Procedure calls and RPC

Procedure call:
well understood mechanism for
transfer of control and data within a program
on a single computer

Why not use the same mechanism for
transfer of control and data across a communication network?

⇒ Development of Remote Procedure Call
The basic idea

- Procedure call
- Procedure suspend
- Procedure continue
- Procedure return

Machine 1 (Caller)

Machine 2 (Callee)
Aims behind RPC

Make distributed computation easy, by removing unnecessary difficulties,
leaving only fundamental difficulties of building distributed systems, i.e.,

Timing
Independent failure of components
Coexistence of independent execution environments
Other performance issues

Call must be efficient (e.g. they considered that it must be within factor of 5 beyond the necessary transmission times of network)

Make semantics of RPC as powerful as possible without loosing simplicity or efficiency else the gains of a single unified communication paradigm are lost
Failure semantics

- At least once
- At most once
- Exactly once
Design issues involved

- Precise semantics of a call in presence of a communication failure
- Semantics of address containing arguments in absence of shared address space
- Integration of RPC into an existing programming environment
- Binding procedure (how a caller determines the location and identity of callee)
- Activation
- Data and control transfer protocol
- Data integrity and security in network
Easy to use?

What’s the guiding principle?

Make the semantics of remote procedure calls as close to local procedure calls as possible

E.g. no timeout mechanism (in absence of machine or communication Failure)
Structure

Five pieces of program involved in one call:

- User
- User stub
- RPC Runtime (communications package)
- Server stub
- Server

Normal local call
Components of an RPC system and interactions

User ➔ Local call ➔ pack args ➔ transmit ➔ receive ➔ unpack result ➔ Local return

Client machine ➔ user stub 1 ➔ RPC ➔ runtime ➔ server stub ➔ server ➔ Callee machine

n/w

call packet ➔ result packet

Call ➔ work ➔ return
The Binding mechanism

How does a client of the binding mechanism specify what is wants to be bound to?

How does the caller determine the machine address and specify to the callee the procedure to be invoked?

Naming scheme and location
Naming Scheme

Interface are exported and imported

2 components of a name:

Type
Instance

E.g. type: mail server, instance: a specific mail server
Location: Grapevine Distributed database

Available from all sites
Data is Replicated
Highly reliable
Supports 2 types of entries:

Individual entries
Group entries
Entries in the location database

- **RName**: connect site network address
- For an individual entry: one **RName**
- For a group entry: member list (list of Rnames)
- For interface types: Group entries
- For instances: individual entries
Export/Import procedure

User  client-stub  |  RPC  |  Runtime  |  server-stub  |  server
User  Client machine  |  return  |  record result  |  get connect address  |  Bind (A,B)  |  record in table  |  set connect address  |  add member  |  table uid lookup  |  export (Typ, ins, dis)  |  export (Typ, ins)  |  return
Building an RPC infrastructure and the program development process

The following proposal has been iteratively worked out in class through your answers to my questions
Implement an RPC system?

Client:

\[ \ldots \]
\[ X = \text{foo}(i); \]

Server:

\[
\begin{align*}
\text{int } & \text{foo}(\text{int } x) \{ \\
& \text{return } x**x; \\
& \}
\end{align*}
\]

for this high level view
A proposal

Client:

main() {
    ....
    int foo-everest(int);
    X = foo(i);
}

C syntax is changed; also ease of use is in danger!

Server:

int foo (int x) {
    return x**x;
}
Another proposal

Client::
int foo (int i) {
    Rpc("everest");
}
Rpc(char *name) {
}

Server:
int foo (int x) {
    return x**x;
}

looks better, but is this sufficient?
A workable proposal: with 1 input argument, 1 return result

Client:
int foo (int i) {
    Int returnval;
    Rpc("everest:int:int", i, &returnval);
    return returnval;
}
Rpc(char *name, ...

Server:
int foo (int
    x) {
    return x*x;
}
With 2 input arguments

Client:
```c
int foo (int i, int j) {
    int retval;
    Rpc("everest:int:int:int",
         i, j, &retval);
    return retval;
}
```

Server:
```c
int foo (int x) {
    return x*x;
}
```

reusable part of client stub
Making it independent of machine name, and adding fn name

Client::

    int foo (int i, int j) {
        int retval;
        Rpc("foo:int:int", i, j, &retval);
        return retval;
    }

    Rpc (char *name, ..) {
    }

but how do you bind to server?

Server:

    int foo (int x) {
        return x*x;
    }
Making it independent of machine name, and adding fn name

Client::
int Poo (int i, int j) {
    int retval;
    Rpc("Poo:int:int:int", i, j, &retval);
    return retval;
}

Rpc (char *name, ...) {
    Char ip_port[32];
    s = bind(name);
    ....
} Char * bind (char *name) {
Char ip_port[32];
    s = Connect("10.105.1.3", 7777);
    s.send(name, "LOOKUP");
    s.read(ip);
    Return ip_port;
}

Bind server/name server::

Main () {
Char ip_port[32];
    while (1) {
        socket serv = listen(7777);
        serv.read(char *arg, char *request);
        ip_port = tablelookup (request);
        serv.send(ip_port);
    }
}
Turning bind/name server into an RPC server

Client::
int foo (int i, int j) {
  int retval;
  Rpc("foo:int:int", i, j, &retval);
  return retval;
}

Rpc (char *name, ..) {
  Char ip_port[32];
  If (name=="lookup"
    ip_port=getbindserver_ip_port();
  Else ip_port = lookup (name);
  }
  Char * getbindserver_ip_port() {
    }

Bind server/name server::
Table t;
Char *Lookup (char *fnname) {
  ...
  return (t.fetch (fnname))
}
Int register (char *fnname, char ip, int port) {
  return t.add(fnname, ip, port);
}
Separating reusable component from client code

Client.c
... x = foo (10); ..
}

ClientStub.c
int foo (int i, int j) {
int retval;
Rpc("foo:int:int:int", i, j, &retval);
return retval;
}

Rpc.c
Rpc (char *name, ..) {
Char ip_port[32];
If (name=="lookup"
    ip_port=getbindserver_ip_port();
Else ip_port = lookup (name);
} Char * getbindserver_ip_port() {

Bindserver.c
Table t;
Char *Lookup (char *fname) {
    ... return (t.fetch (fname))
}
Int register (char *fname, char ip, int port) {
    return t.add(fname, ip, port);
}
Focusing on Rpc.c

Char * GLOBAL_bind_ip_port;

Rpc (char *name, ...) {
  Char ip_port[32];
  If (name=="bindserv"
    ip_port=getbindserver_ip_port();
  Else ip_port = lookup (name);
  ......
}

Char * getbindserver_ip_port() { return GLOBAL_bind_ip_port;

Client.c again: with bind server called automatically from within foo to locate foo

#include <rpc.h>
Main () {
    setbindserver(servip, port, AUTO II
    PERCALL/ONCE);
    x = foo (10);
    y = foo2(...);
}

Client.c again: with bind server called explicitly from client to locate foo

```c
#include <rpc.h>
Main () {

    setrpcsсолver(bindserverip, bindserverport);
    <fooserverip,fooserverport> = lookup("foo")!

    setrpcsсолver(fooserip,fooserport);
    x = foo (10);
    y = foo2(...); // local or remote

} Setting server addresses: parameter passing vs. external variables (side effects)
```
Change rpc.c accordingly

- Pass on server ip and port to low level networking routines, through stub
- The stub sees these values as set by client.c through a side effect
- OO paradigms do better on this aspect
Focusing on Rpc.c

Rpc (char *name, server, port..) {
  Char ip_port[32];
  Pack arguments into one byte(char) * packet
  Call low level socket layer
  Receive result (through low level networking call)
  Return;
}
Client.c again

#include <rpc.h>
Main () {

  setrpcserver(servip, port);
  x = foo (10);

  setrpcserver(servip, port);
  y = foo2(…);
}

Client side RPC infrastructure: Local call vs. rpc call

 RpcClient.h

 include

 include

 Client.c
 Includes Foo.h

 Link together

 RpcClient.c
 Reusable for All servers

 Foo.h

 Foo_stub.c

 Stub generator

 Generated client stub
Server.c

Main() {
    initRPC();
    export_foo();
    serviceloop();
    Program never reaches this point
}

Rpcinit and serviceloop

Rpcinit: create serverport

Serviceloop:
  conn=Listen on serverport
  receive request packet on conn
  fn-name = packet.extractname;
  returnpacket=dispatch(fn-name,packet);
  send(retrunpacket) on conn
  repeat serviceloop

Fn_name includes signature
Export_foo

addentry("foo", (fn_ptr *) stub_foo, "int:int");
Dispatch(char *fnname, byte*packet);

(fnptr *) stub = lookup(fnname);
Char *returnpacket = 
((fn *)(char *))(stub)(packet);
Return returnpacket;
Stub_foo(byte *packet)

Local arg1, ...argn
Local retval
Unpack arg1...argn from packet
retval = Foo (arg1...argn)
Char *ret=Pack (retval)
Return ret;
Server Side Development process

 RpcServer.h
          include
          
          Foo.h
          include
          
          Server.c
          Includes Foo.h
          
          RpcServer.c
          Reusable for All clients
          Link together
          
          foo.c

 Stub generator

 Generated server stub

 Stub_foo.c
          include

 foo.h
          input
Using name service at server side

- Use a name server as an RPC server to register the server ip and port for a given function.

- This is similar to its counterpart in client, where we used the name server as an RPC server to perform a lookup operation to obtain server ip, port for a given name of a function.
Class proposal ends
Readings

Andrew Birrell and BJ Nelson: Implementing Remote Procedure Calls, ACM TCOS Feb 1984