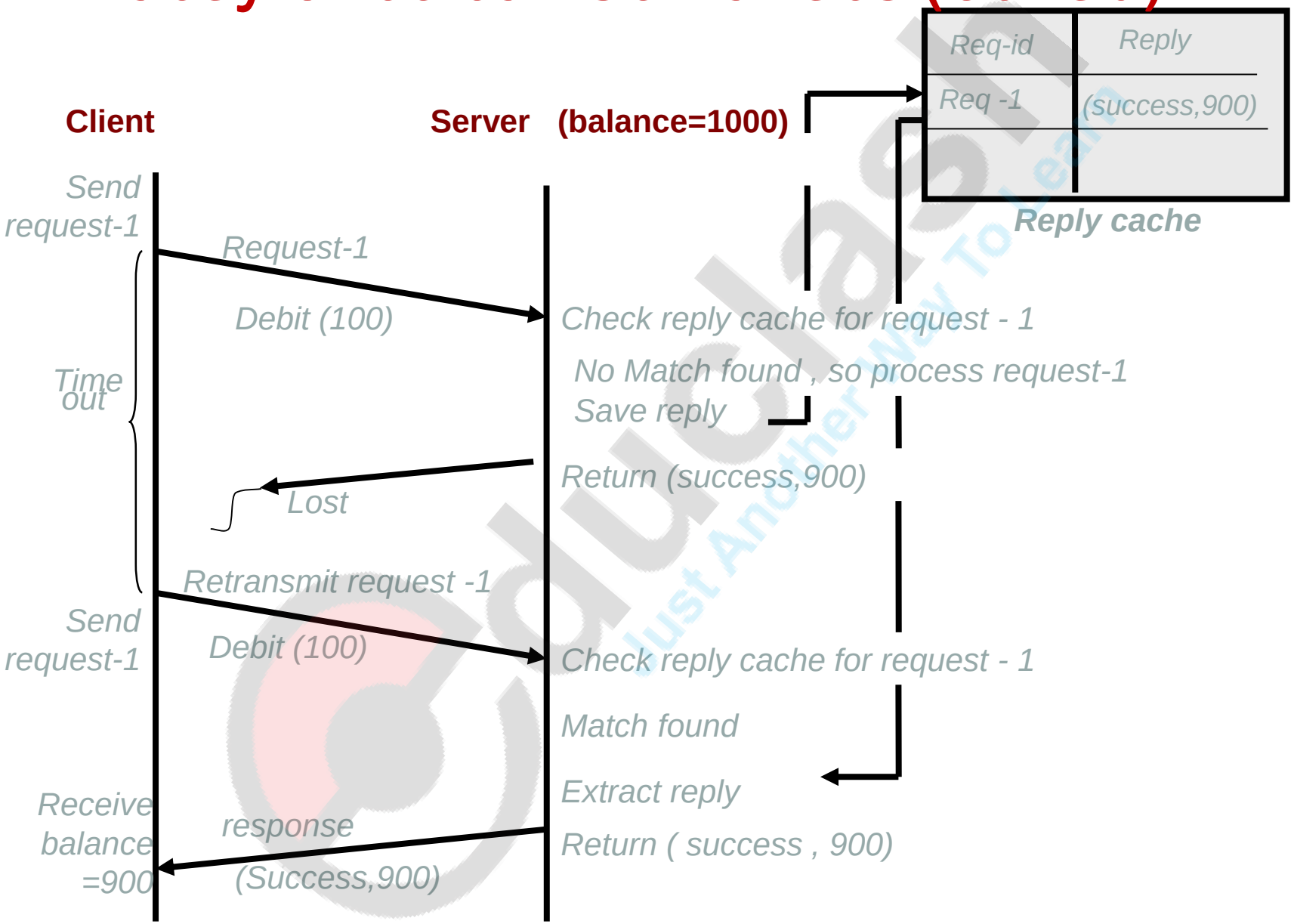
- At-least-once call Semantics
  - This is still weaker than last-of-many call semantics
  - It just guarantees that the call is executed one or more times, but does not specify which results are returned to the caller
  - It can be implemented simply by using timeout–based retransmissions without caring for the orphan calls

Call semantics (Cont'd)

- That is for nested calls , if there are any orphan calls, it takes the result of the first response message, ignoring the others
- It does not bother about if the accepted response is from an orphan or not

- Exactly Once call Semantic
  - It is the strongest and the most desirable call semantics
  - It eliminates the possibility of a procedure being executed more than once no matter how many times a call is retransmitted
  - None of the earlier semantics can guarantee this
  - The main disadvantages of these earlier semantics is that, they force the application programmer to design idempotent interfaces that guarantee that if a procedure is executed more than once with the same parameters, the same results and no side effects will be produced

Exactly Once call Semantics (Cont'd)



- Ques. Which operations are idempotent? If not give semantics to make them idempotent.

1. Read_record (filename, rec_no)
2. Append_record (filename, record)
3. A=sqrt(625)
4. Mpy (integer1, integer2)
5. Increment (integer)
6. Increment (var)
7. Seek (filename, position)
8. Read_next_record (filename)
9. Write_record (filename)
10. Add (sum, 30)
11. T= time(x)

- Append_record (filename, record)
  - N= GetLastRecordNo (Filename)
  - WriteRecordN (Filename, Record, N)

- Increment (var)
  - Var = initial_value
  - Increment (var)

- Read_next_record (filename)
  - Read_record (filename, rec_no)

- Write_record (filename)
  - Write_record (filename,after_record, record)

- Add (sum, 30)
  - sum = initial_value
  - Add (sum, 30)

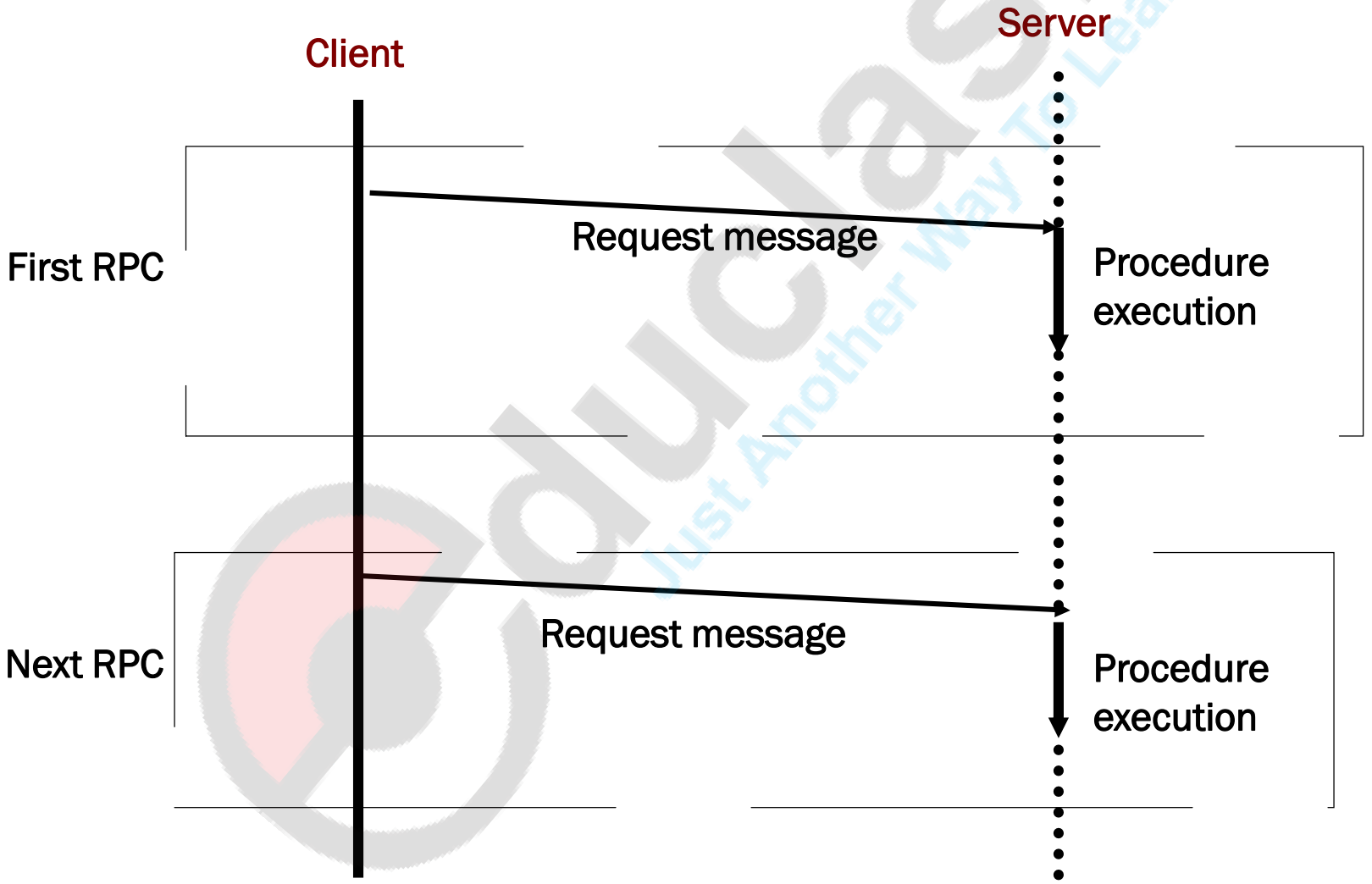
- T= time(x)
  - T = time (Of_particular_event)

Communication protocols for RPC

- Different systems, developed on the basis of RPC, have different IPC requirements
- Based on the needs of different systems, several communication protocols have been proposed for use with RPCs. A Brief description of these are given below
- The Request Protocol
  - This is also known as R(request) protocol. It is used by RPCs in which the called procedure have nothing to return as the result of procedure execution and the client requires no confirmation that the procedure has been executed
  - No acknowledgement/ reply required
  - Client is not blocked on sending request as no need to wait for reply message

Request (R) Protocol

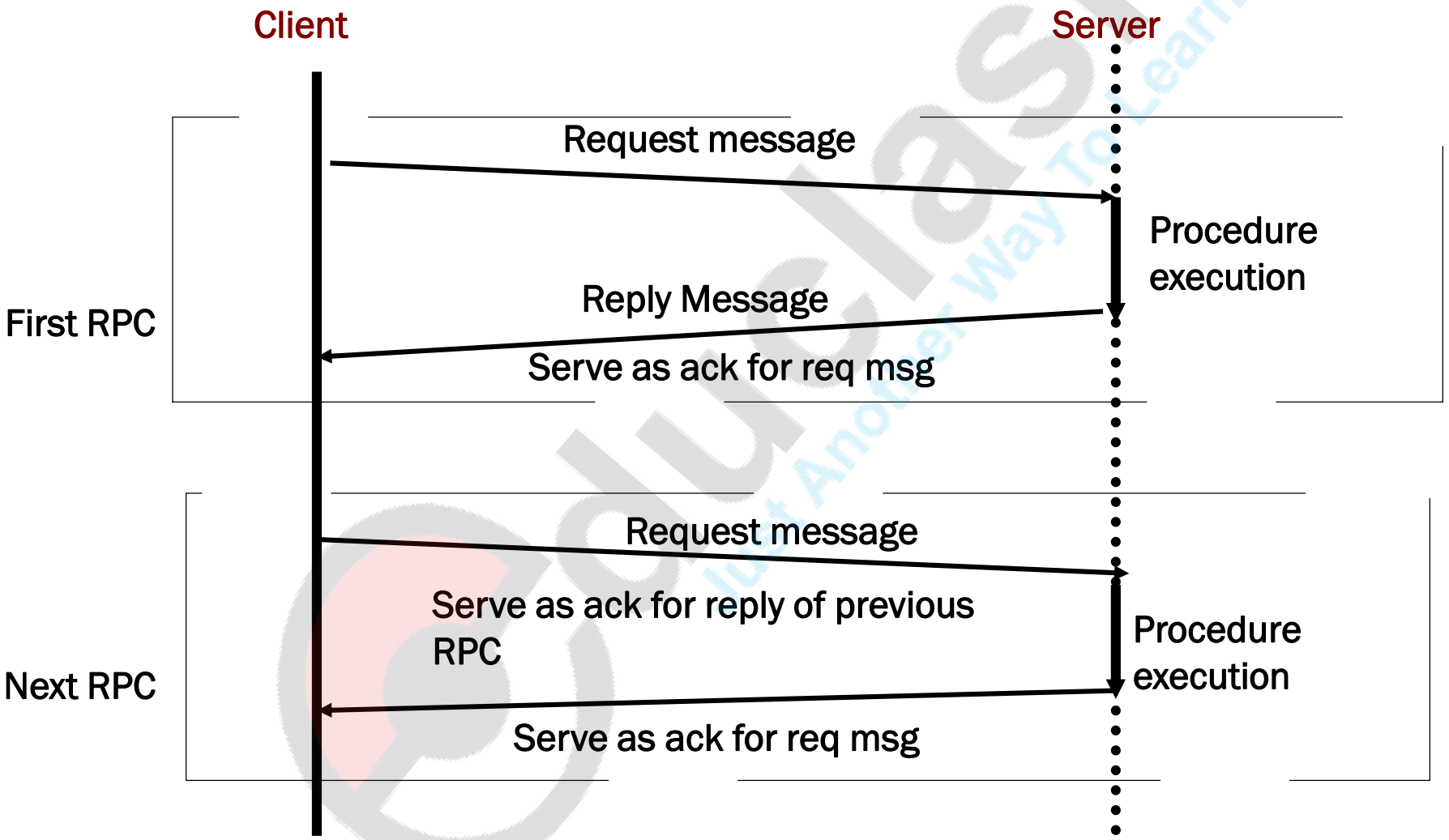
- The Request Protocol



Request/Reply (RR) Protocol

- It is also known as RR protocol
- Used for simple RPC's
- A *Simple* RPC is one in which all arguments & results fit in a single packet buffer and duration of a call and the interval between calls are short (less than the time required to transmit the packet from the client to server)
- Based on the idea of Implicit acknowledgement (servers reply is taken as the acknowledgement of the client's previous request message)
- A subsequent call packet from the client is regarded as an acknowledgement of the servers reply message of the previous call made by the client

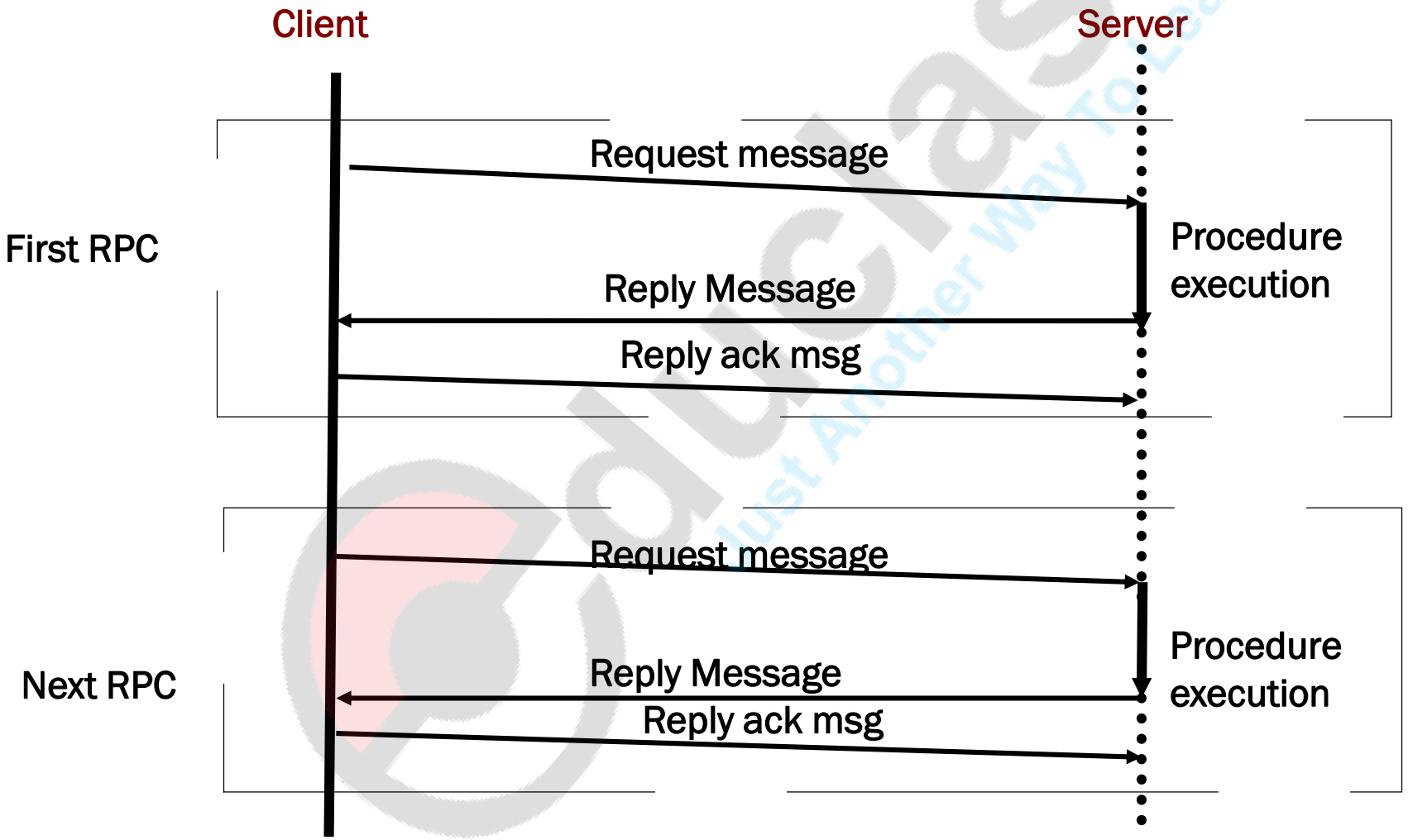
Request/Reply (RR) Protocol (Cont'd)



Request/Reply/Acknowledge - Reply (RRA) Protocol

- This protocol is also known as RRA protocol
- Difficult to maintain reply cache if the server has large number of multiple clients, to implement exactly once call semantics using RR protocol, requires large amount of storage space on the server
- It is much simpler with RRA protocol. Server deletes reply of a request from reply cache once it gets the acknowledgement
- Involves the transmission of 3 messages per call: two from client to server and one from server to client
- There is possibility that acknowledgement is lost hence RRA protocol implementation requires that All three messages have same unique message identifier

Request/Reply/Acknowledge - Reply (RRA) Protocol



Complicated RPC

- The following two types of RPCs are known as complicated RPCs
  - RPCs involving long-duration calls or large gaps between calls
  - RPCs involving arguments/results that are too long to fit in a single-datagram packet
- Different protocols are used for handling these two types of complicated RPCs
- RPCs involving long-duration calls or large gaps between calls may use one of the following methods
  - Periodic probing of server by the client
    - The client sends periodic probe packet to the server, which the server is expected to acknowledge → helps in detecting server failure or communication link failure

Complicated RPC (Cont'd)

- The message identifier of the original request message is included in each probe packet → sever can notify the client if it has not received the original request message as a reply to the probe packet
- On receipt of such notification, client can resend the original request

- Periodic generation of an acknowledgment by the server is done if duration of call is long
  - If the server is not able to generate the next packet significantly sooner than the expected transmission interval, it spontaneously generates an ACK and sends it to the client
  - If the reply or ACK is not received within timeout, the system crash or communication failure is assumed by the client and the concerned user is informed of the exception condition

Complicated RPC (Cont'd)

- RPCs Involving Long Messages
  - In some RPCs arguments or results are too large to fit into a single datagram packet
  - For example File read / write operation, involves transfer of large chunk of data
  - A simple way is to use Several physical RPC's for one logical RPC
  - This solution is inefficient due to the fixed amount of overhead involved with each RPC irrespective of the amount of data sent
  - Another method of handling complicated RPCs is to use multidatagram messages in which a long argument / result is fragmented and transmitted in multiple packets

Client-Server Binding

- It is necessity of a client (actually client stub) to know the location of the server before RPC can take place between them
- The process by which a client (importer) becomes associated with a server (exporter) so that calls can take place is known as binding
- From the point of view of the application, the model of binding is that servers "export" operations to register their willingness to provide service and clients' "import" operations, asking the RPC Runtime system to locate the server and establish any state that may be needed at each end
- This is a two way process as willingness of both server and the client is necessary to establish the binding

Client-Server Binding (Cont'd)

- The client-server binding requires proper handling of several issues :
  - How does client specify server to which it wants to bound with?
  - How does the binding process locate specified server?
  - When is it proper to bind client with server?
  - Is it possible for a client to change the binding during execution?
  - Can client be simultaneously bound with multiple servers that provide same service?
- Sever Naming
  - The client specifies the server with which it wants to communicate.
  - For RPC, it has been proposed to use the interface names for this purpose

Sever Naming

- An interface name has two parts: type and an instance (optional)
- Type specifies the interface itself and instance specifies a server providing the services within that interface
- e.g. interface of type f*ile_server*, may have several instances of servers providing file service
- Interface name is unique identifier of server

Server Locating

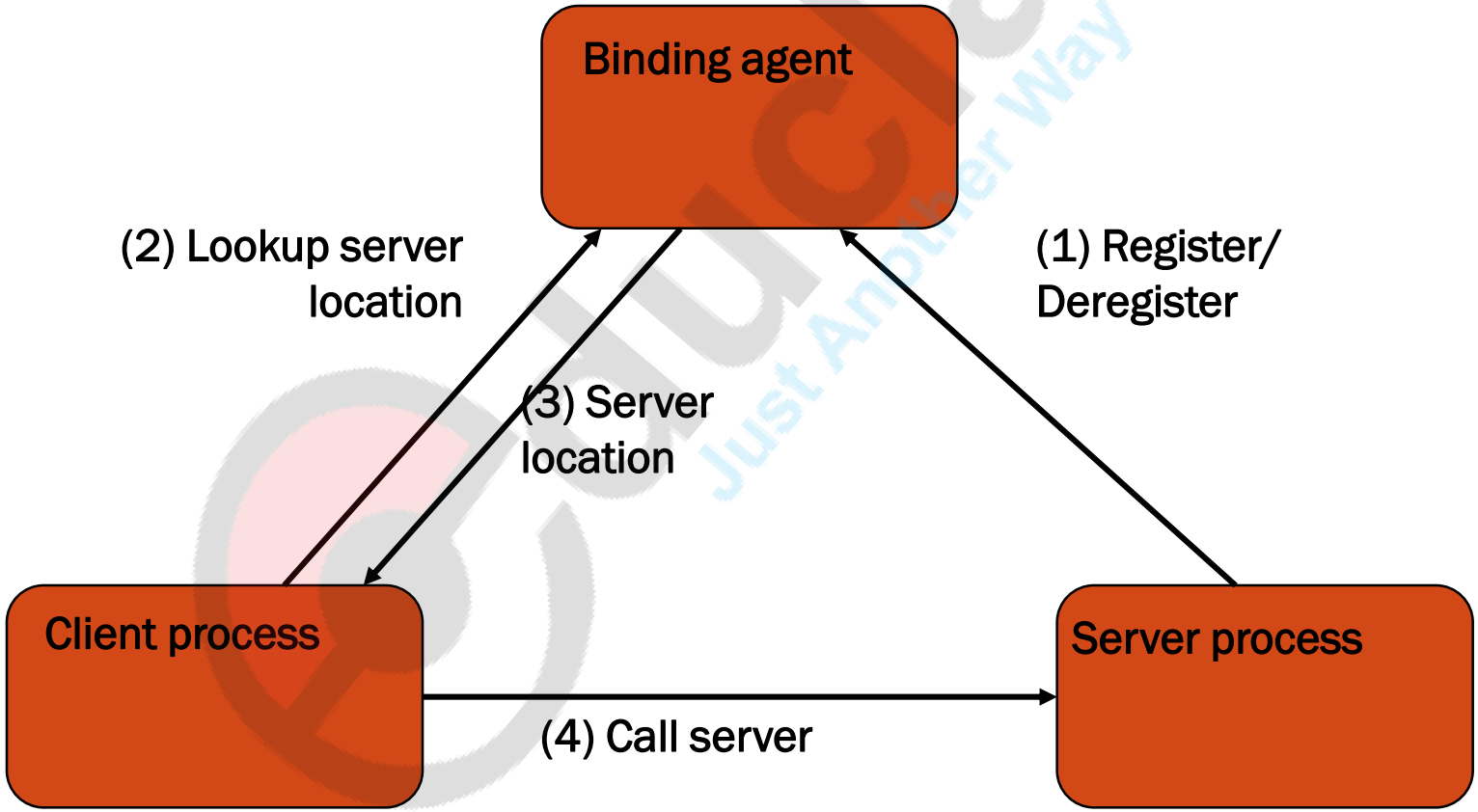
- RPC package only dictates means by which an importer uses the interface name to locate an exporter
- The interface name of a server is a unique identifier
- How to locate specified server? Two commonly used methods:
- Broadcasting –
  - A message to locate the desired server is broadcasted to all the nodes from the client node
  - Nodes in which desired server is located return a response message
  - Since there can be more than one server, client will receive multiple replies
  - Normally the first response is given to the clients process and others are discarded
  - The method is easy and simple to implement
  - Suitable for small networks; expensive for large networks

Binding agent

- Binding agent
  - Binding agent is a Name server which is used to bind a client to a server by providing the client with location information of the desired server
  - Binding agent maintains a binding table, which is a mapping of interface name and its locations
  - All servers register themselves with the binding agent as a part of their initialization process
  - To register with binding agent, a server gives the binder its identification information and handle used to locate it
  - Binding agent's location is known to all nodes. It is usually a fixed address or message is broadcasted to all to locate binding agent when a node is booted

Binding agent (Cont'd)

- In either case, when binding agent is relocated, a message is sent to all nodes informing the new location of the binding agent



Binding agent (Cont'd)

- Binding agent interface has three primitives: (a) *register* & (b) *de-register* is used by a server to register or deregister itself with the binding agent, and (c) *lookup* for client to locate a server
- It has several advantages & disadvantages
  - It can support multiple servers with the same interface name, so that any of the available server may be used to service a client and hence a degree of fault tolerance
  - Better load balancing with multiple servers providing the same service
  - Give list of users permitted to use a particular server, in which case binding agent may refuse to bind certain clients to certain servers
  - Can handle large networks
  - Need to be robust against failures and not a Performance bottleneck
  - How to maintain consistency in replicating binding agents is a question

Binding Time

- When to bind client with server? A client may be bound at compile time, Link time or at call time
- Compile Time
  - Client and server modules are programmed as if they were expected to be linked together. e.g., Server's network address can be compiled into the client code and then found by looking up the server's name in a file i.e. specify server name in program itself
  - Inflexible – server moves, server is replicated, server program versions change then recompile client programs
  - High overhead but fast, useful in an application whose configuration is expected to remain static for long time

Binding Time (Cont'd)

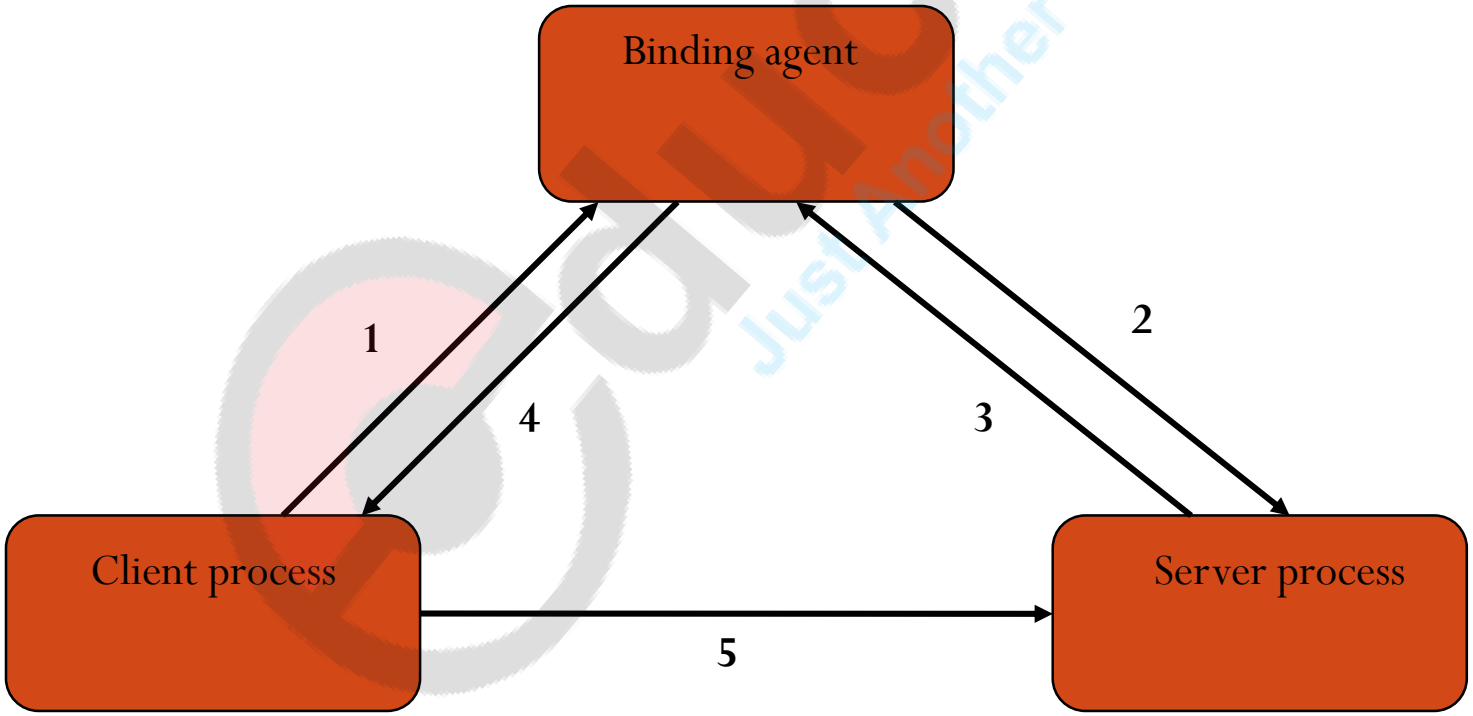
- Link Time
  - A server process exports its service by registering itself with the binding agent as part of its initialization process
  - A client makes an import request to Binding Agent before making a call
  - The Binding Agent binds Client and Server by returning server's handle to the client
  - The servers handle is cached by the client, to avoid contacting the binding agent for subsequent calls for the same server
  - Advantageous for those situations in which a client calls a server several times once it is bound to it

Binding Time (Cont'd)

- Call Time
  - A client is bound to a server , when it calls the server for the first time during its execution
  - A commonly used approach for binding at call time is the indirect call method
  - Indirect call method is shown in the next slide
  - The basic advantage of this method is that client need to go to the binding agent only once
  - Then onward, it can use handle to call target server directly

Binding by Indirect Call

1. Client process passes server's interface name & arguments to binding agent
2. Binding agent sends an RPC call message to server including client's parameters
3. Server returns result to binding agent
4. Binding agent returns result to client with server's handle
5. Client calls server directly subsequently.



Binding Time (Cont'd)

- Changing Bindings
  - Can binding change during execution?
  - The client or server of a connection may like to change the binding at some instance of time due to some change in the system state
  - Binding is altered by concerned server, it ensures that any state data held by server is no longer needed or can be duplicated in the replacement server. Ex. State of open files transferred from old to new file server
- Multiple Simultaneous Binding
  - Can client bind with multiple servers that provide same service?
  - Client can be bound simultaneously to all or multiple servers of the same type. Ex. Client updates file replicated at many nodes at the same time
  - Client uses multicast communication

Exception Handling

- We have seen that, when a remote procedure can not be executed successfully the server reports an error in the reply message
- An RPC also fails when the server can not be contacted
- Hence RPC should have an effective exception handling mechanism for reporting such failures to clients
- One approach is to define an exception condition for each possible error type and have corresponding exception raised when an error of that type occurs
- On occurrence of exception, exception handling procedure is called & executed automatically in client's environment
- This approach can be used with those programming languages that support exception handling constructs
- If the languages do not support, then RPC system generally use the methods provided by conventional operating system

Security

- Some of the RPC implementation include facilities for client and server authentication as well as for providing encryption-based security for calls
- For example the callers are given a guarantee of the identity of callee and vice versa
- For providing end to end encryption of calls, the federal data encryption standard (DES) is used
- Encryption though very useful and powerful tool, it is very expensive both for processor as well as for communication, overhead are concerned
- Hence while designing an application user should consider the following security issues related with the communication messages
  - Is the authentication of the server by the client required?
  - Is the authentication of the client by the server required, when the result is returned?
  - Is it all right if the arguments and results of the RPC are accessible to users other than the caller and callee?

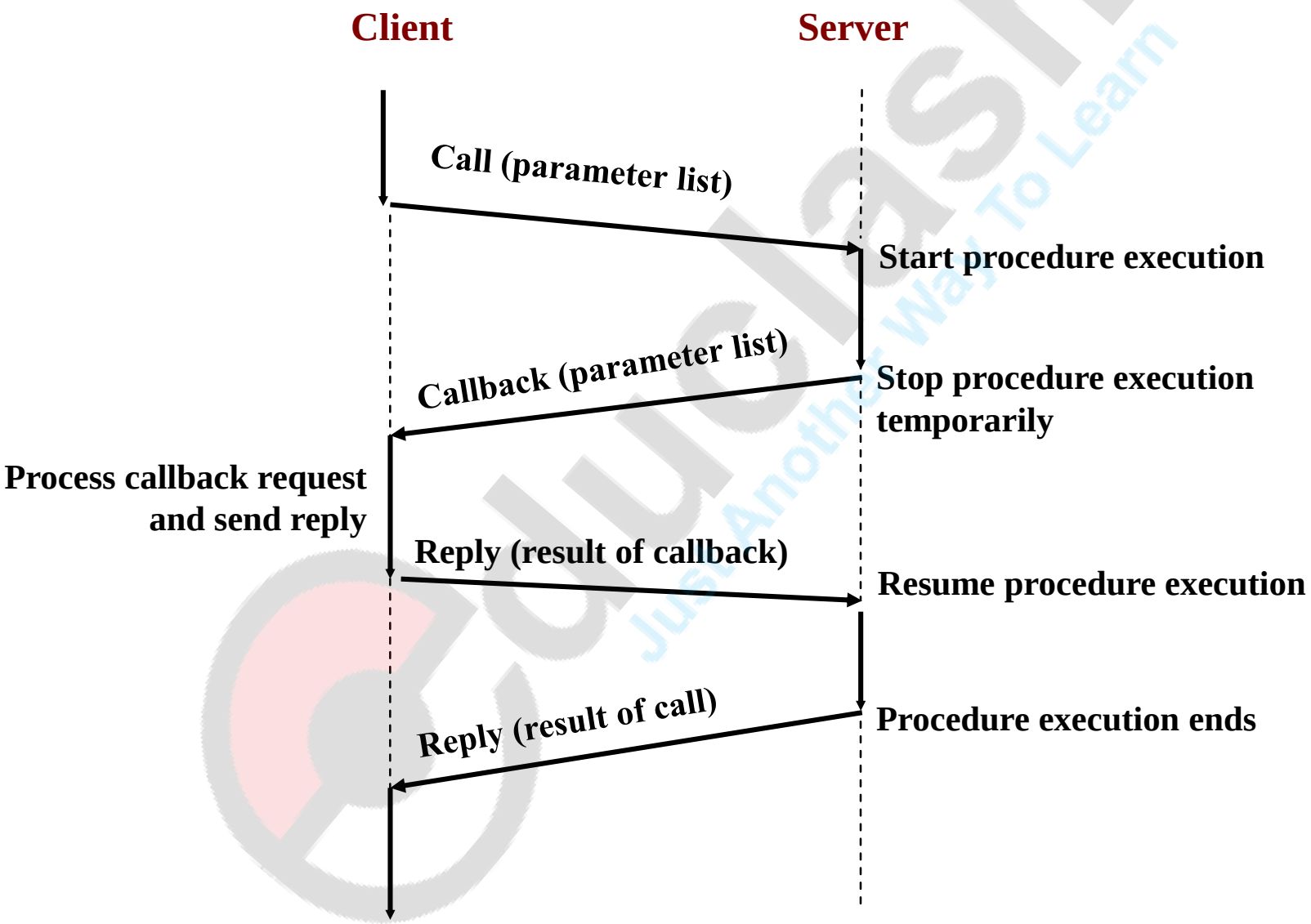
Some Special Types of RPC

- Callback RPC
  - In the usual RPC protocol, the caller and callee processes have a client-server relationship
  - The callback RPC on the other hand facilitates peer-to-peer paradigm among the participating processes
  - It allows a process to be both client as well as server
  - It is very useful in certain class of distributed applications
  - For example a remotely processed interactive application that need user input from time to time for further processing, require this facility
  - As shown in the slide further, client process makes RPC to the concerned server process and during procedure execution for the client, the server process makes a callback RPC to client process

Callback RPC

- The client takes appropriate action based on server request and reply to the callback RPC to the server process
- On receipt of the reply the server resumes the execution
- Specifically to provide callback RPC facility, the following are necessary
  - Providing the server with the clients handle
  - Making the client process wait for the callback RPC
  - Handling callback deadlocks

Callback RPC



Essentials of Callback RPC

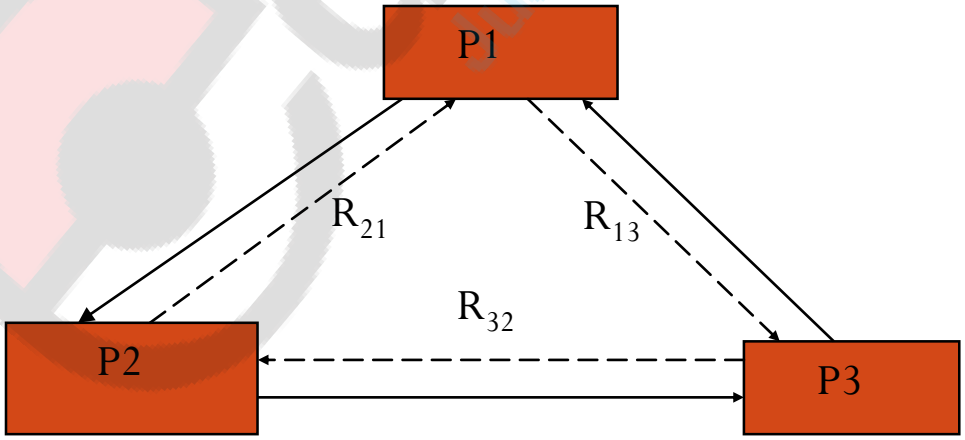
- The server need to have the clients handle to call the client back
- The handle uniquely identifies the client process and provides enough information to server for making the call to it
  - The client process uses a transient program number for the callback service and registers itself with Binding Agent, which is then sent as a part of RPC request to the server
  - To make a callback RPC, the server initiates a normal RPC request to the client using the given program number
  - Client may also send its handle to server such as port number.

Essentials of Callback RPC (Cont'd)

- Making the client process wait for the callback RPC
  - The client process must be waiting for the callback so that it can process the incoming RPC request from the server
  - It should not be confused with reply to the RPC call made by the client
  - To wait for the callback, client process makes a call to svc-routine, which waits for server request & then dispatches request to appropriate procedure

Handling call back deadlocks

- A callback deadlock can occur due to the interdependencies of processes
- For example process P_1 makes an RPC to P_2 and waits for reply
- In the mean time P_2 makes RPC to P_3 and P_3 makes a RPC call to P_1
- But P_1 can not process P_3 request until its request to P_2 is satisfied and P_2 cannot process P_1 until its request to P_3 is satisfied and P_3 can not process P_2 request until its request P_1 is satisfied
- The result is none of the Processes can have their request satisfied and hence will wait indefinitely. In effect we have RPC callback deadlock
- There are various methods to address the deadlocks



Broadcast RPC

- RPC based IPC is normally one-to-one type involving a single client and a single server process
- However a good DCS needs broadcast and multicast communication
- In broadcast RPC, a client's request is broadcast on the network and is processed by all the servers that have the concerned procedure for processing that request
- The client waits for & receives many replies depending on degree of reliability desired
  - Client uses special broadcast primitive indicating that the message has to be broadcasted
  - Request is sent to binding agent, which forwards request to all servers registered with it

Broadcast RPC (Cont'd)

- In this case only those services registered with the binding agent are accessible via the broadcast RPC mechanism
- The second method is to declare broadcast ports
- A network port of each node is connected to a broadcast port
- Client of broadcast RPC first obtains a binding for a broadcast port & then broadcasts RPC message by sending RPC message to this port
- The same procedure can be used for multicast RPC in which the RPC message is sent only to subset of the available servers

Batch-Mode RPC

- Used to queue separate RPC requests in a transmission buffer on the client side and then send them over the network in one batch to the server
  - It reduces overhead of sending each RPC independently to server and waiting for response for each request
  - Applications requiring higher call rates(50 -100 remote calls per sec) may not be feasible with most RPC implementation
  - But become feasible with the use of batch-mode RPC
- Used when client has many RPC requests to send to a server & client does not need any reply for a sequence of requests
- The request are queued on the client side and the entire queue of requests flushed(sent to server) when :-
  - A predetermined interval elapses

Batch-Mode RPC (Cont'd)

- A predetermined number of request have been queued
- Amount of batched data exceeds the buffer size
- A call is made to one of the server's procedures for which a result is expected

Lightweight RPC (LRPC)

- In MicroKernal, various components have to use some form of IPC to communicate with each other, hence becomes less efficient when compared to monolithic kernel
- But it is Simple & flexible

- In microkernel approach , the communication traffic in OS are of two types:
  - Cross domain – communication between domains on the same machine
  - Cross machine - communication between domains located on separate machines.
- Lightweight RPC is designed and optimized for cross- domain communications
- Though normal RPC can be used for this purpose, it unnecessarily involves high communication cost

Simple Control Transfer

- In cross domain communication small-simple parameters are involved, overhead for the heavyweight communication machinery (message buffer, marshaling, message transfer, access validation, scheduling, dispatch, etc) is still paid for, if normal RPC is used
- Simple Control Transfer:
  - When ever possible LRPC uses control transfer mechanism that is simpler than that used in conventional RPC systems
  - LRPC uses a special threads scheduling mechanism called **handsoff scheduling** for direct context switch from client thread to the server thread of an LRPC
  - In this mechanism, when a client calls a server procedure, it provides the server with an argument stack and its own thread of execution

Simple Control Transfer (Cont'd)

- The call causes a trap to the kernel
- The kernel validates the caller, creates a call linkage and dispatches the clients thread directly to the server domain, causing the server to start execution immediately
- When the called procedure completes, control and results return through the kernel back to the point of the client's call
- In contrast, in conventional RPC implementation, context switching between the client and server threads of an RPC is a slow process

Simple Data Transfer

- In RPC arguments and results need to be passed between the client and server domains in the form of messages
- Traditional RPC requires data to be copied four times

  Client stack→ RPC message → kernel domain → Server domain → server stack as shown in the figure in the next slide
- LRPC requires data to be copied only once:
  - From client stub's stack to shared Argument stack (pre-allocated shared argument stack) which can be accessed by both client and the server
  - The server procedure can access the data from the argument stack
  - This provides a private channel between the client and server
  - LRPC uses a simple model of control and data transfer, facilitating the generation of highly optimized stubs

Simple Data Transfer (Cont'd)

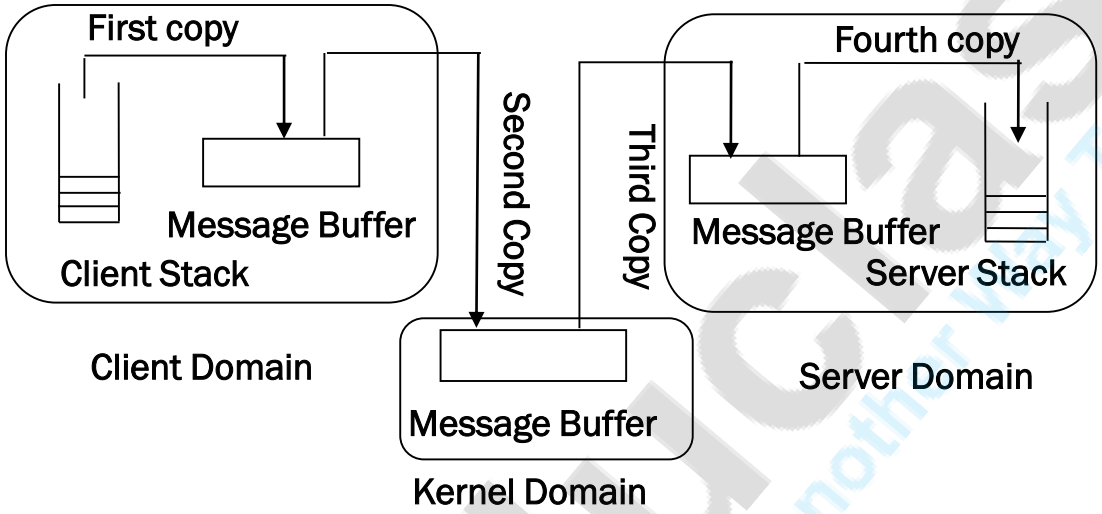


Figure (a)

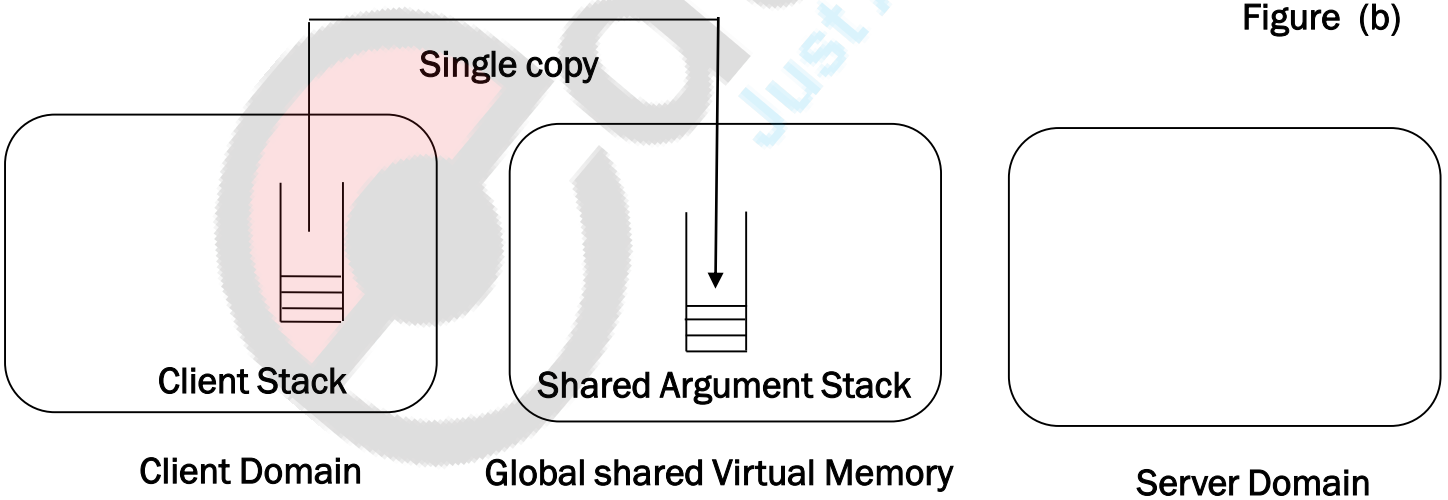


Figure (b)

Simple Stubs

- Every procedure has a call stub in the client's domain and an entry stub in server's domain.
- To reduce the cost of interlayer crossings, LRPC stubs blur the boundaries between protocol layers
- On transferring control, kernel associates execution stacks with initial call frame expected by called server's procedure & directly invokes the corresponding entry in server's domain
- No intermediate message examination or dispatching is done and Server stub starts executing procedure by directly branching to procedure's first instruction

Optimizations for Better Performance

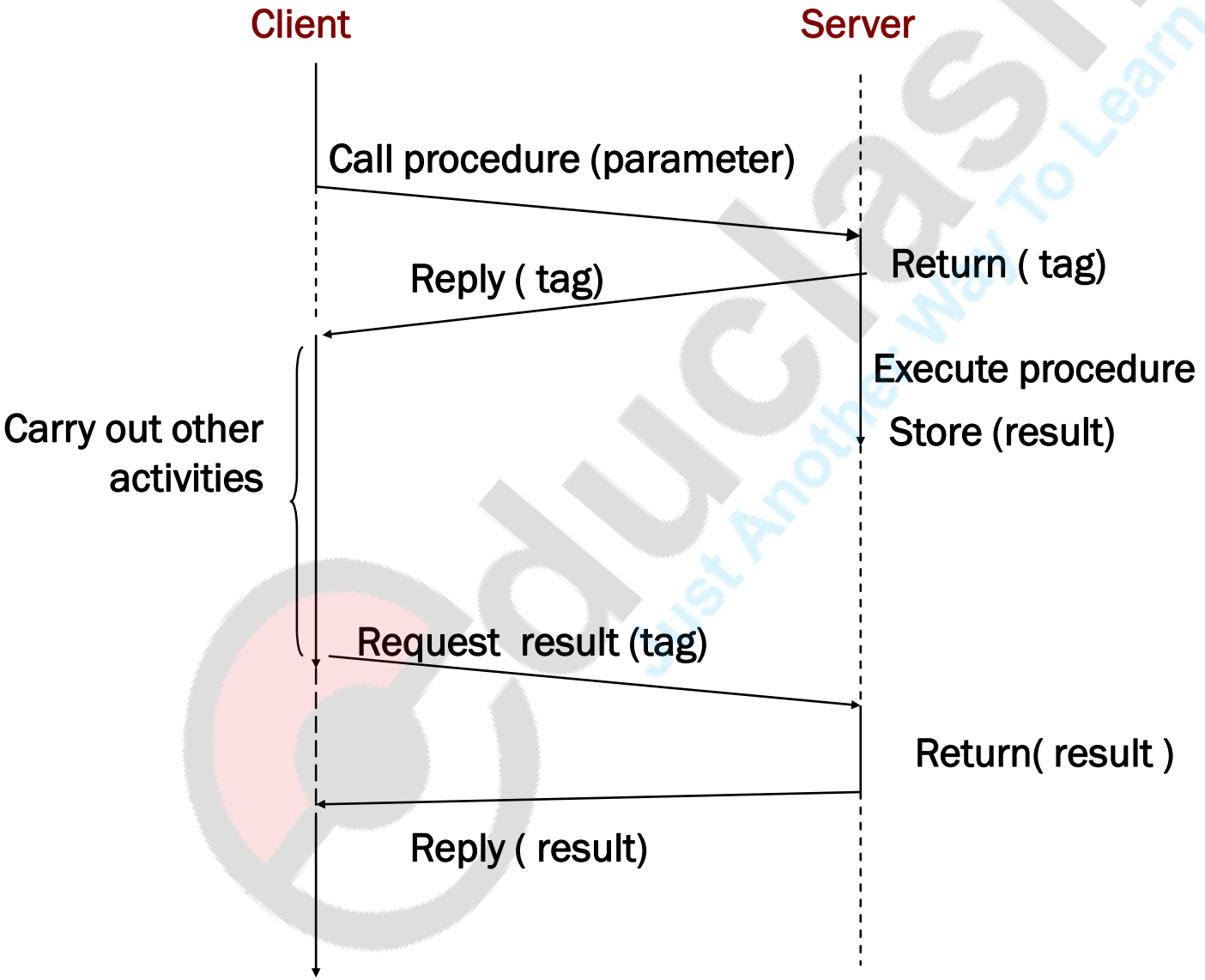
Concurrent access to multiple servers

- The basic issue in the design of a distributed application is performance
- It is possible to improve performance further by considering other methods
- Let us consider some of other aspects of performance improvement
  - Use of threads in implementation of a client process where each thread can independently make remote procedure calls to different servers

Concurrent access to multiple servers (Cont'd)

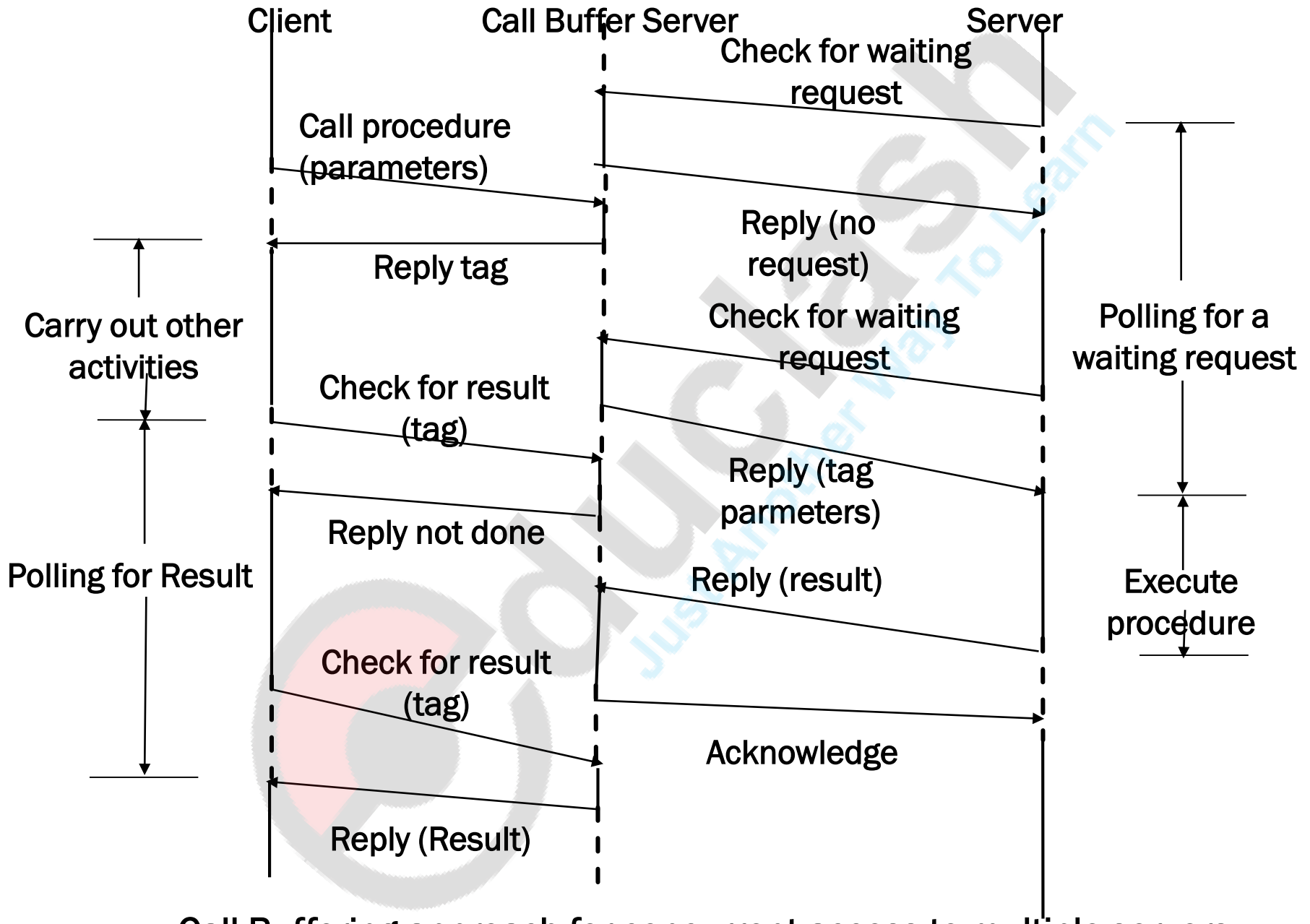
- Another approach is Early reply approach: As shown in the figure in the next slide, in this method a Call split into two separate RPC calls, one passing the parameter to the server & other requesting the result
  - In reply to the first call, server returns a tag that is sent back with the second call to match the call with the result
  - The client decides the time delay between the two calls and carries out other activities during this period
  - But one problem with this method is, that the server has to hold the result of a call until the client makes a request for it
  - Hence, if the request for the result is delayed, it may cause congestion or unnecessary overhead at the server

Early Reply Approach



Concurrent access to multiple servers (Cont'd)

- The third approach is known as call buffering approach using a call buffer server
  - The client and server do not interact directly with each other but interact indirectly through a buffer server
  - To make RPC call, a client sends its call request to the call buffer server, where the request parameters together with the name of the server and the client are buffered
  - The client can then perform other activities until it needs the result of the RPC call, then it starts polling the call buffer server to see if the result of the call is available, if so it recovers the result
  - When the server is free, it periodically polls to server buffer to see if there are any pending calls to it
  - The figure in the next slide shows the sequence of events pictorially



Call Buffering approach for concurrent access to multiple servers

Serving Multiple Requests Simultaneously

- Another aspect of RPC we need to consider is the different types of delays in RPC system
- Following types of delays are commonly encountered in RPC systems:
  - Delay caused while a server waits for a resource that is temporarily unavailable (e.g. shared file that is locked by some other process)
  - The server calls a remote function that involves a considerable amount of computation to complete or involves a considerable transmission delay
- For better performance, good RPC implementations must have mechanisms to allow servers process other requests, while waiting for some operation to complete

Reducing Per call Workload of Servers

- One of the approaches to achieve this is to use a Multiple threaded server with dynamic threads creation facility
- Keep the client requests short, work per request low
- One way of achieving this improvement is to use stateless servers & let clients keep track of progression of their requests sent to servers
- This is meaningful as client is incharge of the flow of information between the client and server

Reply Caching of Idempotent Remote Procedures

- We have already seen the use of Reply Cache for exactly-once semantics
- Reply cache can also be associated with idempotent remote procedures for improving a server's performance when it is heavily loaded
- When client requests to a server arrive at a rate faster than the server can process the requests, a backlog develops and eventually client requests start timing out and clients resend the requests, making the problem worst
- In such cases reply cache helps because the server has to process the request only once
- If a server sends a request, it just sends the cached reply

Proper Selection of Timeout Values

- To deal with failures, timeout based retransmissions are necessary in distributed applications
- Important issue how to select timeout value; short time out will cause frequent unnecessary retransmissions and too large time out makes unnecessary long waits
- It will depend on factors like server load, network routing, network congestion
- Repeated retry transmission by the client make the problem worst
- One method to handle this situation is to use some sort of back-off strategy of exponentially increasing/decreasing timeout values

Proper Design of RPC Protocol Specification

- For better performance, the protocol specification of an RPC system must be designed so as to aim at minimizing amount of data to be sent and its frequency
- Using standard existing protocols may not be good option; e.g. IP suite (to which TCP/IP and UDP/IP belong) have in total 13 fields of which only 3 are useful for RPC (source, destination address and packet length)
- Hence for better performance RPC has to use specially designed protocols
- Of course, new protocol has to be designed from scratch, implemented, tested and embedded into existing systems; i.e. good amount of overhead

Sun RPC

Introduction

- Sun RPC uses automatic stub generation and also provides flexibility to write stubs manually.
- An application's interface definitions written in an IDL called RPCL, an extension of Sun XDR.
- RPCL uses the Rpcgen compiler, which generates the following:
  - A header file containing definitions of common constants and types defined in IDL. Also procedures for marshalling and unmarshalling are automatically generated.
  - An XDR filter file contains XDR marshalling and unmarshalling procedures.
  - A client stub file containing one stub procedure for each procedure defined in IDL.
  - A server stub file contains
    - the main routine(creates transport handles and registers the service),
    - the dispatch routine(dispatch incoming remote procedure calls to remote procedures)
    - and also a stub procedure for each procedure defined in Interface Definition file.

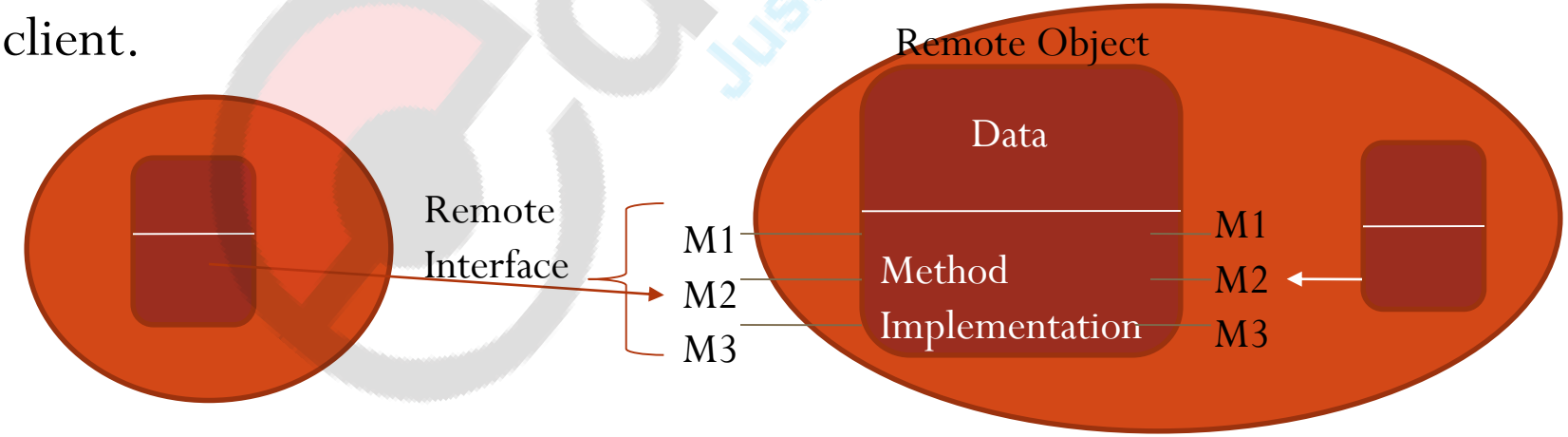
- RPC application is created using the files generated by the Rpcgen compiler.
- The steps involved are:
  - The application programmer manually writes the client program as well as the server program.
  - The client program file is compiled to get a client object file.
  - The server program file is compiled to get a server object file.
  - The client stub file and XDR filter are compiled to get a client stub object file.
  - The server stub file and XDR filter are compiled to get a server stub object file.
  - Client object file, client stub object file, and client stub RPC runtime library are linked together to get a client executable file.
  - Server object file, server stub object file, and server stub RPC runtime library are linked together to get a server executable file.

- Marshalling procedures handles differences in data representation.
- RPC runtime library has procedures for marshalling integers of all sizes, characters, strings, reals and enumerated types.
- Sun RPC supports 'at-least-once' semantics.
- Each node here uses local binding agent called portmapper.
- Portmapper maintains a database mapping of local services and their port numbers.
- The server, on startup, registers its program number, version number, and port number with the local portmapper.
- The server side error handling procedures sends reply to the client indicating the detected error.
- The client side error handling provides flexibility to choose the mechanism.
- Sun RPC supports access rights authentication and DES .

- Sun RPC also supports asynchronous, callback, broadcast and batch mode RPC.
- Shortcomings of Sun RPC:
  - No location transparency
  - Transport dependency (mostly TCP and UDP)
  - No support for network-binding services.
  - At-least-once semantics may not be appropriate for some applications.

Remote Method Invocation (RMI)

- Using this concept, objects in different procedures can communicate with each other
- Distributed object concepts
  - Every object encapsulates data and methods.
  - Any object invokes other objects by invoking its methods.
  - In case of distributed objects, the server manages the objects and clients invoke the methods.
  - The RMI technique sends the request as a message to the server which executes the method of the object and returns the result message to the client.



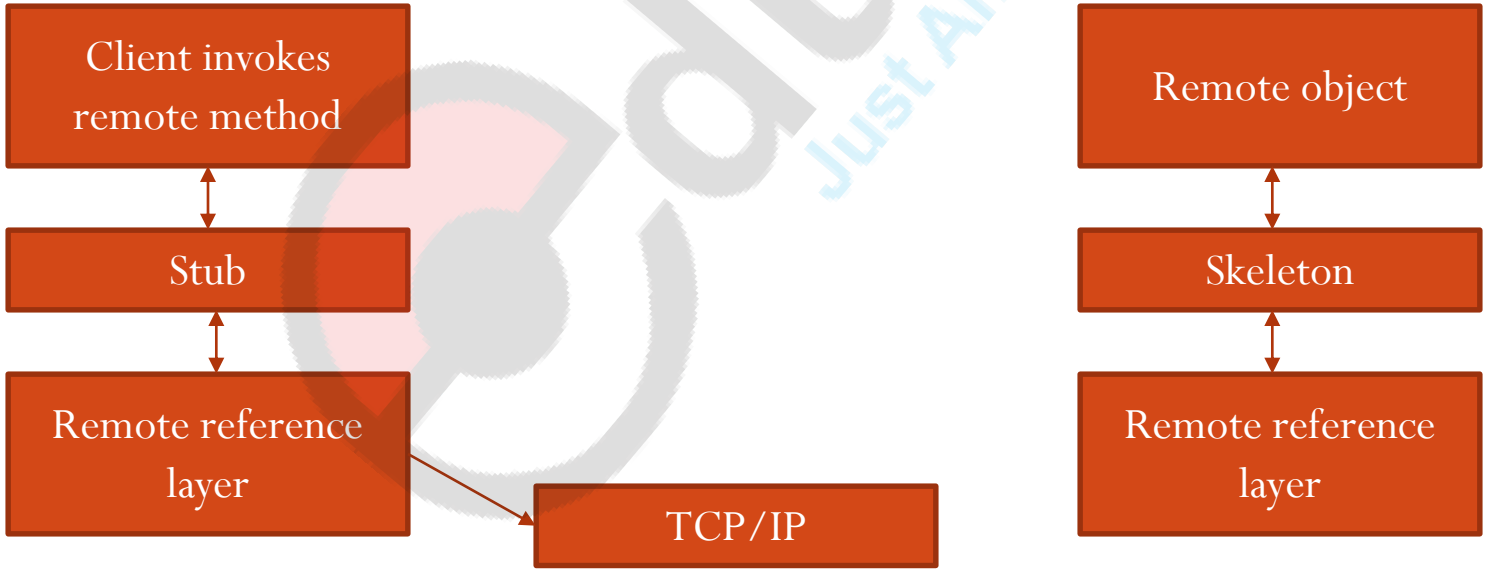
RMI Implementation

- Design Issues in RMI
  - RMI Invocation Semantics
    - Maybe Semantics
    - At-least-once Semantics
    - At-most-once Semantics
  - Level of Transparency – This involves hiding the internal RMI processes, such as marshalling, message passing, locating and contacting the remote object for the client.

Fault-Tolerance Measures			
Retransmit request message	**Duplicate filtering**	**Re-execute procedure of retransmit reply**	**Invocation Semantics**
No	Not Applicable	Not Applicable	Maybe
Yes	No	Re-execute procedure	At-least-once
Yes	Yes	Retransmit reply	At-most-once

RMI Flow

- Client invokes method on the remote object, which is sent to the client stub.
- Remote reference layer in RMI converts the method into a message and sends it over TCP/IP network.
- RMI message is sent to the remote reference layer at the server.
- Now the server skeleton converts this message into a remote object and RMI is invoked.
- Once the RMI is executed, the object results are transferred back through the same path.

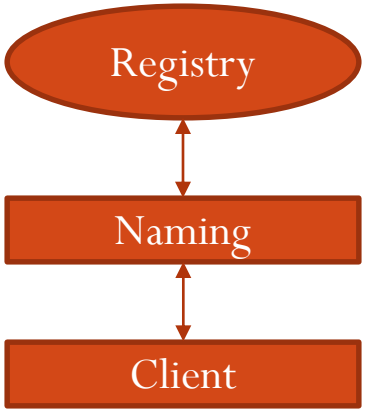


RMI Execution

- Communication Module – the client and the server processes form part of communication module, which uses RR protocol.
- Remote reference module – it is responsible for translating between local and remote object references and creating remote object reference. It also does marshalling and unmarshalling.
- RMI Software – this layer consists of the following:
  - Proxy – It makes RMI transparent to the client and forwards the message to the remote object. It also marshals/unmarshals the result and sends/receives messages from the client. There is exactly one proxy for each remote reference.
  - Dispatcher – this unit receives the request from the communication module, selects the appropriate method in the skeleton and passes the request message.

- Skeleton – It implements the method in the remote interface. It unmarshals the arguments in the request message and invokes the corresponding method in the remote object.
- Server and Client programs – the server program contains classes for dispatcher , skeleton and the remote objects it supports. The client program contains classes of processes for the entire remote object, which it will invoke.
- The Binder – Let us take an example to explain the concept of binders, an object A requests remote object reference for object B. The binder is actually a service , which maintains a table of textual names to the remote object-referenced. Systems use this service to look up remote object references.

Locating Remote Objects



Types of Objects

- Objects are classified into two main classes based on when they are bound and how long they exist.
- Based on the time of binding, objects are classified into :
  - Runtime objects – object binding at runtime.
  - Compile-time objects – object binding at compile time.
- Based on how long they exist, objects are classified into :
  - Persistent objects – which exist even if server is not into existence.
  - Transient objects – exist only for the time server is in existence.
- Bindings can be classified as implicit and explicit:
  - Implicit Binding – the client is transparently bound to the object when the reference is resolved to the actual object.
  - Explicit Binding – the client first calls a specific function to bind the object before the method invocation.
- Parameter passing can be call by value or call by reference.

Case Study on Java RMI

- Java hides the differences between local and remote method invocation from the user.
- After marshalling the object, it is sent as a parameter to the RMI.
- Implementation of Java RMI:
  - Reference to remote object consists of the network address and endpoint of server.
  - It also contains the local ID for the actual object in the server address space.
  - Each object in Java is an instance of a class, which contains the implementation of one or more interfaces.

- A java remote object is built from two classes: (i) server class and (ii) client class.
  - Server Class : this class contains the implementation of server-side code i.e., objects that run on the server. It consist of description of the object state and implementation of methods which operate on the state.
  - Client Class : this class contains the implementation of client-side code and proxy. It is generated from the object interface specification. The proxy basically converts each method call into a message that is sent to the server-side implementation of the remote object.

University Questions

- Short Notes on:-
  - Light Weight RPC (5 marks and 10 marks)
  - Marshalling
- In a fault tolerant communication between client-server, how will you implement 'exactly-once' semantics in following cases:
  - The client-server machines are reliable but the communication links connecting them are unreliable.
  - The client-server machines are unreliable but the communication links connecting them are reliable.
  - The client is unreliable but the server and the communication links are reliable.
  - The client and the communication links are reliable but the server is unreliable.

- What is the difference between LPC and RPC? Explain RPC model with the help of diagram.
- Why do most RPC systems support call-by-value semantics for parameter transfer? Explain with example. How is call-by-value implemented?
- Explain stateless and stateful server concepts. Explain advantages of stateless server paradigm in crash recovery.
- What is stub? How are stubs generated? Explain how the use of stubs help in making an RPC mechanism transparent?
- What is callback RPC facility? Give an example of an application where this facility may be useful.
- How does a binding agent work in a client-server communication?

- What is an orphan call? How are orphan calls implemented in
  - Last-one-call semantics
  - At-least once call semantics
  - Last of many call semantics
- What is an idempotent operation? Give two examples.
- What are different server creation semantics involved in RPC?
- How does a Binding Agent work in Client-Server Communication?
- Why do most RPC systems support call-by-value semantics for parameter passing?
- What do you mean by client-server binding? What is the role of binding agent in locating server.
- Explain client server binding with special focus on server location, simultaneous bindings and exception handling for RPCs.

Assignment 1 (Submission – 20/08/2018)

- What is an orphan call? How are orphan calls implemented in
  - Last-one-call semantics
  - At-least once call semantics
  - Last of many call semantics
- Give a mechanism for consistent ordering of messages in following cases:-
  - One-to-many communications
  - Many-to-one communications
  - Many-to-many communications
- How many types of reliabilities are there in group communication? Give two examples of each.

- Suppose a component of a distributed model suddenly crashes. How will this inconvenience the users when one of the following happens:-
  - The system uses processor-pool model and crashed component is a processor in the pool.
  - In processor-pool model , a user terminal crashes.
  - The system uses a workstation-server model and server crashes.
  - In the workstation-server model , one of the client crashes.
- In a fault tolerant communication between client-server, how will you implement 'exactly-once' semantics in following cases:
  - The client-server machines are reliable but the communication links connecting them are unreliable.
  - The client-server machines are unreliable but the communication links connecting them are reliable.
  - The client is unreliable but the server and the communication links are reliable.
  - The client and the communication links are reliable but the server is unreliable.