Component Architecture and Toolkit for Composing Functionally Scalable Plug-n-Play Web Servers

Daliso Zuze
University College
Oxford

Supervisor: Bernard Sufrin

A Dissertation Submitted in Partial Fulfilment of the Requirements for the Degree of

MASTER OF SCIENCE

in the University of Oxford
Computing Laboratory
Trinity, 2005

Abstract

This dissertation reports on our effort to provide software developers a clear and concise framework with which to assemble web servers. Manifestations of web servers are now seen in a wide range of applications and appliances, however, manufacturers have few standard options for assembling and embedding them in their products. Servers like Apache would be overkill for most application's requirements. As a result, software developers are left to build custom servers from scratch. This adds to the development cost of the original application, conceivably acting as a barrier to their greater use. With the increased accessibility of network resources, we believe that there is merit in building a component toolkit for assembling web servers that is easy to understand and use.

We approached this by examining the processing requirements of HTTP requests as defined in the standard specification document. This led to the discovery of an ontology of components suitable for meeting these requirements, but flexible enough to scale functionally to different deployment scenarios. In this document we also discuss briefly the concepts relevant to component driven software design. The Z specification language proved useful in helping us clarify the meaning of the components that we eventually built.

The outcome of the project was an implementation of the component framework in the Java 2 Platform Standard Edition (J2SE) version 5.0. To demonstrate the utility of our toolkit, we present two case studies that involve the development of an application requiring web services. In each case we describe the solution that our toolkit avails.

Acknowledgements

I would like to thank my supervisor, Bernard Sufrin, for his enthusiastic support and well considered advice during the project. I also wish to thank my family for their continued support and encouragement throughout my academic endeavours.

This dissertation marks the end of a truly memorable year at Oxford. It has been a time of intellectual and spiritual growth, made possible by much appreciated fellowship with my friends at the Computing Lab, at University College, and around Oxford.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation for a Web Server Toolkit

Web services have traditionally been delivered using heavyweight servers acting as gateways to central applications. This was justified when communication costs were high and network resources were scarce.

Improvements in network performance and a steady reduction in usage costs have made network resources available to a wider range of applications and devices. This class of entities includes mobile devices and simple standalone applications that perform well defined functions. For example, a web enabled micro controller in your home lighting system could be used to remotely switch lights on or off, or to report usage statistics over the past week. An appointment scheduling application could have a web interface plugged on to its functions for setting up new appointments or retrieving schedule information.

The protocols that govern Internet and Web communication have demonstrated resilience and effectiveness in facilitating the exchange of messages between disparate systems. We believe that the HTTP communication standards can be used effectively by these applications and devices that now have easier access to physical and wireless networks. Application developers would benefit from a flexible mechanism for embedding this capability into their existing applications.

1.2 Goals of The Project

In response to this need, this project aims to develop a component toolkit that can be used to implement and add web server capabilities onto any piece of software. The objective is to allow for *programmer ease of use*, *scalability*

of functionality through functional composition of components, as well as *flexible extensibility* of service offerings.

The reference document used to specify the required capabilities of a standard web server is the official HTTP/1.1 specification[4]. However, there are usage scenarios that do not require the HTTP/1.1 specification to be implemented in its entirety. Mainstream web servers and client software are obliged to implement the entire specification, but where the protocol of interaction can be explicitly agreed between clients and servers, a subset of the specification need only be implemented. Our toolkit will allow for the creation of standards-compliant servers as well as stripped down servers tailored to specific needs.

Achievements The main achievement of our project was the design and implementation of the architecture described in chapter four. Readers who wish to skip directly to this chapter may do so without loss of any critical information. However, chapter two provides a background to the relevant topics that gave insight and influenced our style of approach in the design of the toolkit, while chapter 3 briefly looks into the HTTP specification and uses the Z specification language to provide a basic abstract view of the system components.

1.3 Organisation of The Document

Chapter 1 *Introduction*:
Introduces the motivation and objectives of the project

Chapter 2 *Relevant Software Patterns and Architectures*:
Looks into the existing software engineering architectures and patterns that are used to develop component software and web servers. This chapter provides a broad background to the design decisions taken in our implementation

Chapter 3 *Requirements and Specification*:
Outlines the particular behavioural features that the envisaged family of servers will need to demonstrate and provides a basic high-level formal specification in Z

Chapter 4 *Proposed Architecture*:
Provides a high level view of the various components, connectors and assembly patterns of the developed solution

Chapter 5 *Implementation*:
Describes our implementation in Java of the abstractions and patterns introduced in Chapter 4.

Chapter 6 *Usage Case Studies*:
Demonstrates the effectiveness of the implementation using two case studies

Chapter 7 *Conclusion and Future Work*:
Summarises the achievements of the project and suggests future work.

Chapter 2

Relevant Software Patterns and Architectures

The aim of our project was to provide a way for developers to customise and assemble web servers. The purpose of this chapter is to lay the foundation for the design decisions that will be expressed in the rest of this document. The strengths and weaknesses of current and emerging technologies will be discussed as well as their applicability to the design of web server components.

A principal challenge in software engineering is to design systems that are flexible enough to evolve over time and/or with changing user requirements. It is important to find ways of optimising programmer effort. This is particularly important if we need to develop a family of related software products.

Engineering disciplines traditionally deal with this productivity challenge by producing general purpose components that meet a variety of user needs based on the way that they have been configured and assembled. Attempts have been made to apply the same technique to the construction of software systems. These efforts have affected the way that the computational parts of a program are organised and the way that software is conceptualised at a high level.

The following sections provide the relevant background to the design approach of our web server toolkit.

2.1 Programming Techniques

The principal relevance of this chapter to our project lies in the use of software components described in the next section. However, it is worth noting that regarding software reuse and assembly, a number of approaches have been

taken to improve the utility of software modules at the programmatic level.

The most prevalent modularity technique is *Object Orientation*, which promotes the clustering of program functionality into self contained units called objects. Specialisations of these objects can be defined and used in different applications to achieve the aims of software reuse. However, experience has shown that the grouping boundaries imposed by Objects are in fact not applicable to all situations and the clean separation of concerns often needs to be violated.

The second technique that we shall mention is *Aspect Orientation*. This technique introduces as its focal point the notion of an Aspect, which is a capability of a system that is achieved by threading together fragments of pre-existing software modules. Aspect oriented programming overcomes the rigidity of object orientation and increases the utility of software modules across a wider range of applications.

The last technique that we shall mention is called *Feature Orientation*[1]. This is a more user centric approach to software development. It similarly makes use of code fragments across separate modules, but the emphasis is more on the definition of product families. With Feature Oriented Programming, products can be defined as the composition of a number of features. The advantage of this is that end user requirements are more efficiently captured and fulfilled.

The implementation of our web server toolkit made use of Object Oriented techniques as we did not discover the need to employ 'Aspects' and 'Features'.

2.2 Components and Architectures

In the previous section, we introduced object-oriented techniques for modularising software, its limitations and new techniques for dealing with these limitations. In this section, we will look at the notion of components, how they relate to objects, and the activities involved in systems architecture. This section provides a foundation for the concept of building 'families' of software products and how the structure of a system influences the ability for it to remain relevant over a longer period of time.

2.2.1 Software Components

Software Components are the building blocks of a software architecture. A component is any static abstraction that defines 'plugs' or well defined ways of communicating with it[12]. Components are more general than objects. The main distinction between *object orientation* and *component orientation*

is that object orientation is primarily concerned with partitioning code into cohesive units and the relationships between these units. *Component Orientation* on the other hand is concerned with how different high level parts of the system *compose* with one another to fulfill a particular set of requirements. Szyperski[15] describes objects as *units of instantiation*, contrasted with components which are *units of deployment*. This distinction is underpinned by the fact that components are static and do not have transient states. Objects, on the other hand, by their very nature are initialised to a certain state and have their internal representation altered throughout their life in a program. Moreover, a software component can be represented by any useful abstraction, or set of abstractions, that can be used in a compositional way[1].

Composition of components can be seen as the manufacturing stage of software development. Nierstrasz and Dami[12] define software composition as *"The process of constructing applications by interconnecting software components through plugs"*.

Through the composition of components, we can generate software *families* and software *product lines*. Families and product lines are similar concepts, but there is a subtle difference between the two. We define a software family as *a collection of applications that share a core set of features and which are constructed using a common set of components*. A product line on the other hand is a set of applications that are *targeted for sale to a well defined market segment*.

When designing a software family out of components, it should be possible to fulfill different requirements by composing the components in different fashions. Components do however have context dependencies and constraints that must be respected for a composition to be valid. One such dependency is a set of *Type* constraints imposed on the direct communication between two distinct components. This constraint would require the output data type of the sender's message, the data type of the connector transporting the message and the data type that the receiver expects to receive to all match.

The primary reason for using components is to aid in the reuse of programmer effort. Based on the non-static nature of a system's requirements, component orientation is seen as the most economical way to approach software development.

The aim of our project was to build a family of servers, and as such, we treated components as first class entities while designing our base system. The process of discovering the ontology of components to use in a family of products is an iterative one, unless you can determine at first hand all the

[1]Components could include things like functions, procedures, modules[12]

usage scenarios. However, in the chapter 3, we propose a set of components that can accommodate the assembly of any envisaged web server product.

2.2.2 Software Architectures

Software architecture is an activity mostly concerned with structure and the evolution of a piece of software in the face of changing user requirements. Software Architecture is the specification of the components of a system and the communication between them. Architecture deals with the structural aspects of a system rather than the computational aspects.

The emergence of software architecture is rooted in the discovery of successful patterns and styles of deployment. There was no detailed analysis as a precursor to the discovery of these patterns, they have just consistently proven to be the most successful in practice.

Work has been done to formalise the field of software architecture through the formation of languages used to describe them. Architectures described using these languages can then be analysed for validity and can be used to run simulations that reveal how a system would fare under different deployment scenarios.

This section looks into the features of these formalisms. This provides a basis for the style of specification we use to describe our family of servers in the next chapter.

Architecture Description Languages

An Architecture Description Language (ADL) is used to specify a software architecture in a way that allows it to be analysed. This language must allow for the declaration of components in a system, their topology and the constraints that the components must respect. In addition to this, an ADL should allow for properties to be associated with components. Some of these properties will specify invariants on the components that must not be violated while others will describe the *non-functional* requirements that are desired but not essential in a valid system. The purpose of an ADL is to ensure that high level design decisions are based on verifiable facts about the behaviour of systems under precise configuration conditions.

A number of Architecture Description Languages have been developed that are specific to individual domains. For example Adage is an ADL designed specifically for describing avionics and guidance systems, C2 can be used for describing User Interfaces and Rapide is a language designed to allow simulations to be carried out on a system based on its description.

In general, ADLs all support[6]:

- the precise definition of components
- multiple component interfaces
- encapsulation of sub-systems as components
- specification and analysis of non-functional requirements
- the definition of constraints on system operation.

Component Ontologies

The architectural design process for a family of systems involves the determination of the ontology of components that are deemed essential for any particular implementation. This collection of components forms the "alphabet" that will be used in the composition of instance members of this family of systems. Architecture description languages, in general, have a common set of elements which are used in the construction of an architecture[6]. Below is a description of these elements:

Components as described earlier, represent units of computation and data stores.

Connectors represent the communication between components. This representation is general and could eventually be constrained to a particular method of communication, such as pipes or procedure calls.

Systems are any configuration of components and connectors. Systems, in general, can be embedded in other systems forming sub-systems.

Properties provide extra information about the attributes of each component and connector, such as the protocol used for a connection or the rate of processing of a component.

Constraints are special types of properties that must be respected for a system to be valid.

Styles represent an architectural pattern. This is based on the components and connectors used as well as the mode of composition. An example would be a Pipe and Filter system.

Composition and Assembly

There are different styles that can be employed in the composition and assembly of components. We shall discuss three, namely, functional composition, blackboard composition and object extension.

Functional composition is the most common form. This involves the encapsulation of components as functions having the ability accept components as parameters[12]. This style of composition is used in Pipe and Filter assemblies such as those used to capture the essence of a data flow application as shown in figure 2.1. Here composition takes place in the direction of the arrow flows, i.e. the *Supplier* and *Transformer* compose to form a component that supplies data in the format appropriate for consumption by the *User*.

Figure 2.1: A Data Flow Composition

Blackboard systems utilise a shared space which all the components in the system are allowed to write to and read from. The components represent agents that collaborate to solve a particular problem. The entire blackboard system can be viewed as the composition of the all the agents working together to solve the problem at hand.

Object extension is the third kind of composition we shall mention. In this case the components considered are classes. In object oriented programming, one is able to create hierarchies of classes. The further one moves down the hierarchy, the more specialised the classes become. If we define the capabilities of a class by the public methods that it exposes, then we can say that each sub-class is its super-class composed with a class containing the remaining methods found in the sub-class but not the super-class.

An Example of an ADL

We will briefly take a look at the features of an ADL named ACME[6], under development at Carnegie Mellon University. This ADL was designed to provide a general way to describe the architecture of any system. Looking at the way families of systems are defined in ACME will give us a good reference for our specification in chapter 3 of the components needed in a family of web servers.

ACME conforms to the general view of ADLs described above and in particular allows system designers to specify a system according to the following factors:

Structure - How the components and connectors are organised

Properties of Interest - Information about the parts of the system that can be useful in reasoning about its overall behaviour

Constraints - Properties that ultimately determine the set of valid system configurations

Types and Styles - The definition of architectural patterns

In an ACME system description, the main types of entities described are components and connectors. A system is defined as a graph in which the nodes represent components and the arcs represent connectors. Each system can contain sub-systems, which would be the internal representation of a component. An example[6] of a simple client server system defined in ACME is shown below:

```
System simple_cs = {
  Component client = { Port sendRequest }
  Component server = { Port receiveRequest }
  Connector rpc    = { Roles {caller, callee} }
  Attachments : {
    client.sendRequest to rpc.caller ;
    server.receiveRequest to rpc.callee }
}
```

Ports identify points of interaction of a component with its environment while *Roles* identify the way a connector interfaces with its environment. The attachments define the topology of the system by identifying the way components and connectors are attached to each other.

Acme defines *Properties* as extra information about the parts of a system that can aid in the analysis of a system configuration. This information has nothing to do with the structure of the system and hence is ignored by the Acme compiler. However, this information can be used by tools that simulate the overall performance of a system, such as a tool to measure expected throughput. As an example, a client component could be annotated with the following property:

```
Component client = {
  Port sendRequest;
  Properties { requestRate : float = 17.0;}
}
```

Constraints are similar to properties but are used to describe values that set boundaries on the set of valid configurations that a system can be transformed into. These constraints are expressed as predicates over any aspect of the design specification. In Acme, constraints fall into two categories: Invariants and Heuristics. Invariants cannot be violated in a valid system while Heuristics are desired properties but not essential. Constraints may also make use of various functions that evaluate the truth of certain properties of the architecture or return sets of architectural entities. For example connected(comp1, comp2) evaluates to true if there is a connector joining comp1 and comp2, and SystemName.Connectors returns the set of connectors in system SystemName.

Types and Styles are used to create templates for system elements and configuration patterns that have been found to reoccur. For example, a template for a Client component could be described as follows:

```
Component Type Client = {
  Port Request = {Property protocol: CSPprotocolT};
  Property request-rate: Float;
  Invariant Forall p in self.Ports @ p.protocol = rpc-client;
  Invariant size(self.Ports) <= 5;
  Invariant request-rate >= 0;
  Heuristic request-rate > 100;
}
```

Instances of the above Type would automatically inherit all Client's properties and constraints. In addition to Types, Acme allows for the definition of families, which are a collection of Types that are valid in a particular set of Architectures. An Acme family consists of a set of property and structural types, a set of constraints and a default structure. The property and structural types determine what can be used in the family and the constraints determine how they can be used. The default structure determines the most basic configuration that is possible for an instance of the family.

The ACME ADL described above provided insight into the issues relevant to architectural design. It also gave us some basic guidance in our use of the Z language in chapter 3, where we describe abstractly our envisaged family of web servers.

2.3 Architecture for Web Services

In this section, we look into two aspects of a web server architecture that were pertinent to our project. Firstly, we discuss opportunities for user customisation of a web server application, and secondly, we discuss techniques used to deal with varying scales of concurrency in a web server application. Since our objective was to build a family of servers configurable to varying deployment scenarios, the component framework that we developed needed to deal with these two aspects.

2.3.1 Customisation and Extensibility

We consider customisation as the ability to add, remove or change the types of requests serviced within a particular application. In our discussion, we examine the mechanisms used in the Apache web server that allow for the server functionality to be customised.

The Apache web server allows for its configuration through the use of configuration files that control aspects such as mime types, virtual hosts, etc, and through the writing and dynamic inclusion of custom built server modules.

The Apache web server processes requests according to a series of predefined and static stages that are executed sequentially by a request processor. A minimal implementation of all these stages is contained in a single component called the server *Core*. In addition to the Core, the server comprises a collection of modules, each providing handlers to process data pertaining to predefined request processing stages. These module handlers are invoked by way of implicit invocation from the server's request processor.

If a user of the system wishes to make use of a module, it must be registered with the server using special directives in a configuration file. The module can either be compiled into the server using *configuration* and *make* utilities, or the module can be loaded dynamically when the server is started. More than one module can contain handlers for the same stage of processing, however, precedence is given to `AddModule` directives that appear later in the configuration file.

The stages of processing defined by the Apache server are the following:

① **Post-Read** occurs after the headers have been read and the request is checked to be a valid HTTP request

② **URI Translation** does the job of converting a URI to an internally usable filename path

③ **Header Parsing** allows any number of filters to modify the request headers

④ **Access Control** checks whether or not the URI requested is allowed

⑤ **Authentication** checks whether the user is who he/she claims to be (checks password)

⑥ **Authorisation** checks whether or not the user is allowed to access the URI

⑦ **Type Checking** determines the type of the response's content so that the correct content handler will be called

⑧ **Content Handling** generates the response

⑨ **Fixups** modify the response before it is shipped to a client

⑩ **Logging** keeps a record of server activity

Certain stages allow for the invocation of multiple handlers, such as authentication, whereas others will allow only one of the declared handlers to execute. The former are said to execute *inclusively* and the later *exclusively*. Whenever a handler is called to handle a stage, it will return an `int` value status code. These codes correspond to *OK*, *Declined* and *Done*. OK means that the processing completed successfully, Decline means that the handler will not handle the request and Done means that processing completed and no further handlers should be invoked.

In summary, the Apache server's architecture allows for extensibility in the way the server deals with particular stages. End users can write their own handlers and arrange for these to be loaded dynamically and invoked by the request processor at the appropriate time, however, the processing stages remain static and cannot be altered by the end user.

2.3.2 Concurrency

We are interested in the ability of a server application to accommodate a high number of concurrent users. We would like to investigate architectural patterns that can be used to achieve this. In this section we investigate several approaches taken to this problem by different server implementations.

Firstly, we look at the approach taken by the Apache server. Apache copes with the need for concurrency by creating replicas of the entire request/response processing mechanism. On Unix platforms this is achieved

by having a set number of child processes running, all of which are allocated connections from a single queue. In this case, each instance of the server has its own private address space. On Windows, Apache makes use of Multi-threading of server instances to achieve concurrency. In this case there exists a shared memory space between all the threads.

Another approach to concurrency taken by the Flash web server[14] is the use of what is called an *Asymmetric Multi-Process Event Driven Architecture* AMPED. This approach overcomes the overhead of context switching by using a single thread that is responsible for executing a series of stages in the processing of a request. In the event of increased server demand, the event thread will accommodate the processing of all the requests by interleaving the processing of the individual server stages. This in effect simulates multiple threads but retains more control over the sharing of computer resources. The Flash Server makes use of mechanisms to simulate non-blocking I/O to ensure that the server does not become idle during times of slow I/O such as disk activity.

A third approach to dealing with concurrency draws from both the multi-process/multi-threaded and event driven architectures. The *Staged Event Driven Architecture* (SEDA), is designed to cope with massive concurrency and simplify the construction of well-conditioned services[16]. Well-conditioned means that as load increases beyond the capacity of the server, the performance degrades linearly and the performance loss is distributed fairly between the current tasks being executed.

A SEDA architecture consists of a network of *stages* that are joined by explicit event queues. A stage can be described as a self-contained component consisting of an *event handler*, an *incoming event queue* and a *thread pool*. Each stage is managed by a *controller* that consumes a batch of tasks from the event queue and distributes these among the available event handler threads.

Multi-threading in SEDA is done at the stage level and this allows for greater control in fine tuning the performance of the server. The controller is capable of making adjustments to the rate of processing by controlling the number of threads in a stage or the number of tasks consumed from the queue in a cycle. The use of event queues to separate the stages leads to greater decoupling with the execution of threads constrained to a single component. This, in turn, allows for better performance analysis of the processing pipeline and provides opportunity for self-tuning at each stage. As with AMPED, SEDA makes use of non-blocking I/O operations to remove unpredictable latency. In addition to improved load conditioning, SEDA presents a highly intelligible structure that aids in ease of programming.

The ideas presented by SEDA, particularly the use of self-contained stages,

greatly influenced the design of our family of servers.

2.4 Summary

In this chapter, we provided a background to the design factors important in the building of our family of web servers. We began by discussing the techniques used in software development to achieve modularity and reuse of code, we then went on to look at the concept of building software through the composition of general purpose components. We demonstrated how high level architectural abstractions are used to tie all the pieces together and we discussed the formalisms used to describe these architectures. Finally, we briefly surveyed architectural features found in mainstream and experimental web servers. The next chapter will begin the formal conceptualisation of our server design.

Chapter 3

Requirements and Specification

This chapter presents the functional requirements of the servers we expect to build with our component toolkit. We also provide a basic abstract architectural description of the envisaged components codified in the Z specification language.

The basis of our requirements are detailed in the HTTP/1.1 specification[4]. This chapter does not attempt to reproduce the entire specification, but outline the important parts that were most relevant to our design decisions. Following this, the toolkit requirements are discussed together with a set of sample usage scenarios that illustrate the extent of flexibility required of the implementation. Finally, we formalise the requirements and constraints using Z and provide a high level specification of the system components.

3.1 The HTTP Protocol

The HTTP protocol was designed to be resilient to problems with data transmission and is thus a *stateless* protocol. This means that each request message contains within it all the information that the server needs to generate an appropriate response message. These messages are sent as byte streams between a client application and a server application. A message is one of the following two types:

Request. This is a message sent from a client to a server. These messages follow the following format:

```
<method> <request-URL> <version>
<headers>

<entity-body>
```

Response This is a message sent from a server to a client. These messages follow the following format:

```
<version> <status> <reason-phrase>
<headers>

<entity-body>
```

A brief description of each part of the message is provided below:

- *method* refers to one of `GET, PUT, DELETE, POST, HEAD, TRACE, OPTIONS and CONNECT`. These are the standard operations that a client can request a server to perform. The most commonly used methods are `GET` and `POST`. A `GET` method requires the server to send back a Response containing the contents of the resource referred to by the *request-URL*. A `POST` method will request the server to execute some operation referred to in the *request-URL*. The information from the Request's *entity-body* is provided as input to this operation.
- *request-URL* is the unique name of the resource that the method should be applied to.
- *version* is the HTTP specification version to which the message format conforms.
- *status* is the three digit code that indicates the outcome of the requested action.
- *reason-phrase* is the human readable version of the status code.
- *headers* are the collection of (property name, value) pairs that provide extra information about the message. For example, there are headers that describe the length of the message's content and provide authentication information for a Request. For a complete list of all the header types, please refer to the HTTP/1.1 specification document[4]
- *entity-body* is data that represents 'input' if present in a Request and 'output' if present in a Response. It could comprise of form data, html pages, or any data file.

An example of an HTTP Request is shown below:

```
GET /docs/ HTTP/1.1
Host: localhost:81
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-GB; rv:1.7.10)
            Gecko/20050717 Firefox/1.0.6
Accept: text/xml,application/xml,application/xhtml+xml,text/html;
        q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-gb,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

An example of an HTTP Response is shown Below:

```
HTTP/1.1 200 OK
Date: Sat, 20 Aug 2005 18:06:02 GMT
Content-length: 276
Server: Daliso's Experimetal Server Codename: Cheetah 0.9
Keep-Alive: timeout=5
Cache-Control: max-age=0
Accept-Ranges: bytes
Connection: Keep-Alive
Content-Type: text/html

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <meta content="text/html; charset=ISO-8859-1"
 http-equiv="content-type">
  <title>test.html</title>
</head>
<body>
<span style="font-weight: bold;">Hello
World</span>
</body>
</html>
```

3.1.1 Persistent Connections

The protocol of behaviour for both clients and servers is covered extensively in the HTTP/1.1 specification[4]. In this section we discuss one particular requirement pertaining to persistent connections, as it played a significant role in the design of our server architecture.

A *connection* in the context of our discussion can be described as the virtual 'wire' through which the client application and the server application

communicate. The first step in any client-server interaction is the establishment of a connection.

There are two ways that a client and a server can make use of connections. In the first scenario, a new connection is established for each individual request that is issued by the client. In the second, multiple requests are sent by the client through a single connection. The client is permitted to send requests through the connection without waiting for responses from previously sent requests. This is known as *pipelining*. A connection used in the second way is called a *persistent connection*. Since there is an overhead involved in the establishment of a connection, and since web pages often contain many linked images, the use of persistent connections can drastically improve the efficiency of the protocol.

The specification stipulates that both the client and the server can assume that all new connections will be persistent. Connections remain open until either the client indicates that it will send no additional requests through a connection or the server indicates that it will accept no additional requests through a connection. The client inserts a header `'Connection:Close'` on the last request it intends to send and the server issues the same header on the last response that it will send through a particular connection.

The specification also requires that all messages include a `content-length` header that indicates the length of the data in the entity body. This is particularly important for responses since there would otherwise be no way of determining the end point of one message and the start of the next message.

Regarding the pipelining of messages, the server is required to ensure that responses for pipelined requests are returned in the same order that they were requested. Client applications that pipeline requests before they are certain of a persistent connection must be prepared to have the connection closed unexpectedly. If a connection does close while requests are being pipelined, the client should reopen a new connection but first confirm that the connection is persistent before pipelining. In addition to this, clients should not pipeline requests containing methods that are non-idempotent (such as POST). Finally, the HTTP/1.1 specification also states that a client may open more than one pipelined connection to a server, but should normally not maintain more than two open connections at any one time.

3.2 Deployment Requirements

This brief section will look at the range of implementation uses that our toolkit will enable its users to satisfy.

Embedded in Host Application In this scenario, developers using the toolkit will be able to add a web interface to any existing application by including our library of standard classes and simply providing a custom implementation of the class that will translate requests to internal system operations and translate the results of those operations into a response sent back to the web client. In this scenario, the developer may determine that part of the HTTP specification is not required and configure the server infrastructure to meet the particular requirements at hand.

As an example, consider the following:

A stand-alone application for quoting the cost of translating documents has the ability to count the number of words in a plain text file. An application developer would like to design a web based front end to this application which accepts documents uploaded in a `POST` request and generates an HTML page containing the quotation as its response. This application is unlikely to need to accommodate many concurrent users or require the use of caching. In addition to this, if we know that the only method being requested is `POST`, we do not need to worry about implementing special behaviour of the server in response to the other methods. Our toolkit would allow the application developer to add on this functionality with the inclusion of a few classes and the custom implementation of a few methods.

Standards-Compliant Stand-Alone Server In this scenario, the server is composed of components configured to ensure that the server will respond appropriately to any standards-compliant web client. The implementer will be able to configure the server infrastructure to cope with the expected concurrency demands that will be placed on it.

3.3 Modularity Requirements

One of the main goals of our project was to achieve enhanced ease of programming in the construction of any web server. To meet this goal, we required a highly intelligible way of modularising the system. To this end we state one of the system requirements as *"A flexible and extensible message processing workflow"*. In addition to this, we define major stages in the system as comprising of:

- Accepting connections from clients
- Transforming streamed data into system manipulatable objects

- Determining Security Authorisation of a Request
- Servicing a Request by Generating a Response
- Decoration of Messages for Maintenance of State, Presentation, etc
- Caching Services
- Delivering the Response to the Client

3.4 Architectural Description Using Z

This section provides a high level description of the different parts that make up our server toolkit and the relationships between them. We begin by looking at the definitions of the messages that flow through the system. We then examine the connectors and components, and describe the operations that can be performed on them. Finally, we describe an instance of a complete system.

3.4.1 Messages and Connections

A ***Message*** is an abstraction of the Requests and Responses that flow internally in the system. The Z schema for a message is shown below. We have first introduced a type named *Byte* that represents a Byte of raw data.

[Byte]

$$
\begin{array}{|l}
\textit{Message} \\
\hline
\textit{headers} : \operatorname{seq} \textit{CHAR} \pfun \operatorname{seq} \textit{CHAR} \\
\textit{entityBody} : \operatorname{seq} \textit{Byte} \\
\textit{version} : \operatorname{seq} \textit{CHAR} \\
\hline
\textit{version} = \text{“}\textit{HTTP}/1.1\text{”} \\
\#\textit{entityBody} > 0 \Rightarrow \text{“}\textit{Content} - \textit{length}\text{”} \in \operatorname{dom} \textit{headers} \\
\hline
\end{array}
$$

The *Message* has a mapping of headers, which represents a (key,value) pair. It also contains an entityBody, which is the ‘payload’ of the message, comprising input or output data. A message is also associated with a particular HTTP version. We have decided to constrain our family of servers to HTTP/1.1 compliance. A message that contains data in its entity body must also declare the length of the data as one of its headers.

A ***Request*** is a type of message that also has a *method*, denoting the command the server should perform, and a *requestURL*, denoting the resource that the method should be applied to.

Request

$$
\begin{array}{l}
Message \\
method : \text{seq}\, CHAR \\
requestURL : \text{seq}\, CHAR \\
\hline
method \in \{GET, POST, PUT, DELETE, OPTIONS, \\
\qquad HEAD, TRACE, CONNECT, ERROR, CHALLENGE\}
\end{array}
$$

Below are two special extensions of the Request definition introduced to simplify the flow of messages in the system. A Request, on its way to being serviced, can be transformed into an Error if something goes wrong, or a Challenge if authentication fails. Their definitions are shown below.

ErrorRequest

$$
\begin{array}{l}
Request \\
\hline
method = \text{“}ERROR\text{”} \\
\text{“}CLIENT\_TEXT\text{”} \in \text{dom}\, headers \\
\text{“}ERROR\_TYPE\text{”} \in \text{dom}\, headers
\end{array}
$$

ChallengeRequest

$$
\begin{array}{l}
Request \\
\hline
method = \text{“}CHALLENGE\text{”} \\
\text{“}REALM\text{”} \in \text{dom}\, headers \\
\text{“}MESSAGE\text{”} \in \text{dom}\, headers \\
\text{“}AUTH\_TYPE\text{”} \in \text{dom}\, headers
\end{array}
$$

For the purpose of defining a *Response*, we need to define a free type, Digit, as follows:

$$Digit ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

A ***Response*** is a type of message that also contains a status, comprising three digits, and a reasonPhrase, that is a human readable interpretation of the status.

Response

$$Message$$
$$status : \text{seq}\, Digit$$
$$reasonPhrase : \text{seq}\, CHAR$$
$$\#status = 3$$

A ***Connection*** represents the *'virtual wire'* between the client and the server. An IpAddress represents any node on the Internet. Below is the definition of a Connection followed by operations for initialising, closing and using a connection.

[IpAddress]

Connection

$$source : IpAddress$$
$$open : true \mid false$$
$$portnum : \mathbb{N}$$

ConnectionInit

$$Connection'$$
$$pnum? : \mathbb{N}$$
$$src? : IpAddress$$
$$source' = src?$$
$$portnum' = pnum?$$
$$open' = true$$

CloseConnection

$$\Delta Connection$$
$$open = true$$
$$open' = false$$
$$source = source'$$
$$portnum = portnum'$$

SendData

$$\Xi Connection$$
$$dataToClient? : \text{seq}\, Byte$$
$$open = true$$

$$
\begin{array}{|l}
\textit{ReceiveData} \\
\hline
\Xi \textit{Connection} \\
\textit{dataFromClient}! : \operatorname{seq} \textit{Byte} \\
\hline
\textit{open} = \textit{true} \\
\hline
\end{array}
$$

3.4.2 Infrastructure and Operations

We shall now describe some of the infrastructure for our component assembly, including the operations that can be performed on each of the entities.

Firstly, we will describe the core of our connectors, which is a buffer that can store a variable number of messages. Below is the generic definition of a message buffer, an initialisation schema, and operations for reading and writing data to it.

$$
\begin{array}{|l}
\textit{Buffer}[\textit{MSG}] \\
\hline
\textit{items} : \operatorname{seq} \textit{MSG} \\
\textit{max} : \mathbb{N} \\
\hline
\#\textit{items} \leq \textit{max} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\textit{BufferInit}[\textit{MSG}] \\
\hline
\textit{Buffer}' \\
\textit{max}? : \mathbb{N} \\
\hline
\textit{items}' = \langle\rangle \\
\textit{max}' = \textit{max}? \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\textit{Read}[X] \\
\hline
\Delta \textit{Buffer}[X] \\
\textit{data}! : X \\
\hline
\textit{items} \neq \emptyset \\
\textit{items} = \langle \textit{data}! \rangle \frown \textit{items}' \\
\textit{max} = \textit{max}' \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
Write[X] \\
\Delta Buffer[X] \\
data? : X \\
\hline
\#items < max \\
items' = items \frown \langle data? \rangle \\
max = max' \\
\hline
\end{array}
$$

Another fundamental concept in our infrastructure is that of a Port. A Port represents any interface that a component or connector has with its environment. Ports in our architecture must be typed, and therefore we have provided a global generic definition of a Port below.

$$
\begin{array}{|l}
\hline
[X] \\
Port : X \\
\hline
\end{array}
$$

Also required in our definitions of components and connectors is the concept of Null, defined globally below.

$$
\begin{array}{|l}
\hline
[X] \\
NULL : X \\
\hline
\end{array}
$$

Connectors There are two types of connectors that can be used to link components. The first one, an ***Asynchronous connector***, shown below, allows for a number of messages to be buffered and thus allows more than one write to occur before a read and vice versa, depending on the condition of the buffer. This helps to decouple a fast writer from a potentially slow reader.

$$
\begin{array}{|l}
\hline
AsynchronousConnector[X] \\
Buffer[X] \\
size : \mathbb{N} \\
in : Port[X];\ out : Port[X] \\
\hline
max = size \\
\hline
\end{array}
$$

A ***Synchronous connector*** has all the properties of an asynchronous connector but with the size of the buffer constrained to one. This type of connector can be used when the reader is guaranteed to operate at least as fast as the writer. It ensures that both reader and writer rendez-vous at each step of processing that involves input or output.

$$
\begin{array}{|l}
\hline
SynchronousConnector[X] \\
AsynchronousConnector \\
\hline
size = 1 \\
\hline
\end{array}
$$

Below are schema definitions for writing to and reading from a connector:

$$
\begin{array}{|l}
\hline
ConnectorWrite[X] \\
\Delta AsynchronousConnector[X] \\
in? : X \\
\hline
in? = in \neq NULL[X] \\
in' = NULL[X] \\
\#items < max \\
items' = items \frown \langle in \rangle \\
max = max' \\
out = out' \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
ConnectorRead[X] \\
\Delta AsynchronousConnector[X] \\
out! : X \\
\hline
out = NULL[X] \\
items \neq \emptyset \\
items = \langle out' \rangle \frown items' \\
max = max' \\
in = in' \\
out! = out' \\
\hline
\end{array}
$$

Please note that the above definitions do not model the temporal behaviour of a synchronous connector. It would not be practical in Z to attempt to model the process of the connector *waiting* when either a reader or a writer is unavailable to enable the operation to succeed. The above provides a snapshot of what a successful operation would look like.

Components In our system, components represent active objects that can be started and stopped. Therefore, we define a component state as follows:

$$
\begin{array}{|l}
\hline
Component \\
started : true \mid false \\
\hline
\end{array}
$$

StartComponent

$\Delta Component$

$started = false$
$started' = true$

StopComponent

$\Delta Component$

$started' = false$
$started = true$

A generic *Producer* is a kind of a Component that has a typed output interface with its environment as shown below:

Producer$[X]$

$Component$
$out : Port[X]$

A *Consumer* is a Component that has a typed input interface with its environment:

Consumer$[X]$

$Component$
$in : Port[X]$

There are kinds of components that have both input and output interfaces. These components effectively convert from one type of message to another. The generic definition of a *Converter* models these kinds of components as shown below.

Converter$[X, Y]$

$Consumer[X]$
$Producer[Y]$

Some components will not perform any change to the static type of a message as it passes through it. This is an *Identity* component and is defined below.

Identity$[X]$

$Converter[X, X]$

We can now move on to some specific examples of the component types that we have described above. Our architecture will comprise a set of components each carrying out a specific task in the chain of processing required to service a Request.

The first kind of component that we shall look at is named a ***Dispatcher***. This kind of component is a Producer since, from the point of view of the system, it is the component that originates all work in the pipeline.

DispatcherState

$$Producer[Connection]$$
$$conn : \text{seq } Connection$$
$$0 \leq \#conn \leq 1$$

DispatcherInit

$$Dispatcher'$$
$$started' = true$$
$$out' = NULL[Connection]$$
$$conn' = \langle\rangle$$

AcceptConn

$$\Delta Dispatcher$$
$$newconn? : Connection$$
$$started = true = started'$$
$$conn' = \langle newconn? \rangle$$
$$conn = \langle\rangle$$
$$out = out'$$

DispatcherOutput

$$\Delta Dispatcher$$
$$out! : Connection$$
$$started = true = started'$$
$$out = NULL[Connection]$$
$$conn \neq \langle\rangle$$
$$conn' = \langle\rangle$$
$$out! = out' = head\ conn$$

We define a Dispatcher component as the composition of the operation that accepts new connections and the operation that sends connections to the environment.

$$Dispatcher \mathrel{\widehat=} AcceptConn \fatsemi DispatcherOutput$$

The next kind of component that can exist in an assembly is named a ***Transformer***. This component conceptually converts Connections to Requests.

TransformerState

$$Converter[Connection, Request]$$
$$requests : \mathrm{seq}_1\ Request$$
$$con2reqs : Connection \pfun \mathrm{seq}_1\ Request$$

TransformerInit

$$Transformer'$$

$$started' = true$$
$$requests' = \langle\rangle$$
$$in' = NULL[Connection]$$
$$out' = NULL[Message]$$

Below is the definition of the operation of the Transformer that actually generates the Requests from the Connection. Note that the Transformer is always expected to produce at least one Request. The MakeRequests operation could generate a sequence of all *normal* Requests or a sequence where the last Request is an *Error Request*[1].

MakeRequests

$$\Delta Transformer$$
$$in? : Connection$$

$$started = true = started'$$
$$in? = in \neq NULL[Connection]$$
$$in' = NULL[Connection]$$
$$requests = \langle\rangle$$
$$requests' = con2reqs(in)$$
$$out = out'$$

[1]See page 27 for details of an Error Request

$$
\begin{array}{|l}
\hline
\textit{TransformerOutput} \\
\Delta \textit{Transformer} \\
\textit{out}! : \textit{Request} \\
\hline
\textit{started} = \textit{true} = \textit{started}' \\
\textit{out} = \textit{NULL}[\textit{Message}] \\
\textit{requests} \neq \langle\rangle \\
\textit{requests} = \langle \textit{out}' \rangle \frown \textit{requests}' \\
\textit{in} = \textit{in}' \\
\textit{out}! = \textit{out}' \\
\hline
\end{array}
$$

$$\textit{Transformer} \mathrel{\widehat=} \textit{MakeRequest} \fatsemi \textit{TransformerOutput}$$

An ***Authoriser*** is a component in our framework that inputs and outputs requests and is thus classified as an Identity.

$$
\begin{array}{|l}
\hline
\textit{AuthoriserState} \\
\textit{Identity}[\textit{Request}] \\
\textit{doneRequest} : \textit{Request} \\
\textit{reqMap} : \textit{Request} \pfun \textit{Request} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{AuthoriserInit} \\
\textit{Authoriser}' \\
\hline
\textit{in}' = \textit{NULL}[\textit{Message}] \\
\textit{out}' = \textit{NULL}[\textit{Message}] \\
\textit{started}' = \textit{true} \\
\textit{doneRequest}' = \textit{NULL}[\textit{Message}] \\
\hline
\end{array}
$$

The DoAuthorise operation below generates the appropriate Request from the input. This generated Request could be the original Request, an Error Request or a Challenge Request.

DoAuthorise

$\Delta Authoriser$
$in? : Request$

$started = true = started'$
$in? = in \neq NULL[Message]$
$in' = NULL[Message]$
$doneRequest = NULL[Message]$
$doneRequest' = reqMap(in)$
$out = out'$

The Output operation below transfers the generated Request to the output port.

AuthoriserOutput

$\Delta Authoriser$
$out! : Request$

$started = true = started'$
$out = NULL[Message]$
$doneRequest \neq NULL[Message]$
$out! = out' = doneRequest$
$doneRequest' = NULL[Message]$
$in = in'$

$Authoriser \mathrel{\widehat=} DoAuthorise \fatsemi AuthoriserOutput$

A ***Responder*** component actually performs the servicing of the Request and is classified as a Converter from Request to Response.

ResponderState

$Converter[Request, Response]$
$doneResponse : Response$
$rspMap : Request \pfun Response$

ResponderInit

$Responder'$

$in' = NULL[Message]$
$out' = NULL[Message]$
$started' = true$
$doneResponse' = NULL[Message]$

DoRespond

$\Delta Responder$
$in? : Request$

$started = true = started'$
$in? = in \neq NULL[Message]$
$in' = NULL[Message]$
$doneResponse = NULL[Message]$
$doneResponse' = rspMap(in)$
$out = out'$

ResponderOutput

$\Delta Responder$
$out! : Response$

$started = true = started'$
$out = NULL[Message]$
$doneResponse \neq NULL[Message]$
$out! = out' = doneResponse$
$doneResponse' = NULL[Message]$
$in = in'$

$Responder \mathrel{\widehat=} DoRespond \fatsemi ResponderOutput$

A ***Deliverer***, from the point of view of the system, is a Consumer. It is a sink for all Responses produced in the system. The Deliverer interfaces with the network via the operating system to arrange for the delivery of the Response data back to the client.

DelivererState

$Consumer[Response]$
$byteMap : Response \pfun \operatorname{seq} Byte$

DelivererInit

$Deliverer'$

$in' = NULL[Message]$
$started' = true$

$$
\begin{array}{|l}
\hline
DoDeliver \\
\Delta Deliverer \\
dataToClient! : \mathrm{seq}\, Byte \\
in? : Response \\
\hline
started = true = started' \\
in? = in \neq NULL[Message] \\
in' = NULL[Message] \\
dataToClient! = byteMap(in) \\
\hline
\end{array}
$$

$Deliverer \mathrel{\widehat=} DoDeliver$

Proxies are often deployed in networks in order to improve their performance. A Proxy component is one that is authorised to act on behalf of another. From the point of view of a client, a Proxy acts on behalf of a Converter. From the point of view of the Converter, the Proxy acts on behalf of the client. Therefore, a Proxy has four interfaces as defined in the schema below.

$$
\begin{array}{|l}
\hline
Proxy[X, Y] \\
Converter[X, Y] \\
subOut : Port[X] \\
subIn : Port[Y] \\
\hline
\end{array}
$$

A ***Cache*** is a construct that is used to store a mapping of character sequence *(keys)* to Response.

$$
\begin{array}{|l}
\hline
Cache \\
Map : \mathrm{seq}\, CHAR \rightarrowtail\!\!\!\!\!\!\!\!\rightarrow Response \\
\hline
\#\,\mathrm{ran}\, Map = \#\,\mathrm{dom}\, Map \\
\exists f : Request \twoheadrightarrow \mathrm{seq}\, CHAR \bullet \\
\quad \forall x \in \mathrm{dom}\, Map \bullet \exists r : Request \bullet x = f(r) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
CacheInit \\
Cache' \\
\hline
domMap' = \emptyset \\
\hline
\end{array}
$$

Having defined the concepts of Proxies and Caches, we can easily describe a ***CachedProxy*** as a Proxy that has a Cache, as in the schema below.

CachedProxy

$$Proxy[Request, Response]$$
$$theCache : Cache$$

CachedProxyInit

$$CachedProxy'$$
$$CacheInit$$

$$started' = true$$
$$in' = out' = subIn' = subOut' = NULL[Message]$$

3.4.3 System Descriptions

Now that we have defined all the constituent parts of a family of servers, we can use a schema notation to define any particular architecture. Below is an example of a server assembly:

System

$$D_1, D_2 : Dispatcher$$
$$T : \mathbb{P}\ Transformer$$
$$R : Responder$$
$$C_1 : SynchronousConnector[Connection]$$
$$C_2 : ASynchronousConnector[Request]$$
$$C_3 : SynchronousConnector[Response]$$
$$W : Deliverer$$

$$\forall\, t \in T \bullet D_1 \gg C_1[in/out, out/in] \gg t$$
$$\qquad \wedge\ D_2 \gg C_1[in/out, out/in] \gg t$$
$$\forall\, t \in T \bullet t \gg C_2[in/out, out/in] \gg R$$
$$R \gg C_3[in/out, out/in] \gg W$$

The above schema declares the collection of components and connectors that comprise this particular system. The constraint part of the schema states that the connector (C_1) links the two Dispatchers to all the Transformers. It states that the second connector (C_2) links all Transformers to a Responder and the third connector (C_3) links the Responder to the Deliverer.

3.5 Summary

In this chapter we looked at the rules that governed the design of our framework and toolkit for web server construction. We began by highlighting the most relevant parts of the HTTP/1.1 specification. We then went on to look at some desirable functional and non-functional requirements of the envisaged server toolkit by looking at deployment scenarios and desirable modularity aspects. Finally, we used the Z specification language to give a basic abstract description of the server components and the constraints on them. The Z definitions served to help us refine our concepts of the necessary objects and components in the system.

In the next chapter, we provide a concrete description of the server architecture that we ultimately implemented. Chapter 5 will go further to detail the constructs used in the Java™ programming language.

Chapter 4

Proposed Architecture

In this chapter, we explain the architectural style that we have used to address the requirements of the previous chapter. Following this structural paradigm, the components of the system will be built and the rules for their assembly formulated.

4.1 'Pipe and Filter' Functional Composition

The nature of the Request-Response message passing communication style combined with the provisions within the HTTP specification that cater for the inherent statelessness of web communication, suggest that a Unix style 'pipe and filter' approach to component assembly would be most appropriate. This approach exhibits the following characteristics:

- The filters can be treated as black boxes facilitating loose coupling and separation of concerns
- Each filter item is implemented as a separate process and can easily be distributed to separate processors
- composition of filters can take place "horizontally" and "vertically". i.e. high level workflow stages compose horizontally while 'enhanced filters' can be constructed by vertically passing and receiving messages from subordinate filters
- It is well suited to problems that can be decomposed into a series of sequential steps

Analysis of the HTTP protocol and the requirements on the part of a server in the servicing of requests suggests that the work of a server can be segmented

into a series of steps, transforming a request object stage by stage into a response object suitable for delivery back to the client. This inherent chain of responsibility, as described by Gamma et al [5], enables the loose coupling of the parts of the server and their dynamic configuration at run time.

4.2 The Building Blocks

This section looks at the main components needed in the servicing of HTTP web requests. The components have been divided into the essential ones needed to build a minimalist server and those that, in concert with the essential components, add richer functionality

4.2.1 Essential Components

1. ***Dispatcher***

   This component interfaces with the host operating system's sockets on a specific port and continually accepts client connections. These sockets are encapsulated into internal system objects named *Connections* which are passed down a channel to the next stage in the servicing pipeline. A graphical representation of this component is shown in figure 4.1

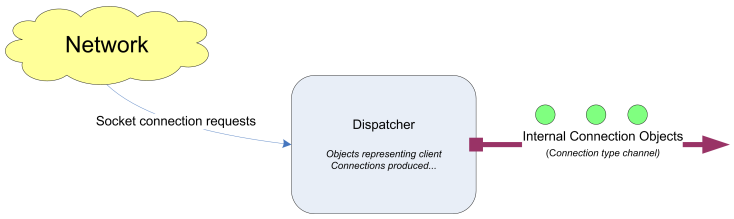

Figure 4.1: Conceptual View of a Dispatcher Component

2. ***Transformer***

   This component maps *Connection* $\mapsto$ *Request*. It accepts *Connection* objects on its input port and produces "packetised" Request objects on its output port. *(For details of message types, see section 4.3 on page 48)*. The Transformer reads the data stream from the client via the *Connection* object and builds up successive object representations of HTTP requests. The HTTP/1.1 specification allows for a client to pipeline multiple requests through a single connection[1], therefore each

[1]For information about pipelining, see section 3.1.1 on page 23

Connection object input to the Transformer component can result in one or more *Request* objects being generated on its output port. The maximum number of requests that a transformer will accept through a single connection is a parameter of a specific implementation. A Transformer must also protect itself from idle connections and 'bail out' if no data is detected coming through the socket. A graphical representation of this component is shown in figure 4.2

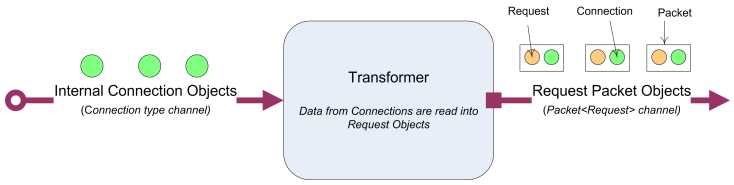

Figure 4.2: Conceptual View of a Transformer Component

3. ***Responder***

   This kind of component is responsible for mapping *Request* $\mapsto$ *Response*. By examining the METHOD, URI and other *Request* parameters, the Responder generates a *Response* object, which is packetised and sent on for further processing through the pipeline. Ideally, for the sake of modularity, a Responder is normally designed to handle a very specific type of *Request*, such as a request to serve a file or execute a query.

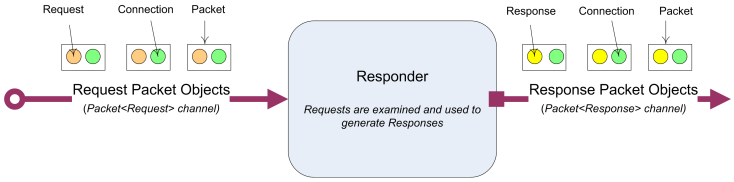

Figure 4.3: Conceptual View of a Responder Component

4. ***Deliverer***

   The last essential kind of component in a server assembly is a Deliverer. A Deliverer takes *Response* packets of any kind and streams the responses back to the client using the connection object contained within the packet. This component is the last to process any data resulting from a request-response interaction with a client. A graphical representation of this component is shown in figure 4.4

In summary, a complete version of the most basic type of server assembly can be visualised as shown in figure 4.5

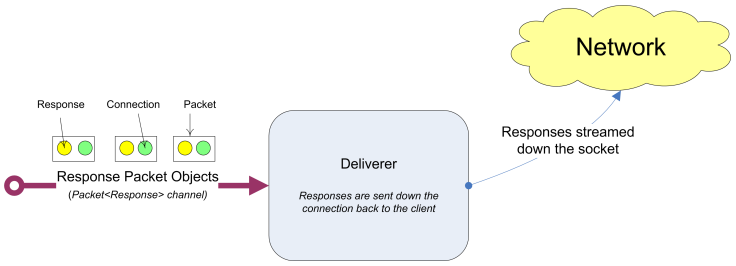

Figure 4.4: Conceptual View of a Deliverer Component

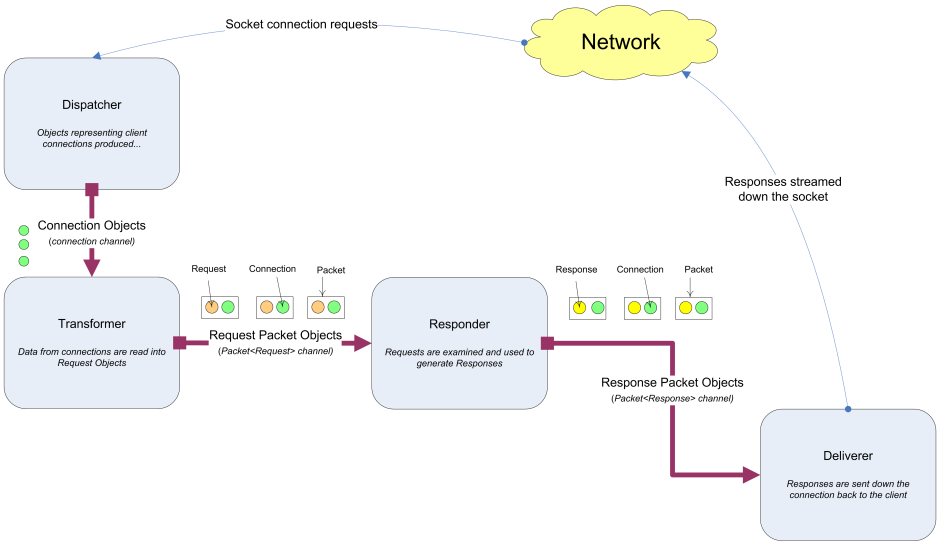

Figure 4.5: A Minimalist Server Assembly

4.2.2 Auxiliary Components

The following component types can be included in a server assembly to add security or enhance the performance and functionality of a Responder subsystem.

1. ***Authoriser***

   An Authoriser is a component that maps *Request* $\mapsto$ *Request*. Internally, it maintains a conceptual mapping of *Request* patterns (made from Method, Authorisation Headers, etc) to boolean values representing whether or not a *Request* has authorisation to be serviced. If authorisation is not permitted, the *Request* is transformed into a *Challenge Request*[2] that causes the destination Responder to issue to the client a *Challenge Response* instead of the originally requested *Response*. If authorisation is granted, the *Request* is left unaltered.

2. ***Cache***

   A Cache is a generic component that can be used to intercept the input and output channels of a Responder assembly. In so doing, it can store *Responses* as they leave the Responder and Intercept *Requests* as they are channelled to the Responder. If an intercepted *Request* can be serviced from the cache's store, the cache will issue the *Response*, bypassing the underlying Responder. The effect of this composition, as viewed from an external entity, is an enhanced Responder component. The original Responder becomes subordinate to the Cache. This Cache works on the same principle as caching proxies on the Internet. It will examine the headers of *Responses* from the Responder and determine whether or not a *Response* qualifies for caching so that no programmatic changes need to be made to any other component in order to utilise a cache. See figure 4.6 for an illustration

3. ***Connection-Aggregator***

   In server implementations where concurrency needs are greater due to increased load, a Connection-Aggregator component can be used to ensure that all *Requests* originating from a single connection follow one another through the same path within the system. As messages are channelled to a pool of identical components for servicing, the first message has its shared *Connection* object 'fingerprinted' with information stating which channel it was routed down. Any subsequent message containing a *Connection* with a 'fingerprint' from this

[2] A sub-type of Request

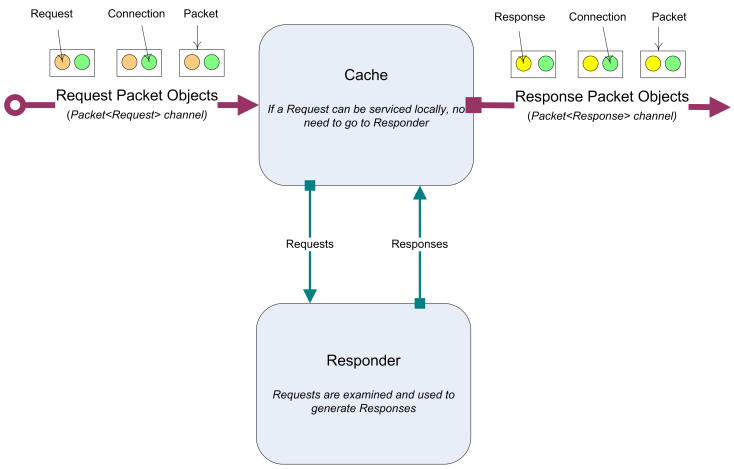

Figure 4.6: A Generic Cache

component is automatically routed down the same path. Typically a Connection-Aggregator would be placed immediately before a pool of identical Responder stacks or a pool of identical Deliverer components. Connection-Aggregators are necessitated by the requirement in the HTTP/1.1 specification[4] that multiple requests pipelined through a single connection must return to the client in the same order. See figure 4.7

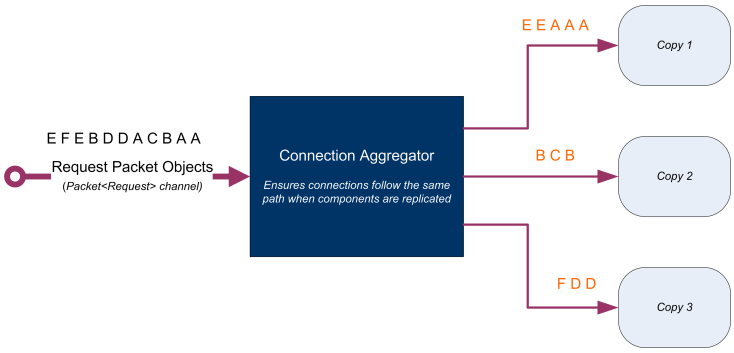

Figure 4.7: An Illustration of A Connection Aggregator (Left hand side is the input, right hand side is the output)

4. ***Router***

   A Router is a component that is used to de-multiplex messages that collectively fall under a particular type, separating them into implementer defined dynamic sub-types. It contains one input channel and

multiple output channels. Each output channel corresponds to one of these sub-types. For example, an implementer may decide to combine a stand-alone File Server Responder with a stand-alone Application Server Responder to create a dual purpose server. URI patterns can be specified to distinguish between *Requests* for the two and used in the Router's routing algorithm to ensure that each Responder receives *Requests* that it is capable of servicing. See figure 4.8

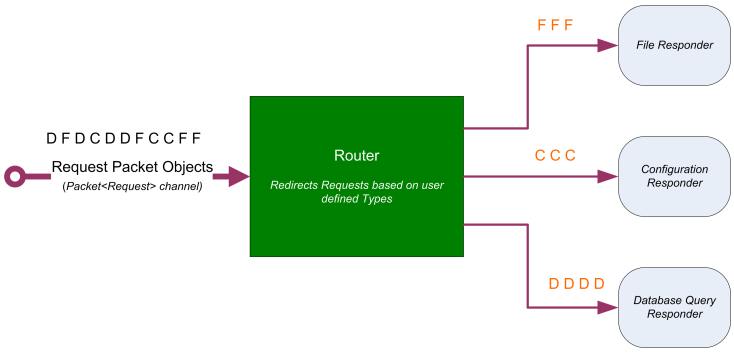

Figure 4.8: An Illustration of a Router (Left hand side is the input, right hand side is the output).

5. ***Session-Decorator***

A Session-Decorator is a component that can further enhance a Responder assembly. For certain types of applications, data must be stored across multiple client-server request-response interactions. When the need for this arises, the Responder will create a Session object to contain all the state information. This Session object is attached to all outgoing *Responses* and all incoming *Requests*. The Responder expects to be able to continue working on the same Session when subsequent *Requests* come back from the same client. This functionality is achieved by having the *Responses* encoded with a Session identifier (using cookies or URL rewriting), and having the Session objects stored in a <sessionID, Session> mapping on the server. This same identifier is returned by the client when a new *Request* is issued, allowing the Session decorator to retrieve the corresponding Session object and attach it to the incoming Request before is reaches the Responder.

4.3 Internal Message/Object Types

4.3.1 Static System Types

1. ***Request***

   A *Request* is an object that embodies the data sent upstream from the client. It provides convenient methods used for retrieving and setting its properties by components that will need to perform operations on it. In a normal request-response interaction, the final destination of a *Request* is the Responder component. There are two special sub-types of Request, namely, *Error-Request* and *Challenge-Request*. A normal *Request* can be transformed into one of these two if an error is generated on the *Request's* way to the Responder, or if authorisation fails. All Responders descend from a type that can automatically handle these special Requests when confronted with them.

2. ***Response***

   A *Response* is an object that encapsulates the data that will be streamed back to the client. The final destination for a Response is the Deliverer. Here we also have *Challenge-Response* and *Error-Response* sub-types.

3. ***Packet***

   A *Packet* is simply a convenience object used to allow a *Connection* object to piggy back with a *Request* and *Response* until it is used at the Delivery stage.

4. ***Connection***

   A *Connection* is an encapsulation of an Operating System socket. There is a 1:N relationship between *Connections* and *Requests*. A *Connection* must travel with whatever object it has directly or indirectly given rise to, so long as there is a possibility that communication will need to be made with the originating client application[3].

4.3.2 Dynamic User Types

The component based architecture proposed here promotes the modularisation of tasks wherever possible. It is therefore possible to build specialised subcategories of the static types (*Requests*, *Responses*, etc) that are routed

[3]The client would normally require some form of feedback from the server

to and processed by specialised components. This sub-typing is based on patterns within the messages that reveal further information about their type. For example, all URIs starting with /docs/ could refer to files and thus be routed to an appropriate File Responder. As this typing is subject to the needs of the implementer and is only enforced at run time, it is referred to as *dynamic*. The implementer is required to create a tree structured collection of custom types[4] that can be used to de-multiplex messages using an auxiliary *Router* component (See page 46).

4.4 The Connectors

The connectors used throughout the system are either non-buffered, synchronised, blocking channels[5] or buffered channels.

The buffered channels are used mainly as input channels to components where there is the possibility for latency of unpredictable length in the processing of any message.

The non-buffered channels can be used to switch a single writer among many readers or many writers to a single reader. This switching is accomplished using synchronisation on the channel[6]. There are cases when this kind of switching is desirable, such as when a single Dispatcher needs to indiscriminately dish out *Connections* to Transformers or when all *Responses* need to be brought to one point for aggregation by *Connection* and distributed to a pool of Deliverers in correct sequence.

4.5 Assembly Rules and Patterns

The server architecture presented thus far has a very simple to understand flow of processing and is designed for ease of use and safe assembly. However, due to the dynamic nature of part of the typing regime and also due to the generic nature of the components, some care needs to be taken to ensure that a configuration is semantically sound.

[4]The sub-types should form a partition of the parent type, i.e. the intersections of their instance spaces should be empty but the union of all the instance spaces should equal the instance space of the parent type

[5]synchronised on their **read** and **write** methods and allow data to be transferred only when both reader and writer are ready - OCCAM style “shared” channels

[6]The underlying Operating System will ensure that when many components contend for access to one end of the channel, each will take its turn

4.5.1 The Straight Pipeline Assembly

The simplest assembly is the one illustrated in figure 4.5. This can be classified as a 'Straight Pipeline Approach'. Here we have a single Dispatcher, Transformer, Responder and Deliverer. Every message follows a single predictable path through the system. In this setup, no Dynamic typing on the part of the implementer was used (evidenced by the absence of an auxiliary Router). As we will see in later sections, configurations are possible using routers that allow for multiple sub-systems to be combined into a single system.

4.5.2 Decorated Responders

This architecture allows for the creation of 'feature stacks' at the point of Response Generation. The Cache and the Session-Decorator components were introduced in section 4.2.2. These components can be stacked above a Responder (by having the lower component's input and output channels connected to the upper component), effectively creating a 'super' Responder which can provide more functionality than its primitive counterpart[7]. See figure 4.9 for an illustration.

4.5.3 Responder Partitioning

If an implementer of the server wishes to design the Response sub-system as a collection of specialised Responders, this can easily be achieved by including a Router component that will filter all incoming *Requests* and channel them to the appropriate Responder sub-assembly. The algorithm used to distinguish one *Request* from another must be supplied by the implementer of the system. Figure 4.8 provides an illustration of the principle used in this design pattern.

4.6 Scaling to Higher Degrees of Concurrency

The minimal configuration of this server is designed to require minimal resources from its host environment. However, it lacks the ability to support a large number of concurrent users. Using the same building blocks, it is possible to introduce degrees of multiplicity at each stage of the process in order to improve the overall capacity and throughput of the system.

[7]It should be noted that any assembly satisfying the requirement of turning a Request into a Response can be classified as a *Responder*. This being the case, the entire system between Transformer and Deliverer can be thought of as a Responder and encapsulated. However, in most cases the stacking of features would occur at a more fine grained level

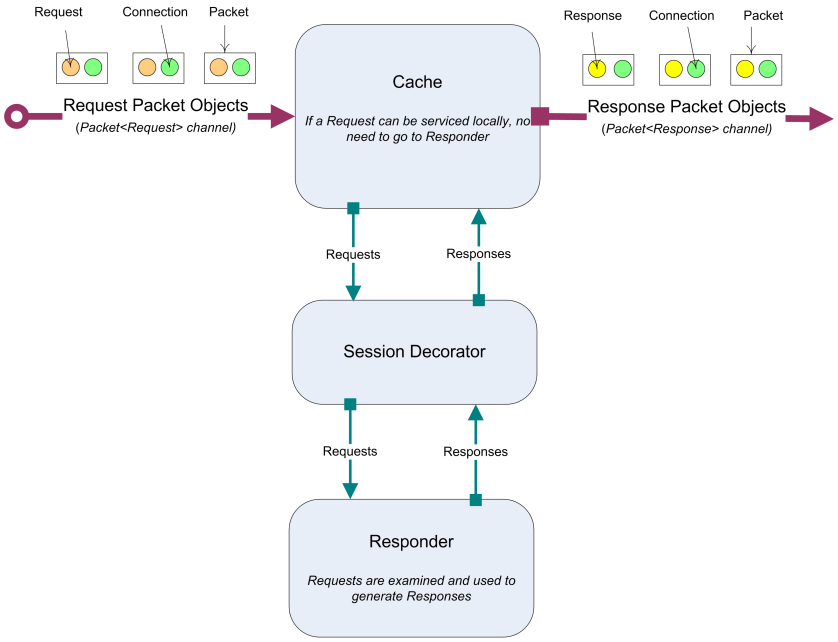

Figure 4.9: A Typical Feature Stack

4.6.1 Introducing Concurrent Transformers

At the Transformation stage, data is read off the channel and used to build up Request objects. While this goes on, new connections are not being read through the channel and at most one additional socket connection from a client could have been accepted by the Dispatcher[8]. The result is that during this time the system is unable to accommodate new clients. A solution to this problem is to introduce replicas of the system's Transformer, each feeding off the same channel[9]. The Dispatcher will 'switch' to any Transformer that is willing to accept a connection. This, in effect, serves to compensate for the potential differences in output rates between the Transformer and Dispatcher during a time of increased activity.

4.6.2 Introducing Concurrent Responders

In order to increase the ability of the server to service more client requests concurrently, multiple copies of an individual type of Responder can be made available. However, due to the strict ordering requirements for responses to

[8]The Dispatcher is either in a state of waiting for new client Requests or blocked while waiting for the Transformer to collect a new Connection

[9]Synchronisation is built into the channel to ensure atomic delivery of messages

pipelined requests, a Connection-Aggregator needs to be placed before each pool of Responder clones (As in figure 4.7) to ensure that messages grouped by connection follow the same path through that part of the system and maintain their order.

4.6.3 Introducing Multiple Dispatchers

The number of Dispatchers that the system has determines the number of clients the server can send data back to at any given time. Having too few of these can result in a bottleneck, particularly if large files are being transferred over slow connections. To counter this, clone Dispatchers can be deployed in the same way clone Responders are deployed. A Connection-Aggregator is placed in the pipeline before the pool of Dispatchers to ensure that no two Dispatchers share a Connection object and that they Receive the associated Responses in the correct sequence.

Figure 4.10 on page 53 illustrates an example of a more complex server assembly.

4.7 Summary

In this chapter we described in greater detail the characteristics of the server components that we implemented to construct our toolkit. Chapter five will expound on the Java implementation of the architecture described in this chapter.

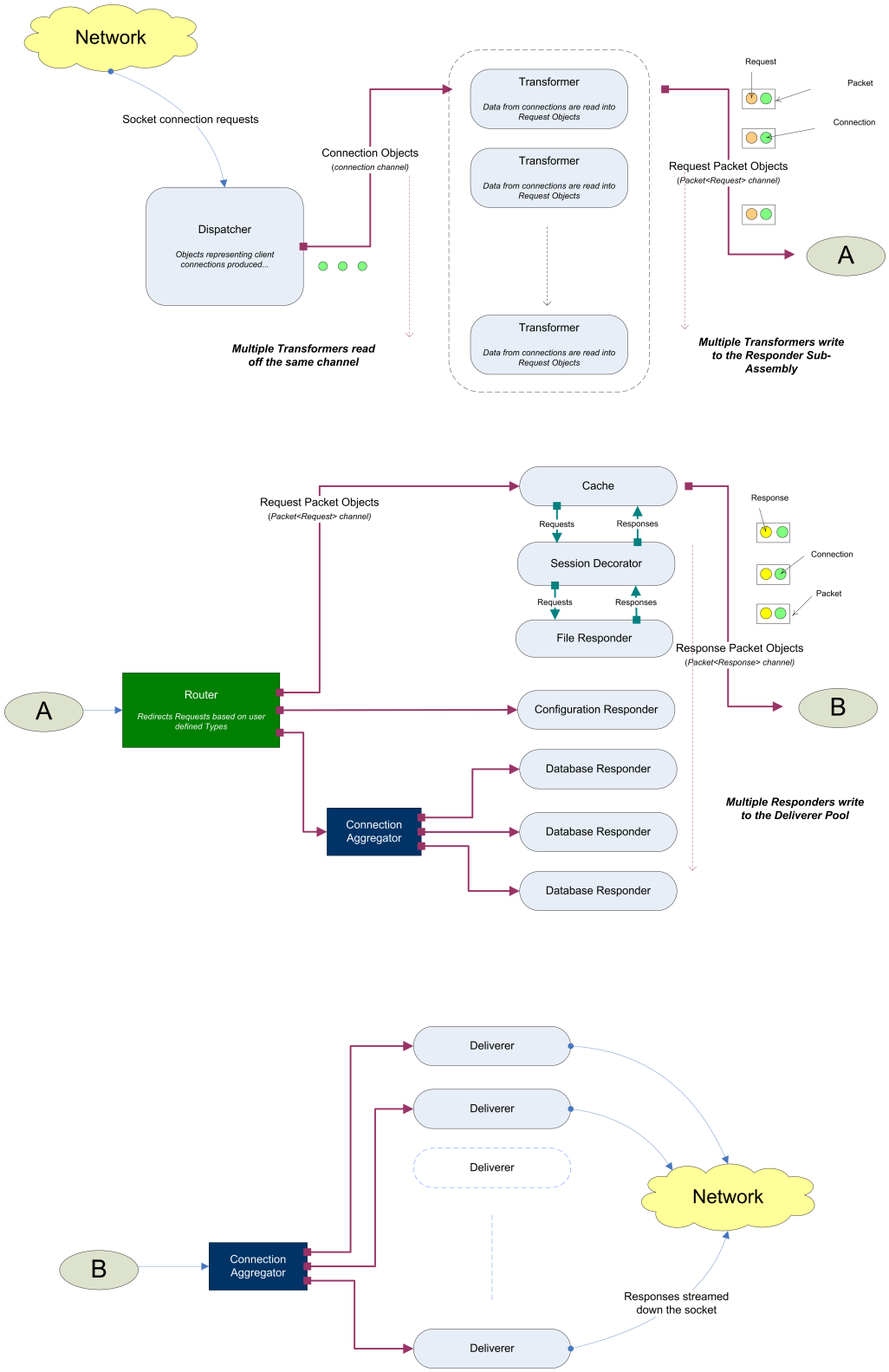

Figure 4.10: A More Complex Assembly

Chapter 5

Implementation

This chapter presents an actual implementation of our component architecture. The programming language used was the Java 2 Platform Standard Edition (J2SE) version 5.0. This platform provided the object-oriented features we required, and in particular provided support for generic types, which are used significantly throughout the source code.

We begin with an overview of the package organisation, after which we provide an explanation of a few key classes in the framework.

5.1 Java Package Overview

The implementation is divided into the following Java packages:

net.zuze.msc.csp

This package contains classes and interfaces that describe the very basic component and communication constructs of the system. The contents of this package are listed below:

- **BufferedChan**<Type>
  A channel with an internal buffer
- **Chan**<Type>
  A synchronised channel
- **Channel**<Type>
  An interface for all kinds of channels

- **Port**
  An interface that describes the behaviour of writing data to and reading data off the channel
- **Port.In**<Type>
  Interface for reading data off a channel
- **Port.Out**<Type>
  Interface for writing data to a channel
- **Port.ReadListener**
  Interface for listeners that get notified when a piece of data is read off the channel
- **Proc**
  An implementation of a Process
- **Process**
  An interface for objects that will provide services
- **Process.Epilogue**
  An interface containing a method that gets invoked when the process terminates

net.zuze.msc.artefacts

This package contains classes which represent the objects that are passed through the channels and processed within the server. The contents of the package are listed below:

- **AbstractMessage**
  An Abstract Message is an ancestor of both Requests and Responses. It provides implementations for their common functionality
- **CachedRawResponse**
  A CachedRawResponse is a 'copy' of the original response that would have been sent back to the client. It is stripped of current context information (such as whether the connection should remain persistent or not)

- **CachedWrappedResponse**
  This class encapsulates a raw cached response and adds the relevant context information about the current connection
- **ChallengeRequest**
  A Challenge Request is the result of a Request that has been through an authoriser but which did not contain the required authentication information
- **ChallengeResponse**
  A Challenge Response is one that contains in its headers the information that will instruct the browser to obtain authentication credentials from the user
- **Connection**
  A Connection encapsulates the InputStream, OutputStream and InetAddress for an accepted socket connection and is passed along in each packet that contains a message that it gave rise to
- **ErrorRequest**
  The result of an error occurring caused by a Request during its processing chain to the Responder
- **ErrorResponse**
  A standard response object created from an ErrorRequest
- **FileResponse**
  A standard file response object
- **Message**
  An interface that any message (Request or Response) should conform to
- **Packet**<TYPE extends Message>
  A general purpose container for messages throughout the system. A packet encapsulates a Message of type 'TYPE' and its associated connection
- **Request**
  An abstract class that all Requests descend from
- **Response**
  An abstract class that all Responses descend from

- **RouteAlgorithm**
  A class that encapsulates the functionality that a Router component needs in order to identify the type of a Request
- **SimpleRequest**
  A general purpose Request object that allows for the retrieval of uploaded form data and files
- **StatusResponse**
  A standard response object created to send back a status condition, for example, when a file gets moved permanently
- **TimeResponse**
  A Response that embodies a message that displays the current time
- **TraceResponse**
  A Trace Response embodies in its entity body the Request that asked for it

net.zuze.msc.artefacts.util

This package contains classes that describe special objects found within the Artefact classes. The contents of the package are described below:

- **Entity**
  This interface represents some data that comes in as the content body of a request
- **Entity.File**
  This interface represents an individual file that has been uploaded as part of an entity body
- **Session**
  An interface that describes a session object, which stores information that must persist across multiple Request-Response interactions
- **SessionMap**
  A modified HashMap used to store Session objects in memory and

periodically cleanup old sessions (that have been idle for a long time)

- **SimpleSession**
  An implementation of a Session
- **SmartHashMap**<KeyType,ValueType>
  A modified HashMap that allows the garbage collector to reclaim the least used of its entries. This is used for caching responses

net.zuze.msc.util

This package contains classes that store standard server information and provide methods that are used throughout the system. The contents of the package are listed below:

- **LineReader**
  This class has methods that are used for converting the streamed input from the client into strings corresponding to each line of the request
- **ServerSettings**
  This class groups together server configuration settings for easy access by the system components
- **ServerUtils**
  This class contains useful utilities for any general purpose server

net.zuze.msc.protocols

This package contains classes that store information about the particular protocols being used (such as their status codes). The contents of the package are listed below:

- **HTTP**
  Stores HTTP specific information

net.zuze.msc.Exceptions

This package contains classes that encapsulate exceptions that can occur during the processing of requests. The contents of the package are listed below:

- **AssemblyException**
  Thrown when an illegal assembly condition is detected
- **BadRequestException**
  This type of exception is thrown if errors are detected during the construction of a request
- **ForbiddenException**
  This exception is usually thrown from one of the responders when a particular forbidden resource is requested, e.g. directory listing
- **InternalServerException**
  This exception is thrown when some unexplainable error occurs within the server
- **NoServiceException**
  This exception is thrown from the router or a responder when the service requested by the request is unknown or unavailable
- **RedirectException**
  A Redirect Exception is thrown from a responder when a resource has been moved permanently
- **TimeoutException**
  This exception is thrown from the request message constructor when incoming data is not detected after a certain period of time

net.zuze.msc.skeletons

This package contains interfaces for all the component types that can exist in a server. The contents of the package are listed below:

- **Authoriser**<Type>
  An interface for an authoriser, which takes in request packets, in-

spects them and carries out authentication and authorisation. If authentication information is missing, a ChallengeRequest is generated and sent forward in its place

- **Cache**<InType, OutType>
  An interface for a generic Cache, which attempts to service requests from its memory. If a Cache cannot service a request from its memory, the request is passed on to the subordinate Responder. A Cache is an extension of a Proxy.

- **Component**
  An interface for the main processing units of the system.

- **Consumer**<InType>
  An interface for a component that takes in messages on its input port

- **Deliverer**<Type>
  An interface for a Deliverer, which takes a response packet and sees to it that the client gets it by streaming the response data through its associated connection

- **Dispatcher**<Type>
  An interface for a Dispatcher, which accepts requests on a given port, makes connection objects out of them and passes them on to a transformer

- **One2NComponent**<Type>
  This is a component that has one input port and more than one output port. Each of the output ports must be addressable.

- **One2OneComponent**<InType, OutType>
  This is a component that has one input port and one output port.

- **Producer**<OutType>
  An interface for a component that outputs messages on its output port

- **Proxy**<InType, OutType>
  A Proxy is a component that acts as a bypass to another component. It contains additional input and output ports that connect to the component that it is bypassing.

- **Responder**<InType, OutType>
  An interface for a Responder, which generates a response out of a request
- **Router**<Type>
  An interface for a Router, which takes in a request packet and routes it to the appropriate responder. A Router is a One2NComponent.
- **RoutingAlgorithm**
  An interface for an object that encapsulates a method used to determine the type of a request. (This is passed as a parameter to a Router)
- **Server**
  An interface for a server assembly. Provides methods for adding components and connectors, making attachments, and starting up the server.
- **Transformer**<InType, OutType>
  An interface for a Transformer, which takes a Connection and generates one or more requests out of it

net.zuze.msc.simpleserver

This package contains classes that implement the components for an actual server instance. The contents of the package are listed below:

- **AbstractResponder**
  An Abstract class that all Responders must descend from. Implements code for handling ErrorRequests and ChallengeRequests.
- **ConnectionAggregator**<T extends Packet<Message>>
  A ConnectionAggregator is a component that ensures that all Messages that share a common connection travel down the same path when there are multiple identical Responders or Deliverers to choose from
- **FileAuthoriser**
  The File Authoriser takes a request packet, tests if the necessary

authentication has been included and either sends it on to the server for servicing or dumps it and sends forward a Challenge Request based on it if authorisation fails.

- **FileServiceResponder**
  This class takes in request packets, retrieves the file from the O/S's file system, and constructs a response object which will get passed on to the deliverer

- **SessionDecorator**
  A Session Decorator stores away session objects from outgoing Responses and marks the Responses with session IDs (as a cookie). It intercepts incoming Requests and retrieves their session objects (if any exist)

- **SimpleCache**
  The SimpleCache intercepts requests and sees if it can service them from its store in memory.

- **SimpleDeliverer**
  This class takes in Response Packets and sends them off to the client

- **SimpleDispatcher**
  This class accepts connections on the given port, creates connection objects out of them and sends them down a channel to a transformer

- **SimpleRouter**
  This class takes in a packet, determines its dynamic type and routes it off to the correct service provider

- **SimpleSSLDispatcher**
  This class accepts SSL connections on the given port, creates connection objects out of them and sends them down a channel to a transformer

- **SimpleTransformer**
  This component is Responsible for generating one or more Request objects from a given connection.

- **SubSystemCap**
  This class is a Proxy that is used to provide a gateway to a subsystem of components.

- **TestServer**
  This class implements an actual server. It contains methods that are needed to add components, add connectors, make attachments and startup the server.
- **TimeResponder**
  Serves the time back to the client
- **TraceResponder**
  This class just sends back as the response body, what came in as the request

5.2 Class Descriptions

The entire Application Programming Interface for the Java implementation can be found in the appendix. This section takes a closer look at a few classes that are of particular interest.

5.2.1 ConnectionAggregator

In chapter 3 we explained the use of persistent connections and the requirement that the responses to pipelined connections be returned in the same order as they were requested. Using our assembly architecture, multiple copies of a component can be used to accommodate the concurrent servicing of requests. The result of this is that for a particular type of request, there is more than one possible path through the system. A set of requests that originated from the same connection could take different paths through the system and be returned to the client out of sequence. To counter this occurrence, a ConnectionAggregator is placed in the pipeline before a pool of cloned components to ensure that this dispersion does not take place. Below is the central piece of code in the ConnectionAggregator

```
public void run() {

  while(true)
  {
    try{
       T job = in.read();
```

```
        Port.Out<T> p = job.getConnection().getPath(this);

        if(p == null)
        {
              p = allocate();
              if(p == null) throw new InternalServerException
                  ("Busy");
              job.getConnection().setPath(this, p);
        }

        p.write(job);

         }
      catch(InternalServerException e){
          System.err.println("Server Too Busy...");
      }
    }

  }
```

5.2.2 Connection

This class encapsulates the Socket that is used to transfer data to and from the client. The Connection object includes two special methods to assist in the correct flow of messages in the pipeline. These allow a ConnectionAggregator to store the path allocation decision that it makes the first time it 'sees' a connection, and to recall this decision for subsequent message packets containing the same connection. A snippet from the source code of the connection class is shown below.

```
.
. // This stores information about where the connection has
    been
 java.util.Hashtable<Proc,Port.Out> fingerPrints;
.
.
.
  /**
   * Retrieves the output port that the connection should
   * go down if it has already been through
   * ConnectionAggregator 'p'
   *
   * @param p The process (connection aggregator) asking
   * @return The pointer to the output port to push the
       packet down
   */
```

```
public Port.Out getPath(Proc p){
      return fingerPrints.get(p);
}

/**
 * This is invoked by a Connection Aggregator the
 * first time it sees a connection and has decided
 * which path to allocate it to
 *
 * @param p The Connection Aggregator
 * @param po The Port that it has been assigned to
 */
public void setPath(Proc p, Port.Out po){
    fingerPrints.put(p,po);
}
```

5.2.3 SessionMap

A SessionDecorator requires a data structure that can map session ids to Session objects. These Session objects are stored from outgoing responses and retrieved from the store for incoming requests. Since these Session objects are stored in memory, we need a way to control the time a Session spends in the map to avoid running out of memory.

This was achieved by encapsulating a HashMap class into a SessionMap class. The SessionMap includes a TimerTask that runs periodically to remove Session entries that have been idle for too long. Below is the java listing.

```
public class SessionMap {

    /**
     * The underlying HashMap
     */
    HashMap<String, Session> theSessions = new HashMap<String
        , Session>();
    /**
     * The interval between cleanup sweeps
     */
    long clean_time;

    /**
     * Creates a new instance of SessionMap
     * @param cleanuptime The interval time between cleanup
         runs
     */
    public SessionMap(int cleanuptime) {
```

```
        clean_time = cleanuptime * 1000;
        Timer t = new Timer();
        CleanupTask ct = new CleanupTask();
        t.schedule(ct,clean_time,clean_time);
    }

    /**
     * puts a Session in memory
     * @param key The Session ID
     * @param value The actual session
     */
    public void put(String key, Session value){
        theSessions.put(key,value);
    }

    /**
     * Retrieves a session from memory
     * @param key the Session ID
     * @return the session referred to by the key
     */
    public Session get(String key){
        return theSessions.get(key);
    }

    /**
     * Removes a Session from memory
     * @param key the ID of the Session to remove
     */
    public void remove(String key){
        theSessions.remove(key);
    }

    /**
     * This class executes a process of sweeping through the
     * session map removing expired sessions from it
     */
    class CleanupTask extends TimerTask {

        /**
         * The process of sweeping through and removing
             expired sessions
         */
        public void run()
        {
            Session sess;
            // clean-up the session mapping, dumping expired
                sessions
            for(Map.Entry e: theSessions.entrySet())
            {
```

```
                sess = (Session) e.getValue();
                if(sess.hasExpired()){
                    theSessions.remove(e.getKey());
                }
            }
        }

    }
}
```

5.2.4 SmartHashMap

Our caching component needs a way to cache items in memory without utilising space that the system may need for more important purposes. We have achieved this by defining our SmartHashMap which has a guaranteed capacity but which can store more items if memory is available. This is achieved through the use of what are called *Hard References* and *Soft References*. When an item is added to our SmartHashMap, it is simultaneously stored with a Hard Reference in a Least Recently Used (LRU) HashMap and with a Soft Reference in another HashMap.

The LRU HashMap will keep the most recently used items in memory when entries exceed its capacity and dump the Least Recently Used ones. Items are never explicitly removed from the two HashMaps. If the garbage collector needs memory, it has the option of reclaiming memory from the objects that have only Soft References to them. The code listing for the SmartHashMap is shown below.

```
package net.zuze.msc.artefacts.util;
import com.opensymphony.oscache.base.algorithm.LRUCache;
import org.apache.commons.collections.map.ReferenceMap;

/**
 * A SmartHashMap is used by a Cache to store Responses.
 * It has a guaranteed minimum capacity which can expand.
 * The most recently used items are guaranteed
 * to stay in the Cache
 *
 * @author Daliso
 */
public class SmartHashMap<KeyType, ValueType> {

    /**
     * The soft references that hold no guarantees,
     *  the gc could get them at any time
```

```
     */
    ReferenceMap softs = new ReferenceMap(ReferenceMap.SOFT,
       ReferenceMap.SOFT);
    /**
     * These are hard references to the cached objects. The
        size of this
     * is the guaranteed capacity of the SmartHashMaps
     */
    LRUCache      hards;

    /**
     * Creates a new instance of SmartHashMap
     * @param size The guaranteed capacity
     */
    public SmartHashMap(int size) {
        hards = new LRUCache(size);
    }

    /**
     * Puts a new item in the Cache
     * @param key The key id
     * @param value the object being cached
     */
    public void put(KeyType key, ValueType value){
        softs.put(key, value);
        hards.put(key, value);
    }

    /**
     * Retrieve an item from the Cache
     * @param key the key of the item to retrieve
     * @return The Cached Item
     */
    public ValueType get(KeyType key){
        ValueType val = (ValueType) hards.get(key);
        if( val == null)
        {
            val = (ValueType) softs.get(key);
            if (val != null) hards.put(key,val);
        }
        return val;
    }

}
```

5.2.5 SimpleSSLDispatcher

The SimpleSSLDispatcher is a Dispatcher that accepts SSL connections instead of normal TCP connections. In order to use SSL Sockets, one must have a file configured that contains the private and public keys that will be used for encrypting and decrypting communication with client applications. The file that stores this information is called a *KeyStore*. Once this file is configured, it can be loaded into the application prior to instantiating the SSLServerSocket. At this point the SSLServerSocket works just like a normal socket, all SSL specific handshaking and encryption is done in the background. The source code for the SimpleSSLDispatcher is shown below.

```
public class SimpleSSLDispatcher extends Proc implements net.
    zuze.msc.skeletons.Dispatcher<Connection> {

    /**
     * The output Port to Transformers
     */
    Port.Out<Connection> out;
    /**
     * The socket for accepting connections
     */
    ServerSocket serverSocket;

    /**
     * Creates a new instance of SSLDispatcher
     * @param out This is the front end of the channel that
     * new connections are pushed down.
     * @param port This is the port number that the server
         will listen on
     */
    public SimpleSSLDispatcher(Port.Out<Connection> out,int
        port) {
      this.out = out;

      try{

          System.setProperty("javax.net.ssl.keyStore", ".
              keystore");
          System.setProperty("javax.net.ssl.keyStorePassword"
              , "changeit");
          serverSocket = (SSLServerSocket)
              SSLServerSocketFactory.getDefault().
              createServerSocket(port);

      }
      catch(IOException e){
```

```
        System.err.println("From the ssl dispatcher
            constructor("+e+")");
    }
  }

  /**
   * Accepts SSL Connections and passes them on to a
       Transformer
   */
  public void run() {
    int i = 1;
    while(true){

      try{
        Connection c = new Connection(serverSocket.accept()
            );
        if (ServerSettings.getInstance().traceConnections()
            )
            System.err.println("Sent: "+(i++)+"  ");
        out.write(c);
      }
      catch (SSLException e){
       if (ServerSettings.getInstance().traceExceptions())
       {
            System.err.println("From the dispatch run("+e+")
                ");
            //e.printStackTrace();
       }
      }
      catch (IOException e){
       if (ServerSettings.getInstance().traceExceptions())
       {
            System.err.println("From the dispatch run("+e+")
                ");
            //e.printStackTrace();
       }
      }
    }
  }

  public void setOutputPort(net.zuze.msc.csp.Port.Out<
      Connection> out) {
       this.out = out;
  }


  public void stop() {
  }
```

```
    public void start() {
        this.fork();
    }

    public void start(int num) {
        for(int i=0;i<num;i++)
            this.fork();
    }
}
```

5.2.6 SimpleRouter

The SimpleRouter directs request packets towards the appropriate Responder. It has a flexible number of resource types that it can accommodate, which is determined by the capabilities of the RoutingAlgorithm. The source code for this component is provided below.

```
public class SimpleRouter extends Proc implements Router<
    Packet<Request>>{

    /**
     * The input Request Port
     */
    Port.In<Packet<Request>> in;
    RoutingAlgorithm alg;
    HashMap<String, Port.Out<Packet<Request>>> theOuts = new
        HashMap<String, Port.Out<Packet<Request>>>();
    Port.Out<Packet<Request>> theOne;
    String defaultService;
    /**
     * Creates a new instance of SimpleRouter
     * @param in The packet from the transformer
     * @param alg The algorithm to determine the type of each
          request
     */
    public SimpleRouter(  Port.In<Packet<Request>> in,
        RoutingAlgorithm alg) {
        this.in = in;
        this.alg = alg;
        this.name = "SimpleRouter";
        for(String s: alg.getServices()){
            theOuts.put(s, null);
        }
        defaultService = = "File"; // this can be changed in
            the future to something universal
    }
```

```
    /**
     * The routing process is executed here. Packets are
     * read in, their type determined and sent down the
     * correct path to the appropriate Responder sub-assembly
     */
    public void run() {
      while(true)
      {
        Packet<Request> job = in.read();
        try {
// get the name of the resource type from the routing
    algorithm
          String resourceType = alg.interpret(job.getMessage
              ());
// get the appropriate output port from the output port
    collection
          theOne = theOuts.get(resourceType);
          if(theOne == null) throw new NoServiceException("
              The path to the Responder for "+resourceType+"'s
               has not been set");
// write the packet
          theOne.write(job);
        }
        catch(NoServiceException e){
         ErrorRequest er = new ErrorRequest(HTTP.getInstance
            ().getCode("HTTP_NOTIMPLEMENTED"), e.getMessage()
            ,job.getMessage());
         theOne = theOuts.get(defaultService);
         if (theOne != null) theOne.write(new Packet<Request
            >(er,job.getConnection()));
         if (ServerSettings.getInstance().traceExceptions())
             System.err.println("\t!! NO SERVICE EXCEPTION
                FROM ROUTER THREAD ("+Thread.currentThread().
                getName()+")");
        }
      }
    }

      public void setInputPort(net.zuze.msc.csp.Port.In<
          Packet<Request>> in) {
          this.in = in;
      }

      public void stop() {
      }

      public void start() {
          this.fork();
```

```
    }

    public void start(int num) {
      for(int i=0;i<num;i++)
          this.fork();
    }

    public void setOutputPort(String id, net.zuze.msc.csp.
        Port.Out<Packet<Request>> o) {
         theOuts.put(id, o);
    }
}
```

5.2.7 FileAuthoriser

The FileAuthoriser checks the URL of incoming requests to determine if authorisation is required. If it is, the FileAuthoriser performs a check on the authentication information supplied in the request by the user. This involves ensuring that the password that the server has in its records for a particular user matches the one supplied in the request. The only complication lies in ensuring that the password is not revealed to a third party during transmission from the client to the server. This is done by encrypting the password. HTTP offers two ways to perform this: *Basic* and *Digest*. Basic Authentication hides the password by encoding the concatenation of `username:password` in Base64. The code for performing the Base64 check in Java is shown below.

```
String[] s  = theUsers.next(); // server's info retrieved
String pass = Base64.encode(s[0]+":"+s[1]);
if(auth.equals("Basic "+ pass)) return true;
```

The drawback to this approach is that the encoding is easily reversible and provides very mild protection of the password. A better approach to protecting the password is the use of Digest Authentication. This uses a digest function that is not easily reversible. The protection of the password is increased further by attaching extra information to the username and password before the digest is performed. The code snippet below shows the steps involved in performing a Digest Authentication check.

```
MessageDigest md5 = MessageDigest.getInstance("MD5");

// authParameters is the header info from the request
```

```
String s1 = authParameters.get("digest username")+":"+
    authParameters.get("realm")+":"+pass;
byte[] b1 = md5.digest(s1.getBytes());

// r is the request
String s2 = r.getMethod()+":"+r.getURI();
byte[] b2 = md5.digest(s2.getBytes());


String s4 = byteArrayToHexString(b1)+":"+authParameters.get(
    "nonce")+":"+authParameters.get("nc")+":"+authParameters.
    get("cnonce")+":"+authParameters.get("qop")+":"+
    byteArrayToHexString(b2);
byte[] b4 = md5.digest(s4.getBytes());

String expected = byteArrayToHexString(b4);

if(expected.equals(authParameters.get("response")))
....
```

5.3 Summary

This chapter provided insight into the Java implementation of our server toolkit. Further information on the Application Programming Interface can be found in the appendix. The source code for all the classes is also available on the accompanying CD-Rom.

Chapter 6

Case Studies

This chapter illustrates the effectiveness of our component toolkit in constructing servers. We present two usage scenarios and explain the process of assembling and deploying each individual server.

6.1 Case 1: An Embedded Server

Our first case is an example of a server embedded in a host application. The following sections explain the requirements and the solution that can be built using our toolkit.

6.1.1 The Requirement

A desktop application is being developed that will keep an archive of a user's personal digital photos. This application has a rich user interface that allows users to scale photographs, create photo collections, produce custom E-Cards and convert from one file format to another. This application will mostly be used in stand-alone mode, however, the developer would like to add the ability to access some of this functionality remotely. This would be useful for sharing photos with friends and colleagues. The developer would like this functionality to be accessible from a web browser but does not want to develop scripts in PHP or ASP and host the application on a central web server.

6.1.2 Proposed Server Assembly

The server that we need to embed in this application would have the following characteristics:

- Comfortably support up to five users concurrently
- Allow communication to occur using Secure Sockets Layer
- Support the uploading and downloading of large files
- Provide password access to certain resources

The assembly that we propose would contain an SSL Dispatcher for secure access and a normal Dispatcher for non-sensitive data. It will contain a pool of ten Transformers to accommodate latency in the uploading of large files. It will also have a pool of ten Deliverers to accommodate the simultaneous delivery of large files back to different clients. The requests will fall into two main categories. The first is a request to browse and view photos, while the second is a request to convert a file to another format. These two types will be distinguishable by analysing a prefix in the URL. Two types of Responders will be deployed, one for each service type and a Router will ensure that requests are routed to them appropriately. A cache will be included in the file browsing region of the assembly and an Authoriser will be placed in the pipeline before the file format conversion responder. Ten of each type of responder will be deployed, each supported by a ConnectionAggregator to prevent the dispersion of requests originating from the same connection.

6.1.3 Customisation

The developer will be able to use the existing Dispatchers and Transformers without modification. A custom routing algorithm will need to be written to distinguish between browsing requests and file conversion requests. This algorithm will be passed as a parameter to a standard Router. Two descendants of an AbstractResponder will need to be written, one for each type of request. They will invoke the appropriate system commands for each URL requested and construct appropriate HTTP responses. A standard Cache will be used for the browsing Responder and a standard Authoriser will be deployed. Standard Deliverers will be used at the end of the pipeline.

6.1.4 Putting The Components Together

After having acquired all the components for the server, the next step is to put the pieces together and set the system in motion. This can be done by instantiating a server object and invoking assembly commands on it. A code fragment of how this is done is shown below.

```
TestServer ts = new TestServer();
try
{
    //----------- Add the Components ----------

    // Add The Dispatchers
    Producer<Connection>
     dispatcher1 = ts.addProducer(new SimpleDispatcher(null
         ,80),1);
    Producer<Connection>
     dispatcher2 = ts.addProducer(new SimpleSSLDispatcher(
         null,441),1);

    // Add a Transformer with 20 child threads
    One2OneComponent<Connection,Packet<Request>>
     transfomerSet = ts.addOne2OneComp(new SimpleTransformer(
         null,null),20);

    // Add a Router with a custom routing algorithm
    One2NComponent<Packet<Request>>
         theRouter = ts.addOne2NComp(new SimpleRouter(null,
             new CustomRouteAlgorithm()));

    // Add an Authoriser
    One2OneComponent<Packet<Request>, Packet<Request>>
     theAuth = ts.addOne2OneComp(new ConvertAuthoriser(null,
         null),1);

    // Add the Responders
    One2OneComponent<Packet<Request>,Packet<Response>>
         PhotoResponderPool = ts.addOne2OnePool(new
             PhotoResponder(null,null),10);
    One2OneComponent<Packet<Request>,Packet<Response>>
         ConvertResponderPool = ts.addOne2OnePool(new
             ConvertFormatResponder(null,null),10);

    // Add the Deliverers
    Consumer<Packet<Response>>
         DelivererPool = ts.addConsumerPool(new
             SimpleDeliverer(null),10);

    // Add a standard Cache
    Proxy<Packet<Request>,Packet<Response>>
        theCache       = ts.addProxy(new SimpleCache(null,
            null,null,null));

    //----------- Add the Connectors -----------
```

```
// Add the shared channel from the Dispatchers to the
    Transformers
Channel<Connection>
 dsp2trn = ts.addConnector(new BufferedChan<Connection>("
    toTrans",50));

// Add the shared channel from the Transformers to the
    Router
Channel<Packet<Request>>
 trn2rtr = ts.addConnector(new BufferedChan<Packet<
    Request>>("toRouter",50));

// Add the channel that will take Requests from the
    Router to the File Browser Cache
Channel<Packet<Request>>
 rtr2csh = ts.addConnector(new BufferedChan<Packet<
    Request>>("toCache",50));

// Add the channel that will take Requests from the
    Router to the Authoriser
Channel<Packet<Request>>
 rtr2cau = ts.addConnector(new BufferedChan<Packet<
    Request>>("toConvertAuthoriser",50));

// Add the channel that will take Requests from the
    Authoriser to the Convert Responder
Channel<Packet<Request>>
 cau2cfr = ts.addConnector(new BufferedChan<Packet<
    Request>>("toConvertFormatResponder",50));

// Add the shared channel from the Responders to the
    Deliverer
Channel<Packet<Response>>
 resp2dlvr = ts.addConnector(new BufferedChan<Packet<
    Response>>("toDeliverer",50));

//----------- Make Attachments --------------

// Attach the Dispatchers to the Transformers via the
    shared channel
ts.attach(dispatcher1, transfomerSet, dsp2trn);
ts.attach(dispatcher2, transfomerSet, dsp2trn);

// Attach the Transformers to the Router
ts.attach(transfomerSet, theRouter , trn2rtr);

// Attach the PhotoBrowser Output of the Router to the
    Cache
ts.attach(theRouter, "PhotoBrowse", theCache , rtr2csh);
```

```
    // Attach the FileConversion Output of the Router to the
       ConvertAuthoriser
    ts.attach(theRouter, "FileConvert", theAuth , rtr2cau);

    // Attach the ConvertAuthoriser to the ConvertResponder
    ts.attach(theAuth, ConvertResponderPool, cau2cfr);

    // Attach the Cache to the Deliverers
    ts.attach(theCache, DelivererPool, resp2dlvr);

    // Attach the ConvertResponders to the Deliverers
    ts.attach(ConvertResponderPool, DelivererPool,resp2dlvr);

    // Attach the PhotoBrowsers as a subordinate to the Cache
    ts.attachSub(theCache, PhotoResponderPool);


    //--------- Start Up The Server --------------
    ts.start();
}
catch(AssemblyException ae){}
```

The piece of code above can be encapsulated and made part of the main application. The semantics of the request processing are clear and the developer can make adjustments if the need arises for more service types or more concurrent users.

6.2 Case 2: A Standard Server

Our second case is a situation where there is a need to develop a standard server that will be used to serve files to a large number of concurrent users.

6.2.1 The Requirement

A University department needs to devise a way of delivering content for e-learning courses. They would like to have a simple language developed that would allow course developers to markup text in a form consistent with existing procedures for developing standard courses. This language must be easily transformable into HTML for web distribution but remain flexible enough for delivery to other media in the future. The course content would potentially be accessed concurrently by hundreds of students.

6.2.2 Proposed Server Assembly

The assembly that we propose for this situation is tuned to high concurrency for a single type of resource. It will be a straightforward pipeline but with a high degree of concurrency at each stage to cope with added load. We will deploy a standard Dispatcher, a large pool of standard Transformers, Authorisers, Custom Responders and Deliverers. We shall also deploy a Cache at the Responder stage.

6.2.3 Customisation

The only component customisation required here is the Responder that will transform the course content from the proprietary language to HTML pages.

6.2.4 Putting The Components Together

Once the custom Responder component has been developed, the parts of the system can be assembled. The code below outlines the assembly steps.

```
TestServer ts = new TestServer();
try
{
    //----------- Add the Components ----------

    // Add The Dispatchers
    Producer<Connection>
      dispatcher1 = ts.addProducer(new SimpleDispatcher(null
          ,85),1);
    Producer<Connection>
      dispatcher2 = ts.addProducer(new SimpleSSLDispatcher(
          null,444),1);

    // Add a Transformer with 200 child threads
    One2OneComponent<Connection,Packet<Request>>
     transfomerSet  = ts.addOne2OneComp(new SimpleTransformer
        (null,null),200);

    // Add an Authoriser Pool
    One2OneComponent<Packet<Request>,Packet<Request>>
         AuthoriserPool = ts.addOne2OnePool(new Authoriser(
            null,null),100);

    // Add a Responder Pool
    One2OneComponent<Packet<Request>,Packet<Response>>
         ContentResponders = ts.addOne2OnePool(new
            ContentResponder(null,null),100);
```

```
// Add a standard Cache
Proxy<Packet<Request>,Packet<Response>>
    theCache = ts.addProxy(new SimpleCache(null,null,null
       ,null));

// Add the Deliverers
Consumer<Packet<Response>>
     DelivererPool = ts.addConsumerPool(new
        SimpleDeliverer(null),200);


//----------- Add the Connectors -----------

// Add the shared channel from the Dispatchers to the
    Transformers
Channel<Connection>
 dsp2trn = ts.addConnector(new BufferedChan<Connection>("
    toTrans",50));

// Add the shared channel from the Transformers to the
    Authorisers
Channel<Packet<Request>>
 trn2cna = ts.addConnector(new BufferedChan<Packet<
    Request>>("toAuth",50));

// Add the channel that will take Requests from the
    Authorisers to the Cache
Channel<Packet<Request>>
 cna2csh = ts.addConnector(new BufferedChan<Packet<
    Request>>("toContentCache",50));

// Add the shared channel from the Responders to the
    Deliverer
Channel<Packet<Response>>
 resp2dlvr = ts.addConnector(new BufferedChan<Packet<
    Response>>("toDeliverer",50));

//----------- Make Attachments --------------

// Attach the Dispatchers to the Transformers via the
    shared channel
ts.attach(dispatcher1, transfomerSet, dsp2trn);
ts.attach(dispatcher2, transfomerSet, dsp2trn);

// Attach the Transformers to the Authoriser
ts.attach(transfomerSet, AuthoriserPool , trn2cna);

// Attach the Authorisers to the Cache
ts.attach(AuthoriserPool, theCache , cna2csh);
```

```
    // Attach the Cache to the Deliverers
    ts.attach(theCache, DelivererPool , resp2dlvr);

    // Attach the ContentResponder as a subordinate to he
        Cache
    ts.attachSub(theCache, ContentResponders);


    //--------- Start Up The Server ---------------
    ts.start();
}
catch(AssemblyException ae){}
```

6.3 Summary

This chapter demonstrated the process of assembling a server out of components. The two case studies we presented gave some insight into the flexibility of the framework in terms of service expansion and concurrency.

Chapter 7

Conclusion and Future Work

This dissertation reflects our work towards the realisation of the goals laid out in Chapter 1. The objective was to develop a programming toolkit that software developers could use to assemble a range of web servers. Our emphasis in this project was on the clear semantics of the operation of the servers and the ease of use in the assembly of their components. We believe that we have come far in the manifestation of this toolkit, while realising that work can indeed be done to further refine our framework.

While it was not possible to expound on every detail of our implementation, we hope that this document highlighted the important technical achievements of the project.

We have provided an intelligible way to conceptualise web servers as a collection of components and connectors organised in a particular topology. This improves the transparency of the process flows and as a result makes errors more visible and correctable. We believe that this is a key advantage of our component framework over other modularity schemes used in mainstream servers, such as the Apache architecture described in Chapter 2.

Our toolkit opens the door to experimentation with different configurations and allows developers to add web based front ends to their applications with relatively little effort. The toolkit itself is easily extensible, in particular, extra functionality could be added to a server object to improve on its introspective capabilities. For example, a server could provide performance predictions based on its current configuration or it could make itself visible and illustrate real-time processing data for the purpose of performance analysis.

Future work that we envisage involves the rigorous analysis and testing of performance characteristics. This is particularly important for servers that will be used by a large number of concurrent users. This testing was not possible in our project due to the elaborate setup that would have been re-

quired to simulate real world load demands. We would also like to see further refinement of the existing components and development of new components to fit into the current framework.

Appendix A

Application Programming Interface

A.1 net.zuze.msc.csp

public class **BufferedChan**<Type>
extends java.lang.Object
implements Channel<Type>

Constructor Summary

BufferedChan()

This is the default constructor

BufferedChan(java.lang.String name)

This constructor takes the name of the channel

BufferedChan(java.lang.String name, int size)

This constructor takes the name and the size of the buffer

Method Summary

boolean canWrite()

Checks if there is room left in the buffer to write to

boolean enabled(Port.ReadListener listener)

Returns: true if the port has an element available; false if there's no element ready; If the listener is non-null, then it is to be inform()ed when next an element becomes available.

Type read()

Read an element from the port if (or as soon as) one becomes available.

java.lang.String toString()

returns the Channel's name

void write(Type item)

Write an element to the port as soon as a reader is ready to accept it.

public class **Chan**<Type>
extends java.lang.Object
implements Channel<Type>

Field Summary

boolean free

false if there is a reader or writer waiting

boolean readerWaiting

true if there is a reader waiting

Constructor Summary

Chan()

This is the default constructor

Chan(java.lang.String name)

This constructor takes a channel name as an argument

Method Summary

boolean canWrite()

Checks if there is a reader waiting

boolean enabled(Port.ReadListener listener)

Returns: true if the port has an element available; false if there's no element ready; If the listener is non-null, then it is to be inform()ed when next an element becomes available.

Type read()

Read an element from the port if (or as soon as) one becomes available.

java.lang.String toString()

returns the Channel's name

void write(Type item)

Write an element to the port as soon as a reader is ready to accept it.

public interface **Channel**<Type>
extends Port.In<Type>, Port.Out<Type>

All Superinterfaces:

Port.In<Type>, Port.Out<Type>

All Known Implementing Classes:

BufferedChan, Chan

no additional info

public interface **Port**

Nested Class Summary

static interface Port.In<Type>
static interface Port.Out<Type>
static interface Port.ReadListener

public static interface **Port.In**<Type>

All Known Subinterfaces:

Channel<Type>

All Known Implementing Classes:

BufferedChan, Chan

Enclosing interface:

Port

Method Summary

boolean enabled(Port.ReadListener listener)

Returns: true if the port has an element available; false if there's no element ready; If the listener is non-null, then it is to be inform()ed when next an element becomes available.

Type read()

Read an element from the port if (or as soon as) one becomes available.

public static interface **Port.Out**<Type>

All Known Subinterfaces:

Channel<Type>

All Known Implementing Classes:

BufferedChan, Chan

Enclosing interface:

Port

Method Summary

boolean canWrite()

Checks if there is a reader waiting

void write(Type item)

Write an element to the port as soon as a reader is ready to accept it.

public static interface **Port.ReadListener**

Enclosing interface:

Port

Method Summary

void inform()

Gets invoked when a read occurs on a channel

public abstract class **Proc**
extends java.lang.Object
implements Process

All Implemented Interfaces:

java.lang.Runnable, Process

Direct Known Subclasses:

AbstractResponder, ConnectionAggregator, FileAuthoriser, SessionDecorator, SimpleDeliverer, SimpleDispatcher, SimpleRouter, SimpleSSLDispatcher, SimpleTransformer

---

Field Summary

static int MAX_PRIORITY
static int MIN_PRIORITY
protected java.lang.String name
static int NORM_PRIORITY
protected int priority
protected static int proc

Count of anonymous processes

Constructor Summary

Proc()

This is the default constructor

Proc(java.lang.String name)

This constructor accepts the Proc's name

Method Summary

void fork()

Start in a new thread; run in the background; dispose of thread on termination

void fork(Process.Epilogue epilogue)

..call (after.invoke()) on termination, before disposing of thread.

abstract void run()

Run to completion (in the current thread)

void serve()

Start as a server in a new thread; run in the background; dispose of thread on termination

void sleep(long ms)

initiates a sleep

Proc withPriority(int priority)

assigns the priority

public interface **Process**
extends java.lang.Runnable

All Superinterfaces:

java.lang.Runnable

All Known Implementing Classes:

AbstractResponder, ConnectionAggregator, FileAuthoriser, FileServiceResponder, Proc, SessionDecorator, SimpleCache, SimpleDeliverer, SimpleDispatcher, SimpleRouter, SimpleSSLDispatcher, SimpleTransformer, TimeResponder, TraceResponder

---

Nested Class Summary

static interface Process.Epilogue

Method Summary

void fork()

Start in a new thread; run in the background; dispose of thread on termination

void fork(Process.Epilogue after)

..call (after.invoke()) on termination, before disposing of thread.

void run()

Run to completion (in the current thread)

void serve()

Start as a server in a new thread; run in the background; dispose of thread on termination

public static interface **Process.Epilogue**

Enclosing interface:

Process

Method Summary

void invoke()

Get's invoked at the completion of the process

A.2 net.zuze.msc.artefacts

public abstract class **AbstractMessage**
extends java.lang.Object
implements Message

Direct Known Subclasses:

Request, Response

Constructor Summary

AbstractMessage()

This is the default constructor

Method Summary

Session getSession()

This method returns the session object associated with this message (if any)

void setSession(Session s)

This method is used for attaching a session object to the message

public class **CachedRawResponse**
extends Response

Field Summary

long timeExpires

This is the time when the Cached Response becomes stale

Constructor Summary

CachedRawResponse(Response r)

This is the default constructor

Method Summary

java.io.InputStream getData()

Sets up a new InputStream on the entity byte array which was copied from the original Response

java.lang.String getDataType()

Delegates to the underlying base Response: Returns the content type of the data in the Response

java.util.Properties getHeaders()

Delegates to the underlying base Response: Retrieves the header mappings

java.lang.String getLength()

Delegates to the underlying base Response: Retrieves the length of the entity body

java.lang.String getStatus()

Delegates to the underlying base Response: Retrieves the Response Status Code

public class **CachedWrappedResponse**
extends Response

Constructor Summary

CachedWrappedResponse(CachedRawResponse c, java.lang.String theConn, java.lang.String theTimeOut)

c - The raw cached response
theConn - The value of the @Connection header from the request
theTimeOut - The value of the @Timeout header from the request

Method Summary

java.io.InputStream getData()

Delegates to the underlying cached response. Sets up a new InputStream on the entity byte array which was copied from the original Response

java.lang.String getDataType()

Delegates to the underlying cached Response: Returns the content type of the data in the Response

java.util.Properties getHeaders()

Delegates to the underlying cached Response: Returns the updated headers with the connection information added

java.lang.String getLength()

Delegates to the underlying cached Response: Retrieves the length of the entity body

java.lang.String getStatus()

Delegates to the underlying cached Response: Retrieves the Response Status Code

public class **ChallengeRequest**
extends Request

Constructor Summary

ChallengeRequest(java.lang.String realm, java.lang.String message, java.lang.String authType)

realm - The Realm that the requested resource is located in
message - The 'unauthorised!' message
authType - Indicates the strength of authentication (Basic or Digest)

Method Summary

java.util.Set<**java.lang.String**> getFormFieldSet()

Non Functional

Entity.File getFormFile(java.lang.String key)

Non Functional

java.lang.String getFormString(java.lang.String key)

Non Functional

java.io.InputStream getNonFormData(int seq)

Non Functional

void release()

Non Functional

public class **ChallengeResponse**
extends Response

Constructor Summary

ChallengeResponse(java.lang.String realm, java.lang.String authType, java.lang.String msg)

authType - The type of Authentication that this challenge demands (Basic or Digest)
realm - The realm to which authentication is required
msg - The 'unauthorised!' message to the user

Method Summary

java.io.InputStream getData()

Returns the Stream of data for the body

java.lang.String getDataType()

Returns the type of the data in the body

java.util.Properties getHeaders()

Returns the Mapping of the header names to their values

java.lang.String getLength()

Returns the length of the body of the response

java.lang.String getStatus()

returns the HTTP response status code

public class **Connection**
extends java.lang.Object

Field Summary

java.io.InputStream in

The InputStream

java.io.OutputStream out

The OutputStream

java.net.InetAddress source

The InetAddress

Constructor Summary

Connection(java.net.Socket soc)

soc - The original socket

Method Summary

void close()

Used to close a connection

Port.Out getPath(Proc p)

Retrieves the output port that the connection should go down when there is a choice

boolean isInputShutDown()

Checks if the InputStream has been shutdown

void setPath(Proc p, Port.Out po)

This is invoked by a Connection Aggregator the first time it sees a connection and has decided which path to allocate it to

void shutDownInput()

Shuts down the InputStream for graceful closing

public class **ErrorRequest**
extends Request

Constructor Summary

ErrorRequest(java.lang.String et, java.lang.String txt, Request r)

et - the HTTP error type
txt - the message for the client
r - the original request

ErrorRequest(java.lang.String et, java.lang.String txt)

This constructor only gets called from the transformer and adds the 'close' command to the error to tell the client that it won't accept any more Requests through the connection
et - the HTTP error type
txt - the message for the client

Method Summary

java.util.Set<**java.lang.String**> getFormFieldSet()

Non Functional

Entity.File getFormFile(java.lang.String key)

Non Functional

java.lang.String getFormString(java.lang.String key)

Non Functional

java.io.InputStream getNonFormData(int seq)

Non Functional

void release()

Non Functional

public class **ErrorResponse**
extends Response

Constructor Summary

ErrorResponse(java.lang.String status, java.lang.String msg)

status - The HTTP error code
msg - The Message to the client

Method Summary

java.io.InputStream getData()

Returns the Stream of data for the body

java.lang.String getDataType()

Returns the type of the data in the body

java.util.Properties getHeaders()

Returns the Mapping of the header names to their values

java.lang.String getLength()

Returns the length of the body of the response

java.lang.String getStatus()

Returns the HTTP response status code

void setStatus(java.lang.String s)

updates the status code of the response, ok, bad, etc

public class **FileResponse**
extends Response

Constructor Summary

FileResponse(java.io.InputStream is, java.lang.String mime, java.lang.String length)

is - the input stream to the data
mime - the MIME type of the data
length - the length of the data

Method Summary

java.io.InputStream getData()

Returns the Stream of data for the body

java.lang.String getDataType()

Returns the type of the data in the body

java.util.Properties getHeaders()

Returns the Mapping of the header names to their values

java.lang.String getLength()

Returns the length of the body of the response

java.lang.String getStatus()

returns the HTTP response status code

public interface **Message**

All Known Implementing Classes:

AbstractMessage, CachedRawResponse, CachedWrappedResponse, ChallengeRequest, ChallengeResponse, ErrorRequest, ErrorResponse, FileResponse, Request, Response, SimpleRequest, StatusResponse, TimeResponse, TraceResponse

Method Summary

Session getSession()

This is used to retrieve the session attached to a message

void setSession(Session s)

This sets the session that is attached to the message

public class **Packet**<TYPE extends Message>
extends java.lang.Object

Constructor Summary

Packet(TYPE m, Connection c)

m - the message object
c - the connection that it is associated with

Method Summary

Connection getConnection()

Retrieve the connection context

TYPE getMessage()

Retrieve the message (Request or Response) from the Packet

public abstract class **Request**
extends AbstractMessage

Direct Known Subclasses:

ChallengeRequest, ErrorRequest, SimpleRequest

Constructor Summary

Request()

This is the default constructor

Method Summary

FileResponse getFileResponse(java.io.InputStream is, java.lang.String mime, java.lang.String length)

Used by a Responder to turn a File Request into a File Response

abstract java.util.Set<**java.lang.String**> getFormFieldSet()

Returns the set of fields from the form data

abstract Entity.File getFormFile(java.lang.String key)

Abstract method which when implemented retrieves the File that was uploaded with the form field specified in the key

abstract java.lang.String getFormString(java.lang.String key)

An abstract method which when implemented retrieves the string version of a form field

java.util.Set<**java.lang.String**> getHeaderSet()

Retrieves the set of header names in the Request

java.lang.String getHeaderValue(java.lang.String id)

gets the header value for a header id

java.lang.String getMethod()

Retrieves the method for the request

abstract java.io.InputStream getNonFormData(int seq)

Abstract method which when implemented returns data that was uploaded as part of the entity but not in a form

java.lang.String getURI()

retrieves the URI for the request

java.lang.String getVersion()

retrieves the HTTP version of the request

abstract void release()

Releases the resources used up by the request (temp files on disk)

void removeHeader(java.lang.String key)

Removes a header with a specific key

void setHeader(java.lang.String key, java.lang.String value)

used to set the value for a header in the request

public abstract class **Response**
extends AbstractMessage

Direct Known Subclasses:

CachedRawResponse, CachedWrappedResponse, ChallengeResponse, ErrorResponse, FileResponse, StatusResponse, TimeResponse, TraceResponse

Constructor Summary

Response()

This is the default constructor

Method Summary

abstract java.io.InputStream getData()

Returns the Stream of data for the body

abstract java.lang.String getDataType()

Returns the type of the data in the body

abstract java.util.Properties getHeaders()

Returns the Mapping of the header names to their values

abstract java.lang.String getLength()

Returns the length of the body of the response

abstract java.lang.String getStatus()

returns the HTTP response status code

public class **RouteAlgorithm**
extends java.lang.Object
implements RoutingAlgorithm

Constructor Summary

RouteAlgorithm()

This is the default constructor

Method Summary

java.util.Set<**java.lang.String**> getServices()

This method returns the set of message types that the algorithm can identify

java.lang.String interpret(Request r)

This method looks at the URI and determines the type of resource that is being requested. The name of the resource is returned.

public class **SimpleRequest**
extends Request

Constructor Summary

SimpleRequest(Connection conn) throws BadRequestException, InternalServerException, TimeoutException

Throws:
BadRequestException - Can happen if Request is Malformed, etc
InternalServerException - An unexplained error
TimeoutException - Happens when no data comes in on the InputStream for a long time

Parameters:
conn - the connection that the request is associated with

Method Summary

protected void finalize()

The code to clean up the object when done

java.util.Set<**java.lang.String**> getFormFieldSet()

This gets the set of keys in the contentFields

Entity.File getFormFile(java.lang.String key)

Retrieves a file that was uploaded as part of a form

java.lang.String getFormString(java.lang.String key)

Retrieves the String representation of a form field

java.io.InputStream getNonFormData(int index)

Retrieve data files that were uploaded without a form

void release()

Releases resources like temporary files

public class **StatusResponse**
extends Response

Constructor Summary

StatusResponse(java.lang.String status)

status - The HTTP status code

Method Summary

java.io.InputStream getData()

Returns the InputStream of data for the body

java.lang.String getDataType()

Returns the type of the data in the body

java.util.Properties getHeaders()

Returns the Mapping of the header names to their values

java.lang.String getLength()

Returns the length of the body of the response

java.lang.String getStatus()

returns the HTTP status code

public class **TimeResponse**
extends Response

Constructor Summary

TimeResponse()

The default constructor

TimeResponse(java.lang.String s)

s - The custom message

Method Summary

java.io.InputStream getData()

Returns the InputStream of data for the body

java.lang.String getDataType()

Returns the type of the data in the body

java.util.Properties getHeaders()

Returns the Mapping of the header names to their values

java.lang.String getLength()

Returns the length of the body of the response

java.lang.String getStatus()

returns the HTTP status code

public class **TraceResponse**
extends Response

Constructor Summary

TraceResponse(Request req)

req - The Request that asked for the Trace

Method Summary

java.io.InputStream getData()

Returns the InputStream of data for the body, which in this case comprises the original request

java.lang.String getDataType()

Returns the type of the data in the body

java.util.Properties getHeaders()

Returns the Mapping of the header names to their values

java.lang.String getLength()

Returns the length of the body of the response

java.lang.String getStatus()

returns the HTTP status code

A.3 net.zuze.msc.artefacts.util

public interface **Entity**

Nested Class Summary

static interface Entity.File

An individual file that has been uploaded

public static interface **Entity.File**

Enclosing interface:

Entity

Method Summary

java.io.InputStream getInputStream()

Retrieves the InputStream to the file

public interface **Session**

All Known Implementing Classes:

SimpleSession

Method Summary

java.lang.Object getAttribute(java.lang.String key)

Retrieves an attribute from the Session

java.util.Enumeration<**java.lang.String**> getAttributeNames()

Retrieves the names of all the attributes in the session

long getCreationTime()

Retrieves the time that the session was created

java.lang.String getId()

Retrieves the session's id value

long getTimeout()

Retrieves the idle timeout value

boolean hasExpired()

Indicates if the session has been idle too long

void invalidate()

Makes a session object invalid

void setAttribute(java.lang.String key, java.lang.Object o)

sets an attribute in the session

void setTimeout(long t)

Sets how long the session can remain in memory without having been accessed

public class **SessionMap**
extends java.lang.Object

Constructor Summary

SessionMap(int cleanuptime)

cleanuptime - The interval time between cleanup runs

Method Summary

Session get(java.lang.String key)

Retrieves a session from memory

void put(java.lang.String key, Session value)

puts a Session in memory

void remove(java.lang.String key)

Removes a Session from memory

public class **SimpleSession**
extends java.lang.Object
implements Session

Constructor Summary

SimpleSession(java.lang.String id, int tout)

id - The Session ID
tout - The Session timeout

Method Summary

java.lang.Object getAttribute(java.lang.String key)

Retrieves an attribute from the session

java.util.Enumeration<**java.lang.String**> getAttributeNames()

Retrieves the names of all the attributes in the Session object

long getCreationTime()

Retrieves the time the Session was created

java.lang.String getId()

Retrieves the ID of the Session

long getTimeout()

Retrieves the idle timeout period value

boolean hasExpired()

Returns a boolean value indicating if the Session has been idle too long

void invalidate()

Sets the valid state to false

void setAttribute(java.lang.String key, java.lang.Object o)

Sets the value of an attribute into the session

void setTimeout(long t)

Sets the idle timeout for the session

public class **SmartHashMap**<KeyType,ValueType>
extends java.lang.Object

Constructor Summary

SmartHashMap(int size)

size - The minimum guaranteed size

Method Summary

ValueType get(KeyType key)

Retrieve an item from the Cache

void put(KeyType key, ValueType value)

Puts a new item in the Cache

A.4 net.zuze.msc.util

public class **LineReader**
extends java.io.BufferedInputStream

Field Summary

java.io.InputStream base

handle on the base stream

Constructor Summary

LineReader(java.io.InputStream base)

base - This is the InputStream that will be turned into a LineReader

Method Summary

int bytesRead()

Returns the number of bytes read so far

int read()

overridden method, keeps a count of bytesRead

java.lang.String readBody(int len)

This method is used to read in a urlencoded entity

java.lang.String readLine()

Reads a line up to an eol sequence consisting of: \r, \r \n, or \n.

public class **ServerSettings**
extends java.lang.Object

Constructor Summary

This is a singleton, so the constructor is not public

Method Summary

boolean allowDirList()

Returns whether or not directory listings should be allowed

int CacheMaxAge()

Returns the maximum age for cached files before they go stale

java.lang.Object clone()

This is not allowed since ServerSettings is a singleton

int connectionPollTime()

Returns the interval time between polls for data on the socket InputStream

int connectionTimeout()

Returns the maximum idle time of that a connection is allowed to have

boolean doCache()

Indicates whether or not caching information will be put in outgoing Responses

static ServerSettings getInstance()

It is a singleton, so return the single instance

java.lang.String homeDir()

Returns the location of the home directory for the file server

int maxRequests()

Returns the maximum number of requests that the server will process through a particular connection

int numTransformers()

Returns the number of Transformers in the system

int serverPort()

The default Port that the Dispatcher should use

boolean traceConnectionDumps()

true if notification should be given about connections being dumped after the maximum number of Requests have been received through it

boolean traceConnections()

Returns a boolean value to indicate whether notification should be given when new connections are received by the Transformer

boolean traceConnectionTimeouts()

Returns true if notification of connection timeouts should be output

boolean traceExceptions()

Returns true if exception information should be output

boolean traceService()

Returns true if notification should be given when the Transformer has processed a Request

public class **ServerUtils**
extends java.lang.Object

Constructor Summary

ServerUtils()

This is the default constructor

Method Summary

static java.lang.String encodeUri(java.lang.String uri)

Applies the standard URL encoding to a String

static java.lang.String getTime()

Returns the current time in the correct HTTP time format

static java.lang.String getTime(java.util.Date d)

Returns the value of the Date Supplied in the correct HTTP time format

A.5 net.zuze.msc.protocols

public class **HTTP**
extends java.lang.Object

Constructor Summary

This is a singleton, so the constructor is not public

Method Summary

java.lang.Object clone()

This is not allowed since HTTP is a singleton

java.lang.String getCode(java.lang.String status)

Retrieves the status code for a given status name

static HTTP getInstance()

Returns the active instance of the HTTP Class

java.lang.String getMimeType(java.lang.String ext)

Retrieves a MIME type for a given file extension

A.6 net.zuze.msc.Exceptions

public class **AssemblyException**
extends java.lang.Exception

Constructor Summary

AssemblyException(java.lang.String s)

s - The exception message

public class **BadRequestException**
extends java.lang.Exception

Constructor Summary

BadRequestException(java.lang.String s)

s - The exception message

public class **ForbiddenException**
extends java.lang.Exception

Constructor Summary

ForbiddenException(java.lang.String s)

s - The exception message

public class **InternalServerException**
extends java.lang.Exception

Constructor Summary

InternalServerException(java.lang.String s)

s - The exception message

public class **NoServiceException**
extends java.lang.Exception

Constructor Summary

NoServiceException(java.lang.String s)

s - The exception message

public class **RedirectException**
extends java.lang.Exception

Field Summary

java.lang.String location

The URL that the client is being redirected to

java.lang.String location

The message about the redirection

Constructor Summary

RedirectException(java.lang.String location, java.lang.String msg)

location - The new location that the client is being redirected to
msg - The message to the client

public class **TimeoutException**
extends java.lang.Exception

Constructor Summary

TimeoutException(java.lang.String s)

s - The exception message

A.7 net.zuze.msc.skeletons

public interface **Authoriser**<Type>

extends One2OneComponent<Type,Type>

All Superinterfaces:

Component, Consumer<Type>, One2OneComponent<Type,Type>, Producer<Type>

All Known Implementing Classes:

FileAuthoriser

public interface **Cache**<InType,OutType>
extends Proxy<InType,OutType>

All Superinterfaces:

Component, Consumer<InType>, One2OneComponent<InType,OutType>, Producer<OutType>, Proxy<InType,OutType>

All Known Implementing Classes:

SimpleCache

public interface **Component**

All Known Subinterfaces:

Authoriser<Type>, Cache<InType,OutType>, Consumer<InType>, Deliverer<Type>, Dispatcher<Type>, One2NComponent<Type>, One2OneComponent<InType,OutType>, Producer<OutType>, Proxy<InType,OutType>, Responder<InType,OutType>, Router<Type>, Transformer<InType,OutType>

All Known Implementing Classes:

AbstractResponder, ConnectionAggregator, FileAuthoriser, FileServiceResponder, SessionDecorator, SimpleCache, SimpleDeliverer, SimpleDispatcher, SimpleRouter, SimpleSSLDispatcher, SimpleTransformer, TimeResponder, TraceResponder

Method Summary

void start()

starts one thread

void start(int num)

starts num threads

void stop()

stops the component

public interface **Consumer**<InType>
extends Component

Method Summary

Consumer<InType> getComponentCopy()

Returns a new copy of this type of component

void setInputPort(Port.In<InType> in)

Assigns the Input Port for the Consumer Component

public interface **Deliverer**<Type>
extends Consumer<Type>

All Superinterfaces:

Component, Consumer<Type>

All Known Implementing Classes:

SimpleDeliverer

public interface **Dispatcher**<Type>

extends Producer<Type>

All Superinterfaces:

Component, Producer<Type>

All Known Implementing Classes:

SimpleDispatcher, SimpleSSLDispatcher

public interface **One2NComponent**<Type>
extends Consumer<Type>

All Superinterfaces:

Component, Consumer<Type>

All Known Subinterfaces:

Router<Type>

All Known Implementing Classes:

ConnectionAggregator, SimpleRouter

Method Summary

void setOutputPort(java.lang.String id, Port.Out<Type> o)

Assigns 'o' to the output port identified by 'id'

public interface **One2OneComponent**<InType,OutType>
extends Producer<OutType>, Consumer<InType>

All Superinterfaces:

Component, Consumer<InType>, Producer<OutType>

All Known Subinterfaces:

Authoriser<Type>, Cache<InType,OutType>, Proxy<InType,OutType>,
Responder<InType,OutType>, Transformer<InType,OutType>

All Known Implementing Classes:

AbstractResponder, FileAuthoriser, FileServiceResponder, SessionDecorator, SimpleCache, SimpleTransformer, TimeResponder, TraceResponder

Method Summary

void setOutputPort(java.lang.String id, Port.Out<Type> o)

Assigns 'o' to the output port identified by 'id'

public interface **Producer**<OutType>
extends Component

Method Summary

void setOutputPort(Port.Out<OutType> out)

Assigns the Output Port for the Producer Component

public interface **Proxy**<InType,OutType>
extends One2OneComponent<InType,OutType>

All Superinterfaces:

Component, Consumer<InType>, One2OneComponent<InType,OutType>, Producer<OutType>

All Known Subinterfaces:

Cache<InType,OutType>

All Known Implementing Classes:

SessionDecorator, SimpleCache, SubSystemCap

Method Summary

void setSubInput(Port.In<OutType> in)

This is the input port for data from the enclosing component

void setSubOutput(Port.Out<InType> out)

This is the output port for data to the enclosing component

public interface **Responder**<InType,OutType>
extends One2OneComponent<InType,OutType>

All Superinterfaces:

Component, Consumer<InType>,
One2OneComponent<InType,OutType>, Producer<OutType>

All Known Implementing Classes:

AbstractResponder, FileServiceResponder, SimpleCache, TimeResponder,
TraceResponder

public interface **Router**<Type>
extends One2NComponent<Type>

All Superinterfaces:

Component, Consumer<Type>, One2NComponent<Type>

All Known Implementing Classes:

SimpleRouter

public interface **RoutingAlgorithm**

All Known Implementing Classes:

RouteAlgorithm

Method Summary

java.util.Set<**java.lang.String**> getServices()

Returns a set of all the types of requests that it can identify

java.lang.String interpret(Request r)

Returns the name of the type of Request 'r'

public interface **Server**

All Known Implementing Classes:

TestServer

Method Summary

<**T**> **Channel**<**T**> addConnector(Channel<T> c)

Adds a Channel to the system

<**T**> **Consumer**<**T**> addConsumer(Consumer<T> cons, int num)

Adds a Consumer Component to the system with `num` child threads

<**T**> **Consumer**<**T**> addConsumerPool(Consumer<T> cons, int num)

Adds a Consumer Pool to the system with `num` cloned Consumers

<**T**> **One2NComponent**<**T**> addOne2NComp(One2NComponent<T> o2n)

Adds a One2NComponent to the system.

<**T,Y**> **One2OneComponent**<**T,Y**> addOne2OneComp(One2OneComponent<T,Y> o2o, int num)

Adds a One2OneComponent to the system with `num` child threads

<**T,Y**> **One2OneComponent**<**T,Y**> addOne2OnePool(One2OneComponent<T,Y> o2o, int num)

Adds a One2OneComponent Pool to the system with `num` cloned One2OneComponents

<**T**> **Producer**<**T**> addProducer(Producer<T> prod, int num)

Adds a Producer to the system with `num` child threads

<**T,Y**> **Proxy**<**T,Y**> addProxy(Proxy<T,Y> prox)

Adds a Proxy to the system

<**T**> **void** attach(One2NComponent<T> producer, java.lang.String outID, Consumer<T> consumer, Channel<T> c)

Attaches the `OutID` port of the One2NComponent to the Consumer using Channel c

<**T**> **void** attach(Producer<T> producer, Consumer<T> consumer, Channel<T> c)

Attaches the producer's output port to the consumer's input port via Channel c

<**T1,T2**> **void** attachSub(Proxy<T1,T2> prox, One2OneComponent<T1,T2> sub)

Attaches a One2OneComponent as a subordinate of a Proxy

void start()

Starts all the components in the Server

void stop()

Stops the Server

public interface **Transformer**<InType,OutType>
extends One2OneComponent<InType,OutType>

All Superinterfaces:

Component, Consumer<InType>, One2OneComponent<InType,OutType>, Producer<OutType>

All Known Implementing Classes:

SimpleTransformer

A.8 net.zuze.msc.simpleserver

public abstract class **AbstractResponder**
extends Proc
implements Responder<Packet<Request>,Packet<Response>>

Constructor Summary

AbstractResponder()

This is the default constructor

Method Summary

void run()

The processing of Requests is set off from here

void setInputPort(Port.In<Packet<Request>> in)

Assigns the Input Port for the Component

void setOutputPort(Port.Out<Packet<Response>> out)

Assigns the Output Port for the Component

void start()

starts one thread

void start(int num)

starts num threads

void stop()

stops the component

public class **ConnectionAggregator**<T extends Packet<Message>>
extends Proc
implements One2NComponent<T>

Constructor Summary

ConnectionAggregator(Port.In<T> in, int numOfOuts)

in - The input port
numOfOuts - The number of output ports

Method Summary

void run()

Messages are input and distributed from here

void setInputPort(Port.In<T> in)

Sets the input port

void setOutputPort(java.lang.String id, Port.Out<T> o)

Sets one of the output ports

void start()

forks 1 child thread

void start(int num)

forks `num` child threads

void stop()

stops the component

public class **FileAuthoriser**
extends Proc
implements Authoriser<Packet<Request>>

Constructor Summary

FileAuthoriser(Port.In<Packet<Request>> in, Port.Out<Packet<Request>> out)

in - The input port
out - The output port

Method Summary

void run()

Reads in request packets and checks for authorisation

void setInputPort(Port.In<Packet<Request>> in)

Sets the input port

void setOutputPort(java.lang.String id, Port.Out<Packet<Request>> o)

Sets one of the output ports

void start()

forks 1 child thread

void start(int num)

forks `num` child threads

void stop()

stops the component

public class **FileServiceResponder**
extends AbstractResponder

Constructor Summary

FileServiceResponder(Port.In<Packet<Request>> in,
Port.Out<Packet<Response>> out)

in - The input port
out - The output port

Method Summary

protected Response doDELETE(Request r)

Does nothing, may eventually be implemented

protected Response doFORM(Request r)

This method currently passes on control to doPost

protected Response doGET(Request r)

This performs the retrieval of the file and the construction of the FileResponse object

protected Response doPOST(Request r)

This method implements posting data to a resource

protected Response doPUT(Request r)

Does nothing, may eventually be implemented

void toExecute(Packet<Request> job)

The code for servicing a request is here

public class **SessionDecorator**
extends Proc
implements Proxy<Packet<Request>,Packet<Response>>

Constructor Summary

SessionDecorator(Port.In<Packet<Request>> in, Port.In<Packet<Response>> rin,
Port.Out<Packet<Response>> out, Port.Out<Packet<Request>> rout)

in - The Request input port
rin - The Response input port from the Responder
out - The Response output port
rout - The Request output port to the Responder

Method Summary

void run()

The processing of Requests is set off from here

void setInputPort(Port.In<Packet<Request>> in)

Assigns the Input Port for the Component

void setOutputPort(Port.Out<Packet<Response>> out)

Assigns the Output Port for the Component

void setSubInput(Port.In<Packet<Response>> in)

Assigns the Input Port for data from the enclosing component

void setSubOutput(Port.Out<Packet<Request>> out)

Assigns the Output Port for data to the enclosing component

void start()

starts one thread

void start(int num)

starts num threads

void stop()

stops the component

public class **SimpleCache**
extends AbstractResponder
implements Cache<Packet<Request>,Packet<Response>>

Constructor Summary

SimpleCache(Port.In<Packet<Request>> in, Port.In<Packet<Response>> rin, Port.Out<Packet<Response>> out, Port.Out<Packet<Request>> rout)

in - The Request input port

rin - The Response input port from the Responder
out - The Response output port
rout - The Request output port to the Responder for requests that the cache cannot service

Method Summary

void setSubInput(Port.In<Packet<Response>> in)

Assigns the Input Port for data from the enclosing component

void setSubOutput(Port.Out<Packet<Request>> out)

Assigns the Output Port for data to the enclosing component

public class **SimpleDeliverer**
extends Proc
implements Deliverer<Packet<Response>>

Constructor Summary

SimpleDeliverer(Port.In<Packet<Response>> in)

in - The input Response channel

Method Summary

void run()

The delivery of Responses is set off from here

void setInputPort(Port.In<Packet<Response>> in)

Assigns the Input Port for the Component

void start()

starts one thread

void start(int num)

starts num threads

void stop()

stops the component

public class **SimpleDispatcher**
extends Proc
implements Dispatcher<Connection>

Constructor Summary

SimpleDispatcher(Port.Out<Connection> out, int port)

out - This is the front end of the Connection channel to a Transformer
port - This is the port number that the server will listen on

Method Summary

void run()

Socket connections are accepted from here and Connections channelled to a Transformer

void setOutputPort(Port.Out<Connection> out)

Assigns the Output Port for the Component

void start()

starts one thread

void start(int num)

starts num threads

void stop()

stops the component

public class **SimpleRouter**
extends Proc
implements Router<Packet<Request>>

Constructor Summary

SimpleRouter(Port.In<Packet<Request>> in, RoutingAlgorithm alg)

in - the input request port
alg - the algorithm used to distinguish one type of request from another

Method Summary

void run()

The routing algorithm is applied to packets which are then channelled to the appropriate responder

void setInputPort(Port.In<Packet<Request>> in)

Assigns the Input Port for the Component

void setOutputPort(java.lang.String id, Port.Out out)

Assigns one of the Output Ports for the Router

void start()

starts one thread

void start(int num)

starts num threads

void stop()

stops the component

public class **SimpleSSLDispatcher**
extends Proc
implements Dispatcher<Connection>

Constructor Summary

SimpleSSLDispatcher(Port.Out<Connection> out, int port)

out - This is the front end of the Connection channel to a Transformer
port - This is the port number that the server will listen on

Method Summary

void run()

SSL Socket connections are accepted from here and Connections channelled to a Transformer

void setOutputPort(Port.Out<Connection> out)

Assigns the Output Port for the Component

void start()

starts one thread

void start(int num)

starts num threads

void stop()

stops the component

public class **SimpleTransformer**
extends Proc
implements Transformer<Connection,Packet<Request>>

Constructor Summary

SimpleTransformer(Port.In<Connection> in, Port.Out<Packet<Request>> out)

in - The Connection Input Port from a Dispatcher
out - The Request Output Port towards a Responder

Method Summary

void run()

Connections are taken off the channel one by one and turned into Requests here

void setInputPort(Port.In<Connection> in)

Assigns the Input Port for the Component

void setOutputPort(Port.Out<Packet<Request>> out)

Assigns the Output Port for the Component

void start()

starts one thread

void start(int num)

starts num threads

void stop()

stops the component

public class **SubSystemCap**<T,Y>
extends java.lang.Object
implements Proxy<T,Y>

Constructor Summary

SubSystemCap()

default constructor

Method Summary

One2OneComponent<T,Y> getComponentCopy()

Returns a new copy of this type of component

void run()

The processing of Requests is set off from here

void setInputPort(Port.In<Packet<Request>> in)

Assigns the Input Port for the Component

void setOutputPort(Port.Out<Packet<Response>> out)

Assigns the Output Port for the Component

void setSubInput(Port.In<Packet<Response>> in)

Assigns the Input Port for data from the enclosing sub-system

void setSubOutput(Port.Out<Packet<Request>> out)

Assigns the Output Port for data to the enclosing sub-system

void start()

starts one thread

void start(int num)

starts num threads

void stop()

stops the component

public class **TestServer**

extends java.lang.Object
implements Server

Method Summary

<**T**> **Channel**<**T**> addConnector(Channel<T> c)

Adds a Channel to the system

<**T**> **Consumer**<**T**> addConsumer(Consumer<T> cons, int num)

Adds a Consumer Component to the system with `num` child threads

<**T**> **Consumer**<**T**> addConsumerPool(Consumer<T> cons, int num)

Adds a Consumer Pool to the system with `num` cloned Consumers

<**T**> **One2NComponent**<**T**> addOne2NComp(One2NComponent<T> o2n)

Adds a One2NComponent to the system.

<**T,Y**> **One2OneComponent**<**T,Y**> addOne2OneComp(One2OneComponent<T,Y> o2o, int num)

Adds a One2OneComponent to the system with `num` child threads

<**T,Y**> **One2OneComponent**<**T,Y**> addOne2OnePool(One2OneComponent<T,Y> o2o, int num)

Adds a One2OneComponent Pool to the system with `num` cloned One2OneComponents

<**T**> **Producer**<**T**> addProducer(Producer<T> prod, int num)

Adds a Producer to the system with `num` child threads

<**T,Y**> **Proxy**<**T,Y**> addProxy(Proxy<T,Y> prox)

Adds a Proxy to the system

<**T**> **void** attach(One2NComponent<T> producer, java.lang.String outID, Consumer<T> consumer, Channel<T> c)

Attaches the `OutID` port of the One2NComponent to the Consumer using Channel c

<**T**> **void** attach(Producer<T> producer, Consumer<T> consumer, Channel<T> c)

Attaches the producer's output port to the consumer's input port via Channel c

<**T1,T2**> **void** attachSub(Proxy<T1,T2> prox, One2OneComponent<T1,T2> sub)

Attaches a One2OneComponent as a subordinate of a Proxy

void start()

Starts all the components in the Server

void stop()

Stops the Server

public class **TimeResponder**
extends AbstractResponder

Constructor Summary

TimeResponder(Port.In<Packet<Request>> in, Port.Out<Packet<Response>> out)

in - The channel for the request packets
out - the channel for the time responses

public class **TraceResponder**
extends AbstractResponder

Constructor Summary

TraceResponder(Port.In<Packet<Request>> in, Port.Out<Packet<Response>> out)

in - The channel for the request packets
out - the channel for the trace responses

Appendix B

Contents of The CD

B.1 /dissertation

This document in PDF form.

B.2 /sourcecode

The Java source files.

B.3 /javadoc

The HTML version of the Application Programming Interface.

B.4 /build

The Java classes and dependency files.

Bibliography

[1] D. Batory: *A Tutorial on Feature Oriented Programming and Product-Lines*, IEEE (2002)

[2] Bowen R., Coar k.: *Apache Server Unleashed*, Sams (2000)

[3] Czarnecki K., Eisenecker U.W.: *Generative Programming*, Addison-Wesley (2000)

[4] Fielding et al: *Hypertext Transfer Protocol – HTTP/1.1*, Network Working Group, RFC 2616 (1999)

[5] Gamma, Helm, Johnson, Vlissides: *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison Wesley (1994)

[6] Garlan, Monroe, and Wile: Acme: Architectural Descriptions of Component-Based Systems, *Foundations of Component-Based Systems pp 47-68*, Cambridge University Press (2000)

[7] Gourley D., Totty B.: *HTTP The Definitive Guide*, O'Reilly (2002)

[8] IBM Research:
Subject Oriented Programming, http://www.research.ibm.com/sop/, Retrieved (July 29th, 2005)

[9] IBM Research:
Hyper/J: Multi-Dimensional Separation of Concerns for Java, http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm, Retrieved (July 30th, 2005)

[10] IBM Research:
The AspectJ Project for Eclipse, http://eclipse.org/aspectj/, Retrieved (July 30th, 2005)

[11] Miles, Russ: *AspectJ Cookbook*, O'Reilly (2005)

[12] Nierstrasz O., Tsichritzis D.: *Object Oriented Software Composition*, Prentice Hall (1995)

[13] Oaks, Scott: *Java Security*, O'Reilly (2001)

[14] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel: Flash: An Efficient and Portable Web Server *Proceedings of the 1999 USENIX Annual Technical Conference*, (1999)

[15] Szyperski, Clemens: *Component Software, Beyond Object Oriented Programming*, Alm Press (2002)

[16] M. Welsh, D. Culler, E. Brewer: SEDA: An Architecture for Well-Conditioned, Scalable Internet Services, *Symposium on Operating Systems Principles pp 230-243*, ACM Press (2001)

[17] Woodcock J., Davies J.: *Using Z: Specification, Refinement, and Proof*, Prentice Hall (1996)