Imperial College London

Computer Networks and Distributed Systems

Part 2.3 – Interaction Implementation

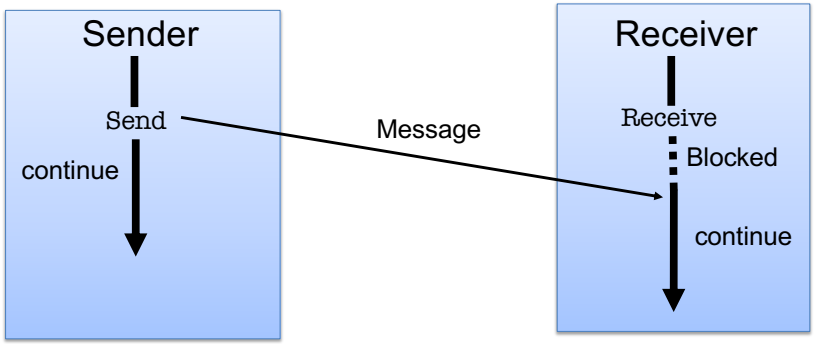
Course 527 – Spring Term 2017-2018

Emil Lupu

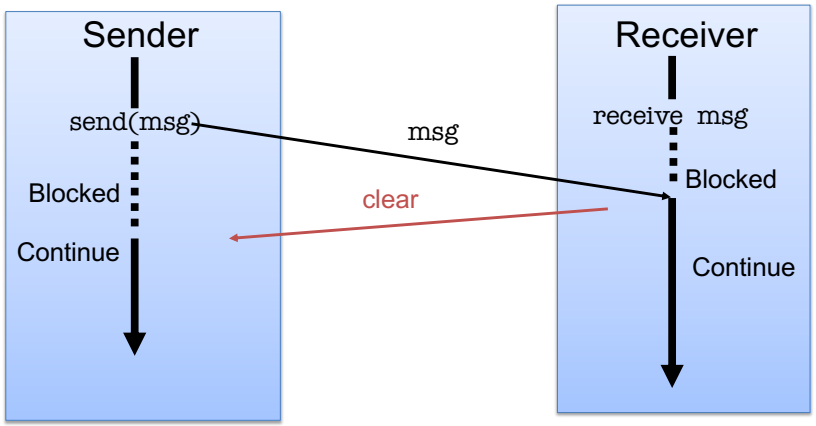
e.c.lupu@imperial.ac.uk

Contents

Implementing Asynchronous Send



Implementing Synchronous Send



Clear is a runtime system message – not sent by application process

Exercise

Modify the synchronous protocol to cater for a timeout on the send
i.e.

send msg delay (t).

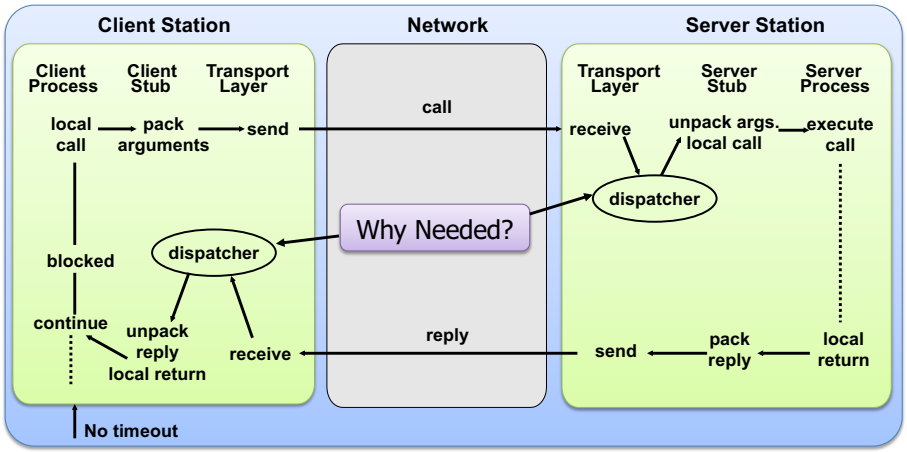
The sender continues after the timeout if the message has not yet been received – this implies the receiver should not get the message if the timeout expires

Show the message exchanges that would occur:

i) if the sender's timeout expires

ii) if the sender's timeout does not expire.

Remote Procedure Call



At most once semantics
client receives reply - procedure executed exactly once
on failure i.e. no reply received - don't know

Dispatcher

Server needs dispatcher to map incoming calls onto relevant procedure.

Dispatcher in client passes incoming reply message to relevant stub procedure.

Interface compiler **generates a number (or name)** for each procedure in interface – inserted into call message by client stub procedure.

Dispatcher at server receives all call messages and uses procedure number (name) to identify called procedure.

RMI Dispatcher

Java uses reflection and a generic dispatcher so no need for skeletons

Client stub(proxy) includes information about a method in request message, by creating instances of Method class containing

- class, types of arguments, type of return value, type of exceptions
- Proxy marshalls object of class method, array of argument objects

RMI Dispatcher

Dispatcher receives request,

- unmarshalls method object,
- uses method information to unmarshall arguments
- converts remote object reference to local object reference
- calls method object's invoke method supplying local object reference and arguments
- when method executed, marshalls result or exceptions into reply message and sends it back to client

See https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Method.html

Binding

Binding is the assignment of a reference value (e.g. address or object reference) to a placeholder (e.g. message port or object reference variable).

First Party Binding: Client initiates binding as in Java RMI, CORBA, etc.

Third Party Binding: A configuration manager establishes binding between clients and servers. Often used with Component based frameworks and architectural description languages. Needs requires as well as provided interface.

RPC Binding

A name server registers exported interfaces and is queried to locate a server when an interface is imported.

Server

- Calls export(interface type, server name, nameserver)
- Dispatcher address added by stub and passed to Transport
- Server's transport generates unique **exportid** & sends a **register** message to name server containing **type, name, exportid**.

Client

- Calls import(interface type, server name, nameserver)
- Dispatcher address added by stub and passed to Transport
- Client Transport:
  - Send query message with **type & name** to nameserver; Reply contains **type and address of server instance**;
  - Query server to check validity of **type, name and exportid**; Return interface reference (address) or error

Failures

Server Failure

- Use `exportid` to detect failed server: on server restart a new `exportid`
- All messages to server include `exportid`
- Dispatcher aborts calls with incorrect `exportid`

Client Failure

- Orphans – client fails after making call but before receiving response
- No ack to response
- Server either implements a form of 'rollback' or does nothing

Interface Type Checking

Client interface must be type compatible with server interface i.e. same interactions and signatures (set of parameters + data types).

Client and server likely to be compiled independently and at different times

- Use same interface type definition to generate client and server interface
- Permit server to be subtype of client interface
- Check for structural compatibility at run-time

Use same interface type definition to generate client and server interface.

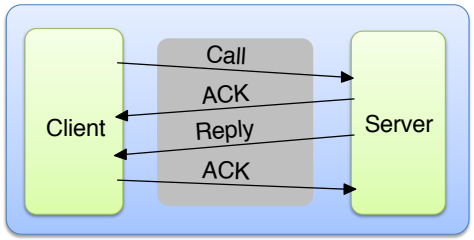
- Client and server hold identity of interface derived from interface definition module.
- Generate Interface identity by
  - checksum over source
  - name + timestamp of last modification or compilation
- At bind time, check type identities are equal
- Strong type compatibility

- Permit server to be subtype of client interface i.e. provides *additional* operations which are not used by client, but must not extend operations in original interface.
- Maintain run-time representation of interface and check for structural compatibility at bind time ***Weak*** type compatibility eg. the following two interfaces are structurally equivalent.

```
interface A {
    opa1 (in string a1,
        in short a2 , out long a4);
    opa2 (in string a4);
}
```

```
interface B {
    opb1 (in string b1,
        in short b2 , out long b3);
    opb2 (in string b4)
}
```

RPC Error Control



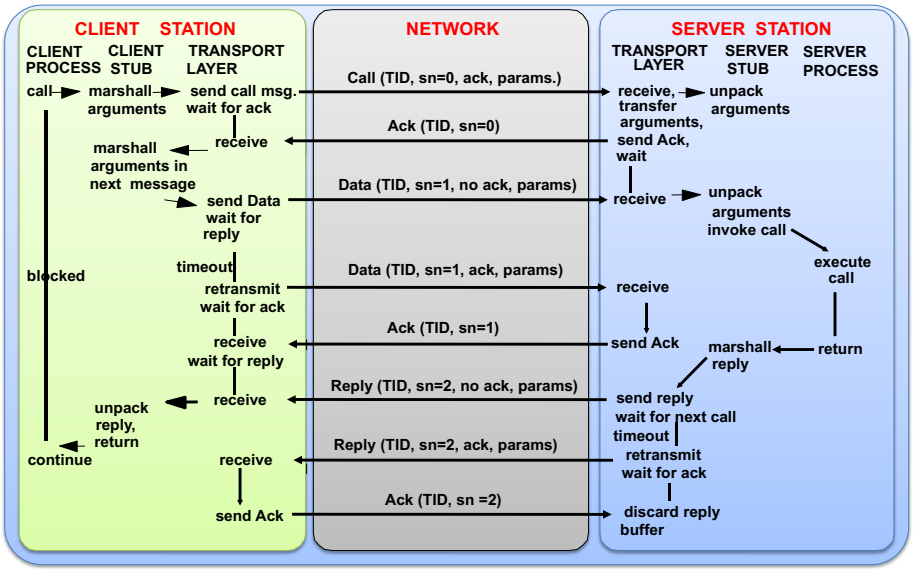
Error Control

- After sending message set timeout
- Retransmit if no ACK
- Save reply until ACK received in case call repeated.

How can this be optimised?

Must also cater for long parameters requiring multiple messages to transfer

RPC Implementation



RPC parameters.

TID = Transaction identifier plus interface export identifier.

sn = message sequence number

ack = please acknowledge message

no ack = no acknowledgement expected

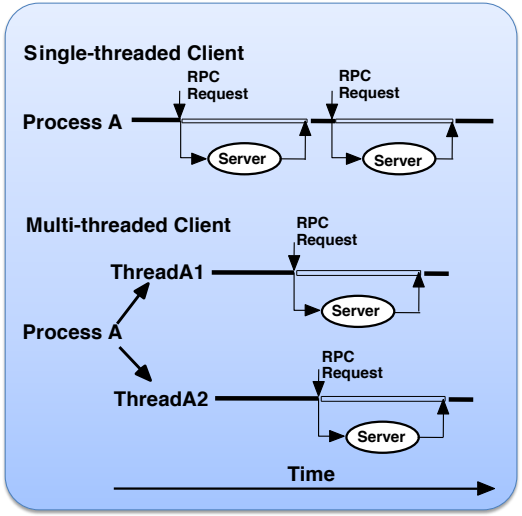
params = in or out parameters

Client Threads

In a single-threaded program which does RPCs to different servers, the RPCs must be done serially.

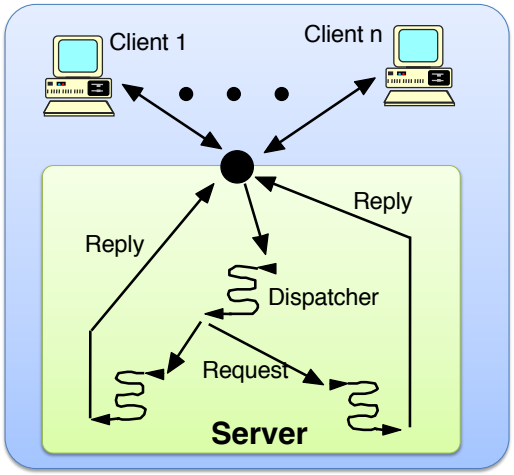
Each RPC blocks the program for at least 2 * the network delay. Throughput is adversely affected.

Using threads, remote invocations (RPC or object invocation) may be performed concurrently by a single client process.



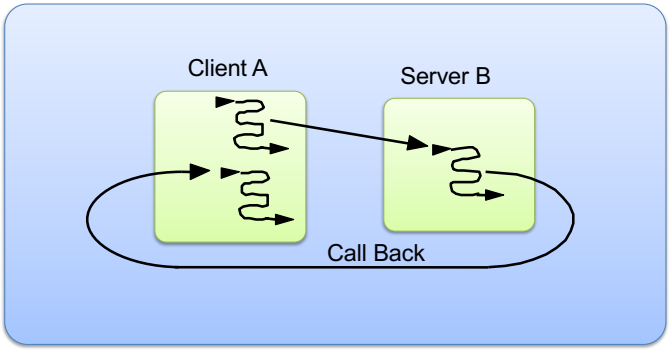
Server Concurrency

Multi-threading can improve server responsiveness since if requests are processed concurrently, long requests will not block short requests.



Client Concurrency

No dead-locks with callbacks if client multi-threaded



Server Implementation Options

Server is single active process

- Dispatcher processes one request at a time and calls the relevant stub procedure which calls the actual procedure. Problems?

Thread-per-Request

- Dispatcher creates a new thread to handle each request. Problems?

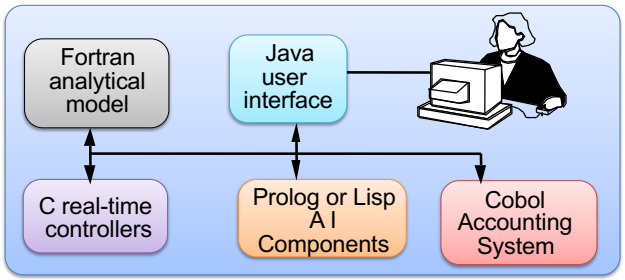
Thread Pool

- A fixed number of threads are generated at start-up and free threads are allocated to requests by the dispatcher
- Concurrency but lower creation overheads

Thread-per-Session

- A thread is created at connection set up to process all requests from the particular client. Problems?

Language Heterogeneity



Data structure representation differences:

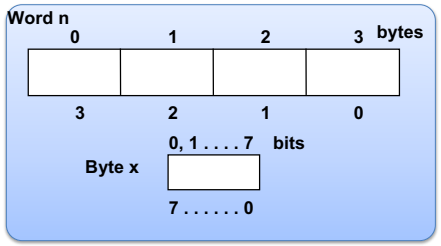
- Array implementation
- Record implementation
- Alignment of bytes on words etc.
- No equivalent data structure eg no records in Fortran, no lists in C

What can be done about this?

Processor Heterogeneity

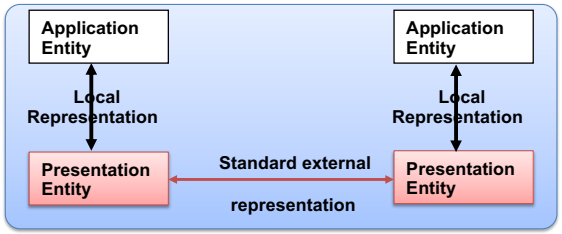
Computers differ in representation of:

- Characters - Ascii, Ebcdic, graphics……
- Integers - 1 or 2's complement
- length
- Reals: mantissa & exponent length, format, base 2, 16 …
- Bit and byte addressing within a word



Need to transform representations when transferring data

Standard External Data Representation (XDR)



Standard external data representation (XDR) reduces number of translators (each machine knows only about its own data representation and the external representation). Transformation must:

- preserve meaning – can be difficult
- resolve syntax differences

Overhead of conversion when communicating between machines of same type.

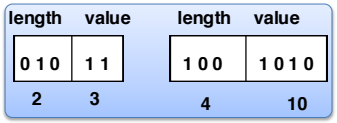
Data Encoding e.g. XDR Characteristics

Fixed Length

- 16 or 32 bit integers
- more efficient transformation
- maximum value limitation -> truncation

Variable length

- Eg. strings of printable characters to represent numbers
- Requires length indicator or end delimiter
- No value limitation
- Inefficient 6 bytes for 16 bit integer.
- Packed binary -> discard leading 0's
- Length field usually fixed length or extensible in bytes
  most significant bit set -> another byte follows



Implicit Type

- Types must be known in advance at receiver
  e.g. ports, object method parameters
- Fewer overheads

Explicit Tag or Type Identifier

- Increased overheads
- Information to perform transformation is self contained in message
- Position independent
- Can perform dynamic type checking

Extensible Markup Language (XML)

- Text based, explicit tags -> human readable
- Very verbose, not human friendly -> really aimed at machine processing
- Data items tagged with 'markup' strings describing logical structure
- Use start and end tags rather than length
- Extensible – users can define own tags
- Used for internet interactions and data storage e.g. XML databases
- Very inefficient encoding but can be compressed.

XML Elements and Attributes

Element: container for data – enclosed by start and end tag

Attribute: used to label data – usually name/value

```
<person id="123456789">
        <name>Smith</name>
        <place>London</place>
        <year>1934</year>
        <!-- a comment -->
</person >
```

Attribute

Place element

Person Element

XML Namespace

- Namespace used to scope names
- A set of names for a collection of element types and attributes
- Referenced by a url
- Specify namespace by a xmlns attribute
- Can use namespace name as prefix for names

```
<person pers:id="123456789" xmlns:pers =
"http://www.cdk4.net/person">
        <pers:name> Smith </pers:name>
        <pers:place> London </pers:place >
        <pers:year> 1934 </pers:year>
</person>
```

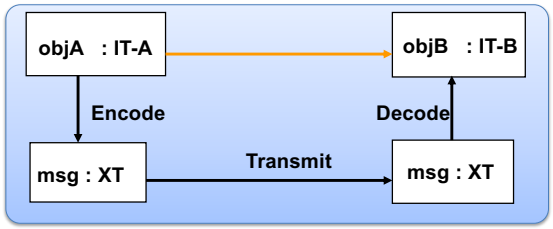
Namespace attribute

XML Schema

- Defines elements and attributes that can appear in a document
- Defines element nesting, number, ordering, whether empty or can include text
- For each element defines type and default value

```
<xsd:schema  xmlns:xsd = URL of XML schema definitions  >
  <xsd:element name= "person" type ="personType" />
    <xsd:complexType name="personType">
      <xsd:sequence>
        <xsd:element name = "name"  type="xs:string"/>
        <xsd:element name = "place"  type="xs:string"/>
        <xsd:element name = "year"  type="xs:positiveInteger"/>
      </xsd:sequence>
      <xsd:attribute name= "id"   type = "xs:positiveInteger"/>
    </xsd:complexType>
</xsd:schema>
```

Representation Transformation



What problems could occur when doing transformations eg with numbers?

Semantics of Representation

Two representations can have similar syntax but different meaning

- eg. complex numbers – (float x,y) = rectangular or polar coordinates -> transformation is application dependent

Type may have no meaning outside own context

- eg. pointer, file name

Procedures passed as parameters

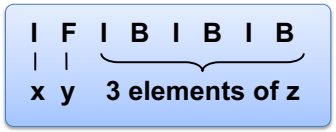
- Cannot always transfer code to different computer for execution.

Example of Use of Encode

```
struct rec {
        int a;
        boolean b;
};
struct form {
        int x;
        float y;
        rec z [ 3]; /* assume 3 elements */
};
form obj = (5, 23.75, 10, true, 5, false, 7, true)
```

can be "flattened" for transfer:

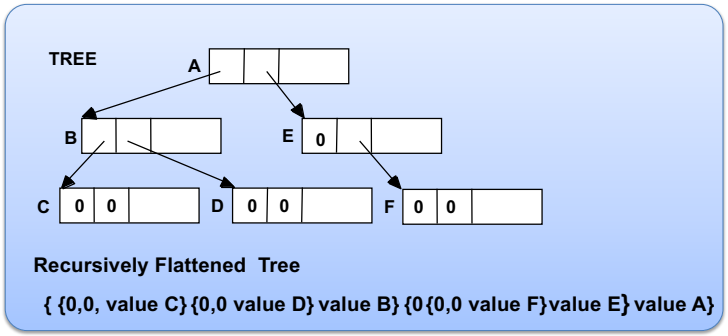
where I = int, F = float, B = boolean



Structural Information

Structural information must be maintained

- Structural information represented internally by pointers (addresses)
- must be flattened into a linear message



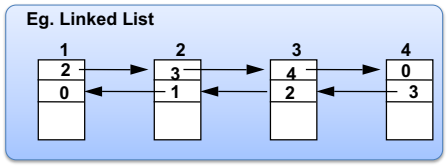
Transferring Cyclic Structures

Use Encode and Decode procedures for primitive types and simple constructed types

Structural information must be flattened:

- Number sub-objects
- Transform pointers into handles (ie. number) of sub-objects.

Sub object 1
Handle 2
Null
Contents
Sub object 2
Handle 3
Handle 1
Contents
Sub object 3
Handle 4
Handle 2
Contents



Sub object 4
Null
Handle 3
Contents

Java Object Serialization

Java objects can be passed as arguments and results in RMI. Object is an instance of a Java serializable classs

```
public class Person implements Serializable {
   private String name;
   private String place;
   private int year;
   public Person (String aName, String aPlace, int aYear) {
        name = aName;
        place = aPlace;
        year = aYear;
        }
   // methods for accessing instance variables
}
```


Serialize with

```
ByteArrayOutputStream byteobj =
        new ByteArrayOutputStream();
ObjectOutputStream out =
        new ObjectOutputStream(byteobj);                );
out.write(new Person("Joe", "Paris", 2003);
out.close();

byte[] buffer = byteobj.toByteArray();
```


Java objects can contain references to other objects
All referenced objects are serialized together

References are converted to handles ie internal references to object within the serialized form

Each object is serialized only once – detect multiple references to same object.

Serialization:

- Write class information
- Write types and names of instance variables
- If instance variables are of a new class, then write their class information followed by types and names of instance variables.
- Uses reflection – ability to enquire about properties of a class eg names and types of instance variables and methods

Summary

Message passing systems map closely onto the underlying communication services, however RPCs and Object invocation are more complex to implement.

They require binding implementation and have to cater for failures of client, server, name servers or communication system.

Translation to a standard external representation. Complex data types must be "flattened" for transfer to a remote machine and addresses transformed to local references (e.g. array index)

Some types cannot be transferred e.g. memory addresses