Tutorial

- Design a Chat Room System
- Use any architecture you wish
  - e.g. centralised or decentralised model or …
- Provide Simple UML model
  - I.e. What components in system
  - How do they interact
- Describe technologies you would use
- Justify technology choices

Distributed Systems

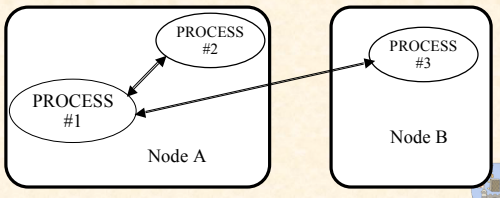
- **Why are distributed algorithms hard?**
- **Complex Problems:** algorithms that are worth distributing, or require distribution, are usually inherently complex
- **Sense data limitations:** distributed algorithms must deal with imperfect information!
  - The propagation delay in transmitting data from one node to another means that the information can be out of date by the time it is used to make a decision.
  - In general in a loosely coupled distributed system, no single process has a coherent view of the entire system at any one time, or can know the state of any other node with 100% confidence

Distributed Systems

- **Failure:** "A distributed system is one where the failure of some node you have never heard of causes your program to crash" (Lamport)
- **Parallelism/concurrency:** Multiple processing elements means that things can go on in parallel
- **Performance and scale:** The algorithms must be efficient and not require change as the system scales
- **Lack of Software Support:** developing distributed applications with technologies such as sockets is very hard. Do technologies such as RMI, CORBA or DCOM help?

Distributed Programming Goal

- Provide abstractions that make it easy to program systems in which processes (potentially on different machines) communicate with each other.



TRANSPARENCY

- **Transparency - The Holy Grail**
- Transparency is the concealment of the underlying physical separation from end users and application programmers.
  - E.g. NFS *allows* files on remote machines to be accessed *as if* they were local.
- The key to achieving ease of programming is the degree to which the system provides transparency
-

Types of Transparency

- **Access** - local and remote resources accessed identically.
- **Location** - resources can be accessed without knowledge of their location.
- **Concurrency** - several users or application programs can concurrently use shared data without mutual interference.
- **Replication** - multiple copies of resources can be used to increase reliability and performance without users or applications being aware of the replicas.

Types of Transparency

- **Failure** - concealment of faults, allowing users and applications to complete their work despite the presence of faults.
- **Execution** - allows the movement of processes within a system (e.g. so as to load share) without affecting operation of users or applications.
- **Scaling** - allows the system to grow without changing the overall system, structure or application programs.
- **Performance** - none of the list above count for anything if the system does not perform "well".

Paradigm Development

- **Paradigm Development**
- The development of **inter process communication** paradigms has been in an upward direction with each new paradigm presenting a higher level of abstraction to the programmer.
- Raw Network I/O … Protocol Stack … File I/O …. Messages ... Remote Procedure Call ... Distributed Object Invocation ….

Sockets (Stream)

- No boundaries in the data
- stream.
- No end to end type checking.
- Manual conversion and buffering
- of data.
- No transparency in error handling.
- Complex Binding.
- Finished up writing pseudo
- message passing primitives.
  - `Send(ProcessID, Structure)`
  - `Receive(SourceID, Structure)`

Message Passing

- Provide a higher level of abstraction.
- Processes communicate (across a network) by exchanging **messages** where a message is a (variable) sequence of bytes.
- The sender must specify the identity of the intended recipient.
- The main message passing primitives are **`SEND(Dest, Msg)`** and **`RECEIVE(Source, Msg)`** with numerous variations.
  - **N.B. pointers are invalid on the remote node.**

Remote Procedure Call - RPC

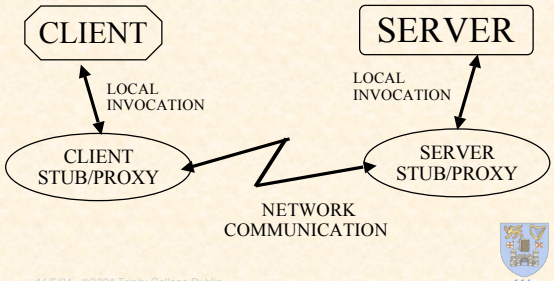
- RPC attempts to make the **procedure call abstraction** available to the distributed systems programmer.
- Appear like local procedure calls at the application level.
  - Stub routines *marshal* the parameters
  - pass them to the runtime for transmission across the network.
- Ideally invoking a remote procedure is identical to calling one in the same address space.
  - This is not totally feasible in practice.
- RPC layered on send/receive/reply.

Marshalling

- *"Marshalling is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message"*
- Convert into the *on the wire format*.
- Use a common external data representation format to handle heterogeneity issues
  - ordering, character code, etc. NOP if system is homogeneous.
- Marshalling Code is needed.

Paradigm Development

- **Marshalling a remote procedure call**

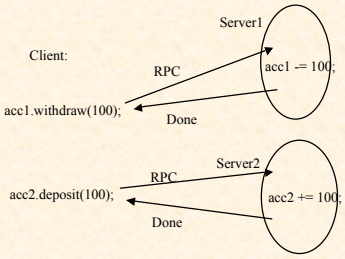


Marshalling

- Define procedures being exported in a separate **Interface Definition Language.**
- Marshalling code (called stub routines) can be automatically produced from this definition for the target language by a compiler/pre-processor.
- This frees the programmer from much of the complexity of distributed systems programming.
- However...

Failure in Remote Procedure Call

- consider moving money between two bank accounts managed by different servers.



Failure in Remote Procedure Calls

- no reply to first request is received?
  - request lost? reply lost? server crashed?
  - has the withdrawal been made or not?
  - retry? but don't want to withdraw money twice!
- Server1 crashes after replying to client?
  - will the client know?
  - will the update to acc1 persist?
- no reply to second request received?
  - as above but what to do about Server1?
  - inconsistency may arise due to partial failure!

Distributed Application programmer must be prepared to handle additional failure modes

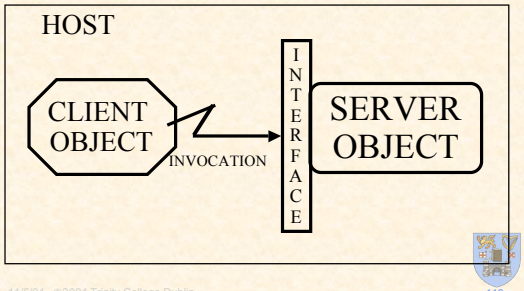
Some programming models returns these failures as language level "events"

Paradigm Development

- **Distributed Object Oriented Programming**
- RPC attempts to make the **procedure call abstraction** available to the distributed systems programmer.
- Object Oriented Distributed Technologies attempt to provide remote access to **objects.**
- *i.e. Make the method call available to the distributed programmer*

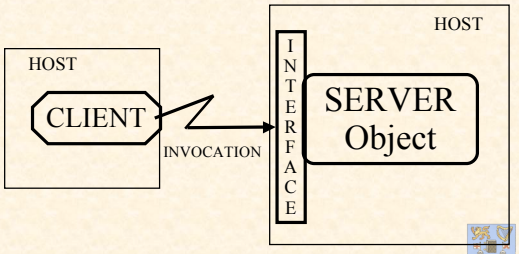
Paradigm Development

- **Object invocation**

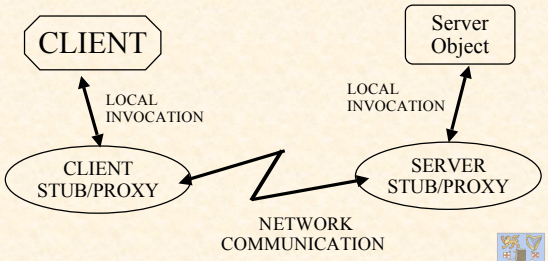


Paradigm Development

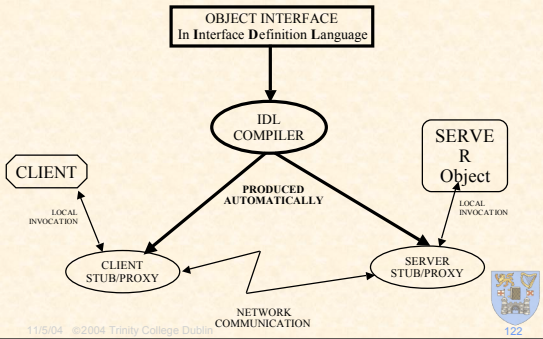
- **Transparent remote object invocation**



Marshalling a remote object invocation

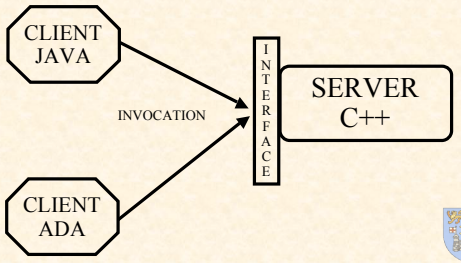


Automatic generation of stubs

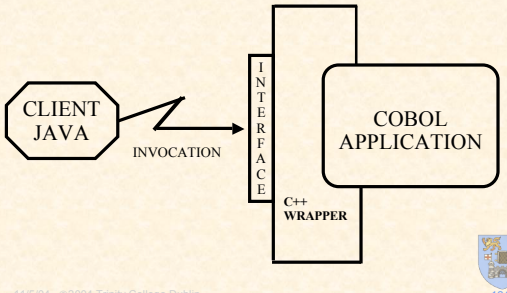


Inter language object invocation

- Interface Description can be Language Independent

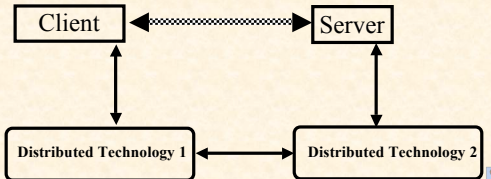


Legacy system integration



Interoperability

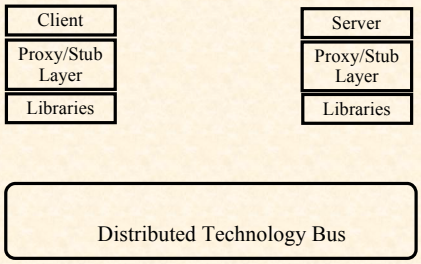
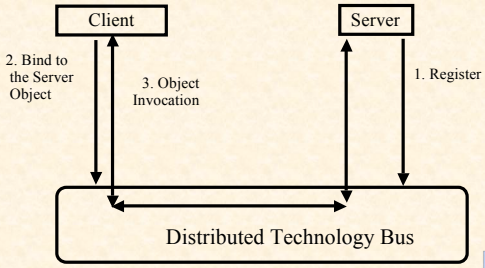
- Enable distributed technologies from different vendors to interoperate.



Paradigm Development

- ***Today's applications are tomorrow's components!***
- Define the interface to the application when it is being written so that it can be reused in another application later on.

Architecture



Architecture in Action