# COMP90015 Distributed Systems

## Protocols



Aaron Harwood

School of Computing and Information Systems





2022 Semester II

1. Exchange Protocols
   - Request Protocol
   - Request/Reply Protocol
   - Request/Reply/Acknowledge Protocol
2. Remote Invocation
   - Remote Procedure Call
   - Remote Method Invocation

## Communication Failure

Let's say we have a client and a server and we use a reliable stream communication protocol like TCP to send requests from the client to the server. Consider the case where the client writes a request (e.g. in the form of a JSON object using UTF8 encoding with newline delimiters) to the socket. The client then writes a subsequent request to the socket however an exception is thrown by the socket. Question: how can the client know if either of the requests were actually received by the server? What should the client do in this case?



> *Under extreme failure conditions, e.g. network outage for an indefinite period of time, every communication protocol either blocks for an indefinite period of time or eventually times out and fails by raising an exception to the application.*

# Requests, Responses and Acknowledgements

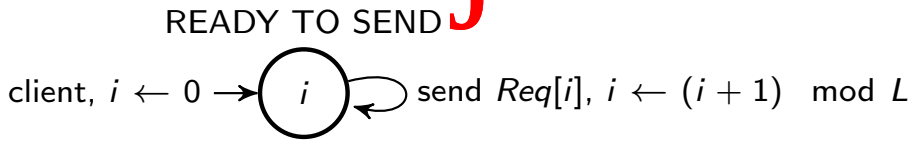


- Exchange protocols are fundamental building blocks of more complicated protocols. They describe how a sender and receiver, or e.g. a client and server, can exchange messages in a systematic way. Usually we talk about the client sending requests to the server. The request may or may not entail the server providing a response or reply. Furthermore we may consider the case when the server requires the client to acknowledge receipt of the response.
- For the purposes of reasoning about the behaviour of exchange protocols we will use *sequence numbers* – usually a simple finite counter, e.g. taking values $0, 1, 2, \ldots, L-1$. The value of $L$ can be determined based on the protocol, e.g. we may only need two sequence numbers, 0 and 1, or we may need more.
- Request with sequence number $i$ from the client will be written as $Req[i]$.
- Response to $Req[i]$ will be written as $Rsp[i]$.
- Acknowledgement of $Rsp[i]$ will be written as $Ack[i]$.

# Send a sequence of requests without expecting replies

The client's sender protocol is modelled as below, which is a FSM with $L$ states (shown in compact form), each representing the current message identifier $i$ that is being sent.

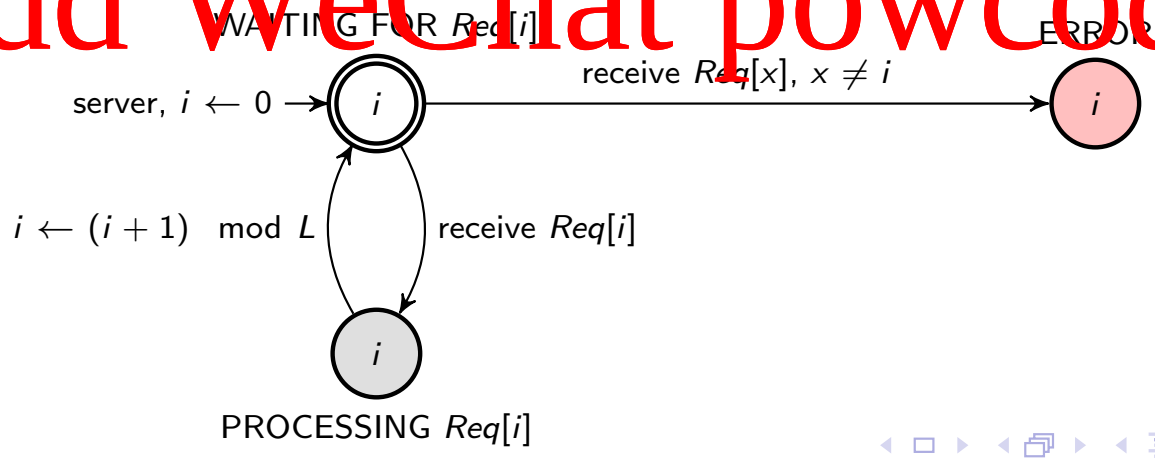







The server's receiver protocol is similarly modelled, in this case an error state is entered when anything other than the expected message is received. The error state raises an exception to the server to indicate the communication protocol has failed to operate as expected.



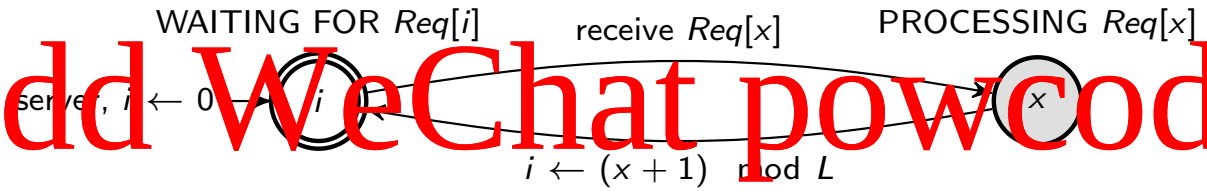


# Handling errors



With the previous protocol example there is no way for the sender to know that the receiver is in error. The sender will simply continue to send new requests. One way to overcome this is for the receiving protocol to recover or tolerate such an error, perhaps simply by accepting and processing the next received request regardless of the sequence. In this example $x$ may or may not equal $i$ and errors in sequence are therefore not considered.








In fact in this case we really do not care about sequence numbers at all.

## "Maybe" semantics



The simple Request protocol provides no guarantees to the client which is called *maybe* semantics. Maybe the request was processed by the server, maybe it was not processed. No errors can arise at the client. If the sequence of request processing is not enforced then the client needs to assume that the server is processing any subset of the sent requests, in any order, e.g.:



$$Req[0], Req[1], Req[3], Req[4], Req[2], Req[10], Req[11], Req[12], \ldots$$

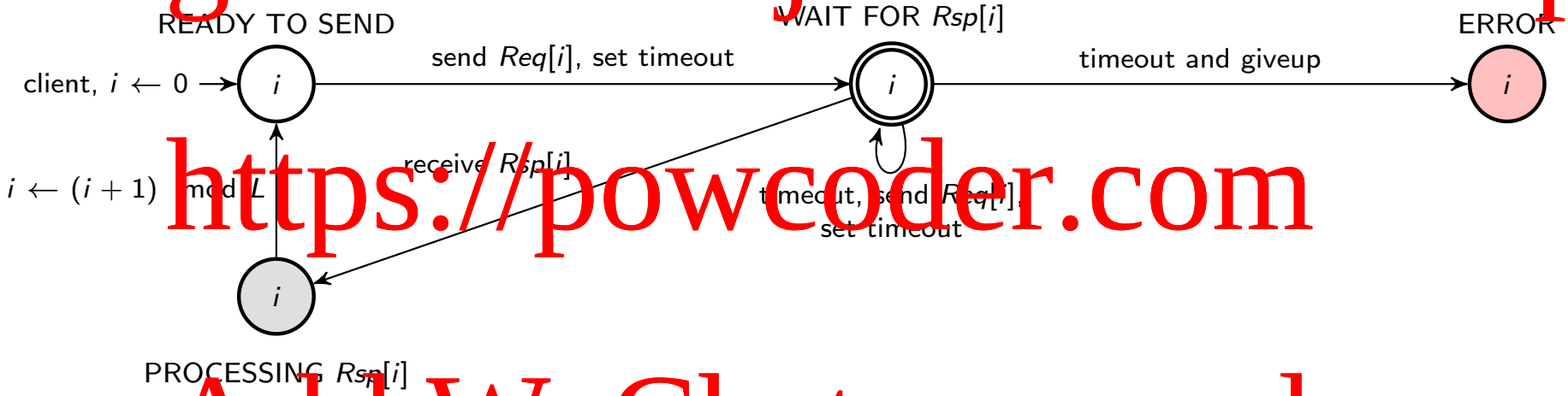


The distributed system must be able to operate correctly with these weak guarantees, otherwise it must use a different protocol.

# Ensuring requests are processed in sequence

To ensure the sequence of requests is processed in the same order as sent, the client needs a response to each request and cannot send the next request until the response for the currently sent request has been received.










Ensuring sequence is *synchronous* – it does not allow another request to be sent until it has received a response for the current request. Ensuring that the request has been processed may be impossible. It may eventually give up and the protocol is then in error (exception raised to the client), or it may continue to timeout and retry forever, which *blocks* the client from sending more requests.

# "At least once" semantics and idempotent requests

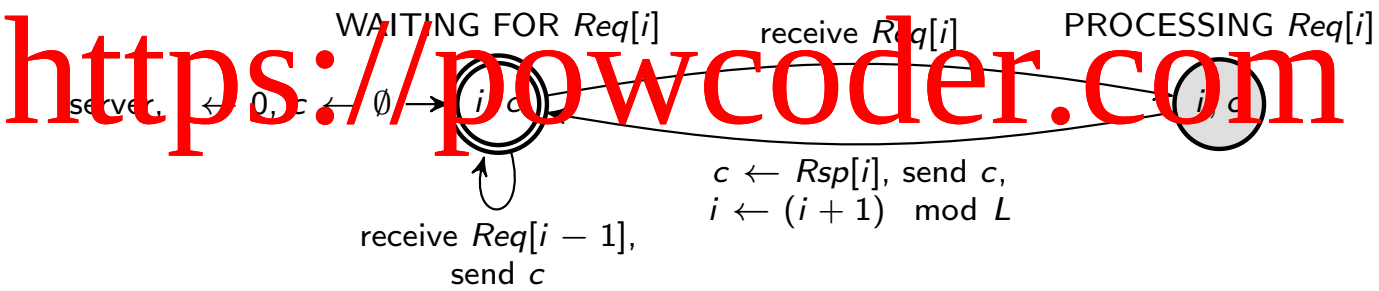
- Waiting for a response and retrying if no response is received within a certain time is guaranteeing that the request was processed by the receiver *at least once* or else an error occurs.
- If the server's receiver protocol is the same as earlier, the server may process the same request more than once. This may lead to an error. There are generally two types of requests:
  - stateless: e.g. the request is a computation request like $5 + 2$ and the response is the answer 7.
  - stateful: e.g. the request is *getAccount*(*accountId*) and the response is *account*. Statefull responses can involve reading and writing state.
    - E.g. *withdraw*(*accountId*, 5) and the response is *newBalance*. This modifies the state of the account.
  - For stateless requests, processing the same request multiple times will not lead to an error but will waste resources at the server.
  - For stateful requests, if the request is *idempotent* then processing it multiple times will not lead to an error. E.g. a request like *setBalance*(*accountId*, 10) if executed multiple times does not lead to an error, but a request like *deposit*(*accountId*, 5) will lead to an error if executed erroneously multiple times.

For non-idempotent requests we would like the protocol to ensure that each request is processed only once.

# "Exactly once" semantics

Since the protocol has introduced the possibility of duplicate requests, the receiver must be able to remove duplicate requests. Note that the sender will not send $Req[i+1]$ until it has received $Rsp[i]$ – the server will never "miss" requests, and so such an error condition never arises.





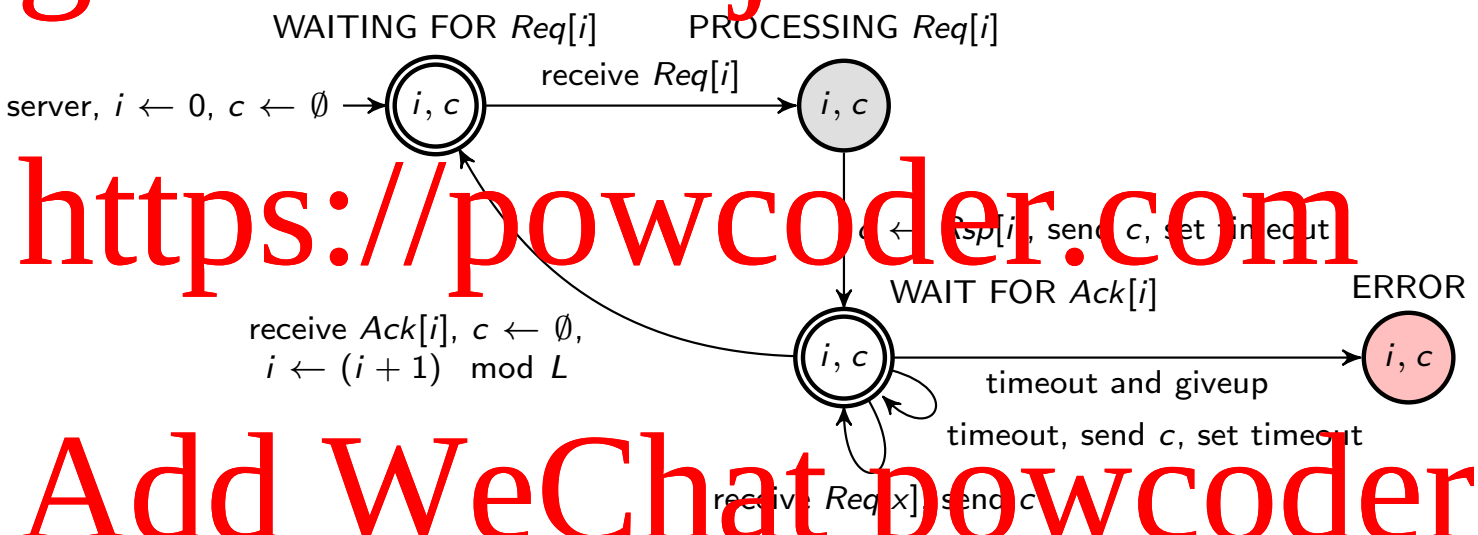



If a duplicate $Req[i]$ was received, it must be that the sender did not receive $Rsp[i]$. Instead of the server reprocessing the duplicate request, keeping a copy of the response any simply resending it can be done by the protocol. The server does not reprocess the duplicate request. In this case the protocol is providing *exactly once* semantics.

# Ensuring cached $Rsp[i]$ can be deleted

The server does not want to cache $Rsp[i]$ indefinitely as it consumes resources. The receiving protocol can require the receipt of $Rsp[i]$ to be acknowledged so that it can safely delete $Rsp[i]$ from its cache.





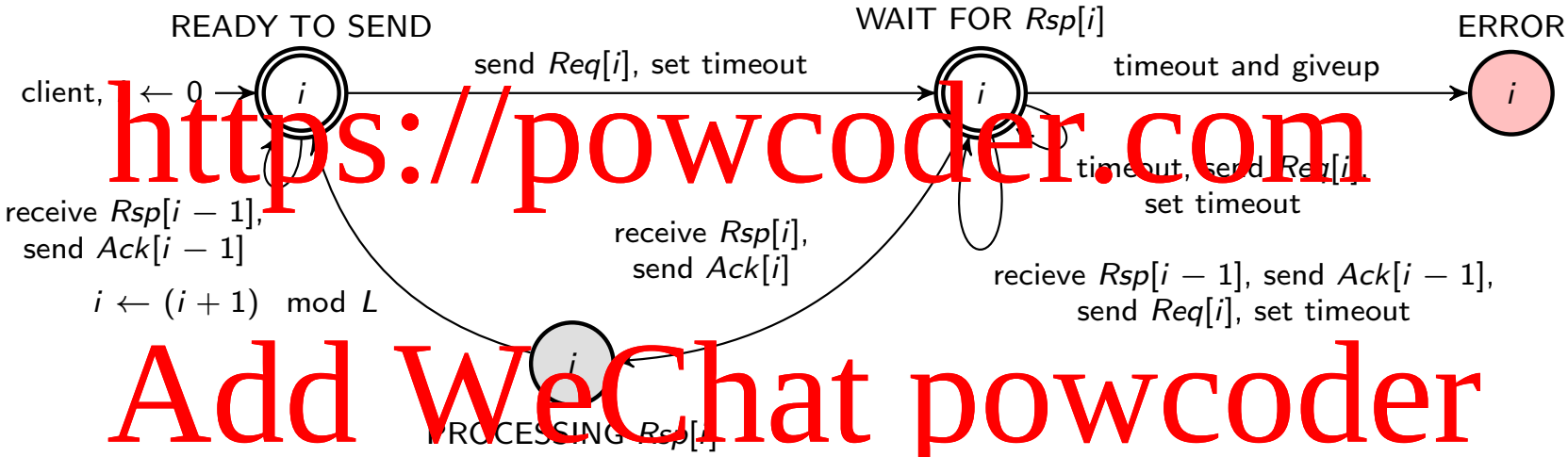

For a synchronous Request/Reply the consumption of resource $c$ takes constant space (i.e. there is only ever 1 cached response at a time) and therefore ensuring that it can be deleted is not really so important. But for asynchronous protocols where several requests and responses can be outstanding, resource consumption needs to be managed.

# Send acknowledgements



The client will need to send acknowledgements, perhaps multiple times due to acknowledgements being lost.








Since acknowledgements do not represent any cached data, there is no notion of that at the client.

# Discussion questions I

**Question (1):** For each of the Reply, Request/Reply and Request/Reply/Acknowledge protocols, draw sequence diagrams that show all of the relevant communication scenarios, including scenarios involving loss of messages.

**Question (2):** Not all kinds of failure are handled by the protocols. E.g. process failure that results in the client or server starting again (i.e. the state of the protocol is lost) is not handled. How can the protocols be improved to handle such possibilities?

**Question (3):** The protocols so far are synchronous in that only one outstanding request is permitted at any one time. Let's say we allow 2 outstanding requests/acknowledgements at any one time. Design a Request/Reply/Acknowledge protocol that allows this. What about allowing up to $k$ outstanding requests/acknowledgements at any one time?

**Question (4):** The protocols so far assumed that there is a sender and a receiver. In a peer-to-peer model, where either peer can make requests of

# Discussion questions II



the other peer, we might consider a single protocol that involves states for both sending and receiving. This can be thought of as two protocols operating concurrently, one for sending and one for receiving. Can you design a protocol that combines sending and receiving?

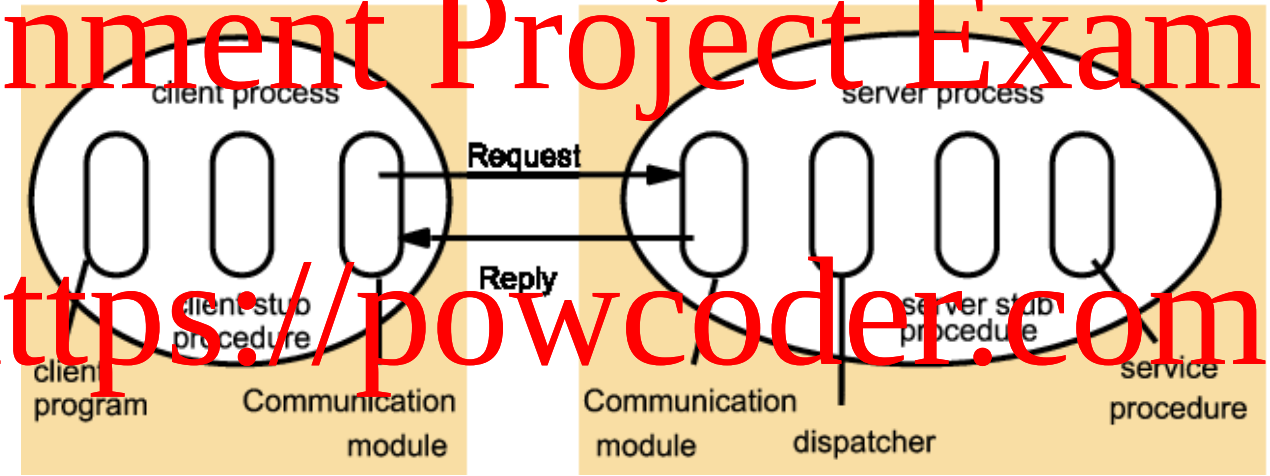


**Question (5).** The protocols so far assumed point-to-point or 2 party communication. Suppose we have a point-to-multipoint protocol, e.g. where 3 peers are communicating such that a request sent by a peer is to be processed by the other 2. Can you design a Request/Reply/Acknowledge protocol for this case? What about for the case of $k$ peers?

# Remote Procedure Call

RPCs enable clients to execute procedures in server processes based on a defined service interface.








- **Communication Module** Implements the desired design choices in terms of retransmission of requests, dealing with duplicates and retransmission of results.
- **Client Stub Procedure** Behaves like a local procedure to the client. Marshals the procedure identifiers and arguments which is handed to the communication module. Unmarshalls the results in the reply.
- **Dispatcher** Selects the server stub based on the procedure identifier and forwards the request to the server stub.
- **Server stub procedure** Unmarshalls the arguments in the request message and forwards it to the service procedure. Marshalls the arguments in the result message and returns it to the client.

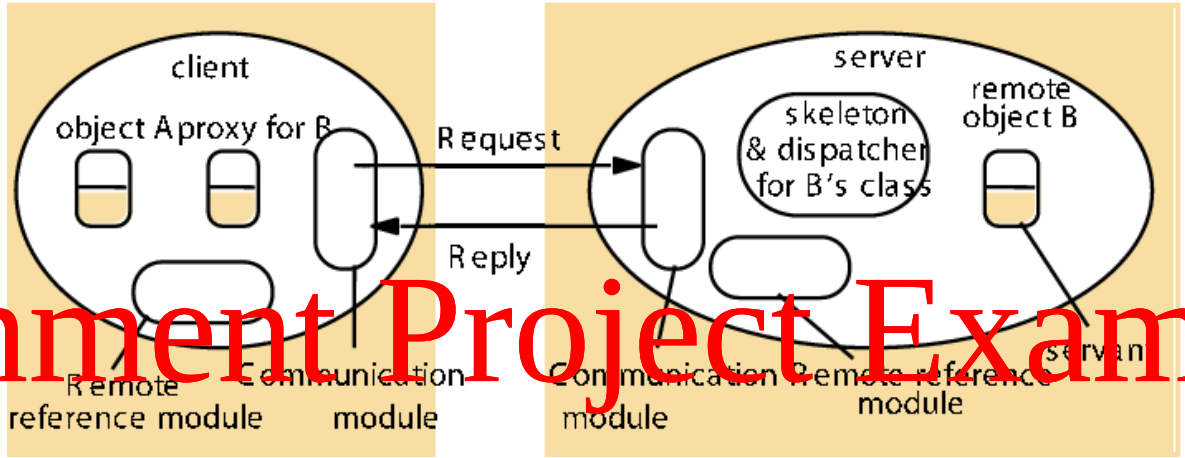
# Remote Method Invocation

An object that can receive remote invocations is called a remote object. A remote object can receive remote invocations as well as local invocations. Remote objects can invoke methods in local objects as well as other remote objects.






A remote object reference is a unique identifier that can be used throughout the distributed system for identifying an object. This is used for invoking methods in a remote object and can be passed as arguments or returned as results of a remote method invocation.






- The **Communication Module** is responsible for communicating messages (requests and replies) between the client and the server.
  - Messages include message type, request ID and remote object reference.
- The **Remote Reference Module** is responsible for creating remote object references and maintaining the remote object table which is used for translating between local and remote object references.
- The **Proxy** plays the role of a local object to the invoking object. There is a proxy for each remote object which is responsible for:
  - Marshalling the reference of the target object, its own method id and the arguments and forwarding them to the communication module.
  - Unmarshalling the results and forwarding them to the invoking object
- There is one **Dispatcher** for each remote object class. It is responsible for mapping to an appropriate method in the skeleton based on the method ID.
- The **Skeleton** is responsible for:
  - Unmarshalling the arguments in the request and forwarding them to the servant.
  - Marshalling the results from the servant to be returned to the client.

## Binder or Registry



- Client programs require a way to obtain the remote object reference of the remote objects in the server.
- A **binder** (called a Registry in Java RMI) is a service in a distributed system that supports this functionality.
- A binder maintains a table containing mappings from textual names to object references.
- Servers register their remote objects (by name) with the binder. Clients look them up by name.

## Discussion Question



**Question (6):** From our understanding of architecture models, the binder is a centralized server that all other processes access to either register their remote objects or to look up remote objects. For Java RMI what happens if the number of JVMs that are accessing the binder grows too large for a single binder service to support? What can be done to solve this? Is this a problem for RPC?

## Garbage collection and Exceptions



- Garbage collection of remote objects occurs via reference counting and must count references held any Remote Reference Module that has a copy of the remote object reference.
- Exceptions need to be communicated to the caller and some Exceptions are RMI specific such as time out exceptions if there is network failure.