

Distributed Programming and Remote Procedure Calls (RPC)

George Porter

CSE 124

February 17, 2015

Announcements

- Project 2 checkpoint – use submission site or email if having difficulties.
- Project 1 recap
- SQL vs. Hadoop
- Graphing tools
 - Log scale
- Change in schedule:
 - Thursday will be an in-class tutorial on RPC with Facebook's Thrift framework

Project 1 recap

```
# NumSamples = 92; Min = 7.00; Max = 110.00
```

```
# Mean = 82.967391; Variance = 716.292415; SD = 26.763640; Median 93.000000
```

each ■ represents a count of 1

```
7.0000 - 10.0000 [ - 2]: ■■
```

10.0000 - 20.0000 [4]: ■■■■

20.0000 - 30.0000 [- 2]: ■■

30.0000 - 40.0000 [3]: ■■■

40.0000 - 50.0000 [1]: ■

50.0000 - 60.0000 [- 0]:

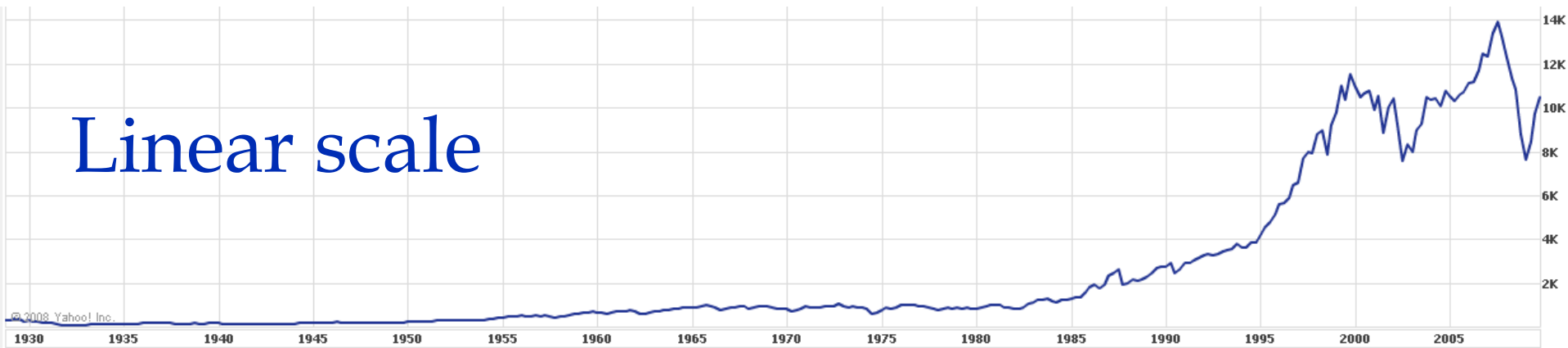
60.0000 - 70.0000 [7]: ■■■■■■

```
70.0000 - 80.0000 [ 16]: ██████████
```

80.0000 - 90.0000 [6] : ■■■■■■

[illegible]

Linear scale



Logarithmic scale



Part 1: RPC Concepts

Remote Procedure Call (RPC)

- Distributed programming is challenging
 - Need common primitives/abstraction to hide complexity
 - E.g., file system abstraction to hide block layout, process abstraction for scheduling/fault isolation
- In early 1980's, researchers at PARC noticed most distributed programming took form of *remote procedure call*
- Popular variant: Remote Method Invocation (RMI)
 - Obtain handle to remote object, invoke method on object
 - Transparency goal: remote object appear as if local object

RPC Example

Local computing

```
X = 3 * 10;  
print(X)  
> 30
```

Remote/RPC computing

```
server = connectToServer(S);
```

```
Try:
```

```
    X = server.mult(3,10);
```

```
    print(X)
```

```
Except e:
```

```
    print “Error!”
```

```
> 30
```

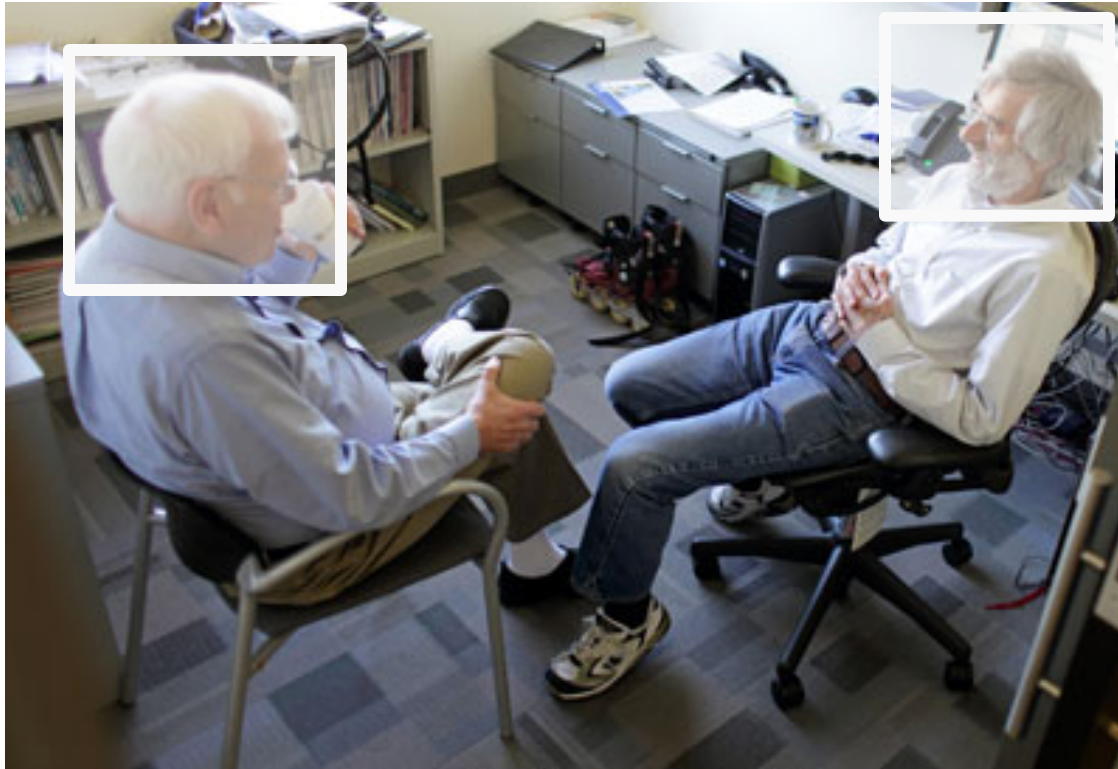
```
or
```

```
> Error!
```

Why RPC?

- Offload computation from devices to servers
 - Phones, raspberry pi, sensors, Nest thermostat, ...
- Hide the implementation of functionality
 - Proprietary algorithms
- Functions that just can't run locally

Why RPC?



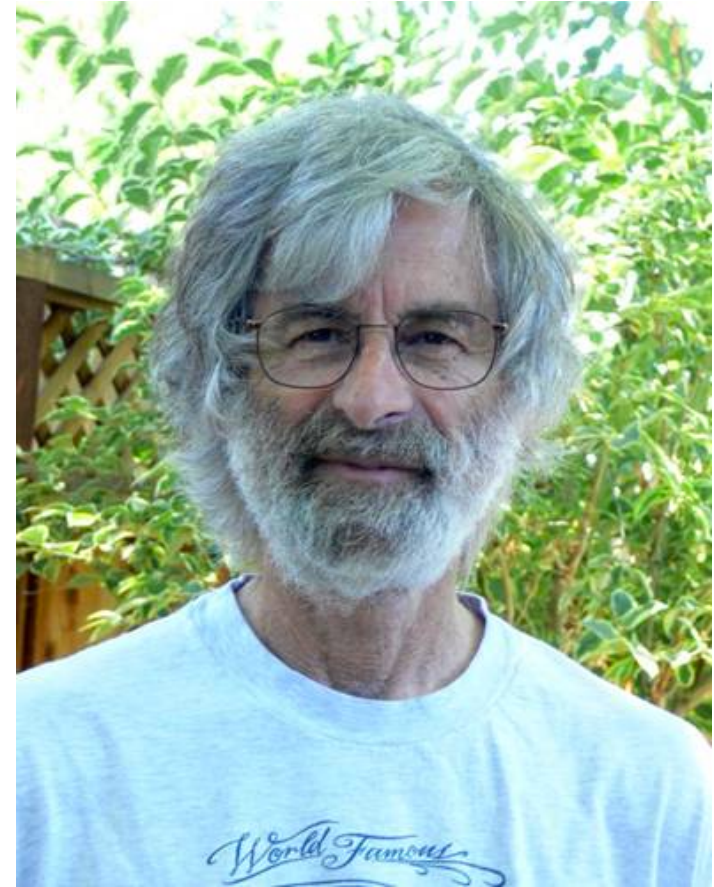
- Functions that just can't run locally
 - `tags = Facebook.MatchFacesInPhoto(photo)`
 - Print tags
 - > ["Chuck Thacker", "Leslie Lamport"]

What makes RPC hard?

- Network vs. computer backplane
 - Message loss, message reorder, ...
- Heterogeneity
 - Client and Server might have different:
 - OS versions
 - Languages
 - Endian-ness
 - Hardware architectures
 - ...

Leslie Lamport

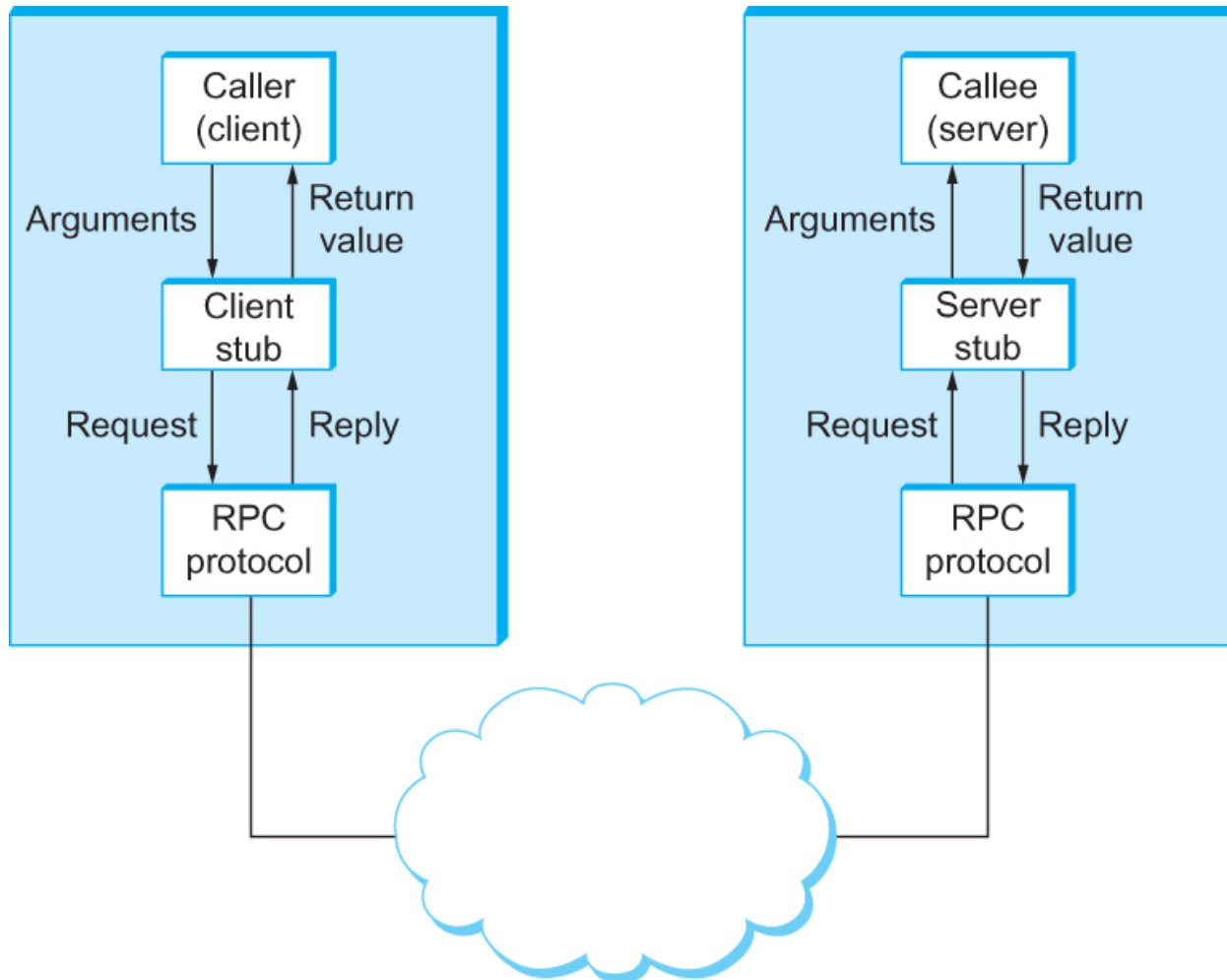
“A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable”



RPC Components

- End-to-end RPC protocol
 - Defines messages, message exchange behavior, ...
- Programming language support
 - Turn “local” functions/methods into RPC
 - Package up arguments to the method/function, unpackage on the server, ...
 - Called a “stub compiler”
 - Process of packaging and unpackaging arguments is called “Marshalling” and “Unmarshalling”

High-level overview



Overview

- RPC Protocol itself
- Stub compiler / marshalling
- RPC Frameworks

Remote Procedure Call Issues

- Underlying protocol
 - UDP vs. TCP
 - Advantages/Disadvantages?
- Semantics
 - What happens on network/host failure?
 - No *fate sharing* (compared to local machine case)
- Transparency
 - Hide network communication/failure from programmer
 - With language support, can make remote procedure call “look just like” local procedure call

Identifying remote function/methods

- Namespace vs flat

- Issues w.r.t. flat: how to assign?

- Matching requests with responses

- Can't just rely on TCP

Why??

- What if the client or server fail during execution?

- Main idea:

- Message ID
 - Client blocked until response returned w/ Message ID

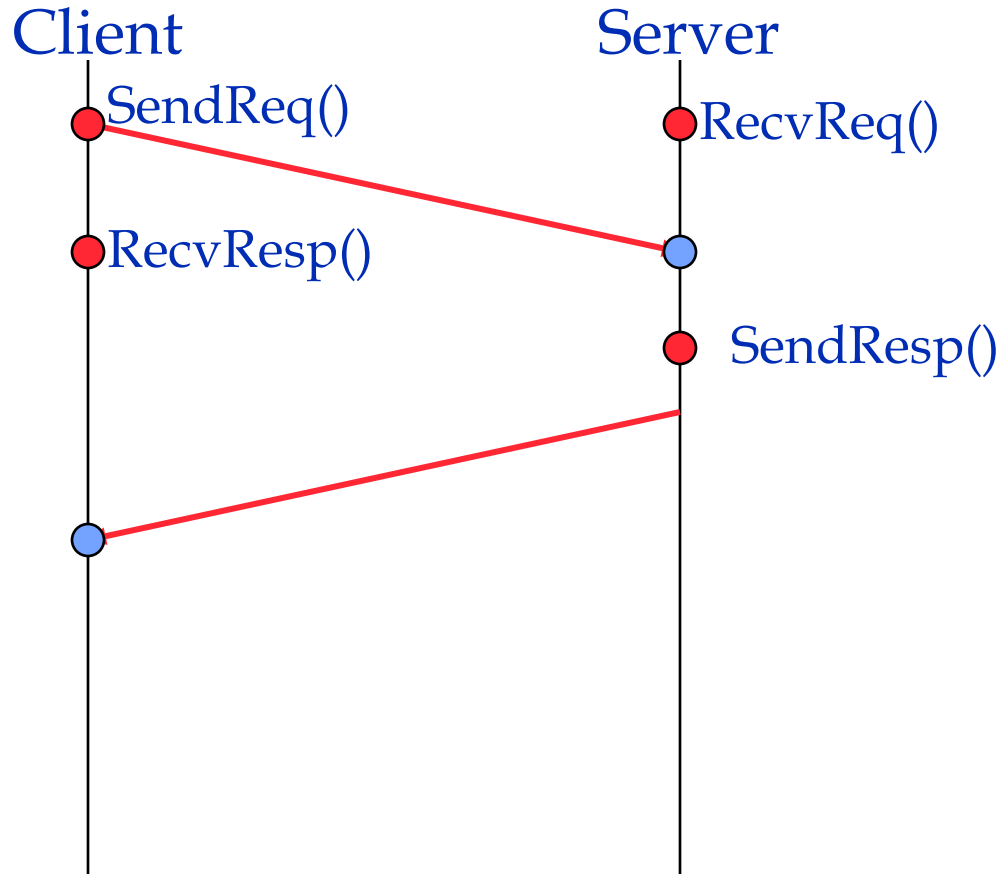
When things go wrong

- Let's assume client and server use TCP
- Client issues request(0)
- Client fails, reboots
- Client issues unrelated request(0)
- Server gets first request(0), executes it, send response
- Server get the second request, thinks it is a duplicate, and so ACKs it but doesn't execute
- Client never gets a response to either request.

Boot ID + Message ID

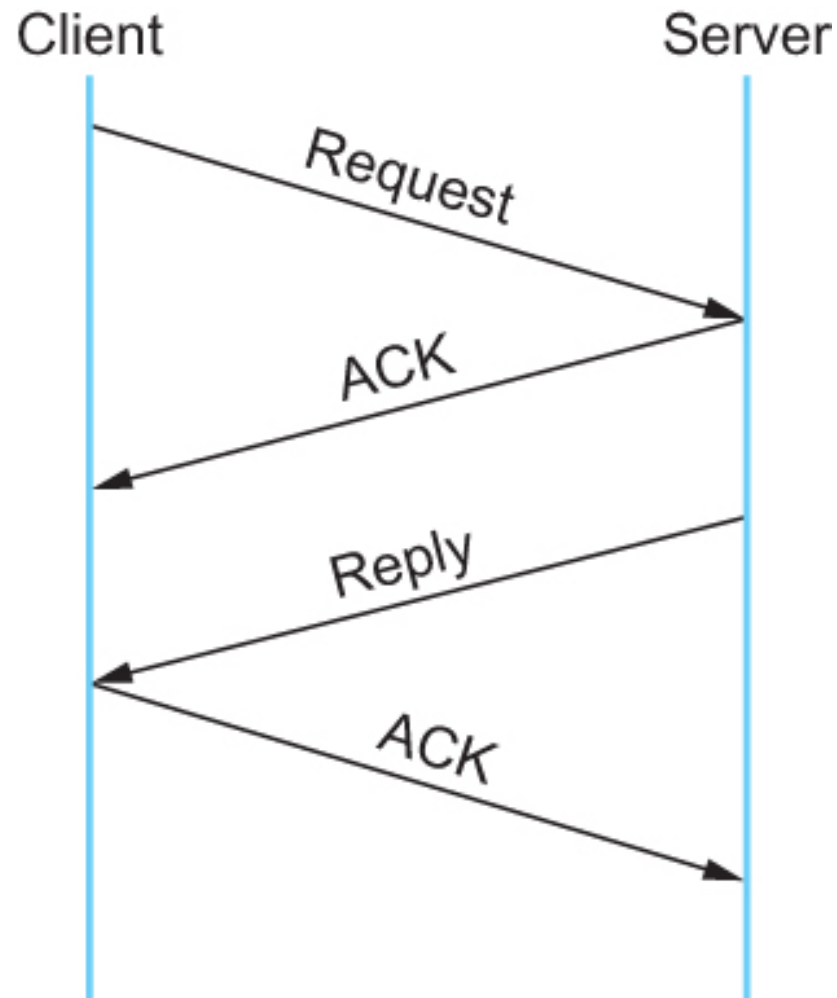
- Idea is to keep non-volatile state that is updated every time the machine boots
- Incremented during the boot-up process
- Each request/response is uniquely identified by a (bootID,MessageID) pair

Client/Server Interaction

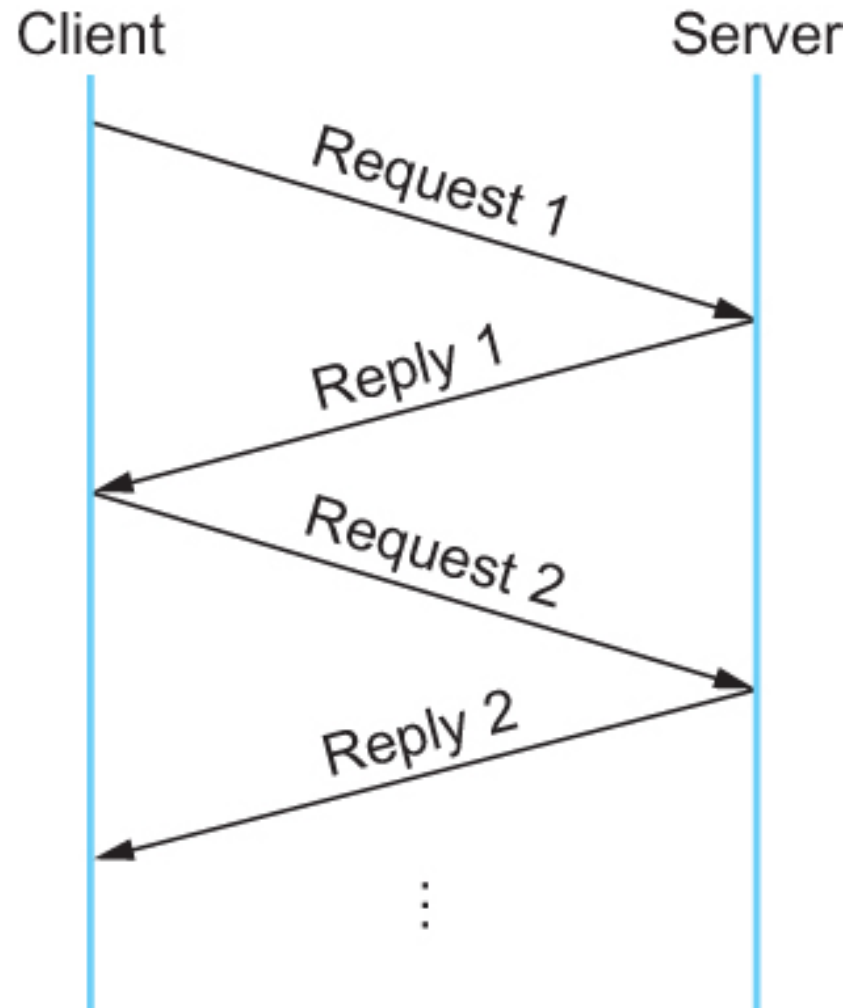


- Client sends request
 - Client waits for response (blocking)
- Server wakes up
 - Computes, sends response
 - Response wakes up client
- Server structure?
 - Process, thread, FIFO

Reliable RPC (explicit acknowledgement)



Reliable RPC (implicit acknowledgement)



What about failures?

- Different options:
 1. Server can keep track of the (BootId,MessageId) and ignore any duplicate requests
 2. Server doesn't keep state—might execute duplicate requests
- What are advantages/disadvantages?

RPC Semantics

Delivery Guarantees			RPC Call Semantics
Retry Request	Duplicate Filtering	Retransmit Response	
No	NA	NA	<i>Maybe</i>
Yes	No	Re-execute Procedure	<i>At-least once</i>
Yes	Yes	Retransmit reply	<i>At-most once</i>

Remote Procedure Call Issues

■ Idempotent operations

- Can you re-execute operations without harmful side effects
- Idempotency means that executing the same procedure does not have visible side effects

■ Timeout value

- How long to wait before re-transmitting request?

Protocol-to-Protocol Interface

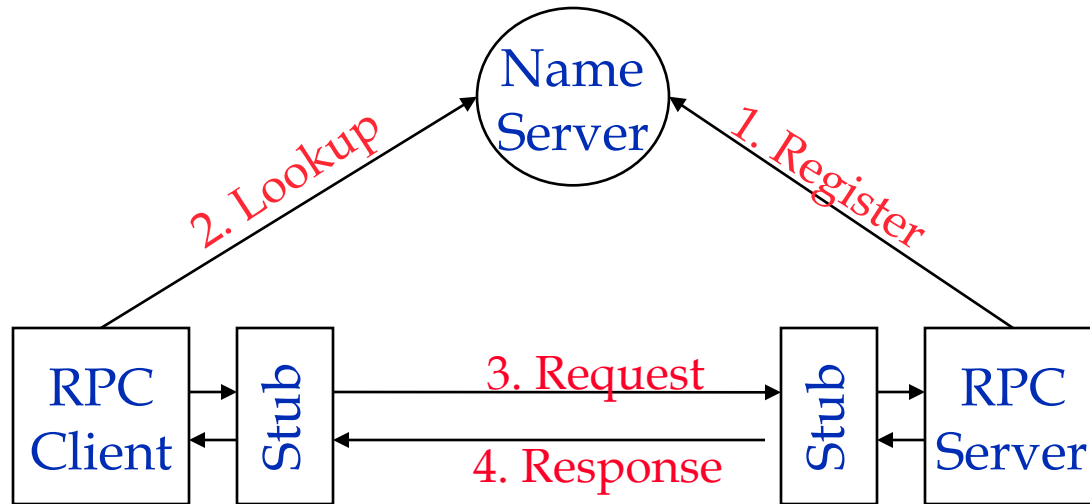
- Send/Receive semantics
 - Synchronous vs. Asynchronous

- Process Model
 - Single process vs. multiple process
 - Avoid context switches

- Buffer Model
 - Avoid data copies

Part 2: RPC Implementations

Request / Response Flow



- Client stub indicates which procedure should run at server
 - Marshals arguments
- Server stub unmarshals arguments
 - Big switch statement to determine local procedure to run

RPC Mechanics

- Client issues request by calling stub procedure
 - Stubs can be automatically generated with compiler support
- Stub procedure marshals arguments, transmits requests, blocks waiting for response
 - RPC layer deals with network issues (e.g., TCP vs. UDP)
- Server stub
 - Unmarshals arguments
 - Determines correct local procedure to invoke, computes
 - Marshals results, transmits to client
 - Can also be automatically generated with language support

RPC Binding

■ Binding

- Servers register the *service* they provide to a name server
- Clients lookup services they need from server
- Bind to server for a particular set of remote procedures
- Operation informs RPC layer which machine to transmit requests to

■ How to locate the RPC name service?

Presentation Formatting

- Marshalling (encoding)
application data into messages
- Unmarshalling (decoding)
messages into application data

- Data types to consider:

- integers
- floats
- strings
- arrays
- structs



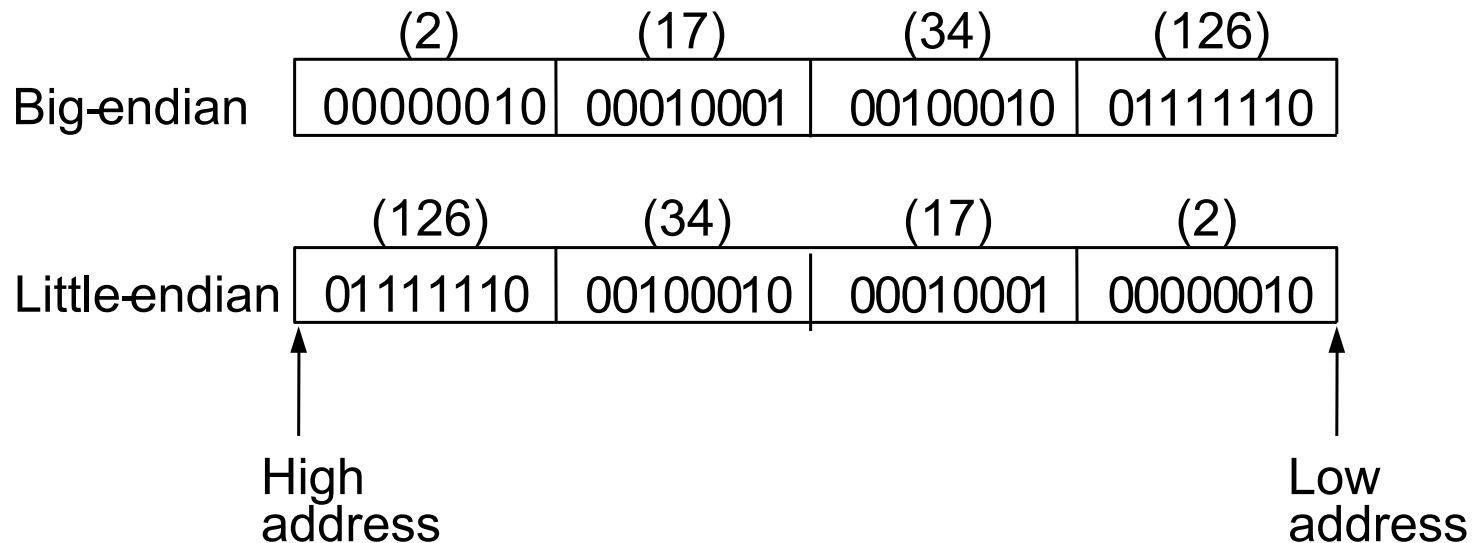
Types of data we do not consider

images
video
multimedia documents

Difficulties

■ Representation of base types

- Floating point: IEEE 754 versus non-standard
- Integer: big-endian versus little-endian (e.g., 34,677,374)

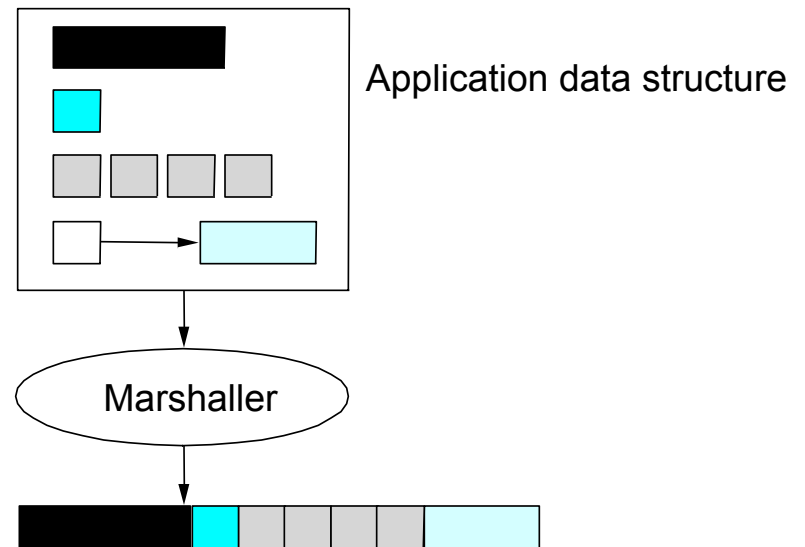


■ Compiler layout of structures

Taxonomy

■ Data types

- Base types (e.g., ints, floats); must convert
- Flat types (e.g., structures, arrays); must pack
- Complex types (e.g., pointers); must linearize



RPC Interface vs. Implementation

■ RPC Interface:

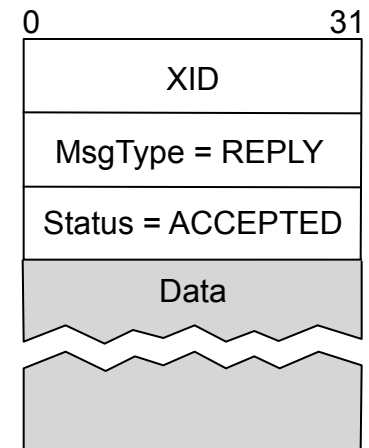
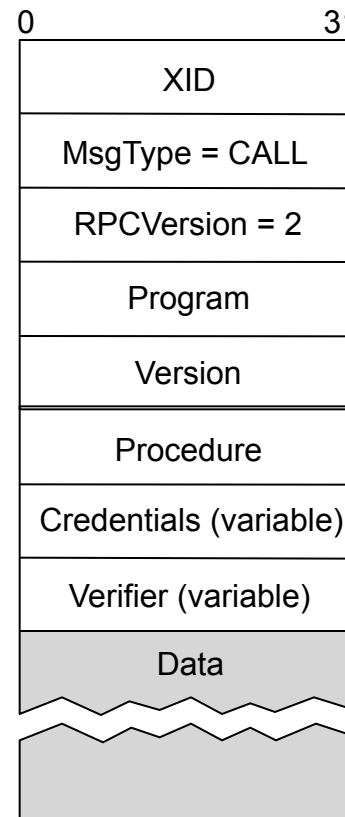
- High-level procedure invocation with arguments, return type
- Asynchronous, Synchronous, 'void', Pipelined...

■ RPC Implementation:

- SunRPC
- Java RMI
- XML RPC
- Apache Thrift
- Google Protocol Buffers*
- Apache Avro

SunRPC

- Originally implemented for popular NFS (network file service)
- XID (transaction id) uniquely identifies request
- Server does not remember last XID it serviced
 - Problem if client retransmits request while reply is in transit
 - Provides *at-least once* semantics



XML-RPC

- XML is a standard for describing structured documents
 - Uses tags to define structure: **<tag> ... </tag>** demarcates an element
 - Tags have no predefined semantics ...
 - ... except when document refers to a specific namespace
 - Elements can have attributes, which are encoded as name-value pairs
 - A well-formed XML document corresponds to an element tree

```
■ <?xml version="1.0"?>
  <methodCall>
    <methodName>SumAndDifference</methodName>
    <params>
      <param><value><i4>40</i4></value></param>
      <param><value><i4>10</i4></value></param>
    </params>
  </methodCall>
```

XML-RPC Wire Format

- Scalar values
 - Represented by a `<value><type> ... </type></value>` block
- Integer
 - `<i4>12</i4>`
- Boolean
 - `<boolean>0</boolean>`
- String
 - `<string>Hello world</string>`
- Double
 - `<double>11.4368</double>`
- Also Base64 (binary), DateTime, etc.

XML-RPC Wire Format (struct)

■ Structures

- Represented as a set of **<member>**s
- Each member contains a **<name>** and a **<value>**

■ **<struct>**

<member>

<name>lowerBound</name>

<value><i4>18</i4></value>

</member>

<member>

<name>upperBound</name>

<value><i4>139</i4></value>

</member>

</struct>

XML-RPC Wire Format (Arrays)

■ Arrays

- A single **<data>** element, which
- contains any number of **<value>** elements

■ **<array>**

<data>

<value><i4>12</i4></value>

<value><string>Egypt</string></value>

<value><boolean>0</boolean></value>

<value><i4>-31</i4></value>

</data>

</array>

XML-RPC Request

- HTTP POST message
 - URI interpreted in an implementation-specific fashion
 - Method name passed to the server program
- **POST /RPC2 HTTP/1.1**
Content-Type: text/xml
User-Agent: XML-RPC.NET
Content-Length: 278
Expect: 100-continue
Connection: Keep-Alive
Host: localhost:8080
<?xml version="1.0"?>
<methodCall>
<methodName>SumAndDifference</methodName>
<params>
<param><value><i4>40</i4></value></param>
<param><value><i4>10</i4></value></param>
</params>
</methodCall>

XML-RPC Response

■ HTTP Response

- Lower-level error returned as an HTTP error code
- Application-level errors returned as a **<fault>** element (next slide)

■ HTTP/1.1 200 OK

Date: Mon, 22 Sep 2003 21:52:34 GMT

Server: Microsoft-IIS/6.0

Content-Type: text/xml

Content-Length: 467

<?xml version="1.0"?>

<methodResponse>

<params><param>

<value><struct>

<member><name>sum</name><value><i4>50</i4></value></member>

<member><name>diff</name><value><i4>30</i4></value></member>

</struct></value>

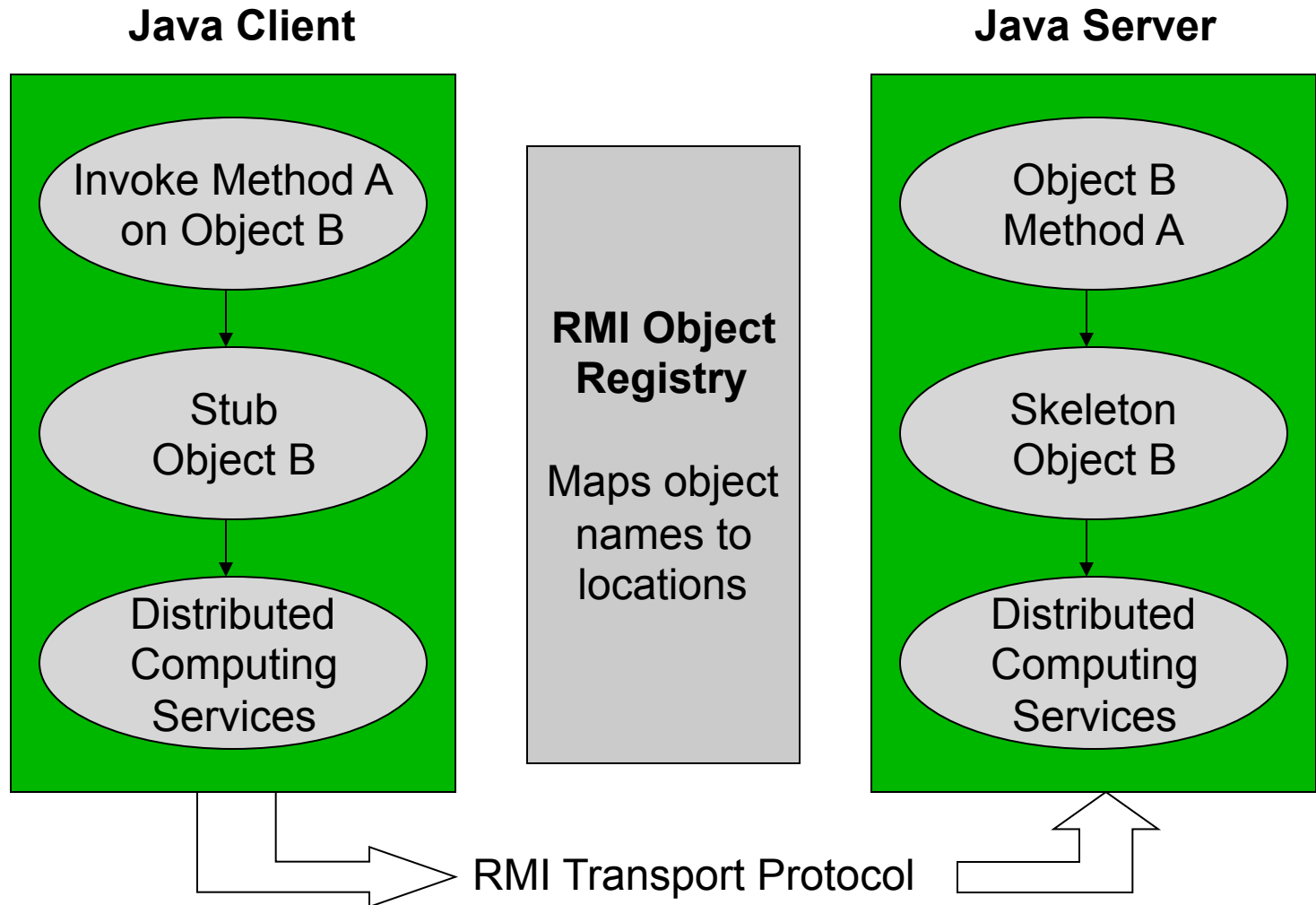
</param></params>

</methodResponse>

XML-RPC Fault Handling

- Another kind of a MethodResponse
- ```
<?xml version="1.0"?>
<methodResponse>
 <fault>
 <value><struct>
 <member>
 <name>faultCode</name>
 <value><i4>500</i4></value>
 </member>
 <member>
 <name>faultString</name>
 <value><string>Arg `a' out of range</string></value>
 </member>
 </struct></value>
 </fault>
</methodResponse>
```

# RMI Architecture



# RMI Example (Server)

## **Hello.java**

---

```
import java.rmi.*;

public interface HelloInterface extends Remote {
 public String say() throws RemoteException;
}
```

## **HelloInterface.java**

---

```
import java.rmi.*; import java.rmi.server.*;
public class Hello extends UnicastRemoteObject implements HelloInterface {
 private String message;

 public Hello (String msg) throws RemoteException { message = msg; }

 public String say() throws RemoteException {
 return message;
 }

 public static void main (String[] args) {
 System.setSecurityManager(new RMISecurityManager());
 try {
 Naming.bind ("Hello", new Hello ("Hello, world!"));
 } catch (Exception e) { System.out.println ("Server failed"); }
 }
}
```

# RMI Example (Client)

## HelloClient.java

---

```
import java.rmi.*;

public class HelloClient {
 public static void main (String[] args) {
 System.setSecurityManager(new RMISecurityManager());
 try {
 HelloInterface hello = (HelloInterface)
 Naming.lookup ("//(hostname)/Hello");
 System.out.println (hello.say());
 } catch (Exception e) {
 System.out.println ("HelloClient exception: " + e);
 }
 }
}
```

- The rmic utility generates stubs and skeletons
- Once started, server will stay active
- RMI daemon (rmid) can be used to create (activate) objects on the fly

# Apache Thrift

