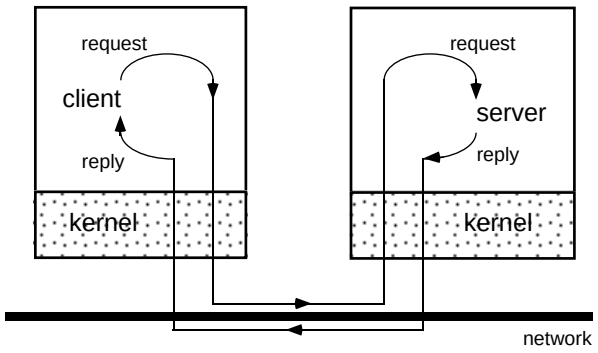
Client / Server Model using Message Passing (Review)



- Client / server model
  - *Server* = process (or collection of processes) that provides a *service*
    - Example: name service, file service
  - *Client* — process that uses the service
  - Request / reply protocol:
    - Client sends **request** message to server, asking it to perform some service
    - Server performs service, sends **reply** message containing results or error code

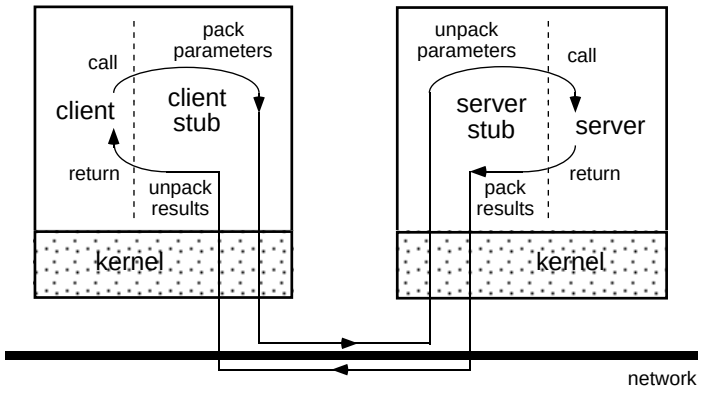
Why is Message Passing not Ideal?

- Disadvantages of client-server communication via message passing:
  - Message passing is I/O oriented, rather than request/result oriented
  - Programmer has to explicitly code all synchronization
  - Programmer may have to code format conversion, flow control, and error control
- Goal — *heterogeneity* — support different machines, different OSs
  - Portability — applications should be trivially portable to machines of other vendors
  - Interoperability — clients will always get same service, regardless of how vendor has implemented that service
  - OS should handle data conversion between different types of machines

Remote Procedure Call (RPC)

- RPC mechanism:
  - Hides message-passing I/O from the programmer
  - Looks (almost) like a procedure call — but client invokes a procedure on a server
- RPC invocation (high-level view):
  - Calling process (client) is suspended
  - Parameters of procedure are passed across network to called process (server)
  - Server executes procedure
  - Return parameters are sent back across network
  - Calling process resumes
- Invented by Birrell & Nelson at Xerox PARC, described in February 1984 *ACM Transactions on Computer Systems*

RPC Invocation



- Each RPC invocation by a client process calls a *client stub*, which builds a message and sends it to a *server stub*
- The server stub uses the message to generate a local procedure call to the server
- If the local procedure call returns a value, the server stub builds a message and sends it to the client stub, which receives it and returns the result(s) to the client

I/O Protection

- To prevent illegal I/O, or simultaneous I/O requests from multiple processes, the OS typically performs all I/O via privileged instructions
  - User programs must make a *system call* to the OS to perform I/O
- When user process makes a system call:
  - A *trap* (software-generated interrupt) occurs, which causes:
    - The appropriate trap handler to be invoked using the trap vector
    - Kernel mode to be set
  - The trap handler:
    - Saves process state
    - Performs requested I/O (if appropriate)
    - Restores state, sets user mode, and returns to calling program

RPC Invocation (More Detailed)

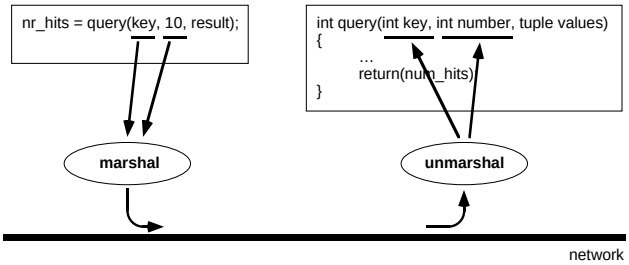
1. Client procedure calls the client stub
2. Client stub packs parameters into message and traps to the kernel
3. Kernel sends message to remote kernel
4. Remote kernel gives message to server stub
5. Server stub unpacks parameters and calls server
6. Server executes procedure and returns results to server stub
7. Server stub packs result(s) in message and traps to kernel
8. Remote kernel sends message to local kernel
9. Local kernel gives message to client stub
10. Client stub unpacks result(s) and returns them to client

Parameter Passing

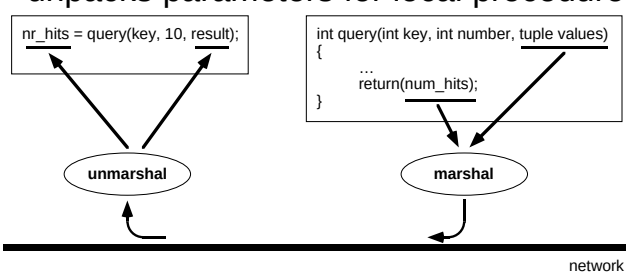
```
typedef struct {
        double item1;
        int item2;
        char *annotation;
} tuple;
```

```
char add(int key, tuple value);
char remove(int key, tuple value);
int query(int key, int number, tuple values[ ]);
```

- *Parameter marshaling* — client stub packs parameters into a message



- *Parameter unmarshaling* — server stub unpacks parameters for local procedure



Parameter Passing (cont.)

- Handle different internal representations
  - ASCII vs. EBCDIC vs. …
  - 1's comp. vs. 2's comp. vs. floating-point
  - Little endian vs. big endian
  - Establish a canonical (standard) form?
- What types of passing are supported?
  - Remote procedure can't access global variables — must pass all necessary data
  - *Call-by-value* (procedure gets a copy of data) — pass parameters in message
  - *Call-by-reference* (procedure gets a pointer to data)
    - Can't do call-by-reference
    - Do *call-by-copy / restore* instead
      - Instead of pointer, pass item pointed to
      - Procedure modifies it, then pass it back
    - Inconsistency if client doesn't block

Generating Stubs

- C and C++ may not be descriptive enough to allow stubs to be generated automatically

```
typedef struct {
        double item1;
        int item2;
        char *annotation;
} tuple;
```

```
char add(int key, tuple value);
char remove(int key, tuple value);
int query(int key, int number, tuple values[ ]);
```

  - Which are in, in-out, and out parameters?
  - Exactly what size are parameters (e.g., integers, arrays)?
  - What does it mean to pass a pointer?

- Using OSF's DCE Interface Definition Language (IDL) to specify procedure signatures for stub generation:

```
inerface db
{
typedef struct {
        double item1;
        long item2;
        [string, ptr]
          ISO_LATIN_1
          *annotation;
} tuple;
```

```
boolean add (
  [in]  long  key,
  [in]  tuple  value
);

boolean remove (
  [in]  long  key,
  [in]  tuple  value
);
```

```
long query (
  [in]  long  key,
  [in]  long  number,
  [out, size_is(number)]
        tuple  values[ ]
);
```


Binding

- *Binding* = determining the server and remote procedure to call
- Static binding — addresses of servers are hardwired (e.g., Ethernet number)
  - Inflexible if a server changes location
  - Poor if there are multiple copies of a server
- Dynamic binding — dynamically assign server names
  - Broadcast a "where is the server?" message, wait for response from server
  - Use a *binding server* (*binder*)
    - Servers register / deregister their services with the binding server
    - When a client calls a remote procedure for the first time, it queries the binding server for a registered server to call

Stateful vs. Stateless Server (Example = File Server)

- *Stateful server* — server maintains state information for each client for each file
  - Connection-oriented (open file, read / write file, close file)
  - ✔ Enables server optimizations like read-ahead (prefetching) and file locking
  - ✘ Difficult to recover state after a crash
- *Stateless server* — server does not maintain state information for each client
  - Each request is self-contained (file, position, access)
    - Connectionless (open and close are implied)
  - ✔ If server crashes, client can simply keep retransmitting requests until it recovers
  - ✘ No server optimizations like above
  - ✘ File operations must be idempotent