Remote Procedure Call (RPC)

"Making Remote Look Local"

Hiding the Network

Dealing with the network is complex

Can we hide this complexity?

Can we preserve "look and feel" of local programming?

Can we leverage programming language support?

- **in particular, the procedure abstraction**
- **use structure and constraints to reduce errors**
- **formalized in Java as "remote method invocation" (RMI)**

Short answer: "yes, but …"

Client-Server Model

How are distributed systems put together?

Many systems split into *client-server pairs*

- **server exports an *interface,* client invokes it**
- **server acts like an *abstract data type* (e.g. “class” in Java-speak)**

“Client” and “Server” are merely roles relative to one interface

- **by definition client-server interface is *asymmetric***
- **client is requestor of service, server is provider**
- **client-server relationship is per-interface instance**
  - **not per-host or per-process**
  - ***A* can be server and *B* client for interface *X***
  - **and *A* can be client and *B* server for interface *Y***

Client-server relationship can be *cascaded* (just like nesting)

- ***A* invokes *B***
- **in servicing *A*, *B* invokes *C* , and so on**

Interfaces

An interface consists of a *set of related operations*

- **wide range of possible interfaces**
- **any local interface is potentially a candidate**

Example: Printer spooler

- `queue file`
- `check file status`
- `delete print request`

Example: Lock manager

- `obtain read lock`
- `obtain write lock`
- `release lock`

Example: File system

- `create file`
- `delete file`
- …

Example: Time server

- `get-time-of-day`
- `set-time-of-day`

Historical Roots of RPC

Apparent opportunity

- **potential *parallelism* in a distributed system**
- **client and server on different machines**
- **no reason for client to block**

Early distributed systems used *message passing*

- ***A* sends invocation message to *B***
- ***A* does other work**
- ***A* polls for *B* 's reply or is interrupted**
- ***A* receives *B* 's reply**

But experience showed

- **often little for *A* to do while awaiting *B*'‘s reply**
- **explicit matching of replies complicates code**
- **parallelism better exploited via *threads***

Lesson

- **message passing offers greater generality**
- **but that generality not useful in common cases**

RPC is a paradigm born of these experiences

- **well suited for the client server model**
- **sacrifices some generality**
- **can be made very efficient**
- **simple to use**
- **emulates familiar *procedure call paradigm***

Characteristics of RPC

Two aspects

- ***control flow***
- ***invocation syntax***

Control flow

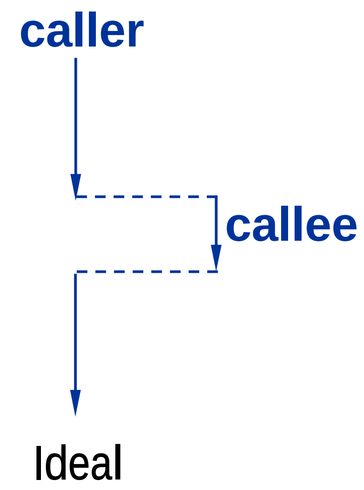
- **caller makes request and blocks**
- **callee services request and replies**
- **caller resumes**

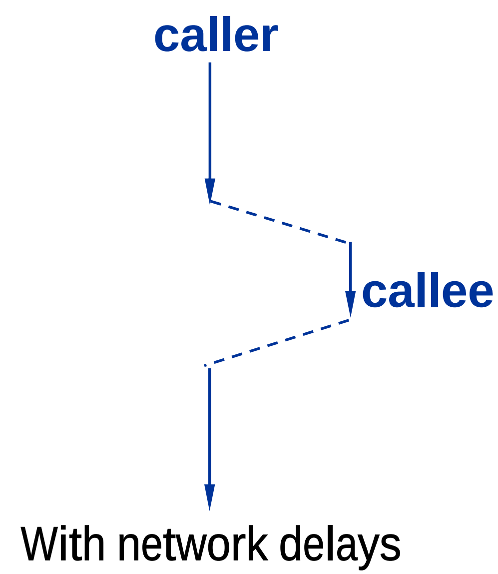
There is thus *synchronous transfer of control*

Invocation syntax

- **Local syntax: y = foo (x1, x2, ...)**
- **Remote syntax: y = foo (cid, x1, x2, ...)**
- **cid is called *connection handle***

Semantics and syntax can match host language





Limitations of RPC

How accurately can local procedure calls be emulated?

Client and server do not share address space

- ***can’t share global data***
- ***can’t use call-by-reference***
- **only call-by-value-result**
- **large data structures can be expensive**
- **embedded pointers in parameters useless**
- **procedure parameters don't work easily**
- **up-level addressing doesn't work easily**

Delayed binding in RPC

- **conceptually similar to dynamic linking**
- **hence runtime linking errors possible**

Multiple instances of callee

- **many servers may export same interface**
- **client may wish to talk to more than one server**
- **no counterpart in local procedure call**

Failure independence **of clients and servers**

- **in local case, client and server live or die together**
- **in remote case, client see new failure modes**
  network failure
  server machine crash
  server process crash
- **nontransitive communication**
  A **can talk to *B,* and *B* to *C,* but not *C* to *A***
- **failure handling code has to be more thorough (and more complex)**

Security Considerations

- **local calls normally within same security domain**
- **remote calls often cross domains**
- **remote RPC implementation may not be trusted!**
- **security concerns play a bigger role in RPC**

Overview of Typical RPC Mechanism

Two aspects: *control* and *syntax*

Control abstraction supported by a runtime system

- **RPC *transport* mechanism**
- **details depend on specific RPC package**

Typical client side transport routines

- makerpc (request-pkt, &reply-pkt)
  blocks until reply or failure detected

Typical server side transport routines

- getrequest (&request-pkt) **blocks until request arrives**
- sendresponse (reply-pkt) **sends reply**

Actual RPC packages

- **more parameters, such as timeouts**
- **more runtime routines**

Syntax abstraction supported by *stub* routines

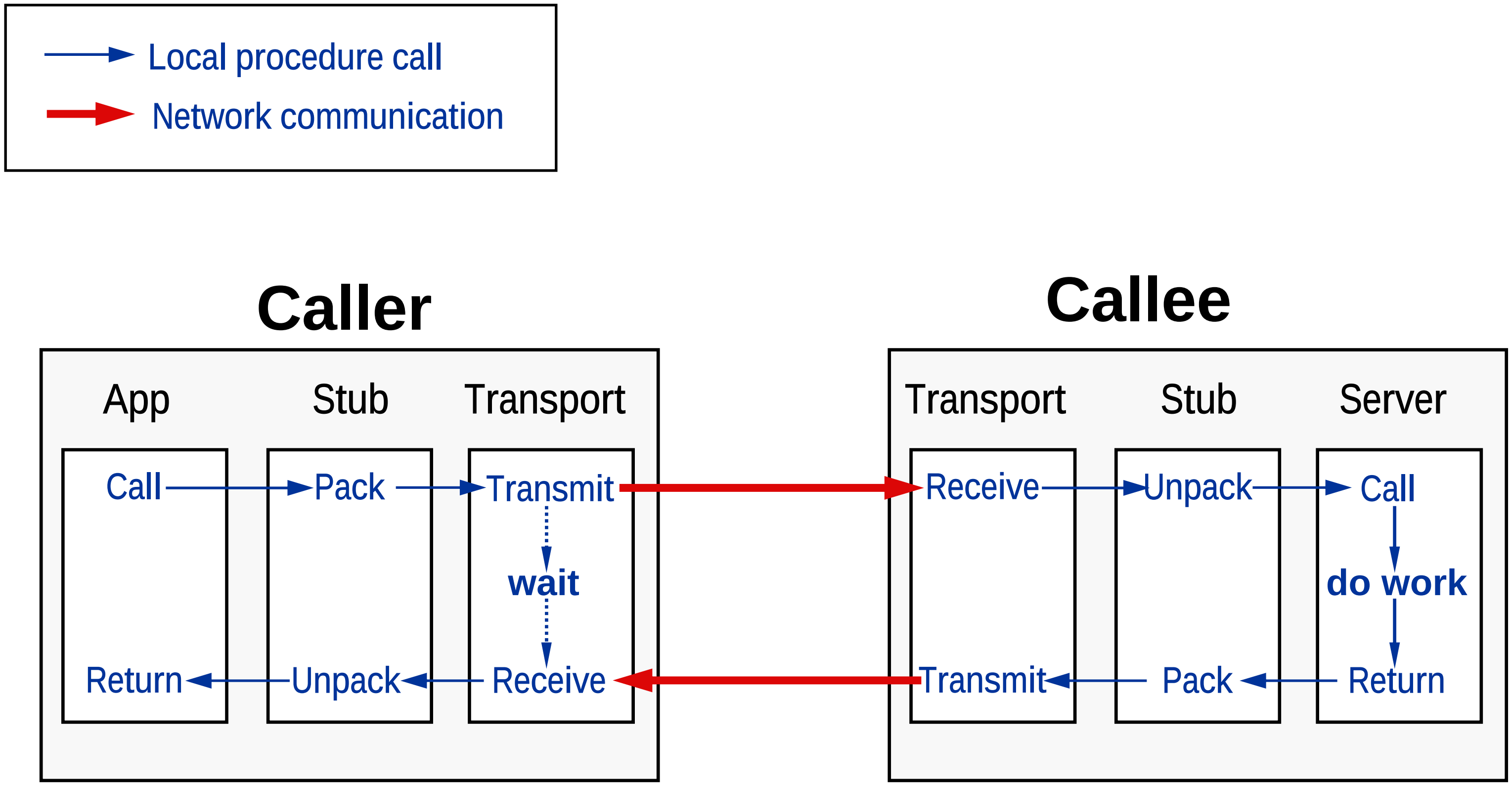
- **code generated by *stub generator***
- **input is *interface description***

Client Stub

- **invoked by user code as a local procedure**
- **packs into parameters into request-pkt (aka "marshall")**
- **invokes** makerpc()
- **unpacks reply-pkt into output parameters (aka "unmarshall")**
- **returns to user code**

Server Stub

- **invoked after** getrequest() **returns**
- **unpacks arguments**
- **demultiplexes opcode, invokes local server code**
- **packs arguments, invokes** sendresponse()
- **returns to top-level server loop**



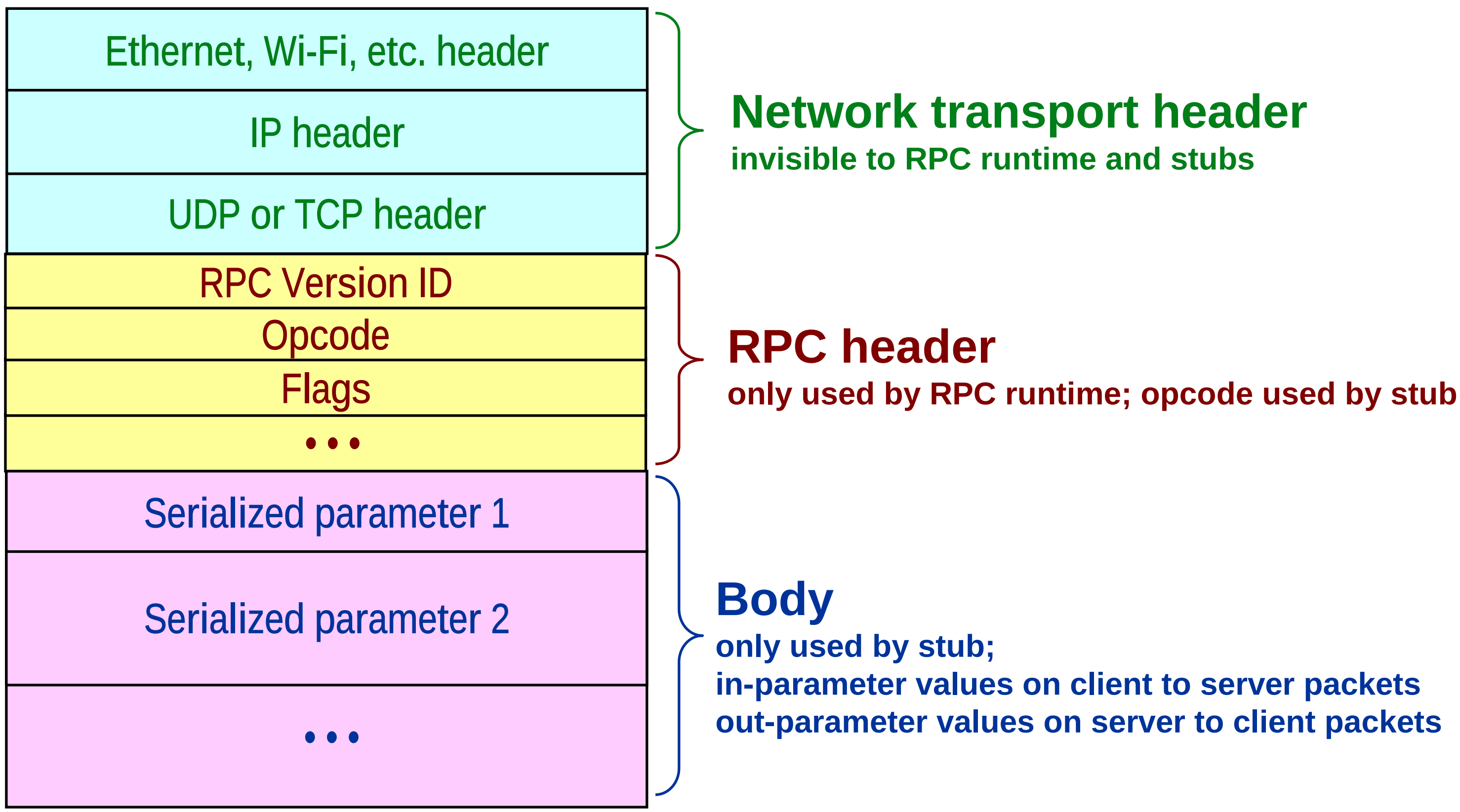
The packed format of an RPC parameter is called its *serialized* format

Converting to/from serialized format is serialization / de-serialization

RPC Packet Format

Details depend on specific RPC package

General format as follows



Stub Generation

Two problems

- **interface description**
- **code generation**

Relies upon compiler technology

- ***stub generator*** **is like a preprocessor**
- **outputs source code (e.g. in C), compiled by usual compiler**

Describing an interface

- **like specifying an abstract data type**
- **customized specification language**

Interface description provides

- **name and parameters of each procedure**
- **parameter usage: *in, out, inout***
- ***out*** **parms not included in request**
- ***in*** **parms not included in reply**

Client Stub

Client stub code structure

- **pack parameters**
- **invoke RPC transport, block, check for runtime errors**
- **unpack reply**

Example:

```
foo (in int32 a, inout int64 b, out int32 c) {
    pack a and b into packet p
    makerpc (p, &q);
    check for runtime errors
    unpack b and c from packet q
    return;
}
```

Server Stub

Typical server main loop

```
while (1) {
        get-request (&p);       /* blocking call */
        execute-request (p);  /* demux based on opcode */
}
```

Server stub code structure

- **unpack parameters**
- **examine opcode and demultiplex**
- ***invoke server procedure locally*** **(this is where real work gets done)**
- **pack reply packet**
- **invoke RPC transport to send reply**

Server Stub: Example

```
execute-request (packet *p, packet *q) {
  switch (p->opcode) {
      case 1:
            allocate memory for parms
            unpack in and inout parms for opcode 1
            proc1 (all parms);
            pack inout and out parameters for opcode 1
            deallocate memory
            send-response (q)
            break;
      case 2:
            similar code for opcode 2
      case 3:
            similar code for opcode 3
            . . . . . . . . . .
  }
}
```

Cross-language RPC

Client and servers may use different languages

- **Java client calling C server**
- **Python client calling Java server**
- …

Mapping of data types **has to be defined**

- **very hard to solve in full generality**
- **usually, restricted subset of types used in RPC**

Exception handling is another difficult problem