Remote Procedure Calls

Arvind Krishnamurthy
Spring 2003

Remote Procedure Call

- Classic RPC System: Birrell, Nelson
  - Different kind of protocol from TCP
  - Not designed for one-way flow
  - Request-response style of interaction (client-server style)
  - Lightweight
  - Ideally suited for single ethernet/LAN;
    - no long distance communication
    - no round-trip calculation
    - no sliding window, etc.
  - Very little state associated with it
- Procedure model:
  - execute code on remote machine
  - return result

Java Remote Method Invocation

- Object oriented version of RPC
- Key components/class definitions in RMI:
  - Name server (provides location information of services)
  - Interface definition of server code
  - Implementation of server
  - Implementation of client
- Example: date server interface definition

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Date;

public interface DateServer extends Remote {
    public Date getDate() throws RemoteException;
}
```

RMI Server Code

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Date;

public class DateServerImpl
    extends UnicastRemoteObject implements DateServer {

    public DateServerImpl() throws RemoteException { }
    public Date getDate() {
        return new Date();
    }
    public static void main(String() args) {
        DateServerImpl dateServer = new DateServerImpl();
        Naming.bind(“Date Server”, dateServer);
    }
}
```

RMI Client Code

```
import java.rmi.Naming;
import java.util.Date;

public class DateClient {
  public static void main(String[] args) {
      DateServer dateServer =
         (DateServer)Naming.lookup(“rmi://” + args[0] +
                                  “/Date Server”);
      Date when = dateServer.getDate();
      System.out.println(when);
  }
}
```

Name Server

- Allow runtime binding of clients to servers
- Basically a table
  - Can be made more fancy with server attributes/geographical preferences etc.
- Central place to perform access control
- Fail over: if server fails, use another
- Provides the bootstrapping mechanism:
  - Need to know where the name server is running
- Java remote objects are represented by:
  - Machine (ip address)
  - Port number where the server is listening on
  - Unique object id number inside the machine
- In Java, start name server by running the command: “rmiregistry”

Garbage collection

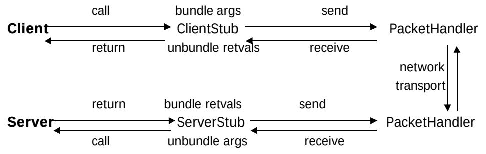
- Problem: Java garbage collects objects that are not being used (referenced)
- Ideal solution: perform distributed garbage collection
  - Difficult, requires global coordination (stop everyone and collect garbage)
  - Inefficient
- Java solution:
  - RMI support code keeps track of who is actively using the remote objects
  - Pings periodically to check whether client is alive
  - While client is alive:
    - Keeps a pointer to the object in a table (not really use the object, but just hang on to a reference so object is not garbage collected)

RPC

- Characteristics:
  - Synchronous
  - Little data
    - Performance dominated by latency issues
- Which one is better?
  - Message based model, or
  - Procedure based model
    - Easier programming model
    - Depends on application

Remote procedure call

- Call a procedure on a remote machine
  - Client calls: `remoteFileSys->Read (“foo”)`
  - Translated into call on server: `fileSys->Read(“foo”)`
- Implementation: (1) request-response message passing (2) “stub” provides glue on client/server



call
bundle args
send

Client
ClientStub
PacketHandler

return
unbundle retvals
receive

network
transport

return
bundle retvals
send

Server
ServerStub
PacketHandler

call
unbundle args
receive

Implementation (cont’d)

- Client stub
  - build message
  - send message
  - wait for response
  - unpack reply
  - return result
- Server stub
  - create N threads to wait for work to do
  - loop: wait for command
    decode and unpack request parameters
    call procedure
    build reply message with results
    send reply

Implementation issues

- Stub generator --- generates stub automatically.
  - For this, only need procedure signature --- types of arguments, return value.
  - Generates code on client to pack message, send it off
  - On server to unpack message, call procedure
- Input: interface definitions in an interface definition language (IDL)
- Output: stub code in the appropriate source language (C, C++, Java, …)
- Stub generator in Java: “rmic”
- Examples of other modern RPC systems:
  - CORBA (Common Object Request Broker Architecture)
  - DCOM (Distributed COM)
  - Microsoft Ole

Announcements

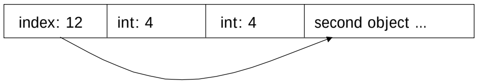
- Readings for next week:
  - Background reading: “Unix Security”
  - Paper review for “Authentication in distributed systems” due on Monday

Stub Code Issues

- Arguments, results are passed by value
- Simple case: procedure take two integers and a short

int: 4	int: 4	short: 2

- More complex case (supported in some Java implementations): object contains pointers to other objects
- Consider: object with a pointer to another object and two ints
- Need to optimize for sharing
- Reverse operation performed while unpacking



index: 12	int: 4	int: 4	second object ...

Implementation Details

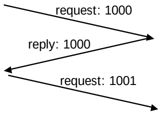
- Client can't locate the server
  - Procedure returns error upon binding time
- Request and reply messages are acknowledged; do timeout and retransmit
- All requests are accompanied by a sequence number for the client
- Server maintains per client:
  - Sequence number of last request
  - Result generated by last request

RPC Failure Situations

- If no acknowledgement to request:
  - Caller retransmits
  - If request message was lost, callee just sees the request as a new request
  - If acknowledgement was lost, callee uses request sequence number to identify duplicate requests
- If no acknowledgement to reply:
  - Callee retransmits
  - If reply message was lost, caller just gets one unique reply
  - If acknowledgement was lost, caller uses the request sequence number (for which this reply was the result) to identify duplicate replies
- Server crash: use another one.
- Client crash
  - log RPC calls and then ask server to kill orphans upon recovery
  - Wait for a while before reusing sequence numbers

Performance

- Typically: one packet for args, one for results
- Each packet must be acknowledged
  - Piggyback the acknowledgement to the result
    - Result acknowledges the request
    - Next request acknowledge the previous result
    - Server state: table of caller sequence ids; can be flushed after a while



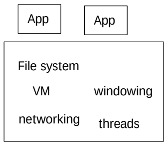
  - Long call: client retransmits, server acknowledges the second request, client prods periodically

Performance (contd.)

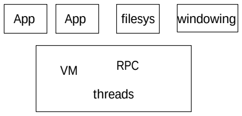
- Large messages: (file blocks not fitting in a packet)
  - Acknowledge per packet
  - Doubles packets, adds latency
  - Alternative: stream data, send back a bit-mask of acknowledgements
  - Extra complexity is probably worth-while
  - Performance: 1.2ms/call in original RPC implementation
    - 2000-5000 instructions
    - 100us (best today?)
    - 3-4x kernel call
  - Overheads:
    - Two copies into kernel and two copies out of kernel
    - Two marshalls and two un-marshalls
    - Two context switches on server and two on client

Cross-Domain Communication

- How do address spaces communicate with one another?
  - File system
  - Shared memory segments
  - Pipes
  - Alternative: "remote" procedure calls
    - RPCs can be used to communicate between address spaces on different machines or between address spaces on the same machine
- Microkernel operating systems:



Monolithic OS Kernel



Microkernel

Microkernel advantages

- Why split OS into separate domains?
  - Fault isolation: bugs are more isolated (builds a firewall)
  - Enforces modularity
    - Allows incremental upgrades of pieces of servers
    - Can have multiple types of file systems running simultaneously
  - Location transparency:
    - Service can be local or remote
    - Example: x-windowing system