Remote Procedure Call Mechanisms

- based on the paper “Implementing Remote Procedure Calls” by Birrell and Nelson
- Tannenbaum & van Steen, Ch 2.2
- the paper is a classic reference on RPC – reading it gives you an idea how all RPC mechanisms are implemented

- later SUN RPC (needed for project), Java RMI…

Remote Procedure Calls (RPC)

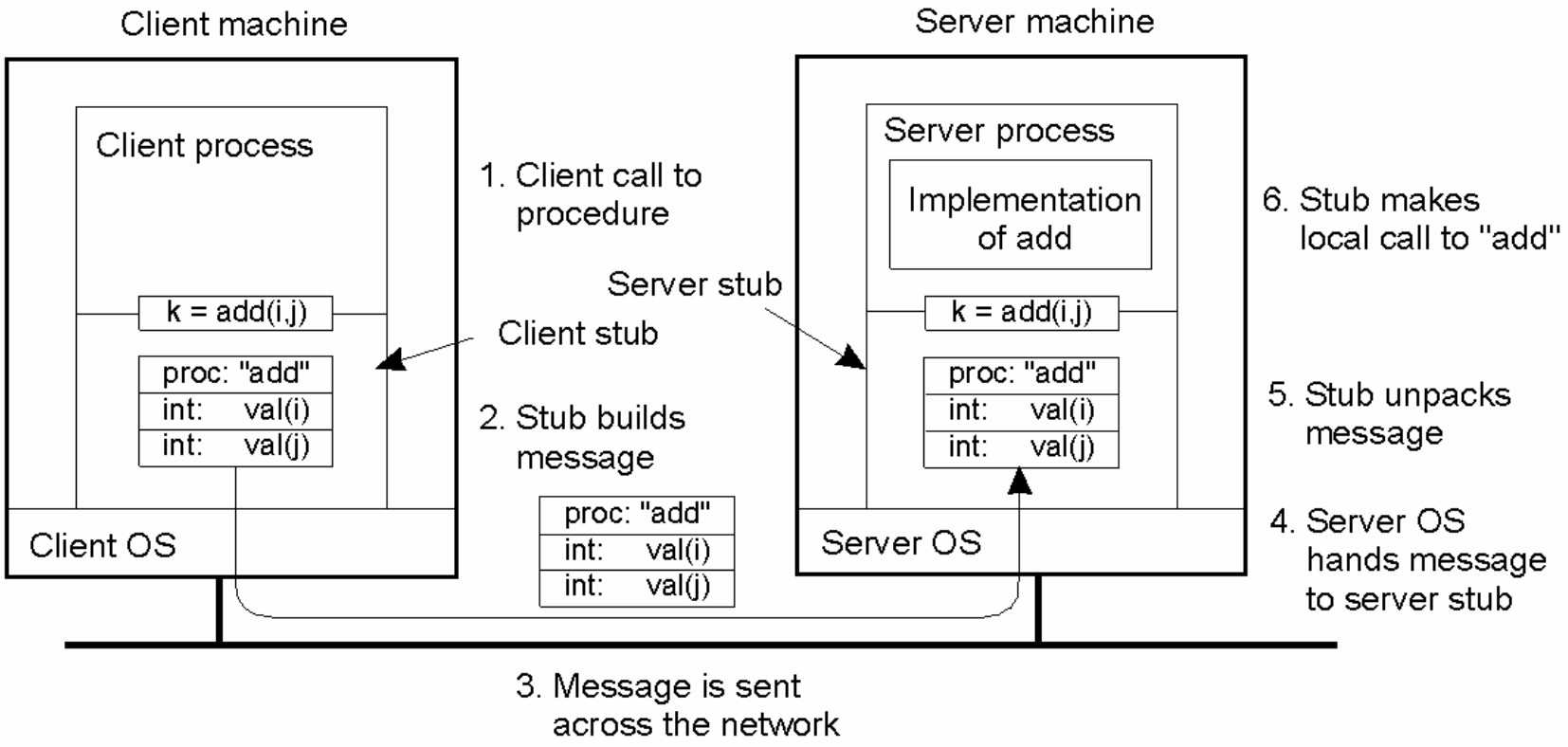
- we'll use term RPC to describe all mechanisms that all calling routines implemented on a remote site
- For example:
  - Sun RPC (for system level implementations), CORBA, DCOM, Java RMI (for application-level implementations, have object granularity)
  - These mechanisms are not qualitatively different
- In RPC communication b/w address spaces is structured similarly to procedure invocations

RPC Overview

- Features/advantages of RPC:
  - procedure calls
    - familiar synchronous semantics
  - type checking
  - automatic conversion b/w data representations
    - to ensure that data transferred correctly: e.g., integers may be little endian in one architecture and big endian in another
  - higher-level than network protocols
    - e.g., multiple transport layer protocols can be used (TCP, UDP, NetBios…)
    - handling of authentication, fault tolerance, specific call semantics, etc…

Structure of RPC Implementations

- Common parts: client, client stub, RPC runtime system, server stub, server



Typical communication pattern

- server binding occurs: the client finds the server that supports the desired functionality and establishes a connection
- calling a remote procedure results in a call to the user stub, client code blocks
- the user stub marshals the arguments and asks the RPC runtime to transmit them to the server
- the RPC runtime implements the actual transmission protocol on both sides
- when the data is transferred, the RPC runtime calls the server stub with the data
- the server stub unmarshals the data and calls the implementation of the procedure
- similarly for the results of the call

Marshalling/Unmarshalling implementation

- Marshalling/unmarshalling code, part of the user/server stubs is usually produced automatically from the interface specification
- The specification is expressed in an Interface Definition Language (IDL)
  - an IDL is used to define the types and the procedure interfaces that use these types
  - an IDL may have multiple encodings (i.e., mappings of data types into binary representations)
    - though commonly it has one
  - and IDL compiler translated the specification into marshalling and unmarshalling routines for a certain encoding

- In the case of a language agnostic mechanism, the IDL is entirely separate from the language type system
  - though it may look similar to some extent
- Example (XDR for Sun RPC)

```
struct data_in {
  string vstr < 128 >;
};
program MY_PROG {
   version MY_VERS {
     data_out MYPROC(data_in) = 1; /* proc# */
   } = 1; /* version# */
} = 0x31230000; /* service id */
```

- For language-specific RPC mechanisms, the IDL can be the same as the type language for the host language
  - same for the compiler

- Example (Java RMI)

```
public interface RS extends Remote {
  public int getL(String s)
      throws RemoteException;
  }
```

Binding Implementation

- Binding implementations are not easy to describe generically, but overall the essential components are:
  - a data base of services (protocol, version number, etc.) and the servers supporting them
    - possibly distributed
    - possibly with a search and registration protocol so that clients know nothing about the servers, or can be bound to any of a group of servers
  - a set of naming conventions so that clients can specify what service they want

Issues with RPC

- complexity
  - especially in the language independent mechanisms
- performance overhead
  - often hand-tuned data transfer is more efficient
  - due to data encoding and to the overhead of authentication and fault tolerance

Design options for RPC

- Handling of partial failure (machine and/or communication)
  - what are the exact semantics of the call? How can the user implement custom policies?
- Handling of pointer data when address space is not shared
  - disallow passing pointers or serialize pointers (i.e., traverse the data structures)?
- Integration of RPC in programming languages
  - using a library and an IDL
  - using a language-specific approach

Design options for RPC (2)

- “Binding”: how to connect to the server implementing the required service
- Marshalling and unmarshalling: what is the data encoding?
- Security and authentication mechanisms

Common options for partial failure handling

- Timeouts for remote calls?
  - the RPC system can have a timeout to abort a remote call after some time and/or repeat the request
  - or, the system can have no timeouts and rely on the user implementing multiple threads that one of them will detect that there is no failure
- Include transaction IDs for idempotency?
- Implement transaction protocols?

- Notify the user that a failure occurred
  - RPC call can return explicit error value
  - caller can be notified asynchronously, e.g., by raising exceptions or sending signals
  - the choice may be tied to the choice of language
- Decision often driven by objective that remote calls should have same (as much as possible) semantics as local calls

Implementation for Optimized Performance

- We will not get into implementation details here, but Birrell paper is an excellent starting point for that
- Some ideas:
  - retransmit until ack'd
  - reply packet is sufficient ack (optimization #1)
    - DCOM merges low level ack packets with actual data and pinging messages
  - first transmission does not need explicit ack, but retransmissions after timeout do (opt#2)
  - communication guides context switching (the network packets contain the process id of the process that accepts them - opt#3)

Sun RPC

- Based on Stevens's "UNIX Network Programming v.2", Ch. 16 (handout online)
- you will need to use Sun RPC for Project 3
- this lecture will only be an introduction to Sun RPC (as a case study of an RPC mechanism)

In Summary (Based on the Classification of RPC Mechanisms)

- Handling of pointer data when address space is not shared:
  - pointers are serialized (traverse data structures)
- Integration of RPC in programming languages:
  - Sun RPC is language-independent, uses a separate IDL (called XDR)
- "Binding": how to connect to a server implementing the required service:
  - services are identified and a daemon is running that knows about them
  - the user has to explicitly specify the machine where a service is to be found
- Security, authentication mechanisms: read the handout

Interface Definition Language: XDR

- Interface for a remote procedure that computes the square of a number

```
struct square_in {
long arg1;
};

struct square_out {
long res1;
};

program SQUARE_PROG {
  version SQUARE_VERS {
     square_out SQUAREPROC(square_in)
                                = 1; /* procedure # */
     } = 1; /* version # */
  } = 0x31230000; /* service id */
}
```

- Service id: identifies the service that is implemented (squaring)
- Version #: services can have multiple versions, each with a different interface
  - can the procedure numbers change?

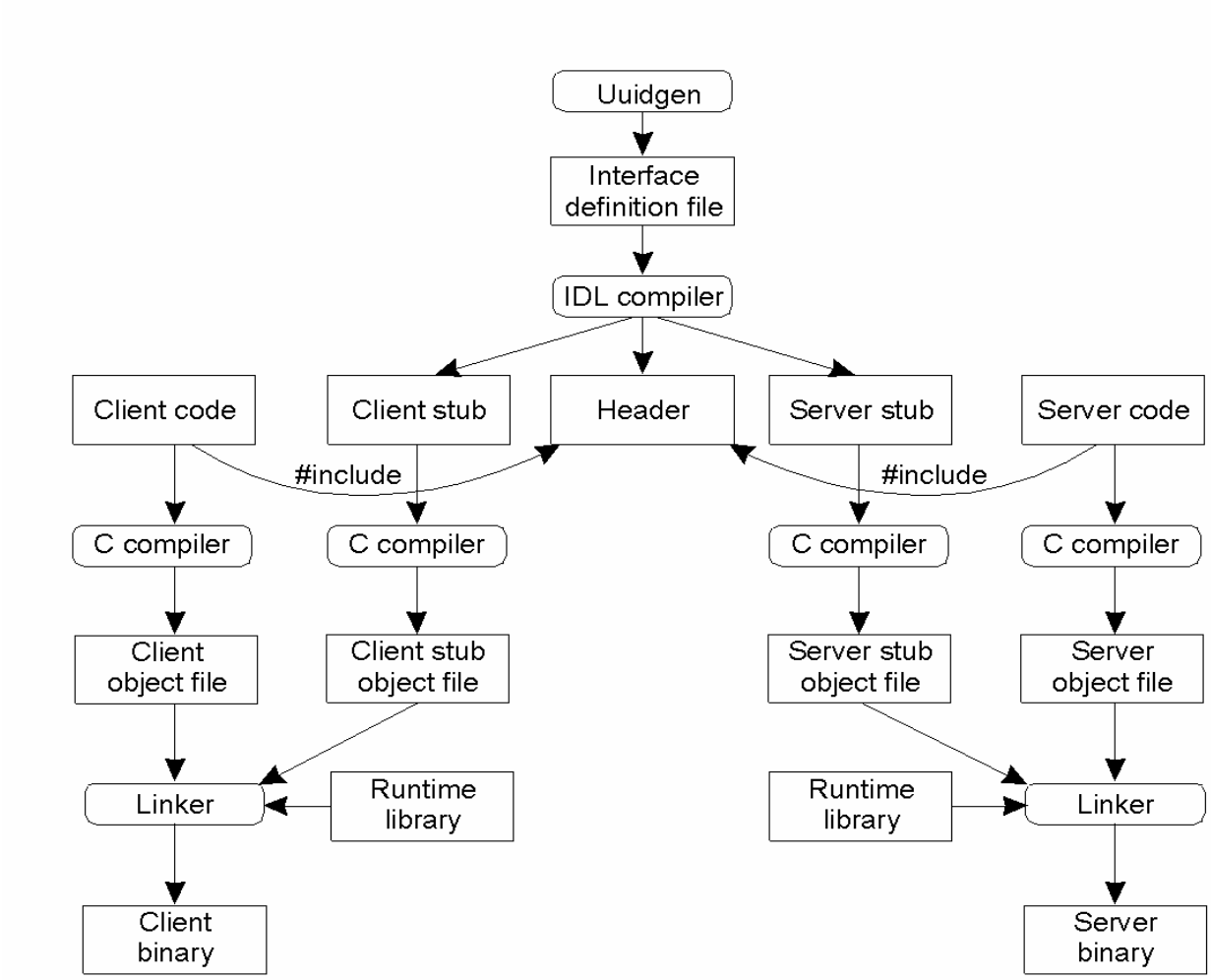
IDL Compilation

- The XDR compiler is called rpcgen
  - e.g., rpcgen -C square.x
  - "-C" forces ANSI C prototypes to be output
- Creates header files, code for the client and server stub, even a main routine for the server program
- In your client code you do:

```
CLIENT *cl; ...
cl = clnt_create(servhost,
                 SQUARE_PROG,
                 SQUARE_VERS,
                 "tcp");
...
outp = squareproc_1(&in, cl);
```

- SQUAREPROC for version #1 becomes squareproc_1
- See the handout for conventions and examples

Writing a Client and a Server



- The steps in writing a client and a server in DCE RPC.

RPC Runtime Issues

- Check out the services that are already running using rpcinfo -p
- Service id conventions:
  - 0x00000000 - 0x1fffffff : defined by Sun
  - 0x20000000 - 0x3fffffff : range to use
  - 0x40000000 - 0x5fffffff : transient
  - 0x60000000 - 0xffffffff : reserved

- Many runtime failures are recognized - check the return pointer of your call and handle the error
- You will probably have to use TCP for your project: UDP cannot transfer arbitrarily large amounts of data
  - like compressed JPEG images

- There are timeouts for both protocols:
  - for TCP the timeout is only for error notices
  - you cannot explicitly retransmit the request on timeout (or I don't know how to do it so that the retransmission is idempotent)
  - of course, within the timeout, TCP takes care of (idempotent) retransmissions
  - hence, if you want to persist on failure, just set the RPC timeout very high

```
struct timeval tv;
clnt_control(cl, CLSET_TIMEOUT, &tv);
```

XDR in Detail

- "XDR" refers both to the IDL and to the encoding of the IDL expressions
  - i.e., the binary representation of data
- Some of the rules:
  - all datatypes are encoded in multiples of 4 bytes
  - big endian is the transmission standard
  - two's complement is used for integers
  - IEEE format is used for floating point
- Example: a 5-character ASCII string would be:
  - 4 bytes for the length (5)
  - 5 bytes for the characters
  - 3 bytes for padding

Use of XDR

- All types defined in your .x file map to C/C++ types
  - because you have to supply the data in some form - you include a generated header file
  - (it does not have to be C - it can be whatever language your IDL compiler knows about)
- The binary encoding in memory is dictated by C/C++/etc. This has *nothing to do* with the binary encoding for the transmission
  - the transmission encoding is enforced by the marshalling code and is the same for all languages and architectures
- Type checking of the RPC runtime is different from C type checking (e.g., it will enforce maximum string lengths)
  - prevents buffer overruns (a security risk)

XDR Syntax

- For XDR syntax and corresponding C types, see p. 428 of your handout
- Interesting types:
  - const types
    - translated into #defines
  - hyper type
    - 64-bit integer
  - quadruple type
    - 128-bit float
  - opaque type: uninterpreted binary data
    - translates into C chars
  - variable-length types (e.g., int dat<80>)
    - translates into a structure with “len” and “val” fields
  - except for strings: string line<80> maps to a C pointer to char
    - stored in memory as a normal null-terminated string
  - encoded (for transmission) as a pair of length and data

XDR Routines

- For every XDR type declares (e.g., `square_in`, in our example) there is an XDR routine generated automatically (e.g., `xdr_square_in`)
- The XDR routine is the front-end to both the marshalling and the unmarshalling process
- You can see how it is used explicitly in p. 434, p. 436 of your handout
- The XDR routine is often passed around in RPC code

Cleaning Up

- After the procedure returns and the results have been sent to the client, the server stub calls a (user-defined) cleanup routine
  - e.g., `square_prog_1_freeresult`
    - convention: this is program `SQUARE_PROG, version 1`
  - see p. 410 of your handout
  - why is this program-specific and not procedure-specific?
    - the second argument is the XDR routine for the result being freed

- Data that have been dynamically allocated by the XDR code (stubs) are deallocated using xdr_free
  - see p. 410, p. 436 of your handout
  - but you don't really need to learn the details of explicit marshalling and unmarshalling
  - the first argument is the XDR routine for the data being freed
  - xdr_free works recursively (based on the information from the XDR routine)
  - but what about user-allocated data?
  - This all works in theory—try to find out if it works in practice!