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);
}
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

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);
}
Marshalled Arguments

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

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
<th>Next</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 …

• Exactly once semantics
  – each RPC request is handled exactly once on the server
RPC Exchanges

Request 1

Client ➔ Server

Response 1

Client ➔ Server

Request 2

Client ➔ Server

Response 2

Client ➔ Server
Uncertainty
Idempotent Procedures

Write File A Block 0

Done

(Ahem …) Write File A Block 0

At-least-once semantics
Non-Idempotent Procedures

Client → Transfer $1000 from DJ’s Account to Indy’s → Server

At-most-once semantics
Maintaining History

Client

Transfer $1000 from DJ’s Account to Indy’s

Done

Server

At-most-once semantics

Client

Do it again!

Server

Done (replay)

Client
No History

Client → Server

Transfer $1000 from DJ’s Account to Indy’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

No

Repeat original response

Perform request
Did It Work?

• No
Problem …

Client  \[\text{write(data) xid=1}\] → nfsd1

Client  \[\text{write(data) xid=1 (retransmission)}\] → nfsd2 → blocks

Client  \[\text{done}\]  → updates file

Client  \[\text{write(newdata) xid=2}\] → nfsd2

Client  \[\text{done}\]  → updates file

Client  \[\text{done}\]  → updates file

Client  \[\text{done}\]  → updates file
Solution

Receive request

Duplicate?

Yes

Original in progress?

Yes

Discard

No

Original successful?

Yes

Perform request

No

Idempotent?

No

Repeat original reply

Yes

No
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

- Any improvement?
  - no ...
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
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
Same Example ...

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
(notices continued)

```c
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[]
    );
}
```
Representing an Array

| Length | Item 1 | Item 2 | ... | Item n |
Representing Pointers

sender

marshalled

receiver

P

*P

on stack

on callee’s stack

sender

marshalled

receiver

P

*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

```c
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

\[\text{uuid} \ (333A2276-0000-0000-0D00-008090C000000), \]
\[\text{version} \ (3.1)\]

\textbf{interface} vectorops \{

\textbf{small} inner \{

\textbf{in} \ \textbf{long} \ size, \n\textbf{in, size_is} (size) \ \textbf{long} \ A[ ], \n\textbf{in, size_is} (size) \ \textbf{long} \ B[ ], \n\textbf{out} \ \textbf{long} *\textbf{result}

\};

\}

UUIDs

32
low bits of time

16
mid bits of time

4
version

12
high bits of time

2 6 8
clock seq high clock seq Low

48
node address

reserved