Remote Procedure Calls

Paulo Sérgio Almeida

Grupo de Sistemas Distribuídos
Departamento de Informática
Universidade do Minho

Remote Procedure Calls – RPC

- Client-server applications can be developed in many ways.
- Direct use of send-receive primitives is considered by some as too low level.
- The idea of RPC (Remote Procedure Call) is to make distributed applications more similar to traditional ones.
- An RPC looks to client code as a normal local procedure invocation, and is implemented in the server as a normal procedure that returns some result.

Mechanics of an RPC

From the seminal paper by Birrel and Nelson “Implementing Remote Procedure Calls” (1984):

- When a remote procedure is invoked, the calling environment is suspended,
- the parameters are passed across the network to the environment where the procedure is to execute . . . and the desired procedure is executed there.
- When the procedure finishes and produces its results, the results are passed backed to the calling environment, where execution resumes as if returning from a simple single-machine call.

Mechanics of an RPC

- Five components: client, client-stub, RPC runtime, server-stub (or skeleton), server.
- When client performs RPC, a local call is made to a procedure in the client-stub;
- Client-stub marshalls arguments, builds message;
- Client runtime sends it to server and blocks;
- Server runtime receives messages, passes it to server-stub;
- Server-stub unmarshalls arguments and invokes local procedure in the server;
- When server procedure returns, the opposite path is made until result is returned to client.

Binding

- How does a client know which machines provide the service?
- Using fixed addresses in client code is undesirable.
- A Name Service can be used.
- Servers export interfaces (set of procedures) to name service;
- Clients can lookup by name or by type of interface.
- Name service translates names to network addresses.

Interface Definition Languages – IDL

- Some languages have RPC as a built-in notion.
- In most cases remote services are described through an interface definition language (IDL).
- In IDL the interface of a service describes the operations, their parameters, result and relevant datatypes.
- An IDL may be neutral and independent from the languages used in client and server.
- In this last case *language mappings* are defined for each language used.

IDL compiler

A IDL compiler takes an IDL file and generates code to be used together with programmer written client and server code, typically:

- Client-stub.
- Server-stub (skeleton).
- Client and server runtime, including server main loop.
- Datatype marshalling code, used by both stubs.
- Header files to be imported by programmer written code.

Parameter passing

- In RPC there are no global variables to both client and server.
- Parameter passing and result must be by value.
- Normally a parameter can be declared as in / out / inout.
- This is useful for large values (e.g., arrays) passed by reference in the language used.
- Pointers to local structures make no sense on the other machine.
- Deep-copying of linked structures commonly supported.

Request-reply protocol

- RPCs are implemented by a request-reply protocol.
- Can be TCP or UDP based, TCP more common.
- UDP gives more control for a custom handling of acknowledgment, duplicates, retransmissions, “connections”.
- Birrel and Nelson proposed such a custom handling.

Types of faults

Several things can go wrong performing RCPs:

- Client fails to locate server;
- Request is lost, not reaching the server;
- Server crashes after receiving request, before replying;
- Reply is lost, not reaching the client;
- Client crashes.

Semantics under faults

Different request-reply protocols can give different guarantees in terms of how many times a request is executed under faults:

Exactly-once the ideal desired situation, but difficult to achieve;

Maybe no guarantees; e.g., when under possible message loss, if no reply arrives the request is not resent;

At-least-once when under message loss or delay in the reply, the requests are resent; the server does not recognise them as duplicates and may re-execute them; should only be used for idempotent operations.

At-most-once when under message loss or delay in the reply, the request is resent; the server, however, recognizes them as duplicates and does not re-execute them; the more expected semantics.

Measures against faults

- Maintain incarnation numbers in client and server, to detect crashes;
- Use sequence numbers in requests for each pair client-thread; (each client thread only sends next request after receiving reply)
- Maintain in server, for each client thread, last sequence number received, to detect duplicates;
- Maintain in server, for each client thread, last result not yet acknowledged; when reply is lost and client resends request.

Case study: ONC RPC

- Created by Sun, later called ONC RPC (Open Network Computing Remote Procedure Call), now specified by RFC 1831.
- Remote services exported by a machine are organized by *programs*, each identified by a number.
- Each program can have several versions, to allow evolution without breaking existing code.
- Each version of a program defines a set of procedures.
- Services register in the *port mapper*, which maintains associations of program-version-protocol tuples to ports.

ONC RPC: IDL

- The XDR (External Data Representation) serialization standard includes a data description language, and was exendend as the IDL for ONC RPC.
- A XDR fragment could be:

```
struct nametel {
  string name<>;
  int tel;
};
...
program addressbook {
  version v1 {
    void insert(nametel) = 1;
    int lookup(string) = 2;
    ...
  } = 1;
} = 0x21234567;
```

ONC RPC: IDL compiler — `rpcgen`

ONC RPC makes available an IDL compiler, `rpcgen`, which given an XDR file `prg.x` generates:

- an header file `prg.h` containing datatype declarations and function protoptypes, to be included by client and server code.
- the client stub `prg_clnt.c`; for each version `v` of a procedure `proc`, defines function `proc_v`.
- server program `prg_svc.c`, containing the server `main`, which registers in the port mapper, and code to dispatch requests to the programmer defined procedures `proc_v_svc`.
- a file `prg_xdr.c` containing *XDR filters*, to serialize the datatypes defined in `fich.x`.

ONC RPC: language mapping from XDR to C

XDR defines a mapping to the C language. For datatypes we have:

XDR	C
`const c = v;`	`#define c v`
`typedef decl;`	`typedef decl;`
`struct name { ...};`	`struct name { ...}; typedef struct name name;`
`enum name { ... };`	`enum name { ... }; typedef enum name name;`
`bool var;`	`bool_t var;`
`type *var;`	`type *var;`
`string var<n>;`	`char *var;`
`opaque var[n];`	`char var[n];`
`opaque var<n>;`	`struct { u_int var_len; char *var_val; } var;`
`type var[n];`	`type var[n];`
`type var<n>;`	`struct { u_int var_len; type *var_val; } var;`

ONC RPC: example

- XDR file:

```
program math {
  version v1 {
    long square(int) = 1;
  } = 1;
} = 0x20000001;
```

- Client program (fragment):

```
CLIENT *clnt; int i = 5; long *lp;
clnt = clnt_create ("127.0.0.1", math, v1, "tcp");
lp = square_1(&i, clnt);
clnt_destroy (clnt);
```

- Procedure implementation:

```
long * square_1_svc(int *ip, struct svc_req *rqstp)
{ static long l;
  l = *ip * *ip;
  return &l;
}
```