Remote Procedure Call Protocols
Local Procedure Calls

// Client code
... result = procedure(arg1, arg2);
...

// Server code
result_t procedure(a1_t arg1, a2_t arg2) {
... return(result);
}

Remote Procedure Calls (1)

// Client code
.
result = procedure(arg1, arg2);
.

// Server code

result_t procedure(a1_t arg1, a2_t arg2) {
.
    return(result);
}

// Client code
.
result = procedure(arg1, arg2);
.

// Server code

result_t procedure(a1_t arg1, a2_t arg2) {
.
    return(result);
}
Remote Procedure Calls (2)

// Client code
...
result = procedure(arg1, arg2);
...

// Server code
result_t procedure(a1_t arg1, a2_t arg2) {
  ...
  return(result);
}

Client-Side Stub

Server-Side Stub
Block Diagram

Client

Application

Stub

RPC support code

Transport protocol

Server

Remote procedure

Stub

RPC support code

Transport protocol
ONC RPC

- Used with NFS
- eXternal Data Representation (XDR)
  - specification for how data is transmitted
  - language for specifying interfaces
Example

typedef struct {
    int    comp1;
    float  comp2[6];
    char   *annotation;
} value_t;

typedef struct {
    value_t   item;
    list_t    *next;
} list_t;

bool add(int key, value_t item);
bool remove(int key, value_t item);
list_t query(int key);
Placing a Call

```c
result = add(key, item);

bool add(int k, value_t v) {
    ...
    return(result),
}
```
Returning From the Call

result = add(key, item);

char add(int k, value_t v) {
    ...
    return(result);
}

unmarshal

marshall

Wire
### Marshalled Arguments

<table>
<thead>
<tr>
<th>int</th>
<th>key</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>comp1</td>
</tr>
<tr>
<td>float (1)</td>
<td>comp2</td>
</tr>
<tr>
<td>float (2)</td>
<td>item</td>
</tr>
<tr>
<td>float (3)</td>
<td>annotation</td>
</tr>
<tr>
<td>float (4)</td>
<td></td>
</tr>
<tr>
<td>float (5)</td>
<td></td>
</tr>
<tr>
<td>float (6)</td>
<td></td>
</tr>
<tr>
<td>string length</td>
<td></td>
</tr>
<tr>
<td>string (1 – 4)</td>
<td></td>
</tr>
<tr>
<td>string (5 – 8)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
## Marshalled Linked List

<table>
<thead>
<tr>
<th></th>
<th>value_t</th>
<th>array length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>value_t</td>
<td>next: 1</td>
</tr>
<tr>
<td>1</td>
<td>value_t</td>
<td>next: 2</td>
</tr>
<tr>
<td>2</td>
<td>value_t</td>
<td>next: 3</td>
</tr>
<tr>
<td>3</td>
<td>value_t</td>
<td>next: 4</td>
</tr>
<tr>
<td>4</td>
<td>value_t</td>
<td>next: 5</td>
</tr>
<tr>
<td>5</td>
<td>value_t</td>
<td>next: 6</td>
</tr>
<tr>
<td>6</td>
<td>value_t</td>
<td>next: -1</td>
</tr>
</tbody>
</table>

**Array Length:** 6
Reliability Explained …

• Assume, for now, that RPC is layered on top of (unreliable) UDP
• Exactly once semantics
  – each RPC request is handled exactly once on the server
RPC Exchanges

1. Client sends Request 1 to Server.
2. Server sends Response 1 to Client.
3. Client sends Request 2 to Server.
4. Server sends Response 2 to Client.
Idempotent Procedures

At-least-once semantics
Non-Idempotent Procedures

Client

Transfer $1000 from Alice’s Account to Bob’s

Server

Done

Client

Do it again and again and again!

Server

Client

At-most-once semantics

Server
Maintaining History

Transfer $1000 from Alice’s Account to Bob’s Account

Client → Server

Done

Do it again!

Client → Server

Done (replay)

Client → Server

At-most-once semantics
No History

Client → Server
Transfer $1000 from Alice’s Account to Bob’s

Client → Server
Transfer $1000 from Alice’s Account to Bob’s
Done

CRASH!!

Client → Server
Do it again!

Client → Server
Sorry ...

At-most-once semantics
Making ONC RPC Reliable

• Each request uniquely identified by *transmission ID* (XID)
  – transmission and retransmission share same XID

• Server maintains *duplicate request cache* (DRC)
  – holds XIDs of non-idempotent requests and copies of their complete responses
  – kept in cache for a few minutes
Algorithm

Receive request

Duplicate?

Yes

Repeat original response

No

Perform request
Did It Work?

• No
Problem ...

Client write(data) xid=1 → nfsd1

Client write(data) xid=1 (retransmission) → nfsd2

Client done

Client write(newdata) xid=2 → nfsd2

Client done

Client done

Client done
Solution

Receive request

Duplicate?

Original in progress?

Original successful?

Idempotent?

Discard

Repeat original reply

Perform request

Yes

No

Yes

No

Yes

No

Yes

No
Quiz 1

An idempotent request from the client is received by the server, executed, and the response sent back. But the response doesn’t make it to the client.

a) The client retransmits its request and the original response is sent back (again) to the client

b) The client retransmits its request, but the original response is not sent back and thus, from the client’s point of view, the server has crashed

c) The client, after multiple retransmissions, eventually gets a response
Did It Work?

• Sort of …
• Works fine in well behaved networks
• Doesn’t work with “Byzantine” routers
  – programmed by your worst (and brightest) enemy
  – probably doesn’t occur in local environment
  – good approximation of behavior on overloaded Internet
• Doesn’t work if server crashes at inopportune moment (and comes back up)
Enter TCP

RPC

TCP

IP
Quiz 2

UDP is easy to implement efficiently. Early implementations of TCP were not terribly efficient, therefore early implementations of RPC were layered on UDP, on the theory that UDP usually provided reliable delivery.

a) TCP is reliable, therefore layering RPC on top of TCP makes RPC reliable

b) The notions of at-most-once and at-least-once semantics are still relevant, even if RPC is layered on top of TCP

c) There are additional reliability concerns when layering RPC on top of TCP
What’s Wrong?

• The problem is the duplicate request cache (DRC)
  – it’s necessary
  – but when may cached entries be removed?
Session-Oriented RPC

Client 1

2-slot DRC for client 1

Client 2

3-slot DRC for client 2

<table>
<thead>
<tr>
<th>XID</th>
<th>seq. #</th>
<th>response</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>XID</th>
<th>seq. #</th>
<th>response</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>XID</th>
<th>seq. #</th>
<th>response</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>XID</th>
<th>seq. #</th>
<th>response</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>XID</th>
<th>seq. #</th>
<th>response</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DCE RPC

• Designed by Apollo and Digital in the 1980s
  – both companies later absorbed by HP
• Does everything ONC RPC can do, and more
• Basis for Microsoft RPC
typedef struct {
    double   comp1;
    int      comp2;
    long long comp3;
    char     *annotation;
} value_t;

char add(int key, value_t value);
char remove(int key, value_t value);
int query(int key, int number, value_t values[]);
An Interface Specification

interface db {
    typedef struct {
        double comp1;
        long comp2;
        hyper comp3;
        [string, ptr]
        ISO_LATIN_1
        *annotation;
    } value_t;

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

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

    [idempotent] long query(
        [in] long key,
        [in] long number,
        [out, size_is(number)]
        value_t values[
    );

}
An Interface Specification (notes continued)

interface db {
    typedef struct {
        double comp1;
        long comp2;
        hyper comp3;
        [string, ptr]
        ISO_LATIN_1 *
    } value_t;

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

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

    [idempotent] long query(
        [in] long key,
        [in] long number,
        [out, size_is(number)]
        value_t values[]
    );
}
Representing an Array

<table>
<thead>
<tr>
<th>Length</th>
<th>Item 1</th>
<th>Item 2</th>
<th>…</th>
<th>Item n</th>
</tr>
</thead>
</table>

Representing Pointers

Sender

\[ P \rightarrow *P \]

Marshalled

\[ *P \]

Receiver

\[ P \rightarrow *P \]

On stack

On callee’s stack
Complications
Marshalling Unrestricted Pointers

A: B: C: D: E:

0 (A):
2
4

2 (B):
-1
6

4 (C):
6
8

6 (D):

8 (E):
Referring to Server State

Client

pointer

Server
Maintaining Client State on Servers

interface trees {
    typedef [context_handle] void *tree_t;

    void create (  
        [in] long value,  
        [out] tree_t pine    
    );

    void insert (  
        [in] long value,  
        [in, out] tree_t pine  
    );
}

Unique Identifiers

[uuid (333A2276-0000-0000-0D00-008090C00000),
  version (3.1)]

interface vectorops {

  small inner (  
    [in] long size,
    [in, size_is (size)] long A[],
    [in, size_is (size)] long B[],
    [out] long *result
  );

}
### UUIDs

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>low bits of time</td>
</tr>
<tr>
<td>48</td>
<td>reserved</td>
</tr>
<tr>
<td>16</td>
<td>mid bits of time</td>
</tr>
<tr>
<td>4</td>
<td>version</td>
</tr>
<tr>
<td>12</td>
<td>high bits of time</td>
</tr>
<tr>
<td>2</td>
<td>clock seq high</td>
</tr>
<tr>
<td>6</td>
<td>clock seq Low</td>
</tr>
<tr>
<td>8</td>
<td>node address</td>
</tr>
</tbody>
</table>