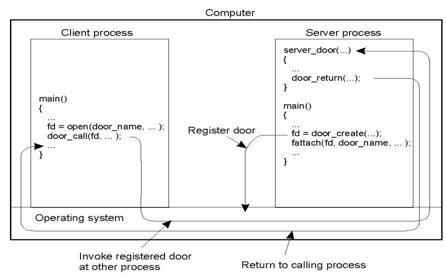
Remote Method Invocation

- Part 1: Alternate RPCs Models
- Part 2: Remote Method Invocation (RMI)
  – Design issues
- Part 3: RMI and RPC Implementation and Examples

Lightweight RPCs

- Many RPCs occur between client and server on same machine
  – Need to optimize RPCs for this special case => use a lightweight RPC mechanism (LRPC)
- Server S exports interface to remote procedures
- Client C on same machine imports interface
- OS kernel creates data structures including an argument stack shared between S and C

Lightweight RPCs



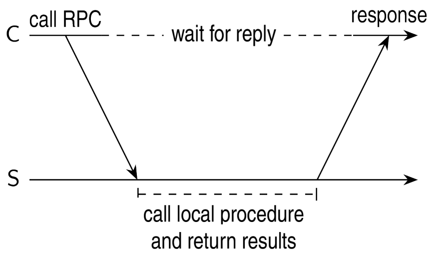
- RPC execution
  - Push arguments onto stack
  - Trap to kernel
  - Kernel changes mem map of client to server address space
  - Client thread executes procedure (OS upcall)
  - Thread traps to kernel upon completion
  - Kernel changes the address space back and returns control to client
- Called “doors” in Solaris
- Which RPC to use? - run-time bit allows stub to choose between LRPC and RPC

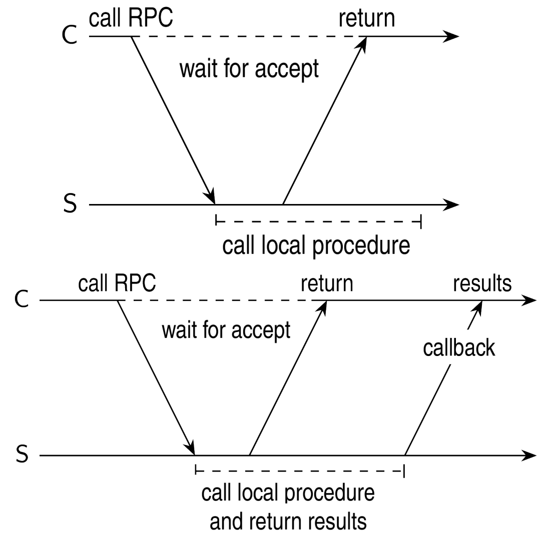
Other RPC Models

- Asynchronous RPC
  - Request-reply behavior often not needed
  - Server can reply as soon as request is received and execute procedure later
- Deferred-synchronous RPC
  - Use two asynchronous RPCs
  - Client needs a reply but can’t wait for it; server sends reply via another asynchronous RPC
- One-way RPC
  - Client does not even wait for an ACK from the server
  - Limitation: reliability not guaranteed (Client does not know if procedure was executed by the server)

– Multicast RPC

Asynchronous RPC



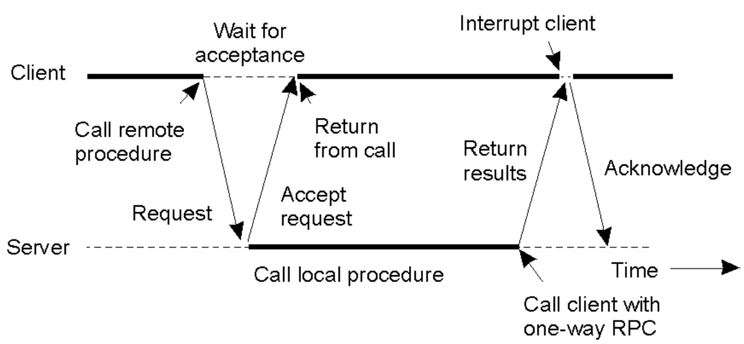


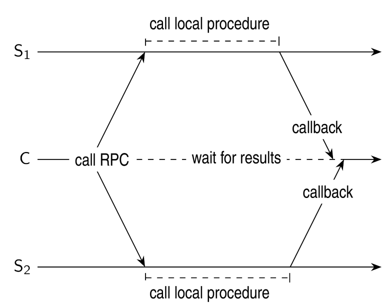
a) The interconnection between client and server in a traditional RPC

b) The interaction using asynchronous RPC

Deferred Synchronous and Multicast RPC

- Interactions for (i) two asynchronous RPCs, (ii) multicast RPC

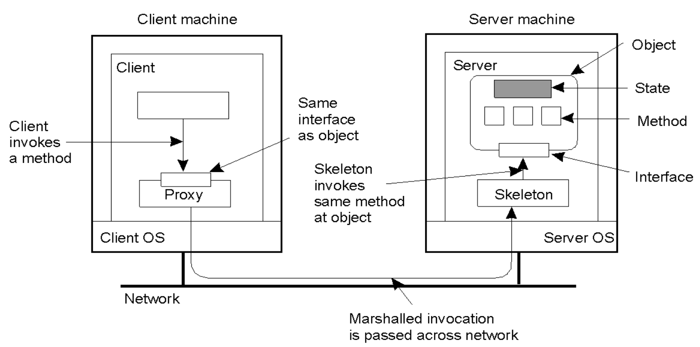




Part 2:Remote Method Invocation (RMI)

- RPCs applied to *objects,* i.e., instances of a class
  - *Class:* object-oriented abstraction; module with data and operations
  - Separation between interface and implementation
  - Interface resides on one machine, implementation on another
- RMIs support system-wide object references
  - Parameters can be object references

Distributed Objects



- When a client binds to a distributed object, load the interface ("proxy") into client address space
  - Proxy analogous to stubs
- Server stub is referred to as a skeleton

Proxies and Skeletons

- Proxy: client stub
  - Maintains server ID, endpoint, object ID
  - Sets up and tears down connection with the server
  - [Java:] does serialization of local object parameters
  - In practice, can be downloaded/constructed on the fly (why can't this be done for RPCs in general?)
- Skeleton: server stub
  - Does deserialization and passes parameters to server and sends result to proxy

Binding a Client to an Object

```
Distr_object* obj_ref;          //Declare a systemwide object reference
obj_ref = …;                    // Initialize the reference to a distributed object
obj_ref-> do_something();       // Implicitly bind and invoke a method
```

(a)

```
Distr_object obj_ref;           //Declare a systemwide object reference
Local_object* obj_ptr;          //Declare a pointer to local objects
obj_ref = …;                    //Initialize the reference to a distributed object
obj_ptr = bind(obj_ref);        //Explicitly bind and obtain a pointer to the local proxy
obj_ptr -> do_something();      //Invoke a method on the local proxy
```

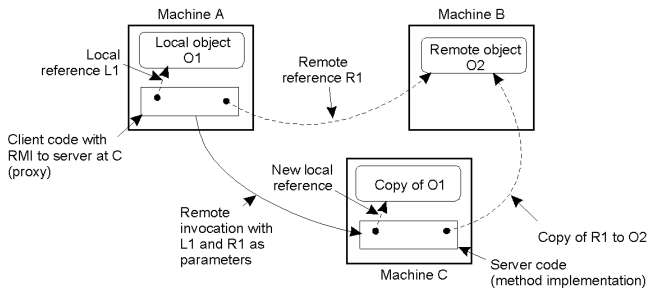
(b)

A. Example with implicit binding using only global references

B. Example with explicit binding using global and local references

Parameter Passing

- Less restrictive than RPCs.
  - Supports system-wide object references
  - [Java] pass local objects by value, pass remote objects by reference
  - Local objects: all normal classes; Remote objects: classes with RMIs (UnicastRemoteObject)



Part 3: Implementation & Examples

- Java RMI
- C RPC
- Python Remote Objects (PyRO)
- gRPC

Java RMI

- Server
  - Defines interface and implements interface methods
  - Server program
    - Creates server object and registers object with “remote object” registry
- Client
  - Looks up server in remote object registry
  - Uses normal method call syntax for remote methods
- Java tools
  - Rmiregistry: server-side name server

Java RMI Example

Interface

```
package example.hello;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

Client

```
String host = (args.length < 1) ? null : args[0];
try {
    Registry registry = LocateRegistry.getRegistry(host);
    Hello stub = (Hello) registry.lookup("Hello");
    String response = stub.sayHello();
    System.out.println("response: " + response);
} catch (Exception e) {
    System.err.println("Client exception: " + e.toString());
    e.printStackTrace();
}
```

Server

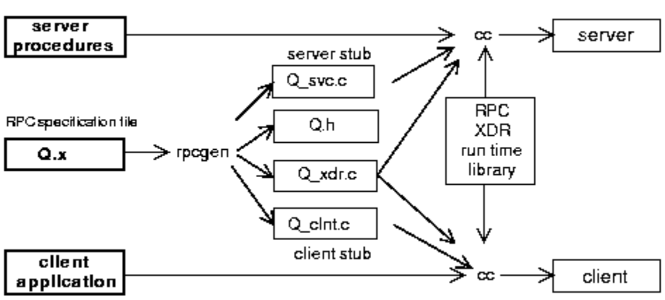
```
try {
    Server obj = new Server();
    Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);

    // Bind the remote object's stub in the registry
    Registry registry = LocateRegistry.getRegistry();
    registry.bind("Hello", stub);

    System.err.println("Server ready");
} catch (Exception e) {
    System.err.println("Server exception: " + e.toString());
    e.printStackTrace();
}
```

Java RMI and Synchronization

- Java supports Monitors: synchronized objects
  - Serializes accesses to objects
  - How does this work for remote objects?
- Options: block at the client or the server
- Block at server
  - Can synchronize across multiple proxies
  - Problem: what if the client crashes while blocked?
- Block at proxy
  - Need to synchronize clients at different machines
  - Explicit distributed locking necessary
- Java uses proxies for blocking
  - No protection for simultaneous access from different clients
  - Applications need to implement distributed locking

C/C++ RPC

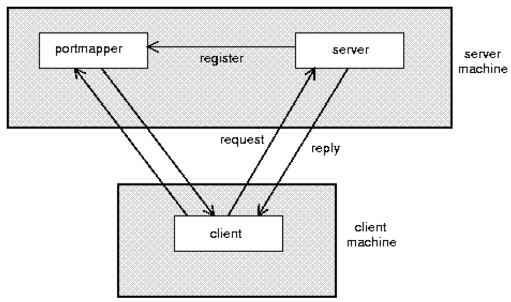
- Uses rpcgen compiler to generate stub code; link with server and client C code



- Q_xdr.c: do XDR conversion
- Sample code in lablet

Binder: Port Mapper

- Server start-up: create port
- Server stub calls *svc_register* to register prog. #, version # with local port mapper
- Port mapper stores prog #, version #, and port
- Client start-up: call *clnt_create* to locate server port
- Upon return, client can call procedures at the server



Python Remote Objects (PyRO)

```
import Pyro5.api

@Pyro5.api.expose
class GreetingMaker(object):
    def get_fortune(self, name):
        return "Hello, {0}. Here is your fortune message:\n" \
               "Behold the warranty -- the bold print giveth and the fine print taketh away.".format(name)

daemon = Pyro5.api.Daemon()             # make a Pyro daemon
uri = daemon.register(GreetingMaker)    # register the greeting maker as a Pyro object

print("Ready. Object uri =", uri)       # print the uri so we can use it in the client later
daemon.requestLoop()                    # start the event loop of the server to wait for calls
```

```
$ python greeting-server.py
Ready. Object uri = PYRO:obj_fbfd1d6f83e44728b4bf89b9466965d5@localhost:35845
```

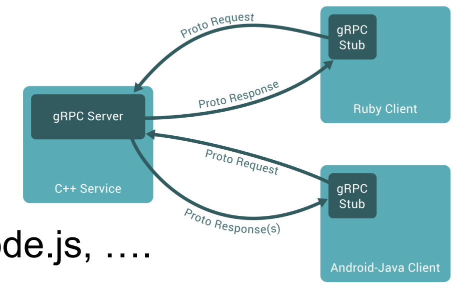
```
import Pyro5.api

uri = input("What is the Pyro uri of the greeting object? ").strip()
name = input("What is your name? ").strip()

greeting_maker = Pyro5.api.Proxy(uri)      # get a Pyro proxy to the greeting object
print(greeting_maker.get_fortune(name))    # call method normally
```

```
uri = daemon.register(GreetingMaker)   # register the greeting maker as a Pyro object
ns.register("example.greeting", uri)   # register the object with a name in the name server
```

```
greeting_maker = Pyro5.api.Proxy("PYRONAME:example.greeting")    # use name server object lookup uri
```

gRPC

- Google's RPC platform: now available to all developers
  - Modern, high-performance framework
  - designed for cloud apps
- Works across OS, hardware and languages
- Supports python, java, C++,C#, Go, Swift, Node.js, ….
- Uses http/2 as transport protocol
- ProtoBuf for *serializing structured* messages



Protocol Buffers (ProtoBuf)

- Allow message structure to be defined for communication
  - Platform-independent; marshalling/serialization built-in
- Define message structure in .proto file

```
message SearchRequest {
  required string query = 1;
  optional int32 page_number = 2;
  optional int32 result_per_page = 3;
}
```

- Use protocol compiler protoc to generate classes
  - Classes provide methods to access fields and serialize / parse from raw bytes e.g., set_page_number()
  - Like JSON, but binary and more compact
  - https://developers.google.com/protocol-buffers

gRPC Example

- Define gRPCs in proto file with RPC methods
  - params and returns are protoBud messages;

```
// The greeter service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

  - use protoc to compile and get client stub code in preferred language
  - gRPC server on server side

gRPC Features

- Four types of RPCs supported - see https://grpc.io/docs/what-is-grpc/
  - Unary RPC, server streaming, client streaming, bi-drectional
  - Unary RPC sends single response message, streaming can send any number of messages

```
rpc LotsOfReplies(HelloRequest) returns (stream HelloResponse);
```

```
rpc LotsOfGreetings(stream HelloRequest) returns (HelloResponse);
```

- Supports synchronous and asynchronous calls
- Deadlines/timeouts: client specifies timeout, server cn query to figure out how much time is left to produce reply
- Cancel RPC: server or client can cancel rpc to terminate it