Patterns for High-Availability Internet Systems

Paul Dyson and Andy Longshaw

Introduction

Many internet-based systems have similar architectures. Such similarity provides an environment rich for pattern mining. This paper sets out a pattern language that considers the common structures and elements identified in internet-based systems we have been involved in building and reviewing.

Internet-based systems

For the purposes of this paper we consider an internet-based system to be one that uses internet channels to deliver content and services to end users and/or other systems. This means that the system has to deal with stateless and ultra-thin client software. Typically (although not exclusively) internet-based systems are also characterised by having an unregulated and largely anonymous user base that interacts with the system in a number of different ways and with a number of different goals.

Such a characterisation covers a broad range of systems, from a personal web-page through to a multi-national e-commerce site such as Amazon.com (the ubiquitous internet system example). This paper is firmly targeted towards the Amazon-end of this spectrum; this is where our experience and interest lies and also where architectural patterns are of the most interest. More specifically, these patterns mainly consider systems that are deployed across multiple hardware and software servers – the most common approach for enterprise-level internet-based systems.

Architectural patterns

Architectural patterns consider the structure rather than the function of a system. In particular, we have considered the following non-functional characteristics of systems when evaluating the patterns and their relationships:

- Availability – carries on functioning even in the face of failure
- Performance – end-to-end responsiveness to external (user or other system) requests
- Scalability – ability to provide (or be easily changed to provide) same or similar levels of performance under increased load
- Maintainability – ability to add new or change functionality and services over the lifetime of the system
- Manageability – ability to organise and control the system as well as the ability to obtain information about the non-functional characteristics at runtime
- Security – resilience to intrusion, protecting both architectural elements of the system and the information it holds
- Flexibility – ability to change functionality or services at runtime
- Portability – ability to migrate the system to a new platform
- Reliability – ability to run without failing

These are all technical non-functional characteristics. There is also a non-technical characteristic that has massive impact on the architectural choices we make:
- Cost – the cost to implement a chosen solution in terms of time and money

**Pattern template**

The following template is used for the pattern language:

- **Problem**: What is the problem we are trying to solve?
- **Context**: In what environment(s) do we see this problem?
- **Example**: An example of the problem.
- **Forces**: What makes this problem a problem? What are the benefits and liabilities of the potential solutions?
- **Solution**: How we solve the problem.
- **Example Resolved**: An example of the solution based on the problem example
- **Resulting Context**: What is the environment created by application of the solution?
- **Hardware/Software**: Does this pattern apply to hardware or software design, or both? We use the term ‘Server’ often in the language to refer to a hardware instance, a software instance or a combination of the two.
- **Related Patterns**: Other patterns in the language related to this one.

**Language history**

The experience captured in these patterns has been gained over the implementation and review of a number of internet-based systems over the past four years. The initial draft of patterns presented here was produced by the authors as part of the materials for training course commissioned by Sun Microsystems – elements of the content in this pattern language are reproduced with their permission.

The authors ran a workshop on patterns of internet architectures at the OT2002 conference in Oxford, England in April 2002 (see [www.ot2002.org](http://www.ot2002.org)). This allowed us to both validate and extend the ideas captured in the original draft of this paper. Both authors would like to thank Anthony Barnes, Andrew Broughton, Mark Campbell, Stephen Hutchinson, Clive Menhinick, Vera Peeters, Rajiv Tyagi, Louise Whelan, and Eoin Woods for their contributions.

This paper was reviewed in a writer’s workshop at EuroPLoP’02. Our shepherd for this paper was Peter Sommerlad and the authors would like to acknowledge his help and guidance in shaping this paper. We would also like to thank the members of the workshop for their insightful suggestions and encouraging praise.

This paper is actually a vastly cut-down version of the original – summaries of the original patterns can be found in the appendix.
About the authors

**Paul Dyson** has spent the past 4 years working exclusively on internet-technology projects for clients such as Philips, Interbrew and lastminute.com. In all of these projects he has been given the job of ‘application architect’ – playing a key role in defining both the logical and physical architecture of the system. Paul is the author of a number of pattern languages including ‘Patterns for Abstract Design’ and ‘State Patterns’.

**Andy Longshaw** used to be a Principal Technologist for QA Training but now has to work for a living. For most of the past 7 years, he has been tasked with learning new technologies, understanding their possibilities and limitations, and fitting them into the real world of software development. Andy has worked on Internet architecture-related projects for clients such as Sun, Microsoft, and Tesco.
The Example

Much of the work the authors have carried out on internet-based systems is still covered by various non-disclosure agreements. The example used throughout the paper is an anonymised amalgam of a number of those systems. Every pattern presented in this language has been observed in at least three systems the authors have had personal involvement with, often in more than one incarnation.

The running example we will use is the fictional GlobeTech site. GlobeTech is a major manufacturer of consumer goods which range from low-value, high-volume goods right through to specialist-market, high-value items. The outlets for these items range from small, independent outlets to multi-national retailers.

The main business driver for the website is to improve their service to both their retail partners and their end customers. The funding for the website is contributed jointly by the Marketing and Sales, and Customer Service departments: Marketing and Sales can see a new channel for attracting and retaining customers, and Customer Service believes it can significantly reduce the cost of after-sale care on low value items by making much of its content (such as FAQ’s and manuals) available on the web. Marketing and Sales is also keen to start to conduct much of its retail-partner business via the site. They see that this can both improve level of service provided to retail partners and reduce costs.

The main areas of site functionality are:

- Product Catalogue – complete catalogue of all products GlobeTech make a sell throughout the world
- Outlet Finder – search engine for GlobeTech outlets by location throughout the world
- Customer Care – searchable online version of all Customer Service content, linked to the product catalogue
- Promotions – various mechanisms for promoting products such as those recently launched or those that might be of interest to particular users
- Customer Database – registration of all users of the site, used to personalise customer care and promotion functionality
- Shopping Basket – limited direct-sell capability for products that are hard to obtain in retail outlets
- Retailer Ordering – bulk ordering facility for retail partners
- Retailer Contacts – retailer-specific searchable contact database to allow retailers to identify GlobeTech personnel they should deal with for specific enquiries and vice-versa

GlobeTech will invest a substantial amount of money in the site and want to make it available globally. That is to say that they will service over 100 countries with the site and present all content and services in over 200 different languages. However, they are not going to spend all the money on an as-yet-unproven concept. The system will initially be launched in 10 countries and 13 languages, then rapidly rolled out to the rest of the world if it proves to be a success.
GlobeTech estimate that the final system will need to cope with around 10,000 users simultaneously accessing the consumer-facing side of the site, with around 2,000 users on the retailer-facing side. Average interaction lengths will be 20 minutes for consumers but 4 hours for retailers.

Even in the initial incarnation of the system the expectation is for around 2000 consumers and 50 retailers. From the outset the system must be truly 24x7x365 operational – stated minimum availability is 99.9% as the initial 10 countries cover most of the globe and the length of the retailer interactions means that there is no foreseeable ‘dead time’ for the system.

The story so far …

There are no assumptions about the final architecture of the GlobeTech website and the initial architecture, before consideration of any of the non-functional requirements, is a monolithic system:

This architecture has one thing going for it: it is extremely simple (which leads to a second benefit – cost). Many systems can, and have been implemented with this architecture. However, we do not expect that it will meet the non-functional requirements of the initial GlobeTech system, never mind the final incarnation.
# The Language

## Language Summary

The table below summarises the affect on the technical, non-functional characteristics the introduction of a particular pattern will have:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Availability</th>
<th>Performance</th>
<th>Scalability</th>
<th>Maintainability</th>
<th>Manageability</th>
<th>Security</th>
<th>Flexibility</th>
<th>Portability</th>
<th>Reliability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functionally Identical Servers: Active-Redundant Servers</td>
<td>+</td>
<td>-</td>
<td>-?</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Session Fail-over</td>
<td>+</td>
<td>-</td>
<td>-?</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Replication</td>
<td>+</td>
<td>-</td>
<td>-?</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Functionally Identical Servers: Load Balanced Servers</td>
<td>+</td>
<td>+/-</td>
<td>+</td>
<td>-/+</td>
<td>-</td>
<td>-</td>
<td>+/-</td>
<td>+/-</td>
<td>+/-</td>
</tr>
<tr>
<td>Common Persistent Storage</td>
<td>+/-</td>
<td>-/+</td>
<td>-/+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+/-</td>
<td>+/-</td>
<td>+/-</td>
</tr>
<tr>
<td>Dedicated Web and Application Servers</td>
<td>+/-?</td>
<td>+</td>
<td>-/+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+/-</td>
<td>+/-</td>
<td>+/-</td>
</tr>
</tbody>
</table>

- **+** Has a positive effect on the characteristic
- **-** Has a negative effect on the characteristic
- **-/+** Has both positive and negative effects on different aspects of the characteristic
- **?** Potentially has an effect (positive or negative as indicated) on the characteristic
Functionally Identical Servers: Active-Redundant Servers

Problem
How do we ensure the system remains available to the user in the event of failure or the need to upgrade or maintain the system?

Context
We are implementing an enterprise-level internet-based system where high availability is one of the prime non-functional requirements.

Example
The GlobeTech website will serve over 100 countries around the world and hence will be in use round the clock. The site is intended to augment, and in some cases totally replace, the business-critical functions of customer service and retailer ordering. Hence, extended periods of downtime are not acceptable.

Forces
- The system needs to have high (minimum 99.9%) availability. This means less than 90 seconds downtime per day. No system is immune to failure, however, whether in hardware or software and this availability has to be achieved even in the event of such failure.
- The system needs to have high availability but it also needs to be maintained and upgraded over time, which typically requires bringing the system down, performing the maintenance/upgrade, and then bringing it back up again.

Solution
Implement the system using two Functionally Identical Servers – servers that are capable of performing the same functions even if they have different non-functional characteristics. Deploy one of these servers as a redundant server – only used in the event of failure or the need to maintain the active server. Introduce a switch that can be used to direct traffic away from the active server to the redundant server.

Example Resolved
We replicate the monolithic GlobeTech system hardware/software combination to introduce a redundant Functionally Identical Server. We also introduce a switch that automatically directs traffic to the redundant server in the event of the active server failing or can be requested to move traffic away from the active server if we want to maintain it. We also introduce a Functionally Identical Server for the switch, which takes over should the active switch fail:
Whilst the redundant servers are functionally identical, they do not share the same non-functional characteristics. The active system hardware is a Sun 450 with four processors, 4GB of memory and four 36GB drives. The redundant system hardware however, is a Sun 250 with one processor, 1GB of memory and two 18GB drives. This reduces the cost of introducing the Functionally Identical Server but means that the system is operating with reduced capacity during the period of failure or maintenance. The switches are completely identical.

**Resulting Context**

- Availability is improved because the redundant capability can be switched in to take over if the active system fails or if maintenance is required.
- Although the introduction of a switch increases the time taken to process a request, the amount of delay added is so small as to have an insignificant impact on Performance.
- Manageability is impacted due to the fact that we have an additional server and two new switches to manage.
- Security is impacted because we now have more system elements to protect from attack.
- Reliability is impacted because the switching mechanism introduces a new point of failure.

**Hardware/Software**

- Both hardware and software servers need to be functionally identical if the system is to remain fully available in the event that the redundant servers are switched in.
- The switches themselves can either be software or hardware.

**Related Patterns**

- *Session Fail-over* between the Functionally Identical Servers is required if the server maintains user session state.
• *Data Replication* between the *Functionally Identical Servers* is required if the server stores persistent, non-static data.

**Session Fail-over**

**Problem**

How do we ensure that user interaction is uninterrupted in the event of failure or the need to upgrade or maintain the system?

**Context**

We are implementing an enterprise-level internet-based system where high availability is one of the prime non-functional requirements. The system maintains interaction state in the form of a user session held on the server\(^1\).

**Example**

The GlobeTech website has many functions that require state to be maintained between individual browser requests. Some of these include:

- Outlet Finder – it allows users to refine searches using only the results of previous searches
- Shopping Basket – the user adds items to the basket over time before ordering the items
- Retailer Ordering – the user follows a multi-stage process to create and place an order

**Forces**

- Whether due to failure or maintenance, the redundant server will be switched in occasionally. The overall availability of the system is maintained but the individual user interaction is interrupted as the session information is lost when the active server goes down and the redundant server knows nothing about user or their session. As far as that user is concerned, the system failed.

**Solution**

Implement a mechanism that persists session information whilst the user is interacting with the system. Make this information available to the redundant server(s). In the event that the redundant server is switched in, have it read in the persistent session information and continue the session as normal. The user will see no perceptible interruption in their interaction.

**Example Resolved**

We introduce a session backup server to the GlobeTech architecture\(^2\):

---

\(^1\) Some internet-based systems bypass the need for Session Fail-over by keeping all state on the client in the form of one or more cookies. This solution is impractical for most systems for two main reasons: many users cannot or will not allow cookies to be stored in their browsers; and the amount of state required to be maintained for complex interactions – such as the Retailer Ordering functionality in the GlobeTech example – makes cookies an unsuitable storage mechanism.

\(^2\) The switches have been omitted from the diagram rather than removed from the architecture.
Whichever system software element is serving users persists session information to the backup server (both have to do this because a single user session might last the time taken to switch over to the redundant server and back to the active server). When one of the system software elements receives a request for a user session it has no knowledge of, it queries the backup server to see if the session is persisted there and, if a session is found, loads it in and continues with the request.

This is a simple and passive form of fail-over – the session is only failed over if the user makes a request in the session after the redundant server is switched in. An alternative implementation would be for the session backup server to detect that the redundant server has been switched in and force it to load all the persisted sessions. This is much more complex but can be useful if the form of user interaction isn’t simple request-response.

Of course, we probably should introduce a second, functionally identical, backup server in case of failure or the need for maintenance. This comes down to cost – the second backup server is only needed if both the active server and the backup server fail at the same time (or there is a need to take both down at the same time for maintenance).

**Resulting Context**

- Availability is further improved – not only is the system more available as a whole, but the site will continue to be available to users in the event of failure or the need for maintenance.
- Performance is impacted because of the overhead of persisting session information.
- Maintainability is potentially impacted if the system software needs to be specially coded to take account of Session Fail-over. Some application servers make session backup almost transparent to the application developer.
- Manageability is impacted because of the additional management required by the session backup server.
• Reliability is impacted because the backup mechanism introduces a new point of failure.

**Hardware/Software**

• Session backup and fail-over is a software solution.

**Related Patterns**

• *Session Fail-over* requires *Functionally Identical Servers* so that the session can fail-over to a server capable of executing against it.

• *Session Fail-over* can employ *Data Replication* techniques if several backup servers are deployed as *Functionally Identical Servers*.

**Data Replication**

**Problem**

How do we ensure the system state is consistent between servers in an architecture that uses multiple *Functionally Identical Servers* if the system maintains dynamic data?

**Context**

We are implementing an enterprise-level internet-based system where high availability is one of the prime non-functional requirements. The persistent system data consists of both static data (data that doesn’t change within the context of the system) and dynamic data (data updated by the system itself).

**Example**

The GlobeTech system requires both static and dynamic data:

<table>
<thead>
<tr>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Product catalogue (updated by a content management system outside of the internet-based system)</td>
<td></td>
</tr>
<tr>
<td>• Outlet information (also maintained by content management)</td>
<td></td>
</tr>
<tr>
<td>• Customer care information (also maintained by content management)</td>
<td></td>
</tr>
<tr>
<td>• Promotions (maintained by eCRM, content management and campaign management systems)</td>
<td></td>
</tr>
<tr>
<td>• ...</td>
<td></td>
</tr>
<tr>
<td>• Customer database (updated every time the customer edits their information)</td>
<td></td>
</tr>
<tr>
<td>• Shopping basket orders (created whenever the shopping basket is used to purchase goods)</td>
<td></td>
</tr>
<tr>
<td>• Retail orders (created whenever a retailer places and order)</td>
<td></td>
</tr>
<tr>
<td>• Retailer contacts (updated whenever the retailer updates their own contact information)</td>
<td></td>
</tr>
<tr>
<td>• ...</td>
<td></td>
</tr>
</tbody>
</table>

In addition to the system dynamic data, we have decided to introduce a second, functionally identical, session backup server to ensure the system remains available even if the session backup server is down at the point the active server fails. The persistent session data needs to be consistent between the two session backup servers.

**Forces**

• Whether due to failure or maintenance, the redundant server will be switched in occasionally. When it does, it needs to use the latest version of the dynamic
data maintained on the active server if it is to present a consistent system state to the user and other systems.

- We could maintain a single copy of the dynamic system data that is accessed and maintained by both the active and redundant servers, but this impacts availability – if the data storage mechanism fails, neither of the servers will be able to function correctly.

**Solution**

Implement a mechanism that replicates the dynamic data between both the active and redundant servers. When one server writes dynamic data, the transaction is not completed until the data is available on the other server, unless that server is not available. In the case where data has been written to only one server, the other cannot be made available until all data has been synchronised.

**Example Resolved**

In the GlobeTech architecture we select a DBMS for persistent data storage that supports *Data Replication* between two instances of the database. We use the DBMS for both the dynamic system data and the persistent session information:

**Resulting Context**

- Availability is improved because *Data Replication* ensures dynamic system data is consistent between both active and redundant servers.
- Performance is impacted due to the effort required to replicate the data.
- Maintainability is potentially impacted if the system software needs to be specially coded to take account of the replication mechanism.
- Manageability is impacted because the replication mechanism introduces another system element that requires management.
• Reliability is impacted because the replication mechanism introduces a new point of failure.

**Hardware/Software**

• **Data Replication** is usually implemented in software, often as part of the database management system product being used.

**Related Patterns**

• **Data Replication** supports **Functionally Identical Servers** if the system needs to maintain dynamic data.

---

**The story so far …**

If we employ **Functionally Identical Servers** with **Session Fail-over** and **Data Replication** we have a highly-available system; the functionality, session data and dynamic persistent data are all still available in the event of failure or the need for maintenance. But what about Performance, Maintainability, Manageability – all impacted by the architectural choices made?

If all of these were sufficient in the current architecture we would probably choose to stop. Again, this is a common architecture that has been successfully deployed many times. However, in the GlobeTech example the impact on these characteristics is too great and we have to think further.

**Functionally Identical Servers: Load Balanced Servers**

**Problem**

How do we achieve consistent performance in the event of failure or the need for maintenance?

**Context**

We are implementing an enterprise-level internet-based system where good performance is a fundamental requirement in order to make the system usable. We have introduced **Functionally Identical Servers** to give high availability but have compromised the non-functional characteristics of the redundant server to reduce cost.

**Example**

The specification of the redundant server in the GlobeTech architecture is massively inferior to that of the active server. Whilst the active server can probably serve around 3000 consumers and 500 retailers with satisfactory performance, the redundant server can probably only handle 600 and 100 respectively. In the event of failure or maintenance, the system performance is going to be very poor, possibly rendering the system totally unusable.

**Forces**

• Whether due to failure or maintenance, the redundant server will be switched in occasionally. When it does, **Session Fail-over** ensures that the user
interaction isn’t noticeably interrupted and *Data Replication* ensures that the dynamic data on the redundant server is consistent with the active server. However, all of this is pointless if the system performance suddenly degrades so heavily that most or all the users now find it unusable.

- We could restrict the number of users allowed to access the redundant server to a level it can happily deal with, but this means that the system is effectively unavailable to the remainder. Having a system that is only available to a small proportion of the users that want to access it may not be satisfactory.
- We could specify the redundant server to the same level as that of the active server but this effectively doubles the cost of the infrastructure. This is a very inefficient use of resources if the redundant server is only required for a small percentage of the time.

**Solution**

Deploy *Functionally Identical Servers* but introduce a mechanism to balance user load across them continuously rather than simply switching in a redundant server when the active server fails. Ensure that the non-functional characteristics of these *Load Balanced Servers* are broadly similar (they do not have to be identical if the load-balancing algorithm takes account of the differences) so that performance is reasonably consistent in the event that a server goes down. Consider replacing one high-powered active server with one low-powered redundant server with a number of medium-powered servers to achieve better levels of performance at a roughly similar cost.

**Example Resolved**

First, we introduce a further two *Functionally Identical Servers* to the GlobeTech architecture. Next, we replace the switches that only direct traffic to the redundant server in the event of failure or explicit request with a load balancer[^3] that continuously spreads traffic across all servers depending on the individual server load:

[^3]: Actually, two load balancers, both Functionally Identical, one only used if the other fails. This degree of redundancy is inexpensive and there is always an entry point where we either have a single point of failure or we introduce redundancy. The redundant load balancer is not shown in the architecture diagram.
Finally we scrap the hardware specification we started with (1x450 fully-populated, 1x250 partially-populated) and replace it with a specification for three Sun 250s, each with two processors, 2GB memory and two 18GB hard drives. This will cost a little more than the original specification, but also gives us a slightly improved capacity, approx. 4600 consumers and 600 retailers. It certainly costs less than two fully-populated Sun 450s.

**Resulting Context**

- Availability is improved over the monolithic system architecture as one of the *Functionally Identical Servers* can go down and the traffic will continue to be balanced between the remaining servers. Additionally, the system remains usable in the event of the loss of a server rather than just “technically available”.

- Overall system performance (throughput) is improved because each server can be used optimally while all servers are available

- Individual user performance (end-to-end) is slightly impacted because of the introduction of a layer of indirection that executes an algorithm.

- Manageability is impacted because multiple servers have been added to the system – all of which need to be managed.

- Security is impacted because there are now multiple points of attacking the system software and data.

- Reliability is improved because traffic is spread evenly across all servers rather than concentrated on a single server for the majority of the time.

- The Scalability characteristics of the system are improved since it becomes comparatively simpler to add another functionally identical server to deliver more capacity (serve more customers).

**Hardware/Software**

- Load Balancing can between hardware servers, software servers or a combination of the two.
The load balancer itself can be either hardware or software.

**Related Patterns**

- Functionally Identical Servers: Load Balanced Servers is an alternative or extension to Functionally Identical Servers: Active-Redundant Servers
- Data Replication is still required to ensure all servers have up-to-date copies of the dynamic data.

**Common Persistent Store**

**Problem**

How do we deploy Load Balanced Servers if our Data Replication mechanism doesn’t support peer-to-peer, rather than master-slave, replication? Even if it does, how do we prevent performance problems when we add multiple servers that all need to maintain the dynamic data?

**Context**

We are implementing an enterprise-level internet-based system where good performance is a fundamental requirement in order to make the system usable. We have introduced Load Balanced Servers to give good performance even in the event of failure but are concerned that the Data Replication mechanism might become a bottle-neck, or even not be able to work, if there are more than two servers.

**Example**

The current GlobeTech system architecture uses Load Balanced Servers but our choice of DBMS doesn’t support peer-to-peer Data Replication. If we want to use three (or more) Load Balanced Servers we need to change the architecture.

**Forces**

- Load Balanced Servers need to access a consistent set of dynamic data just as Active-Redundant Servers do. However, all the servers will be updating the data simultaneously, rather than the active server updating the information and the redundant server taking over as necessary. Such peer-to-peer replication is not often supported by persistent storage mechanisms and, if it has, can suffer from performance problems.

**Solution**

Implement a single persistent store that is shared between all the Load Balanced Servers. This Common Persistent Store will hold a single set of the dynamic data, with all the servers updating that single set. In order to avoid the persistent store becoming a single point of failure (the problem that originally led us to use Data Replication), implement the Common Persistent Store as a pair of Active-Redundant Servers that use Data Replication to ensure that the redundant server can be used, with an up-to-date version of the data, in the event of the active server failing or requiring maintenance.
**Example Resolved**

We split out the persistent storage element of GlobeTech system and run it as a pair of *Active-Redundant Servers* in their own hardware. We have selected a further two Sun 250’s with the same specification as the other servers:

![Diagram showing system architecture with load balancer, system software, system hardware, DBMS, persistent storage hardware, active servers, and redundant server.]

**Resulting Context**

- Performance is improved because we no longer have the overhead of replicating the data amongst multiple active servers. *Data Replication* is confined to a single pair of *Active-Redundant Servers*. However performance is also impacted over the original monolithic architecture as we have split the persistent data storage from the system software, which requires a new communication channel to be introduced.

- Manageability is improved because the *Data Replication* mechanism is now back to its simple form. However Manageability is also impacted because we now have a persistent storage server (and its redundant pair) to manage.

- Security is impacted because there are separate functional and storage servers to protect.

- Reliability is improved because the complex multi-server *Data Replication* mechanism has been removed. However, Reliability is also impacted due to the addition of a new point of failure.

---

4 The session backup server has been omitted from this diagram in the interests of clarity
**Hardware/Software**

- The *Common Persistent Store* is often deployed to its own hardware. However, a variation could be employed where the system hardware is used to house persistent store software. This reduces hardware requirements (and, hence, cost) but means that the system has more work to do to remain available in the event of hardware failure.

**Related Patterns**

- The *Common Persistent Store* servers should be implemented as *Active-Redundant Servers* in order to avoid it becoming a single point of failure.
- *Data Replication* will be required to keep *Active-Redundant Servers* consistent

---

**The story so far …**

We have replaced *Active-Redundant Servers* with *Load Balanced Servers* for the system functionality in order to achieve good and consistent Performance in the event of failure. We have also split out the data storage element of the system into a *Common Persist Store* to make the Performance improvements achievable. However, the version of the architecture above is not commonly encountered; often another layer of *Load Balanced Servers* is usually introduced …

---

**Dedicated Web and Application Servers**

**Problem**

How do we optimise the Availability and Performance of a system spread across a number of *Functionally Identical Servers* of different types?

**Context**

We are implementing an enterprise-level internet-based system for which availability is one of the prime non-functional requirements, and good performance is a fundamental requirement in order to make the system usable. We have a number of *Functionally Identical Servers*, deployed as both *Load Balanced Servers* and *Active-Redundant Servers*.

**Example**

In the GlobeTech system architecture we can see several groups of *Functionally Identical Servers* emerging that are deployed either as *Load Balanced Servers* (the functionality servers) or *Active-Redundant Servers* (the load balancers themselves, the session backup servers, the persistent storage servers). Each of these groups has been introduced to improve Availability, Performance, or both.

**Forces**

- We have been working to maximise Availability by introducing redundancy, either in the form of redundant servers or servers that form part of a load-balanced group. However, maximising Availability isn’t just about making the system able to respond in the event of failure, it is also about making that
response in a time acceptable to the user. Is a system that fails to respond for 35 seconds working to produce the response or simply unavailable? To the user, there is no distinction between the two.

- To make a timely response the user we need to optimise the processing of the user’s request – handling session tracking, business logic processing and the production of the response. However, we also need to optimise the handling of that request – handling the low level request/response protocols and requests that require no processing such as passing images and other ‘static assets’ back to the browser.

**Solution**

Split the system software into software optimised for handling user requests and software optimised for processing those requests. Physically deploy these software elements as either *Load Balanced Servers* or *Active-Redundant Servers*.

**Example Resolved**

We separate the system software into web server and application server software elements:
We now have two sets of *Load Balanced Servers*: the web-servers and application servers. The web-servers themselves are load balanced as well as the application servers.

**Resulting Context**

- Availability is maximised because we can deal with loss of web or application servers (even simultaneously) and still serve user requests with little or no degradation of Performance.
- Overall Performance (throughput) is potentially improved due to the further use of load balancing in both sets of servers.
- User-perceived Performance (end-to-end) is impacted because of the introduction of additional layers and communication paths due to the additional servers.
- Maintainability is potentially improved because each different type of function can be considered, and maintained, separately.
• Manageability is both impacted and improved: the addition of new servers gives a larger management problem but the function, and hence the management, of each set of servers is simplified.

• Security is impacted because each set of servers provides an opportunity for attack.

• Reliability is both impacted and improved: each additional server adds a new point of failure but the simplification of the function of each server means that it can be made more reliable.

**Hardware/Software**

• This pattern is typically applied to software elements that are then deployed to their own set of hardware.

**Related Patterns**

• *Common Persistent Store* is usually employed in addition to Dedicated Web and Application Servers to give the ‘classic’ three-tier internet system architecture.

• The web and application servers are deployed as either *Load Balanced Servers* or *Active-Redundant Servers*.

• We can further improve manageability and maintainability by introducing appropriate Isolation Layers or One Way Dependencies.
The story

In this paper we have presented three common architectures for internet systems:

1. Single-box: the de-facto starting point for any architecture is itself a deployable solution and has been used successfully in many cases. These cases, though, typically have minimal requirements for availability, maintenance, etc., but a pressing requirement for low cost.

2. Active-redundant servers: the system is implemented as an active server that backs up its session information and dynamic data to a redundant server which is switched in when the active server fails or is taken down for maintenance. To keep cost down, the redundant server is typically of a much lower cost (and hence specification) than the active server.

3. Load-balanced servers: the system uses multiple layers of Load Balanced Servers, plus several active-redundant server pairs to achieve high-availability at the expense of high cost. This is the architecture we have seen most prevalently in the enterprise-level internet-based systems we have worked on.

The evolution of a simple single-box architecture to these more complex, and more highly-available architectures is described in terms of a small pattern language that expresses the commonality and differences between these architectures and why you would choose one (or a bespoke variation of one) over the others. The patterns themselves describe the benefits and liabilities of applying them (they all improve availability but have varying effects on the other non-functional characteristics) and the diagram below gives some indication of the cost of the different paths of evolution ($ indicates relative cost and √ indicates relative improvement in availability):
Appendix A - Related Potential Patterns in the Internet Architectures Pattern Language

The patterns presented here form part of a much wider pattern language, which considers the issues of performance, scalability, maintainability, manageability, security, flexibility, portability and reliability. Thumbnail sketches of these other patterns are provided below.

Peripheral Specialist Servers

"Extraordinary" functionality is delegated to specialist servers that are focused on the provision of a specialist service. Examples of services would be security (e.g. based on Microsoft Active Directory) or batch processing (sending emails or processing asynchronous transactions).

Isolation Layer

Provide a level of insulation between different parts of the system to improve flexibility and maintainability. An example would be the addition of a persistence layer that would decouple the application and the database used.

Virtual Platform

Provide a complete Isolation Layer that insulates the application from the underlying hardware, database, and operating system. An example of this would be a J2EE application server and the Apache Web Server, both of which can be found on multiple platforms of varying power.

One Way Dependencies

Make dependencies between system elements one-way to increase flexibility and maintainability of the system. For example, the Web servers will make calls on the application servers, but the application servers are unaware of the nature of their clients (they could be browser clients, SOAP clients, or mobile clients).

Dynamically Discoverable System Elements

Provide additional configurability and flexibility for highly available systems by having the application discover new system elements at runtime. An example would be a load balancing server that can detect newly added servers and add them to its balance pool.

Runtime Configuration

Improve maintainability and flexibility by having the application read its configuration information as required. An example would be the ability to change the level of logging performed by an application and then signalling it to re-read its configuration information.
Local Caching
Cache frequently used information "locally" to the application to improve performance. An example would be the use of a local cache for catalog data in a Web shopping application.

Logical Server
Provide a single mechanism to manage different types of system elements on a server or across multiple servers. An example would be the use of SNMP monitoring information that could be aggregated by a single management console.

Continual Status Reporting
Have each system continually report its status as a predictive guide to application usage and the provision of additional capacity. An example would be to have the application servers report their current load every 10 seconds via SNMP.

Logging
Log system events and the information around it to allow for the tracing of errors or exceptional conditions. An example would be the use of a configurable level of error logging in the application server layer.

Demilitarised Zone
Improve the security of the system by limiting the number of servers exposed on a separate, shielded network. An example would be to have only the Web servers on the first network accessible from the outside and restrict network traffic to and from those servers.

Secure Channels
Secure sensitive data in transit by using encryption and authentication. An example would be the use of HTTPS when passing credit card information between a browser client and a Web server.

Secure Servers
Protect a whole server or set of servers by requiring authentication. An example would be extranet functionality provided to business partners through a Virtual Private Network (VPN).

Expandable Hardware
Procure hardware that can be expanded without having to increase the number of servers switches/routers in the system. An example would be purchasing a higher-range server that can increase its number of processors and memory beyond that of a lower-range server.

Connection Limitation
Ensure a minimum level of service by restricting the number of concurrent users of a constrained resource. An example would be to restrict the number of concurrent database users so that each can obtain the data they need in a reasonable timeframe.
Resource Pooling

Improve performance by holding a pool of resources that are expensive to initialise. An example would be database connections that are pooled by application servers.