Remote Procedure Calls

- Another way of doing communication across machines (opposed to sockets).
- RPC makes a call to a remote service look like a local procedure call.
- Typically **Synchronous**: caller is suspended untill the remote procedure returns.
- RPC implementations are simpler and abstractions are much higher level.
- Serialization and packetizing is abstracted away.

Parameter Passing

- *call by reference* : procedure can modify the address space of the caller
- *call by value* : actual parameter values are copied to the formal parameter values.
- *call by copy/restore* : call by value + copy value back in caller on return.
- Values are copied to the server, then used in the function, then copied back once function is done. This means that the changes are not reflected until the remote procedure returns.

Issues in RPC

- **Transparency** :
- make messaging / communication invisible to the programmer.
- support different OSes
- **Parameter Passing**
- **Binding**
- Locate remote process *and* bind to it during executiion
- *Fault handling/tolerance*

RPC Stubs

- Client calls a **client stub** that is on the local machine and acts as a "proxy" or "wrapper: for the remote procedure.
- There is a similar **server stub** that wraps the call to the actual function on the server.
- **These are machine-generated**

Client Stub (Local application user space)

- Marshal arguments - machine independent - allows different OSes to communicate
- Send request to server

- wait for response
- unmarshal and return to caller

Server Stub

- Unmarshal arguments and makes the local procedure call
- sends the result

Steps of RPC

1. Client procedure calls client stub
2. Client stub builds message and makes local OS syscalls
3. Client OS sends the message to server.
4. Remote OS gives message to server stub
5. Server stub unpacks the params
6. Server calls the actual procedure and returns result to the stub
7. Server stub packs the result in a message and makes local OS syscalls for tx.
8. Server OS sends data to client OS via network.
9. Client OS gets the data and passes it to the client stub.
10. Client stub unmarshals the result and returns to the client.

Parameter passing

- The client stub is responsible for taking a function's parameters, pack them on a message and send them to the server stub.
- **Client and server must agree on message format and data encoding**.
- **Be very careful about passing `ints` because of endianness conversions**.
- Passing by reference is very tough. It has to be *emulated* by using copy/restore. By default, parameter passing is *by value*.
- Dealing with large objects using copy-restore becomes expensive - **Avoid sending large data structures across to RPCs**

RPC Binding (DCE RPC)

- The client/servers are written by different programmers. Binding refers to the process of locating the remote process and identifying the procedure to be executed / server stub.

- A centralized registry on the server maintains the (server process, end point) pairs. The client locates server machin, then service, and queries the server process on the end point.
- **Directory Server** : does the mapping of service name and ip address- Like DNS. The RPC server stub must register itself to the directory server. It should also setup a local (DCE) daemon endpoint location (server ip + port)
- Client queries the directory server fids the address of server from directory server. It then gets the endpoint information from the DCE daemon. *DCE daemon has a well-known end point.*
- Client can then invoke RPC using the obtained endpoint.

Writing Server and Client Code

- Server/Client stubs are machine-generated and is generated at compile time.
- Interface definition language(IDL): defines the server's interface like names, parameters, types.
- A stub compiler reads the IDL and produces **two** stubs and a header file for each server procedure.
- The header file must be included in both stubs.
- **All this is very similar to rosmsgs - gives clues to how they work**.
- UUIDGen generates a globally unique identifier to identify the RPC implementation at runtime.
- The IDL compiler generates the header file and the server stub/client stub.
- The server stub is compiled and the object file is linked with the server binary and so is the client stub.

Asynchronous RPC

- In traditional RPC, the remote call blocks, we want something that is non-blocking.
- In asynch rpc, the client merely waits for an ack from the server indicating that it has received the request. The server can continue to process the call while the client has either slept or moved on.
- Yet another variant is possible wherein the client doesn't even expect an ack and just sends another request if reply is not received within a specific time period.
- For RPC's with a return value, the server needs to make a **one-way RPC** to invoke an interrupt in the client and pass the result to the client.

Dealing with failures

Classes of failures

- Client cannot locate the server (either due to directory server or DCE daemon)
- Request/Reply message is lost.
- Server/Client crashes after requests
- In local procedure calls, the "client" and "server" failures are correlated.
- In RPC, it is tough because the two actors don't know about each other's state - the client may think that the server is running even after it has crashed.

Client is unable to locate server

- *Causes*: server down. new interfaces, new stubs
- Use exceptions in the client code to make sure that if you fail a server-lookup, you terminate gracefully.

Lost Messages

- Issue: client doesn't exactly know what happened at the server? crash? latency?
- Fix: resend the same request until it works - will only work if the operation is **idempotent** - i.e. can be repeated without any bad consequences.
- *Idempotent operations:* File reads.
- If operation is not idempotent, write timestamps to each message to uniquely identify messages.
- Try to make server side code *as idempotent as possible.*
- Can report "client unable to locate server" if it fails multiple times.

Server Crashes

- You don't know that the server is crashed.
- When the client timer expires, there are **3 semantics come into play:**
- *At least once* - client *stub* keeps trying until it gets a reply (all server operations are idempotent and stateless).
- *At most once* - client *stub* gives up on failure. The application using the stub determines what to do after this. It may to choose to do application-level retrying. This is the useful portion of this semantic.
- *Exactly once* - Difficult to implement in practice - because if the server crashes, and then reboots, it needs to guarantee tha it will process *all* requests that were queued before/during the crash.

Client Crashes

- The server may do "useless" work - that may no longer be needed.
- If the client reboots, it may receive a reply to an older (now meaningless) request.
- Fix: *Logging* - The client logs each request it makes and stores the record on disk. This way it can know if there were any pending requests when it crashed. *Reincarnation* - Client broadcasts epoch when rebots. The server can listen to this and kill the orphans from previous epochs. *Expiration* - Client sends heartbeat signal (a *soft-state*) which the server reads. If the server doesn't get the heartbeat, it assumes the client is dead and kills all its orphans.