CS 138: Communication II

Today's Lecture

- Sockets
- RPC
  - Overview
  - Challenges
  - Examples

Sockets

Sockets

- TCP and UDP allow sending and receiving of bytes over the network
  - TCP: reliable infinite stream of bytes between two processes
  - UDP: unreliable messages (up to 64KB)
- How do applications access these protocols?

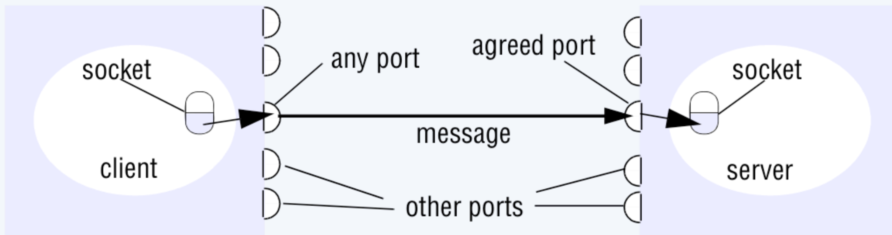
Using TCP/IP

- *Sockets* API.
  - Originally from BSD, widely implemented (*BSD, Linux, Mac OS X, Windows, …)
  - Important do know and do once
  - Higher-level APIs build on them
- After basic setup, much like files
  - Sockets are file descriptors

Types of Sockets

- Datagram sockets: unreliable message delivery
  - With IP, gives you UDP
  - Send atomic messages, which may be reordered or lost
- Stream sockets: bi-directional pipes
  - With IP, gives you TCP
  - Bytes written on one end read on another
- There are other types
  - Eg. Unix domain sockets
    - Endpoints are filenames

Sockets



From Colouris, chapter 4

System calls for using TCP

Client	**Server**
	`socket` – **create socket**
	`bind` – **assign address, port**
	`listen` – **listen for clients**
`socket` – **create socket**	
`bind*` – **assign address (optional)**	
`connect` – **connect to listening socket**	
	`accept` – **accept connection**

Both can `read` **and** `write` **from the connection.**
Both can call `close` **to end the connection.**

Go's Interface is not too different

Server:

```
ln, err := net.Listen("tcp", ":8080")
if err != nil {
    // handle error
}
for {
    conn, err := ln.Accept()
    if err != nil {
        // handle error
    }
    go handleConnection(conn)
}
```

Go's Interface is not too different

Client:

```
conn, err := net.Dial("tcp", "google.com:80")
if err != nil {
    // handle error
}
fmt.Fprintf(conn, "GET / HTTP/1.0\r\n\r\n")
status, err :=
bufio.NewReader(conn).ReadString('\n')
// ...
```

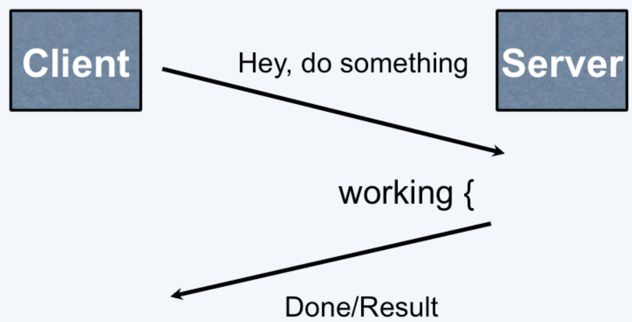
Limitations

- Strictly an interface to the transport layer
  - (or lower)
- Reliability
  - if the receiving machine is temporarily not available, will sent data eventually reach it?
  - how is the sender notified if sent data does not arrive at destination machine?
  - how is the sender notified if sent data does not arrive at destination application?

Writing Distributed Programs

- Concerns
  - transparency
  - portability
  - interoperability
- Solutions
  - RPC
  - RMI

Common communication pattern



Writing it by hand...

- **eg, if you had to write a, say, password cracker**

```
struct foomsg {
  u_int32_t len;
}

send_foo(char *contents) {
   int msglen = sizeof(struct foomsg) + strlen(contents);
   char buf = malloc(msglen);
   struct foomsg *fm = (struct foomsg *)buf;
   fm->len = htonl(strlen(contents));
   memcpy(buf + sizeof(struct foomsg),
          contents,
          strlen(contents));
   write(outsock, buf, msglen);
}
```

Then wait for response, etc.

RPC

- A type of client/server communication
- Attempts to make remote procedure calls look like local ones

```
{ ...
   foo()
}
void foo() {
  invoke_remote_foo()
}
```

Go Example

- **Need some setup in advance of this but…**

```
// Synchronous call
args := &server.Args{7,8}
var reply int
err = client.Call("Arith.Multiply", args,
&reply)
if err != nil {
      log.Fatal("arith error:", err)
}
fmt.Printf("Arith: %d*%d=%d", args.A, args.B,
reply)
```

RPC Goals

- Ease of programming
- Hide complexity
- Automates task of implementing distributed computation
- Familiar model for programmers (just make a function call)

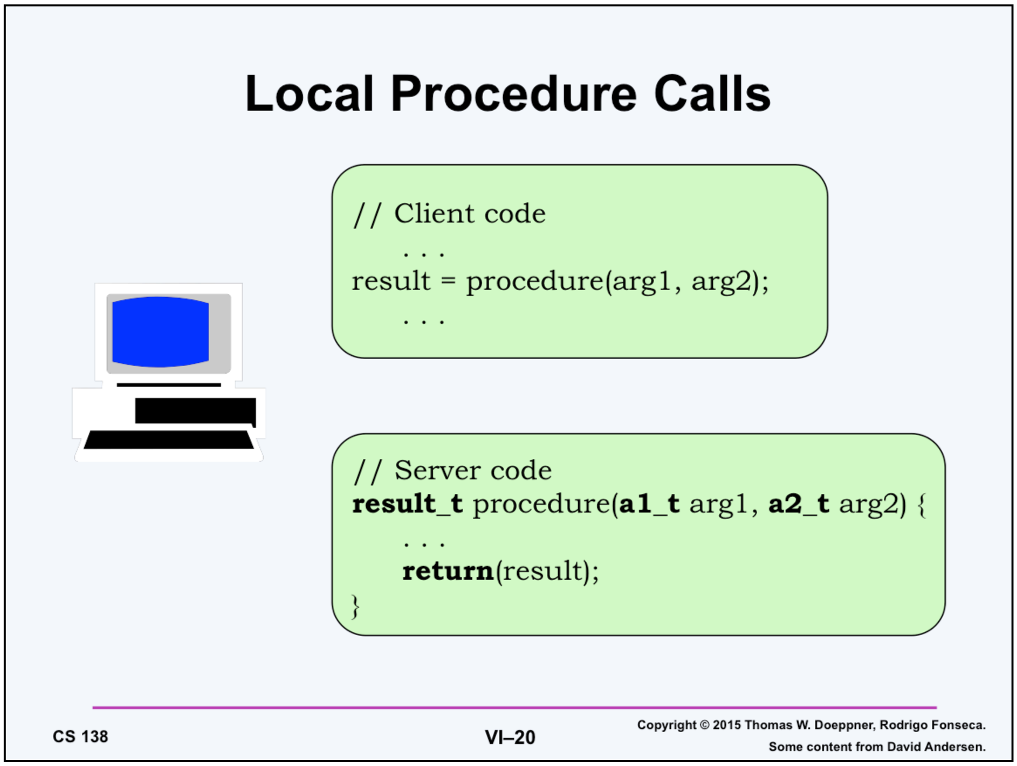
Historical note: Seems obvious in retrospect, but RPC was only invented in the '80s. See Birrell & Nelson, "Implementing Remote Procedure Call" ... or Bruce Nelson, Ph.D. Thesis, Carnegie Mellon University: Remote Procedure Call., 1981 :)

Hiding Complexity

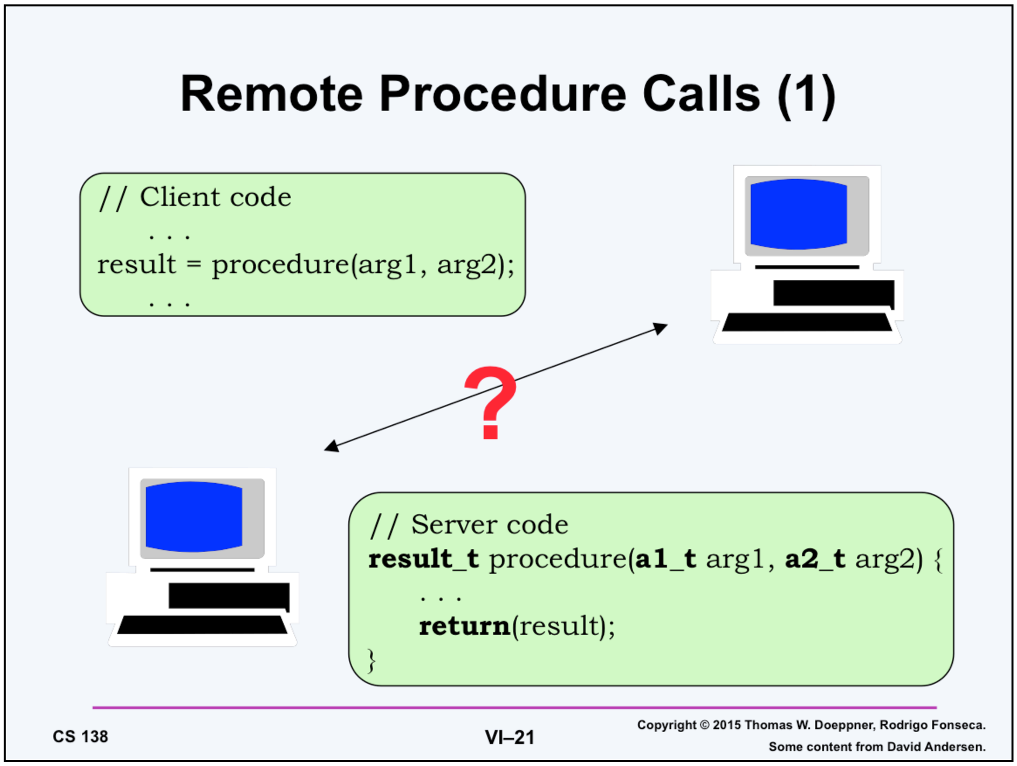
- Makes a call to a remote service look like a local call
  - RPC makes transparent whether server is local or remote
  - RPC allows applications to become distributed transparently
  - RPC makes architecture of remote machine transparent

But it's not always simple

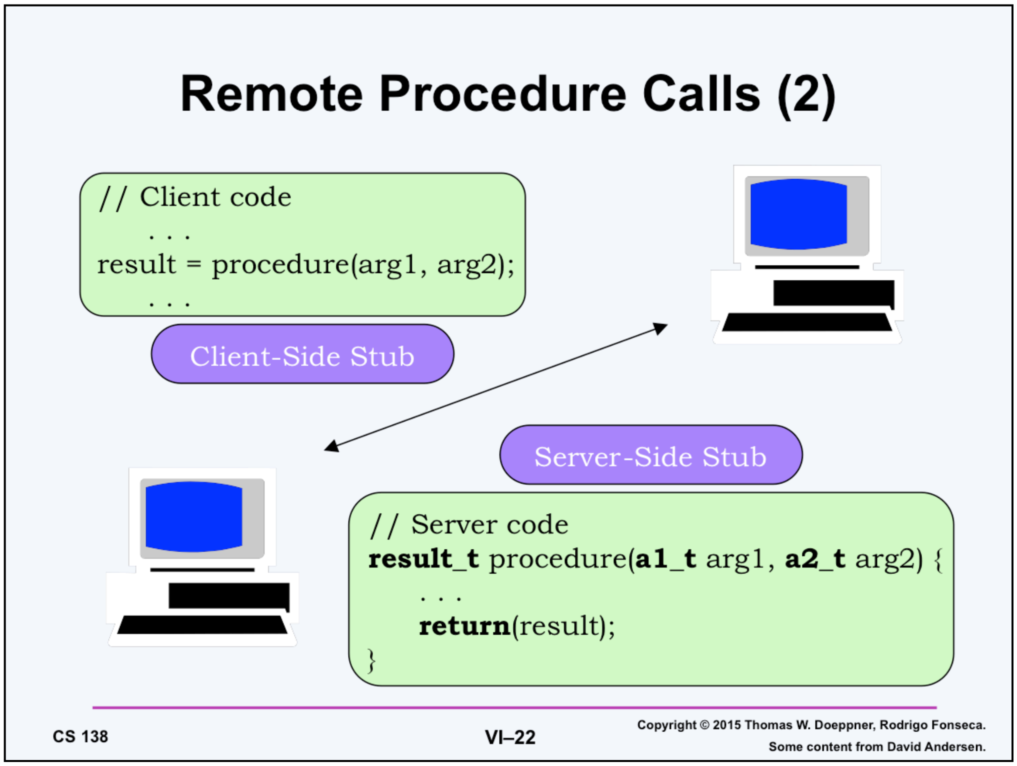
- Calling and called procedures run on different machines, with different address spaces
  - And perhaps different environments .. or operating systems ..
- Must convert to local representation of data
- Machines and network can fail



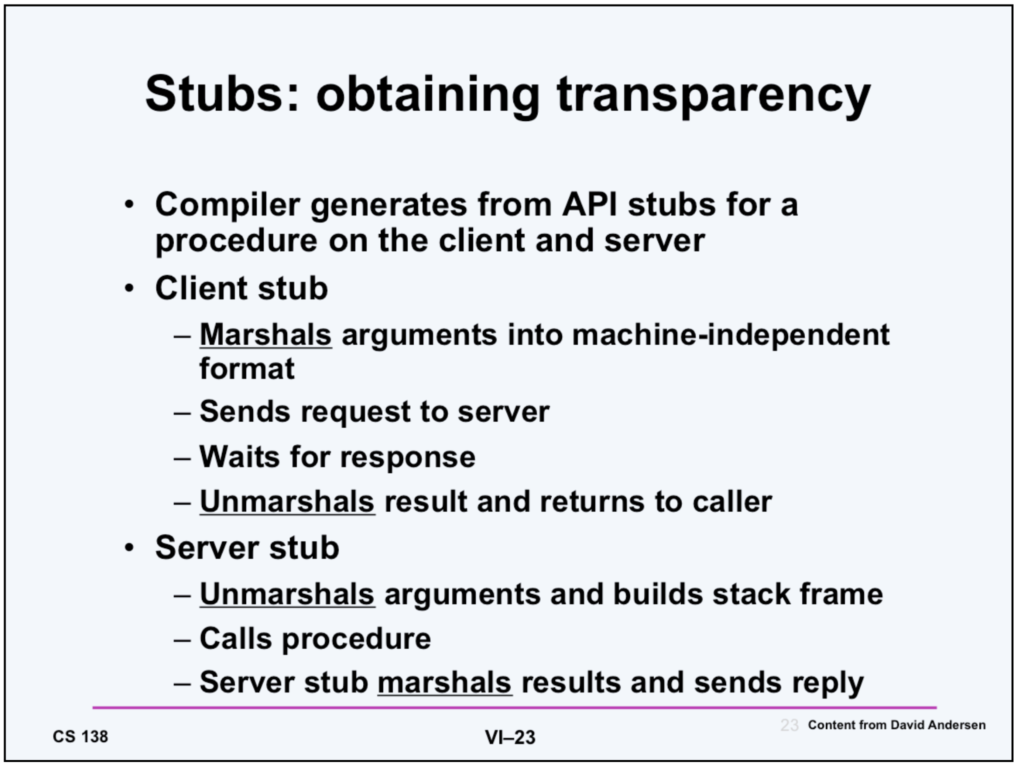
The basic theory of operation of RPC is pretty straightforward. But, to understand *remote* procedure calls, let's first make sure that we understand *local* procedure calls. The client (or caller) supplies some number of arguments and invokes the procedure. The server (or callee) receives the invocation and gets a (shallow) copy of the arguments (other languages, such as C++, provide other argument-passing modes, but copying is all that is provided in C). In the usual implementation, the callee's copy of the arguments have been placed on the runtime stack by the caller—the callee code knows exactly where to find them. When the call completes, a return value may be supplied by the callee to the caller. Some of the arguments might be out arguments—changes to their value are reflected back to the caller. This is handled in C indirectly—the actual argument, passed by copying, is a pointer to some value. The callee follows the pointer and modifies the value.



Now suppose that the client and server are on separate machines. As much as possible, we would like remote procedure calling to look and behave like local procedure calling. Furthermore, we would like to use the same languages and compilers for the remote case as in the local case. But how do we make this work? A remote call is very different from a local call. For example, in the local call, the caller simply puts the arguments on the runtime stack and expects the callee to find them there. In C, the callee returns data through out arguments by following a pointer into the space of the caller. These techniques simply don't work in the remote case.



The solution is to use *stub procedures*: the client places a call to something that has the name of the desired procedure, but is actually a proxy for it, known as the *client-side stub*. This proxy gathers together all of the arguments (actually, just the in and in-out arguments) and packages them into a message that it sends to the server. The server has a corresponding *server-side stub* that receives the invocation message, pulls out the arguments, and calls the actual (remote) procedure. When this procedure returns, returned data is packaged by the server-side stub into another message, which is transmitted back to the client-side stub, which pulls out the data and returns it to the original caller. From the points of view of the caller and callee procedure, the entire process appears to be a local procedure call—they behave no differently for the remote case.



We will see later how these are generated.

“stubs” and IDLs

- RPC stubs do the work of marshaling and unmarshaling data
- *But how do they know how to do it?*
- Two basic alternatives
  - – Write a description of the function signature using an IDL -- interface description language.
    - - Lots of these. Some look like C, some look like XML, ... details don’t matter much.
  - – Use reflection information to do this
    - - Go-rpc, Java RMI

Remote Procedure Calls (1)

- A remote procedure call occurs in the following steps:

1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and calls the local operating system.
3. The client's OS sends the message to the remote OS.
4. The remote OS gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.

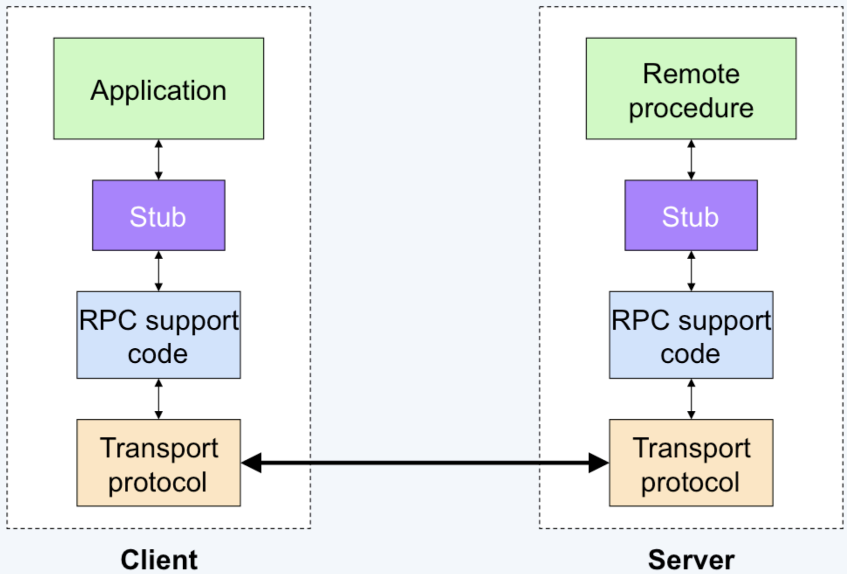
Continued …

Remote Procedure Calls (2)

…

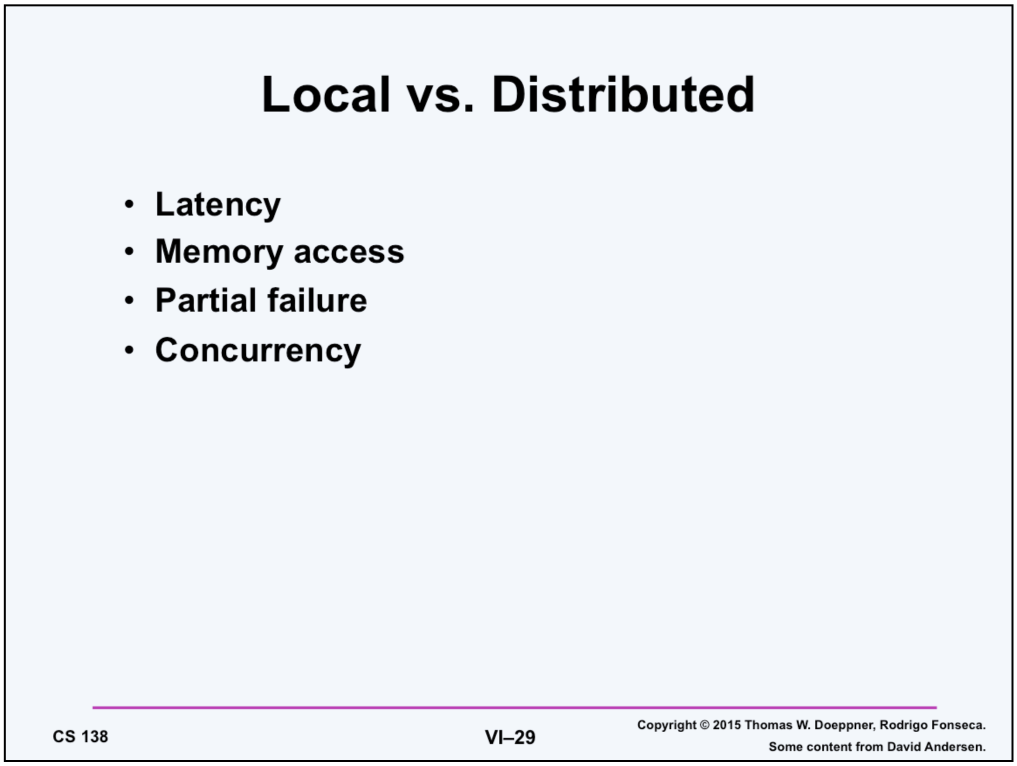
6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and calls its local OS.
8. The server's OS sends the message to the client's OS.
9. The client's OS gives the message to the client stub.
10. The stub unpacks the result and returns to the client.

Block Diagram



Today's Lecture

- Sockets
- RPC
  – Overview
  – Challenges
  – Examples

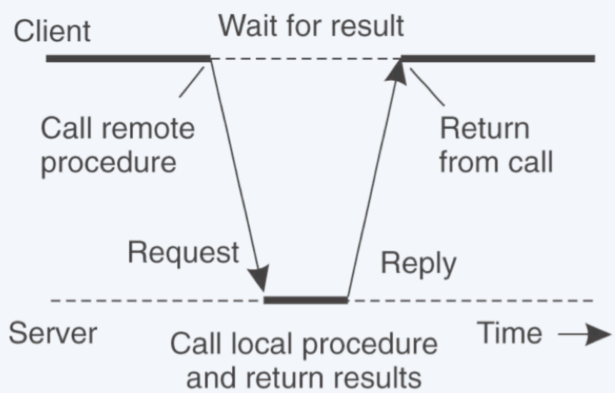


So, what are the differences between local and distributed, and can we create a proper illusion?

Latency

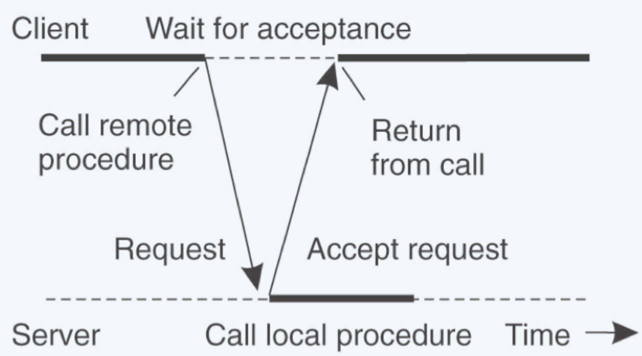
- Remote invocation of objects takes much longer than local invocation
  - can this be ignored at first and dealt with later?

Synchronous RPC



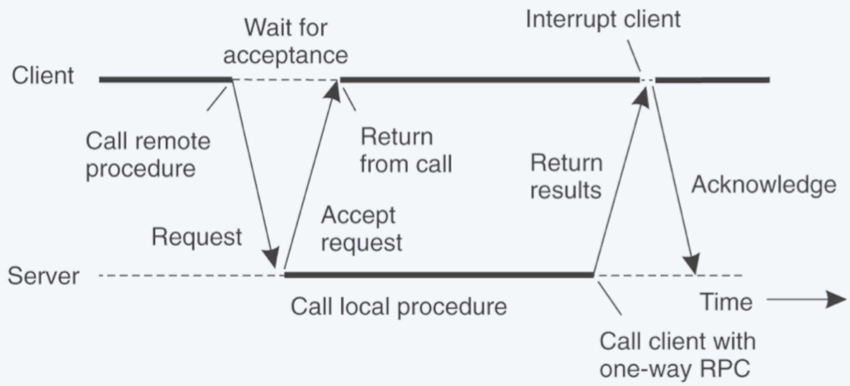
- **The interaction between client and server in a traditional RPC.**

Asynchronous RPC (1)



- **The interaction using asynchronous RPC.**

Asynchronous RPC (2)



- **A client and server interacting through two asynchronous RPCs.**

Concurrency

- Distributed programs have the same concurrency issues as multithreaded programs
  - Do they? In a single address space…
    - all threads are under control of a common OS
    - synchronization is easy
    - fate sharing

RPC failures

- Request from cli → srv lost
- Reply from srv → cli lost
- Server crashes after receiving request
- Client crashes after sending request

Partial failures

- In local computing:
  – if machine fails, application fails
- In distributed computing:
  – if a machine fails, part of application fails
  – one cannot tell the difference between a machine failure and network failure
  – one cannot (in principle) tell the difference between a failure and a really long execution!
- How to make partial failures transparent to client?

Strawman solution

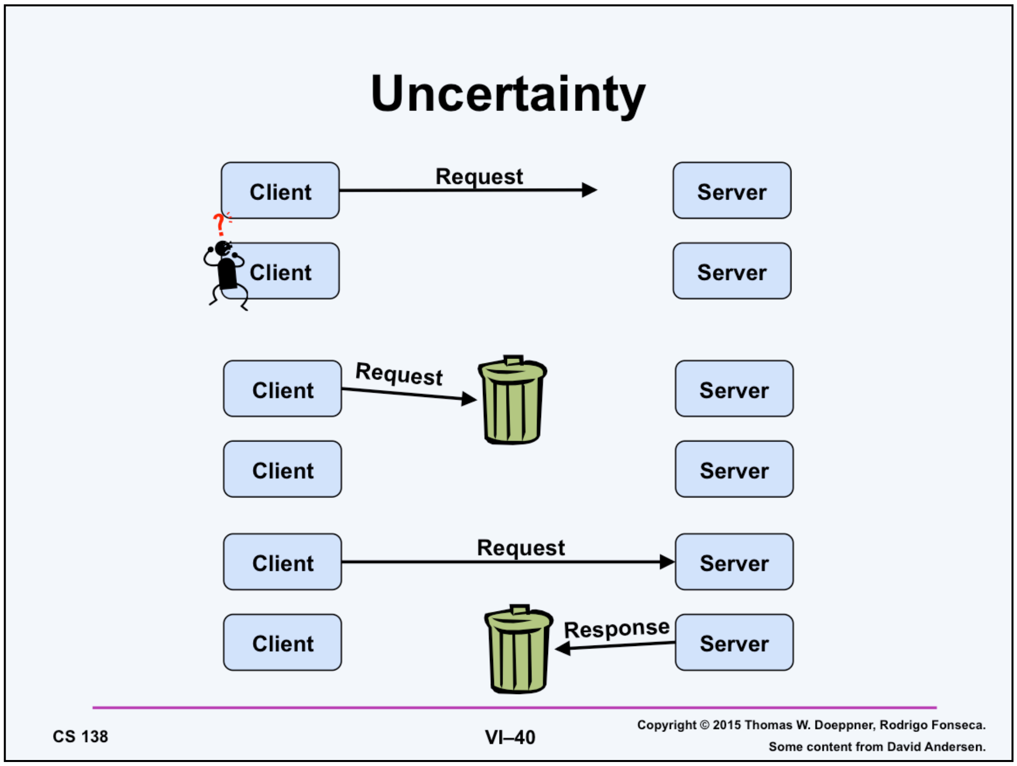
- Make remote behavior identical to local behavior:
  - Every partial failure results in complete failure
    - You abort and reboot the whole system
  - You wait patiently until system is repaired
- Problems with this solution:
  - Many catastrophic failures
  - Clients block for long periods
    - System might not be able to recover

Real solution: break transparency

- Possible semantics for RPC:
  - Exactly-once
    - Impossible in practice
  - At least once:
    - Only for idempotent operations
  - At most once
    - Zero, don’t know, or once
  - Zero or once
    - Transactional semantics

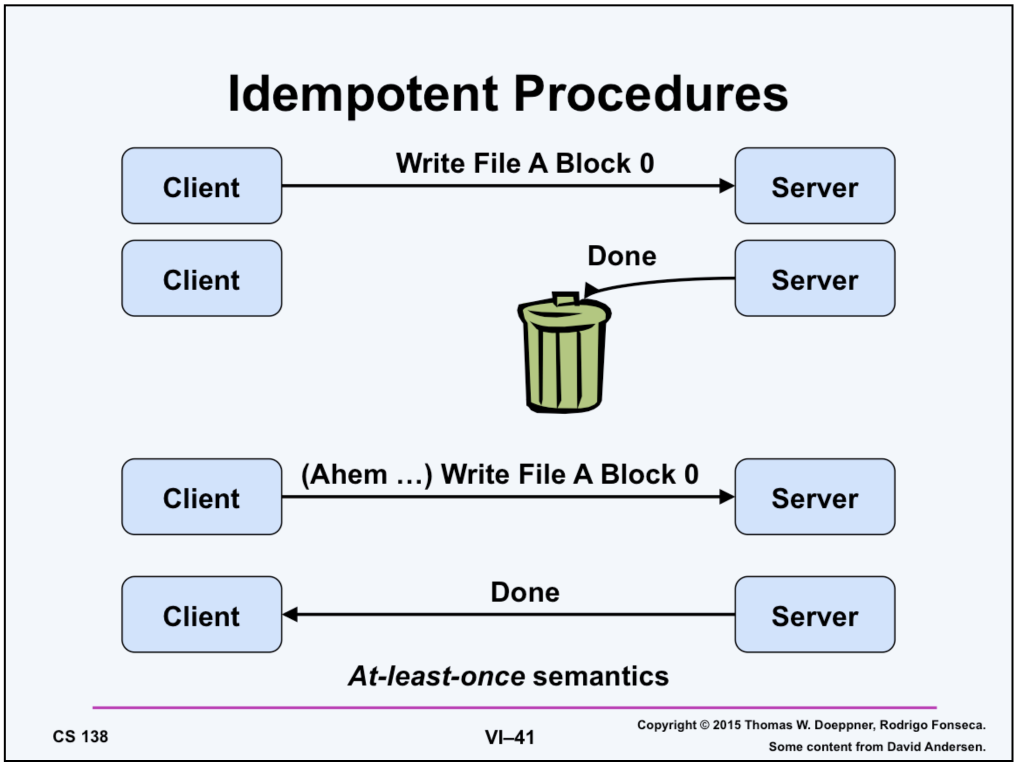
Real solution: break transparency

- <u>At-least-once</u>: Just keep retrying on client side until you get a response.
  - Server just processes requests as normal, doesn't remember anything. Simple!
- <u>At-most-once</u>: Server might get same request twice...
  - Must re-send *previous* reply and not process request (implies: keep cache of handled requests/responses)
  - Must be able to identify requests
  - Strawman: remember *all* RPC IDs handled. -> Ugh! Requires infinite memory.
  - Real: Keep sliding window of valid RPC IDs, have client number them sequentially.

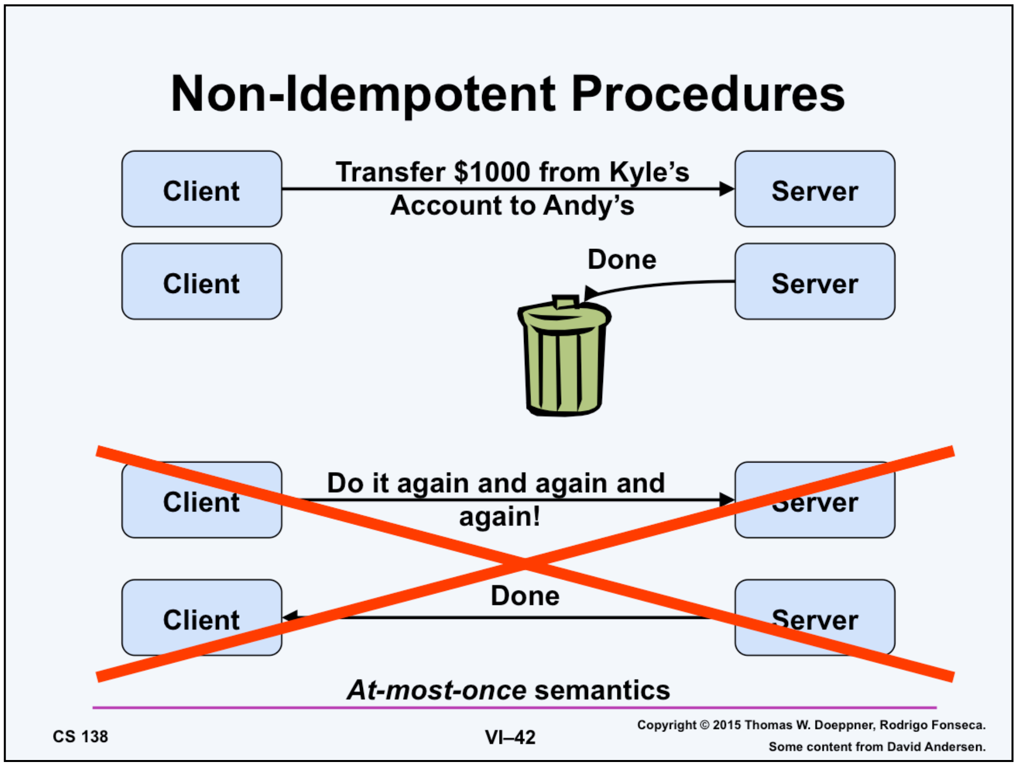


Consider this slide first with the assumption that RPC is layered on UDP. Thus, since the response acts as the acknowledgement, there is uncertainty as to whether the request was handled by the server.

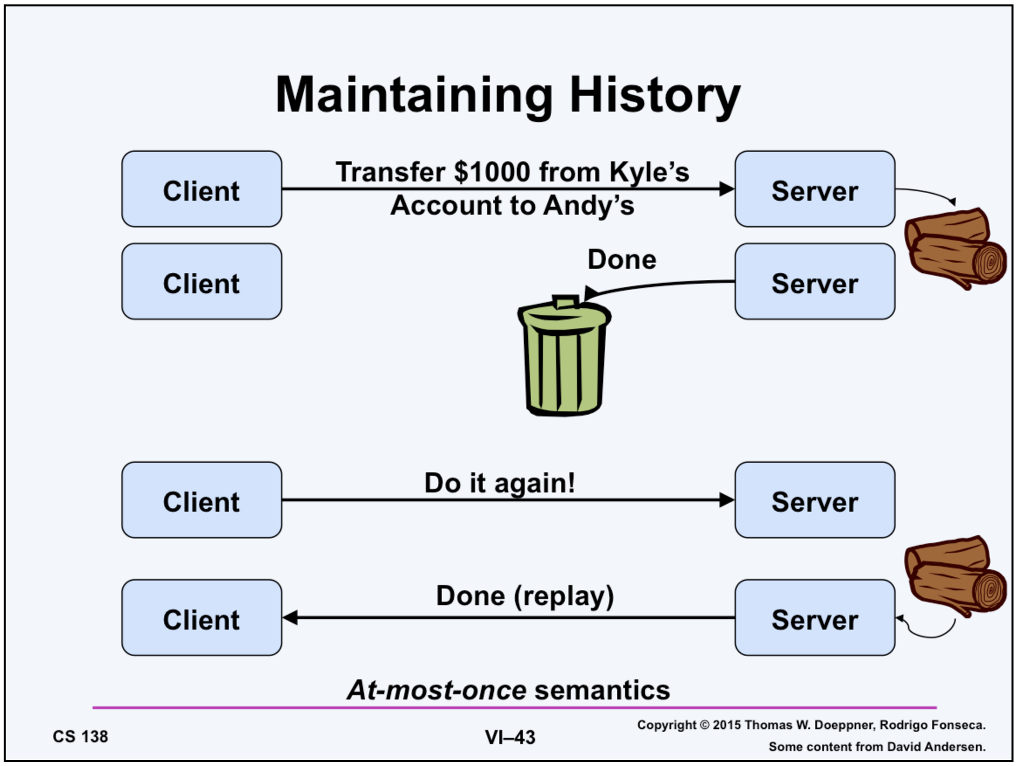
Does this uncertainty go away if RPC is layered on TCP? If you consider the possibility that the TCP connection might be lost, perhaps due to a transient network problem, the answer is clearly no. For example, suppose the TCP connection is lost just after the server receives the request. With no connection, the server cannot send a response, so the client is uncertain about what happened.



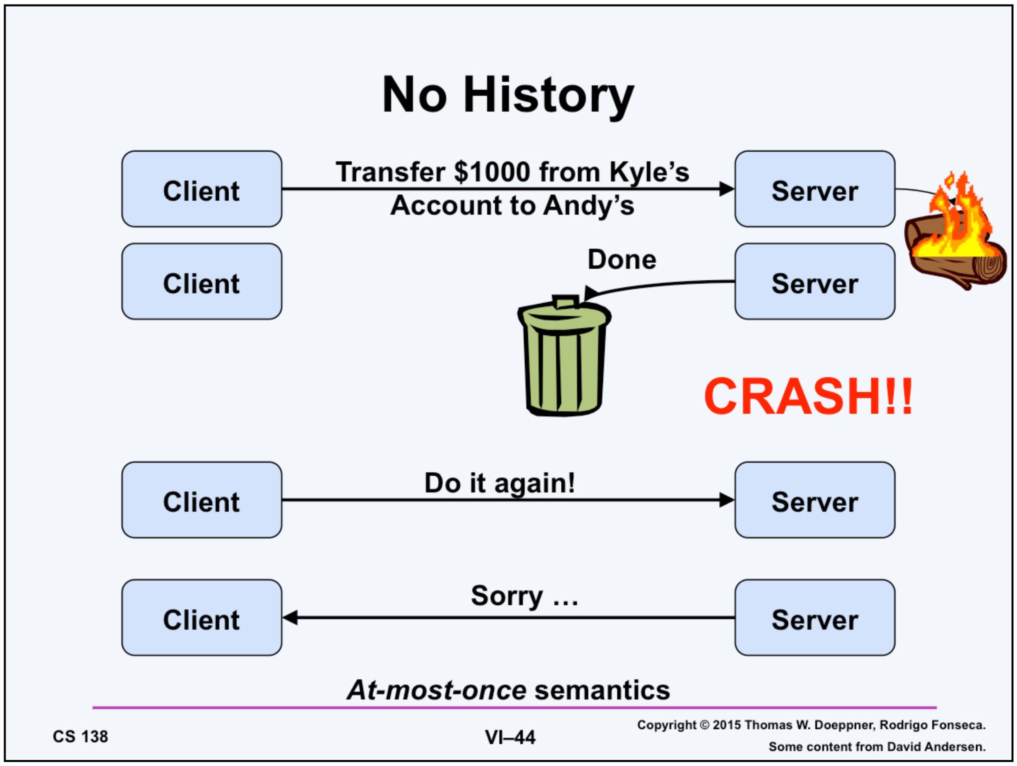
A procedure is *idempotent* if the effect of executing it twice in a row is the same as executing it just once. With such procedures, the client may repeatedly send a request until it finally gets a response. If an RPC protocol depends on such retries, it is said to have *at-least-once semantics* — clients are assured that, after all the retries, the remote procedure is executed at least once.



Not everything is idempotent! If we have non-idempotent procedures, then RPC requests should not be blindly retried, but instead should be sent just once. RPC protocols that do this are said to have *at-most-once semantics*. DCE RPC guarantees at-most-once semantics by default, though a remote procedure may be declared (in its IDL description) to be idempotent, in which case calls are done using at-least-once semantics.



The server might keep track of what operations it has already performed and what the responses were. If it gets a repeat of a previous request, it merely repeats the original response.



If the server crashed and no longer has its history information, it can respond by raising an exception at the client, indicating that it has no knowledge as to whether the operation has taken place. But it guarantees that it hasn't taken place more than once.

Coping

Fault tolerance measures			Invocation Semantics
Retransmit request message	Duplicate filtering	Re-execute procedure or retransmit reply	
No	Not applicable	Not applicable	Maybe
Yes	No	Re-execute procedure	At-least-once
Yes	Yes	Retransmit reply	At-most-once

This table is Figure 5.9 of the text. Note the distinction made between "maybe" and "at-most-once" semantics.

Exactly-Once?

- Sorry - no can do *in general*.
- Imagine that message triggers an external physical thing (say, a robot fires a nerf dart at the professor)
- The robot could crash immediately before or after firing and lose its state. Don't know which one happened. Can, however, make this window very small.

Memory Access

- Pointers work locally
- Can they be made to work remotely?
  - yes … (but it's complicated)
  - but don't use a remote pointer thinking it's just like a local pointer

Implementation Concerns

- As a general library, performance is often a big concern for RPC systems
- Major source of overhead: copies and marshaling/unmarshaling overhead
- Zero-copy tricks:
  - Representation: Send on the wire in native format and indicate that format with a bit/byte beforehand. What does this do? Think about sending uint32 between two little-endian machines (DEC RPC)
  - Scatter-gather writes (writev() and friends)

Summary:
expose remoteness to client

- Expose RPC properties to client, since you cannot hide them

- Application writers have to decide how to deal with partial failures
  - Consider: E-commerce application vs. game

Important Lessons

- Procedure calls
  - Simple way to pass control and data
  - Elegant transparent way to distribute application
  - Not only way…

- Hard to provide true transparency
  - Failures
  - Performance
  - Memory access
  - Etc.

- How to deal with hard problem → give up and let programmer deal with it
  - "Worse is better"

Today's Lecture

- Sockets
- RPC
  - Overview
  - Challenges
  - Examples

Two styles of RPC implementation

- Shallow integration. Must use lots of library calls to set things up:
  – How to format data
  – Registering which functions are available and how they are invoked.

- Deep integration.
  – Data formatting done based on type declarations
  – (Almost) all public methods of object are registered.

- Sun RPC, XMLRPC, GRPC, Thrift
- Go and Java use the latter.

Example: Sun XDR (RFC 4506)

- *External Data Representation* for SunRPC
- Types: most of C types
- No tags (except for array lengths)
  - Code needs to know structure of message
- Usage:
  - Create a program description file (.x)
  - Run rpcgen program
  - Include generated .h files, use stub functions
- Very C/C++ oriented
  - Although encoders/decoders exist for other languages

Example

```
typedef struct {
    int         comp1;
    double      comp2;
    long long   comp3[6];
    char        *annotation;
} value_t;

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

char add(int key, value_t value);
char remove(int key, value_t value);
list_t query(int key);
```


Here's an example of a C declaration for a collection of simple database procedures.

A Specification

```
typedef struct value {
      int                    comp1;
      double                 comp2;
      hyper                  comp3[6];
      string                 annotation<255>;
} value_t;

typedef struct list {
      value_t       element;
      struct list   *next;
} list_t;

program DB {
    version DBVERS {
            bool add(int key, value_t value) = 1;
            bool remove(int key, value_t value) = 2;
            list_t query(int key) = 3;
    } = 1;
} = 0x2000000A;
```

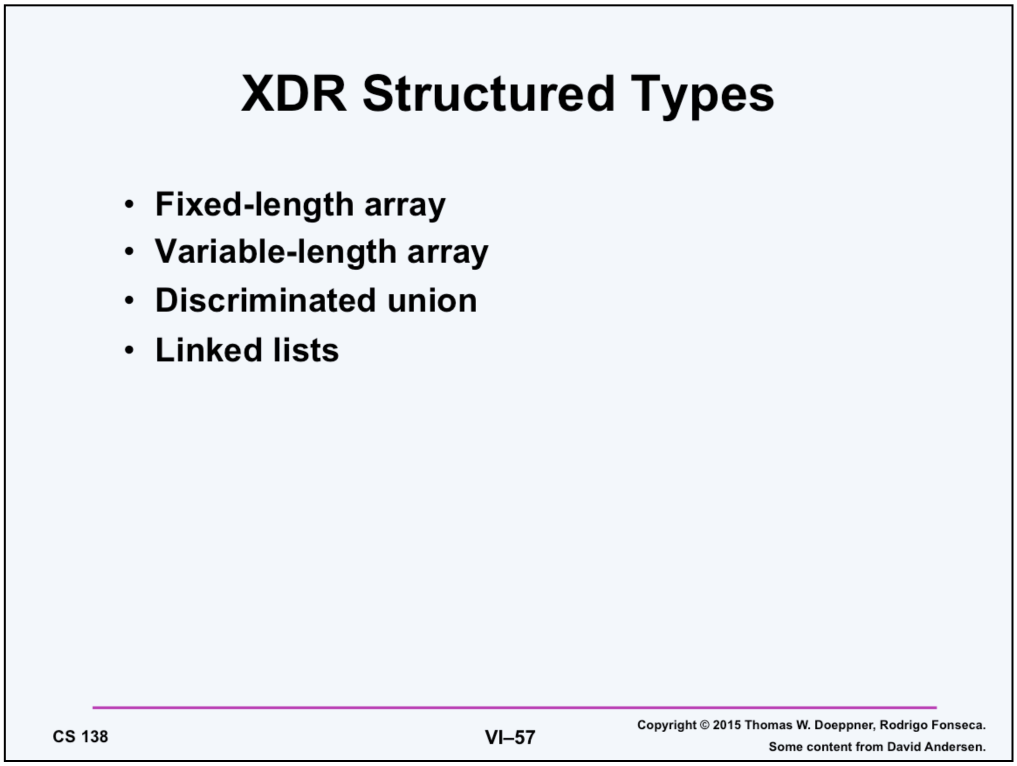
- **Rpcgen generates marshalling/unmarshalling code, stub functions, you fill out the actual code**

Here's a specification of our database interface written in the XDR language. This can now be compiled by *rpcgen* into a pair of stubs, server and client. See http://docs.oracle.com/cd/E19120-01/open.solaris/816-1435/rpcgenpguide-24243/index.html for a description of how to use rpcgen.

XDR Primitive Types

- Integer
- Unsigned integer
- Boolean
- Hyper integer
- Unsigned hyper integer
- Fixed-length opaque data
- Variable-length opaque data
- String

XDR is based on a number of primitive types, whose representation is fixed. We won't go into exactly what these representations are, see RFC 1832 (http://www.faqs.org/rfcs/rfc1832.html) for details.



One builds more complicated data types from the primitives by using the XDR's structured type constructors. We mention the most important ones on the slide; we see them in use in the next several slides.

Generated Header File

```
struct value {
    int comp1;
    double comp2;
    int64_t comp3[6];
    char *annotation;
};
typedef struct value value;

typedef value value_t;

struct list {
    value_t element;
    struct list *next;
};
typedef struct list list;
```

```
typedef list list_t;

struct add_1_argument {
    int key;
    value_t value;
};
typedef struct add_1_argument
        add_1_argument;

struct remove_1_argument {
    int key;
    value_t value;
};
typedef struct remove_1_argument
        remove_1_argument;
```


After running *rpcgen* on our XDR description, we get our stubs, code for marshalling and unmarshalling, and a common header file, shown here.

Placing Calls

```
result_1 = add_1(add_1_key, add_1_value, clnt);
if (result_1 == (bool_t *) NULL) {
        clnt_perror (clnt, "call failed");
}
result_2 = remove_1(remove_1_key, remove_1_value, clnt);
if (result_2 == (bool_t *) NULL) {
        clnt_perror (clnt, "call failed");
}
result_3 = query_1(query_1_key, clnt);
if (result_3 == (list_t *) NULL) {
        clnt_perror (clnt, "call failed");
}
clnt_destroy (clnt);
}
```


Here the client places calls to each of the procedures in our remote program.

DCE RPC

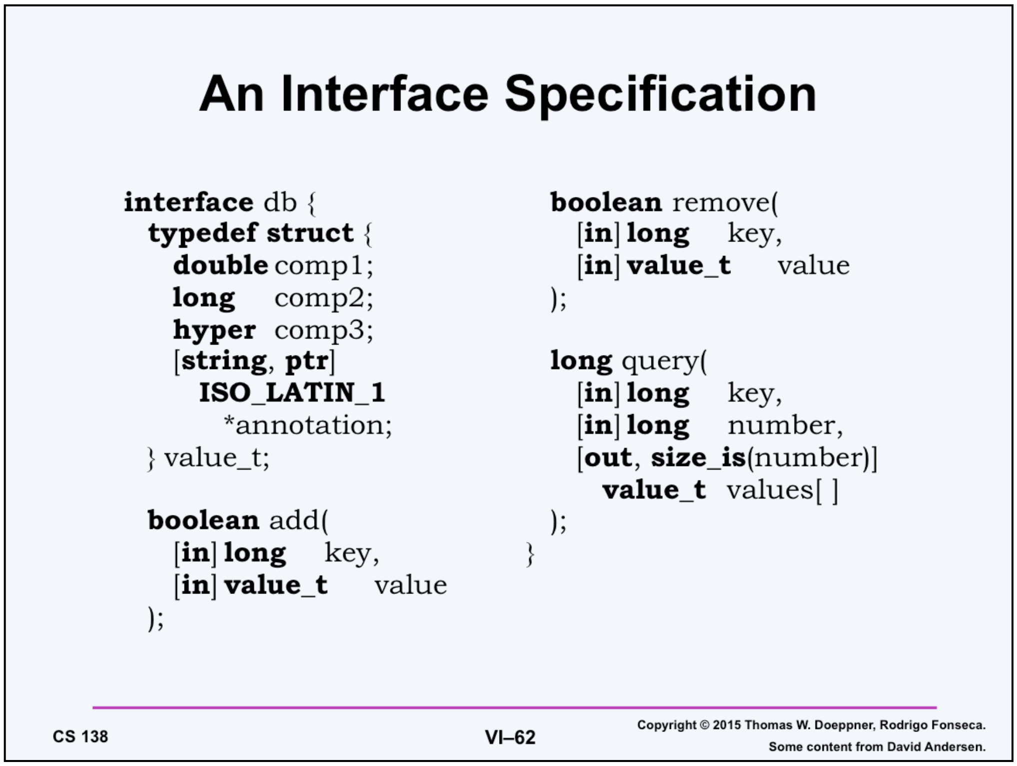
- Designed by Apollo and Digital in the 1980s
  - both companies later absorbed by HP
- Does everything ONC RPC can do, and more
- Basis for Microsoft RPC

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[ ]);
```


Now we re-do the example using DCE RPC.

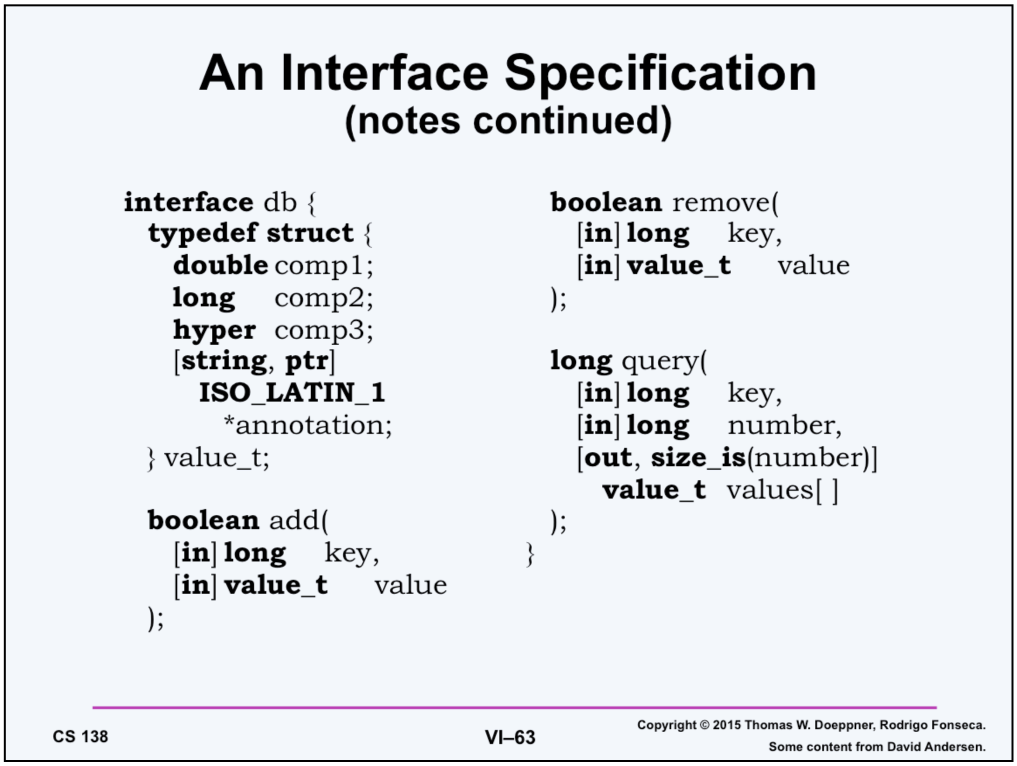


DCE RPC uses an augmented C syntax known as *interface definition language* (IDL) to express its interfaces. By compiling a description written in IDL (using a special IDL compiler), one automatically produces the client- and server-side stubs.

Here we have the IDL description for the simple database example of the previous slide. It starts with the declaration of a data type that is that of the somewhat complex items stored in the database. Following this are the specifications of the three (remote) procedures that clients may call—one for adding values to the database, one for removing values, and one for issuing queries. Values are entered into the database in association with keys. The query operation returns all values, up to an indicated maximum number, that share the given key.

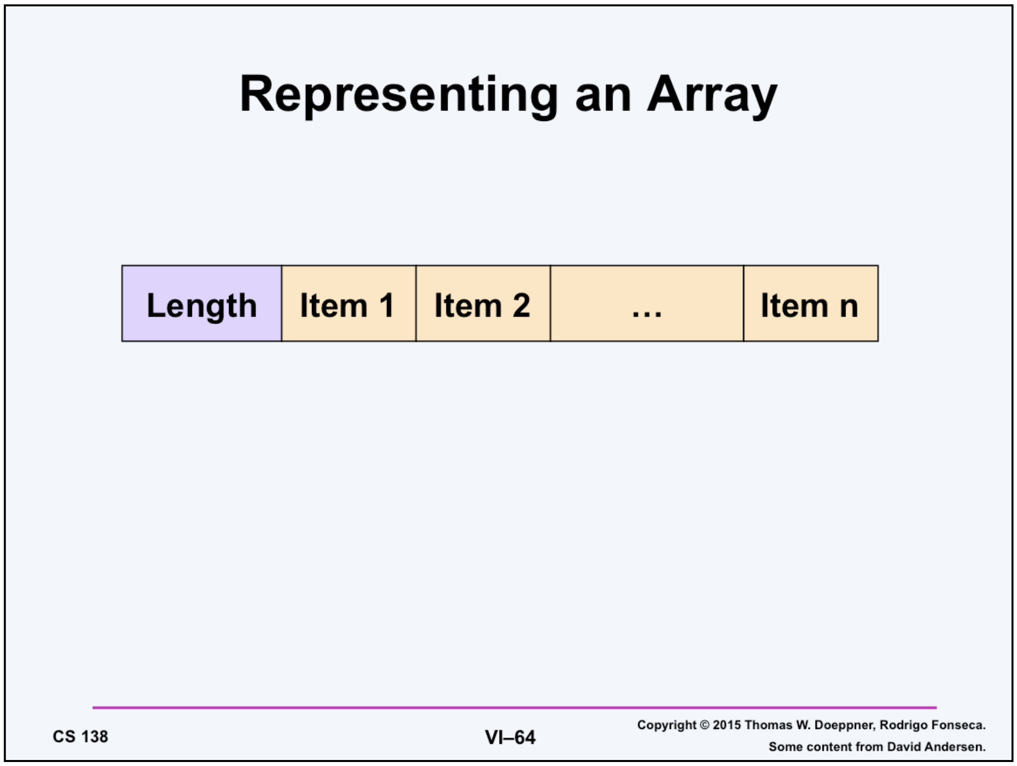
One of the purposes of IDL is to overcome various shortcomings of the C syntax for declaring procedures. Among the issues are:

- Which arguments are input arguments, which are output arguments, and which are both? There is no way to determine this from standard C syntax. With IDL, we have new attributes, enclosed in square brackets, that identify the use of the arguments.
- What is an integer? C has three signed integer types: *char*, *short*, and *long*. One can also declare something to be an *int*, but, depending on the architecture, it will either be a *short* or a *long*. (Of course, we must at some point deal with 64-bit architectures. IDL has a 64-bit integer type called *hyper*.) *Short* and *long* are pretty straightforward, but what is a *char*? Its name certainly implies some ambiguity. To eliminate this ambiguity, in IDL, if one wants an 8-bit signed integer, one calls it a *small*.

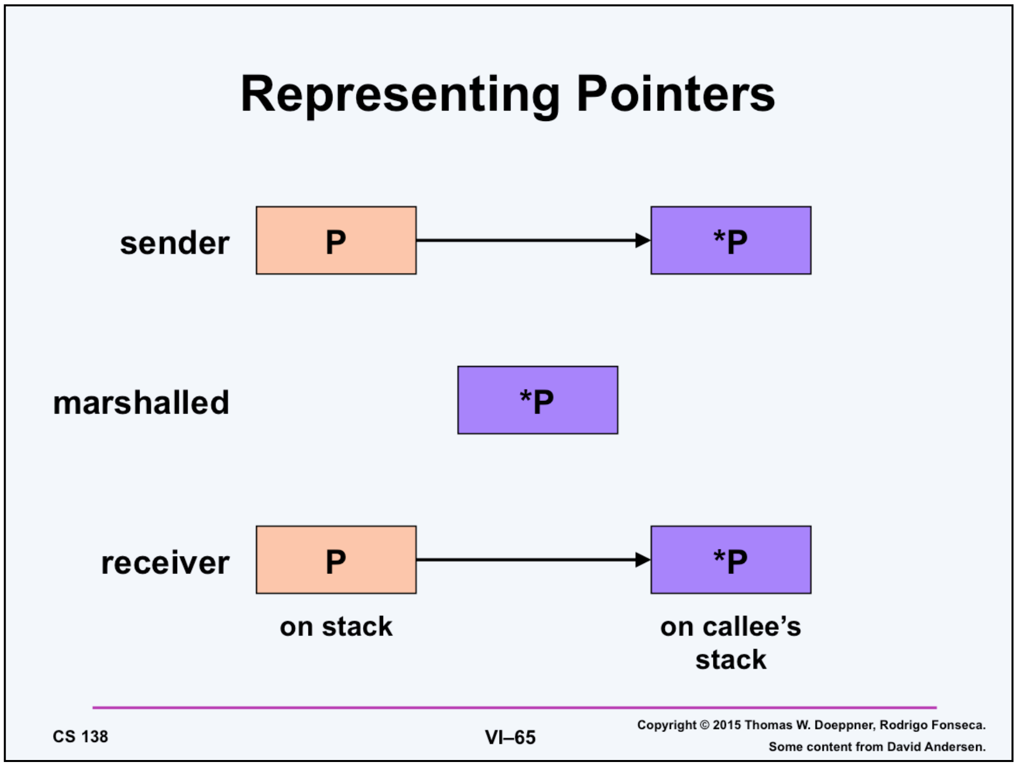


- If we pass an array, how big is it? From our knowledge of the program, we can see that one of the arguments is the size of the two arrays, but how would the IDL compiler know this? It must know the size of the arrays, so that it can determine how much data to pass to the server (and how much data the server should pass to the client for output arguments). In the example here, we use the *size_is* attribute to notify the IDL compiler that one of the arguments is the size of the array.
- What does it mean to pass a pointer argument? For example, if one of the arguments is declared *char* * what would that mean? We certainly don't want to follow a pointer on the server back to the client, so we should pass the data pointed to along with the pointer. But how much should be passed? Is a *char* * really a pointer to a single character? Is it a pointer to a null-terminated string? Is it a pointer to a counted array of bytes? The IDL syntax allows us to distinguish these cases (and supply whatever additional information is necessary).
- When we pass strings of characters, which character set are we referring to? ASCII is only acceptable (and just barely) in English-speaking countries (for which a seven-bit character set suffices). In the example we specify ISO_LATIN_1, an eight-bit character set, which is suitable for most of Europe and the Americas, but for little of Asia, which needs a sixteen-bit (at least) character set.

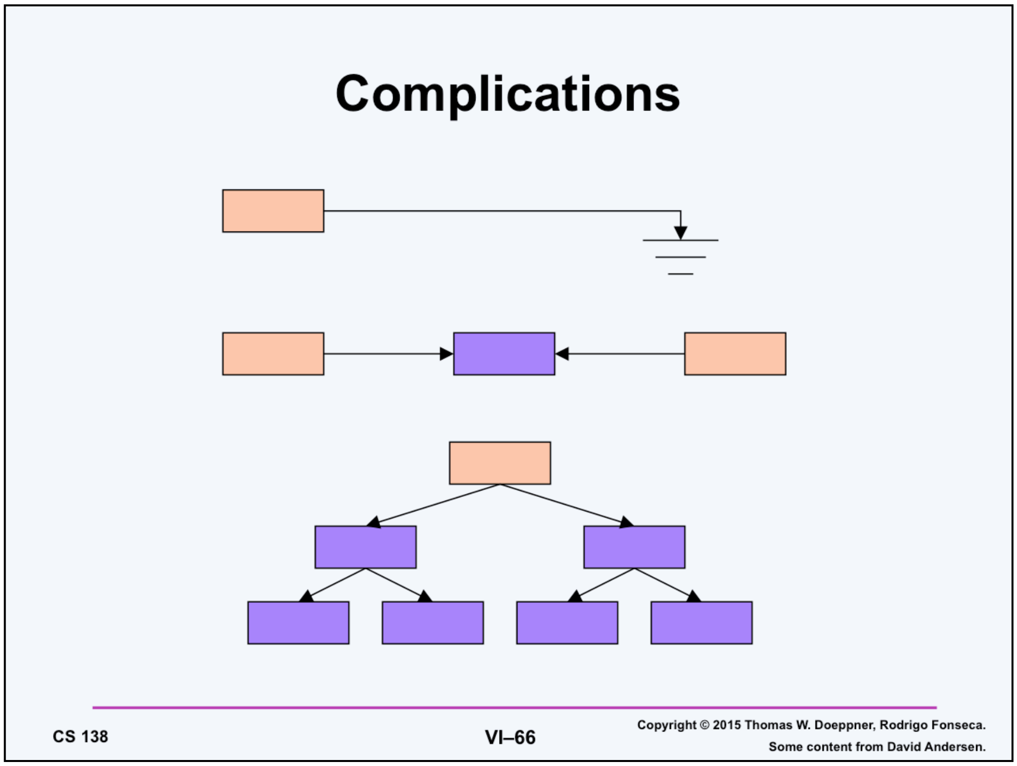
There are alternatives to this approach to handling the data types of parameters. One popular approach, used in a early RPC protocol developed at Xerox in the '70s and now popularized in Microsoft's .Net and other recent systems, is, rather than have linked-in stubs that "know" the types of parameters, to send the typing information along with the parameters. Thus, rather than simply sending the integer "6", what would be sent is "INT 6". This is currently being done in conjunction with XML.



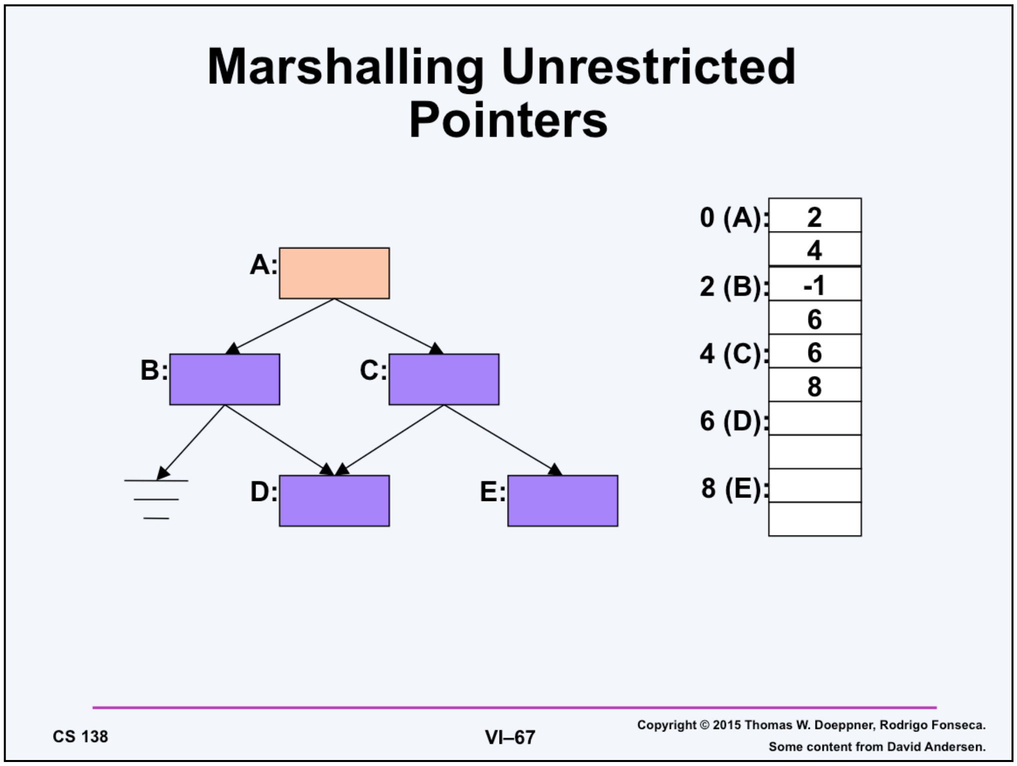
To represent an array, we need to include its length.



Marshalling pointers is sometimes pretty simple: one simply transmits the target of the pointer, rather than the pointer. Unmarshalling depends on whether the receiver is a callee (i.e, the pointer is an input parameter) or is a caller (the pointer is an output parameter). For a callee, the pointed-to item is copied into storage on the receiver's stack in the server-side stub's frame; the stub passes a pointer to the item to the remote procedure. For a caller, the marshaled item is copied into the original target of the pointer; the pointer itself doesn't change.

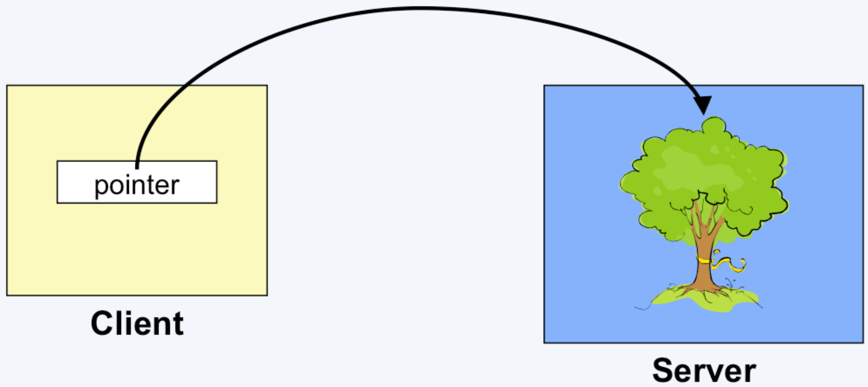


Three situations can complicate the marshalling of pointers. The first is when a pointer contains a null value: since it's pointing at nothing, there's nothing to send! The second is when two different pointers point to the same location (this is known as *aliasing*). It's not enough to send the value of what the pointer points to: the reconstructed pointers on the receiver must also point to the same location. Lastly, what if the pointer points to a data structure containing another pointer?



Marshalling unrestricted pointers, i.e., pointers that might be null, might be aliased, or might point to data structures containing other pointers, requires that one send a representation of how the data structure is organized. One such representation is illustrated in the slide: what the pointers point to is represented as an array and the pointers are represented as indices of the array.

Referring to Server State



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
  );
}
```


Rather than pass a tree back and forth between client and server, it might make more sense to leave the tree on the server and have the client merely send the server requests to perform operations on it. The interface shown here has two procedures—one to create and initialize a tree, and another to add new items to it. From the client's point of view, the tree is represented as an opaque pointer of type *context handle*. This is created implicitly (on both client and server) via the use of the *out* parameter of the *create* procedure. The client holds onto this handle; whenever it uses the handle with the *insert* procedure, it is converted on the server side to point to whatever the pointer pointed to that the server originally returned via the *out* parameter of the *create* procedure.

If the server crashes, then, from the client's point of view, the context handle becomes useless. The client will be notified of a server failure if it tries to use the context handle after the server is known to have crashed.

If the client crashes, then the server might want to be notified, especially if the client is the only one interested in the tree represented by the context handle. In the event of a crash, the server runtime will clean up its state. If the server is interested, it can define a cleanup (or *rundown*) procedure. The name of the cleanup procedure in this case would be *tree_t_rundown*, which will be called with the server-side pointer (to the tree) as the argument. The server, in this example, would free the storage that had been allocated for the tree.

Go RPC

```
package server

type Args struct {
    A, B int
}
type Quotient struct {
    Quo, Rem int
}
type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
    *reply = args.A * args.B
    return nil
}
func (t *Arith) Divide(args *Args, quo *Quotient) error {
    if args.B == 0 {
        return errors.New("divide by zero")
    }
    quo.Quo = args.A / args.B
    quo.Rem = args.A % args.B
    return nil
}
```


This go example from the gorpc page: http://golang.org/pkg/net/rpc/

Server Startup

```
arith := new(Arith)
rpc.Register(arith)
rpc.HandleHTTP()
l, e := net.Listen("tcp", ":1234")
if e != nil {
    log.Fatal("listen error:", e)
}
go http.Serve(l, nil)
```

Client Startup

```
client, err := rpc.DialHTTP("tcp", serverAddress + ":1234")
if err != nil {
    log.Fatal("dialing:", err)
}
```

Client Call

```
// Synchronous call
args := &server.Args{7,8}
var reply int
err = client.Call("Arith.Multiply", args,
&reply)
if err != nil {
      log.Fatal("arith error:", err)
}
fmt.Printf("Arith: %d*%d=%d", args.A, args.B,
reply)
```

Client Call

```
// Asynchronous call
quotient := new(Quotient)
divCall := client.Go("Arith.Divide", args,
quotient, nil)
replyCall := <-divCall.Done   // will be equal to
divCall
// check errors, print, etc.
```

RMI: a Remote Interface

```
package compute;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Compute extends Remote {
    <T> T executeTask(Task<T> t) throws RemoteException;
}
```


This example (extending through the next five slides) is taken from http://download.oracle.com/javase/tutorial/rmi/overview.html (and the code is copyrighted by Oracle). The idea is that we are designing a “compute server” to which we can send an object representing a computation to be performed. The server performs the computation and sends back the result.

We start with a declaration of the server’s interface. By extending Remote, it becomes the interface to a remote object, meaning that clients with references to the object can invoke its methods remotely. Note that all methods of remote objects must be declared as throwing *RemoteException*, which occurs when there is some sort of problem, such as a communication error.

The interface provides one method, that takes a parameterized type, Task<T> as an argument, and returns something of type T. Thus effectively the method has two parameters: Task<T> and T.

RMI: the Argument

```
package compute;

public interface Task<T> {
    T execute();
}
```


Here is the declaration of Task<T>. Note that it is not a remote object, but will be passed (by copying) to the Compute object.

RMI: the Server (1)

```
package engine;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import compute.Compute;
import compute.Task;

public class ComputeEngine implements Compute {
    public ComputeEngine() { super(); }
    public <T> T executeTask(Task<T> t) {
        return t.execute();
    }
```


Now that we've seen the declaration of the interface, we look at the server itself, whose code begins on this slide. The class Compute Engine implements the Compute interface. Its constructor simply calls upon the superclass constructor. Its executeTask method simply invokes the execute method of the supplied Task<T> argument.

RMI: the Server (2)

```
public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String name = "Compute";
            Compute engine = new ComputeEngine();
            Compute stub =
                (Compute) UnicastRemoteObject.exportObject(engine, 0);
            Registry registry = LocateRegistry.getRegistry();
            registry.rebind(name, stub);
            System.out.println("ComputeEngine bound");
        } catch (Exception e) {
            System.err.println("ComputeEngine exception:");
            e.printStackTrace();
        }
    }
}
```


The main routine, which is static, after setting up a security manager (which restricts what remote invocations of the object can do) creates an instance of a *ComputeEngine* object and makes it available for clients to access. The first step in this is to set things up so that the object can receive invocations of its method from remote clients. The *exportObject* method of *UnicastRemoteObject* does this: the first argument is the remote interface being offered to clients and the second is the port on which to receive requests (0 means to use the default port).

Note that remote interfaces do not have constructors, thus clients must be given some external means for getting references to remote objects. This is done here by putting the name of the object in the local registry along with the reference to the remote object provided by *exportObject*. Thus clients can contact the registry and get the reference associated with the name.

RMI: the Client

```
public class ComputePi {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String name = "Compute";
            Registry registry = LocateRegistry.getRegistry(args[0]);
            Compute comp = (Compute) registry.lookup(name);
            Pi task = new Pi(Integer.parseInt(args[1]));
            BigDecimal pi = comp.executeTask(task);
            System.out.println(pi);
        } catch (Exception e) {
            System.err.println("ComputePi exception:");
            e.printStackTrace();
        }
    }
}
```


Here we have code that's run by clients. As with servers, a security manager must be set up to enforce restrictions on what objects can do that are returned by servers. The client then gets the object reference from the server's registry (the name of the server is passed to the client code as an argument). It then creates an instance of an object that represents a computation to be done (in this case: compute the value of pi to a given number of digits). This object is passed to the remote object as an argument and the result of the computation is returned.

RMI: the Client's Compute Object

```
package client;

import compute.Task;
import java.io.Serializable;
import java.math.BigDecimal;

public class Pi implements Task<BigDecimal>, Serializable {
    private final int digits;
    public Pi(int digits) {this.digits = digits;} // constructor
    // lots of stuff deleted …
    public BigDecimal execute() {
        return computePi(digits);
    }
    // lots more stuff deleted …
}
```


Lastly we have the skeleton of the object that computes pi. Note that the object must implement *Serializable* so that it can be marshalled and unmarshalled.

Other examples

- Grpc. Developed at Google
  - Protocol Buffers as an IDL
  - HTTP2 as a transport
  - Many languages (C, C++, Java, Go, Node.js, Python, Ruby, Objective-C, PHP and C#)
- Thrift.
  - Developed at Facebook.
  - Now Apache Open Source. Supports multiple data encodings & transport mechanisms. Even more languages.
- Avro. Also Apache standard. Created as part of Hadoop project. Uses JSON. Not as elaborate as Thrift.