Improving Open Web Architectures

Michael Collins

B.Sc. (Hons)

A dissertation submitted to the University of Dublin,

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science

September 2000

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed: _____

Michael Collins

15 September 2000

Permission to lend and/or copy

I agree that Trinity College Library may lend or copy this dissertation upon request.

Signed: _____

Michael Collins

15 September 2000

Abstract

When people use the Internet today, they use their browsers to connect to a web server located anywhere in the world and download a specified page that they have requested. Unless this page contains a Form, CGI-Script, Java Script or a Java Applet (providing of course the user's browser supports Java), there is no other way the user can interact with the web page. Even with this interaction, the user is still physically unable to edit the page itself so the source HTML code for the page can never be edited remotely.

Jiki is an open web architecture that will allow anyone to edit pages freely on the web. The system is written as a set of distributed Java components that communicate with HTTP. Although Jiki offers an adequate solution for editing in a permissive environment, its lack of security (users can edit web-pages and manipulate them freely without any restrictions or version control) means that they can place not only text, but also images, sounds and perhaps scripts that may not be desired by the "owner" of the page. The objective of this dissertation is to add authenticity and trust to Jiki. This involves designing, developing and integrating a security framework into the Jiki architecture.

Acknowledgements

I would like to thank my supervisor Dr. Simon Dobson for all the help and effort he has given me during the course of this dissertation. He gave me some great ideas and encouraged me all the way. Simon, thanks for everything. I appreciate all your help.

I would also like to thank Joe Kiniry at Caltech University, California for all the help and advice he gave me throughout the year regarding Jiki. Joe is the designer and one of the developers of Jiki. His help and suggestions were gratefully appreciated. Thanks for your input Joe. Thanks also to the Computer Science Department in Trinity, particularly those in DSG. Their help and advice regarding certain areas of this dissertation made a big difference.

Many thanks to all my family for all their encouragement and understanding during the year. Their support was invaluable and I'm truly grateful for everything they did.

Finally, to all my M.Sc. class friends. I have never met a greater bunch of people in all my life. The many times of laughter and joking we all shared really helped get through the year. I will always look back at the year we spent together with terrific memories.

Table of Contents

Tables and illustrative materials

“"I have travelled the length and breadth of this country and talked with the best people, and I can assure you that data processing is a fad that won't last out the year."

— The editor in charge of business book for Prentice Hall, 1957

1. Introduction

1.1 Introduction

The use of the Internet today has facilitated a massive growth in almost every industry. People all over the world are now able to connect to the Internet using a range of different techniques from the traditional personal computer to the personal digital assistant (PDA). There are now virtually no demographic or hardware restrictions to accessing data on the Internet.

Making data available over the Internet is a relatively straightforward process. The data must be stored on some machine running a web-server. This web-server can be connected to the Internet via an Internet Service Provider (ISP) using its own unique *IP address* or it can be connected to a network that is itself connected to an ISP. Each machine connected to the network will have its own IP address. A Domain Name Service (DNS) running on some machine will then map this IP address to some domain name (or *address*) unique to the web-server. This means that any data stored on the web-server to be made available over the Internet will contain a unique *sub-address* to the web-server *address*. This hierarchy of unique *addresses* and *sub-addresses* means that any person can enter a specific combination of addresses and download the data stored there.

It is the primary responsibility of the web-server where the data is stored to handle all the requests for that data. When a request arrives from a client, the web-server will read the request and see what the client is requesting. Assuming there are no errors in this request and no authorisation is required, the web-server will fetch the data and send it back to the client. This is the fundamental operation for all web-servers and there are several different types of them.

Each different type of web-server has its own unique characteristics and functionality. However, there is one service that very few of them provide – the ability to allow clients to change the data they requested ***dynamically on a form*** and send these

changes back to the web-server for others to download. One such web-server that does provide this service is called *Jiki*.

Jiki is a web-server that allows clients to request data, download that data and allow them to edit the data freely. When finished, the client can save the changes they made and the changed data is then stored back on the web-server for other clients to request and download. There are huge advantages for allowing this type of service. Educational, Scientific and Business organisations could benefit enormously from allowing data be changed like this over the Internet. However, there is one major downside to a service like this – the need for proper **security**.

1.2 Objectives

The lack of security with Jiki by allowing *any* client to edit data they download and have these changes saved back on the web-server can lead to very serious security problems. The potential for malicious damage is enormous. This is something most people would like to avoid.

The objectives of this project are to take the Jiki architecture and add accountability and trust to it. This will involve designing, developing and integrating new components that will add different levels of security to the architecture (e.g. to grant access to certain people to allow them edit certain pages in a certain way). Once security has been addressed, extra functionality will be layered onto the new, more secure architecture.

What is hoped to achieve is a secure, accountable and trustworthy web-server that will provide a range of security services. These services will include assigning read and edit privileges to clients for Jiki web pages, a security policy manager, the ability to create new Jiki web pages and some kind of registration process for clients to register themselves with Jiki.

1.3 Roadmap

The layout and contents of each chapter in this dissertation are as follows:

Chapter 2

Chapter 2 discusses the literature survey conducted for the dissertation. This includes a detailed look at the architecture of the Jiki web-server and the technologies used to implement it. The survey includes an in-depth description of HTTP 1.1, which is the underlying transport protocol for Jiki. Several other technologies are looked at including the possible advantages they may provide in the implementation of the new security framework.

Chapter 3

Chapter 3 discusses the design of the new security framework to be integrated into the Jiki architecture. The chapter gives a description of the security scenarios that the new security framework must be able to deal with. This includes describing how clients can be authenticated and different types of permissions they may be assigned. A discussion on the encryption algorithm to be used and reasons for using this concludes the chapter.

Chapter 4

Chapter 4 describes the implementation of the new security framework. It first gives a short discussion on the mechanisms used to implement the current Jiki architecture. Following this, a description of the use of *components* to implement the new security framework is given. This includes describing how these components are used to implement the different security scenarios that were identified and explained in the previous design chapter. The chapter concludes with explanations of the layout of the different GUIs for the security framework.

Chapter 5

Chapter 5 discusses the evaluation of the newly integrated security framework. It gives a critical analysis of the usefulness and worthiness of the system and how well it has integrated with the original Jiki architecture.

Chapter 6

Chapter 6 gives the conclusions of this dissertation. It discusses the final conclusions gained from completing this dissertation and looks at improvements that can be made to the security framework. It concludes with suggestions of possible directions that extra research may be carried out with Jiki.

Chapter 7

Chapter 7 contains the bibliography for all book and URL references used in this dissertation.

Chapter 8

Chapter 8 shows the GUI screen-shots of the new security framework.

1.4 Summary

This chapter discussed the idea and concept behind this dissertation. It introduced the Jiki web-server and the unique service it provides by allowing changes be made dynamically by clients to web pages it hosts. The chapter discussed the objectives of the dissertation and the requirements of a security infrastructure to be integrated into Jiki.

2. Literature Survey

2.1 Introduction

In this chapter, the technologies that encompass the existing web architecture will be investigated. Current research technologies of improving this architecture, namely Jiki, will also be investigated. The areas that are examined are Hypermedia, HTTP and the ability to perform updates on the web and Web Security. A discussion of the ways in which security has been integrated into one commonly used web server, namely Apache™, will follow this. The chapter concludes with short descriptions on XML and Servlets. These are areas that are core to understanding how the current web architecture operates before any improvements can be considered and proposed.

2.2 Hypermedia

> *"For many people, the most common experience of hypermedia is the World Wide Web" [Lowe99]*

There are many interpretations of what hypermedia is but [Lowe99] states that hypermedia is an application that allows a person to navigate through an information space using associative linking. Hypermedia is a conjunction of hypertext and multimedia and has provided an effective way of improving the use rather than the provision of information. One of the most significant characteristics of hypermedia applications is *non-linearity* (i.e. there are multiple possible paths through the information as opposed to just one as in a book or film). Hence the term *Hyper*media. Another characteristic includes the use of multiple types of media and different ways of accessing these, hence the latter part of the term Hyper*media*. [Lowe99] states that

there are many definitions of exactly what hypermedia is. A good definition is given as

Hypermedia 'An application which uses associative relationships among information contained within multiple media data for the purpose of facilitating access to, and, manipulation of, the information encapsulated by the data'.

2.2.1 Goals of Hypermedia

According to [Lowe99], there are three main goals that have been set out for hypermedia. These are

- To support (using the associative relationships between information sources) the carrying out of actions which result in the identification of appropriate information (with appropriateness being based on a given set of contextually defined criteria)
- To support the carrying out of actions which facilitate the effective use of information
- To support the carrying out of actions which result in control of appropriate information

In summary of the above goals, hypermedia applications should support the carrying out of actions, which result in the identification, effective utilisation and control of appropriate information.

2.2.2 Hypermedia engineering

Thirty years ago, proper structuring techniques barely existed in software development. It was at much the same stage that hypermedia development is at now [Lowe99]. Since then, software engineering has evolved into a significant sub-discipline of computer science. Many of the problems that triggered this evolution are similar to those now becoming increasingly significant in hypermedia development.

It is clear that a structured hypermedia engineering approach is required. This should be carried out in a way that is consistent with both an approach designed to most effectively yield results (an engineering approach) and the goals of hypermedia (managing information using associative linking) [Lowe99].

Different process models can be designed to suit different types of development, which in turn will be suited to different types of applications. Examples would be where a model incorporating iterative refinement of an initial prototype may be best suited to small scale applications, whereas educational applications probably require a model that considers the desired learning objectives. Correct hypermedia engineering would review all models before choosing the most suitable and appropriate one for each different problem.

The field of hypermedia development is still very young and although every hypermedia application development involves some form of engineering process, there has been little formalising of this process to date [Lowe99].

2.3 HTTP

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems [W3C1]. HTTP has been in use by the World Wide Web global information initiative since 1990 and the first version of HTTP, referred to as HTTP/0.9, was a simple protocol for raw data transfer across the Internet. HTTP/1.0, as defined by RFC 1945 [W3C2], improved the protocol by allowing messages to be in the format of MIME-like (Multipurpose Internet Mail Extensions) messages, containing meta-information about the data

transferred and modifiers on the HTTP request and response messages. HTTP is also used as a generic protocol for communication between user agents i.e. a client that initiates a request, and proxies / gateways to other Internet systems.

2.3.1 HTTP operation

The HTTP protocol is a request / response protocol. A client sends a request to the server in the form of a request method, URI (Uniform Resource Identifier), and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content over a connection with a server. The server responds with a status line, including the message's protocol version and a success or error code, followed by a MIME-like message containing server information, entity meta-information, and possible entity-body content.

Most HTTP communication is initiated by a user agent and consists of a request to be applied to a resource on some origin server. In the simplest case, this may be accomplished via a single connection (V) between the user agent (UA) and the origin server (O) (refer to **Figure 2.1**).

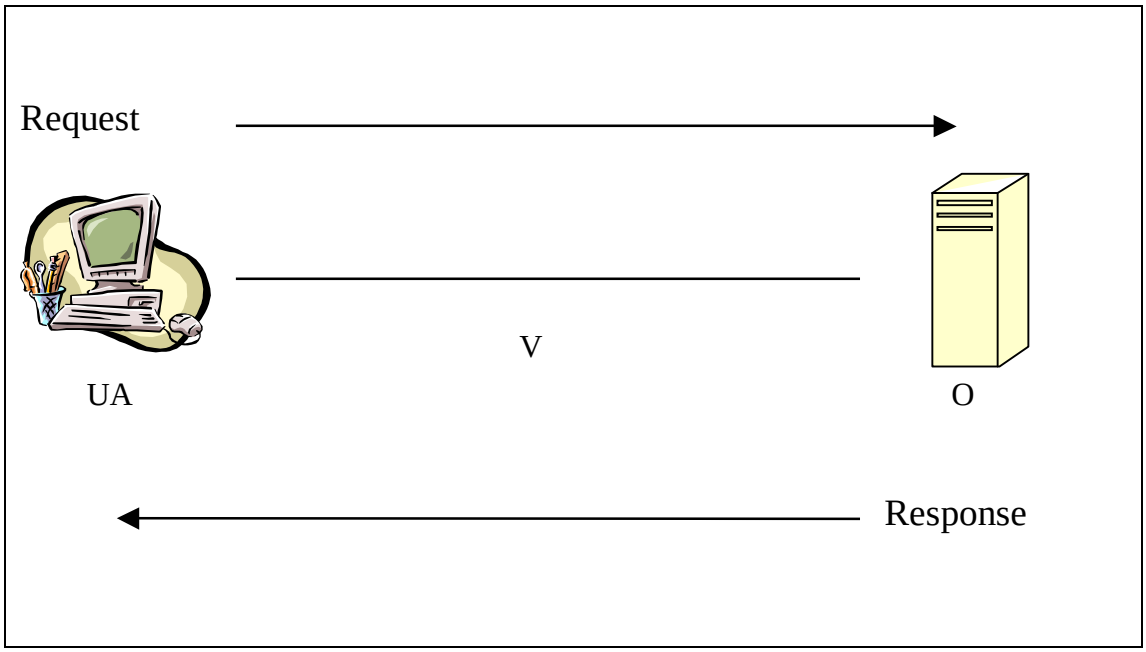

Figure 2.1 HTTP Communication

A more complicated situation occurs when one or more intermediaries are present in the request / response chain. There are three common forms of intermediary:

1. Proxy

A proxy is a forwarding agent, receiving requests for a URI in its absolute form, rewriting all or part of the message, and forwarding the reformatted request toward the server identified by the URI.

2. Gateway

A gateway is a receiving agent, acting as a layer above some other server(s) and, if necessary, translating the requests to the underlying server's protocol.

3. Tunnel

A tunnel acts as a relay point between two connections without changing the messages. Tunnels are used when the communication needs to pass through an intermediary (such as a firewall) even when the intermediary cannot understand the contents of the messages.

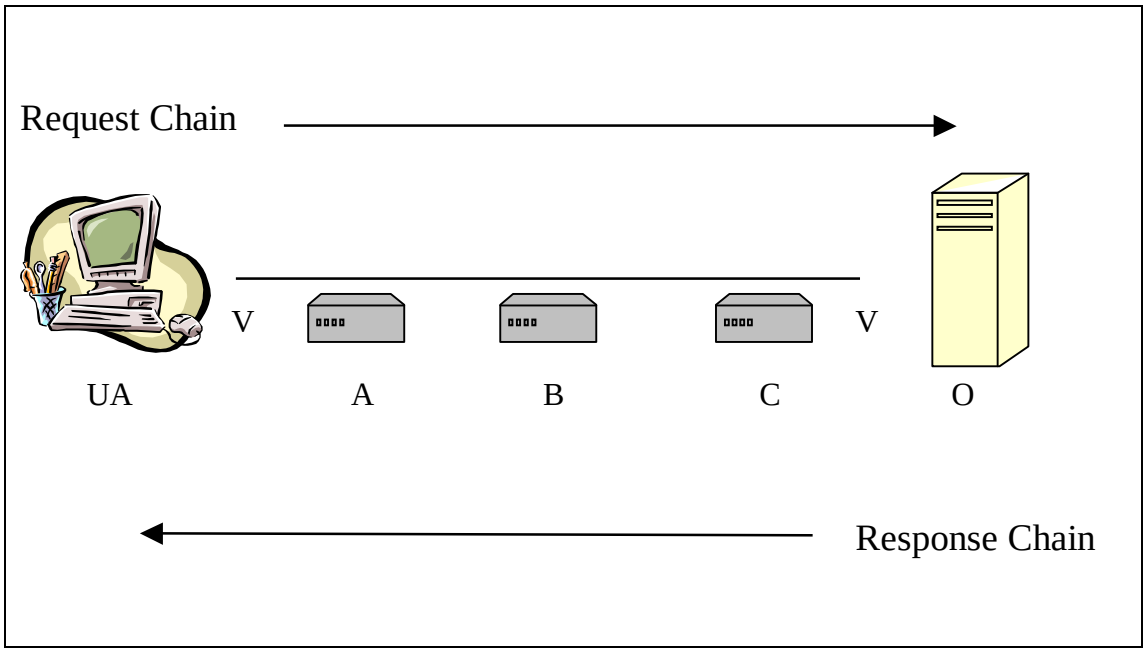

Figure 2.2 HTTP Communication with three intermediaries

Figure 2.2 shows three intermediaries between the user agent and origin server. A request or response message that travels the whole chain will pass through four separate connections. This distinction is important because some HTTP communication options may apply only to the connection with the nearest, non-tunnel neighbour, only to the end-points of the chain, or to all connections along the chain. Although the diagram is linear, each participant may be engaged in multiple, simultaneous communications. For example, B may be receiving requests from many clients other than A, and / or forwarding requests to servers other than C, at the same time that it is handling A's request.

Any party to the communication that is not acting as a tunnel may employ an internal cache for handling requests. The effect of a cache is that the request / response chain is shortened if one of the participants along the chain has a cached response applicable to that request. **Figure 2.3** illustrates the resulting chain if B has a cached copy of an earlier response from O (via C) for a request, which has not been cached by UA or A.

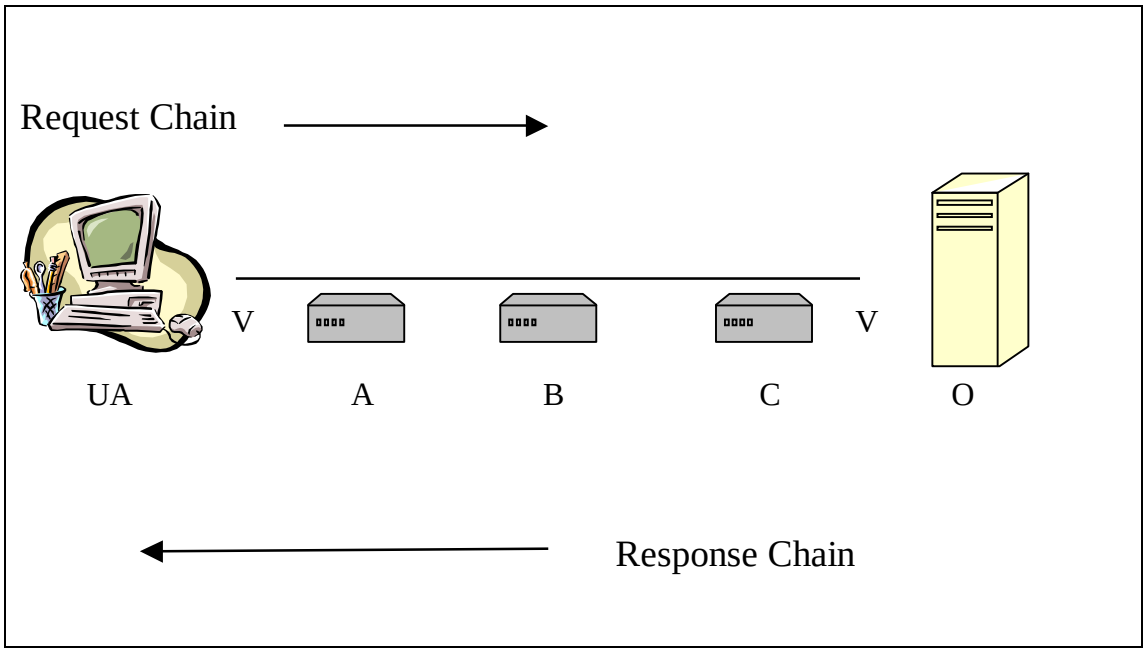

Figure 2.3 HTTP Communication with caching

Not all responses are usefully cacheable, and some requests may contain modifiers, which place special requirements on cache behaviour.

HTTP communication usually takes place over TCP/IP connections. The default port is usually 80 [W3C1], but other ports can be used. This does not preclude HTTP from being implemented on top of any other protocol on the Internet, or on other networks. HTTP only presumes a reliable transport and any protocol that provides such guarantees can be used.

2.3.2 HTTP Protocol Parameters

2.3.2.1 HTTP version

HTTP uses a "<*major*>.<*minor*>" numbering scheme to indicate the different versions of the protocol. This protocol versioning policy is intended to allow the sender to indicate the format of a message and its capacity for understanding further HTTP communication, rather than the features obtained via that communication. There are no changes made to the version number for the addition of message components which do not affect communication behaviour or which only add to extensible field values. The <*minor*> number is incremented when the changes made to the protocol add features which do not change the general message parsing algorithm, but which may add to the message semantics and imply additional capabilities of the sender. The <*major*> number is incremented when the format of a message within the protocol is changed [W3C1].

A HTTP-version field indicates the version of a HTTP message in the first line of that message

```
HTTP-Version   = "HTTP" "/" 1*DIGIT "." 1*DIGIT
```

2.3.2.2 HTTP URL (Uniform Resource Locator)

The "http" scheme is used to locate network resources via the HTTP protocol. The scheme-specific syntax and semantics for http URLs are as follows:

```
http_URL = "http:" "//" host [ ":" port ] [ abs_path [
                    "?" query ]]
```

“If the port is empty or is not given, port 80 is assumed. The semantics are that the identified resource is located at the server listening for TCP connections on that port of that host, and the Request-URI for the resource is the *abs_path*. If the *abs_path* is not present in the URL, it **must** be given as "/" when used as a Request-URI for a resource. If a proxy receives a host name, which is not a fully qualified domain name, it **may** add its domain to the host name it received. If a proxy receives a fully qualified domain name, the proxy **must not** change the host name” [W3C1].

2.3.3 HTTP Method

The set of common methods for HTTP/1.1 are defined as follows:

2.3.3.1 GET

The GET method is used to retrieve whatever information (in the form of an entity) is identified by the Request-URI. If the Request-URI refers to a data-producing process, it is the produced data that is returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.

The semantics of the GET method change to a "conditional GET" if the request message includes an *If-Modified-Since*, *If-Unmodified-Since*, *If-Match*, *If-None-Match* or *If-Range* header field. A conditional GET method requests that the entity be transferred only under the certain described circumstances identified in the

conditional header field(s). The main idea behind the conditional GET method is to reduce unnecessary network usage by allowing cached entities to be refreshed without requiring multiple requests or transferring data already held by the client [W3C1].

A GET method will change to a "partial GET" if the request message includes a *Range* header field. A partial GET requests that only part of the entity be transferred. The partial GET method is similar to the conditional GET method and is intended to reduce unnecessary network usage by allowing partially retrieved entities to be completed without transferring data already held by the client.

2.3.3.2 HEAD

The HEAD method is identical to the GET method except that the server **must not** return a message-body in the response. The meta-information contained in the HTTP response headers to a HEAD request **should** be identical to the information sent in a HTTP response to a GET request. This method can be used for obtaining meta-information about the entity without transferring the entity-body itself. This method is often used for testing hypertext links for validity, accessibility, and recent modification [W3C1].

The response to a HEAD request **may** be cacheable in the sense that the information contained in the response **may** be used to update a previously cached entity from that resource. If the new field values are different to the cached entity, then the cache **must** treat its present contents as stale.

2.3.3.3 POST

The POST method is used to request that the origin server accept the enclosed entity in a new request. POST is designed to allow a uniform method to cover the following functions:

- Annotation of existing resources

- Posting a message to a bulletin board, newsgroup, mailing list or similar group of articles
- Providing a block of data such as the result of submitting a form to a data-handling process
- Extending a database through an append operation

The actual function performed by the POST method is determined by the server and is usually dependent on the Request-URI. It is important to note that the action performed by the POST method may not necessarily result in a resource that can be identified by a URI [W3C2].

If a resource has been created on the origin server, the response **should** contain an entity that describes the status of the request and refers to the new resource and a Location header. Responses to this method are not cacheable, unless the response includes appropriate Cache-Control or Expires header fields. However, the response can be used to direct the client to a place where it can retrieve a cacheable resource.

2.3.3.4 PUT

The fundamental difference between the POST and PUT requests is reflected in the different meaning of the Request-URI. The URI in a POST request identifies the resource that will handle the enclosed entity. That resource might be a data-accepting process, a gateway to some other protocol or a separate entity that accepts annotations. In contrast, the URI in a PUT request identifies the entity enclosed with the request and not the resource. This means that the client knows what URI is intended and the server **must not** attempt to apply the request to some other resource.

HTTP/1.1 does not define how a PUT method affects the state of a server. Unless otherwise specified for a particular entity-header, the entity-headers in the PUT request should be applied to the resource created or modified by the PUT [W3C1].

2.3.3.5 Delete

The DELETE method requests that the origin server delete the resource identified by the Request-URI. This method may be overridden by human intervention (or other means) on the origin server. The client cannot be guaranteed that the operation has been carried out, even if the status code returned from the origin server indicates that the action has been completed successfully. However, the server should not indicate success unless, at the time the response is given, it intends to delete the resource or move it to an inaccessible location.

If the Delete request passes through a cache and the Request-URI identifies one or more currently cached entities, then those entries are treated as stale. Responses to the Delete request are not cacheable [W3C1].

2.3.4 HTTP Security

There are various security issues that surround HTTP/1.1. These include:

- Personal Information
  - Abuse of Server Log Information
  - Sensitive Information
- File and Path name attacks
- DNS (Domain Name Service) Spoofing
- Location Headers and Spoofing
- Proxies and Caching

2.4 Web Security

> *"Within the computer security community, 'Trust Management' has emerged as a new philosophy for protecting open, decentralised systems, in contrast to traditional tools for securing closed systems. Trust Management is an essential approach because the Web crosses may trust boundaries that old-school computer security cannot even begin to handle."* [KHA97]

Originally, the World Wide Web was developed as a publishing medium for public documents, so it provided few controls for restricting access to information [Corm97]. As the web became more popular, a larger number of documents and services were made available. These needed improved security facilities and a number of systems were proposed to satisfy the new requirements. In the following section, the security needs of users, publishers and authors on the web are set out, and two alternative solutions are examined.

2.4.1 Web Authentication

When a user uses the FTP or Telnet service, they are authenticated during the initial login process, commands get sent during the service and then the user logs out. Until the point where the user logs out, the initial authentication at the start of the service remains in effect for all operations performed. This is regarded as a *single session*. The HTTP protocol has no concept of a session. When a user makes a connection to a server, only a single request and response is sent and this is independent of any other connection between the same two parties. The HTTP protocol was designed this way to ensure that the server remains *Stateless*[1]. Therefore, the server will store no authentication information about any client and all client requests must be accompanied by the necessary authentication. Unfortunately, any system that reuses the same authentication information is vulnerable to replay attacks.

[1] The Server does not retain any information about connections established between it and clients that have sent requests

HTTP/1.0 provided only very basic authentication using only a static username and password. If the user requested a protected document, the server would send an "Unauthenticated" response. Upon receiving this response, the user's browser would prompt for an authorised username and password and resend these details in a new request. If the server accepts these credentials, the requested document will be returned. The browser can later use the same username and password, without consulting the user, in response to other "Unauthenticated" errors from the same server and realm. In HTTP/1.0, these credentials are encoded but not encrypted in the request so they can easily be detected by monitoring the network.

HTTP/1.1 made an improvement to this authentication weakness. It introduced a concept called *digest authentication*. The same exchange of requests are used as in basic authentication, but now the 'Unauthenticated' reply uses a value known as a *nonce* which acts as a challenge. Instead of the client reply with a username and password, the client calculates a message digest (using the MD5 algorithm) from the username, password and nonce and returns this with the username as authentication information [Corm97]. The server then repeats the MD5 calculation, using the user's correct password, and returns the document if the two digests match. For the server to be able to do this, it must store each user's password for calculating the MD5 digest. It is **very** important that these passwords are stored securely to prevent masquerading taking place.

The HTTP server is stateless so cannot "remember" the nonce value between each challenge and its response. The nonce must therefore be derived from some combination of information from the request packet along with values held centrally on the server [Corm97]. However, there is still the threat of attacks since the server does not ensure that nonces are unique to a single request. Choosing carefully the values of nonces can reduce this risk[2]. Only the server needs to know the value of the nonce and as a result, different servers can choose appropriate methods to the sensitivity of the information they hold.

[2] A good nonce calculation will usually include the URL of the document requested so that a successful replay attack can only retrieve a single document, rather than the whole realm as with basic authentication.

[Corm97] states "Digest Authentication does not provide a strong authentication mechanism. That is not its intent. It is intended solely to replace a much weaker and even more dangerous authentication mechanism: Basic Authentication. An important design constraint is that the new authentication scheme be free of patent and other export restrictions. Digest Authentication cannot meet most needs for secure HTTP transactions. For those needs SSL (Secure Socket Layer) or SHTTP (Secure HTTP) are more appropriate protocols."

2.4.2 Authentication Servers

Currently, most web servers perform their own authentication. However, a server could refer to another authentication server if it needs to check a client's credentials. Normally an authentication server is contacted **after** the initial exchange of username and password between the client and the specific web server. This means that the authentication server has no control of what value is used as the nonce. The authentication server may provide information (e.g. a Public Key) that will allow the web server to perform the authentication or it will perform the authentication itself. It is imperative that a secure connection is established between the authentication server and the web server to prevent any information from being stolen and later used under false pretences.

A problem with token-based systems on a network where the web server and authentication server both reside on the same network is the lack of recognisable sessions in the web protocol. The web server has no good way of knowing if sequences of connections arrive from the 'same' client so it may be necessary to authenticate each request individually. The result of such a scenario would mean a very slow system and would often be intolerable for most users. An alternative to this would be to allow web servers to cache this authentication information but it runs the risk of replay attacks. *Asymmetric Key* encryption or Public Key encryption is one way of securely caching a user's authentication information on a web server. The web server could be used to cache the user's public key so only the first request sent by the user in a session would incur a delay in contacting the authentication server.

2.4.3 Security requirements

User authentication and authorisation is of little value to a web server if the document is transmitted in clear text over the Internet. Encryption should be used to "prevent" information from being read. The user of a web service may also have their own security requirements. Examples of this might be the contents of a form containing personal and/or financial information.

There are documents that exist whose text is freely available to the public but which need to be certified as genuine e.g. price lists and journal articles. This authentication of authorship is different from authentication of the server where the document is held. It does not matter where the document is held but that it is genuine. Attaching a *digital signature* to the document usually does this. This is done by calculating a message digest value for the document (MD5, etc.) and then using the author's private key to encrypt this. This is then appended to the document before publication. In order to ensure that the text has not been altered, the reader can decrypt the signature using the author's public key and re-calculating the message digest. If the values match, the document is authentic.

2.4.4 SHTTP (Secure HTTP)

[Corm97] states that SHTTP provides a mechanism for browser and server to agree on their security requirements and adds information to the normal HTTP headers to allow signed and encrypted requests and responses to be sent and received. The basic mechanism is to take a normal HTTP request or response, encrypt and/or sign it as agreed, and then enclose it in an SHTTP request which carries only sufficient information to allow the authorised recipient to decrypt the contents.

2.4.5 SSL (Secure Socket Layer)

SSL is a protocol developed by Netscape™ and others and has been adopted by the IETF (Internet Engineering Task Force) under the name Transport Layer Security (TLS). This provides an encrypted TCP connection between a client and server.

Thus with strong encryption, transactions cannot be read from the network by any third party.

SSL is a general-purpose system so it cannot offer services that are tailored to the application that is using it [Corm97]. In order for SSL channels to pass through firewalls and other proxies, special arrangements must be set up between the client and the server. Unfortunately, these may allow unauthorised users to use the same route through the firewall since the channel is encrypted and therefore the firewall has no way of monitoring what is passing through it.

SSL does provide confidentiality and authentication of request and response messages [QUOTE]. It can be used to exchange certificates to authenticate the server and client machines[3]. No record is kept of each authentication so non-repudiation is not possible. A big problem with SSL is the low level of security available in the export version (due to US law export encryption restrictions). Another problem is the difficulty of interacting with application-specific intermediaries such as proxies and caches. Even though there is a lot of commercial support for SSL, it is important to note that it is not a complete solution to web security.

2.4.6 PGP (Pretty Good Privacy)

Authentication and encryption that is common on the Internet, as opposed to just the World Wide Web, uses a technique called *Pretty Good Privacy*. PGP is an international standard and is used for E-mail and FTP (File Transfer Protocol) traffic because of its full availability worldwide. PGP uses a random key to encrypt each transaction using the IDEA symmetric algorithm and encrypts the IDEA key using RSA asymmetric keys [Corm97]. This form of encryption is strong and difficult to break. Since it is the most widely available form of strong encryption, PGP is likely to be used for many Web Transaction Security proposals.

In order to use a PGP-based service, the user must know the public key of the service they want and the server must know the public key of the user. This is achieved quite easily since signed PGP keys are freely distributed and can therefore be copied.

[3] These assume the presence of third party Certificate Authorities

When users become registered, the service can obtain the public key of each user and the user in turn can obtain the public key of the service. Each user and server can then maintain a file of all the public keys that are required for secure communications. This reduces the need for authentication since the keys are stored at both ends of the communicating parties.

Using this method means that both user and server need to store files containing public keys. The Massachusetts Institute of Technology developed a way of overcoming this by designing a public key server that would store all public keys. This public key server provided interfaces that allow users and services to add or retrieve public keys. If a user wishes to use a service, they would first obtain the public key for that service from the public key server, check the signatures on the key to ensure that it is genuine, and add the key to its own file of public keys.

PGP keys are relatively small, typically a few hundred bytes, so the amount of disk storage required to store the different public keys is not normally a major concern.

2.5 The Apache™ web server security

In order to set up user authentication, there are two steps involved.

- Create a file containing the usernames and passwords
- Indicate to the server what resources are to be protected and which users are permitted (upon entering a valid password) to access them

2.5.1 Creating a User Database

In order to create a user database, a list of users and passwords needs to be created in a file. The file will contain a list of different usernames and their associated passwords. It is similar to the standard UNIX password file where each username and password is separated by a colon. All the passwords stored in the file are encrypted for obvious security reasons [ApacheSec].

2.5.2 The htpasswd program

The htpasswd[4] program is used to create a user file and to add or modify users. To create a new user file and add the username "*michael*" with the password "*nebula*" to the file `/usr/local/etc/httpd/users:`

```
htpasswd –c /usr/local/etc/httpd/users michael
```

The *–c* argument tells *htpasswd* to create a new user file. When this program is run, it will ask for a password to be entered for "*michael*". This will need to be entered a second time for confirmation. Other users can be added to the existing file in the same way without using the *–c* argument. The same command is also used to modify the password of an existing user. A typical users file might look something like the following

```
michael:FrtYZ5i9HJ7T

abby:vCX2L39QwCfby8x

katie:7Fyve4HsR1kBMt
```

[4] htpasswd is a C program that is supplied in the *support* directory of the Apache distribution.

2.5.3 Server configuration

In order for Apache to use the usernames and passwords in the file, a ***realm*** needs to be configured. A realm is a section of a web site containing web documents that is to be restricted to some or all of the users in the user file. Realms are usually set up on a per-directory basis, with a directory (and all its sub-directories) being protected [ApacheSec].

To configure and allow a directory to be restricted within a *.htaccess* file, the *access.conf* file must first allow user authentication to be set up in a *.htaccess* file. This is controlled by the **AuthConfig** override. "The *.htaccess* file should include `AllowOverride AuthConfig` to allow the authentication directives to be used in a *.htaccess* file" [ApacheSec].

To restrict a directory to any user listed in the users file just created, a *.htaccess* should contain

```
AuthName "authorised personnel"

AuthType Basic

AuthUserFile /usr/local/etc/httpd/users


require valid-user
```

- `AuthName` specifies the **realm** name. Once a valid username and password is entered, all other resources within the same realm name can be accessed with the same username and password.

- `AuthType` tells the server what prototype is to be used for authentication.
- `AuthUserFile` tells the server the location of the user file created by the htpasswd program.

The above directives together tell the server where to find the usernames and passwords and what authentication protocol to use. The server now knows that this resource is restricted to valid users.

The `require` directive is used to tell the server which usernames from the file are valid for particular access methods. The argument used with this directive, `valid-user`, tells the server that any username in the users file can be used [ApacheSec]. However, it can be configured to permit only certain users access, for example:

```
require user michael abby
```

If the `require` directive was used as above, it would only allow users *michael* and *abby* access to the resources contained in the directory (after entering a correct password). If user *katie* tried to access the directory, she would be denied even with the correct password. This use of the `require` directive is useful to restrict different directories in a server to different people in the same users file. As a result, if a user is permitted to access different directories, they only have to remember a single password[5].

[5] **Note**: if the **realm** name differs in the different areas, the user will have tore-enter their password

2.5.4 Using Groups

If a situation requires that only selected users from the users file are permitted to access certain directories, these users can be listed on the `require` line. However, this would mean building username information into the *.htaccess* file and this would be very cumbersome if there are a lot of users. The **Group** file is a way of solving this problem. The Group file operates similarly to standard UNIX groups i.e. any particular user can be a member of any number of groups. The `require` line can then be used to restrict users to one or more particular groups. For example, a group called **msc-class** could be created containing users who are allowed to access all internal web pages. To restrict access to just users in the msc-class group, the following would be used

```
require group msc-class
```

Apache allows multiple groups to be listed and `require user` can also be stated, in which case any user in any of the listed groups, or any user listed explicitly, can access the resource [ApacheSec]. For example

```
require group msc-class phd-students

require user dsg-director
```

would allow any user in group *msc-class* or group *phd-students*, or the user *dsg-director*, to access the resource after entering a valid password. A group file consists of lines giving a group name followed by a space-separated list of users in that group. An example might look something like the following

```
msc-class:michael joe mark ciaran

phd-students:ray mads
```

2.5.5 Limiting Methods

In the *.htaccess* file above, the `require` directory was not given inside a <*Limit*> section. Apache uses this to mean that the same directives apply to all request methods.

```
<Limit GET POST PUT>

require valid-user

</Limit>
```

If Apache was set up to limit just the POST method, the following would be declared in the *.htaccess* file

```
AuthName “restrict posting”

AuthType Basic

AuthUserFile /usr/local/etc/httpd/users


<Limit POST>

require group msc-class

</Limit>
```

Here, only members of group *msc-class* are allowed to POST. Other users (unauthenticated) can use other methods such as GET [ApacheSec].

2.6 XML

The eXtensible Markup Language (XML) was first proposed by the World Wide Web Consortium (W3C) as an alternative to HTML. Unlike HTML, XML is a meta-language, i.e. a language that allows one to create their own markup language for their own purpose [Architag98].

HTML is widely accepted as the means for describing information for transmission over the web. HTML uses tags to describe how information should appear and browsers interpret these tags and display the marked up information on a screen. These tags are primarily used as formatting tools. Although HTML is successful as an information-delivery language, it does lack extensibility [Architag98].

XML addresses many of HTML’s shortcomings. Unlike HTML where the formatting of a document depends on the tags it contains, in an XML document, the tag is

separate from the formatting. This means that in an XML document, the information is based on content and then the content markup is assigned a format. Individuals can therefore create their own tag-set that represents the information they want to exchange.

XML provides a set of rules that allow the definitions of individual tag-sets rather than abiding by the rules enforced by HTML. The syntax used for defining XML is very similar to HTML except for three main differences [Architag98]:

- All open tags **must** have a corresponding close tag
- All attribute values **must** be in quotes
- Empty tags (such as those used for images in HTML) must not have a close tag. The start tag has a back slash in the close angle bracket - <image src="image.gif"/>.

The eXtensible Style Language (XSL) is used as the means for displaying an XML document. It has a similar effect as Cascading Style Sheets (CSS) without any effects of a proprietary style language. XSL separates the formatting from the content of XML.

Figure 2.4 shows an example of the structure of an XML document.

```
<?xml version=”1.0” standalone=”yes”?>
<conversation>
<greeting>Hello, world!</greeting>
     <response>We are not alone!</response>
</conversation>
```

Figure 2.4 XML document

2.7 Servlets

Servlets are modules that run inside request/response-oriented services and extend them in some manner. An example would be when a HTTP service that responds to its clients by delivering the HTML files that it requests.

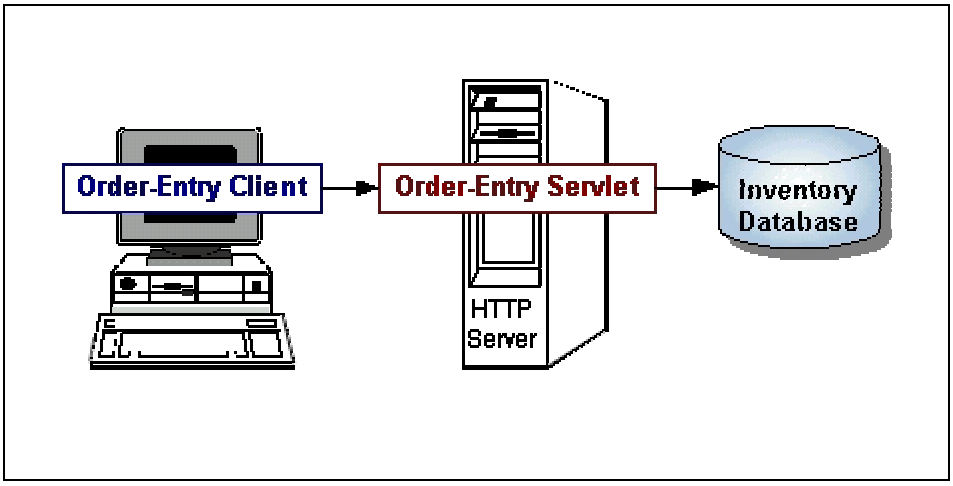

Figure 2.5 Example Servlet configuration

A Servlet can extend the capabilities of the HTTP service, for example, by taking the data entered by a client in a HTML-entry form and applying the appropriate logic used to update a database.

Servlets are to servers what applets are to browsers. Unlike applets, however, Servlets have no graphical user interface. Servlets can be embedded in many different servers because the Servlet API (Application Programming Interface), which is used to write Servlets, assumes nothing about the server's environment or protocol [Bloch99]. Servlets have become most widely used within HTTP servers and most current web servers now support the Servlet API. As a result, a Servlet can call on other Servlets and services to satisfy a request, if appropriate.

2.7.1 Servlets v CGI scripts

Servlets are a common server-side alternative to using CGI scripts. They provide a method of generating dynamic documents that are relatively easy to develop and are fast to run as compared to CGI scripts [Bloch99]. With traditional CGI, a new process is started for each HTTP request. If the CGI program does a relatively fast operation, the overhead of starting the process can dominate the execution time. However with Servlets, the Java Virtual Machine remains up, and each request is handled by a lightweight Java thread, not a heavyweight operating system process. Similarly, in traditional CGI, if there are *N* simultaneous request to the same CGI program, then the code for the CGI program is loaded into memory *N* times. On the other hand, with Servlets, there are *N* threads but only a single copy of the Servlet class. Servlets also have more alternatives than do regular CGI programs for optimisations such as caching previous computations and keeping database connections open [Hall].

Servlets are a way of doing server-side development using platform-specific APIs. These APIs are part of the Java™ Servlet API class from Sun Microsystems™.

Servlets could therefore be used to handle many different types of HTTP client requests. These would include data posted from HTML forms typically used on on-line shopping forms or banking systems.

2.7.2 Uses of Servlets

There are many applications where Servlets are used. These applications might include

- Processing data POSTed from a HTML form. Servlets are very good when used as part of order-processing systems, on-line payment systems, etc. where sensitive data is sent from the client to the server and needs to be processed.
- Interaction between users. Servlets can be used to handle concurrent requests and can support systems that provide on-line conferencing.
- Forwarding requests. Servlets can be used to forward client requests to other Servlets and/or services. An advantage of allowing this might be for load balancing. Another reason might be when a single service has been partitioned over a number of servers.
- Communities of active agents. A Servlet could be used to define active agents that could be used to share tasks amongst each other. Agents would be a Servlet themselves and they could pass data to each other if needed.

2.7.3 Servlet Architecture

All Servlets implement the JSDK (Java Servlet Development Kit) *Servlet* interface or extend the *HttpServlet* class [Servlet00]. The Servlet interface provides APIs to methods that manage the Servlet and its communications with clients. When a client sends a request to the server, the Servlet that deals with the request accepts two objects. These are:

- *ServletRequest*

  This class encapsulates the communication from the client to the server.

- *ServletResponse*

  This class encapsulates the communication from the Servlet back to the client.

The *ServletRequest* interface allows Servlets to access a lot of information. This would include the names of any parameters passed from the client request, the protocol being used by the client, and the names of the remote host that made the request and the server that received it. If clients use application protocols such as HTTP POSTs and HTTP PUTs, the interface provides an input stream called *ServletInputStream* through which the Servlet can get the client data.

The *ServletResponse* interface provides APIs for methods that allow Servlets to respond to clients. The interface allows Servlets to set the content length and MIME type of the response, provides an output stream called *ServletOutputStream*, and a writer through which the Servlet can send the response data [Servlet00].

Both of these interfaces constitute a basic Servlet. There are many other classes and interfaces that provide extra functionality for Servlets. **Figure 2.6** shows an example of a simple servlet [Bloch99].

```
public class SimpleServlet extends HttpServlet

{
/**
   Handle the HTTP GET method by building a simple web
   page.
*/

public void doGet (HttpServletRequest request,
                   HttpServletResponse response)
        throws ServletException, IOException
{
        PrintWriter out;
        String title = "Simple Servlet Output";

// set content type and other response header fields first
        response.setContentType("text/html");

// then write the data of the response
        out = response.getWriter();

        out.println("<HTML><HEAD><TITLE>");
        out.println(title);
        out.println("</TITLE></HEAD><BODY>");
        out.println("<H1>" + title + "</H1>");
        out.println("<P>This is output from
                    SimpleServlet.");
        out.println("</BODY></HTML>");
        out.close();

        }
}
```

Figure 2.6 Example Servlet

2.7.4 Servlet Lifecycle

Servlets are loaded and then run in a service that accept requests from clients and return responses. When a Servlet is loaded in a service, the Servlet's ***init*** method is run. The service always calls the Servlet's *init* method when the Servlet is loaded and

it will not call it again unless the Servlet is reloaded[6]. The *init* method is always called before any client requests are handled (i.e. before the *service* method is called) or the Servlet is destroyed.

Upon initialisation of the Servlet, all client requests can now be dealt with. The Servlet's ***service*** method is used to deal with these requests. When a client request is sent to the servlet, the servlet forks a separate servlet-thread to allow the request call run its own *service* method. This means Servlets can run multiple *service* methods at any one time. Therefore appropriate precautions must be taken to ensure that the *service* methods run in a thread-safe manner [Servlet00]. For example, if a `service` method updates a field in the servlet object, that access should be synchronized. If for some reason, a service should not run multiple `service` methods concurrently, the servlet should implement the `SingleThreadModel` interface. This interface guarantees that no two threads will execute the Servlet's `service` methods concurrently [Servlet00].

Servlets run until they are removed from the service, for example, at the request of a system administrator. When a service removes a servlet, it runs the Servlet's *destroy* method. This method is run once. The service will not run it again until after it reloads and reinitialises the servlet. When the `destroy` method runs, however, other threads might be running service requests. If, in cleaning up, it is necessary to access shared resources (such as network connections to be closed), that access should be synchronized [Servlet00].

2.7.5 Servlet development

Servlets implement the *javax.servlet.Servlet* interface. While developers can develop Servlets by implementing this interface directly, it is not necessary. Since most Servlets extend web servers that use the HTTP protocol to interact with clients, the most common way to develop Servlets is by specialising the *javax.servlet.http.HttpServlet* class [Servlet00].

[6] The service cannot reload a servlet until it has removed that same servlet by calling the *destroy* method.

"The *HttpServlet* class implements the Servlet interface by extending the GenericServlet base class, and provides a framework for handling the HTTP protocol. Its *service* method supports standard HTTP/1.1 requests by dispatching each request to a method designed to handle it" [Servlet00].

By default, Servlets written by specialising the *HttpServlet* class can have multiple threads concurrently running its *service* method. If there was some reason that only a single thread was allowed to run a *service* method, then in addition to extending the *HttpServlet* class, the servlet must also implement the *SingleThread* interface. **Figure 2.7** shows how this is done [Servlet00].

```
public class SurveyServlet extends HttpServlet
                          implements SingleThreadModel

{

/* typical servlet code, with no threading concerns
 * in the service method.  No extra code for the
 * SingleThreadModel interface.
*/

}
```

Figure 2.7 Single Threaded Servlet

2.7.5.1 Client Interaction

Clients that interact with Servlets and extend the *HttpServlet* class must include one or more of the following methods

- doGet for handling GET, conditional GET and HEAD requests
- doPost for handling POST requests

- doPut for handling PUT requests
- doDelete for handling DELETE requests

2.8 Open Web Architecture

An Open Web Architecture is an architecture that is designed and implemented to support modification, extension and reconfiguration [Kin98]. The reconfiguration typically would occur either at install-time, use-time or run-time. An Open Web Architecture allows the modification of either documents on the web, the web-software itself or possibly both. *Wiki* is one such architecture where users are permitted to manipulate documents without many restrictions using a standard web browser [Jiki.org].

2.8.1 Jiki

Jiki is an open web architecture that allows the editing of web pages in a free and non-restrictive manner. It is a distributed, component-based (Open Source), Wiki-like server designed and built by members of the Distributed Coalition [Distrib.org]. **Figure 2.8** shows a very high level architecture of Jiki.

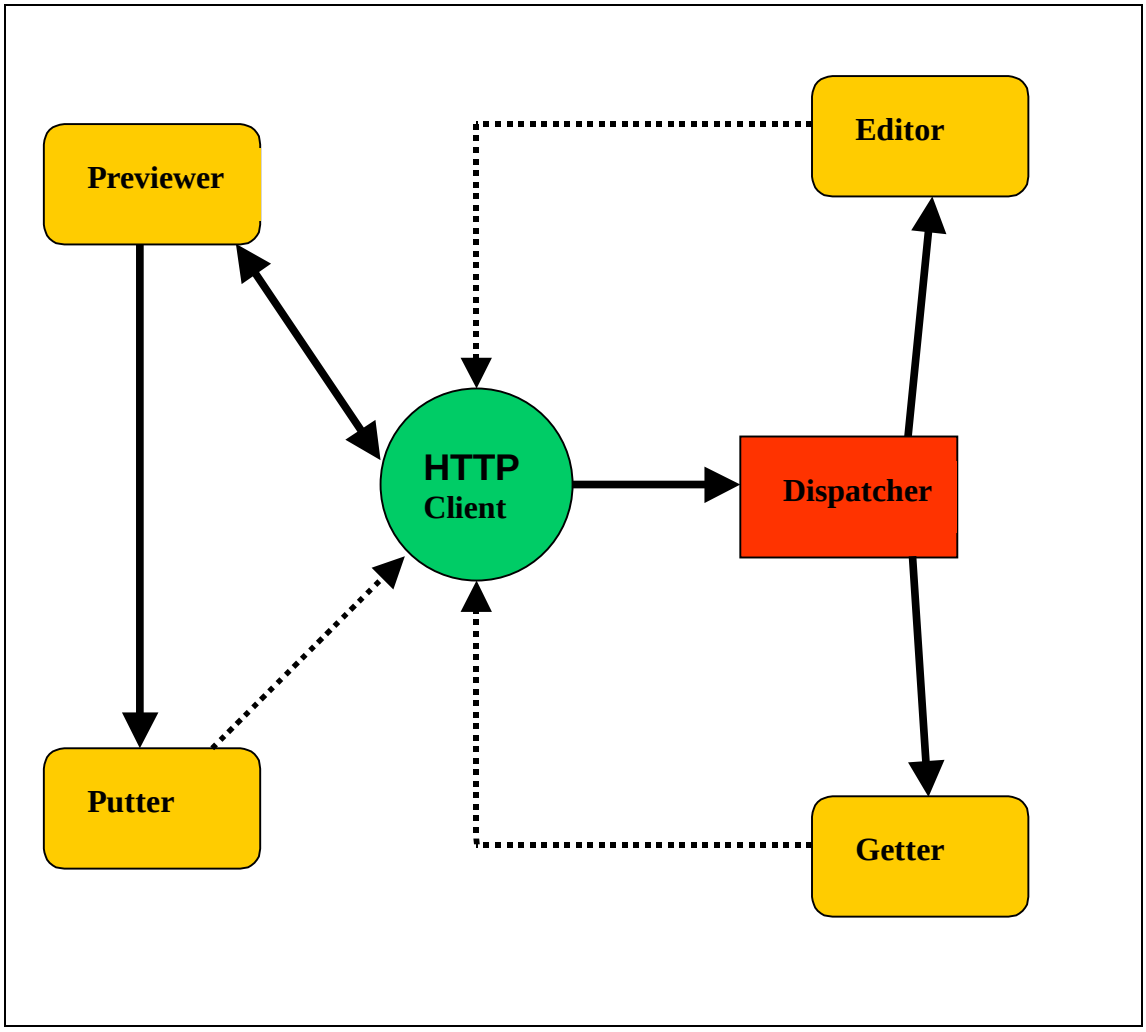

Figure 2.8 Jiki High-Level Architecture

The Jiki architecture is composed of several types of components. Each component is a Java™ Servlet that interacts with one another. **Figure 2.8** shows the main components used in Jiki. These are

- Dispatcher
- Getter
- Editor
- Previewer

- Putter

All of these interact with one another by either using HTTP to communicate (currently just GET and POST) or use local method calls [Jiki.org]. Each component is resolved using a *properties* file. This file maps a Servlet class that implements a specific component to a Servlet name. There are no limits to the number of names that can be mapped to Servlet components. As long as the Servlet name associated with a component is listed in the *properties* file, the Jiki server will be able to find the class implementing the Servlet component. Therefore, when a client sends a request to the Jiki server, Jiki will look at the name of the Servlet in the URL and find this name in the *properties* file. When it finds the Servlet name, it will dynamically load the class associated with this Servlet name.

The designers and developers of Jiki purposely designed the Jiki architecture to be generic and extensible. This promoted people to download and modify the architecture to suit their needs and not have to work with a rigid, non-modifiable web-server.

Jiki has several advantages and disadvantages over other commonly used web-servers. These include:

Advantages

1. Jiki is written in pure Java.

2. The Jiki architecture is very generic and extensible.

3. Jiki is free to download, including all source code and documentation.

Disadvantages

1. There is **no** security built into the Jiki architecture. This means any client can download and edit any page without restriction.

2. Jiki uses flat files for storing web pages.

3. It is slower than most other web-servers due to the flat-file storage problem and all the inter-component communication.

4. Jiki does not have as many functions like most other web-servers (e.g. search functionality, database backend, etc..).

2.8.2 Jiki Page Format

The content of every Jiki page is in `plain text`. When a client either edits the content of an existing page or creates a new page, they do **not** have to use HTML tags to format their text. No HTML knowledge is therefore required.

Instead, the designers decided to design their own rules on formatting text in Jiki pages. This is what they termed "*The Jiki Data Format*" [Jiki.org]. In the original Jiki architecture, the designers placed a Help page (using this data format) that explains how to use different symbols to format the text. This page is called *TextFormattingRules*. Since the page is one of the help pages in the original Jiki architecture, it will not require any read authorisation after the security framework has been integrated. It will therefore be available to **all** clients to read. Examples of some of the formatting rules are as follows [Jiki.org]:

- **Paragraphs**
    - Don't Indent paragraphs
    - Words wrap and fill as needed
    - Use blank lines as separators
    - Four or more minus signs make a horizontal rule
- **Lists**

    - tab-* for first level
    - tab-tab-* for second level, etc.
    - Use * for bullet lists, 1. for numbered lists (mix at will)
    - Any digit or string of digits works fine for numbered lists.
    - tab-Term:-tab Definition for definition lists
    - One line for each item
    - Other leading white space signals preformatted text
- **Emphasis**
    - Use doubled single-quotes ('') for emphasis
    - Use tripled single-quotes (''') for strong emphasis
    - At most one per line.
    - Don't cross line boundaries
- **References**
    - Local references are indicated by [words inside of square brackets].
    - Remote references are indicated by enclosing the name of the link and its URL, separated by a vertical bar ('|'), inside of square brackets (e.g. go to the [W3C|http://www.w3.org/] for information on HTTP 1.1.
    - Or precede URLs with "http:", "ftp:" "mailto:",etc. to create links automatically as in: http://c2.com/

This data format was a design decision taken and implemented into Jiki by the original authors of the Jiki Architecture [Jiki.org]. The new security infrastructure

being integrated into the architecture did not change or alter this in any way. Therefore, the design decision made to allow the creation of new Jiki pages in the security framework uses the same technique as that used to edit a page in the original architecture. **Figure 2.9** shows an example of a Jiki page whose text contains some of these symbols for formatting purposes.

```
'''Welcome to the Jiki Web!'''

Jiki is cool. Jiki is fun. Jiki is Jiki. Get Jiki with
it.

Test foobar. And test again

The Jiki server started as a quick-hack
[WikiWikiWeb|http://c2.com/cgi/wiki?WikiWikiWeb] server
written in Java. From spec to implementation took 5 1/2
hours in Vancouver, BC at
[OOPSLA'98|http://www.acm.org/sigplan/oopsla/oopsla98/].

See us [HardAtWorkOnJiki]. Now it is a full-blow
distributed component-based server will all sorts of cool
functionality being added everyday.



It is the first product of the
[DistributedCoalition|http://www.distributedcoalition.org
/].

Bookmark [jiki.recent changes] to keep abreast of Jiki
developments. Please add to [jiki.suggestions] if you
have a suggestion for Jiki. Check/add to the [jiki.bugs]
page if you find or fix a bug.

Jiki was written by several [people.authors], many of who
are Canadian, eh?

Please read the [help.welcome visitors] page to know
where to start. Also, learn how to use [help.good style].
```

Figure 2.9 Jiki Page data format

2.9 Summary

This chapter described how aspects of the current web architecture operate and the newly developed Jiki (Open Web) architecture. It gave a discussion about Web security and its implications, and the technologies that will be used in adding authenticity and security to the Jiki architecture. Before looking at how an open web architecture is designed, aspects of the existing architecture needed to be researched and understood. This included the HTTP/1.1 application protocol and some of the methods that it supported (GET, POST, etc.).

Hypermedia is a new area of research and is becoming a technology that is being adopted by many web developers. As a result, hypermedia and how it is engineered was researched and the main points covering this have been discussed.

One of the main disadvantages with the Jiki web architecture is that it lacks support for authentication and security. To this regard, existing web security was looked at and how it is implemented over the present web architecture. There are several ways how this is done and some of these techniques have been discussed. These included Secure HTTP (SHTTP), Secure Socket Layer (SSL) and Pretty Good Privacy (PGP).

Jiki is essentially a web server so most of the development work for the dissertation will be server-side. There are several technologies currently available for server-side development and some of these were researched. XML and its implications were examined and also Java™ Servlets and their strengths and weaknesses.

3. Jiki Security Design

3.1 Introduction

[ApacheSec] states that there are two ways of restricting access to web documents: either by the hostname of the browser being used, or by asking for a username and password. The former can be used, for example, to restrict documents to use within a company. "However, if the people who are allowed to access the documents are widely dispersed, or the server administrator needs to be able to control access on an individual basis, it is possible to require a username and password before being allowed access to a document". This is called **user authentication**.

This chapter will discuss the design of integrating security features into the Jiki web server architecture. The chapter will then discuss the practicality of employing such security techniques into Jiki and appropriate ways of doing so.

3.2 Jiki security integration

In chapter 2, it was explained that one of the main problems with the Jiki web server architecture was its lack of security and authentication. The existing design permits any user to send a HTTP request to the server requesting any web page without any form of identification (if necessary). The user does not require any authorisation since the Jiki server is not concerned who they are. This means users are permitted to edit any web page they have requested with **no** restrictions. Any changes subsequently made to a web page are then stored at the server. Before a design can be made to resolve this problem and add security to the Jiki architecture, several scenarios must first be identified.

3.2.1 Security scenarios

The following are the possible scenarios that the Jiki web-server must be capable of handling with appropriate security mechanisms:

1. **Single author**

This scenario is where only a single person is the author of the file. The Jiki server must provide appropriate security to allow ***only*** the author of the file permission to edit it.

- *Advantages*

  Only the author has the rights to edit the web page so any unauthorised people who try to edit the page will not be able to do so.

- *Threats*

  There are no significant threats associated with this scenario. The only real threat would be if a user guessed the correct password that would allow them to edit the page.

2. **Collaborative authoring**

This scenario is where more than one person co-authored the file. Appropriate security measures must be provided by the Jiki server to allow ***only*** the co-authors of the file permission to edit it.

- *Advantages*

  Changes can be made to the web page only by those who have the rights to do so. An example might be the use of an information base where only those who have the rights to edit the web page can add any extra information to it.

- *Threats*

As a result of more than one person knowing the password to allow them edit the web page, there is a higher chance of this password being stolen. This can occur either by hackers who use network sniffers to steal passwords, or by simply telling someone who then uses the password to edit the web page maliciously.

3. Communities

Communities are a form of collaborative authoring whereby different communities have different rights/privileges to author and/or edit files stored on the Jiki server. The server must be able to distinguish what community a user belongs to. Depending on this, the server will thus know what privileges the user has been granted and will act accordingly whether the user can author and/or edit files.

- *Advantages*

  The use of communities means that users can belong to more than one community. Each user that does belong to more than one community may not necessarily have the same rights and permissions to author/edit files. They may have full privileges to edit files in one particular community but may only have read-only rights in another.

- *Threats*

  If users are part of more than one community, it might be easy for them to either steal passwords from colleagues or try to gain rights in communities by using a 'backdoor'. Since they may have full privileges in one community, it is important that the security in the server is robust enough to detect and prevent this from occurring.

4. No restrictions

There may be situations where specific web pages will have no security access restrictions. After the author has created the page, there are no restrictions on people

who may download it. Similarly, each person who downloads the page, also has full rights to edit the page in whatever fashion they desire. This includes not only editing the text, but also hypermedia such as pictures, audio and video can be added to the page. HTML links to other web sites may also be placed on the page. When the changes have been saved, the page will be available for any other people who make a request for it from the Jiki web-server.

- *Advantages*

  Having no restrictions means that everyone has full rights to edit a web page. This would be useful if the author(s) of the page wanted others to add extra knowledge to a knowledge base for example. It is a very easy way to gather information and does not require any HTML knowledge by users who edit the page.

- *Threats*

  Having no restrictions means that all the web pages are prone to attack from anyone. Any page may be requested from the server and edited freely. Users could then introduce any kind of material and the chances of introducing some kind of virus are big.

3.2.2 Security Policy Manager

Security policies are a way of implementing some or all of the scenarios described in **Section 3.3.1**. Such security policies can be designed to restrict access to Jiki web pages only to authorised users. The author of a new Jiki page would select these users at the time they create the page. A *security policy manager* would then be used to manage all the security policies applicable to each Jiki web page from then on.

Figure 3.1 shows a high level view of how the security policy manager would manage the security policies for each Jiki web page.

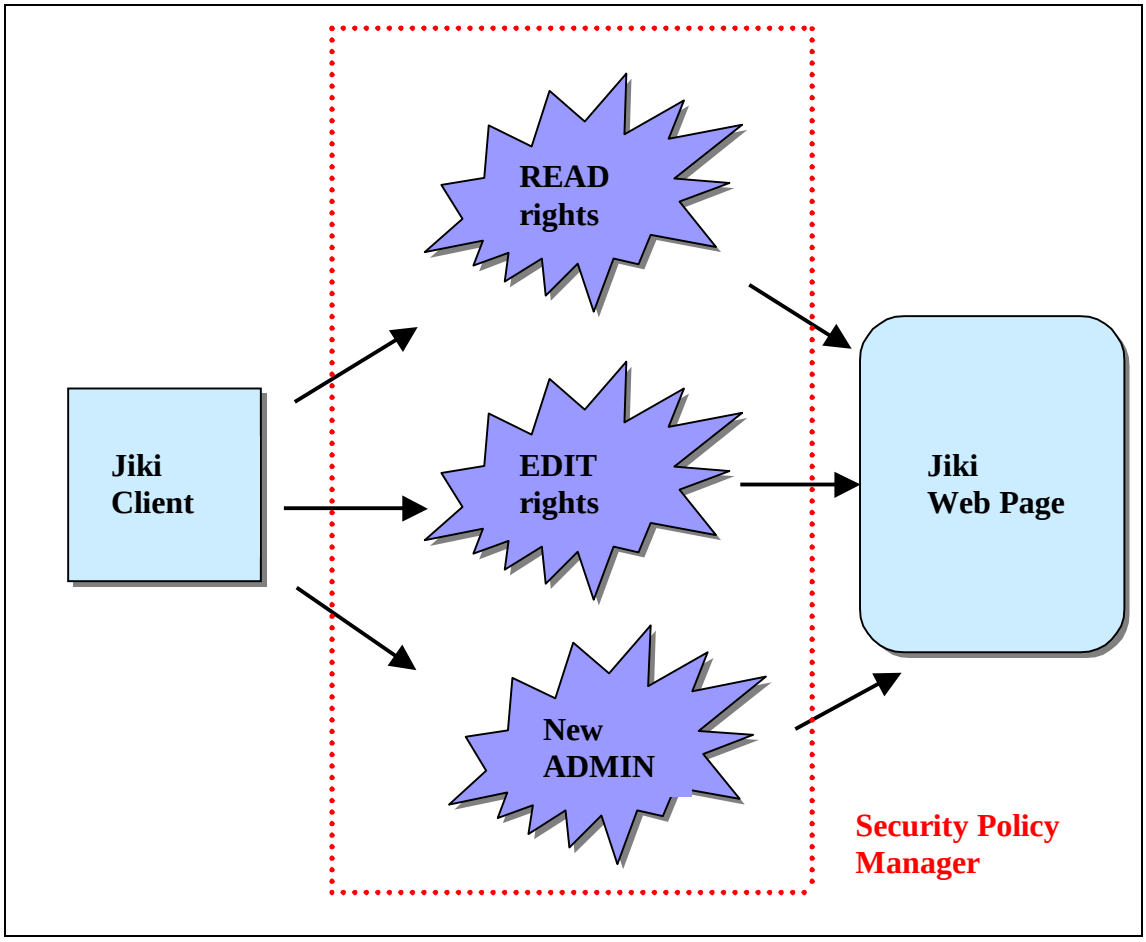

Figure 3.1 High-level view of a Jiki page Security Policy Manager

When the client creates a new Jiki Page, it can set whatever security policies deemed necessary at that time. The *security policy manager* would then enforce these policies for the lifetime of that page. Since the *security policy manager* is vital in the security of each Jiki page, it is important that access to it is restricted to only one specified client. This client will initially be the author of the page. Only with an authorised client name and password, will access be granted to the internal settings of the *security policy manager*. In order to keep resilience in the security features of the Jiki Architecture, authorised access to the *security policy manager* can be assigned to any client, other than the author[7]. The **administrator** of a Jiki page is the only person who has authorised access to the *security policy manager* for that page and can edit the security settings.

[7] By default, the author of a Jiki page has initial authorised access to the *security policy manager*.

The *security policy manager* allows three types of security settings to be set for every Jiki page. These are:

- **READ** – access clients

  These are clients who have authorised read – access rights to the Jiki page. Clients who do not have read – access rights are unable to read the page. Clients that do have read – access rights do not necessarily have edit – access rights too.

- **EDIT** – access clients

  These are clients who have edit – access rights to the Jiki page. Only clients who have been granted this right are able to edit the Jiki page. By default, all clients with edit – access rights to a page, also have read – access rights.

- **New Administrator** for the page

  The author of a Jiki page is by default the administrator of the page. The author is the only client who has authorised access to the security policy manager. If the author wishes to cease being the administrator for a Jiki page, they can select another client and authorise them as the new administrator. At that point, the author will cease being the administrator for the page and will no longer be able to access the security settings in the security policy manager.

The mechanisms above were deemed to be the most appropriate method of implementing the security settings for each Jiki page. Designing a security policy manager to meet these three settings for every Jiki page means that all the security scenarios as discussed in **Section 3.2.1** can now be set.

3.2.3 Establishing a HTTP session

Whenever a client is authenticated using their name and password, the Jiki server should have some mechanism of remembering these details. It is human nature that people who have to re-enter the same information over and over will get frustrated and not like using the system. For this reason, it is important that the design of a security framework within Jiki must include a mechanism enabling the Jiki server to 'remember' the client's identity and password. Since every client interacts with the server through a web-browser and communicates using HTTP, there are two possible ways for the server to retain the client's details. These are:

- Cookies

  "Cookies are a way for a server (or a Servlet, as part of a server) to send some information to a client to store, and for the server to later retrieve its data from that client. Servlets send cookies to clients by adding fields to HTTP response headers. Clients automatically return cookies by adding fields to HTTP request headers" [Sun00].

- HTTP session

  "Session tracking is a mechanism that Servlets use to maintain state about a series of requests from the same user (that is, requests originating from the same browser) across some period of time. Sessions are shared among the Servlets accessed by a client. This is convenient for applications made up of multiple Servlets. For example, on-line book stores uses session tracking to keep track of the books being ordered by a user. All the Servlets in the example have access to the user's session" [Sun00].

Both methods above have their advantages and disadvantages. However, the use of a HTTP session does have an advantage over the use of cookies in this case. There are two main reasons for this and these are as follows:

1. Whenever a client sends an initial HTTP request to the Jiki server and is authenticated, this will be done using a web-browser. **Only** while this web-browser remains open, will the HTTP session remain alive and will retain all the client's authenticated details. As soon as the web-browser is closed, the HTTP session is closed and the threat of another using trying to pretend to be the client is no longer a worry.

2. If a cookie was used, there is always the security threat of '*Masquerading*'. This is a situation where a malicious user might masquerade or pretend to be an authentic client and use their computer where the cookie resides, to connect to the Jiki server. This is a serious security breach that is difficult to resolve since the server has no way of knowing if the client, is in fact, the authentic person.

3.2.4 Encryption (MD5 Message – Digest Algorithm)

Each client that sends HTTP requests to the Jiki server must first be authenticated. Due to the "untrustworthiness" nature of Internet users today, there are many ways of authenticating users before they can connect to a server and access data being stored there. Encryption is a common mechanism used for authentication. There are several well-known encryption algorithms being used today, some of which include:

- DES (Data Encryption Standard) family algorithms
- RSA family algorithms
- Message – Digest encryption algorithms

All of these algorithms have their advantages and disadvantages and each one is most applicable for use in different types of situations. The main criteria that was wanted for the encryption of data stored on the Jiki server was that the encryption algorithm

had to be fast, robust and most obviously, secure. One algorithm fitted all of these and is readily available. This was the **MD5 Message – Digest** encryption algorithm.

[Rivest92] states “the MD5 algorithm takes as input a message of arbitrary length and produces as output a 128-bit "fingerprint" or "message digest" of the input. It is conjectured that it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given pre-specified target message digest. The MD5 algorithm is intended for digital signature applications, where a large file must be "compressed" in a secure manner before being encrypted with a private (secret) key under a public-key cryptosystem such as RSA. The MD5 algorithm is designed to be quite fast on 32-bit machines. In addition, the MD5 algorithm does not require any large substitution tables; the algorithm can be coded quite compactly”.

[Rivest92] also states “the MD5 algorithm is an extension of the MD4 message-digest algorithm. MD5 is slightly slower than MD4, but is more "conservative" in design. MD5 was designed because it was felt that MD4 was perhaps being adopted for use more quickly than justified by the existing critical review; because MD4 was designed to be exceptionally fast, it is "at the edge" in terms of risking successful cryptanalytic attack. MD5 backs off a bit, giving up a little in speed for a much greater likelihood of ultimate security. It incorporates some suggestions made by various reviewers, and contains additional optimisations. The MD5 algorithm is being placed in the public domain for review and possible adoption as a standard“.

The MD5 algorithm was seen to be ideally suited for the encryption of client passwords so that they could be stored “securely” at the server. Since the MD5 algorithm is a one-way hashing algorithm, any attempt to decrypt an encrypted password would be **extremely** difficult. For this reason, a design decision was taken to store each client name and encrypted password in a file at the server. Even if there was a breach in security and this file was copied or stolen, all the passwords contained in the file are encrypted. The file therefore would be of little use to the thief, other than the names of all the registered clients.

The first HTTP request a client sends to the server requesting data that requires authentication, the client would need to enter their name and password. Due to the one-way hashing function of the MD5 algorithm, the server is **unable** to decrypt the client's password that it stores on file to compare it with the password entered. Instead, the opposite occurs. The server uses the algorithm to encrypt the password entered and compares it to the encrypted password for the client stored on file. If the two encrypted password match, then the client is authenticated.

3.3 UML design

Figure 3.2 and **Figure 3.3** show a Use-case and Sequence diagram respectively of the design of the new security framework.

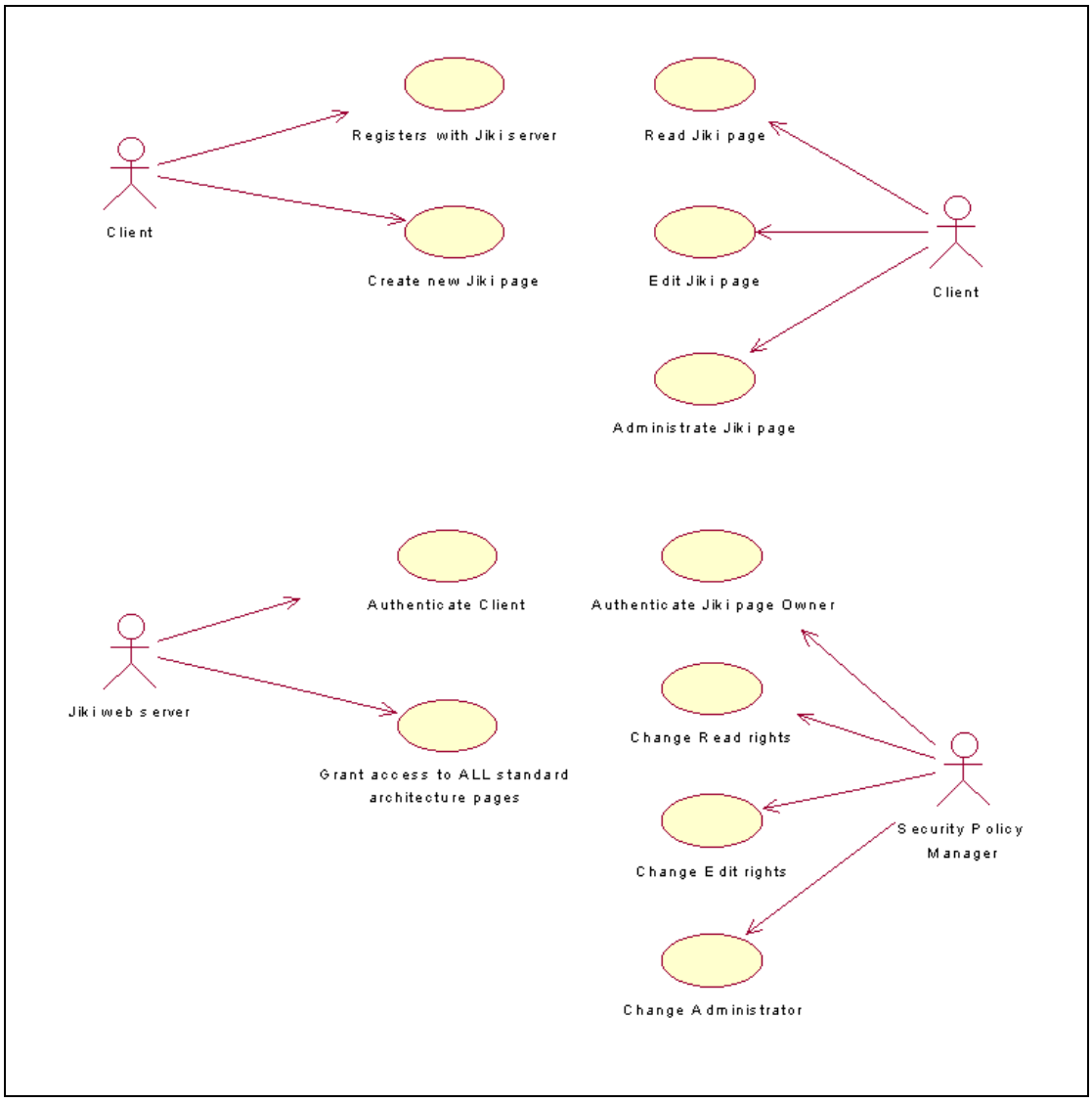

Figure 3.2 Jiki Use-Case diagram

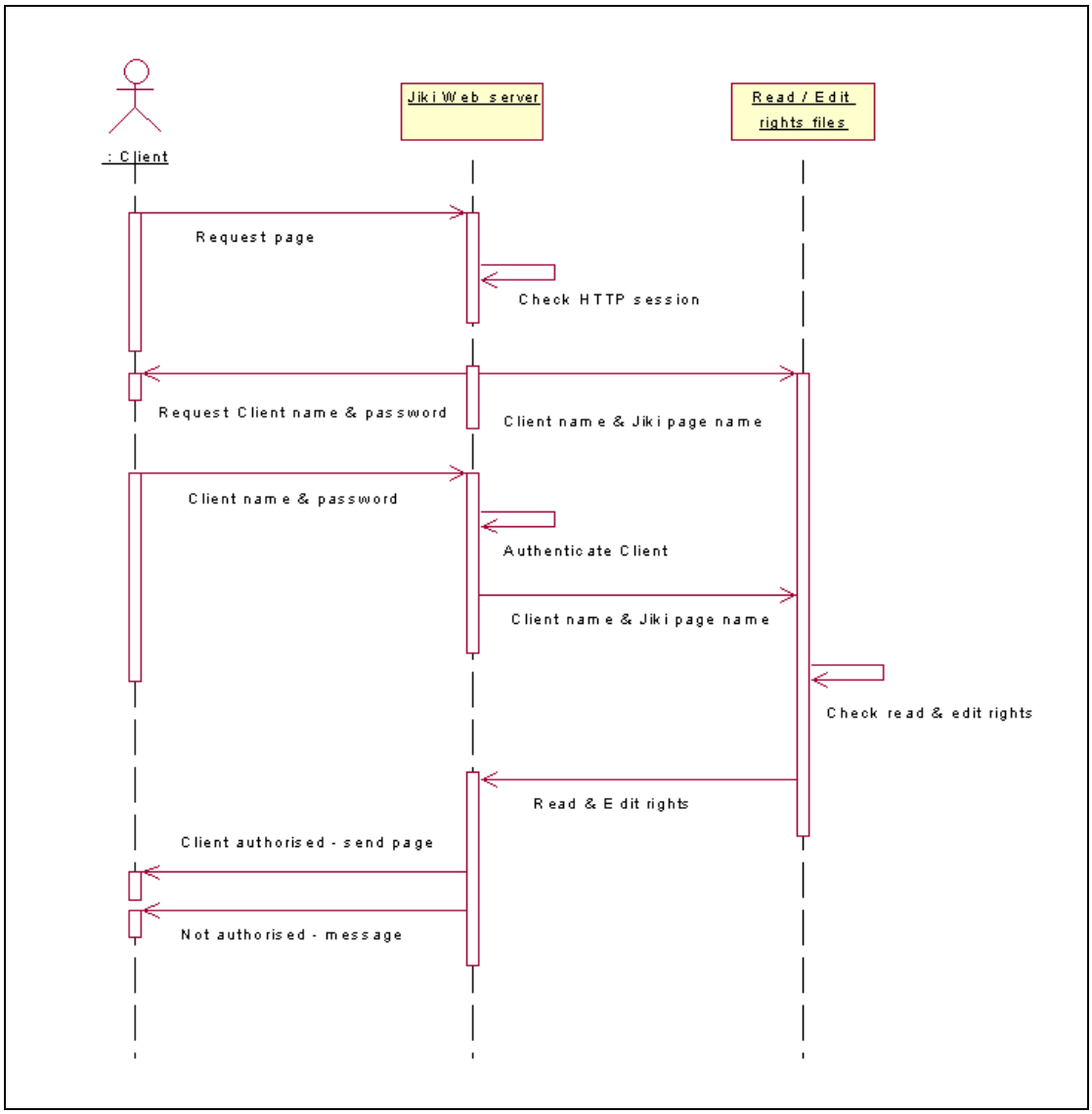

Figure 3.3 Sequence Diagram to Read / Edit a Jiki page

3.4 High Level Architecture

The main requirement of this dissertation was to improve an open web architecture (Jiki) by adding a security infrastructure into its overall framework. One of the principle key-points decided to be adhered to during the design of this infrastructure was that the design should keep any substantial changes to the existing architecture to a minimum. Instead, the new security infrastructure should be integrated into the existing architecture with a minimum amount of change to *areas not requiring* these security features. **Figure 3.4** shows the existing Jiki High Level architecture with a security policy manager.

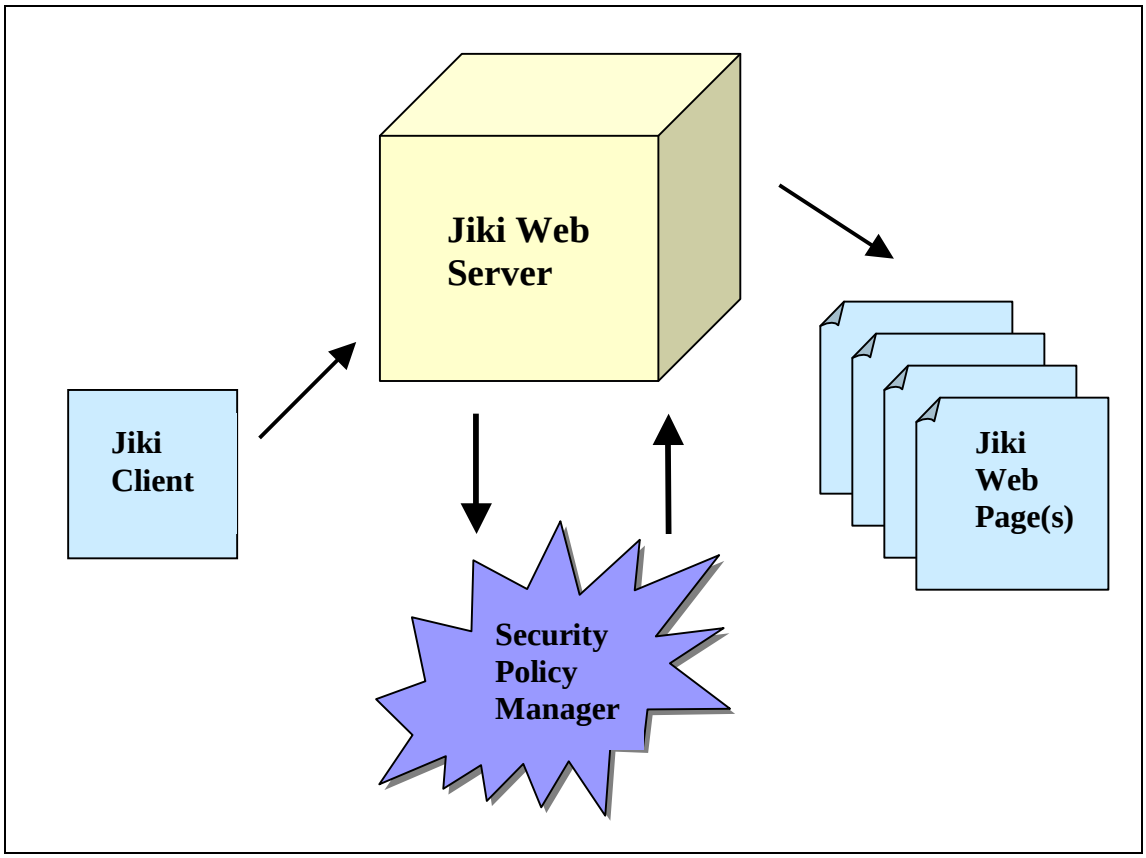

Figure 3.4 Jiki High-level Architecture

3.4.1 GUI

The client will communicate with the Jiki server through a number of different GUIs. Since Jiki is a web-server, both it and the client will communicate using HTTP. Therefore, the use of a web-browser is used as the front-end interface to the client. Some of the GUIs that will be used will include:

- A new client registering with Jiki for the first time
- A client creating a new Jiki page
- A client editing the text of a Jiki page
- All authentication procedures
- An administrator of a Jiki page making changes to the security policy manager for that page

3.4.2 Jiki Security Infrastructure

The security infrastructure needs to be designed in a way that it can integrate into the Jiki architecture without changes having to be made to areas where security is not required. Keeping the look and feel of the existing architecture after the security infrastructure has been integrated was one of the main design objectives.

Security within Jiki centres on whether a client has the authorisation to read and/or edit a Jiki web page. However, there are certain pages built into the existing architecture that are displayed for various reasons that should not require such authorisation. Some of these include error messages, acknowledgements, queries, etc.. These are core pages within Jiki that any client can read without any authentication. Only pages created by different clients will involve the new security elements.

The functions that the new secure architecture will provide include:

- All clients must **register** with the Jiki server.
- New clients can create an unlimited number of Jiki web pages.
- **All** new Jiki web pages will involve selecting registered clients who have Read-access rights and Edit-access rights.
- Authentication of **every** client.
    - Reading a page.
    - Editing a page.
    - Access to the Security Policy Manager.
- A Security Policy Manager
    - Edit clients with read and/or edit rights for any Jiki web page.
    - Change the administrator of a page.

The front-end to the new security features being incorporated into Jiki will be primarily responsible for reading the client name and password in various areas, parsing all input variables and passing these to the Jiki server using HTTP requests. The front-end will also display all relevant HTTP responses received from the server. The back-end will store all the Jiki web pages in one repository, authenticate clients, maintain files of clients with read-rights and edit-rights to specific Jiki pages and allow Jiki pages be edited by authorised clients.

3.4.3 Storage Framework

When designing a security infrastructure for Jiki, three obvious storage requirements were required. These are:

1. Registered client list

2. Authorised Read-access client list

3. Authorised Edit-access client list

3.4.3.1 Registered client list

The registered client list is required to store all the names of clients who have registered with Jiki and their respective passwords. For security reasons, each client's password that is stored will be in an encrypted form. This list will be used to authenticate clients if they try to read or edit a Jiki page. It will also be used if the client tries to access the Security Policy Manager for a page.

3.4.3.2 Read-access client list

The Read-access client list is required to store the names of every Jiki page created and the names of clients who have authorised read-access rights to a specific Jiki page. This list will be used to authenticate a client if they try to read a Jiki page. It is possible that a client's name may not be associated with any Jiki page, in which case they would not have the authorisation to read any other client's page. In this case, the client would only be able to read the core architecture pages that do not require read-authorisation for any client. Conversely, the client's name may be listed after every page listed and therefore would be authorised to read every page created.

3.4.3.3 Edit-access client list

The Edit-access client list is used for a similar reason to the Read-access client list. It is also required to store the names of every Jiki page created and the names of clients who have authorised edit-access rights to a specific Jiki page. This list will be used to authenticate a client if they try to edit a Jiki page. The very same scenario might also occur here as with the Read-access list, i.e. it is possible that a client's name may not be associated with any Jiki page, in which case they would not have the authorisation

to edit any other client's page. Conversely, the client's name may be listed after every page listed and therefore would be authorised to edit every Jiki page created.

For all three storage requirements described, the use of either flat-files or a database would implement these. Both of these storage mechanisms have their advantages and disadvantages. However, after careful consideration, it was decided that the use of flat-files on the server would best suit these storage requirements. The reasons for this decision are:

1. The complexity involved for all three storage requirements are minimal.

2. The contents of the files are in simple plain-text format so the Jiki server-administrator can easily check the files for any problems, errors or simply out of curiosity.

3.4.3.4 Additional storage requirements

During the design of the security framework, it became apparent that an additional storage requirement was going to be needed. This is a temporary file to be used to store the contents of a new Jiki page before it is saved in the repository of all Jiki pages on the server. The reason this file is needed is as follows:

- The original Jiki web architecture does **not** allow for the creation of new Jiki pages by the client. Every client could read every existing page and there were no restrictions on who could edit a page. By allowing a client to create a new page using this present architecture, the contents of the new page would overwrite the contents of the **previous** page the client had read, before this new page is saved using its own page name.

- Using a temporary file to store the contents of a new Jiki page before it is saved will prevent the contents of the previous page from being overwritten.

This temporary storage file would also be located in the repository of Jiki pages on the server.

3.4.3.5 Storage file format

The file format for all the storage files used in the new security framework is `plain text`. Each of the files are described as follows:

1. **Registered Clients file**

   The Registered Clients file is a list of every client who is registered with the Jiki server. The list is composed of the client's name and their password, both separated by a colon. The password is stored in an encrypted form for security reasons[8]. The file is used whenever a client needs to be authenticated by the server (e.g. if they try to read a Jiki page or access the security policy manager for a page). **Figure 3.5** shows an example of this file.

```
collinmr:mn.5bfPlhmiQ2

olearycs:CbA866I8n7LHA

dobsons:9vvovW8DOdH4o

joe:joTp/xfcrWpEI

Gary:Gal00RZdIU6FM
```

Figure 3.5 Example of the Registered Clients file

[8] The MD5 message – digest encryption algorithm is used to encrypt the password

2. **Read – access file**

The Read – access file contains a list of the names of every Jiki page stored at the server. The list is composed of the page name and a sequence of client names that are authorised to read the page. The page name and the client names are separated by a colon. This file is used to keep a list of clients who are authorised to read specific Jiki web pages. If a client's name is not listed in this file after a particular page name, they are **not** authorised to read that page. **Figure 3.6** shows an example of the Read – access file.

```
JikiJikiJava:joe,collinmr

FindPage:joe

HardAtWorkOnJiki:joe

LinkInferencing:joe

PageTitleSynonyms:joe

PostOopsla98DcBof:joe

Mike9:collinmr,Mike,Katie,Peter,Gary

Test3:collinmr,Gary,olearycs,joe
```

Figure 3.6 Example of the Read - access file

3. **Edit – access file**

The Edit – access file is very similar to the Read – access file. It also contains a list of the names of every Jiki page stored at the server. The list is composed

of the page name and a sequence of client names that are authorised to edit the page. The page name and the client names are also separated by a colon. This file is used to keep a list of clients who are authorised to edit specific Jiki web pages. By default, if a client is authorised to edit a page, they are automatically authorised to read that page. Therefore, their name will also be associated with the page in the Read – access list. If a client's name is not listed in this file after a particular page name, then they are **not** authorised to edit that page. They *may* however have read rights for the page. **Figure 3.7** shows an example of the Edit – access file.

```
JikiJikiJava:joe,collinmr,Gary

FindPage:joe

HardAtWorkOnJiki:joe

LinkInferencing:joe

PageTitleSynonyms:joe

PostOopsla98DcBof:joe

Mike9:collinmr,Gary

Test3:collinmr,joe
```

Figure 3.7 Example of the Edit - access file

4. **Temp (New Page) file**

The Temp file is used to store the contents of a new Jiki page until it has been saved[9]. This is needed because if not used, the contents of the new page will overwrite the contents of the previous page. Using a temp file to store the contents of the new page until it has been saved will prevent this problem. The Temp file is also stored in the repository of Jiki pages on the server. **Figure 3.8** shows an example of what this temp file looks like.

```
'Welcome' to my new Jiki web page

Please browse through the page –
```

Figure 3.8 Example of the Temp (New Page) file

3.4.3.6 File Retrieval methods

The methods used to read the files explained in **Section 3.4.3.5** are standard disk Input/Output. The contents of all the files used are in plain text and are not in any unconventional configuration. Even with many registered clients, the contents of these files are relatively small. Therefore, reading and writing to these files using disk i/o is not expensive in terms of processing power and does not produce large overheads or time delays.

[9] **NOTE:** ALL Jiki web pages have different page names. The new security infrastructure will **not** allow any two pages have the same name.

3.5 Summary

This chapter has described the design of the security infrastructure to be integrated into the Jiki architecture. The reasons why particular design decisions were taken were also explained.

The chapter began with a description of the different security scenarios that might exist, which Jiki must provide mechanisms to deal with. Each of the mechanisms needed here have their advantages and disadvantages and these were explained in detail.

The chapter followed on with descriptions about the *Security Policy Manager*, the use of HTTP sessions and the MD5 message – digest encryption algorithm. The design decisions and reasons for choosing these to be used in the security framework were explained.

Finally, the high-level architecture of the security infrastructure was explained. This included an explanation of the front-end GUI to the system. A design of the storage framework followed this and how data would be saved, stored and retrieved. The chapter concluded with an explanation and description of the format of the storage files.

4. Jiki Security Implementation

4.1 Introduction

This chapter will discuss the implementation of the security infrastructure and the way it is integrated into the overall Jiki architecture. The chapter will give an explanation of the mechanisms used in the present architecture that allow Jiki pages to be edited and the advantages and disadvantages this provides. Following this, the technologies used to implement the new security framework will be discussed including the reasons why they were chosen. The chapter will then give some examples of GUIs that the client uses to interact with the server. Finally, the chapter will conclude with a summary.

4.2 Security component framework

Due to the generic and extensibility features of Jiki, the architecture provided a lot of avenues to approach the implementation of the new security infrastructure. New components could be developed to implement the design (the reader is referred to chapter three for the security infrastructure design) and have these integrate with the existing architecture components.

There are several main components required to implement the security features. The functions these must implement are:

- Registering new clients
- Creating new Jiki pages
- Authenticating a client

- Access to the Security Policy Manager
- Changing the administrator of a page
- General utilities

These main components will be used to implement the overall security infrastructure in Jiki.

4.2.1 Registering new clients

The component used to register a new client must accept a name and password from the client. The component must ensure that no registered client is using the same name that the new client selected. It must read every name in the file listing all the registered clients and compare each one against the name the new client has selected. If it is already being used, the new client must select another name.

Assuming the name is not being used, the component must encrypt the new client's password using the MD5 encryption algorithm. The name and encrypted password must then be **successfully** appended to the end of the file containing the names of every registered client.

4.2.2 Creating new Jiki pages

When a client tries to create a new Jiki page, the component must first authenticate them. The file containing a list of every registered client and their associated password will be used to authenticate the client. Only registered clients who have been authenticated can create a new page.

The client must also enter the name of their new page. The component must ensure that no other Jiki page is using this name already. To do this, it must check the Read-access file that contains a list of every Jiki page and the associated names of clients who have authorised read-rights. If a page already uses the file-name entered, the client must re-enter another name.

In order to assign read and edit rights to other registered clients for the new page, the component must display a list of every registered client and allow the author to select the names they wish to assign these rights to. Finally, a text area should then be displayed to allow the client to enter the content of their new page.

4.2.3 Authenticating a client

Several components will be involved with the authentication of clients. Before any client can read or edit a page, other than the core architecture pages (e.g. help, error, message pages, etc..), they must first be authenticated. Components that require the authentication of a client will first request the client's name and password. The name will be checked to see if it is listed in the file containing all the names of registered clients. If the name is **not** listed, the client is not a registered client and thus the component will not permit the client to proceed.

Assuming the name is listed, the component will encrypt the password using the MD5 encryption algorithm and compare this to the encrypted password stored with the name on file. If they match, this authenticates the client and they can then proceed.

4.2.4 Security Policy Manager access

The security policy manager is responsible for **all** permissions associated with a page. These permissions are the Read and Edit rights as well as any changes to the administrator for a page. A special component will be used to act as the security policy manager and must be able to deal with all of the above functions.

For security reasons, **only** the administrator of a page will be authorised to access the security policy manager. By default, the author of a page becomes the administrator of the page. The component must authenticate any client who tries to access the security policy manager. It does this by checking the Edit-rights file. In this file, a list of every Jiki page name is stored. Following each page name are the names of clients who are authorised to edit that page. When the page was created, its name was placed in this file with the author's name placed as the first name with edit-rights. A design

decision was taken to make the first name following the page name as the administrator of that page.

Therefore, when the component authenticates a client trying to access the security policy manager, it must check whether their name is the **first** name following the page name in the Edit-rights file. If it is not the first name or is not listed at all, the client will be denied access.

The component providing the security policy manager functionality will provide the administrator with functions to edit the names of clients with read and edit rights for a page. It will also allow the administrator to change their administration privileges and grant these to another registered client (section **4.2.5** explains this in full).

4.2.5 Changing Administrator

A design decision was taken to allow changes be made to the administrator of any Jiki page. Reasons for providing this function include:

- The administrator might not want to continue being the page administrator.
- The administrator might be leaving the department or company where the page is hosted and therefore will require a new administrator.
- There may be security implications why a new administrator for the page is needed.

Even when a new administrator has been assigned for a page, the component will ensure that the former administrator will continue to have read and edit rights. After the new administrator has been selected, this administrator will be the only client from that point onward who will have authorised access to the security policy manager.

4.2.6 General utilities

Due to the large functionality provided by the security infrastructure, there are many small functions that will be needed. For this reason, a decision was made to design a component whose specific role was to deal with the small and arduous functions that other components required.

The utilities component is designed to handle functions such as file input/output, formatting list contents, error checking, etc.. One big advantage of using such a component is the re-use of code. A lot of the other components perform the same operations so in order to maximise code re-use, a utilities component (a Java Bean Servlet) will be used to handle these operations.

4.3 Jiki pages

The present Jiki architecture allows clients to edit a page by displaying the page data in a text-area. The client can then add, edit or delete the data as they wish. As was explained in section **2.8.2**, the client does not use HTML tags to format the data. Instead, a pre-defined syntax for formatting page data was designed by the original developers of Jiki. Using this syntax, certain characters in specific sequences before and/or after text will format the page data.

In keeping with the overall architecture of Jiki, it was decided to use the very same method in the creation of new pages as used to edit pages. This was an option not provided by Jiki before the integration of the new security framework. This means that when a client creates a new Jiki page, a text area will be provided to allow them enter the page data.

4.4 GUIs

Introducing a security infrastructure into the Jiki architecture meant that there were many areas where clients needed to be authenticated. Extra functionality was also introduced such as the creation of new pages, a security policy manager and the

registration of new clients. All communication is passed between the client and server using HTTP and therefore a web-browser is used to allow them to interface and communicate. Various GUIs were designed for the new security framework that kept the look and feel of the original Jiki architecture GUI. Each GUI was designed to keep client input to a minimum. The principle GUIs for this security framework are:

1. New client registration
2. New Jiki page creation
3. Editing a page
4. Page administration

4.4.1 New client registration

Before a client can read or edit any Jiki pages, other than the core Jiki pages (help pages, error pages, etc..), they must be registered with the Jiki server using the new security framework. **Figure 8.1** in Chapter 8 shows the GUI designed to allow a new Jiki client register themselves.

4.4.2 New Jiki page creation

There are two scenarios when a new Jiki page can be created. This can be done by either:

1. **New-registered client**

   When a client registers with the Jiki server, they have an option to create a new page as they are registering. **Figure 8.2** in Chapter 8 shows the GUI designed for this scenario.

2. **Old-registered client**

   All clients that are registered with the Jiki server can at any time create a new page. The process of creating the page includes selecting clients with authorised read and edit rights. **Figure 8.3** in Chapter 8 shows the GUI designed for this scenario.

4.4.3 Editing a page

When a client wants to edit a Jiki page, they **must** have the authorisation to be able to do this. Only the administrator of the page can select the clients with these read and edit rights. Upon successful authentication, the client can then edit the data on the page. **Figure 8.4** in Chapter 8 shows the GUI designed for allowing authorised clients to edit a page.

4.4.4 Page administration

Every Jiki page is administered using a *security policy manager*. Only the administrator of the page has authorised access to the security policy manager. The manager is used to edit the administrative settings for the page. These settings include the names of clients with read rights and edit rights. The security policy manager also allows the page administrator to select a new administrator if required. If a new administrator is selected, after exiting the security policy manager, the present administrator will no longer be able to access the settings again. However, they will continue to have read and edit rights for the page. **Figure 8.5** and **Figure 8.6** in Chapter 8 show the GUI designed for the security policy manager. Only the administrator has authorised access to this GUI.

4.5 JSDK and Servlet runner

All of the components used in the original Jiki architecture and in the new security framework are Java Bean Servlets. In order for certain web servers to support the running of Servlets, Sun™ Microsystems developed JSDK (Java Server Development Kit). JSDK provides all the necessary *class* files and *jar* files required for running Servlets. Jiki is one such web server that requires JSDK.

The original Jiki architecture uses JSDK 2.0 for supporting Servlets. A later version of JSDK (2.1) was made available by Sun since the development of Jiki. In order for the new security framework to integrate correctly with the Jiki architecture, the older version of JSDK (2.0) was used. This older version had to be used since changes made to the *class* files and *jar* files in the newer version of JSDK resulted in incompatibilities with existing classes on the server. If the newer version (JSDK 2.1) were used, the server would crash as a result of certain methods being invoked.

JSDK provides an application that when run, allow web servers who previously could not support the use of Servlets, to now do so. It is called *servletrunner*. Jiki uses the *servletrunner* to state what port number the server is running, where to find properties files and in what directory and its sub-directories to find all the necessary files and classes to support the use of Servlets.

4.6 Summary

This chapter discussed the implementation of the new security infrastructure. The chapter began by looking at the methods used to develop the existing architecture and the advantages and disadvantages associated with these. The use of components to develop the new security infrastructure was then discussed. A detailed explanation was given for each of the main components required and the main functions they provide.

The chapter followed with a discussion on the format of new Jiki pages and how a client can use the pre-formatted text, designed by the original Jiki authors, to create a new page in Jiki.

The chapter then explained the decisions for the GUI designs. Some of these GUIs were designed for specific reasons in order to integrate smoothly with the existing architecture. Some examples of these GUIs can be seen in chapter 7.

Finally, the chapter concluded with a discussion on using JSDK 2.0 and its *servletrunner* for providing support for the use of Servlets in web servers.

5. Evaluation

5.1 Introduction

This chapter will evaluate the security framework that was integrated into the Jiki architecture. Each of the main security scenarios identified in the design (Chapter **3**) will be evaluated to see how well the security framework deals with them. The chapter will also evaluate the security policy manager to see how well it handles the administration of Jiki web pages. There will be an evaluation of Jiki compared to other web servers and a discussion on whether it really does have advantages over them.

5.2 Security scenario evaluation

Each of the different security scenarios identified in the design will be evaluated to see if they have been implemented correctly in the security framework.

5.2.1 Single Author

The original Jiki architecture did not provide a service to clients that allowed them to create a new Jiki page. However, in the new security framework, this service is provided. Before a client can create a page, they must register themselves with Jiki. When creating a new page, they can select other registered Jiki clients who they wish to grant read or edit rights. Only these clients will be authorised to either read and / or edit that page.

This feature works well and the new framework authenticates any client before they can read or edit a page.

5.2.2 Collaborative authoring

The new security framework only allows **one** client to be the author of a new Jiki page. Collaborative authoring was a service supposed to allow more than one client to author a page. A decision not to support collaborative authoring was made because after a client creates a new page, they can grant edit rights to other clients for that page. Therefore, by having edit rights to a page, this is very similar to co-authoring the page in the first instance. All clients with edit rights can edit the page freely and make any changes they would have made when the page was first created.

5.2.3 Communities

Communities are used where clients have different rights for different Jiki pages. This means that a client might have edit rights for some pages but only read rights for others. They may of course have no rights for some pages. Different communities can be used therefore to keep a list of client names that have different rights for different pages.

The security framework doesn't really implement the use of communities to its exact definition. Instead, a Read-permission and Edit-permission file is used to keep a list of client names that have read and edit rights for specific pages. Each file contains the name of **every** Jiki page created. If a client name is listed with the name of the page, then they have either read or edit rights for that page.

Even though this is a kind of workaround for the proper use of communities, it does work well in the security framework according to its design.

5.2.4 No restrictions

When a page has no restrictions, it means every client can read and edit the page. The security framework easily implements this when the author of a page grants edit rights to every registered Jiki client. Having edit rights means a client automatically has read rights too. Therefore, every client can read and edit the page.

This was straightforward to implement and works without any problems in the integrated Jiki architecture.

5.3 Evaluation of the Security Policy Manager

The security policy manager was designed to allow the administrator of a Jiki page to change the security settings. **Only** the administrator has authorised access to this manager. The design also catered for allowing an administrator to relinquish their rights as the administrator for a page and grant these rights to another client.

The implemented security policy manager in the security framework did work very effectively. By default, when a client creates a new page, they automatically become the administrator for the page. When authenticated, the manager would then present the administrator with a list of clients who *do* and *do not* have read and edit rights for that page. The administrator can then edit these lists if needed.

Another service that the manager provides to the administrator is to allow them relinquish their administration rights and grant these to another client. Again, a list of every registered client is given and the administrator can select the name of the client who they wish become the new administrator. After exiting the manager, the administrator will no longer be able to access these settings again. Only the new administrator will. However, they will still continue to have read and edit rights for the Jiki page.

5.4 Using HTTP sessions

In section **3.2.3**, the reasons are explained why the use of a HTTP session was selected over the use of cookies. Using HTTP sessions in the implemented security framework proved to be a very good design decision and worked very well. Once a client had been authenticated, they did not have to authenticate themselves again (as long as their browser remained open). The HTTP session object retained the client's details and validated them in all the areas where authentication is required. It is quick and runs without the client's knowledge.

5.5 Encryption algorithm evaluation

The encryption algorithm used to encrypt client passwords is the MD5 Message – Digest algorithm. This algorithm was very successful and worked well in the security framework. Section **3.2.4** explains the algorithm in a lot more detail and the design reasons why it was chosen.

One of the initial main concerns about using this algorithm was that it would be slow to encrypt the passwords. If this were the case, it would obviously have a cascading effect slowing other parts of the framework. However this did not occur and the algorithm did work fast. Any delays that did occur were very minor and negligible. Since the client's details were entered onto a form on a web page, any delays that did occur as a result of the encryption process merely seemed like ordinary delays we are all used to when using the Internet everyday. This was by chance a lucky way of masking any delays caused by the algorithm to the client.

5.6 Evaluation of the use of flat files

The new security framework for Jiki uses three flat files. These files are:

- User names and encrypted passwords
- List of clients with Read – access rights
- List of clients with Edit – access rights

All of these files are in standard ASCII `plain-text` format. Section **3.3.3** explains the reasons why a design decision was taken to use flat files over a database. Unfortunately, the use of flat files to store data instead of a database does mean that the architecture is slightly slower with the integrated security framework. This is a result of the disk I/O overhead produced when the server needs to authenticate a client and read or write to any of these three files.

The use of the HTTP session object to retain the client's name and password does help a little to speed the system up a little, but not anything very significant.

The use of these flat files is probably the main contributor to slowing the security framework. However, one way of improving this situation might be to cache the contents of the password and Read / Edit access files. This would involve a small re-design of the security framework so that specific contents of these files are stored in memory with periodic flushing of any changes back to permanent storage (disk). Until such a re-design is made, advances in computer hardware should help reduce the amount of disk '*thrashing*' from slowing the system – in the short term.

5.7 Scalability issues

The new security framework designed to integrate into the Jiki architecture applies to Jiki pages **on a single server**. Since Jiki is a web server, it runs on a single designated machine and serves requests from clients requesting Jiki pages. Therefore all security information is passed **only** between the client and the Jiki server. No other third party is involved (i.e. other Jiki servers). This means that all Jiki servers are completely independent of each other. A client may therefore be authorised to request and read a page from one server, but not another.

As more and more Jiki servers are established and set running, this can become a problem. Clients therefore will need to be authenticated by every separate server. However, if the Jiki architecture were re-designed to incorporate inter-communication between servers, it would be easy to allow them share security details. Files could then be shared that contain client names, passwords, Read and Edit access rights.

The main focus of scalability of the new security framework itself only really centres on the number of registered clients with Jiki. The registering of clients with the server is a feature that was introduced by the security system. It was not a service provided by the original architecture. All other scalability issues relate the overall architecture and are not a consequence of the integration of the new security system.

As more and more clients register with Jiki, the file containing their names and encrypted passwords gets larger. If any of these clients create one or more Jiki pages, then the Read and Edit – access files will also grow larger. As a result of these large files, it means:

1. The time taken to search the file containing client names and encrypted passwords to find a specific client name for authentication will increase.

2. Locating specific Jiki page names in the Read and Edit – access files to read the associated names of authorised clients will also increase.

These are two very important issues. As the files grow larger, the time taken to search these will increase. This slows the entire Jiki web server. One solution to help avoid this problem is to use a database to store the details contained in the files. Efficient querying mechanisms would then maintain a high level of speed in the framework even with large amounts of data.

5.8 Jiki v Other web servers

The one question that is mostly asked about Jiki is why would someone want to use it. What advantages does it provide over other popular web servers? Jiki has one main advantage over most other web servers. This is the ability to create new web pages and edit the contents of an existing web page in a text area at the client side and send the data back to the server using a HTTP POST.

This is a feature very rare in web servers. Some web servers can be set up to allow a client invoke a CGI script at the server to create a new page. This process allows the client to enter the data of a page and have this sent to the web server using a HTTP POST method. The server then runs a CGI script to create the new page, passing the page data sent by the client as a parameter to the script. The page can then be saved at the server.

This is a very non-efficient and complex way of creating a new web page. When the CGI script is run, a lot of overhead is produced, slowing the server considerably. Another problem exists in that for every single new web page created, the same CGI script is run. This is a waste of valuable resources and CPU time.

This is where Jiki provides a solution to the problem. Jiki does **not** use any CGI scripts to create new web pages. Instead, a series of Java Servlet components are used to GET and POST the new page using HTTP. The format for the data on every Jiki page is `plain-text`. Therefore, the client can enter the contents of every new page into a text-area located inside a Form. When the client enters the page contents, the Form containing the text-area is then POSTed to the server. The server then saves the new page containing the contents of the text-area.

This is a very unique way of creating new web pages and is the very same process for each new page. As can be seen, when the client sends the new page to the server, the server does not pass this to a CGI script or any other program. All that is done is the page gets saved in permanent storage (on disk). Therefore, the processing load and overheads produced are very low and nothing as high as those with other web servers.

This is the same process when editing an existing page with Jiki. The server will send the page to the client, and the contents of the page will be displayed in a text-area on a Form. The client can then edit the contents of the page and the Form will POST this back to the server.

Taking a critical look at the way Jiki provides this service, it does work well and effectively. Before the security framework was integrated into the architecture, there were serious security issues in using a technique like this. However, the new security framework has solved this problem by only allowing authorised people to read and / or edit pages.

5.9 Jiki Data Format and XML

In **Section 2.8.2**, it was seen how the data stored on every Jiki page uses a pre-defined format. This format is very different to conventional HTML tags used to format data

on most web pages. The designers of Jiki designed this data format in a way that is similar to a condensed form of XML. It is very generic in that it can be changed, manipulated and rendered in several different ways. This means that if a client wanted to define and use their own specific data format, they could edit the file containing the rules that specify the present Jiki data format. For example, instead of placing data inside a double-quote to make it appear in *italics*, this could be changed so that a single-quote is used instead.

Alternatively, the client could define their own rule for formatting data. They could specify a particular character or character-set within which any data placed will be formatted to that stated rule. This is in a way similar to using a DTD in XML. Instead of specifying what order tags appear and the attributes associated with those tags as used in a DTD, here the client can specify what data format is associated with each data format rule.

Looking at the way Jiki formats its page data, it really does appear very similar to a scaled-down version of XML.

5.10 Summary

This chapter discussed the evaluation of the new security infrastructure. The chapter began by evaluating how well the security system implemented the main scenarios outlined in the design. It was seen that it did implement these quite well.

The chapter followed with a discussion on how well the security policy manager worked and the ways it provided access to administrators of pages to change the security settings. This was something completely new to the Jiki architecture and it also worked very well. A short discussion of the use of HTTP sessions and the use of the MD5 encryption algorithm followed.

There was a critical analysis of the use of flat-files by the security framework and this included how well these worked when the server was running. This led to the evaluation of how scalable the architecture was with the new security system. It was seen that there was really only one definitive concern in the security system that

would cause scalability problems. This was the problem of the flat files becoming very large.

The chapter concluded with a discussion on the usefulness of Jiki and why someone might want to use it over other well-known web servers. This included an explanation of the services Jiki provides that other web servers do not. This mainly concerned the services it provides by allowing clients to create and edit web pages in a unique way. A brief comparison of the data format used for Jiki pages and XML follows this.

6. Conclusion

6.1 Introduction

The Jiki web server is free to download and use by anyone. It is located at [Jiki.org]. New versions of Jiki are available to download whenever updates are made to it or extra features are added. The designers and developers of Jiki encourage users to download the web server and to design and add any new services or features. This is because they have made all the Jiki source code and documentation freely available.

6.2 Achievements

Designing, developing and integrating an entire security infrastructure into the Jiki architecture was a challenging and exciting project. Obviously due to time constraints, not all the work that was initially hoped to be achieved was delivered. However, the new security integrated into the architecture is a massive improvement to the way it was. Not alone is there a now security service, but extra functionality has also been added. This includes

1. A registration process for clients. Jiki now has a list of the names of all clients who send requests to it for pages (except for the core Jiki pages where no authentication is required e.g. Jiki main page, help pages, etc.).
2. The ability for clients to create new pages.
3. Read and Edit permissions for pages.
4. A security policy manager that allows the administrator of a page to change the read and edit permissions.

The original architecture used Java components that interrelate with each other to operate. The new security infrastructure was developed using similar components. These components were specially designed not only communicate with the security components, but also the existing ones. This had to be done well in order for the successful integration of the new system.

6.3 Future work

The unique openness and extensibility of Jiki makes it perfect for adding new services and features. Looking at the security infrastructure that was integrated into the Jiki architecture, there are other extra services that could be added. These include

- The use of a database instead of flat files. The files used to store the client names and encrypted passwords, and the Read and Edit – access rights might be changed and integrated into a database instead.
- Design a service that permits authorised clients to edit only the data on a page and **not** the hyperlinks. A similar service might then provide authorised clients with **full** edit rights may edit the data and the hyperlinks.
- Design some way of storing the contents of a Jiki page in XML. This would mean including the use of an XML parser to parse the page contents and then convert this into HTML using XSL.

The security framework developed for Jiki does work well with its present features. Adding some or all of the features above would make it even better and more secure.

Jiki is a standalone web server and will run on some designated machine. This means that the possibility exists within an organisation that two or more Jiki web servers could be running at the same time. Clients would therefore have to register with each one of them separately, neither Jiki server having any knowledge of the other. This

would become frustrating for clients, especially if they wanted to edit pages and have to authenticate themselves on each server.

Looking at this scenario, a possible future development solution would be the design of some kind of distributed facility that would allow multiple Jiki servers to communicate with each other and share files and resources. A single database could be used to store the Jiki pages, client names, encrypted passwords, access-lists, etc.. and each Jiki server would be able to access the data stored there.

This is just one of the many possible future extensions that could be made to Jiki. Its open-source software and documentation allows for anyone to design and develop lots of new and useful features. Let's hope Jiki promotes many other great ideas for the Internet.

7. Bibliography

[Kin98] Joseph R. Kiniry – 1999. Jiki Documentation. Department of Computer Science, Caltech University, California, USA

http://www.jiki.org

[W3C1] R. Fielding (UC Irvine), J. Gettys (Compaq/W3C), J. Mogul (Compaq), H. Frystyk (W3C/MIT), L. Masinter (Xerox), P. Leach (Microsoft), and T. Berners-Lee, World Wide Web consortium (W3C.org) – Hypertext Transfer Protocol – HTTP/1.1 RFC 2616, June 1999

[W3C2] Berners-Lee, T., Fielding, R. and H. Frystyk, "Hypertext Transfer Protocol – HTTP/1.0", RFC 1945, May 1996.

[W3C3] Carpenter, B. and Y. Rekhter, "Renumbering Needs Work", RFC 1900, February 1996.

[Kha97] Rohit Khare, and Adam Rifkin, "Weaving a Web of Trust". (1997). Web Security : A Matter of Trust. O'Reilly.

[Corm97] Andrew Cormack, White Paper "Web Security", 1997, Joint Information Systems Committee

http:// www.jisc.ac.uk/acn/authent/

[Oreilly99] Caroline O'Reilly, M.Sc. Dissertation, "*BeanBag*, An Extensible Framework for Describing, Storing and Querying Components", Trinity College Dublin, 1999

[Lowe99] David Lowe and Wendy Hall, "Hypermedia and the Web. An Engineering Approach". (1999). Wiley.

[Jiki.org] Jiki Documentation – CVS checkout Open Source Software

http://www.jiki.org

[Distrib.org] Distributed Coalition

http://www.distributedcoalition.org

[Architag98] The Architag International Corporation (1998)

http://www.architag.com/solutions/980106-01.html

[XML98] Norman Walsh, White Paper "Technical Introduction to XML", October 1998, XML.com

http://www.xml.com/pub/98/10/guide1.html

[Greenspan98] Jay Greenspan, Introduction to XML for HotWired, 13 October 1998

http://www.hotwired.com/webmonkey/98/41/index1a.html?tw=xml

[Gorman98] Trisha Gorman, (1998), 20 questions on XML

http://builder.cnet.com/Authoring/Xml20/index.html

[Rein99] Lisa Rein, (March 1999), The Quest for an XML Query Standard

http://www.xml.com/xml/pub/1999/03/quest/index.html

[Bloch99] Cynthia Bloch, (1999) The Java™ Tutorial, A practical guide for programmers, "Servlets"

http://java.sun.com/docs/books/tutorial/servlets/index.html

[Servlet00] The Java™ Web Server, Servlet Tutorial, (2000)

http://jserv.java.sun.com/products/java-server/documentation/webserver1.1/servlets/servlet_tutorial.html

[Hall] Marty Hall, Servlets and JavaServer Pages 1.0

http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/

[ApacheSec] Apache Server Security Documentation

http://www.apacheweek.com/features/userauth

[Sun00] Sun – Java Tutorial Documentation, (2000)

http://java.sun.com/docs/books/tutorial/

[Rivest92] Ron Rivest, “The MD5 Message-Digest Algorithm”, RFC1321, MIT Laboratory for Computer Science and RSA Data Security, April 1992

http://www.w3.org/People/Raggett/security/rfc1321.txt

8. Appendix

8.1 Screen shots

Figure 8.1 New client registration

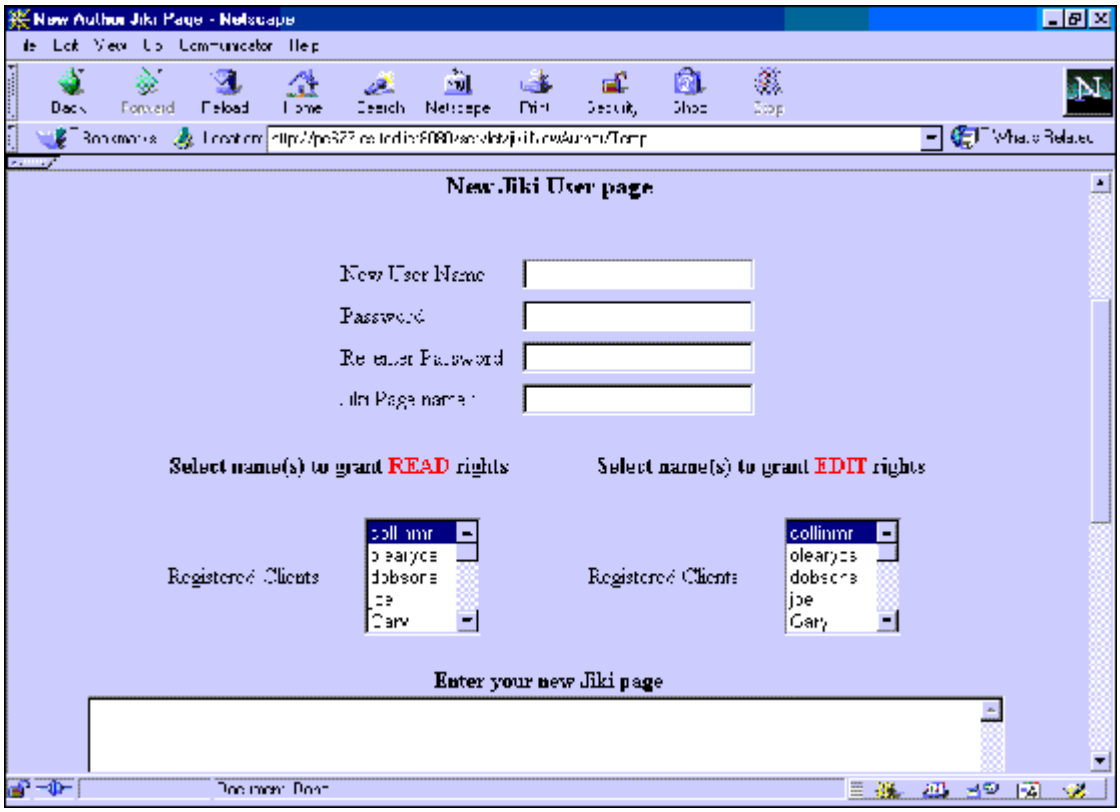

Figure 8.2 New registered client Jiki page

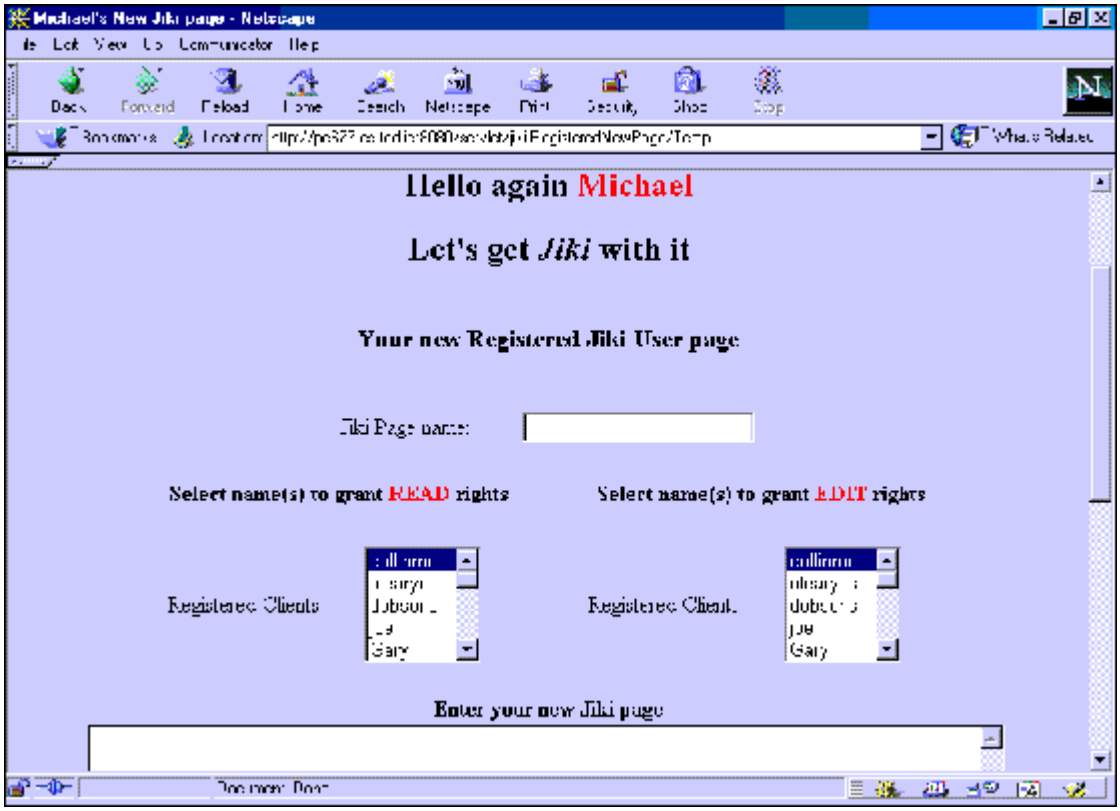

Figure 8.3 Old registered client Jiki page

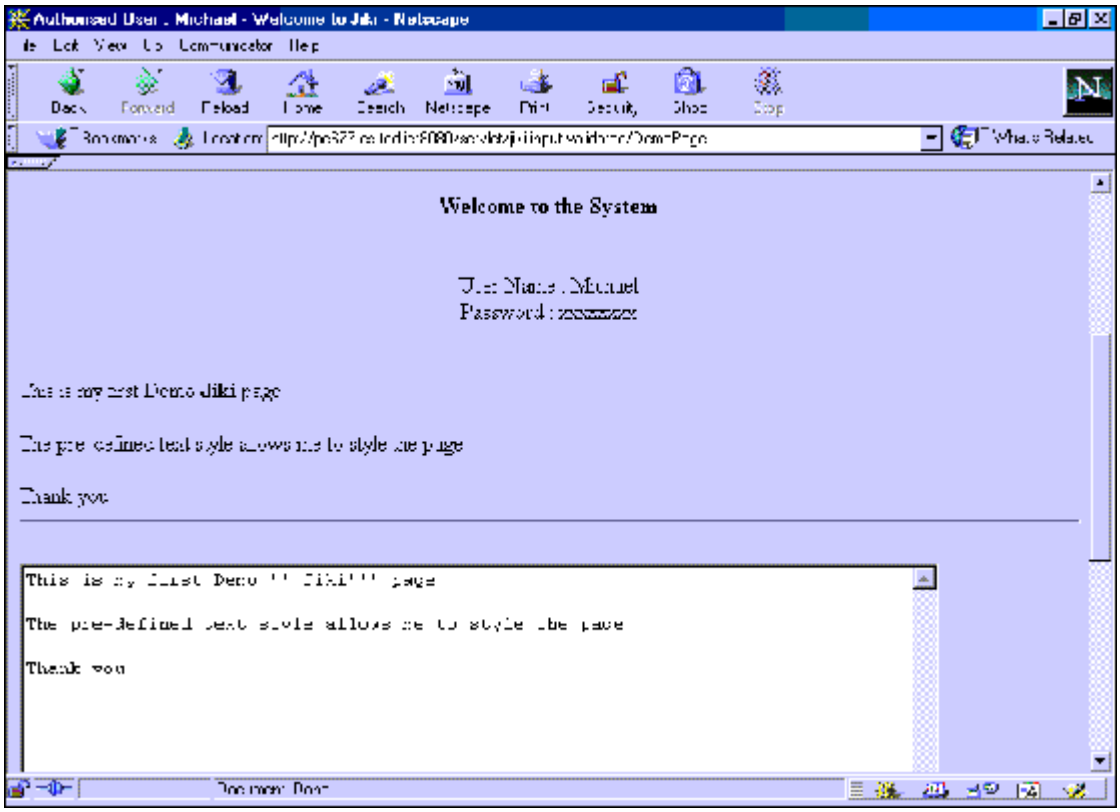

Figure 8.4 Authorised Jiki page editing

Figure 8.5 Security Policy Manager GUI

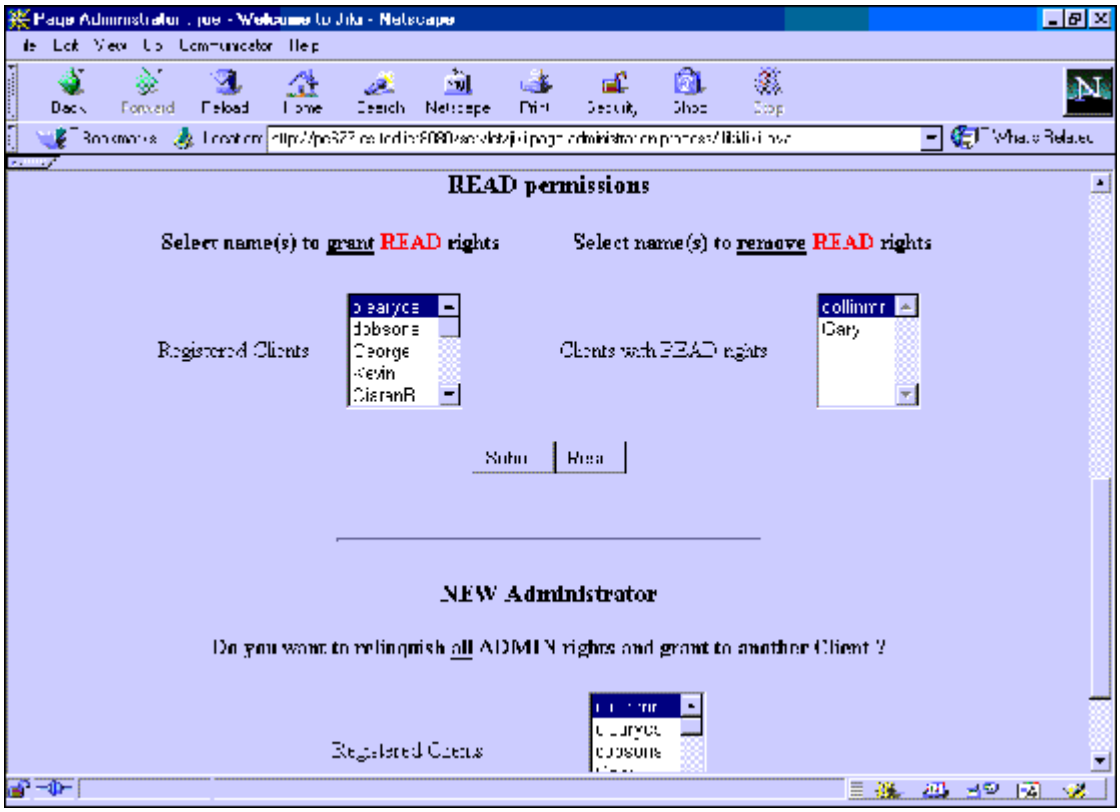

Figure 8.6 Security Policy Manager GUI