Introduction - Middleware

- See System Models 1 for first encounter
- Middleware consists of two layers as shown:
  
  - Applications, Services
  - RMI and RPC
  - Request – Reply Protocol
  - Marshalling and Data Representation
  - UDP and TCP / Operating System

Middleware

- RMI and RPC Layer
  - Concerned with integrating communication into a programming paradigm by providing RMI or RPC
  - Remote Method Invocation
    - Allows an object to invoke a method in an object in a remote process
    - Examples of systems for RMI are CORBA and Java RMI
  - Remote Procedure Call
    - Allows a client to invoke a procedure in a remote server

Middleware

- Request-Reply Protocol, Marshalling and the External Data Representation Layer
  - Concerned with suitable protocols that support client-server (and group communication)
  - Concerned with the translation of objects and data structures into a form suitable for sending in messages over the network
    - Takes into account different computers may use different representations for simple data items
    - CDK consider a suitable representation for object references in a distributed system

Characteristics of InterProcess Communication

IPC

- IPC best provided through a message-passing system?
  - See CDK Section 3.3.2 - Data Streaming, p74 - audio and video streams
- Function of Message-Passing System
  - To facilitate communication between processes without need to resort to shared data
- IPC facility provides at least two operations:
  - send(message)
  - receive(message)

Send and Receive

- A process p performs a send by inserting a message m into its outgoing message buffer
- The communication channel transports m to process q’s incoming message buffer
- Process q performs a receive by taking m from its message buffer and delivering it
**Synchronous and Asynchronous Communication**

- **Synchronous Communication**
  - Send and receive processes synchronise at every message
  - Both send and receive are blocking operations
    - When a send is issued the sending process is blocked until the corresponding receive is issued
    - Whenever a receive is issued the receive process blocks until a message arrives

- **Asynchronous Communication**
  - Send operation is non-blocking
    - Sending process is allowed to proceed as soon as the message has been copied into local buffer
  - Transmission of message is in parallel with send process
  - Receive operation can be blocking or non-blocking
    - Non blocking case:
      - Receiving process
        - proceeds with its program after issuing receive operation
        - provides a buffer to be filled in background
        - Must separately receive notification that buffer is filled by polling or interrupt
      - Not usually provided by current systems – see CDK p128

**Message Destinations & Location Transparency**

IP messages are sent to (internet address, local port) pairs
- If a client uses a fixed internet address to refer to a service then that service must always run on the same computer for its address to be valid.
- To avoid this and provide location transparency:
  - Client programs can refer to services by name and use a name server or binder to translate the names into server locations at run time.
    - Allows service relocation not migration (move while system running)
  - O/S (e.g. MACH) provides location independent identifiers for message destinations these are mapped to lower level addresses in order to deliver messages to ports
    - Allows service relocation and migration

**Request-Reply Protocol**

- `doOperation` used by clients to invoke remote operations. Specifies remote object, method to invoke and additional information (arguments) required by method
  - It is assumed that client carries out marshalling of request and unmarshalling of reply
- `getRequest` is used by the server to acquire service requests
- `sendReply` is used to send the reply message to the client once the server has invoked the method in specified object
External Data Representation

- Defined to be:
  - An agreed standard for the representation of data structures and primitive values
  - Information stored in running programs is represented as Data Structures for example by sets of interconnected objects
  - Information in messages given as sequence of bytes
  - Irrespective of the form of communication used data structures must be flattened (converted to a sequence of bytes) prior to transmission and rebuilt once they arrive at their destination

External Data Representation

- Primitive data items
  - can be of many different types
  - Different architectures may store different types (e.g. integers, floating point numbers) in different order – big-endian/little-endian

External Data Representation

- Possible Method for Exchange of Data Values
  - Convert values
    - to agreed external format prior to sending
    - To local form upon receipt
  - If computers known to be of same type then omit conversion
  - Alternatively
    - Transmit values in sender's format with detail of format used
    - Let recipient convert the values as necessary
  - Note: bytes are never altered during transmission
    - To support RMI/RPC must be able to
      - Flatten any data types that can be passed as an argument or as result
      - Agree format

Marshalling/Unmarshalling

- Marshalling is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message
  - The translation of structured data items and primitive values into an external data representation
- Unmarshalling is the process of disassembling the received messages to produce an equivalent collection of data items at the destination
  - The generation of primitive values from their external data representation and the rebuilding of the data structures

Design Issues for RMI

- RMI is a natural extension of local method invocation
- CDK pp170-174
  - 5.2.1 The object model
  - 5.2.2 Distributed objects
  - 5.2.3 The distributed object model
Transparency

- The originators of RPC Birrell and Nelson [1984] intended that RPC
  - be as much like local procedure calls as possible
  - Have no distinction between local and remote procedural calls
  - To make semantics of RPC like local proc. calls all
    • necessary calls to Marshalling and message passing
      procedures hidden from the programmer making the call
    • Retransmissions due to timeout transparent to the caller

Transparency

- Transparency has been extended to apply to distributed objects
  - This involves HIDING:
    – marshalling and message passing
    – Task of locating and contacting remote objects
  - Example
    – Java RMI makes RMI invocations like local ones
      by allowing them to use the same syntax

Transparency

- Remote invocations are more vulnerable to failure than local ones
- They involve:
  – Network(s)
  – another Computer
  – another Process
- Possibility of no reply always exists
  – In the case of failure difficult to identify the problem
    • Network or remote process
  • Objects making remote invocations must be able to recover from such failures

Transparency

- Latency for remote invocations
  – Much greater than for local invocations
  – Programs using remote invocations need to take this into account
    • Perhaps by minimising remote interactions
  • Choice regarding transparency for remote invocations is available to designers of IDL’s (Interface Definition Languages)
  – Interface definition languages are designed to allow objects implemented in different languages to invoke one another (See CDK pp168-169) Example CORBA IDL (See Chap 17)

Transparency

In CORBA
(Common Object Request Broker Architecture)
- Remote invocations
  – throw an exception when the client is unable to communicate with a remote object
- Client program
  – must be able to handle such exceptions
  – Allowing client program to deal with such exceptions

Transparency

- IDL’s
  – can also provide a facility for specifying the call semantics of a method
  – This can help the designer of a remote object
    – E.g. If at-least-once call semantics are chosen to avoid the overheads of at-most-once, the operations of the object are designed to be idempotent
    – (idempotent operations are operations that can be performed repeatedly with the same effect as if they were performed just once
      • adding an element to a set is idempotent
      • adding to a list is not idempotent since - list increases each time