An Empirical Study of Java-Based Web Application Architectures

Shaun Richard Tipson

A thesis submitted in partial fulfillment of the degree of
Bachelor of Science (Honours) at
The Department of Computer Science
Australian National University

November 2001

© Shaun Richard Tipson

Typeset in Palatino by TEX and LATEX 2ε.

Except where otherwise indicated, this thesis is my own original work.

Shaun Richard Tipson
23 November 2001

To my mother and father,
in the hope that this is Communication.

Acknowledgements

First thanks must go to Shuping, for her constant support and wealth of technical knowledge, and to Richard, for his timely intervention when focus needed to be regained.

Thanks also to my editorial team, ably headed up by Dara, who did not let such trivial things as a ten hour time difference affect the constant flow of corrections and suggestions, and Kirsty, whose contribution can be seen from the very first page.

At the CSIRO, thanks to Paul, for always taking time out to answer my questions and listen to my rants. It is nice to know that someone cares when the software starts hating you.

For everyone else, it is hard to know what to say, especially since this will inevitably be the most well–read section of the thesis. To mention everyone is impossible so, rather than leaving anyone out, I would like to thank all the people that helped to make the year enjoyable, and especially my fellow chons.

Abstract

Tiered architectures are widely used in the construction of commercial web applications because they are commonly perceived, and often cited, as a mechanism for the production of high quality software systems. Unfortunately, this perception is rarely supported by empirical analysis, an issue that this thesis addresses.

Two types of tiered architecture—two–tier and three–tier—are implemented as non–trivial, commercial–quality web applications. These applications are then empirically evaluated, and the results are used to compare the two implemented architectures, and to analyse the quality of the implemented applications.

The comparative analysis focuses on two system properties, robustness and performance, and concludes that two–tier architectures are more robust, but three–tier architectures are capable of greater performance. The qualitative analysis concludes that the tiered architecture does produce high quality web applications, but only if the architecture is well suited to the system's purpose. In addition, guidance is given for the selection of a suitable tiered architecture through a set of recommendations, the most important of which being that the architecture should optimise system complexity.

The implementation, evaluation and analysis verifies the quality of the tiered architecture type as a model for the construction of commercial web applications, and thereby contributes to the creation of better electronic commerce systems.

Contents

List of Figures

Chapter 1

Introduction

It seems amazing that in spite of the many ways in which the Internet has revolutionised our lives, there is still doubt over whether or not electronic commerce is a viable business model. Many businesses have e-commerce sites for reasons of promotion rather than profit, and any loss that the site makes is written off as a marketing expense. Of course, this approach is not viable for companies whose core business is e-commerce. Many online retailers have collapsed or been swallowed up because they have not managed to sustain a profitable implementation of their commercial model.

Even the biggest e-commerce vendors are struggling to make money online. Retailer `Amazon.com` has accumulated US$2.72 billion worth of losses in the six years that it has been operating.[1] `CDNow.com` managed to accumulate US$212 million in losses in the six years of operation before it was acquired by the German media group Bertelsmann.[2] Online toy retailer `eToys.com` accumulated over US$272 million in debts before collapsing in March this year. [3] The `eToys.com` web site is now operated by K*B Toys, the United States' largest toy retailer.

This pattern would seem to suggest that the only companies capable of operating a viable e-commerce site are those capable of absorbing large losses, in the short term at least. The result is a marketplace dominated by a few large firms, perhaps operating anti–competitively.

Maintaining competitiveness in the e-commerce marketplace can, therefore, only be achieved if it is possible to enter the market and produce a profit in the short term. This goal is applicable to both startup e-commerce companies and existing "bricks-and-mortar" businesses looking to move onto the World Wide Web ("the web").

One method of achieving a more competitive e-commerce marketplace is to find ways to build online applications better, faster and for less money. This thesis demonstrates how applying the concept of software **architectures** to the construction of commercial web applications can achieve this result.

The specific architectures examined in this thesis possess a **tiered** structure. The concept of tiers is explained and explored in chapter 2. Chapter 3 introduces a tiered

[1] `http://www.freep.com/money/business/amaz7_20010907.htm`
[2] `http://news.cnet.com/news/0-1007-200-2466723.html?pt.yfin.cat_fin.txt.ne`
[3] `http://news.bbc.co.uk/hi/english/business/newsid_1208000/1208079.stm`

architecture specification called J2EE. Chapter 4 explains the architecture evaluation method, which is used to empirically evaluate two types (chapters 5 and 6) of tiered architecture. The results from the empirical evaluations are combined in chapter 7 to perform a comparison of the two evaluated architectures, and to present a summary of the properties of tiered architectures. This summary is followed in chapter 8 by a brief statement of conclusions that can be drawn from this thesis, as well as indications for future work.

The research goals of this thesis are stated more explicitly in section 2.4, following the discussion of tiered architectures. However, before proceeding to this discussion, it is necessary to understand the context in which tiered architectures operate.

1.1 Internet 101

The Internet was first created for the United States' Department of Defense, who wanted a command and control system that could survive nuclear war. This first **internetwork**, or network of networks, was rapidly adopted by scientists for the exchange of technical information.

A major shift in the focus of the Internet began when TCP/IP was adopted as the official Internet protocol suite on 1 January 1983. The number of computers connected to the Internet started growing at an exponential rate, and this number continues to double nearly every year.

As traffic volume spiralled upwards, the dominant type of network traffic shifted from the scientific to the general. Groups of people with shared interests started to meet and exchange information, a process leading to the formation of "virtual communities".[4]

1.2 Internet Commerce

The formation of online communities inevitably created a need for a way to exchange goods and services, a process that has become known as **electronic commerce**.

This thesis evaluates the ability of a particular architectural model, tiers, to construct electronic commerce software systems. In other words, it investigates how *appropriate* tiered architectures are for the construction of e-commerce applications.[5] This statement is refined and formalised in section 2.4, once the twin concepts of tiered architectures and web applications have been defined (section 2.2) and explored (section 2.3).

In conducting the evaluation, it is realised that the appropriateness of an architecture is greatly influenced by context. A good place to start the discussion is, therefore,

[4]For more information on the history and the evolution of the Internet, see Chapter One of Tanenbaum [1996].

[5]From this point on, web application and e-commerce application are considered to be terms with the same meaning. Web application is explicitly defined in section 2.2.

the nature of e-commerce relationships, since they define the context in which a web application architecture must operate.

E-commerce relationships are a subset of normal commercial interactions. As in normal commerce, any electronic relationship has two characteristics:

- The relationship of the parties to each other (**framework**); and
- The process of exercising the relationship (**transaction**).

Transactions are discussed in section 1.2.1, and frameworks are discussed in section 1.3. This order has been chosen as frameworks facilitate, and therefore require, some understanding of transactions. In both cases, the discussion illustrates the constraints that operate on commercial web applications.

1.2.1 Transactions

Transactions occur whenever two parties engage in a commercial relationship. The pervasiveness of transactions has made their study a popular area of computer science.

At the simplest level, features that are desirable in a transaction are either functional or technical. The study of functional requirements has been undertaken in the field of communications in computer networks. The technical requirements, on the other hand, originate from the study of relational database systems.

Garfinkel and Spafford [1997] define the functional requirements applicable to parties in a transaction as consisting of four elements:

- **Confidentiality** Only involved parties should be able to view a transaction.
- **Integrity** Transaction information should be formed in such a way that any tampering can be detected.
- **Authentication** Each party should be able to establish with certainty the identity of the other party.
- **Non–repudiation** Once a transaction has occurred, it should be impossible for one party to deny that they participated in the completed transaction.

The technical requirements are applicable to the transaction itself, and complement the functional requirements [Haerder and Reuter 1983]: [6]

- **Atomicity** Either a transaction is executed in its entirety, or it is not executed at all. A completed transaction is said to be **committed** in the system.[7]

[6] These are known as the ACID properties.

[7] Tygar [1996] has carried out a study of the relevance of atomicity to e–commerce systems.

- **Consistency** All transactions should take the database from one "consistent" state to another. Consistency in this sense means constraints on the data. For example, after a stock transaction, the total amount of stock should remain unchanged (even though different customers may now own different amounts).

- **Isolation** Events within a transaction should be hidden from other transactions that are running concurrently. This implies that transactions are completely independent of each other, and the failure of a single transaction will therefore not affect any other transaction.

- **Durability** Once a transaction has been committed, any changes that have been made to the database should be permanent. Permanent in this context means that the changes should persist even if there is a system failure.

Before discussing frameworks, it is worth noting that transactional requirements are just one of the many constraints on e-commerce systems. Another major constraint, software characteristics, is discussed in the context of tiered applications in section 2.3.[8]

These two types of requirements, software and transactional, have dominant effects on the cost of a web application.[9] Minimising these costs is a substantial step towards producing a more cost-effective e-commerce application. The application will still be of little use, however, unless it is appropriate to the framework it is deployed in.

1.3 Frameworks: Past, Present and Future

A framework describes a way for technically divergent pieces of software to interrelate. The problem for producers of web applications is that the Internet framework is constantly changing and evolving.

The proposition[10] and economic evaluation[11] of frameworks is a recurring subject in the literature. However, a gap in the literature remains, as a coherent approach towards categorisation and rationalisation of Internet frameworks has yet to be developed.

This section attempts to remedy this gap by presenting some example e-commerce frameworks. These examples have been classified according to the mapping of the relationships between customers and retailers. For example, "many to one" denotes many customers for a single retailer. The different relationship types are compared diagrammatically in figure 1.1.

[8] These characteristics place requirements on the system. Example characteristics are flexibility, scalability and robustness.

[9] Daoud [2000] uses "e-commerce phases" to derive a slightly different list of e-commerce system requirements.

[10] [Shim et al. 2000; Grimm et al. 2000]

[11] [Subramani and Walden 2000; Liu and Arnett 2000; Yarden 1997]

1.3.1 One to One: Serial

Serial relationships were the first type of e-commerce framework. As an extension of the physical buy–and–sell process, the basic idea is that the customer sends money to a retailer, and the retailer sends the customer the requested product. Third-party mechanisms, such as escrow, can be introduced to ensure that the customer receives the product, and that the retailer receives payment.

The problem with this architecture is that since a vendor can only talk to one client at a time, it can not be easily scaled. Transactions must be performed serially, so processing one hundred clients will take one hundred times as long as processing one client.

1.3.2 Many to One: Clients and Servers

Client/server frameworks became popular in the late 1980s as a method for linking many client computers to a single central resource. Clients submit queries to, and receive services from, this shared resource (the **server**). Clients are classified by the level of functionality they implement: a high level for a **fat client**, and a low level for a **thin client**.[12]

The client/server framework on the Internet is the model of the Big Fat Web Server[Shirky 1999]. Using this model, all transactions are processed in one place: the server. This means that clients have only limited functionality, and are therefore thin. The advantages of this model are:

- For the **server**, centralising the processing of transactions means that the retailer can control the performance and security of the system. The server does not have to rely on the client providing any services; and
- For the **client**, participating in a transaction is an easy process since the server provides all the functionality. All the client has to do is maintain one end of a communications channel.

Client/server is currently the dominant Internet framework. This is mainly because the Internet protocol, TCP/IP, is based on a request/response communication model, which integrates well into a client/server structure.[13]

1.3.3 One to Many: Agents and Servers

The idea of autonomous mobile agents in electronic commerce is relatively new [Prasad 1996]. The basic principle is that a customer can give a set of parameters to a piece of software called an **agent**. This agent can then search for the good or service that best matches the supplied set of parameters.

[12] For more information on the history and structure of client/server frameworks, see [Edelstein 1994].
[13] Bhattacharjee and Ramesh [2000] present a more general discussion of communication models.

As an example, a customer might require an airline ticket between two cities at a certain time. Once it has been given this information, an agent can survey many different travel operators to find the cheapest ticket that matches the specified destination and time.

An agent can move from machine to machine, interacting with processes on the current machine. It has been shown that significant performance benefits over the client/server model can be achieved by using a mobile agents framework. This is mainly due to elimination of the lag times associated with client/server communication [Hagimont and Ismail 1999].

1.3.4 Many to Many: Peers

The most recent framework is peer-to-peer (P2P). In this model each network node is considered to be an equal **peer**. Peers form connections with each other based on need, and there is generally no need for a centralised control structure.

The application that drew attention to the P2P model was the music-sharing site Napster,[14] whose closure was indirectly the result of not having a true P2P model.[15] Napster successors such as Gnutella[16] are true peer-to-peer networks, where each node running a Gnutella client is an equal participant in the network.

A peer-to-peer network requires that all nodes be created equal, since dominant nodes will change the network type to client/server. P2P networks are therefore useful only when nodes are not required to possess specialised functionality, such as when information is exchanged or processing load needs to be shared.

1.3.5 Comparing Frameworks

Figure 1.1 gives a graphical representation of the frameworks described above. The order in which the frameworks have been presented is also intended to mirror the chronological order of their adoption. A summary of the properties of the different frameworks is presented in table 1.1.

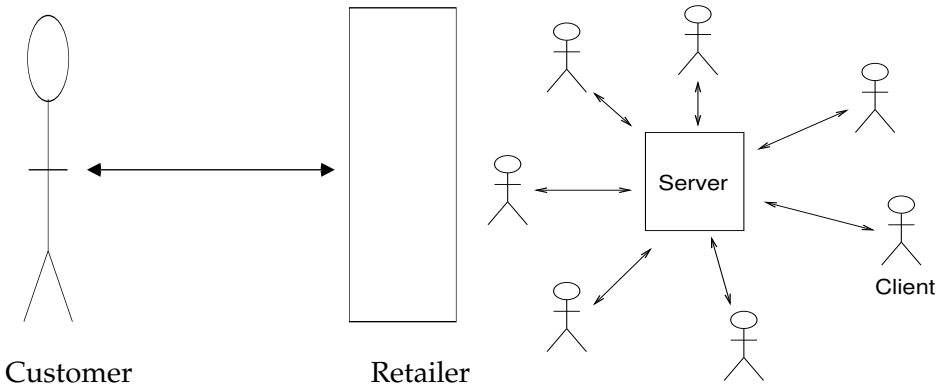
It should be noted that the frameworks overlap in many ways. For example, the connections required by the serial and P2P models are similar, and the client/server and mobile agent models both require the existence of specialised server machines.

Transactions and frameworks form the context for the discussion in the next chapter, where tiered architectures and web applications are introduced.

[14] `http://www.napster.com`

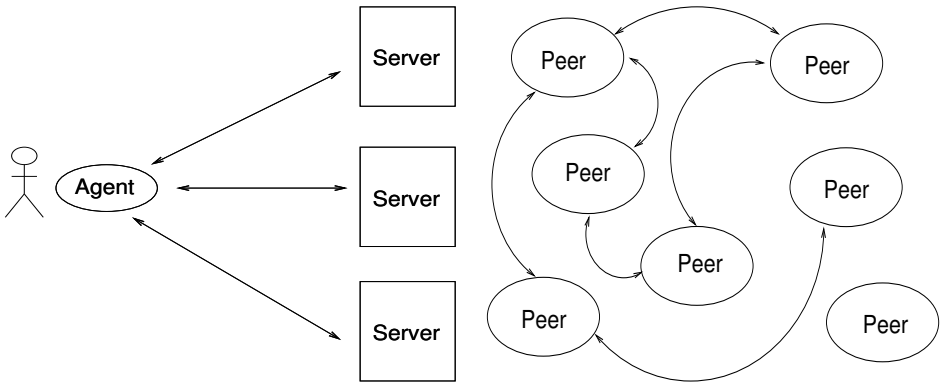
[15] Napster had a centralised directory service. This meant that the music swapping service could be terminated simply by closing the directory service.

[16] `http://www.gnutella.wego.com`



(a) Serial (one to one)

(b) Client/Server (many to one)



(c) Mobile Agent (one to many)

(d) Peers (many to many)

Figure 1.1: Electronic Commerce Frameworks

Framework	Dominant Feature	Good For
Serial	Equal parties	One-off, asynchronous relationships
Client/Server	Central server	High bandwidth, transactional networks
Agents	Autonomous customer representative	High lag, server-based networks
Peers	Equal parties, dynamic communication structures	Egalitarian, informational networks

Table 1.1: Comparing Frameworks

Chapter 2

Tiered Architectures

This chapter gives an introduction to (sections 2.1 and 2.2), as well as providing a rationale for (section 2.3), the theoretical idea of tiered architectures. Once this background has been established, section 2.4 describes, in greater detail, the research goals of this thesis.

2.1 Introduction

Tiered architectures can be introduced from two perspectives:

1. As a refinement of the modular software design principle [Parnas 1972], itself an extension of the stepwise refinement [Wirth 1971] and structured programming [Dahl et al. 1972]; and
2. As the product of distributed computing practice. Tiered architectures are extensions of the client/server architecture, itself an extension of mainframe and file–sharing architectures.[1]

The first perspective, as the more theoretically developed, is used in sections 2.2 and 2.3 as the context in which to define and describe tiered architectures. However, the second perspective should not be neglected since it represents the real–world use of tiered architectures. The difference between the two approaches is that software design defines application structure, whereas computing practice defines application functionality. The functionality required of applications in the web context has already been discussed in chapter 1, and will be returned to in section 2.4.

In the meantime, the structure of tiered architectures, and the relationship between tiered architectures and web applications, will be addressed.

2.2 Definitions

This section serves two purposes: first, to standardise the use of terms that often have ambiguous meaning in both academic literature and popular publications and; sec-

[1] For more detail on the history of these architectures see Edelstein [1994].

ond, to use these terms to derive a meaning for what is meant by the phrases *web application* and *tiered architecture*.

The definition of terms will proceed in a top-down fashion, where each successive term (or group of terms) possesses a smaller scope than the one it follows. The top of the list is therefore the highest level of abstraction, the bottom the lowest.

Framework Frameworks define the relationships and interactions between autonomous entities, whether they are people or software systems. An example of a framework is the client/server communication model described in the previous chapter.

Application Applications (or **systems**) are software structures that serve the requirements of commercial entities. **Enterprise applications** deliver the wide range of services that are required in the day-to-day running of a large business, such as data storage, message passing and transaction management. Applications are potentially composed of many different pieces of software, integrated into a structure.

Unless stated otherwise, the term **architecture** will be used at this level of abstraction.

Tier A tier is a specialised type of system **component**, and, like general components, are obtained when a software system is modularised. Tiers differ from general components in the communication channels that they are allowed to form with other tiers (see figure 2.1). Software architectures formed through the use of tiers are evaluated in this thesis.

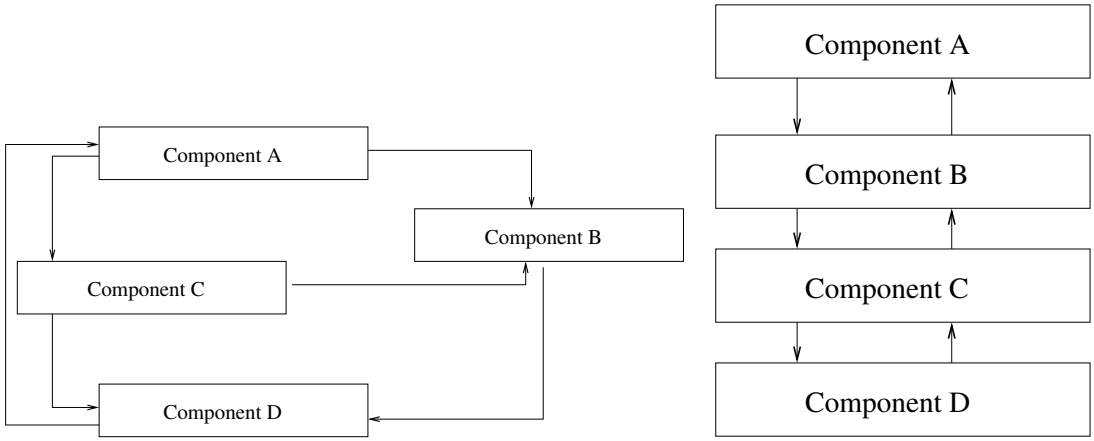
Container A container is a software subsystem that is responsible for loading, managing and executing objects. A container is normally associated with a single operating system **process**, which in Java means a single Java Virtual Machine (JVM).

Containers are an important abstraction idea in the Java 2 Enterprise Edition specification (see section 3.4).

Package A package is a collection of objects. The group of objects required to perform a specific software function will generally be stored in a single package.

Object An object (or **class**) is a construct that is used to encapsulate program code. This adheres to the Java definition of an object, and means that the term object will only be used to refer to a single Java class.

This definition of object reflects a deliberate decision to differentiate the terms "component" and "object", even though they are often used interchangeably in the academic literature. Components are the product of application decomposition, objects are produced by a programmer. This decision was made because



(a) General Modular Architecture (b) Tiered Architecture

Figure 2.1: Architectures Viewed at the Component Level

for web systems, modularisation can be undertaken at more than one level of abstraction.[2]

The specifications that implement frameworks are produced by specialist organisations such as the World Wide Web Consortium (W3C), normally with the sole purpose of facilitating the use of a specific technology or idea. Applications, on the other hand, are implemented by individual businesses. Frameworks are implemented once, whereas applications are potentially implemented millions of times.

Although it is possible to begin anew every time an application needs to be implemented, a more sensible approach is to build on experience through the use of an **architecture**:

Architecture An architecture is a model that is used to guide the construction of a software system.

Tiered Architecture A tiered architecture is an architecture consisting of components and communication channels forming a tiered structure (see figure 2.1).

Web Application A web architecture is an application capable of offering *commercial*[3] services to clients using the HyperText Transfer Protocol (HTTP).

Web applications almost invariably contain a **web tier**, i.e. a tier that separates the internal system from the Internet.

[2]System decomposition produces components, whereas component decomposition produces objects.
[3]This is a limitation on the generic meaning of web application, which can mean any HTTP–enabled application.

Tiered architectures can be used to construct web applications, and are therefore **web application architectures**. Measuring the quality of web applications produced from tiered architectures will be returned to in section 2.4.

2.3 Exploring the Tiered Approach

It has been argued that any architecture can be completely described by three criteria: "elements, form and rationale" [Perry and Wolf 1992].[4] This section begins the description of tiered architectures by presenting the rationale. A discussion of elements and form is presented, in the context of J2EE, in the next chapter.

The following list is a set of desirable characteristics for a software system, as extracted from Meyer [1997]. Next to each element is a description of how the characteristic is exhibited (or not) by a tiered architecture, and, where necessary, an explanation why the characteristic is also desirable in web applications. The intention of this list is to provide both a familiarisation with the tiered model, and explore the suitability of the tiered structure for the creation of web applications.

Correctness The quality of any software application is largely dependant on its correctness, defined as the degree to which it satisfies the specified requirements.

For modularised systems, correctness is divided into two parts: the correctness of each tier, and the correctness of inter-tier communication. This simplification is made possible by the abstraction of components and links, where at the system level each component is a "black box".

Since communication is an unchanging requirement in the architecture, it is possible to implement many different tiered designs with the same communication infrastructure. This is beneficial for the business implementing an application, because they can purchase and immediately apply a commercial communication product.[5]

The black box property also means that, where appropriate, a tier can be implemented as a third–party product. A common example is the use of a commercial relational database management system (RDBMS).

The purchase (rather than creation) of functionality means that modular architectures are likely to produce applications that are cheaper and easier to maintain.[6] A second *code minimisation*[7] strategy, container management, is discussed in section 3.4.

[4] Quoted in [Ran 1998].

[5] For example, one implementing the CORBA specification (see section 3.7).

[6] The standard of comparison is non–modularised applications. The rationale is that specialised products are better at doing their particular task, and mass production means that the product costs less.

[7] Code minimisation means that the business creates as little code as possible (i.e. third-party vendors create most of the code). Since software construction is normally an expensive process, less code implies a cheaper application.

Robustness Robustness is the ability of a system to reduce the total number of failure occurrences, and to recover quickly when a failure occurs. Robustness helps define the system quality of a web application, and is therefore a vital characteristic.[8]

In a tiered architecture, the failure is generally isolated within one tier. As a consequence, the only system components affected by the failure are the damaged tier and its neighbours. The "black box" nature of the tiers also means that the system can be returned to full functionality by repairing or replacing the damaged tier.

Even greater robustness can be achieved through the physical separation and replication of tiers. Each tier then becomes a **cluster** (see figure 2.2), and the failure of any one machine will not cause a system failure (although system performance may be affected).

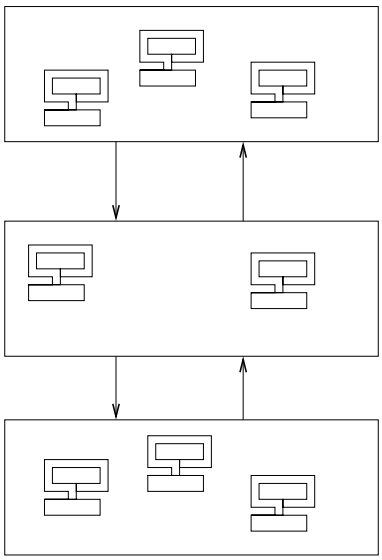


Figure 2.2: Each tier as a cluster

Scalability Also known as **extendability**[9], this concept can be defined as:

1. The ability to insert new functionality or components into an architecture (*Architecture Scalability*); or
2. The ability to replicate the functionality of a component across several machines or processes in response to increasing demand (*Performance Scalabil-*

[8]System quality was identified as one of four critical factors for the success of commercial web sites by Liu and Arnett [2000]. The other factors were system use, information quality and playfulness. See also Helander and Khalid [2000].

[9]As distinct from *extensibility*, a functionality requirement.

ity).

The first definition represents a general application integration issue (see Illback and Sholberg [2000]), and is achieved in modular architectures by the component/connectors abstraction.

The second definition relates to the regulation of work flow, and is a strong attribute of tiered architectures. Tiers allow resources to be dynamically allocated in response to fluctuating demand. For example, if a business's web tier is experiencing a surge in customer demand, the tiered architecture allows the site manager to spread the web tier load across many processes or machines.

Tiered architectures possess a high degree of scalability because each tier can be scaled separately with the effects of scaling limited, in the worst case, to the adjacent tiers. In the best case, the effects of scaling a tier will not be externally visible at all.

Scalable web applications can change to suit the needs of web retailers. Scalability allows the application to respond to increasing demand and to incorporate new functionality.

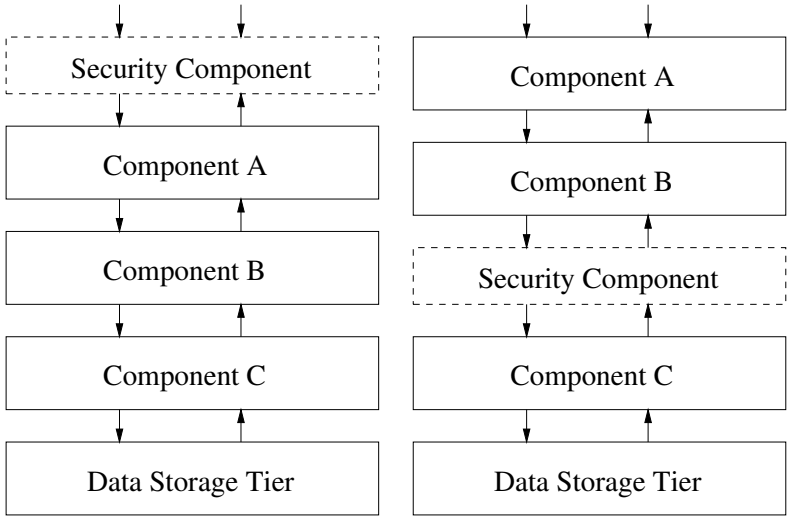
Compatibility Compatibility can be defined at two levels in tiered architectures: client–application compatibility and intra–application compatibility. In the web domain, this former is not an important issue because the structure of all communication is mandated by the HyperText Transfer Protocol (HTTP).

The latter, component compatibility, is, however, important in all component–based applications since the components need to understand each other. This understanding can be achieved either:

- By the use of a uniform communication infrastructure for all components (see CORBA, section 3.7 below); or
- By limiting the number and type of communication channels that can be formed. Use of this strategy is common in tiered architectures since each tier is only required to be able to communicate with adjacent tiers.

In tiered architectures, problems arise when different component vendors implement different communication methods. This is especially common when the same company makes more then one tier, and develops proprietary connectors between them.

Functionality Another benefit of the modularisation process is that common functionality can be gathered together in one place. The grouping of functionality allows specialisation to occur. This is a desirable outcome because it means that software (and people) are able to do a small job well rather than a large job poorly.



(a) Security Tier as Gatekeeper (b) Arbitrary Placement

Figure 2.3: Placing a Security Tier in the Architecture

As an example, the creation of a web tier has allowed specialisation to take place in the web domain. Specialised software, such as web servers and web containers (see section 3.6.1.1), performs limited tasks well. Specialised roles, such as web developers and graphic designers, focus on presentation without needing to have knowledge about programming, protocols or databases.

Functionality also incorporates the idea of **extensibility**, the ability to extend the functionality of the system after construction. An example is an application constructed for web clients, that is subsequently required to offer the same services to mobile phone clients.

Efficiency This characteristic focuses on the **performance** of the system. Although the throughput of a tiered architecture is limited to its slowest tier,[10] the Extensibility property discussed above means that resources can be added and reapportioned easily.

Efficiency is a subject that will be revisited in the architecture evaluations. It is also an important property for web applications since the ability to scale is of little use if it takes one hundred servers to provide sufficient system throughput.

Integrity The aspect of system integrity relating to reliability has already been discussed above under Robustness, which leaves the issue of system **security**. Se-

[10] In the cases relevant to this thesis. See section 2.3.1.

curity for a web application means that the system should prevent unauthorised access to, and modification of, stored information.

A tiered architecture enhances system security in two ways:

- **Protection** Data is stored in the bottom (hardest to reach) tier of the architecture. In other words, a data access can only be achieved by a request that passes through every tier, and each time the request is forwarded it can be checked and, if necessary, filtered; and
- **Abstraction** Security policies can be defined for the application or for particular tiers. Security can therefore be managed with better control and at a higher level of granularity. This flexibility is a result of allowing different levels of abstraction. Abstraction is an issue that is return to in the next chapter.

The Protection aspect can be further enhanced by creating an entire tier dedicated to security, and inserting it at the appropriate level (see figure 2.3).

Maintainability Also known as **serviceability**, maintainability is defined in [International Organization for Standardization 1996] by reference to four metrics:

1. The difficulty associated with **analysing** the system;
2. The difficulty associated with **changing** the system;
3. The difficulty associated with **testing** the system; and
4. The **stability** of the system.

Maintainability is measured over the lifetime of the system, and comprises a large fraction of the overall system cost.

The most important maintainability feature possessed by tiered architectures is simplicity. This makes the architecture easy to understand, and hence analyse and test. The tiered structure also means that the effects of any changes are localised to adjacent tiers. This makes it easier to modify the system, and once modified the system is likely to be more stable.[11]

All of the above elements demonstrate the advantages of using tier architectures,[12] however the widespread adoption of tiered architectures by business has been driven by the one remaining characteristic:

Integration With Legacy Systems Many companies already possess an internal computer system that is responsible for common tasks such as purchasing, billing and information storage.

[11] Than general component or monolithic (non-modularised) systems.

[12] Further discussion of tiered architectures can be found in Steiert [1998], Intel Corporation [2001], Chaffee [2000] and Olague [2000].

Such an application can be easily "web-enabled" by the incorporation of a web tier into the system. There are several motivations for adding this functionality to the system:

- Customers are able to interact directly with the company across the Internet, increasing the pool of potential customers and decreasing transaction costs;
- Employees can interact with each other and the system without being physically present. Money and time lost due to commuting can therefore be reduced; and
- Commercial relationships with other companies (business-to-business) can be established. These relationships reduce costs by making transactions with these companies more efficient.

Given that the process of software construction is extremely expensive, there is a real incentive for system managers to integrate a web tier into their existing system. The alternative is rebuilding the system from scratch, an extremely expensive proposition.

2.3.1 Performance Limitations

This section concludes the rationale for tiered architectures by discussing the limits on tiered application performance.

Queries from a client to a tiered application proceed only as far as necessary into the application. For example, in a web application a request for a static web page only requires functionality possessed by the web tier. This means that the query can be processed at the web tier, and does not need to proceed any further into the application. Predicting performance limits for the application will therefore be difficult since the load on each tier will depend on the mix of incoming requests.

The situation can be simplified by assuming that each query traverses the entire application. In other words, a query that reaches the top tier will continue through the application to the bottom tier, and then back up to the client.[13]

Given this assumption, application performance will improve as the number of concurrent queries increases until the application becomes **saturated**. The nature of the tiered structure means that saturation will be caused by a bottleneck at a single tier because the upper bound on application performance is the performance of the weakest tier.

There are two possible causes of tier saturation:

1. A tier cannot accept additional work when its processing capacity is completely utilised. Processor saturation is a **hardware** constraint, and can be resolved by employing more physical resources; or

[13] All queries used in the evaluation process were of this type (see section 4.2).

2. A tier cannot execute more concurrent transactions than the software resources allow. For example, if only twenty concurrent database connections are allowed, a tier attempting to access the database can only process twenty concurrent requests.

   Resource saturation is a **software** constraint, and is difficult to resolve because the software managing the resource may need to be modified or replaced.

Overall application performance will therefore be limited either by processing or resource constraints.[14]

2.4 Research Goals

The broad aim of this thesis, as introduced in the previous chapter, is to investigate how appropriate tiered architectures are to the construction of web applications. This investigation addresses a significant gap in the academic literature: the empirical analysis of tiered architectures in the context of the World Wide Web (the "web").

The need for such an analysis is acute, since tiered architectures are widely used in the web environment (see section 2.4.1), and although tiered architectures are often cited as producers of high quality applications,[15] this claim is rarely supported by empirical data. In addition, there is also a poor understanding of the effects that are produced by modification of the tiered architecture. Again, in this area there is much popular knowledge but little empirical analysis.

In order to simplify the analysis, a decision was made to focus the investigation on the system properties that relate to *performance*. The scope was limited in this way because the other properties were either not readily converted to empirical values (e.g. compatibility and functionality) or required measurement over a large time scale (e.g maintainability). With this restriction in mind, the following research goals were created:

- **Implementation**

  The implementation, as a non-trivial, commercial web application, of two types of tiered architecture. Where possible, two applications—using different commercial products—were implemented for each architecture. This was done to ensure that the results of the evaluation were independent of product-type.

  The rationale for the choice of the two chosen types of tiered architecture is given in section 2.4.2. The technology used in the implementation process is described in chapter 3, and the implementations are described in detail in sections 5.4 and 6.6.

[14] Network bandwidth is a potential third constraint, however, the speed of the testbed network (section B.2) meant that resource and processing saturation occurred well before network saturation.

[15] For example Intel Corporation [2001], Chaffee [2000], and Olague [2000].

- **Evaluation**

  The performance evaluation of the implemented architectures, with the aim of ascertaining the behaviour of the implementations under different stresses.

  The evaluation method is explained in chapter 4. The results of the evaluations are presented in sections 5.5 and 6.7.

- **Derivation**

  The derivation of architectural properties from the performance evaluation data, and determination of the quality of web applications produced from tiered architectures.

  Architectural properties are derived in sections 5.5 and 6.7, and summarised in chapter 7. Conclusions on architectural quality are made in section 7.3.

- **Comparison**

  Comparison of the two implemented architectures, with conclusions as to the suitability of each type.

  The comparison and derivation of conclusions is undertaken in section 7.2.

- **Recommendation**

  The production of a set of recommendations, based on the evaluation, applicable to the construction of web applications using tiered architectures.

  The set of recommendations is presented in section 7.4.

Before beginning the investigation, it is necessary to understand the significance of tiered architectures to web applications, and to briefly describe and justify the architectural types that were chosen for evaluation.

2.4.1 Tiered Architectures and Web Applications

Tiered architectures were chosen over other web application architectures for evaluation in this thesis for the following reasons:

Theoretical Properties As has already been shown in section 2.3, tiered architectures theoretically possess many desirable qualities.

Current Use All web systems containing an application server[16] are tiered in structure. The application server market is currently worth over US$1.5 billion per annum, and is growing at an exponential rate [Gilpin and Zetie 2000].

Future Potential The frameworks most likely to provide future growth in e-commerce are client/server and mobile agents[Mariotti and Sgobbi 2001]. Both of these models require servers, and therefore centralised web applications.

[16] See section 6.3

The Big Fat Web Server[17] will be a critical part of web frameworks for at least the near–future. As e–commerce transaction volume increases, there will be a corresponding increase in demand for cost–effective, quality commercial web applications. If it can be shown that tiered architectures create this kind of application, the architecture is likely to become hugely popular in the web environment.

The tiered architecture is also the type of architecture being evaluated by the CSIRO MTE project,[18] which this work is intended to complement.

2.4.2 Choosing the Number of Tiers

Choosing the number of tiers for an application is a critical decision because system structure is likely to outlive all of its component pieces of software [Fayad and Hamu 2000]. In other words, although over time tiers may be repaired, replaced or upgraded, the overall system structure is likely to remain constant. While adding a tier retrospectively to the architecture is relatively simple if the new tier is at the top,[19] it is much more difficult to add a new tier at some intermediate point, where the component interactions are more complicated.

Finding the optimal number of tiers is a question that will be considered in chapter 7. In the meantime, several tiered implementations will be briefly introduced:

One Tier One tier means that all operations are performed by a monolithic piece of software, normally on a single machine. The performance of such a system can be extremely fast since little time is wasted on interprocess communication. On the other hand, the all-in-one nature of the architecture means that it is likely to be less scalable and extensible than a more modularised architecture.

An example of a product implementing only one tier is the *Bullant Net Application Server* [20], a product that integrates functionality in order to achieve maximal transaction throughput.

The single–tier architecture was not evaluated in this thesis because it is only implemented by a small proportion of web applications.[21]

Two Tiers The two–tier architecture is discussed and evaluated in chapter 5.

The most commonly used two–tier product is *Macromedia ColdFusion*[22], a combined server and development environment that allows a data tier to be integrated into the application.

[17] [Shirky 1999]

[18] See Appendix A and Ran, Brebner, and Gorton [2001]

[19] For example a web tier to a transaction system. In this case the new tier is just a wrapper for the existing system. For an illustration of component insertion see figure 2.3.

[20] `http://www.bullant.com`

[21] The CSIRO MTE project (see Appendix A) only tests middleware products with a significant market share. There are no single tier products in this category.

[22] `http://www.macromedia.com/software/coldfusion/`

Three Tiers The three–tier architecture is discussed and evaluated in chapter 6.

The central part of a three–tier architecture is an **application server**. The application server idea and examples of commercial products are described in sections 6.3 and 6.4 respectively.

Architectures with more then three tiers are uncommon in the web context because it is difficult to derive more than three layers of functionality for a typical web application.[23] In addition, each extra tier extends the length of the longest path in the system, potentially slowing down performance since each transaction takes longer to execute. Finally, each extra tier adds complexity to the system, and a corresponding greater potential for errors.

For these reasons, two– and three-tier architectures are likely to be the best choices for web applications, and were therefore chosen as the architectures to implement.

2.5 Summary

This chapter introduced the concept of tiered architectures, and explored the properties of applications created from this type of architecture. The research goals of this thesis were then presented, along with justifications for the choice of tiered architectures, and the particular architectures that were implemented.

The next chapter completes the discussion of tiered architectures by describing the J2EE specification, which defines a method for the construction of tiered architectures.

[23] See sections 3.2 and 6.1 for a rationale of the division into three tiers. The basic idea is a division into presentation, business logic and information management tiers.

Chapter 3

Java 2 Enterprise Edition

As mentioned previously, Perry and Wolf [1992] claim that an architecture can be classified by three characteristics: "elements, form and rationale". This chapter completes the classification of tiered architectures by describing the elements and form of tiered architectures. The context for this description is the specification that was used to implement all of the evaluated applications: Java 2 Enterprise Edition (J2EE). The decision to use J2EE was made for two reasons:

- J2EE is widely used, and as a consequence products that comply with the J2EE standard are common. The results of an evaluation using J2EE are also directly applicable to real-world situations; and
- J2EE is flexible, and so one standard could be used for implementing all of the applications. This consistency was necessary for the comparison of different architecture types.

3.1 Architectures Mandated by Standards

A standard is a mechanism for regulating the "elements and form" of an architecture. However, the importance of standardisation goes beyond its purpose as a definition tool. A widely adopted standard creates a market of users encountering the same situations and problems. It becomes possible (and profitable) for vendors to produce products that address these common problems.

Such a situation has occurred in the web application market. In this domain, the common situation has been the increasing need for **enterprise systems**. As noted above, an enterprise system is one that is capable of supporting the set of functions required by an enterprise, including: transaction management, data storage, message passing, application integration and (most relevantly) web services.

The piece of software central to the enterprise systems considered in this thesis is the **application server** (see section 6.3), and although vendors were offering application server products prior to the publication of the J2EE Specification [Sun Microsystems 1999a], massive growth in the market only occurred subsequent to the adoption of the J2EE standard by most vendors.

The Java 2 Enterprise Edition was first published as the J2EE version 1.2 standard [Sun Microsystems 1999a]. The current version is 1.3 [Sun Microsystems 2001a], and substantial refinements have been made since the release of the original version. The specification has four elements:

J2EE Platform A standard platform for hosting J2EE applications.

J2EE Compatibility Test Suite A suite of compatibility tests for verifying that a J2EE product complies with the J2EE platform standard.

J2EE Reference Implementation A reference implementation for prototyping J2EE applications and for providing an operational definition of the J2EE platform.

J2EE BluePrints A set of best practices for developing multi–tier, thin–client services in a J2EE environment.

The remainder of this chapter is devoted to a more detailed discussion of J2EE. Section 3.2 gives a broad outline and section 3.3 introduces the J2EE technologies. Sections 3.4 and 3.5 describe two important J2EE ideas: container management and object deployment. Section 3.6 provides more detail on the J2EE technologies most relevant to the performance evaluation. The chapter concludes with a discussion of CORBA, a distributed communication standard.

Detailed discussion of the J2EE specification is undertaken only with respect to those aspects that relate directly to the implementation and evaluation process described in the following chapters. More discussion and further detail on topics not covered below can be found in the current release of the J2EE Specification [Sun Microsystems 2001a].

3.2 J2EE Outline

The J2EE architecture is the result of combining a tiered approach with the Model-View-Controller (MVC) design model. The MVC model uses the following three roles to decomposes an application (see figure 3.1):

Model The model maintains an application's central data structure. In a small program this could be an integer or array. In large systems it is more likely to be a database and specialised database access components.

View The view stores **Presentation Logic**. Presentation logic consists of the methods that format data in a required format for display. The view translates data into a form—normally protocol dependent—suitable for output (for example HTML, XML, PDF or plaintext).

Controller The controller is responsible for request processing and data manipulation. These functions are normally grouped into a single category called **Business Logic**.

The controller receives format–neutral requests from the view, processes requests by obtaining information from the model, and forwards format–neutral responses back to the view.

An important characteristic of the MVC model is that it is easy to accommodate multiple views and models. This implies that the MVC model encourages scalability and extensibility. The MVC model originated in Smalltalk (see Krasner and Pope [1988]) but has been adopted by Sun as the recommended design strategy for J2EE.

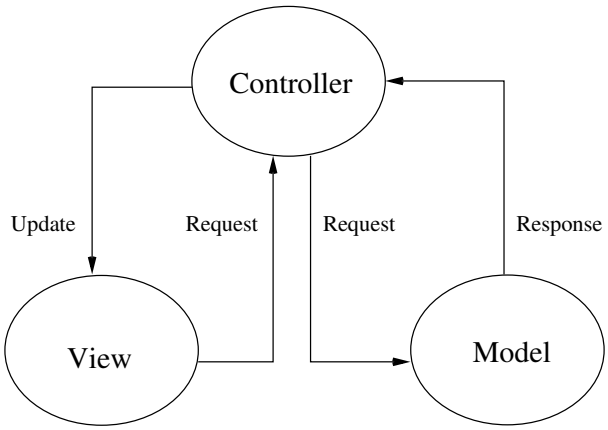


Figure 3.1: Model-View-Controller Relationships

The combination of tiers and MVC results in the three–tier architecture described by the J2EE Specification (see figure 3.2):

Web Tier The web tier is a specific type of view that is capable of listening to HTTP requests and generating HTML responses. In most cases the web tier contains a single web container.

Middle Tier The middle tier performs the controller functions described above. In most cases the middle tier contains a single EJB container (see section 3.6.2.

Enterprise Information Systems (EIS) Tier The EIS tier is a model potentially composed of a wide variety of enterprise data sources. For the sake of clarity, from this point on the EIS tier will be referred to as the database tier, since the only type of data source considered in the evaluations are databases.

There are several ways in which information can travel from the client to the database. The pathways representing the architectures implemented and evaluated in chapters 5 and 6 have been highlighted in figure 3.2.

Auxiliary services such as the Naming (see section 3.7.1), Transaction and Messaging Services are administered in the middle tier, although they run as separate processes. Such independence is necessary because the functions performed by these

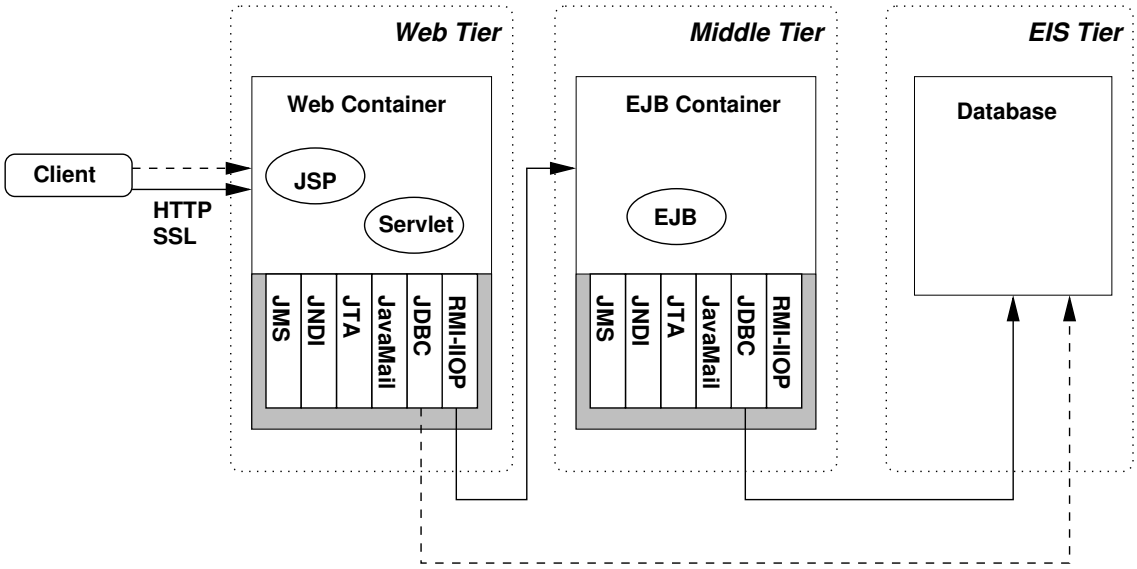


Figure 3.2: J2EE Application Architectures. Modified version of the diagram on page 16 of Sun Microsystems [2001a]. The dotted path represents a two–tier architecture, the solid path a three–tier architecture.

programs need to be able to be accessible to any part of the J2EE architecture. These auxiliary programs are also known as **enterprise services**.

3.3 J2EE Technologies

The J2EE platform includes the following technologies:[1]

HTTP / HTTPS Support for the HyperText Transfer Protocol (plain and over a Secure Sockets Layer (SSL) connection).

Java Transaction API[2] (JTA) The JTA is an application-level interface that is used by the container and application objects to manage transaction integrity.

RMI-IIOP Remote Method Invocation over Internet Inter-Orb Protocol allows objects to make method calls on remote *Java* objects. RMI is a Java technology, IIOP is part of the CORBA specification (see section 3.7 below).

Java IDL JavaIDL allows J2EE application components to invoke external CORBA objects using the IIOP protocol. These CORBA objects may be written *in any language* and typically are located external to the J2EE application.

[1]Summarised from Sun Microsystems [2001a] at pages 19-20.

JDBC API JDBC is a Sun Microsystems trademark, sometimes referred to as being an acronym for Java DataBase Connectivity. This API is described in below in section 3.6.3.

Java Messaging Service (JMS) The Java Messaging Service is a standard API for messaging that supports reliable point–to–point messaging as well as the publish–subscribe model.

Java Naming and Directory Interface (JNDI) The JNDI API is the standard API for naming and directory access. The JNDI API has two parts: an application-level interface used by the application components to access naming and directory services, and a service provider interface to attach a provider of a naming and directory service.

JavaMail JavaMail enables application components to send Internet mail.

Java API for XML Parsing (JAXP) The JAXP API provides support for programming with eXtensible Markup Language (XML) format documents.

J2EE Connector Architecture The Connector architecture is a J2EE API that allows resource adapters that support access to Enterprise Information Systems (EIS) to be plugged in to any J2EE product. The Connector architecture defines a standard set of system-level contracts between a J2EE server and a resource adapter.

Java Authentication and Authorisation Service (JAAS) JAAS enables services to authenticate and enforce access controls upon users. It implements a Java technology version of the standard Pluggable Authentication Module (PAM) framework, and extends the access control architecture of the Java 2 Platform in a compatible fashion to support user-based authorisation.

J2EE functionality is supplied by program objects such as Enterprise JavaBeans (section 3.6.2), Servlets (section 3.6.1) and JavaServer Pages. Figure 3.2 shows their respective locations in the architecture.

Sun publishes a separate specification document for each platform feature. New versions are constantly being produced, sometimes in conjunction with the Java Community Process. [3]

3.4 J2EE Containers

The J2EE specification introduces a layer of abstraction below the tier called the **container**. This abstraction is enforced by the condition that J2EE classes are not allowed to directly communicate with each other, instead a class must use the protocols and services provided by its parent container to interact with other J2EE objects.

[3] Sun's consultation program. `http://jcp.org`

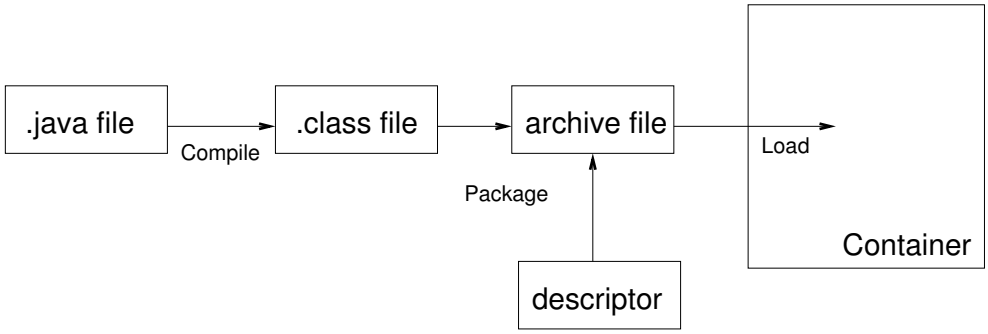


Figure 3.3: The J2EE Deployment Process

The J2EE architecture is therefore a **federation** of containers. Each container can run as a separate process or on a separate machine. Given a suitable message passing architecture, containers can be replicated and added dynamically (see section 3.7 below).

The greatest advantage of the container approach is that all outgoing and incoming communication must pass through a layer of **container management**. This means that a container can transparently supply many services that would otherwise have to be implemented by the application programmer, such as load balancing, state management, resource pools, transaction management and security checking.

Containers also insulate the application programmer from client type, since all request parsing is performed at the container perimeter. This means that application code is smaller, more portable and hence likely to contain fewer errors. An example container, Jakarta Tomcat, is described in section 3.6.1.1.

3.5 J2EE Deployment

The structure of J2EE encourages the placement of program objects into an already established architecture. For example, servlets are **deployed** to a servlet engine, EJBs are deployed to an EJB container, and databases are deployed as `DataSource` objects (see section 3.6 for descriptions of these technologies).

Getting a J2EE program object to run, however, will usually require more than one deployment step. The full process of J2EE deployment is shown in figure 3.3, where the major steps are compilation, packaging and placement.

The purpose of the intermediate packaging step is to introduce an additional layer of abstraction: the **package**, the fifth and final level of granularity specified by J2EE. This means that J2EE possesses the following abstraction levels (from largest to smallest): application, tier, container, package and class. The relationships between these levels was defined in section 2.2, and their implementation as J2EE concepts is shown in figure 3.4.

The importance of deployment in J2EE is a consequence of the practice of container

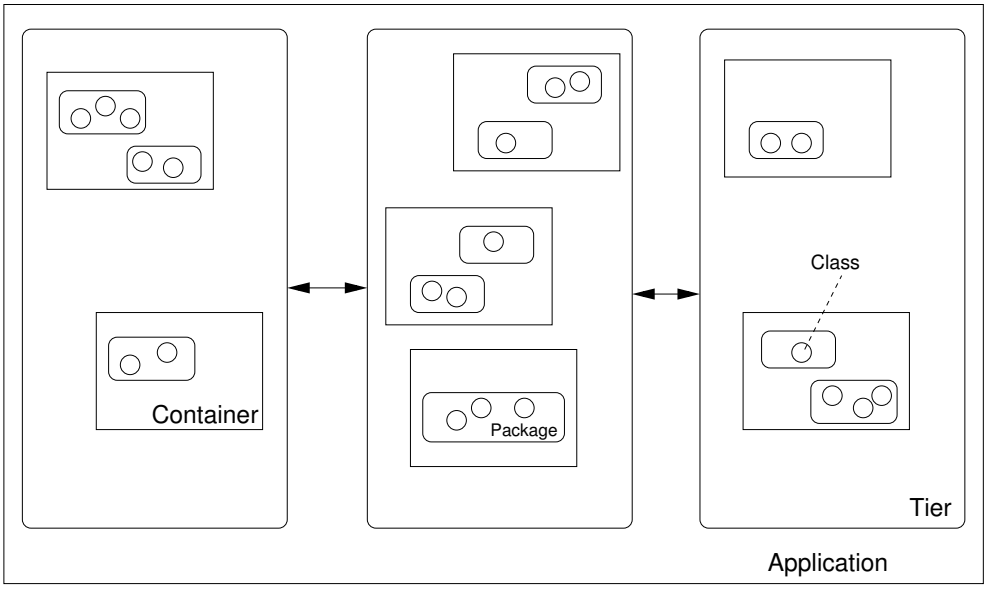


Figure 3.4: J2EE Layers of Abstraction

management (see section 3.4 above). The deployment process allows a package to tell the container what it contains, how it is structured, and what resources it is likely to require. This is all achieved through the use of an XML file called a **deployment descriptor**.

Components deployed to the servlet container are described by a file called `web.xml`. The corresponding file for EJB deployment is `ejb-jar.xml`. An example `web.xml` file is presented below:

```
<?xml version="1.0"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD 2.2//EN"
     "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
  <!-- SERVLET DEFINITIONS AND MAPPINGS -->
  <servlet>
    <servlet-name>snoop</servlet-name>
    <servlet-class>SnoopServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>snoop</servlet-name>
    <url-pattern>/snoop</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>snoop</servlet-name>
    <url-pattern>*.snp</url-pattern>
  </servlet-mapping>
```

```
  <!-- SECURITY POLICIES -->
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Protected Area</web-resource-name>
      <description>Sample security restriction</description>
      <url-pattern>/jsp/security/protected/*</url-pattern>
      <http-method>GET</http-method>
  </security-constraint>

  <!-- RESOURCES -->
  <ejb-ref>
    <description>Sample EJB Reference</description>
    <ejb-ref-name>StockBean</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>stockonline.StockHome</home>
    <remote>stockonline.Stock</remote>
  </ejb-ref>
</web-app>
```

As well as defining package structure, the XML files are important tools for managing package characteristics and performance. The advantage of using XML files for this purpose is that modifications made to the XML do not required a re–compilation of the program objects.

3.6 Implementing J2EE Applications

As described above, the recommended structure of a J2EE application is to have a web tier (defining the presentation logic), middle tier (defining the business logic) and database tier (defining the structure and management of persistent data structures). Chapter 6 evaluates implementations that maintain this separation. Chapter 5, on the other hand, evaluates applications where presentation and business logic are both implemented in the web tier.

The J2EE technologies discussed in detail are:

- The objects that implement middle tier functionality (section 3.6.2);
- The objects implement web tier functionality (section 3.6.1);
- The method that objects use to access the database tier (section 3.6.3);

Other J2EE technologies are also described, when relevant to the discussion, in later chapters.

3.6.1 Servlets

A servlet is a web component, managed by a container, that generates dynamic content. Servlets are small, platform independent Java classes compiled to an architecture neutral byte code that can be loaded dynamically

> into and run by a web server. Servlets interact with web clients via a request/response paradigm implemented by the servlet container. This request/response model is based on the behaviour of the HyperText Transfer Protocol (HTTP).[4]

Put another way, a servlet is an object located in the web tier that performs functions in response to client requests. There are many technologies that perform this kind of function, [5] however, servlets are different because they are a type of J2EE object. This means that servlets inherit all the J2EE restrictions and benefits—most relevantly the idea of container management (the Tomcat servlet container is discussed in the next section).

A servlet has a life cycle that travels through three states (see figure 3.5). Each state corresponds to one or more methods that need to be implemented by the application code:

Creation When a servlet is first loaded into a container its `init()` method is called. This provides a place to put expensive, one–off operations like resource lookups.

Service Once loaded, a servlet is notified by its container whenever a HTTP request is made for the servlet's Uniform Resource Locator (URL). There are two common HTTP request types, GET and POST, although only GET will be described here as it was the request type used in the performance evaluations.

The structure of a URL that corresponds to an HTTP GET request is shown in figure 3.6. The information contained in this request (the variable/value pairs) is passed to the `doGet()` method of the servlet.

The servlet then processes the information (possibly by calling upon external resources) and returns a HTML page as a response.

Destruction When a servlet is removed from a container its `destroy()` method is called before it is deallocated. This allows the servlet to *exit gracefully*, closing any open connections and returning any shared resources.

Servlets are the web tier technology that will be used in all of the architecture implementations evaluated in this thesis. The only alternative to servlets in the J2EE specification are JavaServer Pages (JSPs), essentially HTML pages with embedded Java code fragments called **scriptlets**. Servlets were the better choice because:

- They are easier to test. Servlet response time is just the time between a request and a response; and
- Servlets are a superset of JSPs. This is because JSPs are converted to servlets before being loaded into the servlet container. This conversion process means

[4] [Davidson and Coward 1999] at page 11.

[5] For example, scripting languages like Perl and Python, and program objects like Active Server Pages (ASPs).

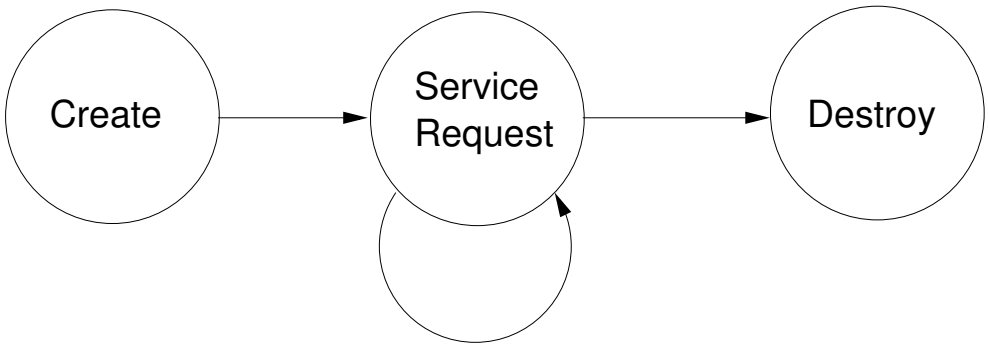


Figure 3.5: The Servlet Lifecycle

Figure 3.6: An HTTP GET Uniform Resource Locator (URL)

that JSPs are just an abstraction to allow designers to focus on the graphical interface rather than the Java code.

The greatest restriction on servlets is that they cannot run without a servlet container. This is because programs written in Java are not compiled to machine code. Instead, the Java byte code needs to be executed by a Java Virtual Machine (JVM), itself a running program.

Servlets must, therefore, run within another process. However, this also turns out to be a considerable advantage since a **servlet container** is much more than just a JVM. An example servlet container, Tomcat, is discussed in the next section.

3.6.1.1 Jakarta Tomcat

The Jakarta Project[6] is an initiative of the Apache Software Foundation. Jakarta produces Tomcat, the official servlet container Reference Implementation (see section 3.2 above).

The current Tomcat production release is 3.2, the version used in the Borland AppServer product described in section 6.4.2.[7] Tomcat is an Open Source product, and the use of its source code is regulated by the Apache License[8]. The only restrictions on

[6] `http://jakarta.apache.org`

[7] The J2EE 1.3 specification recommends Tomcat 4, however this version is not yet of commercial quality.

[8] `http://jarkarta.apache.org/LICENSE`

redistribution imposed by this license are a small number of procedural requirements (such as the preservation of copyright).

Tomcat provides many of the container management features introduced above.

Flexibility Tomcat can run either as a stand-alone program or incorporated into a web server (typically Apache). The component nature of the J2EE architecture also means that Tomcat containers can be added or upgraded within a system with minimal effects on other components.

Protocol Independence The Tomcat connector architecture allows Tomcat to listen for a variety of incoming request types. By default Tomcat listens for HTTP, HTTPS and Apache requests, and this functionality can be augmented by the addition of custom connector types.

Service Provision Tomcat can be configured to provide services like security checking, session management and DataSource management. These tasks need only be defined once for all deployed classes.

In summary, Tomcat performs two important functions:

1. It is a promoter of servlet technology. Many of the early J2EE Reference Implementation were themselves buggy or only toy applications. The Tomcat servlet container produces acceptable performance and robustness for commercial–size applications; and

2. It is a product to build upon. Tomcat has been incorporated into, or formed the base for, the servlet containers of many application server products. The availability of the source code also means that Tomcat can be tuned to run best on a particular vendor's platform.

3.6.2 Enterprise JavaBeans

> The Enterprise JavaBeans architecture is a component architecture for the development and deployment of component–based distributed business applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional, and multi–user secure. These applications may be written once, and then deployed on any server platform that supports the Enterprise JavaBeans specification.[9]

In the context of web applications, EJBs are Java objects that encapsulate all of the business logic necessary to respond to client requests. This includes request filtering as well as information retrieval and processing. The power of EJBs—and the reason that Sun could make the claims in the above definition—is that all EJBs are located

[9] [DeMichiel et al. 2001] at page 25.

inside an **EJB Container**. The benefits of containers have already been discussed in sections 3.4 and 3.6.1.1.

The type of EJBs used in all of the three-tier architectures evaluated in chapter 6 are **Stateless Session Beans**. The characteristics of this bean type are:

Session This bean may only exist for the duration of one session, although the session may consist of many transactions. A session bean does not *persist* beyond one session, so it will be garbage–collected or returned to a bean pool once the session finishes.

Stateless A stateless session bean is not capable of keeping state information on behalf of a client. Operations on stateless session beans are therefore atomic (a single request/response transaction). This is the simplest EJB component type.

The stateless session bean type was chosen since it is the fastest type of EJB. This is because the minimal overhead associated with this bean type (no state or client information) means that it can be instantiated and executed very quickly.

For completeness, the full EJB component tree is displayed in figure 3.7. A more detailed description of bean types as well as the CSIRO's StockOnline application is contained in Ran, Brebner, and Gorton [2001].[10]

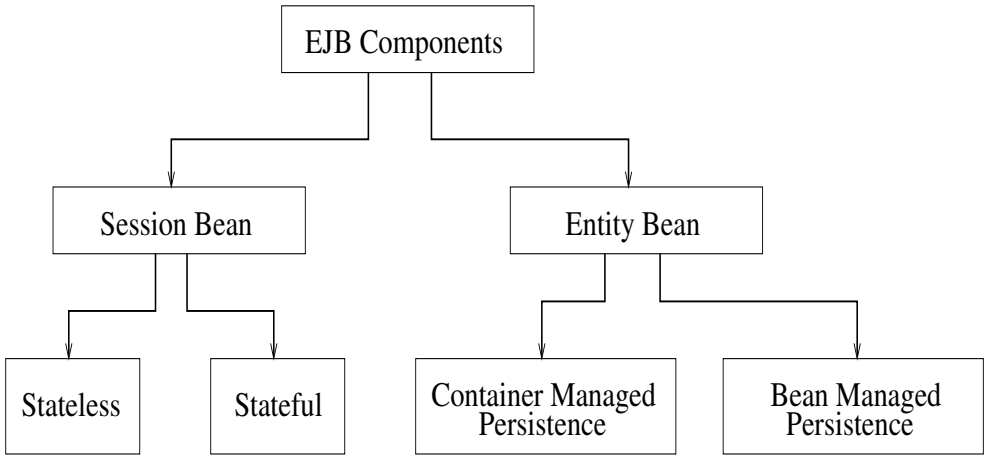


Figure 3.7: EJB Component Types

The EJB specification is currently at version 2.0 [DeMichiel et al. 2001]. However, since this specification is extremely new, the products used in the evaluation process could only be relied upon to implement the EJB 1.0 [Matena and Hapner 1998] and EJB 1.1 [Matena and Hapner 1999] specifications.[11]

[10] Also see Appendix A.
[11] Most often a product would implement a subset of each.

3.6.3 JDBC

The JDBC specification describes how Java programs can use a special type of object—called a **Database Driver**—to execute Structured Query Language (SQL) queries on relational databases.

The original JDBC version [Hamilton and Cattell 1997] was superseded by JDBC Version 2 in 1999 [Sun Microsystems 1999c]. There are important differences in the connection philosophies of the two versions, difference that are especially for performance testing.

JDBC 1.2 drivers in this version are managed by a `DriverManager` object. Any Java class that wants to access a database must:

1. register the appropriate driver with the `DriverManager`;
2. request a connection (specifying database URL, table name, user name and password);
3. execute one or more SQL statements using the connection; and
4. close the connection.

This process is shown in the following Java code fragment.

```
// 1.
Class.forName("org.gjt.mm.mysql.Driver").newInstance()

// 2.
Connection c = DriverManager.getConnection(
        "jdbc:mysql://wolseley/stockonline?user=shaun&password=pass");

// 3.
Statement stmt = c.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM Accounts");

/*** Process the ResultSet ****/

// 4.
c.close();
```

JDBC 2 The more recent version introduced an abstraction object called a `DataSource`. This class acts as a wrapper for all of the technical details required for communication with a particular database, including the driver class and the database location information.

Once deployed (see section 3.5), a `DataSource` is a distributed object. Methods can be invoked on this object once a reference to the `DataSource` has been obtained (this process is described in section 5.1).

The following code fragment illustrates using the JDBC 2 API to access a relational database.

```
// 1. Get a naming context
javax.naming.Context ctx =
        (javax.naming.Context) new InitialContext();

// 2. Retrieve a reference to the DataSource
DataSource ds =
        (java.sql.DataSource) ctx.lookup ("jdbc/StockDataSource");

// 2. Get a connection from the DataSource
Connection c = ds.getConnection("shaun", "pass");

// 3. Same as previous example
Statement stmt = c.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM Accounts");

/*** Process the ResultSet ****/

// 4. Same as previous example
c.close();
```

Application performance can be substantially improved by using JDBC 2. This subject is revisited in the Tuning section of chapter 4.

3.7 CORBA

J2EE applications are **distributable**, meaning that components can be physically and logically separated. In addition, components may be dynamically added and removed while the application is running. Ad hoc communication methods, like local pipes and sockets, cannot work efficiently in such an environment.

The solution is to introduce a communication structure that enables objects to request the location of, and communicate with, remote objects. This structure can insulate the program objects from unnecessary details, and allows general communication amongst objects without the necessity of implementing separate communication channels.

The communication structure[12] most commonly used by J2EE applications is CORBA, the Common Object Request Broker Architecture. The specification that defines the CORBA standard is produced by a not-for-profit consortium called the Object Management Group (OMG) [Object Management Group 2001a]. Nearly all major middleware, component and application server vendors are members of the OMG.[13]

The CORBA features most commonly used by J2EE classes are:

Object Location Remote object location is achieved through the combination of a Naming Service and the JNDI API. The process is explained in section 3.7.1.

[12] Communication structures regulate architectural "form".

[13] "Application server" and "middleware" are defined in chapter 6.

Object Communication Remote object communication is achieved through the combination of two technologies, Remote Method Invocation (RMI) and the Internet Inter-ORB Protocol (IIOP). This process is explained in section 3.7.2.

Both of these methods are driven by a program called an **Object Request Broker (ORB)**. The ORB

> ...is responsible for all of the mechanisms required to find the object implementation for the request, to prepare the object implementation to receive the request, and to communicate the data making up the request. The interface the client sees is completely independent of where the object is located, what programming language it is implemented in, or any other aspect that is not reflected in the object's interface.[14]

The ORB allows one class to transparently locate, and request a reference to, another class instance. As an example, a servlet can obtain a reference to an EJB, without worrying about where the bean is or how to create it. The servlet is only concerned with the services offered by the EJB.

3.7.1 Object Location

Large commercial applications require **enterprise services**: services that exist to facilitate common tasks, or to implement functions that need to be available to the whole application.

One such enterprise service is a system program whose sole purpose is maintaining a list of current object availability and location. This software is called the **Naming Service**, and J2EE objects access the Naming Service by using the Java Naming and Directory Interface (JNDI) API,[15] a part of the J2EE specification (see section 3.6). The Naming Service implementation used in the applications evaluated in this thesis is **Cos Naming**.[16]

The process involved in locating remote objects is:

1. When a remote object is deployed, it notifies the Naming Service (this process is called **registration**); and

2. A local program can then request a reference to the remote object from the Naming Service (this process is called **binding**);

The remote object that needs to be located in the case of database access is a `DataSource` object. This Java class is discussed in sections 3.6.3 and 4.1.5.

[14] Object Management Group [2001a] at page 62.

[15] [Sun Microsystems 1999b]

[16] Common Object Services Naming [Object Management Group 2001b] Also known as the CORBA Naming Service. Other popular naming services are LDAP [Yeong et al. 1995] and DNS [Mockapetris 1987].

3.7.2 Object Communication

In J2EE, communication between Java objects is achieved by using a standard called Remote Method Invocation (RMI) over the Internet Inter-Orb Protocol (IIOP). RMI was created by Sun Microsystems [Sun Microsystems 2001b], and IIOP was created by the OMG [Object Management Group 2001a].

RMI/IIOP is a technology that was jointly developed by Sun Microsystems and IBM, and allows local Java objects to make calls transparently on remote Java objects without worrying about how the communication is taking place.

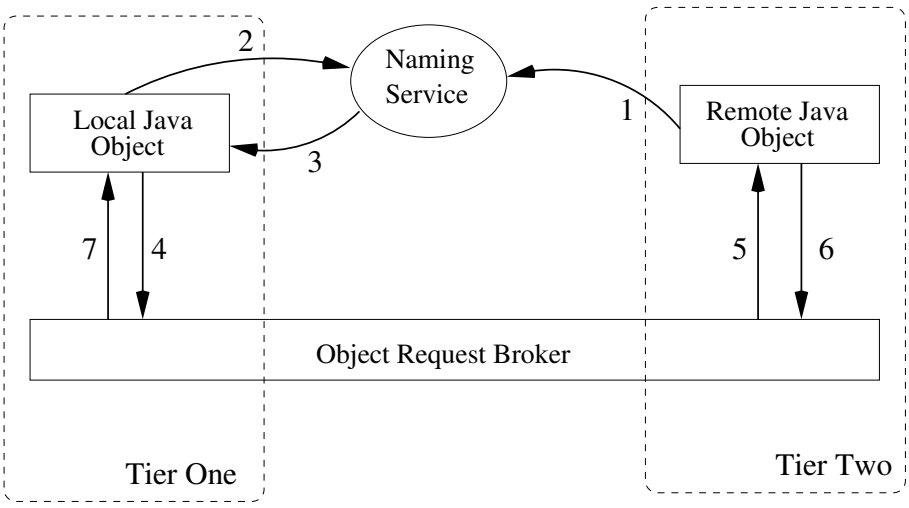


Figure 3.8: Invoking a Method on a Remote Object

The steps involved in a remote method call are (see figure 3.8):

1. The deployed remote object registers with the Naming Service;
2. The local object requests ...
3. And receives a reference to the remote object from the Naming Service;
4. The local object invokes a method on the reference ...
5. That is passed to the ORB and transported to the remote object using IIOP.
6. The remote object generates a response ...
7. Which is transported by the ORB back to the calling object.

All of the intermediate (ORB controlled) communication is invisible to the local object. In fact, the only extra API that a Java application programmer has to know about is JNDI.

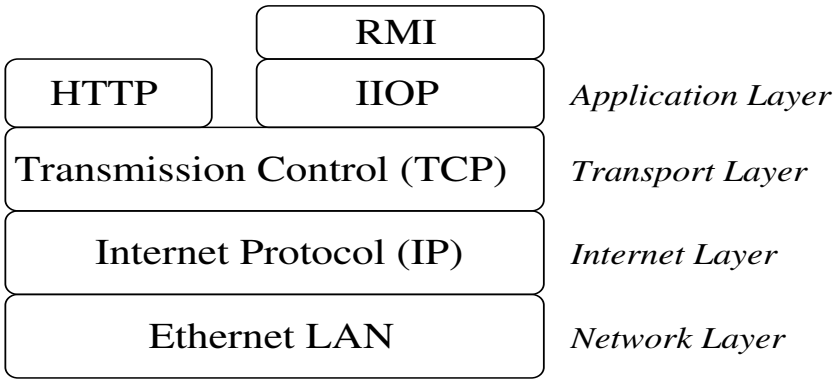


Figure 3.9: Network Protocols Used in Distributed Tiered Applications

Remote Invocation Example

The following code fragment illustrates how to use JNDI and RMI in a Java program:

```
// 1. Locate
InitialContext ic = new javax.naming.InitialContext();
Stock s = (Stock) ic.lookup(``stockonline'');

// 2. Communicate
int StockID = 10;
Price p = stock.queryByID(stockID);
```

The steps are:

1. Retrieve a reference to the remote object through a Naming Service lookup operation (**JNDI**), and;

2. Call `queryByID()` on the obtained reference (**RMI/IIOP**).

The overall aim of JNDI and RMI/IIOP is to hide as much detail as possible from the application programmer. The full protocol stack is shown in figure 3.9.

3.8 Summary

This chapter completed the description of tiered architectures by presenting J2EE, a tiered architecture specification. The elements and form of J2EE were presented, along with descriptions of the J2EE technologies relevant to the implementation of web applications.

The next chapter describes the web application evaluation method, which is applied to applications implemented using in J2EE technologies in chapters 5 and 6.

Chapter 4

The Evaluation Method

This chapter serves as an introduction to the performance evaluations conducted in chapters 5 and 6. In any testing environment the following two questions need to be asked:[1]

- **What** is being tested?
- **How** is it being tested?

The first question is answered in section 4.1, with a discussion of the tuning and standardisation strategies used to improve architectural performance and test consistency. The second question is answered in section 4.2, where the client used to test each application architecture is described.

4.1 Tuning and Standardisation

The evaluation process focuses on web applications implemented on two different products: Borland AppServer and IBM WebSphere. Any given test run would therefore involves a particular architecture combined with a particular product. In order to ensure the greatest possible standardisation across test cases, two principles must be adhered to:

1. In test runs on the same product, everything except the application structure should be kept as close to invariant as possible.

   This meant that for any given product, the same database, database driver, DataSource (see section 4.1.5), Naming Service and ORB were used. In tests that involved an application server tier, the same EJB session bean implementation was used each time. In tests without an application server tier, identical functionality was implemented at the servlet level.[2]

2. In test runs on different products, the best possible performance for a given architecture should be used in comparisons involving the architecture.

[1] The third question, "Why is it being tested?", has already been addressed in section 2.4.
[2] In other words, identical number and complexity of SQL statements.

In addition, architectural performance is considered to be consistent across products if the same performance ordering is preserved, regardless of implementation. In other words, an architecture is consistent if it performs similarly for all implementations.

The first principle, application invariance, is discussed below in section 4.1.6, and where appropriate in the evaluation chapters. The second principle, maximal performance, required the creation of a rigorous tuning process, described in the next section.

A comprehensive list of the software used in the test applications is presented in Appendix B. The two application server products, Borland AppServer and IBM WebSphere, are discussed in section 6.4.

4.1.1 Tuning Strategies

Broadly speaking, there are two types of tuning:

1. Strategies that increase **efficiency**. This results in a diminished CPU usage for the same amount of work; and

2. Strategies that increase **throughput**. This is achieved through the removal of a bottleneck in the system, normally through the reallocation of resources.

The upper bound on the amount of tuning that can be done is the processing capacity of the slowest tier. It is therefore impossible to achieve greater performance if the computers that make up this tier are operating at 100% capacity (for more details see section 2.3.1).

Before proceeding to specific approaches, it is worth noting that the solution to a performance bottleneck is not always allocating more resources to the problem. This is especially true when more than one tier is running on a single machine since one tier can only benefit to the detriment of another.

4.1.2 Connection Tuning

A connection is a communication channel that enables the exchange of information. There are two types of connections relevant to each architecture: client-application and intra-application. The constraints on throughput for these connections are:

- The number of concurrent connections that can be created between the components;

- The efficiency of the connection method; and

- The relative locations of the communicating components.

Concurrent Connections

The maximum number of concurrent connections is a parameter that can be varied for each tier. Finding the right combination of numbers for any given implementation involves guided trial and error.

The guidance in this case comes from WebSphere Tuning Guide[Cuomo 2000]. This document recommends a hierarchical connection structure, on the basis that it is better to ensure that no tier is needlessly idle even if queues form.

As an example, figure 4.1 shows one hundred clients attempting to access the application. At each stage the number of forward connections is smaller than the number of requests, so queues will form at each tier.

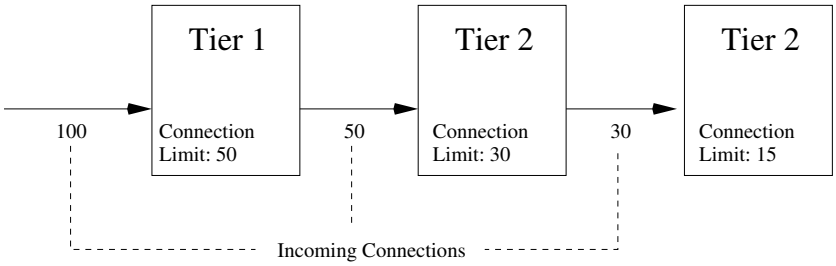


Figure 4.1: A Hierarchical Connection Structure

The bottleneck in this system is the link between the deepest two tiers, most often the middle tier and database. This structure is better then a model where each tier has the same connection limit (no bottleneck) for the following reasons:

- Requests to a web application do not necessarily make it to the bottom tier of the application. An example is a request for a static web page, which can be responded to by the web server without sending the request any deeper into the application; and
- In a more equal structure, one tier slowing down will cause all lower tiers to become idle. A hierarchical structure is better able to deal with fluctuating demand and performance since the queues mean that each tier has work to go on with while it is waiting for the higher tier to catch up.

The hierarchical structure is a good place to start when defining connection limits. However, the exact values of this limit will be highly implementation dependant.

Connection Efficiency

Connection efficiency is achieved by using an efficient protocol in an efficient manner. Only protocol efficiency is discussed in this section, since efficient resource management is discussed below under the heading of resource pooling (section 4.1.3).

In this area, vendors have to make a choice between open and proprietary protocols. Open protocols tend be improve system inter-operability and allow a greater degree of component integration. Proprietary protocols can be highly customised, possess fewer overheads and are therefore faster. When relevant, the nature of the protocol used in the test implementations is noted (see chapters 5 and 6).

Component Location

The relative location of components is also an important factor, even though the network lag time in tiered applications is negligible because the machines are geographically very close.

Component location has the greatest influence on performance in relation to the physical location of components. When more than one component is located on the same physical machine, these components can communicate using specialised methods like local pipes, without the overheads that are introduced by inter-computer communication. The result is that for small client loads a one-machine architecture is likely to be fastest, simply because of the minimal communication overheads.

On the other hand, physically separated tiers are likely to take longer to overload. This implies better application scalability. This issue of component location addressed in the course of the three–tier evaluation (section 6.7).

4.1.3 Tuning the Tiers

This section describes tuning strategies that can be applied to components.

Load Balancing

Whenever the number of tiers is not equal to the number of computers, load balancing becomes an issue.

Fewer computers than tiers implies that there is at least one machine that hosts multiple tiers. In this situation, an even distribution of work between tiers will cause the computer hosting multiple tiers to saturate more quickly than the other computers in the application.

More computers than tiers implies that work for one tier needs to be divided between multiple physical machines. In such systems, a piece of software called a **dispatcher** shares work between the computers that make up the tier.[3]

An efficient workload distribution therefore relies upon knowledge of the system's physical configuration. It also follows that modifying the physical configuration of the system will alter the efficiency of the workload distribution. The physical decomposition of the system is the subject of the evaluation in sections 5.5.5 and 6.7.5.

[3]This is a *clustered* type of system. No systems of this type are evaluated in this thesis.

Clients	Local Services	Remote Services
50	336	310
100	375	364
150	342	346
200	326	319
300	318	328
400	313	314

Table 4.1: Effect of Service Location on Transaction Throughput. These numbers were obtained during the tuning of the Borland implementations, and represent system performance in transactions per second (TPS).

Resource Pools

Resource pooling is an attempt to minimise the overheads associated with the creation and destruction of resources. The basic idea is that a **pool** of resources is maintained by a **resource manager**, and whenever a resource is required it is retrieved from the pool. Resource managers are normally associated with containers, and therefore form part of the container management strategy (see section 3.4).

Once the resource has been used, rather than being destroyed it is returned to the pool. Given that the creation and destruction of resources is generally a costly process, resource pooling provides significant performance benefits.

Commonly pooled resources are database connections and object instances such as servlets and EJBs.

Location of Enterprise Services

The Naming Service and Object Request Broker (ORB) will be frequently used throughout the test implementations for object location and communication.

This raises the issue of whether performance suffers when these services are located on a computer remote to the calling tier. As table 4.1 shows, it turns out that for large client numbers the effect of locating services remotely is minimal.

Table 4.1 displays transaction throughput for two identical two-tier implementations, where in one instance the enterprise services are run on the local machine, and in the other remotely.

Warm–up

The final strategy to increase tier performance is to allow a warm–up period prior to each test run. A warm–up consists of each client thread executing a set number of transactions before the test process commences.

The purpose of the warm-up is to ensure that as many application initialisations as possible are performed before the test commences. In particular, warm-up causes:

- Resource pools to be populated. This means objects and connections can be created and are ready to service client requests; and
- Caches to be filled. An example is the prepared statement cache for database queries (see section 4.1.5).

4.1.4 Java Tuning

Java is an interpreted language, meaning that program code is compiled to an intermediate byte-code level and then executed by a virtual machine. Java efficiency is therefore a combination of the virtual machine efficiency, and the quality of the program code.

The Java Virtual Machine (JVM)[4]

The most significant control parameter that a user has over any JVM is the **heapsize**. This value is the size of the memory heap in which the JVM operates. The heapsize is controlled by command-line arguments. For example the command

```
java -Xmx256m -Xms128m <Java class>
```

invokes the JVM with an initial heap size of 128 megabytes and a maximum heap size of 256 megabytes.

Heapsize impacts on performance because program execution has to pause whenever the heap becomes full. The JVM then invokes the **garbage collector** to reclaim any unused memory.

With a large heap there are large intervals between garbage collector invocations, however, the garbage collection process takes more time (see figure 4.2). This is because the time that one garbage collection takes to execute is proportional to the size of the memory space that needs to be examined.

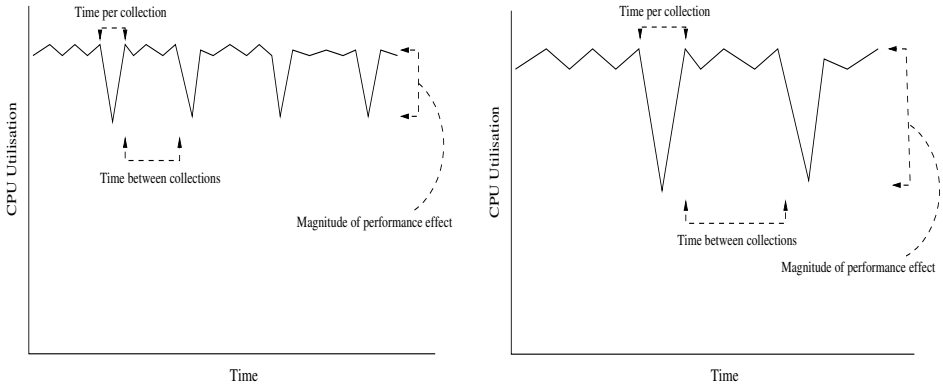
As the evaluation process involves many concurrent requests, a large heapsize is generally more desirable because a large heap allows a container to create many object instances, thereby decreasing the time a request needs to queue before being dealt with.

Tuning the Code

Good programming practices can also help overall performance. The trade-off in this section is between optimised code and understandable code. The first type may run faster, however, if the code breaks it will be more difficult to fix.

An important consideration for web applications is that once a class is written it will be executed thousands of times. This means that an incremental saving at development time can create significant performance benefits at run time.

[4]**The JVM versions used in the evaluation are listed in Appendix B.**



(a) Many, small garbage collections (b) Few, large garbage collections

Figure 4.2: Impact of Heap Size on Performance

A few of the more relevant good coding practices are listed below (for a comprehensive treatment see Gunther [2000]):

- **Perform Expensive Operations Once** J2EE objects generally have an initialisation method that is called only once during the lifetime of a particular instantiation [5].

  The `init` method is a good place to group expensive operations such as a resource lookup. The resource pooling policy means that an object may serve thousands of requests but only be initialised once. Therefore, anything that can be put in the initialisation method will create long–term performance gains.

- **Release Resources** Resource pools work most effectively when there is a rapid turnover of resources. If clients hang on to resources then the container is forced to either wait or create new object instances in order to respond to incoming requests.

  In the code this means that shared resources should be used for as long as necessary and no longer. An example is a database connection, which should be kept open for as little time as possible.

- **Use Print Statements Sparingly** Print statements require the application to pause while an output stream is acquired. Again, one print statement will become a serious issue if the statement is printed every time an object instance is executed.

4.1.5 Tuning the Database Connection

This section describes tuning strategies that can be applied to database connections.

[5] In EJBs it is called `create()` and in servlets it is called `init()`.

DataSources

DataSources were introduced in the JDBC 2 specification in an attempt to simplify and streamline the process of communicating with a database through the JDBC API3.6.3.

A DataSource is essentially a wrapper class that exists as an object in the Naming Service, and can therefore be located. Once located, methods (such as `getConnection()`) can be invoked on a remote object.

DataSources can wrap around a variety of functionality, including (see figure 4.3):

- a prepared statement cache (see below);
- a database driver. This is the piece of software that actually communicates with the database;
- a database connection pool; and
- database connection information. Including the database URL, table name, user name and password.

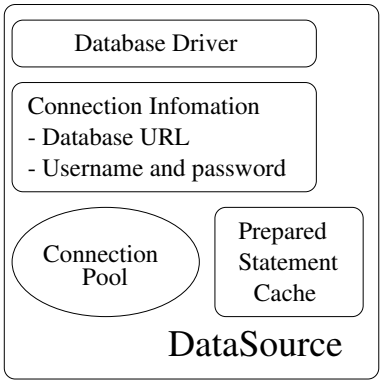


Figure 4.3: The Java `DataSource` Object

All of this information can potentially be changed dynamically, although support for this feature varies between different products.

The important aspect of DataSources is that they hide complexity from the calling program. DataSources are simpler to use and much quicker then the JDBC `DriverManager` class (see section 3.6.3).

DataSource Implementation

An important consistency requirement was that all tests with the same product should use the same DataSource object. This was found to be an issue in Borland AppServer, which offered two types of DataSource. The performance differences between the DataSource types is shown in table 4.2.

Clients	Type 1	Type 2
100	320	345
150	316	343
200	317	357
300	319	363

Table 4.2: Performance comparison of Borland DataSources. These numbers were obtained during the tuning of the Borland implementations, and represent system performance in transactions per second (TPS).

Clients	Prepared Statements	Normal Statements
100	364	311
150	346	308
200	319	310
300	328	308

Table 4.3: The Effect of Prepared Statements on Application Throughput. These numbers were obtained during the tuning of the Borland implementations, and represent system performance in transactions per second (TPS).

Prepared Statements

All statements must be compiled before they are executed by the database. Prepared statements are a mechanism to minimise the overhead associated with compilation by ensuring that compilation only occurs once. This strategy, performing expensive operations once, is also used by resource pools (section 4.1.3).

Prepared statements are statements that have been compiled ahead of time, and can be immediately executed with a set of supplied arguments. The DataSource works out which statements to prepare by maintaining a **prepared statement cache**, a record of the prepared statements that have so far been connected.

When a client program uses a connection to execute a prepared statement, the DataSource checks to see if the statement is in the cache. If it is, execution is performed immediately. If it isn't, the statement is compiled and added to the cache.

Prepared statements are especially useful when the same queries are repeated many times with different arguments, a common situation for web applications.

The performance benefits that can be obtained through the use of prepared statements are shown in tables 4.3 and 4.4. These tables compare transaction throughput and database CPU utilisation respectively, with prepared statements performing best in both cases. These figures show that the not only does the use of prepared statements make the application perform better, it also makes the database is much more efficient. This means that the database tier can accept a much higher number of concurrent requests before reaching its processing capacity.

Clients	Prepared Statements	Normal Statements
100	56	82
150	59	81
200	60	84
300	61	84
400	60	84

Table 4.4: The Effect of Prepared Statements on Database Efficiency. These numbers were obtained during the tuning of the Borland implementations, and represent the database machine's CPU utilisation percentage.

4.1.6 Standardisation

Most standardisation issues will be discussed as they are introduced in the evaluation process. The purpose of this section is to briefly discuss a couple of issues that have global relevance to application standardisation.

4.1.6.1 Compliance with Standards

In J2EE, as elsewhere, the great thing about standards is that there are so many to choose from. Vendors are often at different stages of standards compliance, and any given product may implement an older standard, a more recent standard or (more commonly) parts of both.

In theory, J2EE objects should be able to be written once in order to run in any certified J2EE container. This is not the case in practice, partially due to this checkerboard of standards compliance.

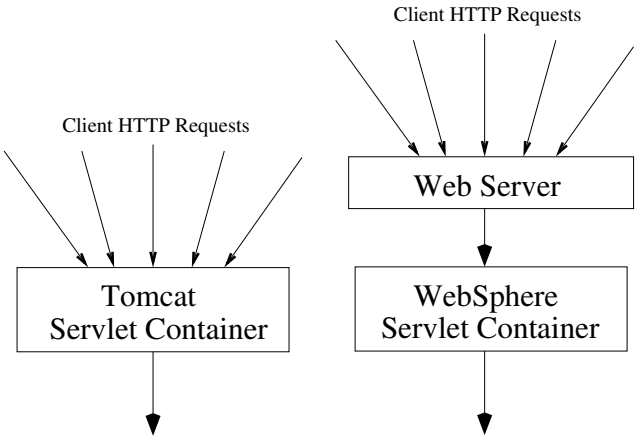
4.1.6.2 Proprietary Vendor Features

The other main reason J2EE objects are not "write once, run anywhere",[6] is that vendors add proprietary "features" to their products. These features are normally performance tweaks, so using them will increase application performance. The price paid for this extra performance is compatibility—not a huge concern for the vendor since now the customer has an incentive to keep using their product.

The origin of this problem is that J2EE is a specification that is heavily focused on the idea of container management (see section 3.4). Unfortunately, different vendors have different definitions of what *management* should mean. In addition, J2EE specifications are often unclear, forcing vendors to resolve the ambiguities on their own.

One example of this divergence is in relation to how different products deal with inter-container communication. In their WebSphere product, IBM has chosen to link the middle and web tiers with a proprietary protocol called OSE.[7] Borland, on the

[6] A Sun Microsystems trademark.
[7] Open Servlet Engine.



(a) Borland AppServer (b) IBM WebSphere

Figure 4.4: Different Approaches to Processing HTTP Requests

other hand, has implemented standard CORBA object calls between these tiers for their AppServer product.

Another example is the different methods that are used to process HTTP requests. WebSphere requires integration with a specialised web server, which passes requests on to the servlet container. By contrast, Borland AppServer allows, but does not require, this integration since it is possible to route requests directly to the Tomcat servlet container (see figure 4.4).[8]

The lesson learnt from these standardisation issues was that care needs had to be taken to ensure that appropriate architectures were compared to each other.

4.2 Client

The client described in this section is used in the performance testing for all of the architectures that are evaluated for this thesis.

The client is called `WebClient`, and it was designed to replicate the functionality of the `StockClient` described in Ran, Brebner, and Gorton [2001]. There were two reasons for this replication:

1. To ensure consistency with the tests that had already been conducted by the MTE Project team; and

[8]This issue will be returned to in the evaluation chapters.

2. Since the `StockClient` already represented a concerted effort at developing a realistic and varied transaction mix (supported by an existing implementation of the corresponding database schema), this effort could be reused in an evaluation scheme with similar testing goals.

4.2.1 Client Functionality

`WebClient` was implemented as a multi-threaded Java program. Each thread attempts to complete a predefined mix of transactions (see below), and all threads run concurrently. The client requires two arguments:

- **Number of Threads** The number of client threads to create. These threads run concurrently, simulating parallel user accesses; and
- **Uniform Resource Locator (URL)** The URL corresponding to the location of a servlet. For example the URL `http://wolseley/ProcessServlet` points to a servlet called `ProcessServlet` on the machine `wolseley` (see also figure 3.6).

4.2.2 Transaction Detail

The `WebClient` performs seven types of transaction:

`BuyStock` Attempt by a client to purchase the given stock. A client cannot buy a stock if she cannot afford the purchase.

`SellStock` Attempt by a client to sell the given stock. A client cannot sell a stock that he does not own.

`QueryByID` A request for the current price information of the stock specified by `stockID`, the stock's primary key.

`QueryByCode` A request for the current price information of the stock specified by `stockCode`, the string corresponding to the stock.

`GetHoldings` A request for the stock holdings currently owned by the given account number. This query only returns the first twenty holdings, formatted as a list of `stockID`s and `amount`s.

`CreateAccount` Creates a new account based on the supplied personal information. A new account number is assigned to the created account.

`UpdateBalance` Updates the current account balance of a specified account.

These transactions are summarised in table 4.5, and discussed further in Appendix A.

Transaction	Arguments	Returns
`BuyStock`	`AccountNumber, StockID, Amount`	Success or failure.
`SellStock`	`AccountNumber, StockID, Amount`	Success or failure.
`QueryByID`	`StockID`	Price of the stock with `StockID`.
`QueryByCode`	`StockCode`	Price of the stock with `StockCode`.
`GetHoldings`	`AccountNumber`	Return the first twenty holdings of the customer with `AccountNumber`.
`CreateAccount`	`Name, Address, Balance`	Success or failure.
`UpdateBalance`	`AccountNumber, Balance`	Assigns `Balance` to the account with `AccountNumber`.

Table 4.5: `StockOnline` Transactions

4.2.3 Transaction Mix

Each client thread executes a 43-item transaction mix (see table 4.6) ten times. The total number of transactions executed in a test run is therefore 430 times the number of clients.

The transaction mix was derived by the CSIRO to emulate the variety and frequency of transactions encountered by a commercial web site. The **type** of each transaction represents the level of database interaction that it requires.[9]

The transaction mix is 81 percent read-only (browse) operations, five percent write-only (update) operations, and 14 percent more complicated (buy or sell) operations.

Transaction	Repetitions	Type
BuyStock	3	ReadWrite
SellStock	3	ReadWrite
QueryByID	15	ReadOnly
QueryByCode	15	ReadOnly
GetHoldings	5	ReadOnly
CreateAccount	1	WriteOnly
UpdateBalance	1	WriteOnly

Table 4.6: The `StockOnline` Transaction Mix

Each test is performed multiple times, and the database is initialised (see section A.1.1) before the start of each run.

[9] See section A.2.1.

4.2.4 Performance Measurement

Four measurements are recorded for every test run:

Throughput Throughput is a measure of the number of transactions the application processes in a given time, and is calculated in the following way:

$$transactions = 10 \times 43 \times numClients \tag{4.1}$$

$$throughput = transactions \div \frac{clientTime}{numClients} \tag{4.2}$$

Where

- *numClients* is the number of clients being run concurrently;
- *clientTime* is the sum of the client execution times. $clientTime \div numClients$ is the average time is takes for a client to execute; and
- *throughput* is measured in transactions per second (TPS).

Response Time The `WebClient` records the amount of time that it takes for each transaction to finish (this is the time difference between the request of a URL and the response from the web server).

These accumulated numbers are used to produce an average response time for each transaction type.

Refused Connections A large number of concurrent access attempts can cause the application to start refusing connections. There are two types of Java Exceptions associated with these connections:

- **`ConnectException`** Occurs when the current connection times out.
- **`NoRouteToHostException`** Occurs when the server starts to refuse all incoming connection requests.

The `WebClient` records whenever a connection fails. It then waits for half a second and retries the same transaction. The percentage of refused connections is output once the test finishes.

CPU Utilisation The Windows NT Performance Monitor is used to monitor the CPU usage of each machine while the tests are running. The average CPU usage for each test—as displayed by the Performance Monitor—is then recorded.

CPU usage indicates whether an implementation is saturated (see section 2.3.1) or not, and how hard the tiers in the application are working relative to each other.

Figure 4.5 shows the Performance Monitor in operation. Information that can be drawn from this illustration includes:

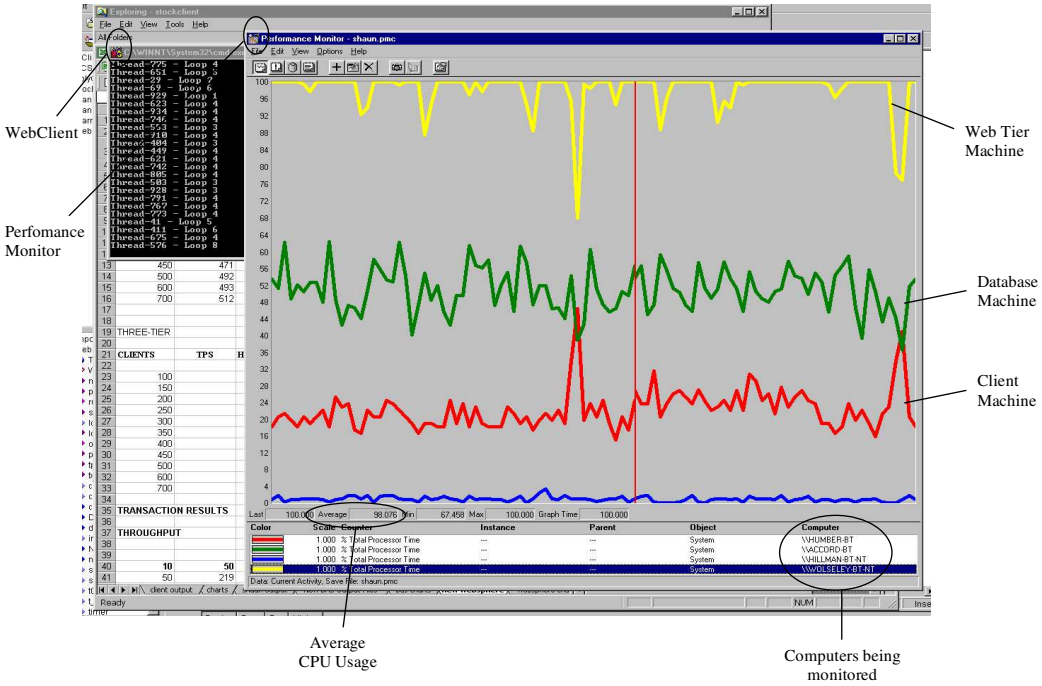


Figure 4.5: Windows NT Performance Monitor Screen Shot

- The architecture being tested is two tiers across three computers;
- The web tier machine is operating at 100% of capacity, and therefore the application is saturated (the CPU utilisation dips represent garbage collection operations); and
- The test information in the background indicates that this test is almost halfway through (at iteration 4 of 10).

4.3 Summary

This chapter explained the web application evaluation method, which included a comprehensive tuning methodology and a description of `WebClient`—a program for the evaluation of web applications. The `WebClient` is used in the next two chapters to evaluate two types of tiered implementations: two–tier (chapter 5) and three–tier (chapter 6).