Large Scale Distributed Systems

Comp3019

1 Introductory Lecture

a) Motivation

Computing Power increased exponentially but not any more
Can use parallelism to get around this
But nobody knows how to control this power effectively
(limits of about 5Ghz on cores physically as its bounded by the wavelength of light)

b) Million Processor Problem

Instead of CPU controlling things have the data controlling processors
Model on the brain with neurons doing processing and a rather chaotic structure
Multiple neurons put together can simulate simple organisms

c) Multicores

General Purpose Parallelism is one of the key unsolved problems of Computer Science
Imperative in the many core world

d) Biology

Lots of distributed command and control structures in human and other animal bodies
Can easily deal with component failure
Components are relatively weak but perform complex tasks for little energy
Fairly regular highlevel structure but low level is almost random (its what makes us unique)

e) Storage

In neurons the weights of the inputs change over time (and stay that way) and that is what holds information within the system

f) Design

Bounded Asynchronousity
Non-deterministic in nature
Theory of self-organising computers
    i.e. ones which almost program themselves
    Distributed finite state machines

2 Establishing Networks

Based on algorithms used in BIMPA project

Assumes there is one node connected to a laptop via ethernet which acts as a root

a) Establishing working communications channels

Nodes each with at least 1 communications channels

Want to establish which channels work and which don't

Send symbols down each channel

Algorithm:

- Received R on a channel
  - Send A back, label channel A
  - Send R on all other ports labelling them R
- Received A on a channel
  - Label channel as A

Outcome:

- All working channels are labelled A, else R

A flooding algorithm but with no real end. May require a timeout to stop but could be ran periodically to keep track of network changes

b) Establishing a tree (simple)

Assumes nodes are arranged in a rough tree to start with i.e. no loops

Only uses working communications channels

Two symbols used: (F)oward and (B)ackwards

case F:

- label channel IN (points to parent)
- label all other channels OUT and set a counter to *n* the number of channels
- send F to all OUT channels
- if counter = 0 send B back on IN

case B:

- decrement counter
- if counter = 0 then all child notes are acknowledged so send B back on IN

This is simple an assumes there is a tree like structure already existing it just tells a node to label its children and parent appropriately

Normally an external connection to a particular node which becomes the root

c) Establishing a tree from graph (advanced)

There can be loops

Only uses working communications channels

Uses 3 symbols : (F)orward, (B)ack, e(X)clude

X used to stop loops by ignoring extra nodes claiming they are parents

May not arrange nodes in the most efficient tree

case F:

- if no pin labelled IN
  - label incoming pin IN

        - wait = 0
        - if no unlabelled pins
            - send B to IN
        - do while there are unlabelled pins
            - send F down next channel
            - label pin OUT
            - wait++
    - else: (loop case)
        - label incoming pin X
        - send X to incoming

case B:

- wait--
- if wait = 0
    - send B to pin IN

case X:

- label incoming port X
- wait--
- if wait = 0
    - send B to pin IN

This is a breath-first-tree-traversal algorithm but there is a faster depth first one available
On a given node: parent labelled IN, children labelled OUT and X's are ignored

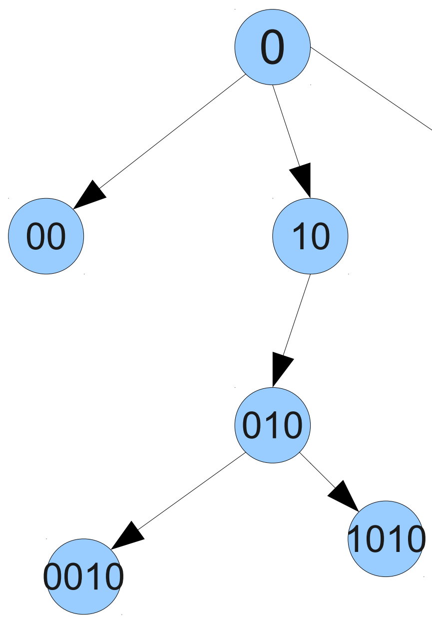
d) Node labelling

Once we have a tree, one might want to label the nodes
Useful to label them uniquely, with a level/depth number and with a path back to the root

Algorithm:

- label root node 0
- on each node label OUT channels 0 to *n*
- On each level label the node:
    - [comms chan number][parent label]

Produces a tree labelled like so:



e) Point to Point Tables

Aka routing

Want to know which route to take to reach other nodes

Each node keeps a table of all other nodeIDs and the route to take to reach them

On start up they broadcast a packet containing a label or themselves on every channel

When they receive a packet they check the label:

    if its their label ignore it

    else update their routing label mapping LABEL to the pin received on

This is essentially RIP

However, the idea is extended to use barriers (synchronisation points) to tell when all tables have been labelled

3 Distributed Algorithms

a) Fireflies

A method of synchronising different nodes onto the same frequency/clock

Based on fly flies in nature who begin flashing randomly but the become in sync and flash at the same time

"One of the easiest algorithm to explain this behaviour goes like this: You have a value that holds the power to flash. As time passes this power will slightly raise. If the power reaches a certain level, the firefly flashes and the power is consumed. The rate at which the power raises is nearly the same for all fireflies. So they have the same frequency but not the same point in time to flash.

While slowly charging with power the firefly is able to detect a flash of another firefly nearby. It adds then a higher value to its power value. Some kind of power boost, if you wish. That means the next flash will occur earlier than the one before. And next one even earlier, until these two are flashing exactly at the same point in time and with the same speed.

“ **http://www.instructables.com/id/Synchronizing-Fireflies/**

More information:
http://www.rlocman.ru/i/File/2007/10/24/2006_WSL_Firefly_Synchronization_Ad_Hoc_Networks.pdf

Requires a totally required network and adjusting for time delays (i.e. transfer times)

But completely ad hoc with no central control or hierarchy

b) Peterson's Mutual Exclusion Algorithm

Solution for concurrent programming to allow for mutually exclusive access to a shared resource

Original solution was for 2 processes but can be generalized for more processes

```
#Global Variables
flag[0]; flag[1]; turn
#P0
flag[0] = 1;
turn  = 1;
while (flag[1] == 1 && turn = 1) {}
// critical section
flag[0] = 0;
#P1
flag[1] = 1;
turn  = 0;
while(flag[0] == 1 &&  turn = 0) {}
// critical section
flag[1] = 0
```

flag indicates that a process wishes to enter its critical section

turn indicates which process is in the critical section

4 Traditional Introduction

a) Definition of Distributed Computing

Autonomous Processors communicating over a network

Characteristics:

- different types of devices
- no shared memory
- autonomous devices
- geographical separation

b) Components

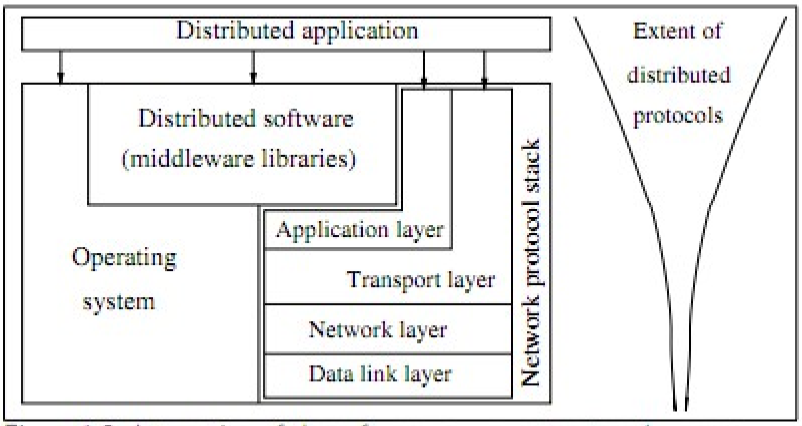


Figure 1.2: Interaction of the software components at each process.

Distributed Application handling the higher-level operations of the program

Middle ware typically a software library used to abstract away from the harder aspects of distributed computing (i.e. the communication)

c) Motivation

Resource sharing and accessing remote resources

- making better use of under-used resources through load handling
- distribute use of resources i.e. letting different scientists control satellites from their labs

Increased performance/cost ratio

Reliability, availability, integrity and fault tolerance

Scalability, modularity and incremental expandability

- i.e. the internet where many devices can be added without a performance loss

d) Parallel Systems

Can follow different models and types

Multiprocessor Systems

- direct access to shared memory
- UMA model
- Interconnection network or bus linking components
- Routing function for communicating between nodes
- examples: Omega and Butterfly
- dedicated memory nodes

Multicomputer parallel systems (i.e. clusters)

- no direct access to shared memory
- NUMA model
- bus, ring, mesh and/or hypercube topologies
- were big in the late 90s but died off, now only used really in clusters

Array Processor

- collocated and tightly coupled
- common system clock

Used in Digital System Programming

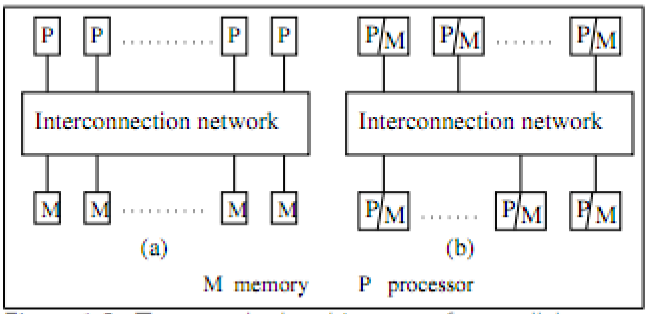


Figure 1.3: Two standard architectures for parallel systems. (a) Uniform memory access (UMA) multiprocessor system. (b) Non-uniform memory access (NUMA) multiprocessor. In both architectures, the processors may locally cache data from memory.

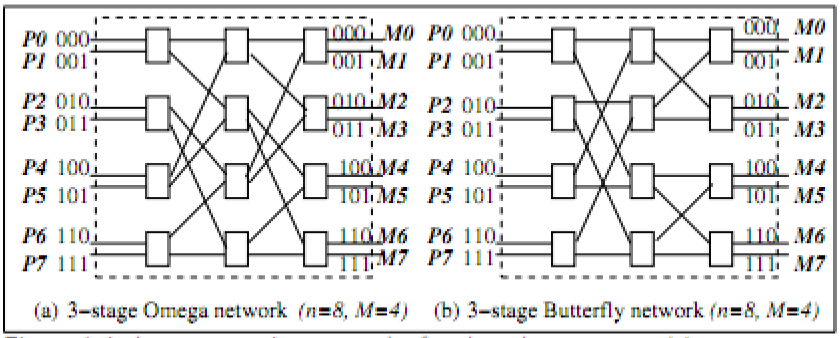


Figure 1.4: Interconnection networks for shared memory multiprocessor systems. (a) Omega network (b) Butterfly network.

e) Omega Network

N processors and memory banks

log n stages with *n/2* swtiches each 2x2

interconnection function: output *I* of a stage connected to input *j* of next stage where:

$$j = 2i \quad \text{for} \quad 0 \le i \le \frac{n}{2} - 1$$

$$j = 2i + 1 - n \quad \text{for} \quad \frac{n}{2} \le i \le n - 1$$

Routing function: in any stage *s* at any switch to route to destination *j*

if *s + 1th* MSB of *j* == 0 then route on upper wire

else route on lower wire

f) Flynn's Taxonomy

Different types of computing operation

SISD : Single Instruction Stream, Single Data Stream

traditional computer

SIMD : Single Instruction Stream, Multiple Data Stream

scientific applications

applications on large arrays

vector processors

Performs the same operation on different data at the same time

GPU?

MISD : Multiple Instruction Stream, Single Data Stream

Data analysis?

MIMD: Multiple Instruction Stream, Multiple Data Stream

Distributed systems and the vast majority of parallel systems

Can be hard to program

g) Terminology

Coupling : Interdependency/binding among modules

Parallelism: *T(1)/T(N)* ratio showing the speed up of using the parellel system v a single computer

Concurrency of a program: measures productive CPU time v waiting for synchronous operations

h) Primitives

Synchronous

- Handshake between sender and receiver
- Send completes when Receive completes
- Can be blocking because of this
- Can cause deadlock when all nodes want to send at the same time and none are listening

Asynchronous

- Send only
- Control returns to process when data copied out of user-specified buffer
- Actual sending typically handled by middleware

Blocking

- Control returns after primitive operation completes
  - for sync send: after receiver has got the data
  - for async send : after data copied out of buffer
- Easier to debug/work out control flow but sub optimal

Nonblocking

- Control returns to process immediately after invocation
  - async send : after request made to copy data out of buffer
- Invoker has no idea whether the operation is successful or not
- Tend to use handles for the data so that the tasks can be done 'in the background' allowing for better use of CPU time as the handle can be checked later on

5 Amdahl and Brent Laws

a) Amdahl Law

Assuming a program that runs on a single processor has a part that can be run in parallel and a part that must be sequential (i.e. vector add and file access) so the time to execute on a single processor is:

$$T_1 = T_s + T_p$$

Assume that the parallel part scales *perfectly* on *p* processors the time taken will be

$$T_P = T_s + \frac{T_p}{P}$$

This means that the speed up factor is

$$S_P = \frac{T_1}{T_P} = \frac{T_s + T_p}{T_s + \frac{T_p}{P}}$$

In the limit i.e. with an infinite number of processors this means the maximum speed up possible is:

$$S_\infty = 1 + \frac{T_p}{T_s}$$

b) PRAM

c) Parallel Reduction

Assume an algorithm for a problem size *N* which is a power of 2

Requires $\log_2 N$ steps to complete on $\frac{N}{2}$ processors and costs $\log_2 \frac{N}{2}$ to spawn them

Overall complexity $\Theta(\log_2 N)$

d) Cost of an algorithm

The order of the number of processors times the number time complexity
In the case previously this would be $\Theta(N \log_2 N)$
The optimal parallel algorithm is will have the same or better cost as the sequential algorithm
From the previous example the optimal parallel solution is $\Theta(N)$ so the best parallel solution will use $\frac{N}{\log_2 N}$ processors
Brent's theorem shows that this is possible

e) Brent's Theorem

If a parallel algorithm *A* can perform *M* operations in time *T* then *P* processors can execute the algorithm in time $T + \frac{(M-T)}{P}$

Let s_i be the number of operations performed between $0 \le i \le T$ to that $M = \sum_{i=0}^{T} s_i$

$$\sum_{i=0}^{T} \frac{s_i}{P} \le \sum_{i=0}^{T} \frac{s_i + P - 1}{P}$$

$$.. \le \sum_{i=0}^{T} \frac{P}{P} + \sum_{i=0}^{T} \frac{S_i - 1}{P}$$

$$.. \le T + \frac{M - T}{P}$$

Using the example from Parallel Reduction there are $\log_2 N$ time steps, $N-1$ operations and at best $\frac{N}{\log_2 N}$ processors. Applying this to the formula we get:

$$T = \log_2 N$$

$$M = N - 1$$

$$P = \frac{N}{\log_2 N}$$

$T_P = \Theta(\log_2 N)$ i.e. using a parallel algorithm the problem can be solved in log N time

6 Paradigms of Parallel Programs

a) Splitting Programs

Where and what to split?

Splitting too much could overload the processors

Splitting too little could fragment the work too much leading to idle processors and increased communication costs

Why split?

Better speed

Because there is too much memory used for once machine to handle

To increase reliability

b) Parametric Parallelism

Parameters of the problem are distributed

Typically single master multiple slave configuration used

master controls work

each slave gets a small part of the problem

All slaves get ALL the information (shared memory?) but get instructions about what to do

Example: Ray Tracing

- master controls rendering
- slaves to paint a row of pixels
- shared scene graph
- can give way of easily load balancing

c) Contributing Factors

Lots of different things can affect the complexity of the parallel program but the following are quiet important:

Number of slaves

Amount of work

Communications overhead

d) Ray Tracing Complexity

W the number of lines

s the number of slaves

tline the time to complete a line of pixels

tsend time to send and receive messages

$$T_s = \lceil \frac{w}{s} \rceil t_{line} + w(t_{send})$$

Could increase speed by getting slaves to process more lines at once

e) Algorithmic Parallelism

Code is distributed to form a pipeline of operations

Data fed through a sequence of operations which performs operations

Complexity determined by slowest component (limiting factor)

f) Geometric Parallelism

Data distributed over processors

Each 'node' doesn't necessarily have all the data

Used when there is more memory required than is possible for 1 machine to handle

i.e. matrix maths on large scale

Need to minimise the interface between sets of data in order to optimise

g) Summary

Many different paradigms and architectures can be used

No one perfect solution

One needs to work out the reasons for paralleling the system and what could constrain its operation the most

7 Wireless Sensor Networks

a) Issues

Lots of intelligent sensors in the network to detect something

Might have to be left for a long time in the environment so they need to be energy efficient

Systems need to be able to scale easily

Information costs energy to send some need to optimise algorithms to extend the battery life time

b) Information

Sensor can save power by only submitting surprising data to the base station i.e. messages with high amounts of information

According to Shannon's theory the amount of information given when a symbol *j* is received is:

$I(j)=-\log_N(P_j)$ where *N* is the total number of symbols and *Pj* is the probability of receiving that symbol

Shannon Capacity states the theoretical maximum for the amount of information that can be received on a channel

$C=B\log_2(1+\frac{S}{N})$ where *C* is the symbol-per-second capacity, *B* is the bandwidth, *S* is the signal power (in decibels) and *N* is the noise power in decibels

c) Radio Propogation

In a perfect channel the amount of power received is given by the following formular

$P_r=P_t(\frac{c}{fd})^2$ where Pr is the power received, Pt is the power transmitted, c is the speed of light

in the volume, f is the frequency of the carrier and d is the distance
This means that more power is needed to transmit for longer distances and for higher frequencies hence why FM radio has a lower range (it uses higher frequencies)

The Pr value sets an upper limit for the capacity of the wireless channel as it affects the *S* value in the capacity formula.

d) Multiple Access

Time Division : Each interlocking node gets *x* amount of time in sequence

Space Division : Spread nodes physically so that their signals don't overlap

Frequency Division : divide bandwidth into channels

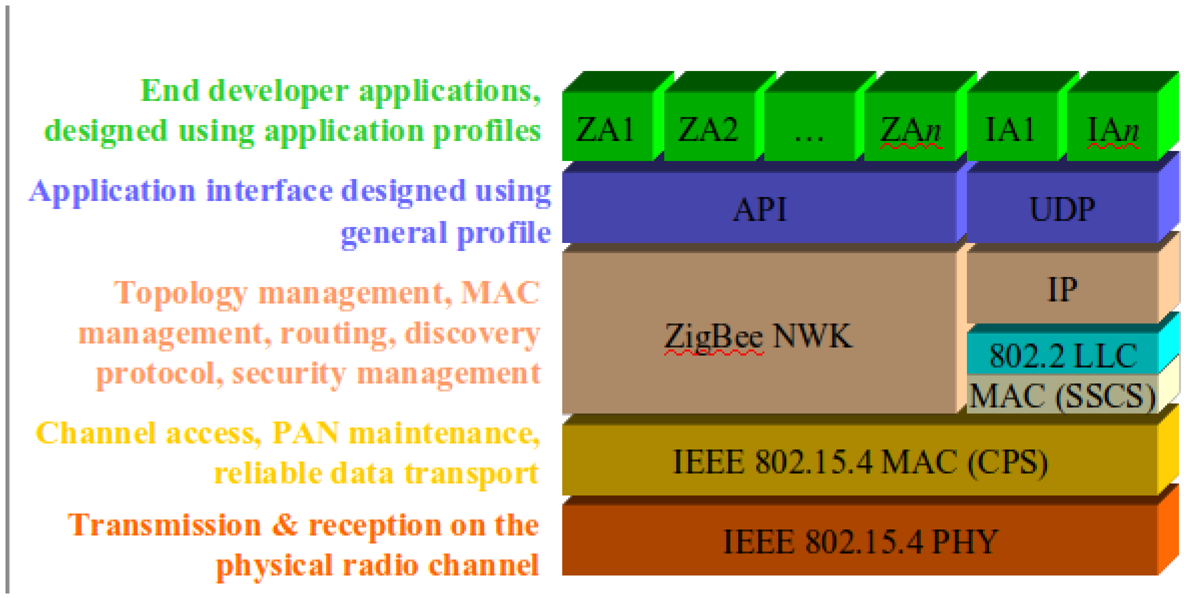
Code Division : see networks

8 Zigbee

Standardised protocol for remote agents to communicate with each other

Low data rates, low range but also low power usage

Quick pairing ability



Full protocol stack < 32k but coordinator nodes need more RAM as they remember state

Adhoc, on-demand, distance vector routing

On demand : routes not maintained, they are created as needed

9 Cloud Computing

a) More Power

Sometimes there is not enough memory or computing power locally to complete tasks in a reasonable time

Want to be able to easily access a more capable resource

b) Grid & Cloud

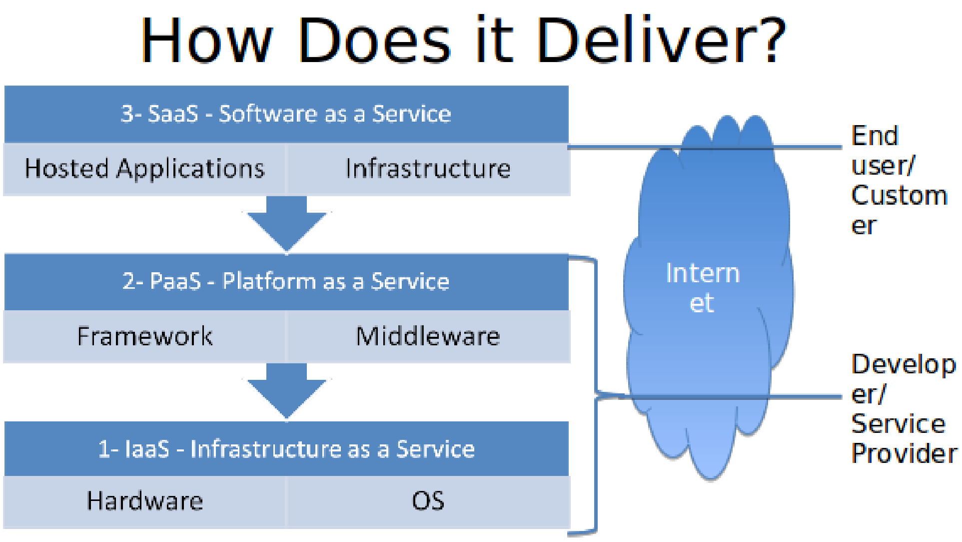
Utility computing
Like water & electricity but for computing power
Grid : more structured & complex but powerful
Cloud : a lot of abstractions, don't need to worry about low level details
Allows for scaling up from the Personal Computer to world wide networks

c) What is cloud computing

Depends on provider's definition
Key Characteristics:

- elasticity : adapts demand dynamically
- abstraction : automatically set-up & run
- self-service based provision
- billed for what you use
- emerging standards for cloud computing (OCCI)

Lots of abstractions to hide the complexities
Typically divided into 3 layers



d) IaaS

Infrastructure as a Service
You get access to virtualised hardware
You're responsible for OS. Middleware, runtime, data and applications
Its like having your own machine
Example: Amazon EC2

e) PaaS

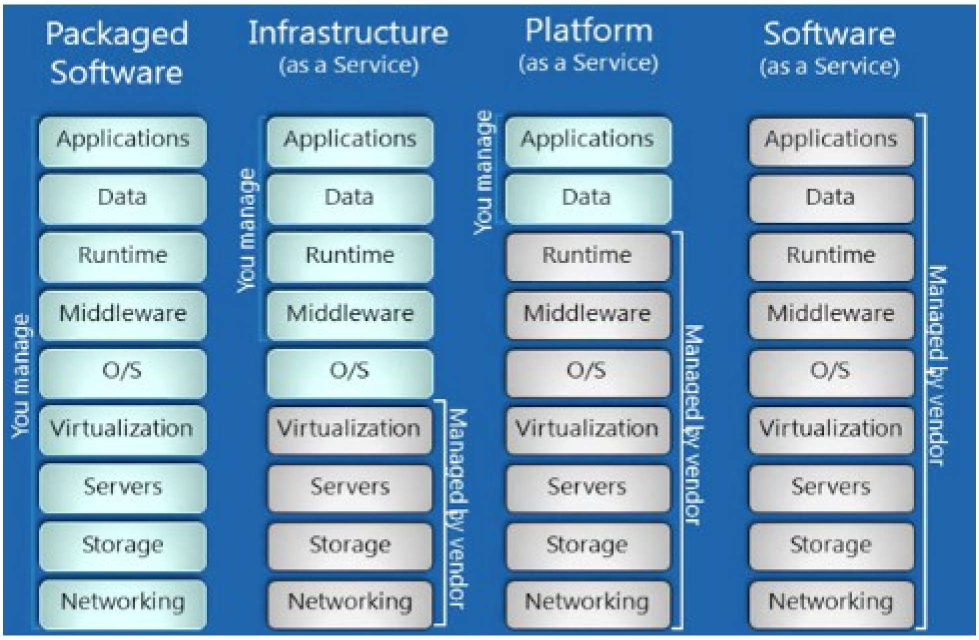
Platform as a Service
Develop applications on top of an existing infrastructure service
Includes more abstractions from the OS

User still responsible for managing data
Examples : MS Azure and Google App Engine

f) Software as a Service

Examples : facebook, google docs, google sites, google mail
The consumer end of the technologies

g) Layers Summary



h) Cloud Standards

Being developed but meeting resistance from proprietary vendors
Should allow services to be easily transferred across providers
OCCI : Open Cloud Computing Interface

10 Work Flow

a) What is it?

Captures processes and interactions between their elements
Typically the most important aspects are the inputs & outputs of the processes
To understand a process you must also understand its data

b) Historical view

Manufacturing production lines

  Physical objects

  want to improve efficiency

NASA space flights

  mission critical

extremely complex
both hardware and software
introduced abstract objects and information to be processed

c) Small Scale

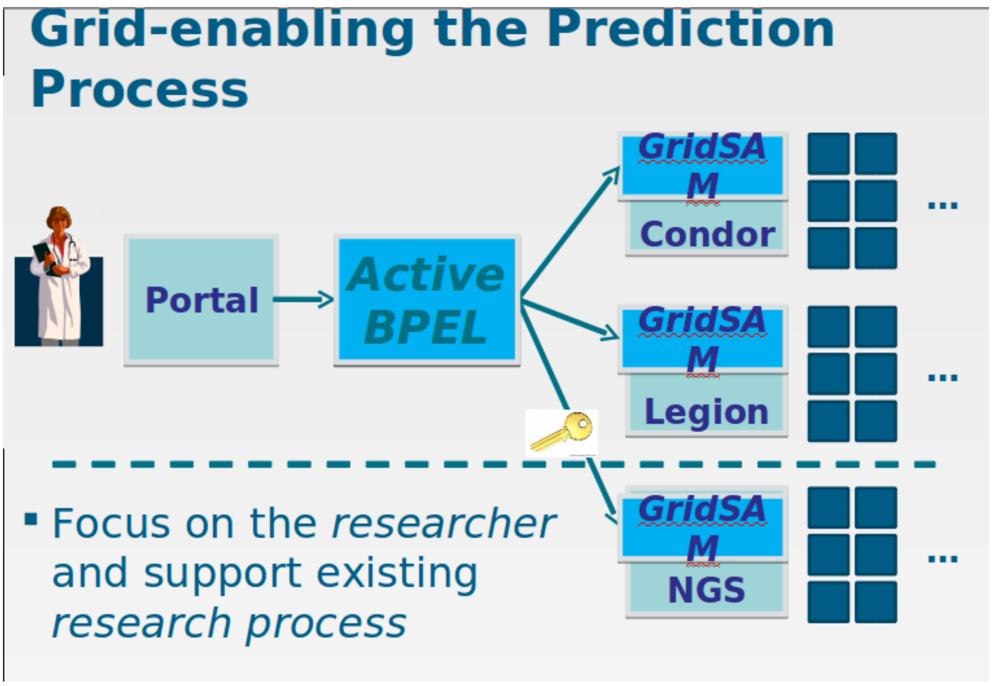
Probably already do this
Creating/capturing data between processes and their interconnections
i.e. how one uses HTTP requests to read webpage
Workflow takes this idea to a higher level of abstraction

d) BPEL

Business Process Execution Language
Want to be able to describe processes at a high level
BPEL is a webservice *orchestration* not *choreography*
orchestration gives centralised control of the system
choreography gives rules and protocols for interaction between sub systems
Developed between IBM, Microsoft and other big companies
Both open-source and commercial engine support
Key Design Goals:
Business processes interact with external web services using WSDL 1.1
Business processes defined in XML
Can build on existing systems
Allow for hierarchical and graph based design to avoid fragmentation
Build on existing web standards and encourage modularity

e) BPEL Case Study

Workflow for computational chemistry
Want to find the crystal structure of molecules
This was a well understood but slow process
Wanted to use GRID technology to speed this up



This achieved a massive speed-up from one calculation being performed in a week to many hundreds being performed in an hour

Gave between utilisation of resources

f) Taverna

Emerged from myGrid project

Original idea was to support the bioinformatics research area

Everything is distributed

Heterogeneous data sources

Uses the SCFUL workflow language

Open source and free release from Manchester University

Consists of:

- work bench : designer, executor and monitor
- enaction engine : does the execution work
- server : expose a workflow as a service
- commandline tools

g) Taverna v BPEL

Taverna is dataflow-orientated

BPEL is processed-orientated

Taverna is more fitting for science applications

BPEL is much more complex

Taverna is easier to extend

h) SCUFL

Simple Conceptual Unified Flow Language

Is used in the myGrid project. It defines a high-level workflow description language that allows the user to map a conceptual task to a single entity with a minimal amount of implementation specific information (the translation to lower-level entities is done by the underlying system called the IT Innovation Enactment Engine). Three main entities are defined: processors, data-links and coordinations. Unfortunately it is not possible to specify user-defined constraints neither to processors nor to data-links.

There is a basic exception handling mechnism included. It supports retry of invocation with configurable timeout and number of retries, and user-defined alternatives for processors failing constantly.

Consists of:

Set of inputs and outputs

Set of processors

- each processor presents an atomic step and a logical service
- has a set of input and output ports

Set of data links

- links data sources to destinations

Set of control links

specifying order