

Last Class: RPCs

- RPCs make distributed computations look like local computations
- Issues:
 - Parameter passing
 - Binding
 - Failure handling



Today:

- Case Study: Sun RPC
- Lightweight RPCs
- Remote Method Invocation (RMI)
 - Design issues

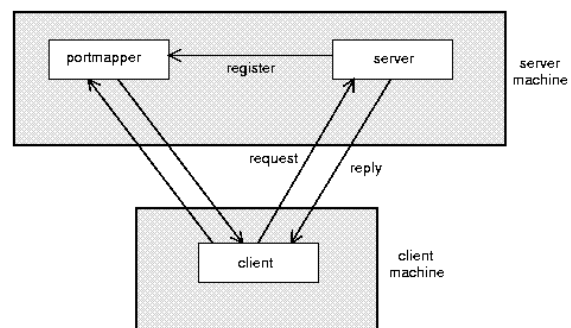


Case Study: SUNRPC

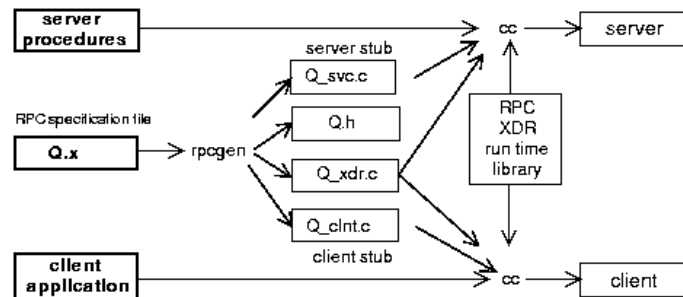
- One of the most widely used RPC systems
- Developed for use with NFS
- Built on top of UDP or TCP
 - TCP: stream is divided into records
 - UDP: max packet size < 8912 bytes
 - UDP: timeout plus limited number of retransmissions
 - TCP: return error if connection is terminated by server
- Multiple arguments marshaled into a single structure
- At-least-once semantics if reply received, at-least-zero semantics if no reply. With UDP tries at-most-once
- Use SUN's eXternal Data Representation (XDR)
 - Big endian order for 32 bit integers, handle arbitrarily large data structures

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



Rpcgen: generating stubs



- Q_xdr.c: do XDR conversion
- Detailed example: later in this course

Lightweight RPCs

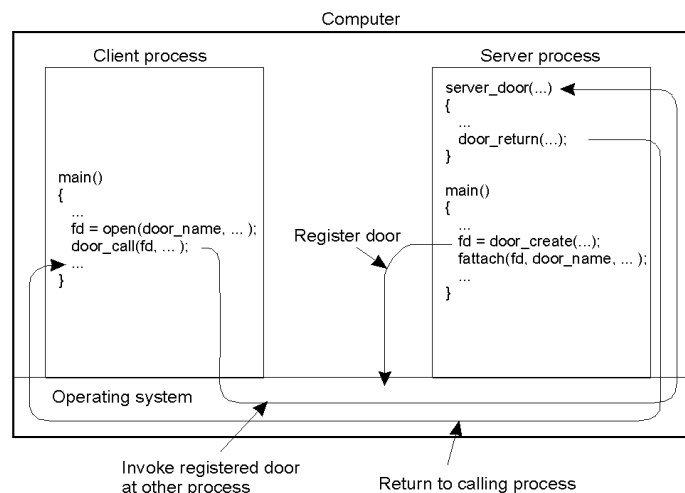
- Many RPCs occur between client and server on same machine
 - Need to optimize RPCs for this special case => use a lightweight RPC mechanism (LRPC)
- Server *S* exports interface to remote procedures
- Client *C* on same machine imports interface
- OS kernel creates data structures including an argument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



Doors



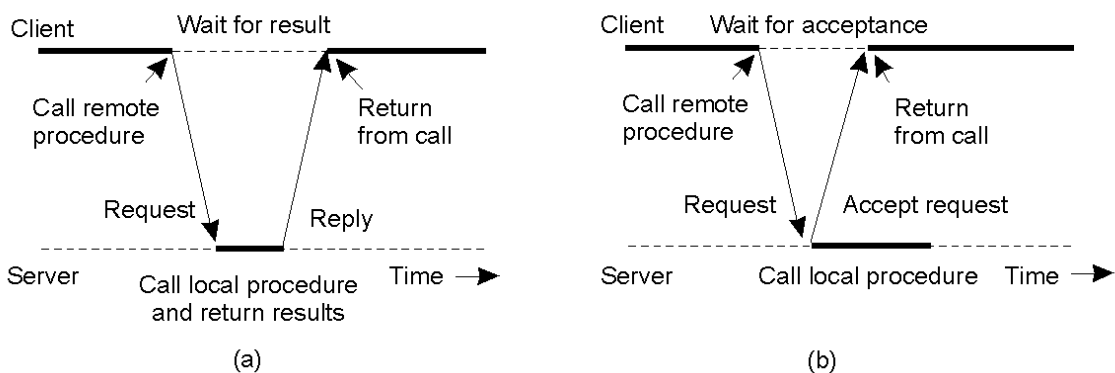
- Which RPC to use? - run-time bit allows stub to choose between LRPC and RPC



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).

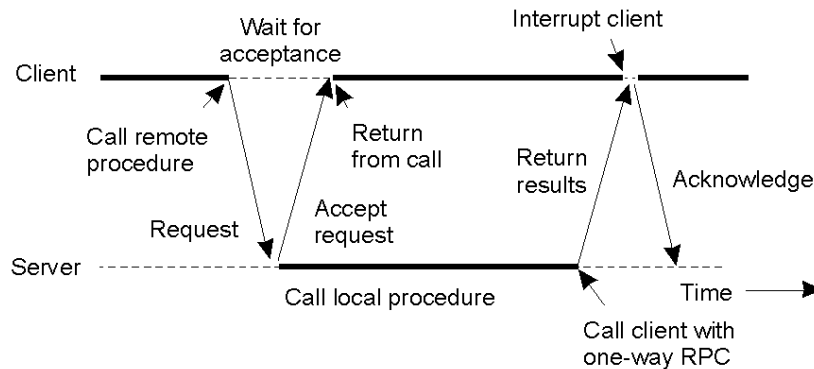
Asynchronous RPC



- a) The interconnection between client and server in a traditional RPC
- b) The interaction using asynchronous RPC

Deferred Synchronous RPC

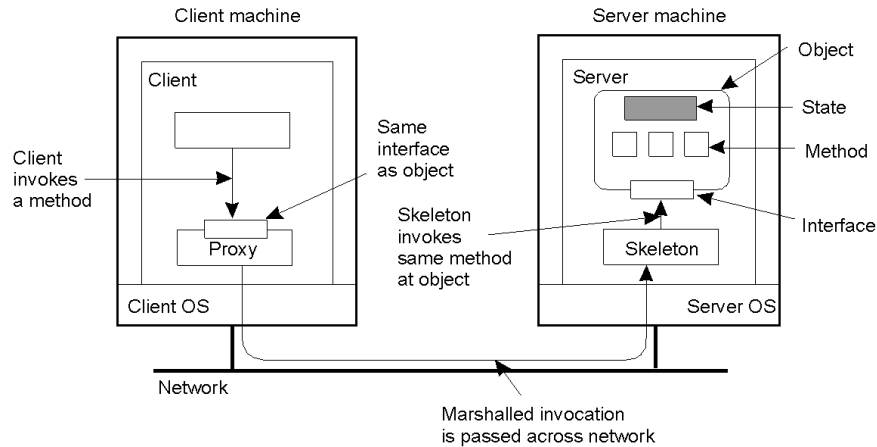
- A client and server interacting through two asynchronous RPCs



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 objRef;           //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
```

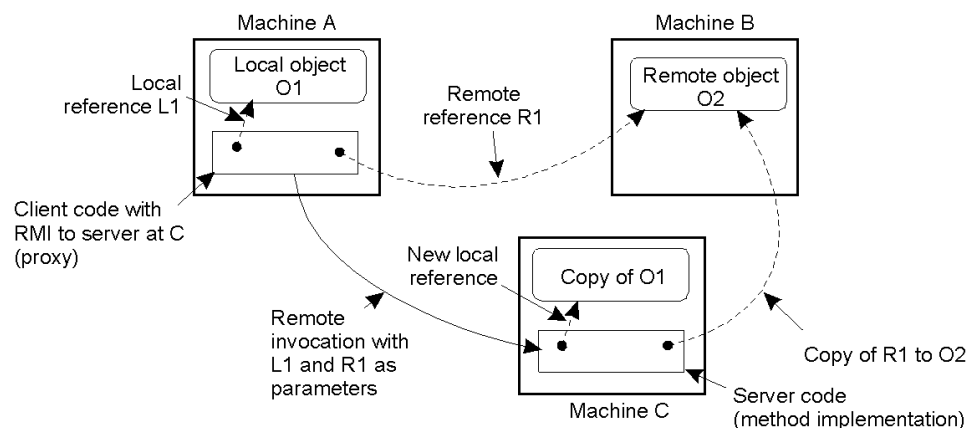
(b)

- a) (a) Example with implicit binding using only global references
- b) (b) Example with explicit binding using global and local references

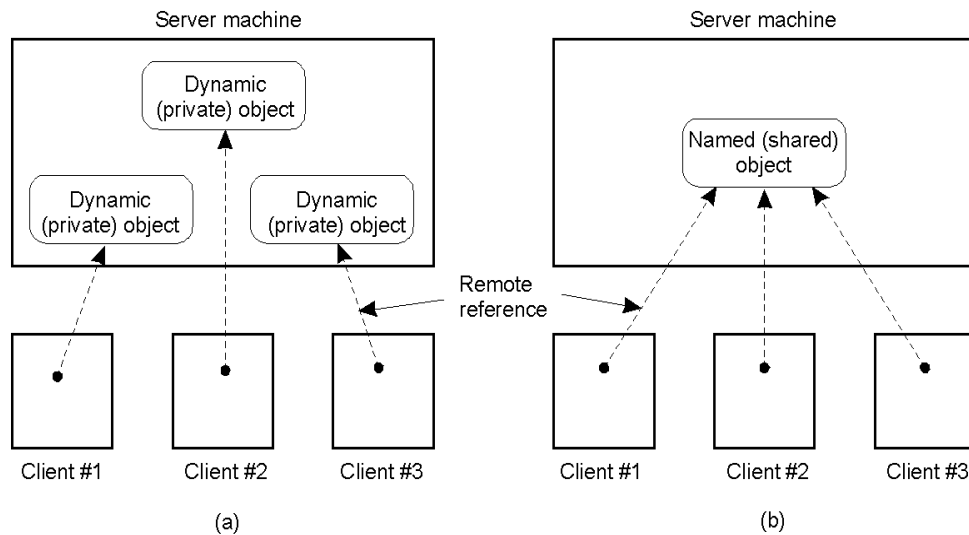


Parameter Passing

- Less restrictive than RPCs.
 - Supports system-wide object references
 - [Java] pass local objects by value, pass remote objects by reference



DCE Distributed-Object Model



- a) Distributed dynamic objects in DCE.
- b) Distributed named objects

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
 - Rmic: uses server interface to create client and server stubs

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

