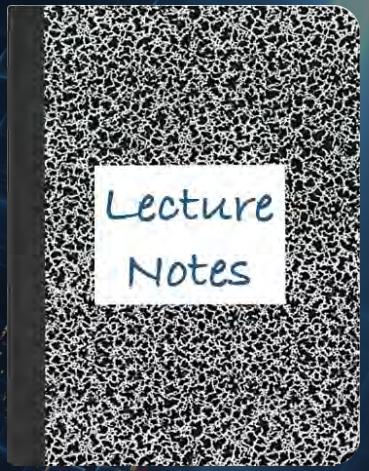
CS 417 – DISTRIBUTED SYSTEMS

Week 2: Part 3

Examples of RPC Systems



Paul Krzyzanowski

Three generations (roughly)

Remote Procedure Calls

Remote Objects

Web Services

ONC (Sun) RPC

ONC (Sun) RPC

- RPC for Unix System V, Linux, BSD, macOS
  - Created by Sun (now Oracle)
  - ONC = Open Network Computing
    - The consortium of companies that supported the standard
  - Defined in RFC 1831 (1995), RFC 5531 (2009)
  - Remains in use mostly because of NFS (Network File System)

- Interfaces defined in an Interface Definition Language (IDL)

- IDL compiler is *rpcgen*

Sample IDL file

name.x

```
program GETNAME {
    version GET_VERS {
        long GET_ID(string<50>) = 1;
        string GET_ADDR(long) = 2;
    } = 1;    /* version */
    version GET_VERS2 {
        long GET_ID(string<50>) = 1;
        string GET_ADDR(string<128>) = 2;
    } = 2;    /* version */
} = 0x31223456;
```

Why is versioning important?

Interface definition: version 2

rpcgen

`rpcgen name.x`

produces:

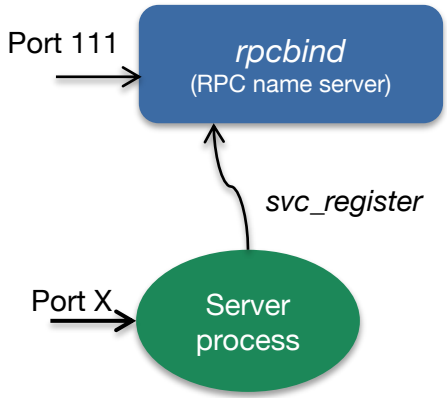
- `name.h` header
- `name_svc.c` server skeleton (stub)
- `name_clnt.c` client stub (proxy)
- [`name_xdr.c`] optional XDR data conversion routines

- Function names derived from IDL function names and version numbers
- Client gets *pointer* to result
  - Allows it to identify failed RPC (null return)
  - Reminder: C doesn't have exceptions!

What goes on in the system: server

Start server

- Server skeleton creates a socket and binds any available local port to it
- Calls a function in the RPC library:
  - *svc_register* to register program #, port #, protocol (TCP/UDP)
  - Contacts the port mapper, *rpcbind*:
    - Name server
    - Keeps track of {program #, version #, protocol} → port # bindings
- Server then listens and waits to accept connections



What goes on in the system: client

Client calls *clnt_create* with:

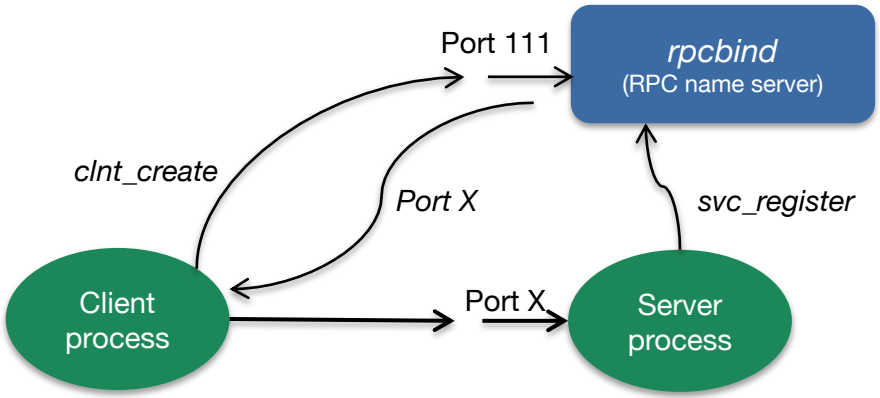
- Name of server
- Program #
- Version #
- Protocol (TCP or UDP)

clnt_create contacts port mapper on that server to get the port for that interface

- **early binding** – done once, not per procedure call

Communications

- Marshaling to XDR format (eXternal Data Representation)
  - Binary format using implicit typing



ONC RPC Advantages

- Don’t worry about getting a unique transport address (port)
  - But with you need a unique program number per server
  - Greater portability
- Transport independent
  - Protocol can be selected at run-time
- Application does not have to deal with maintaining message boundaries, fragmentation, reassembly
- Applications need to know only one transport address
  - Port mapper (rpcbind process)
- Function call model can be used instead of send/receive
- Versioning support between client & server

DCE RPC

http://www.opengroup.org/dce/

DCE RPC

- Similar to ONC RPC
- Interfaces written in an Interface Definition Notation (IDN)
  - Definitions look like function prototypes
- Run-time libraries
  - One for TCP/IP and one for UDP/IP
- Authenticated RPC support with DCE security services
- Integration with DCE directory services to locate servers

Unique IDs

ONC RPC required a programmer to pick a “unique” 32-bit number

DCE: get unique ID with the uuidgen command

- Generates prototype IDN file with a 128-bit Unique Universal ID (UUID)
- 10-byte timestamp with version number
- 6-byte node identifier (ethernet address on ethernet systems)

IDN compiler

Similar to rpcgen:

Generates header, client stub, and server skeleton

Service lookup

Sun RPC requires client to know name of server

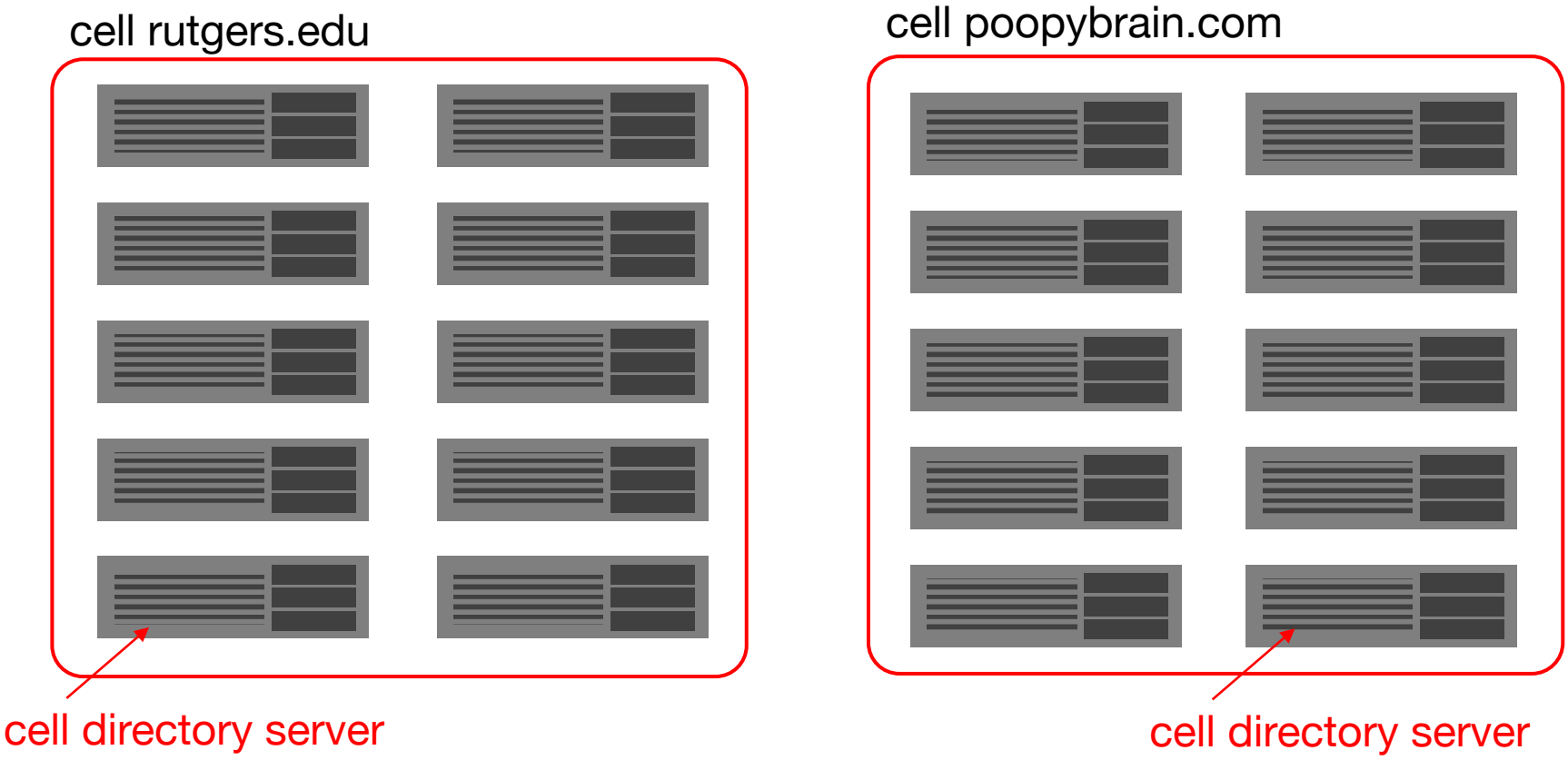
DCE allows several machines to be organized into an administrative entity

- **cell** (collection of machines, files, users)

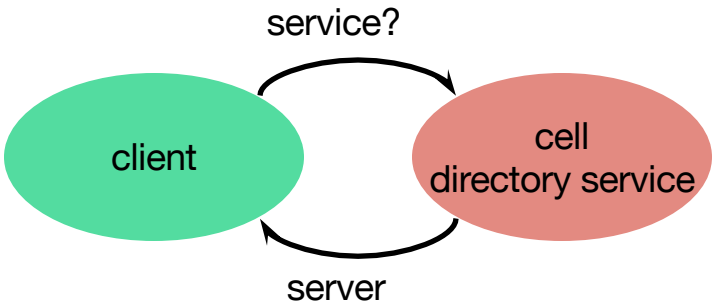
Cell directory server

- Each machine communicates with it for cell services information

Cells & the cell directory server



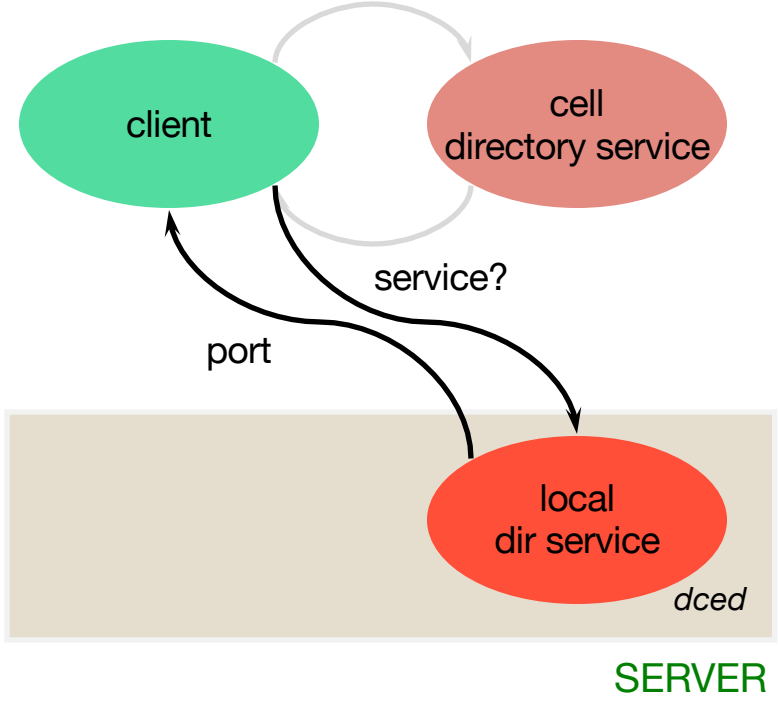
DCE service lookup



Request service lookup from cell directory server

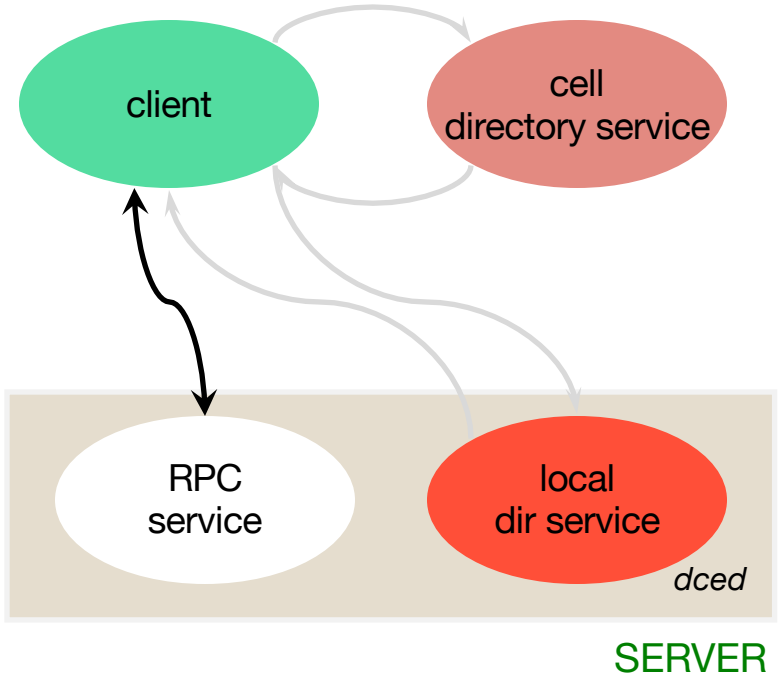
Return server machine name

DCE service lookup



Connect to endpoint mapper service and get port binding from this local name server

DCE service lookup



Connect to service and request remote procedure execution

Marshalling

Standard formats for data

- NDR: Network Data Representation

Goal

- *Multi-canonical* approach to data conversion
  - Fixed set of alternate representations
  - Byte order, character sets, and floating-point representation can assume one of several forms
  - Sender can (hopefully) use native format
  - Receiver may have to convert

What's Good

- DCE RPC improved Sun RPC
  - Universally Unique ID (UUID)
  - Multi-canonical marshalling format
  - *Cell* of machines with a cell directory server
    - No need to know which machine provides a service

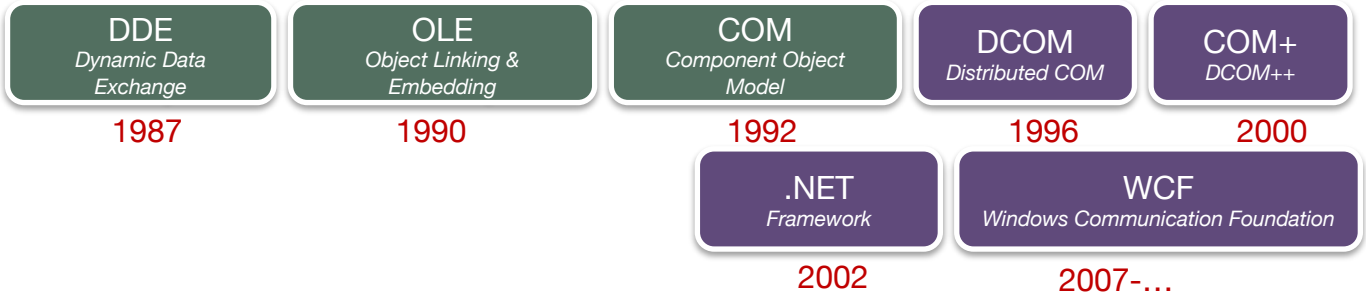
The next generation of RPCs

Distributed objects:
support for object-oriented languages

DOA: Distributed Object Architecture

Microsoft COM+ (DCOM)

Microsoft DCOM/COM+



COM+: introduced with Windows 2000

- Unified COM and DCOM plus support for transactions, resource pooling, publish-subscribe communication

Extends Component Object Model (COM) to allow objects to communicate between machines

Activation on server

Service Control Manager (SCM)

- Started at system boot. Functions as RPC server
- Maintains database of installed services
- Starts services on system startup or on demand
- Requests creation of object on server

Surrogate process runs components: dllhost.exe

- Process that loads DLL-based COM objects

One surrogate can handle multiple clients simultaneously

Beneath COM+

Data transfer and function invocation via Object RPC (ORPC)

- Small extension of the DCE RPC protocol

  Standard DCE RPC messages plus:
  - Interface pointer identifier (IPID)
    - Identifies interface and object where the call will be processed
    - Referrals: can pass remote object references
  - Versioning & extensibility information

Marshalling

- Marshalling mechanism: NDR
  same Network Data Representation used by DCE RPC
  - One new data type added: represents a marshaled interface
    - Allows one to pass interfaces to objects

- Remember: NDR is multi-canonical
  - Efficient when both systems have the same architecture

MIDL

MIDL = Microsoft Interface Definition Language

MIDL files are compiled with an IDL compiler

DCE IDN + object definitions

Generates C++ code for marshalling, unmarshalling, & stubs

- Client side is called the *proxy*
- Server side is called the *stub*

Both are COM objects that are loaded by the COM libraries as needed: the application loads the client COM object, which contacts the server to load the server COM object

COM+ Distributed Garbage Collection

Object lifetime controlled by remote reference counting

- *RemAddRef*, *RemRelease* calls
- Object elided when reference count = 0

COM+ Distributed Garbage Collection

Abnormal client termination

- Insufficient number of *RemRelease* messages sent to server
- Object will not be deleted

In addition to reference counting:

Client Pinging

- Server has *pingPeriod*, *numPingsToTimeOut*
- Background client process sends **ping set**
  - *IDs of all remote objects used on that server*
- If ping period expires with no pings received, all references are cleared

Microsoft DCOM/COM+ Contributions

- Fits into Microsoft COM model
- Support for references to instantiated objects
- Generic server hosts dynamically loaded objects
  - Requires unloading objects (dealing with dead clients)
  - Reference counting and pinging
- But… COM+ was a Microsoft-only solution
  - And it did not work well across firewalls because of dynamic ports

Java RMI

Java RMI

- Java language had no mechanism for invoking remote methods

- 1995: Sun added extension
  - Remote Method Invocation (RMI)
  - Allow programmer to create distributed applications where methods of remote objects can be invoked from other JVMs

RMI components

Client

- Invokes method on remote object

Server

- Process that owns the remote object

Object registry

- Name server that relates objects with names

Interoperability

RMI is built for Java only!

- No goal of OS interoperability
- No language interoperability
- No architecture interoperability

No need for external data representation

- All sides run a JVM

Benefit: simple and clean design

RMI similarities

Similar to local objects

- References to remote objects can be passed as parameters *(not as pointers, of course)*
  - You can execute methods on a remote object
- Objects can be passed as parameters to remote methods
- Object can be cast to any of the set of interfaces supported by the implementation
  - Operations can be invoked on these objects

RMI differences

- Objects (parameters or return data) passed by value
  - Changes will visible only locally
- Remote objects are passed by reference
  - Not by copying remote implementation
  - The “reference” is not a pointer. It’s a data structure:
    { IP address, port, time, object #, interface of remote object }
- RMI generates extra exceptions

Classes to support RMI

- remote class:
  - One whose instances can be used remotely
  - Within its address space: regular object
  - Other address spaces:
    - Remote methods can be referenced via an object handle

- serializable class:
  - Object that can be marshaled
  - Support serialization of parameters or return values
  - If a parameter is a remote object, only the object handle is copied

Classes to support RMI

- remote class:
  - One whose instances can be used remotely
  - Within its address space
  - Other address spaces:
    - Remote methods can be referenced via an object handle

needed for remote objects

- serializable class:
  - Object that can be marshaled
  - Support serialization of parameters
  - If a parameter is a remote object, only the object handle is copied

needed for parameters

Stub & Skeleton Generation

- Automatic stub generation since Java 1.5
  - Need stubs and skeletons for the remote interfaces
  - Automatically built from java files
  - Pre 1.5 (still supported) generated by separate compiler: *rmic*

- Auto-generated code:
  - Skeleton
    - Server-side code that calls the actual remote object implementation
  - Stub
    - Client-side proxy for the remote object
    - Communicates method invocations on remote objects to the server

Naming service

We need to look an object up by name

Get back a remote object reference to perform remote object invocations

Object registry does this: rmiregistry running on the server

Server

Register object(s) with Object Registry

```
Stuff obj = new Stuff();
Naming.bind("MyStuff", obj);
```

Client

Client contacts *rmiregistry* to look up the name

```
MyInterface test = (MyInterface)
    Naming.lookup("rmi://www.pk.org/MyStuff");
```

rmiregistry service returns a remote object reference.

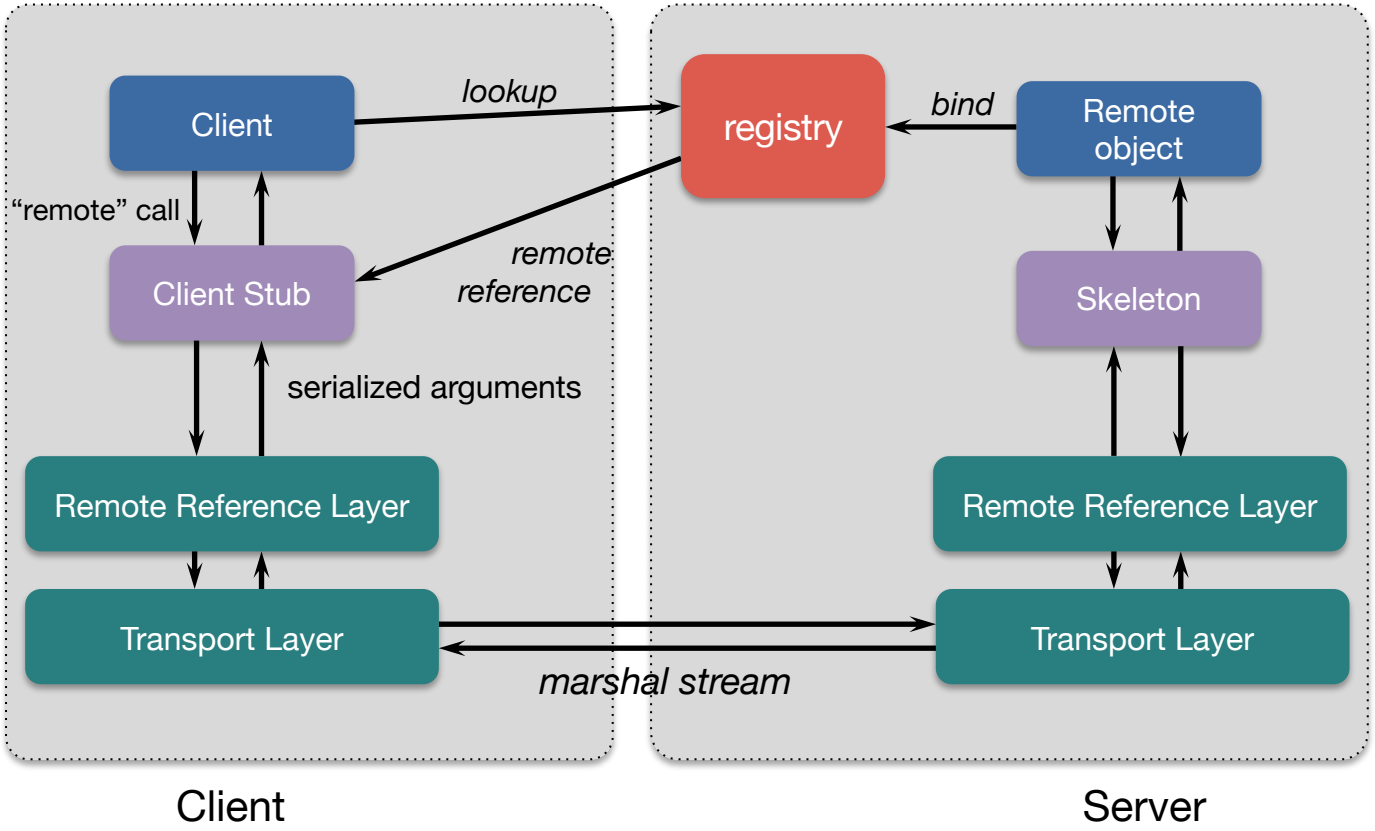
lookup method gives reference to local stub.

The stub now knows where to send requests

Invoke remote method(s):

```
test.func(1, 2, "hi");
```

Java RMI infrastructure



RMI Distributed Garbage Collection

- Lease-based garbage collection
  - Two operations: *dirty* and *clean*

- Local JVM sends a *dirty* call to the server JVM when the object is in use
  - The *dirty* call is refreshed based on the lease time given by the server
- Local JVM sends a *clean* call when there are no more local references to the object

- Unlike DCOM:
  no incrementing/decrementing of references

Python RPyC

RPC in Python

- Various implementations: PyRO, PyInvoke, RPyC, ZeroRPC
- What helps Python achieve transparency
  - Inspection of live objects through the **inspect** module
    - Examine the contents of a class, retrieve source code for a method, and extract the argument list for a function
- General idea of implementing RPC on Pythin
  - Create a connection using an RPC object
  - Then invoke remote methods using that object

RPyC Goals

- **Transparent RPC interface**
  - No definition files, stub compilers, name servers, transport services
- **Symmetric operation**
  - Both sides can invoke RPCs on each other – enables **callback functions**
- **Server**
  - RPyC ThreadedServer started on the server program
  - Binds to a default port (18812) or you specify the host's IP address and port
- **Client**
  - Connects to the server
  - Performs remote operations through the `modules` property, which exposes the server module's namespace

Serialization: passing data

- **By value**
  - Simple types (immutable objects – strings, ints, tuples)
    - Sent directly to the remote side
- **By reference**
  - Objects: reference (object name) to an object is passed
    - Remote contacts the client to access attributes and invoke methods on these objects
    - Changes will be reflected onto actual object
  - Enables passing of location-sensitive objects, like files or other OS resources
    - Remote process can write to the `stdout` of a local process by getting its `sys.stdout`
  - Implementation: **netrefs** = transparent object proxies.
    - Local objects that forward all operations to the corresponding remote object
    - They make remote objects look & feel like local objects.

Stubs (object proxies)

- **Client creates local proxy objects for remote modules**
  - Allows for transparent access
  - Reference wrapped in a special object called a **proxy** that looks like the actual object
  - Any operation on the proxy is delivered to the target
  - Client is unaware of this
- **Synchronous & asynchronous calls**
  - **Synchronous**: code that issues the operation and waits for a return
  - **Asynchronous**: immediate return, notification when complete
    - Calls can be made asynchronous by wrapping the proxy with an asynchronous wrapper

Services & security

- RPyC is built around **services**
  - Each end of the connection exposes a service that is responsible for the policy
  - Policy = set of supported remote operations
- Services are classes that derive from `rpyc.core.service.Service` and define exposed methods
  - Methods whose names begin with `exposed_` or use the `@rpyc.exposed` decorator
  - All *exposed* members of a service class will be available to the other side

Exposed methods in a service

Example from RPyC documentation

Server

```
import rpyc

class CalculatorService(rpyc.Service):
    def exposed_add(self, a, b):
        return a + b
    def exposed_sub(self, a, b):
        return a - b
    def exposed_mul(self, a, b):
        return a * b
    def exposed_div(self, a, b):
        return a / b
    def foo(self):
        print("foo")
```

Client

```
import rpyc

conn = rpyc.connect("hostname", 12345)
x = conn.root.add(4,7)
assert x == 11

try:
    conn.root.div(4,0)
except ZeroDivisionError:
    pass
```

Simple RPyC program

Server

```
import rpyc
from rpyc.utils.server import ThreadedServer

@rpyc.service
class TestService(rpyc.Service):
    @rpyc.exposed
    def add(self, a, b):
        return a+b

    @rpyc.exposed
    def sub(self, a, b):
        return a - b

    @rpyc.exposed
    def whoami(self):
        return 'calculator'

print('starting server')
server = ThreadedServer(TestService, port=12345)
server.start()
```

Client

```
import rpyc

conn = rpyc.connect('localhost', 12345)

print(conn.root.add(5,6))
print(conn.root.sub(10,4))
print(conn.root.whoami())
```

On the server, run

```
python3 ./calcserver.py
```

On the client, run

```
python3 ./calcserver.py
```

The End