Larger-scale design

Small- and medium-scale are the province of small teams of programmers

Larger scales need larger teams

- Usually cross responsibilities within an organisation
- Typically cross organisations
- In the extreme involve the entire community, or at least a sizeable fraction of it

In this lecture we'll trace the concerns of extensibility, flexibility and change management through building applications, systems, enterprise-level systems and up to the global Internet

Scalability

How do the issues change?

- The same issues recur at different scales – always worry about change management, re-use *etc*
- But the relative importance of the issues changes as systems grow

At smaller scales we deal with issues which affect a relatively small number of people

- Mainly affect the project team
- Affect others by code re-use

As scale increases, the number of people affected by decisions – and who need to agree to them – increases

- Often non-linearly

Scalability levels

Global

Enterprises

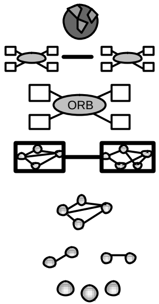
Systems

Applications (sub-systems)

Macro (frameworks)

Micro

Objects



Issues – small and medium scales

Object level

- Getting the algorithms and data structures right
- Worry about method signatures and visibility

Micro level

- Combinations of a few objects

Macro level - frameworks

- Combinations of objects covering fairly large abstractions (*e.g.* Java's AWT/Swing frameworks, Microsoft's MFC)

In all these we're basically concerned with local programming problems – what to do and how to do it

Issues – larger scales

Application level – sub-systems

- The external view of a software system
- Emphasis on a “good fit” to specific task

System level

- Integrate different applications
- Innovation gets more risky, changes tend to propagate across application boundaries

Enterprise level

- An organisation’s entire IT support
- Changes can directly affect the organisation’s economics

Global level

- Co-operation between organisations
- Interoperability, policy and security

Making software work together

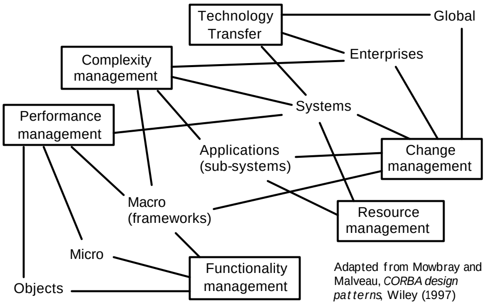
- ORBs, XML

Making organisations work together

- Banks, e-payments

- Co-operation between Microsoft & Intel; IBM & Linux

Primal forces



Adapted from Mowbray and Malveau, *CORBA design patterns*, Wiley (1997)

The enterprise and above

Most large organisations are now extensively based on software

- Even companies such as Boeing – whose core business is making aeroplanes – would regard their IT support as mission critical
- BoI, AIB, Amazon, Tesco,
- Much of it is *mission critical*: if it fails, so does the company

Nowadays enterprise systems are very much tied up with the global infrastructure

- No-one wants to put in a private network when the Internet’s just out there
- This brings a host of problems that are outside the organisation’s control but which directly affect its business, e.g., scalability, security, reliability, trust, availability, ..
- Need to learn how to harness the characteristics of the infrastructure to deliver maximum benefit

All things distributed

When we talk about enterprises, we’re almost always talking about *distributed systems*

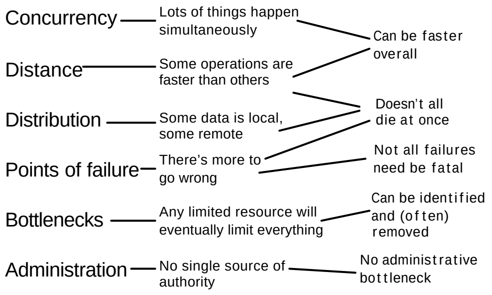
- Different parts of the overall computing solution on different sites (or at least different machines)

Its fair to say that *all* interesting systems are now distributed to some degree – the web impacts everything

- Very few centralised systems being developed – although a lot are still being maintained

This course doesn’t teach the mechanics of a particular distributed systems technology: instead, we focus on the issues in using that technology to solve problems

A whole new set of capabilities



Concurrency — Lots of things happen simultaneously

Distance — Some operations are faster than others

Distribution — Some data is local, some remote

Points of failure — There’s more to go wrong

Bottlenecks — Any limited resource will eventually limit everything

Administration — No single source of authority

Can be faster overall

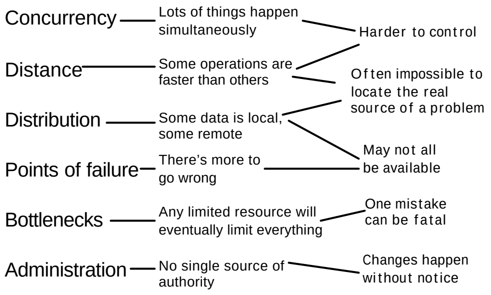
Doesn’t all die at once

Not all failures need be fatal

Can be identified and (often) removed

No administrative bottleneck

A whole new set of problems



Concurrency — Lots of things happen simultaneously

Distance — Some operations are faster than others

Distribution — Some data is local, some remote

Points of failure — There’s more to go wrong

Bottlenecks — Any limited resource will eventually limit everything

Administration — No single source of authority

Harder to control

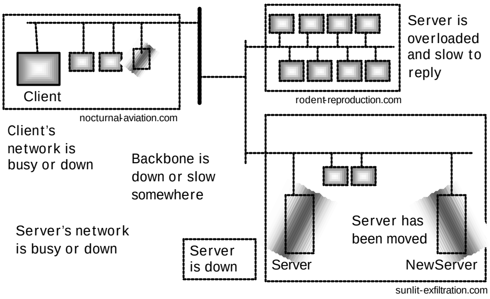
Often impossible to locate the real source of a problem

May not all be available

One mistake can be fatal

Changes happen without notice

Why is the server inaccessible?



Server is overloaded and slow to reply

Client’s network is busy or down

Backbone is down or slow somewhere

Server’s network is busy or down

Server is down

Server has been moved

Or, to put it another way, …

The number of systems in a large enterprise might be upwards of 100,000. The chances of all of them being constantly available, running the same software, fully functional and uncorrupted are about zero. It’s impossible to maintain a single authority over the system, so local changes are inevitable.

And it still has to be made to work, and kept working…

What to do?

Think big: algorithms, data structures, individual applications aren't the important part at this stage

- …because *any* particular choices will cause the same problems

The real problem is to manage the complexity of this structure as it grows

- Isolate points of change as far as possible – affect the whole system uniformly with a single change
- (Try to) keep track of what's where

It's still the same set of issues, but now you need to forget the details and concentrate on the big picture

The highest paid software engineering jobs are the large-scale architects and the people with an exceptionally deep understanding of a particular necessary technology

Transparencies

One way to think about this is in terms of *transparencies*: what features should the user have to deal with, and which should be invisible?

- Location transparency – interact with any service regardless of location
- Failure transparency – if a machine goes down, the service continues uninterrupted
- Load transparency – no matter how many people use the system, it stays responsive
- Upgrade transparency – a component can be changed, moved, maintained or added to without bringing the system down

These ideas were elaborated by, among others, Jim Waldo at Sun

When this breaks down…

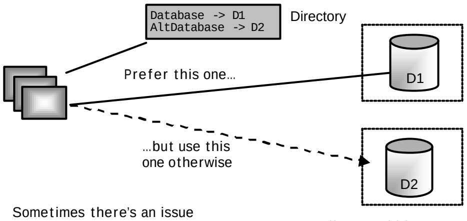
…is when a machine or network breaks down

- Single point of data = *single point of failure* – a failure in one component can bring down the whole enterprise
- Even a relatively short problem can cause cascade failures – even fixing the initial problem may not fix its effects – corrupt data may "remain" in a system (cache) after the problem source has been fixed

Distribution can work to your advantage here

- If you've only got one machine, failures are fatal
- …but if you've got two, you may be able to switch to the back-up
- If the machine is far away, it'll be slow and unreliable
- …but build a closer copy and things might be better
- If a machine is heavily used, it'll be slower
- …so it might be worth spreading the load between several machines

Replication – alternative source



```
Database -> D1
AltDatabase -> D2
```

Directory

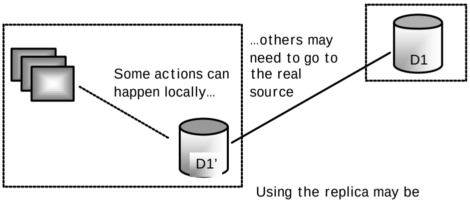
Prefer this one…

…but use this one otherwise

Sometimes there's an issue of consistency – both making sure all clients use the same replica and keeping the replicas up to date

It's usually a good idea to put the replicas on different machines/networks/sites/continents

Replication – local copies



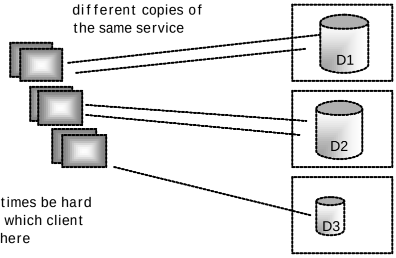
Some actions can happen locally…

…others may need to go to the real source

Using the replica may be transparent (like a web cache) or not (like a mirror site)

Same issues of consistency as other kinds of replication

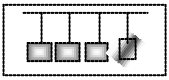
Replication – load balancing



Direct clients to different copies of the same service

Can sometimes be hard to decide which client to send where

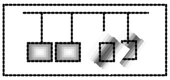
The mind-set



Who uses this server most? Can I move it closer to them?

Are my users distributed? Or all one one site?

How critical is it if this site is inaccessible? Can I alleviate that?



The great thing about location transparency is that these changes are possible

- Relatively simple to play with the locations at the start
- …and changing them should be easy too, if you've got the right infrastructure

Architecture of the enterprise

Think big

Many systems architects never program – they design the overall structure and let others build it

Still have lots of worries, though

- How to make services sufficiently accessible
- How to keep performance acceptable
- How to build a system that can be changed and maintained

There are some programming approaches that can help

To infinity and beyond

Adding *structure* to the enterprise's software

- What accesses what, and what you *want* to access what
- Lots of higher-level structure

It turns out that the ideas from smaller scales re-appear, although with different technology

We'll look at several architectural solutions to the same problem – providing a flexible distributed enterprise IT solution – and see how the large context affects what technology you might adopt

The problem

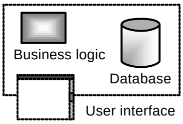
A business wants to use computers throughout its process

- Lots of clients
- Lots of sites
- Lots of data shared between them
- Complex processes and policies
- Everything subject to rapid change

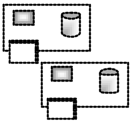
This describes just about every business...

What is the best way to structure this system?

The application



Locally quite a good Model-View-Controller architecture design



MVC - Separate concerns

- The model of the data
- ...from the presentation of that data to the user
- ...from the way the user manipulates that data

Everything in a single application

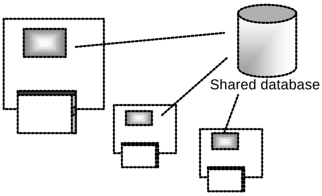
- To change we have to re-compile

Everything is per-client

- …although both the logic and the data are conceptually shared between all clients

The programmer's conceptual level and her implementation level don't match

The shared server

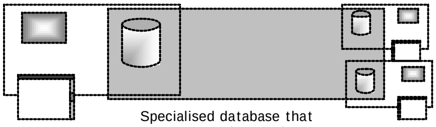


Factor the data into its own abstraction

Sometimes called *two-tier* architecture

- Shared server may be a bottleneck and a single point of failure
- Business logic still scattered per-client

Federated databases

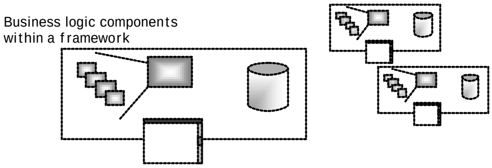


Specialised database that manages consistent views across several systems

Buy-in the expertise – let the experts handle the data consistency for you

- May be too expensive, too inflexible, too slow – generally all three

Component clients

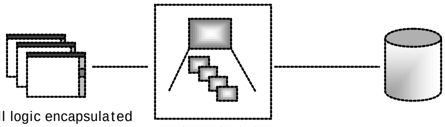


Business logic components within a framework

Stress flexibility

- Components may vary per client – which might be good or bad
- Data still independent – although could be compared with a shared server, of course

Three-tier



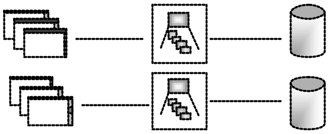
All logic encapsulated into a single server shared by all clients

Business logic layer

More single points of failure

- …but also single points of configuration – change the business logic layer to change the business
- Clients can be very “thin” – just a user interface

Replication in three-tier

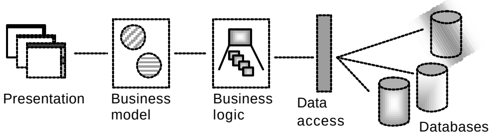


Relatively small numbers of servers to keep up to date

Attractive for certain data usages

- Data mostly read and never updated, or updated in “partitions” which don’t interact with each other
- Simplifies management – two business logic servers instead of a thousand clients to be maintained

Five-tier variation on a theme



Separate presentation from (per-business-unit) models from (enterprise-wide) business logic

Provide a single abstract view onto multiple databases

- Operations may span physical databases

Holding the threads

These architectures all use distribution creatively – and in many ways use it in the way we used classes on the smaller scales

- Cohesive services separated by well-defined interfaces
- Change services while maintaining the interfaces
- Get the benefits of distribution – partial failure, performance, diverse and distributed user communities, …
- Handle the disadvantages – bottlenecks, failures – using implementation techniques within each service

…and many of the same techniques of design (especially abstraction) re-appear

Uses the same structural ideas in different guises

Summary – the largest scales

The very largest scales require that you think as big as possible

- Complete enterprises, co-operation across companies, access by everyone in the world
- *Not* simply a matter of adopting the right standard or low-level technology – architecture counts for more

Almost always multiple solutions to the same problems

- Balancing them is an engineering compromise – the essence of engineering
- Try out the options on paper – sometimes there are some surprising combinations that work well