Remote Procedure Calls

Remote Procedure Calls Overview

1. IPC mechanism that specifies that the processes interact via procedure call interface
    - GetFile App (client/server)
        - Create and initialize sockets
        - Allocate and populate buffers
        - Include 'protocol' information (GetFile, size)
        - Copy data into buffers (filename, data)
    - ModifyImage App (client/server)
        - Create and initialize sockets
        - Allocate and populate buffers
        - Include 'protocol' information (algorithm, parameters)
        - Copy data into buffers (image data)
2. Lots of common steps between each application
    - RPC defines ways to perform these common steps

Benefits of RPC

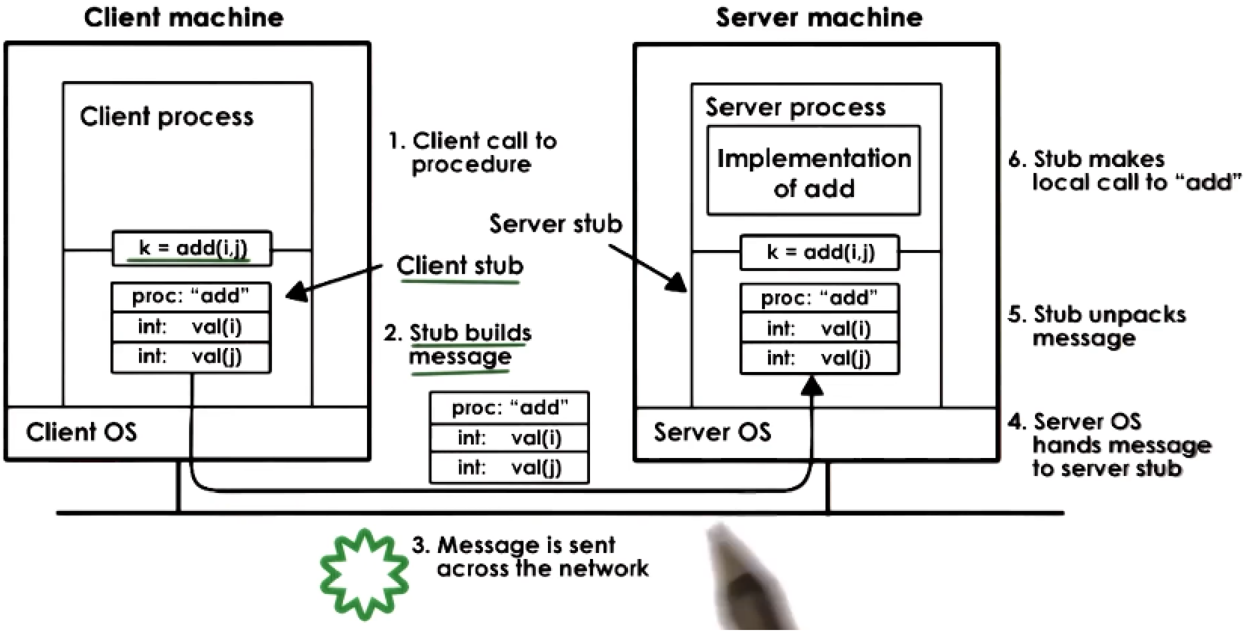
1. Higher-level interface for data movement and communication
2. Error handling
3. Hiding complexities of cross-machine interactions
    - Network may fail, machine may fail

RPC Requirements

1. Client/Server interactions - Client must be able to issue request to server
2. Procedure Call Interface -> RPC
    - Synchronous call semantics
    - Calling process blocks until procedure completes and returns result
3. Type Checking
    - Error handling
    - Packet bytes interpretation
4. Cross-Machine Conversion
    - Big/little endian
    - Data sent/received in network format
5. Higher-level Protocol
    - Access control, fault tolerance, ...
    - Different transport protocols

Structure of RPC

1. Client calls procedure
2. Stub builds message
3. Message is sent across the network
4. Server OS hands message to server stub
5. Stub unpacks message
6. Stub makes local call
7. Go through reverse path to communicate answer to client



RPC

Steps in RPC

0. register: server “registers” procedure, args types, location, ...
1. bind: client finds and “binds” to desired server
2. call: client makes RPC call; control passed to stub, client code blocks
3. marshal: client stub “marshals” arguments (serialize args into buffer)
4. send: client sends message to server (TCP, UDP, shared memory, ...)
5. receive: server receives message; passes msg to server-stub; access ctrl
6. unmarshal: server stub “unmarshals” args (extracts args & creates data structures)
7. actual call: server stub calls local procedure implementation
8. result: server performs operation and computes result of RPC operation

Interface Definition Language

1. What can the server do?
2. What arguments are required for the various operations?
    - Require an agreement
3. Why
    - Client-side bind decision
    - Runtime to automate stub generation
4. Interface definition language (IDL) serve as protocol for how agreement will be expressed

Specifying an IDL

1. IDL is used to describe the interface the server exports
    - Procedure name, argument, return types
    - Version number (which server has most current implementation)
2. RPC can use IDL that is:
    - Language-agnostic
        - External Data Representation (XDR) in SunRPC

    - Language specific: Java in JavaRMI
3. ONLY DEFINE THE INTERFACE, NOT IMPLEMENTATION

Marshalling

1. Serialize arguments of procedure into contiguous memory location to send to the server
2. Encode the data into an agreed-upon format to be correctly decoded on the receiving side

Unmarshalling

1. Decode the data received using the inverse operations from those used to marshal the data
2. IDL Specification defines how the data will be marshalled
    - RPC system compiler generates the marshal/unmarshal routines

Binding and Registry

1. Client determines…
    - Which server should it connect to?
        - Service name, version number, …
    - How will it connect to that server?
        - IP address, network protocol, …
2. Registry: Database of available services
    - Search for service name to find which service/how to connect
    - Distributed: Any RPC service can register
    - Machine-specific: For services running on same machine
        - Clients must know machine address -> Registry provides port number needed for connection
    - Requires some naming protocol
        - Exact match for "add"
        - Or consider "summation", "sum", "addition"
3. Who can provide services?
    - Look up registry for image processing
4. What services are provided?
    - Compress, filter, version # -> IDL
5. How will they send/receive?
    - TCP/UDP -> registry

Pointers in RPC

3. Pointers are local to the caller address space; server can't possibly access
4. Solutions:
    - No pointers
    - Serialize pointres; copy referenced ("pointed to") data structure to send buffer

Handling Partial Failures

1. When a client hangs… What's the problem?
    - Server down? Service down? Network down? Message lost?
    - Timeout and retry -> no guarantees!
2. Special RPC error notification (signal, exception, …)
    - Catch all possible ways in which the RPC call can (partially) fail

RPC Design Choice Summary

1. Binding: How to find the server
2. IDL: How to talk to server, how to package data

3. Pointers as arguments: Disallow or serialize pointer data
4. Partial failures: Special error notifications

What is SunRPC?

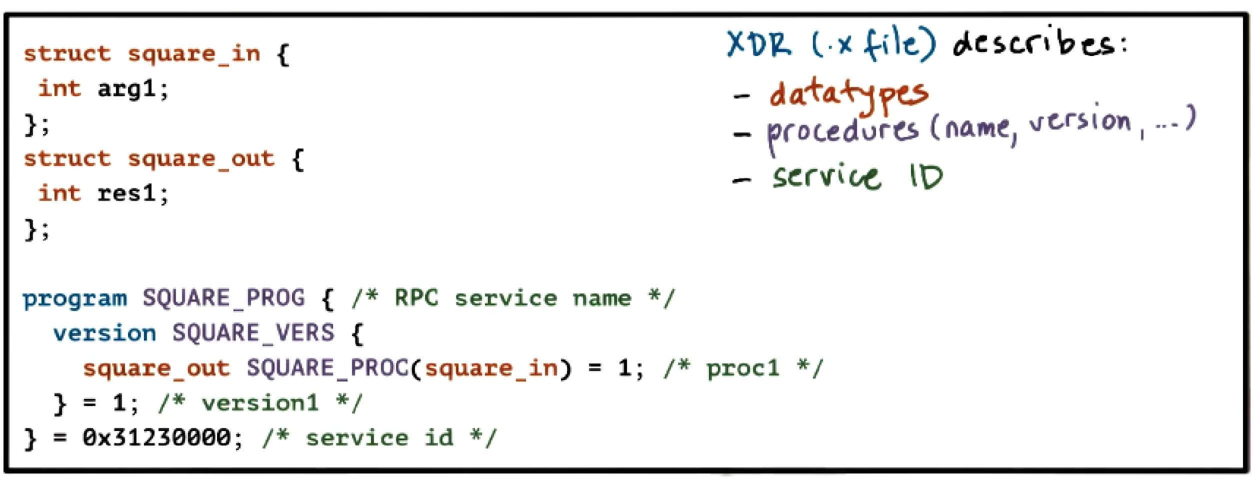
1. Developed in 80s by Sun for Unix systems using NFS (network file share)
2. Design choices:
    - Binding: Per-machine registry daemon
    - IDL: XDR (for interface specification and encoding)
    - Pointers: Allowed and serialized
    - Failures: Retries; return as much information as possible

SunRPC Overview

1. Client/server interact via procedure calls (same or different machines)
2. Interface specified via XDR (.x file)
3. rpcgen compiler: Converts .x to language-specific stubs
4. Server registers with local registry daemon
5. Registry (per-machine): Name of service, version, protocol(s), port number
6. Binding creates handle
    - Client uses handle in calls
    - RPC runtime uses handle to track per-client RPC state
7. TI-RPC: Transport-independent SunRPC

SunRPC XDR Example

1. Client: Send x; Server: Return x^2
2. Version is not used by programmer, only internally to identify procedure
3. Service ID Conventions:
    - 0x0000 0000 - 0x1fff ffff == Defined by Sun
    - 0x2000 0000 - 0x3fff ffff == Range to use
    - 0x4000 0000 - 0x5fff ffff == Transient
    - 0x6000 0000 - 0xffff ffff == Reserved



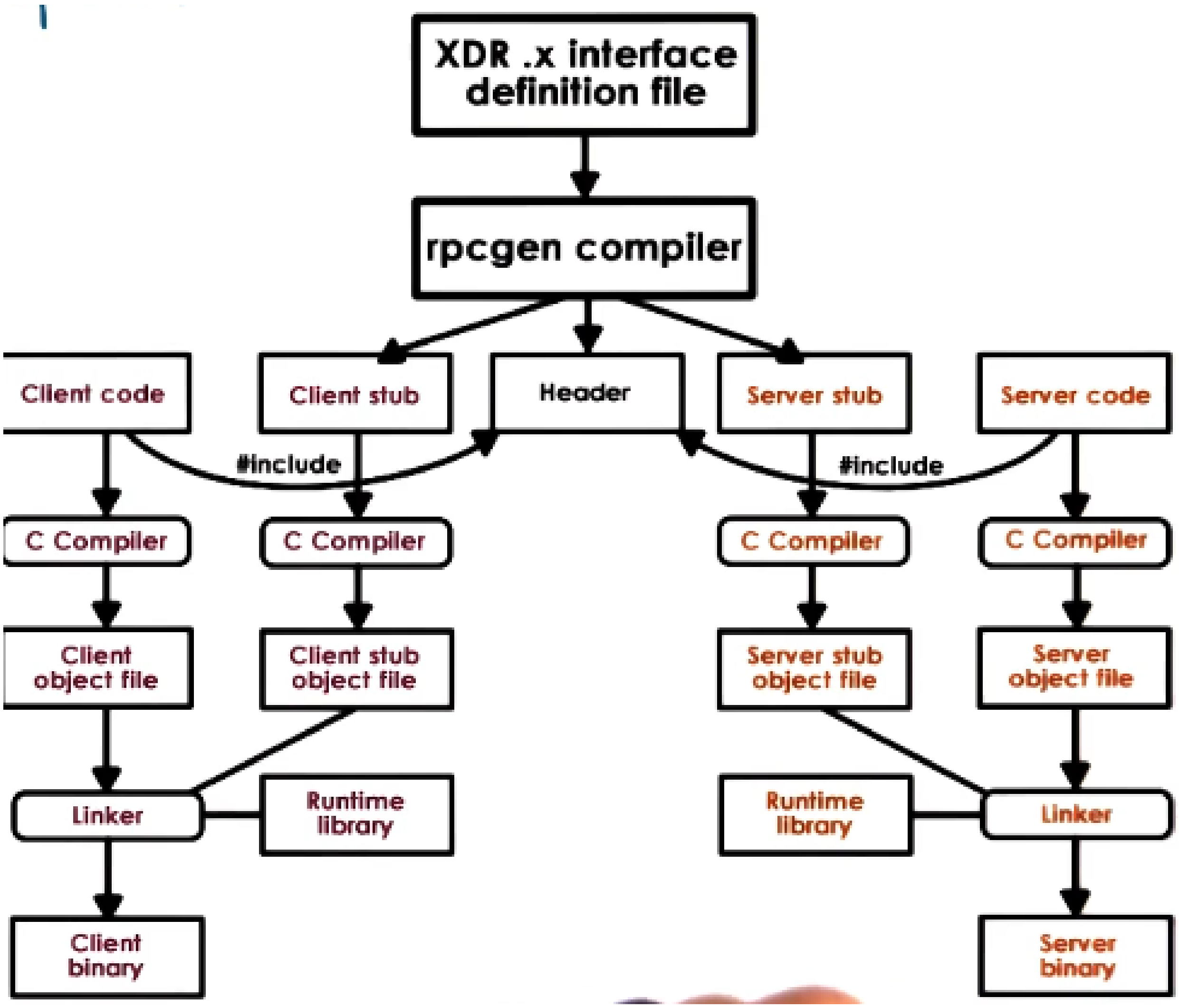
XDR

Compiling XDR

1. rpcgen compiler: rpcgen -c square.x

- Creates square.h -> data types and function definitions
- square_svc.c -> Server stub and skeleton (main)
  - main -> registration/housekeeping
  - square_prog_1 (internal code, request parsing, arg marshalling; 1 indicates version 1)
  - square_proc_1_svc -> actual procedure; must be implemented by developer
- square_clnt.c -> client stub
  - squareproc_1: wrapper for RPC call to square_proc_1_svc
  - y = squareproc_1(&x);
- square_xdr.c -> Common marshalling routines

2. From .x -> header, stubs, …
3. Developer
   - Server side (implementation of square_proc_1_svc)
   - Client side (call squareproc_1())
   - #include .h
   - Link with stub objects
4. RPC Runtime handles the rest
   - OS interactions, communication management, …
5. rpcgen -C square.x -> not thread safe!
   - y = squareproc_1(&x, client_handle)
6. rpcgen -C -M square.x -> multithreading safe!
   - status = squareproc_1(&x, &y, client_handle)
   - Dynamically allocates instead of statically
   - Doesn't make a multithreaded "_svc.c" server
     - On Solaris "-a" -> MT server
     - Linux requires manual creation of multithreading server



Compiling XDR

SunRPC Registry

1. RPC daemon: portmapper
   - /sbin/portmap (need sudo privileges)
2. Query with rpcinfo -p
   - /usr/sbin/rpcinfo -p
   - Program id, version, protocol (TCP/UDP), socket port number, service name, ...
   - portmapper runs with tcp and udp on port 111

SunRPC Binding

1. CLIENT Type
   - client handle
   - status, error, authentication, ...

```
// in client
CLIENT* clnt_create(char* host, unsigned long prog,
                    unsigned long vers, char* proto);
```

```
// for square example
CLIENT* clnt_handle;
clnt_handle = clnt_create(rpc_host_name, SQUARE_PROG, SQUARE_VERS, “tcp”);
```

Binding in SunRPC

XDR Data Types

1. Default types: char, byte, int, float, ...
2. Additional XDR types:
   - const (#define)
   - hyper (64-bit integer)
   - quadruple (128-bit float)
   - opaque (~C byte; uninterpreted binary data)
3. Fixed-length array:
   - int data[80]
4. Variable-length array:
   - int data<80> (80 is maximum expected size)
   - Translates into a data structure with “len” and “val” fields
5. Strings
   - string line<80> (c pointer to char)
   - stored in memory as a normal null-terminated string
   - encoded (for transmission) as a pair of length and data

XDR Routines

1. Marshalling/unmarshalling: found in square_xdr.c
2. Clean-up
   - xdr_free()
   - user-defined _free_result procedure
   - e.g., square_prog_1_free_result called after results returned

XDR Encoding

1. Transport header
   - TCP, UDP
2. RPC header: service procedure ID, version number, request ID, ...
3. Actual data: arguments or results
   - Encoded into a bytestream depending on data type
4. XDR specifies the IDL and encoding
   - Binary representation of data “on-the-wire”
5. XDR Encoding Rules
   - All data types are encoded in multiples of 4 bytes (alignment)
   - Big endian is the transmission standard
   - Two’s complement is used for integers
   - IEEE format is used for floating point

Java Remote Method Invocations (RMI)

1. Developed by Sun
   - Among address spaces in JVM(s)
   - Matches Java OO semantics
   - IDL is Java (language-specific)
2. RMI Runtime
   - Remote Reference Layer
     - Unicast, broadcast, return-first response, return-if-all-match
   - Transport
     - TCP, UDP, shared memory