Bringing Programming Languages up to Date

John Florentin & Geoff Sharman

BCS APSG

March 2009

Towards New Languages

- ***High level languages*** *are the prime tool for creating system and application software*
- *Their design can make a* ***major difference*** *to the cost, difficulty, and likelihood of success in implementing business applications*
- *Computing technology has* ***changed*** *radically since the first HLL's were introduced, and so has the business environment*
- *But the design of languages* ***has not improved*** *sufficiently nor kept up with these changes*

What we'll Discuss

- *What is the problem?*
- *A brief history of programming languages*
- *The 21st century computing environment & why a new model is needed*
- *Closing thoughts*

What's the Problem?

- *Despite many advances, current programming languages are often non-intuitive and still produce programs that are:*
  - *Complex (hard to read & understand)*
  - *Error-prone (especially w.r.t. concurrency)*
  - *Brittle (hard to change)*
  - *Unmanageable (hard to recompile, refactor)*

What's the Problem? (ii)

- ***Modern** computing environments have:*
  - *Many different types of computing device with a wide range of form factors*
  - *Worldwide high speed networks*
  - *Huge volumes of data*
  - *Applications with many concurrent users*
- *But languages **do not enable programmers** to exploit these environments easily*
  - *Support only via libraries and/or subsystems*
  - *Many applications fail to scale*

A Brief History of Programming Languages

Machine Languages

- *Widely used in 1950s - 1960s*
- *Programmer aware of hardware architecture:*
  - *Operation codes for program steps*
  - *Addressing scheme*
  - *Real memory addresses for data*
  - *Character encoding*
- *Programs limited to a specific architecture and physical realisation*
  - *Not portable beyond single machine*

Assembler Languages

- *Widely used in 1960s – 1970s*
- *Programmer less aware of hardware architecture:*
  - *Symbolic names for instructions*
  - *Symbolic names for data items*
  - *Multi-pass assembly for resolution of names*
  - *Assistance with addressing scheme*
- *Programs limited to specific architecture*
  - *Portable between different physical machines*

High Level Languages

- *Introduced with FORTRAN (1953), LISP (1958), COBOL (1959) and ALGOL (1960)*
- *Programmer aware of an "**abstract machine**":*
  - *Reserved names for program statements and control structures, e.g. loops*
  - *Symbolic names for data items and data structures, e.g. arrays, records*
  - *Basic input/output facilities*
  - *Multi-pass compilation, large/virtual memory*
- *Programs portable between h/w architectures*
  - *Separate compilation for modular programs*

The HLL Abstract Machine

- *HLL programs run as batch jobs:*
  - *Programs typically processed sequential data files, which were owned for the duration of the job*
    - *e.g. Master file/transaction file update*
- *Operating system handles resource allocation, job scheduling, print spooling and virtual memory management*
  - *Consolidated as the "**process abstraction**"*
  - *Normally with a single thread of execution*
- *Many programs still in use (aka "legacy")*

Transaction/Data Languages

- *From 1960s & 1970s online applications became common, e.g. airline reservation systems, retail banking, point of sale:*
  - *Networks of terminals attached to a central server*
  - *Fast response required for customer requests*
  - *Direct access to individual records rather than files*
- *Supported by the development of database managers and transaction processing monitors*
  - *Functionality accessed via "sublanguages", e.g. CICS Command Level (1974) and SQL(1978)*
  - *Embedded in subset of HLL*

The OLTP Abstract Machine

- *Online programs are initiated via a message and produce a reply message within seconds*
- *The TP monitor and DB manager provide:*
  - *rapid dispatching via pre-allocated resources,*
  - *efficient concurrent execution*
  - *distribution over multiple processors*
  - *record level data access and other services*
- *Consolidated as the "**transaction abstraction**"*
  - *Individual transactions usually single threaded*
- *OLTP supports **most** consumer transactions today*

The C Language

- *C (1972) was developed as the systems programming language for Unix*
  - *Derived from ALGOL via BCPL*
  - *Minimal language but includes key HLL features*
  - *Efficient execution, replacing assembler language*
  - *Also adopted for applications*
  - *Programs somewhat portable between Unix systems*
- *Still ranked as the "**most popular**" programming language*

The C Abstract Machine

- *C uses an HLL abstract machine for interactive programs, rather than batch*
- *Its de facto abstract machine contains HLL plus Unix features:*
  - *User interaction*
  - *Hierarchical file system*
  - *Network communication, client/server access*
  - *Threading*
- *Adapted for PCs and many subsequent devices*

Object Oriented Languages

- *Smalltalk (1970s) was developed as a language for robust programming*
  - *Derived from ALGOL via Simula 67*
  - *Models an application domain via classes & objects*
  - *Provides encapsulation, polymorphism, inheritance*
- *OO features later incorporated in C++, Java etc.*
  - *From 1990s, used to implement graphical user interfaces on personal workstations*
  - *Programs somewhat portable*
  - *Java, C++ also ranked as “very popular” languages*

The OO Abstract Machine

- *OO uses a modified form of the C abstract machine*
  - *C++ provides upwards compatibility from C*
  - *Supported by Standard Template Library for extended data structures and algorithms*
  - *Java abstracts as the **Java Virtual Machine** (JVM)*
  - *JSRs define extensive libraries for GUI, forms, communications, database access, etc.*
- *Fully encapsulated memory management is one of its important advantages*

Functional Languages

- *LISP was developed as language based on Lambda Calculus*
  - *Data structures based on lists and trees*
  - *Operations on lists, no loops*
  - *Dynamic typing*
  - *Modifiable program source code*
- *Focus on concise and provable programs*
  - *Later FLs include ML, Miranda, Haskell*
  - *Single assignment or no assignment*
  - *Influential but never mainstream*

The FL Abstract Machine

- *Because FLs are primarily designed for mathematical computation, they are usually employed in a single user context*
  - *e.g. complex analysis, derivatives trading*
- *Their de facto abstract machine is very similar to that of HLL programs*
  - *Attempt to abstract from procedural semantic*
  - *Operating system process with few extra facilities*
  - *Little recognition of interactive user interfaces, networking,databases etc.*

The 21st Century Programming Environment
Why a New Model is Needed

Pervasive Computing

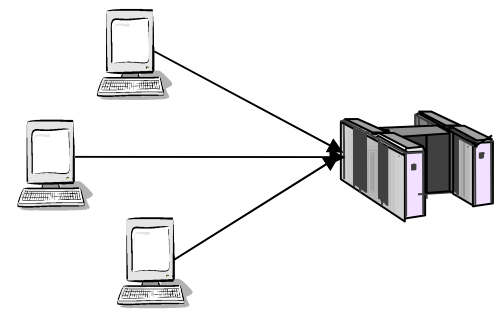
- *Steadily decreasing cost of computing has enabled computing systems with a wide range of form factors:*
  - *RFID tags, smart cards*
  - *Embedded domestic systems*
  - *Handheld systems (mobile phones etc.)*
  - *Personal workstations*
  - *Enterprise servers*
  - *Embedded commercial/industrial systems*
  - *Warehouse-sized computers*
- *Average home has < 100 digital devices*

Standalone Applications

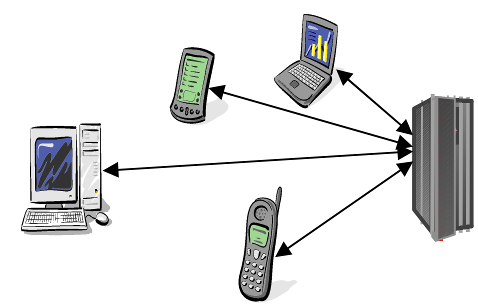
- *Many computing devices support an operating system and a few pre-loaded applications with a simple user interface*
- *For this class of devices:*
  - *C programming is the best fit. It's likely to be chosen for cost reasons – especially at the low end*
- *These applications will probably have obvious bugs and vulnerabilities*
  - *In standalone devices, it may not matter (much)*

Networked Configurations

- *1980s*
  - *~ 1000 active terminals*
  - *9.6 kbps network*
  - *1 MIPS processor*
  - *10 megabytes memory*



- *2000s*
  - *~ 100k – 1 million active devices*
  - *100 mbps Ethernet*
  - *Broadband Internet*
  - *1 gigahertz processor(s)*
  - *1 terabyte memory*
  - *Most servers are clusters with shared data*



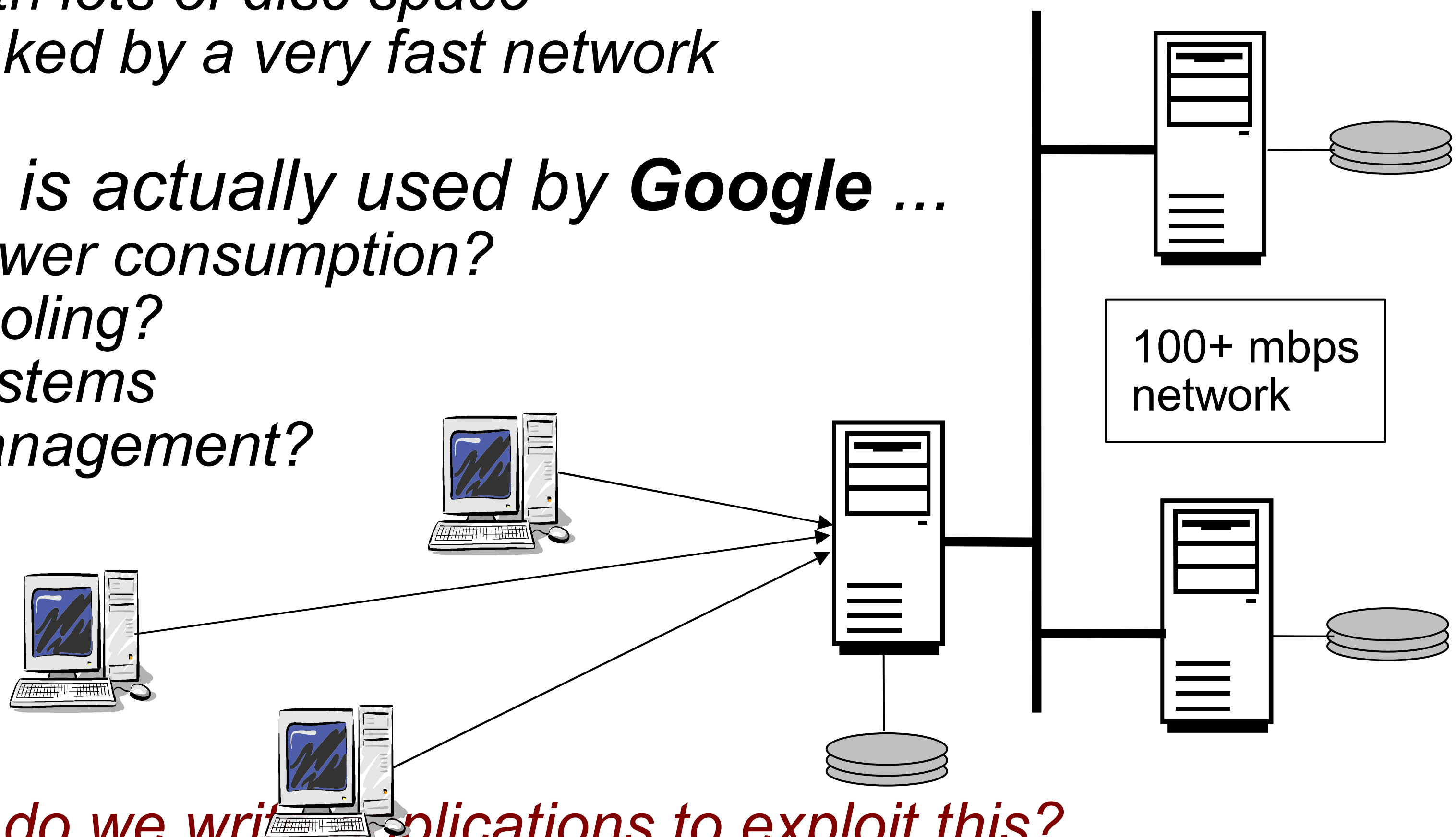
- *Over this period:*
  - *Processors improved by a factor of 1000 (Moore's Law)*
  - *Memory improved by a factor of 5,000*
  - *Networks improved by a factor of 10,000*
- *Average person probably does 10 - 100 transactions/day*

Applications for User Devices

- ***Rapid growth in end user devices***:
  - *Personal workstations*
    - *$10 laptop project in India*
  - *Handheld systems*
    - *Latest devices offer voice, music, image, text, etc.*
- ***Established programming techniques***:
  - *OO programming well understood for GUI*
  - *But **complex user interfaces** limit acceptance*
- ***Web browser may become the base for new applications***: *[how will programming change?]*
  - *e.g. Google Chrome*

Warehouse Sized Servers ...

- *Build a cluster of cheap machines*
  - *Thousands of custom designed PC boards*
  - *Running Linux*
  - *With lots of disc space*
  - *Linked by a very fast network*
- *This is actually used by* ***Google*** *...*
  - *Power consumption?*
  - *Cooling?*
  - *Systems Management?*



- *How do we write applications to exploit this?*

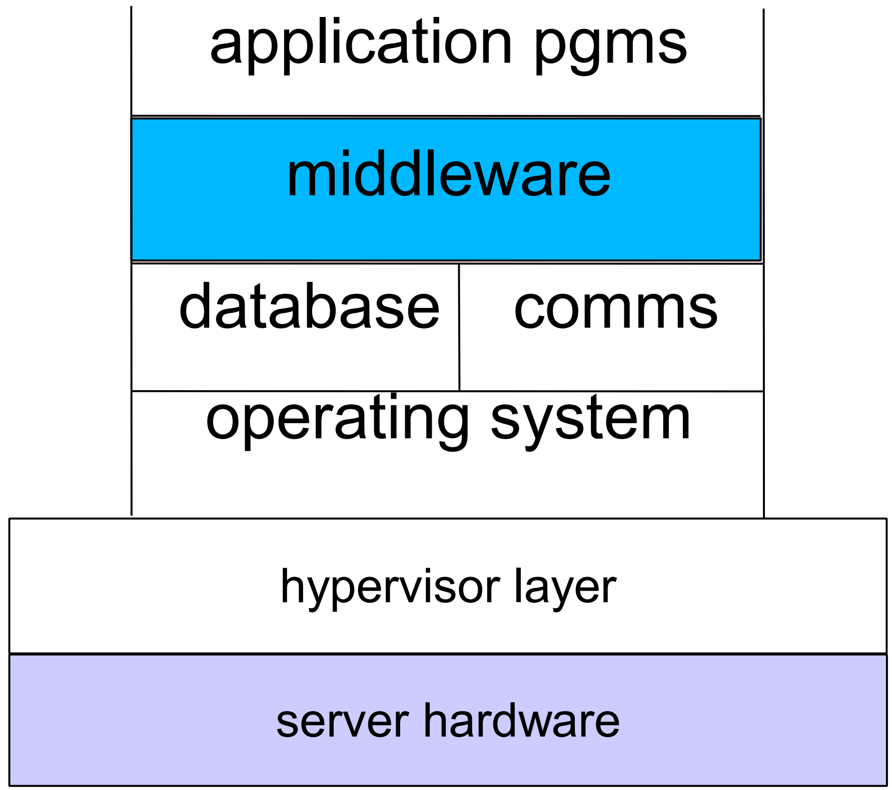
Applications for Servers

- ***Internet use growing rapidly due to widespread adoption of broadband**:*
  - *Web servers now support large populations of concurrently active end users & many new applications*
  - *OLTP model suited to this workload and already adopted by web servers for static information*
  - *"Cloud computing" expected to be the future*
- ***But applications still have many challenges**:*
  - *Achieving scalability with large server clusters & networks*
  - *Accessing/processing large shared databases*
  - *Dealing with systems management issues*
  - *Achieving very low error rates*

Problem Statement - Refined

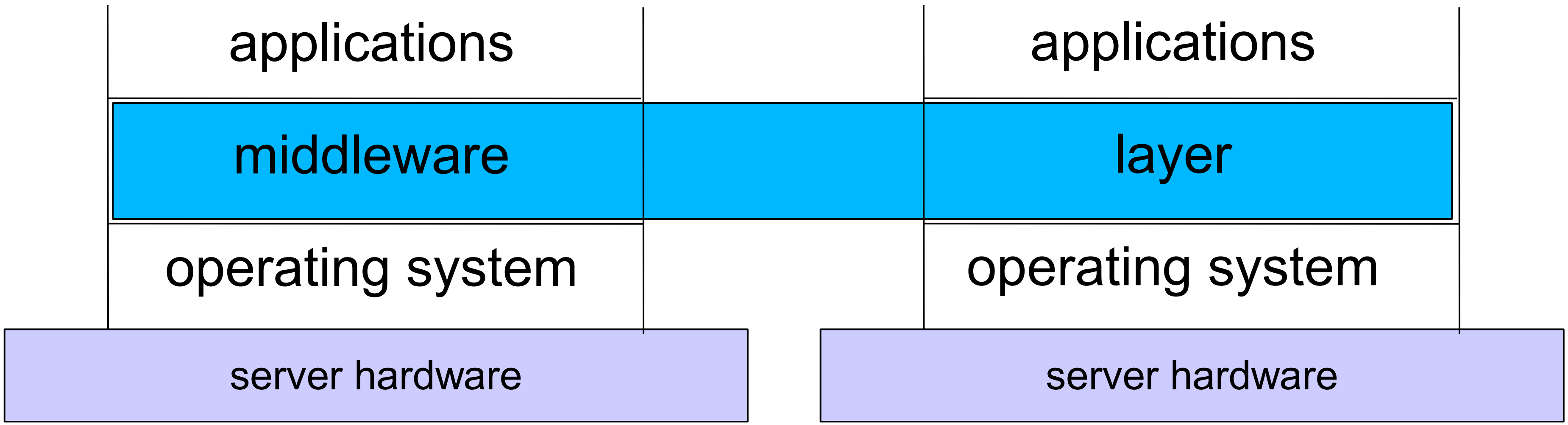
- *We don't yet have an effective model for **server applications** in the Internet era. We need to:*
  - *Exploit concurrency and parallelism, free of infrastructure concerns*
  - *Enable change as rapidly as needed for business reasons*
  - *Achieve a high degree of correctness without extensive testing*
- *We need a **new abstract machine**, based on the OLTP model, adding the best of OO and FL*
  - *Automatically provides concurrency*
  - *With programmable functionality c.f. AOP/AOSD*
  - *Basis for new languages which exploit it*
  - *"Container" concept in Java EE provides an example*

Abstract Machine = Middleware



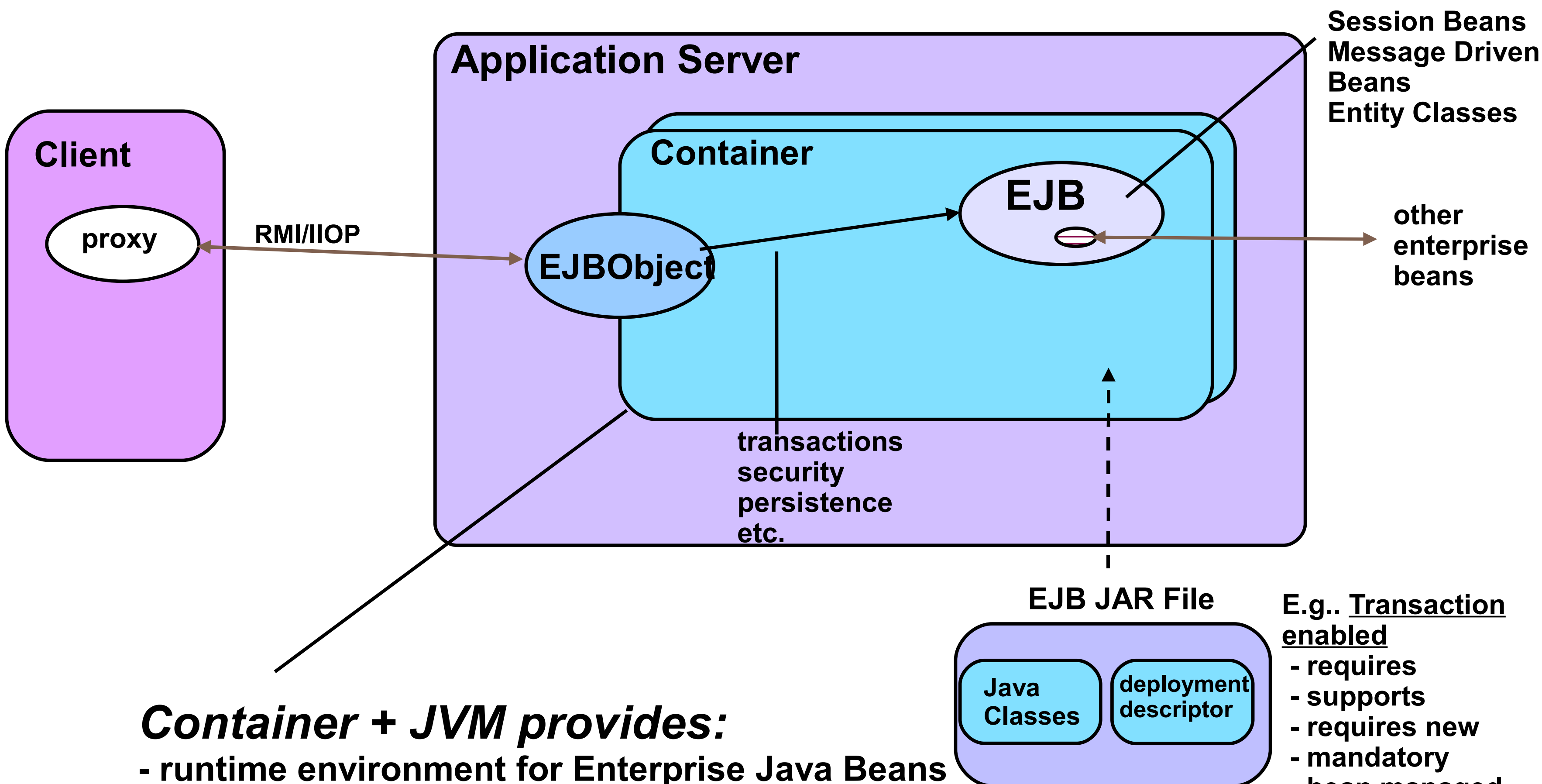
- *Middleware Layer exploits **database and comms layer** as well as operating system*
- *Provides **virtualisation**: applications **portable** if only middleware services used*

Middleware may Span Clusters



- *Middleware layer:*
  - *May exist on **multiple** physical systems, different operating systems or hardware architecture:*
  - *A **multi-system** virtual environment*
  - *May have a **longer lifetime** than any one system*
  - *Provides a **higher level of abstraction** than OS – to make application programming easier*

Example: Java EE Runtime



Container + JVM provides:

- runtime environment for Enterprise Java Beans
- creation & destruction of EJBs (lifecycle)
- additional services mapped to local interfaces
- common execution semantic for multiple environments

But what if we need to add to/change its functionality?

Towards a New Model

- *The new abstract machine should provide:*
  - ***Better modularity**, enabling flexibility to respond to business change [cf. "cell" concept in Erasmus]*
  - ***Parallelism**, enabling faster response to user requests [cf. OCCAM, Erasmus, etc.]*
  - ***Loose coupling**, enabling asynchronous operation [cf. MQSeries messaging, JMS]*
  - ***Better integration with database managers,** avoiding the "impedance mismatch" problem [cf. Functional database/language unification]*
  - ***Robust scripting**, enabling us to represent long lived business processes [cf. BPEL, CICS BTS]*

Closing Thoughts

Future Work

- *We **haven't** talked about:*
  - *Declarative vs. imperative languages*
  - *Dynamic vs. static type checking*
  - *Bytecode interpretation, JIT compilation*
  - *Managing multiple versions of modules/programs*
  - *The role of metalanguages*
  - *Features for data mining, grid computing, etc.*
- *All these things should be considered by any language designed for a new abstract machine*

Expanded Role of the IDE

- *IDEs (e.g. Eclipse) are widely used to enable interactive editing, syntax checking, compilation, debug etc.*
- *But should be **extended** to support:*
  - *Alternative syntactic forms*
  - *Subassembly build based on new modular forms*
  - *Debug with simulated data, in simulated time*
  - *Performance modelling*
  - *Etc.*