Implementing Remote Procedure Calls

By Andrew D. Birrell and Bruce Jay Nelson

Adapted from an earlier presentation by Abdussalam Alawini
Contents

1. Introduction
   • Context
   • Remote Procedure Call vs Local Procedure Call
   • Design Choices

2. RPC Implementation
   • Basic Structure
   • Binding
   • Network Protocol
   • Exception Handling
   • Use of Processes

3. Evaluation
1. Introduction
CS533 context

- We’ve studied the:
  - Procedural (thread-based) model
  - Message passing (event-based) model
  - Duality of the two models
  - Wrappers to translate calls between models

- Here we study an implementation of a procedural model on a distributed, message passing substrate
  - Conceptually, threads pass from one machine to the other and back again
  - The programmer is unaware of the underlying message-based substrate
What is a “Remote” Procedure Call?

\[
Z = F(x, y)
\]

\[
F(x, y)
\]

Result

Server

Compute \( F(x, y) \)

Communication Network

Client
Design Alternatives

- Explicit message passing
  - i.e. force the programmer to change models
- Shared virtual address space
  - Access remote data ... but how to move computation?
- Remote fork
  - Parallelism in addition to distribution
- Remote procedure call
  - threads libraries, such as Pthreads, make a similar choice
Goals of implementing RPC

- Simplicity
  - Make RPCs as similar to procedure calls as possible
  - Simplify distrusted computation

- Efficiency
  - Anticipated heavy use justifies special support

- Security
  - Secure end-to-end communications with RPC
2. Implementing RPC
RPC Structure

- **Caller Machine**
  - User (application code module)
  - User-Stub
  - Caller instance of RPCRuntime (RPC communications package)

- **Callee Machine**
  - Server (server code module)
  - Server-stub
  - Callee instance of RPCRuntime
RPC Components Interactions

Caller Machine

User Application

User-stub

Call Packet

Caller RPCRuntime

Network

Do work and return results

Unpack & make local call

Pass them to server-stub

Callee Machine

Libraries (Server code)

Server-stub

Results Packet

Callee RPCRuntime

Normal Local Proc Call

Import Interface arg spec proc arguments

Pack target

Call Packet transmit packet "Reliably"

Pass them to server-stub

Unpack & make local call

Do work and return results
Who Does What?

Caller Machine

User Application

User-stub

Caller RPCRuntime

Mesa Interface Modules

Programmer

Interface

Lupine (Auto Generation)

Part of Cedar

Network

Callee Machine

Libraries (Server code)

Server-stub

Callee RPCRuntime
Binding Process

- How does a client of the binding mechanism specify what to be bound to?
  - Naming

- How does the caller specify the callee machine address and the specific procedure to invoke?
  - Locating
Naming (Interface Name)

- **Type**: Which interface the caller expects the callee to implement.
  - Service Name (e.g. Mail-server)
- **Instance**: Which particular implementer of an abstract interface is desired.
  - Machine Address (e.g. Specific mail-server address)
Design Alternatives for Location

1- Include network address in user application
   - Binding is too early!

2- Broadcasting Protocol
   - Too much interference with innocent bystanders
   - Not convenient for binding machines not in the same local network
   - Not scalable
Locating an appropriate exporter

<table>
<thead>
<tr>
<th>Type (Group)</th>
<th>Member-list</th>
<th>Instance (Individual)</th>
<th>Connect-site</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileAccess</td>
<td>{Ebbets, Luther, Facc}</td>
<td>Ebbets</td>
<td>3#22#</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Luther</td>
<td>3#276#</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Facc</td>
<td>3#43#</td>
</tr>
</tbody>
</table>
Steps of Exporting an Interface
(Making it available to a client)

- Export
- Make sure that type and instance is correct
- RPCRuntime
- Record exported interface in a table
- Set the address of the current machine in the connect-site
- Adds the instance to the member-list of the type of the instance
- Do update
- AddMember
- Return

Grapevine

Callee Machine

RPCRuntime

Server-stub

Server

Export [FA, Ebbets,...]

Interface exported (Can be accessed remotely)

Record exported interface in a table

SetConnect

AddMember

Table [tableIndex, InterfaceName, DispatcherProduct, UniqueID]
Steps of Binding An Importer to An Exporter Interface
(Getting ready for remote calls)
Caller Machine

User

RPCRuntime

Server

Now let see how this will work

Callee Machine

User

RPCRuntime

Server

Lookup the table of current exports
Send the corresponding binding info to the local procedure using provided arguments

Export Interface is ready for your next remote procedure calls

1- Retrieve interface binding information
2- Prepare call packet

Alright, I will transmit it to the machine with address “connect-site”

Retrieves all the exported Interface

X = openFile(y)

openFile -> 2

transmit

Check UID In table

2 -> openFile

X = openFile(y)

1- Lookup current exports
2- verify UID
3- send call packet to dispatcher

After unpacking, Dispatcher uses the info to map to the right procedure

Call the local procedure using provided arguments

Lookup the Grapevine DB and get the NT address of exporting Interface

Here’s the type and instance we need

Here’s the NT address

Normal call to procedure openFile

OK, now I’ve got UID, TableIndex, and exporter NT Address. I will record them

Exporter Interface is ready for your next remote procedure calls

1- Retrieve interface binding information
2- Prepare call packet

Alright, I will transmit it to the machine with address “connect-site”

Retrieves all the exported Interface

X = openFile(y)

openFile -> 2

transmit

Check UID In table

2 -> openFile

X = openFile(y)
Binding Mechanism Advantages

- Stateless
  - Importing an interface has no effect on the state of the exporting machine
- The use of UID means that bindings are implicitly broken if the exporter crashes and restarts.
- Restricting the set of users who can update Grapevine DB.
  - To avoid security problems.
- Several choices of binding time
  - Importer specifies the type only
Packet-Level Transport Protocol (Why?)

- Substantial performance gains
- Minimize the elapsed real-time between initializing a call and getting results.
  - Unacceptable to have large amount of state info.
  - Unacceptable to have expensive handshaking.
- Guarantee procedure in the server has been invoked precisely once
  - Not just “at most once” or “at least once”
Simple Calls Example

Result packet is sufficient Ack to the caller.

[Machine Id, Process] Seq#

Caller Machine

User

RPC+Stub

Call

Send Call Packet

Wait Ack or Result

Return

Callee Machine

RPC+Stub

Call

CallID DP info

Args

RPC compares Seq# in CID to the one it has in a table that maintains seq# of the last call invoked by each calling activity.

Monotonic for each activity.

No repeats (calls might eliminated as dup.)

CallID DP info

Result

CallID results

Invoke proc

Send results

Server

Do call

Return

Result packet is sufficient Ack to the caller.

Desired Procedure Info:

UId, Table Index, Procedure entry point.

Procedure arguments which is the input or output that will be used by RP.
Is this “Stack Ripping”? 

- In manual stack management the necessary data is taken off the stack and put on the heap.
- In RPC it’s taken off the stack and put in a message:
  - Sent to the other side
  - Put on a different stack ...
- Similar to taking the state off the stack and putting it in a continuation:
  - Executed by a separate event handler
Complicated Call Example

**Caller Machine**

- **User**: Call
  - Send CallPkt Wait for Ack
  - Build next pkt
  - Transmit it Wait for Ack
  - Retransmit Wait for Ack
  - Wait for result
  - Return

- **RPC+Stub**: Call
  - CallID, Pkt=0, PlsAck, ….
  - CallID, Pkt=1, dontAck, ….
  - Data
    - CallID, Pkt=1, PlsAck, ….
  - Ack
    - CallID, Pkt=1,
  - Result
    - CallID, Pkt=2, dontAck, ….
  - Result
    - CallID, Pkt=2, PlsAck, ….
  - Ack
    - CallID, Pkt=2,

**Callee Machine**

- **RPC+Stub**: Start arg record
  - Acknowledge Wait for next pkt
  - Retransmit Wait for ack
  - Invoke call
  - Acknowledge
  - Send result Wait for ack
  - Retransmit Wait for ack
  - Acknowledge
  - Send result
  - Wait for Ack

**Server**

- **idle**: Do call
  - Return
  - Send result Wait for Ack
  - Retransmit Wait for Ack
  - Acknowledge
  - Send result
  - Wait for Ack

No subsequent call arrived, so ask the caller to do an explicit Ack that it has received the results.

Arguments going to be in 2 packets

Send the rest of the data in Data Pkt

In case of lost packet, long call duration or long gaps between calls retransmit the last pkt and ask for Ack

Satisfying Ack: process waits for results

Caller sends probe pkts periodically (can detect communication failure)

Now all arguments are ready, so the dispatching process starts

Send results And wait for another call "works as an Ack"

Return results to caller process (user code)

No subsequent call arrived, so ask the caller to do an explicit Ack that it has received the results
4. Data Integrity and Security
Exception Handling (Remote Process Exception)

- Two level of exception
  - Communication Failure Exception (Explained with complicated call example), considered to be the primary difference between procedure call and RPC
  - Remote Process Exception
If there’s a catch phrase, exception will be handled and results will be sent back to the callee machine.
Use of Processes

If there’s no corresponding process waiting for this packet then it will be dispatched to one of the free processes (Current, New, or Dup) that have the same CallID.

Current, New, or Dup

Process 7 made a remote procedure call to process 3. This part is the process identifiers, which is part of the activity info within the CallID.

If there’s no corresponding process waiting for this packet then it will be dispatched to one of the free processes (Current, New, or Dup) that have the same CallID.

 RPC now transmit the packet back to the caller (result, exception, or Ack).

Caller Machine

Callee Machine

SrcProcess DestProcess

Network

3 Processes Waiting For RPC packet
4. Evaluation
Optimizations

- Use of thread pool (idle processes) in caller and callee machines to reduce process creation costs.
- The use of process source and destination allow processes to get the packets they’re waiting for directly from the interrupt handler.
- Use of subsequent packet for implicit acknowledgments of previous packets.
- Avoid the cost of establishing and terminating connections by the implementation of packet-level protocol.
Performance evaluation

They’ve measured the elapsed time between two machines for 12,000 calls for each of the following procedures:

- 0-10 arguments/results.
- 1-100 word array.
- Caller Resume and unwind exception handling.
## Performance results

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Minimum</th>
<th>Median</th>
<th>Transmission</th>
<th>Local-only</th>
</tr>
</thead>
<tbody>
<tr>
<td>no args/results</td>
<td>1059</td>
<td>1097</td>
<td>131</td>
<td>9</td>
</tr>
<tr>
<td>1 arg/result</td>
<td>1070</td>
<td>1105</td>
<td>142</td>
<td>10</td>
</tr>
<tr>
<td>2 args/results</td>
<td>1077</td>
<td>1127</td>
<td>152</td>
<td>11</td>
</tr>
<tr>
<td>4 args/results</td>
<td>1115</td>
<td>1171</td>
<td>174</td>
<td>12</td>
</tr>
<tr>
<td>10 args/results</td>
<td>1222</td>
<td>1278</td>
<td>239</td>
<td>17</td>
</tr>
<tr>
<td>1 word array</td>
<td>1069</td>
<td>1111</td>
<td>131</td>
<td>10</td>
</tr>
<tr>
<td>4 word array</td>
<td>1106</td>
<td>1153</td>
<td>174</td>
<td>13</td>
</tr>
<tr>
<td>10 word array</td>
<td>1214</td>
<td>1250</td>
<td>239</td>
<td>16</td>
</tr>
<tr>
<td>40 word array</td>
<td>1643</td>
<td>1695</td>
<td>566</td>
<td>51</td>
</tr>
<tr>
<td>100 word array</td>
<td>2915</td>
<td>2926</td>
<td>1219</td>
<td>98</td>
</tr>
<tr>
<td>resume except’n</td>
<td>2555</td>
<td>2637</td>
<td>284</td>
<td>134</td>
</tr>
<tr>
<td>unwind except’n</td>
<td>3374</td>
<td>3467</td>
<td>284</td>
<td>196</td>
</tr>
</tbody>
</table>
Performance summary

- For transferring large amounts of data in one direction products other than RPC have advantage.
  - Transfer fewer packets in the other direction
- They haven’t measure the performance of exporting or importing interfaces.
- Used by:
  - Alpine [File server supports multi-machine transactions]
  - Control communication for an Ethernet-based for telephone and audio project
  - Networking games.
Conclusion

- RPC is used to communicate between processes in different address spaces
  - May be on separate machines or on the same machine
- Communication is implicit
  - RPC makes distributed programming easier
  - Allows thread-based programming model on distributed platforms
- Reliably transfers data and control