Common Architecture notes.

Architecture attributes.

There are *7* widely accepted attributes that affect architecture :

- reliability (part of RAS)
  An end-user of a system will regard the service as reliable if it's available when they want to use it, working (i.e. without errors) and performing within acceptable limits.

- availability (part of RAS)
  "The degree to which a system suffers degradation or interruption in its service to the customer as a consequence of failures of one or more of its parts." Components in the system are available upon request. Usually stated in terms of uptime (e.g. 99.999% a.k.a. the "5 9s")

- serviceability (part of RAS)
  A.k.a. *manageability*. "The ease with which corrective or preventative maintenance can be performed on a system (e.g. by a hardware engineer). Higher serviceability improves availability and reduces service cost." The effort required to maintain the system on a day-to-day basis – the more complex the system, the more difficult it is to add components, troubleshoot, etc. A system can be made more serviceable with the use of tools – e.g. network monitors, deployment tools, etc.

- performance
  The system performs within acceptable limits. Usually measured in terms of response time or transactions per second. There's no point to the service being available if it takes 10x the normal time to process a transaction. Modular architectures are easier to performance tune as components can be individually targeted.

- scalability
  How the system copes (or would cope) with additional demand. Usually stated in terms of a graph of performance under various loads – i.e. *linear scalability*. A system can be scaled *horizontally* (e.g. add more web servers), *vertically* (e.g. use the expansion capabilities of existing kit) or *diagonally* (horizontally and vertically).

- security
  How well the system protects components against potential attacks (confidentiality, integrity or denial-of-service). "A combination of processes, products and people." Modular architectures are easier to secure as components can be individually targeted.

- extensibility
  A.k.a. *maintainability, adaptability*. A measure of the flexibility of a system over time – can it be extended / modified to suit new business requirements. Modular architectures are inherently extensible as components can be added/replaced without impact existing functionality. Open standards/systems assists with extensibility – as the system isn't tied to a particular vendor, any suitable product can be used for extensions to the system.

Relationships between attributes.

Attribute	***Function of***
Reliability	**Availability, Performance** – if the system isn't available or is running very slow, it isn't reliable; **Serviceability** – a highly available system may require clustering, etc. so complex to manage
Availability	**Serviceability** – if there are multiple instances available, one can be brought down for maintenance but users can still access resources
Serviceability	**Extensibility** – extensibility and serviceability are at opposite ends of the scale; the more extensible (i.e. modular) the system is, the harder it is to manage
Performance	**Extensibility, Scalability** – if the system is modular, specific areas can be tuned; the system can be scaled to increase performance
Scalability	**Extensibility** – if modular, easier to scale vertically/horizontally
Security	**Serviceability** – if the architecture is complex, management of security is difficult
Extensibility	**Serviceability** – if the architecture is complex, it's difficult to extend

When designing an architecture, it's necessary to balance the attributes above as required for the target system.

Architecture models.

1-tier

Monolithic, "all-in-one" model – e.g. clients are "dumb" terminals connected directly to the mainframe.
Problems with the 1-tier model are :

- any change affects the entire system
- vertical scalability only – limited to the physical expansion capabilities of a single server
- single point of failure
- interoperability – limited connectivity

2-tier

Typically composed of multiple clients and a single server; the clients connect to the server over a network :

- client
  The client is *fat* – e.g. implements the GUI, retrieves data from the server(s), performs business logic based on the data.

- server
  Typically a database. Provides a shared data-store for the client.
  The server doesn't typically implement business logic. Later revisions to the model implement some of the business logic on the server as stored procedures (a.k.a. *fat servers*).

Problems with the 2-tier model are :

- the business logic may be complex and computationally expensive. Consequently, the client may require powerful hardware
- client components are tightly coupled, not modular – e.g. a change to the GUI means shipping the whole application to every client
- data retrieval - each client has to make a direct connection to each server it needs data from. Also, the results transferred may be large and transferred using an inefficient (and perhaps proprietary) protocol
- fat servers use stored procedures which aren't very portable.

n-tier

Composed of :

- client tier
  Components that execute on the client. The business tier is common to all client types - some client components talk to components in the business tier directly (e.g. *thick* clients such as EJB application clients), others via the presentation tier (e.g. *thin* clients such as web browsers).

- presentation tier
  Provides presentation specific support for the client tier. Acts as a proxy between the client tier and the business tier. E.g. a servlet accepts HTTP requests from a web browser in the client tier. The request is translated to a generic business event and passed to the business tier. Responses are reformatted for presentation purposes (e.g. store result in session, generate HTML redirect for JSP) and transmitted back over HTTP for display by the web browser.

- business tier
  Implemented as *business process objects* (provide the business logic – e.g. session beans) and *business objects* (provide an integrated, objectified view of the data tier with support for automated synchronization – e.g. entity beans). The application server provides support for security, transactions, pooling, caching etc. in this tier.

- data tier
  Also known as the EIS tier (Enterprise Information System). Typical components include databases, mainframes, socket servers, etc. Typically components in the business tier access the data tier abstractly using a DAO (Data Access Object) helper class. The DAO takes care of the specifics of data retrieval on behalf of the business objects.

Architecture attribute / model evaluation.

	1-tier	2-tier	n-tier
Reliability	**Mixed** – risk of failure	**Mixed** – availability and performance issues	**Good** – availability and performance are good
Availability	**Poor** – single point of failure	**Poor** – single point of failure (usually a single database)	**Good** – *Pros:* multiple instances (no single point of failure); *Cons:* expensive
Serviceability	**Good** – *Pros:* everything in one place, simple to troubleshoot; *Cons:* tightly-coupled	**Poor** – *Client:* updates have to be applied to all clients in turn; *Server:* ok - everything in one place; simple to troubleshoot	**Mixed** – *Pros:* multiple instances (facilitates day-to-day maintenance); centralised business logic so easy to update; *Cons:* complex; distributed in nature; difficult to troubleshoot; requires ASAs (app server admins), DBAs
Performance	**Mixed** – *Pros:* no remote process communication; *Cons:* a runaway process could affect others	**Poor** – *Client:* result may be large; result data transfer protocol may be inefficient; client may be underpowered (local processing may be intensive); *Server:* ok – e.g. DB caching	**Good** – *Pros:* load balanced over multiple instances; instance and DB pooling; caching; specific modules can be tuned in isolation; *Cons:* process communication overhead (e.g. remote method invocation creates an overhead which affects performance)
Scalability	**Poor** – vertical scalability only; expansion capabilities are physically limited (e.g. free CPU slots, terminal ports)	**Poor** – *Client:* each client requires a separate DB connection; *Server:* server can be scaled vertically; some horizontal scaling is possible but potentially requires client intelligence	**Good** – modular which supports vertical/horizontal scaling; multiple instances; demand-based pooling
Security	**Good** – connectivity is physically limited	**Poor** – *Client:* difficult to enforce the security policy (fat clients are distributed all over the place; could be used/accessed by anyone); available options are SSL/VPN/user authentication (too coarse); *Server:* has to be exposed to the "outside" world for client access so vulnerable to attack	**Good** – *Pros:* security can be applied to each tier and targeted where required; *Cons:* complex architecture so something could be missed; more to secure so could be expensive
Extensibility	**Poor** – any change affects the entire system; component reuse difficult due to tight-coupling	**Poor** – *Client:* client tightly-coupled to the business logic (although code may be reused for clients that support the language the code is written in, there would be multiple deployments); *Server:* if using a fat server, stored procedures have to be provided for each DB used	**Good** – modular in nature so components can be added/replaced as required; business logic is centralised on the app servers

Summary: 1-tier : **MPG-MPG-P**,
2-tier : Mostly **Poor**, reliability **Mixed**
n-tier : Mostly **Good**, serviceability **Mixed**

Common concepts.

Load sharing.

Load sharing is different to load balancing – the load is distributed arbitrarily without any feedback from the components in use.

- DNS Round-Robin - setup multiple alias records for a host
- DNS MX records - MX allows multiple mail hosts to be set
- Web server redirect - setup a redirect from www to www1/www2 using JSP/servlets/Javascript

Pros	*Cons*
1. simple to setup 2. cheap	1. if a server goes down, clients will continue to be directed to it. With DNS the problem is aggravated by the fact that clients will cache DNS and any changes to DNS may take time to propagate through the public DNS servers. 2. load sharing doesn't distribute load where it's needed – e.g. server 1 overloaded, server 2 idle 3. the Web server redirect is only suitable for HTTP

Load partitioning

Clients are sent to particular servers based on their state – e.g. if (userNum < 10,000) go to server 1. Yahoo/Hotmail use this technique for webmail - us.f206.mail.yahoo.com *Cons:* doesn't distribute load, users locked to one server.

Load balancing & fault tolerance.

Load balancing uses a family of algorithms (e.g. allocate 60% to server 1 and 40% server 2) and feedback from the components (e.g. if a server is down it's removed from the valid servers list) to intelligently distribute load and assist with fault tolerance.

- hardware
  Usually a network device (such as Cisco Local Director, BIG-IP, Alteon) that appears as a single IP address and routes traffic to any servers in the active server list for any protocol (done at the network level). The device monitors the state of the machines in the active server list and automatically patches them out if they go down. If required, the device can use "sticky" connections – once a connection is established, always route the client to the same server (unless the server dies)

- software
  Either used in conjunction with a hardware device or standalone. One or more application servers appear as a single server and communicate with each other via some form of broadcast mechanism. The server uses the naming service and transparent request redirection to balance clients; the clients use smart stubs to balance requests across servers.

Pros	*Cons*
1. uses intelligent algorithms and feedback to truly balance the load / provide fault tolerance 2. transparency – client is unaware of any changes 3. hardware : appears as a single IP so cannot be cached by clients 4. hardware : redirects at the network level so works for any application	1. difficult to setup 2. expensive

Firewalls.

DMZ - "A network added between a protected network and an external network in order to provide an additional layer of security." – i.e. provides physical separation of the networks by putting each on it's own subnet. Subnets can only be traversed by going through a router/firewall – control can be applied at this point.

Traditional Firewall	*Modern firewall*
• packet based filtering - filter at network layer; block by IP, port, etc. • application level control - filter at application layer; understand HTTP, FTP, etc.	• stateful inspection implement control at network layer but understand higher layers (e.g. FTP, HTTP). Store state for individual connections and control access - e.g. open up FTP for a particular user; close when user session completes • proxy style access control Filter by request type, content type, etc. Deny web surfing during office hours, etc. Built in user-authentication - e.g. pops up browser authentication; telnet password is handled by firewall

Architecture layers.
Architecture deals with components at three levels :

- *infra-structural*
  Underpins the platform components. Typical infra-structural components include the network (cable, hubs, firewalls, routers), the UPS, the Internet connection, etc.

- *platform*
  Underpins the application components. Typical platform components include *hardware* (servers, disk arrays), *services* (operating system, Web/EJB containers, DB servers, DNS servers, clustering, network monitoring) and *software* (home-grown load monitors)

- *application*
  The application components run on top of the platform components. Typical application components include web applications, EJB components, standalone daemons/applications.

Sample application areas for architecture attributes.
Key : I = infrastructure, P = platform, A = application.

- *reliability*
  (I) clustering – HLB (hardware LB) with feedback
  (I) redundant kit – active/passive firewall pair, spare kit at site
  (I) co-location – fire/power protection, redundant telecomms, monitors
  (I) DNS – MX, RR
  (I) staged deployment of new services
  (I) DR site
  (I) backup telecomms sourced from multiple suppliers
  (P) clustering – application servers
  (P) hardware – hardware monitors/watchdogs, EDO memory, RAID, managed file system
  (P) software – app resilience / load monitors, transaction logs for DR
  (P) general – use best hardware/software products within budget
  (P) general – QA for release procedures, software/component testing, etc.
  (A) graceful degradation (timeouts, bypass, pass-through, etc.)

- *scalability*
  (I) horizontal – hardware clustering via HLB/DNS RR; partition network by use (e.g. perimeter, app, etc.), add extra kit/push up kit from lower layer; common OS/app build
  (I) co-located sites – burstable bandwidth, rackspace, etc.
  (P) horizontal – app server clustering; distributed components
  (P) vertical – add extra CPU/memory/disk; app server pooling, activation/passivation
  (A) multi-threading
  (A) short lived transactions
  (A) partition content by type – e.g. static / dynamic
  (A) graceful degradation

- *security*
  (I) firewalls, DMZ's
  (I) NAT, PAT
  (I) proxy servers
  (I) DNS – only expose required addresses
  (I) physical access
  (I) third-party network testing services
  (P) OS security – OS hardening, TCP wrappers, ACL
  (P) JVM security – security.policy
  (P) app server security – EJB security mechanisms, certificates, etc.
  (A) SSL, server sessions, domain range checks, lockdown, audit trails