

Patia: Adaptive Distributed Webserver

(A Position Paper)

Julie A McCann, Gawesh Jawaheer & Linxue Sun

Department of Computing

Imperial College London

SW1 2BZ, UK

{jamm, gawesh, linxue}@doc.ic.ac.uk

1 Abstract

This paper introduces the Patia Adaptive Webserver architecture, which is distributed and consists of semi-autonomous agents called FLYs. The FLY carries with it the set of rules and adaptivity policies required to deliver the data to the requesting client. Where a change in the FLY's external environment could affect performance, it is the FLY's responsibility to change the method of delivery (or the actual object being delivered). It is our conjecture that the success of today's multimedia websites in terms of performance lies in the architecture of the underlying servers and their ability to adapt to changes in demand and resource availability, as well as their ability to scale. We believe that the distributed and autonomous nature of this system are key factors in achieving this.

2 Introduction

Webservers play a very important role on the World Wide Web (WWW). With the constant growth of the Internet, and the increasing number of Web based applications, Webservers are under pressure to perform. When they fail to perform, end-users experience increased access latency. Indeed, [Huitema 2000] claims that 40% of Web delays are due to Webservers. Though Webservers only form one component of the WWW infrastructure, their study is motivated by the notion that server delays are becoming an increasingly dominant factor in user perceived Web performance [Barford 2001] and that the situation will likely worsen in the future, given Moore's Law (hardware capacity doubles every 18 months) and the prediction by [Gilder 1997] that network bandwidth will triple every year for the next 25 years. Further, the phenomenon known as *flash crowds*, referring to a situation where very large numbers of users simultaneously access a Website [Jung 2002], is becoming more prevalent. Technologies such as Web caches, which have been used to improve Webserver performance, have been proven to not eliminate flash crowds [Arlitt 2000].

Today web sites have moved beyond static HTML pages on single webservers and are required to host many types of media and documents whilst generating dynamic content or processing transactions. Further, with the introduction of technology for mobile phones and PDAs these sites must also maintain a lightweight version of their pages and must also store the mechanisms to generate this. Therefore, it is our conjecture that the success of such 'Websites' in terms of performance, lies in the architecture of the underlying servers and their ability to adapt to changes in demand and resource availability, as well as their ability to scale.

To this end the Patia project has designed and is implementing an adaptive Webserver. The key to this adaptation and scalability lies in the Patia Architecture, which is designed as a semi-autonomous decentralized system. The Patia system consists of essentially Webserver agents (each called a *FLY*) that carry out the function of a traditional Webserver but over a distributed collection of data. The FLY carries with it the set of rules and adaptivity policies required to deliver the data to the requesting client. Where a change in the FLY's external environment could affect performance (positively or negatively) it is the FLY's responsibility to change the method of delivery (or the actual object being delivered). Furthermore, where the FLY is currently residing on a given computing node, and it detects that that node is failing or is performing poorly, it can safely *fly* to another machine and continue communicating with the client. Should large numbers of users suddenly request pages as in a flash crowd, the FLY based system will expand gracefully by making use of less utilised machines (such as those found in a typing-pool etc.).

3 Distributed Webservers

Distributed Web-server systems are groups of Webservers connected over a LAN or WAN and operating as a single entity to serve requests for Web resources from clients over the Internet. Such LAN and WAN based systems are defined as locally and globally distributed Webserver systems respectively. One requirement for distributed Webserver systems is to be ‘client transparent’. Hence, this excludes mirrored Webserver systems where the user is presented with a choice of Webservers and has to decide which Webserver will serve the HTTP request (each server being a mirror image of the origin server in terms of Web content). The main aim of distributed Webserver systems is to achieve sustainable scalability in order to respond to increasing demands.

Over the years, various architectures for distributed Webserver systems have been proposed. [Schroeder 2000] broadly classified the locally distributed Webserver systems architectures. However, the latest, most consistent and rigorous taxonomy of locally distributed Webserver systems has been produced by [Cardellini 2002]. To summarise, [Cardellini 2002] divided locally distributed Webserver architectures into three classes: Web cluster, virtual Web cluster and distributed Web systems. They distinguished among these classes by examining the HTTP request routing mechanisms and request dispatching algorithms. Request routing pertains to the means by which an HTTP request from a client reaches a particular server node in the distributed Webserver system¹. Routing may involve Web switches to direct traffic or in the case of a virtual Web cluster, incoming Web traffic is directed to every node. Alternatively in a distributed Web system, where the IP address of each server node is visible on the Internet, routing and dispatching functionalities are achieved through the use of the authoritative Domain Name Server (DNS) [Brisco 1995].

Poor performance (or lack of response) is becoming more common, which is accentuated by unexpected flash crowds. Typically the solution to this is to add more and faster hardware and replicate the data to overcome this. However this solution means that some companies run Webservers at a capacity of over three times what is normally demanded so that the system can cope with flash crowding. Our alternative is to design an architecture that is less costly and yet can cope with sudden increases in requests and even failure in a graceful manner.

4 Self-Adaptive Systems

An adaptive system is one that can modify its behaviour based on stimulus produced either from within its system or from external sources. Today’s push towards *pervasive* computing requires that system’s units and their services have a degree of autonomy and therefore adaptivity would have to be quite self-contained. The more autonomy the unit requires, the more complex the processing where the system may have to evolve new behaviours. This is where the major tradeoffs lie – the more ‘intelligent’ the system is the more complex

the task to decide how to adapt is. This is in terms of the processing and information storage of monitoring feedback, the change optimisation and the cost of the reconfiguration itself

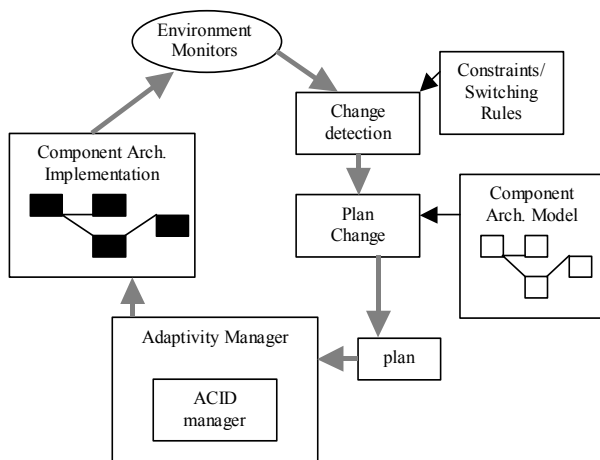


Figure 1 Adaptation Framework

Two major areas of computing have examined this subject from different angles. Artificial intelligence has focused on the rules and mechanisms to allow open-adaptive systems to evolve new behaviours, rules etc., typically focusing on very specific application areas [Oreizy 1999]. Alternatively Software Engineering has considered component-based architecture configuration languages and adaptation policy management [Magee 1995]. Thus far, closed-adaptive systems have only been considered, that is systems that do not modify their own behaviour. Either way, an adaptive system’s architecture must not preclude the type

¹ On the other hand, request dispatching algorithms relate to the decision policies upon which a server node is selected to serve a request

of adaptive system it supports. This means that the system must be able to provide facilities for dynamic reconfiguration, the ability to store adaptivity rules, and the ability to react to those rules in a way that does not compromise performance. There is no point in a system reacting to a problem so slowly that system fails before it can do anything about it. Furthermore componentisation itself must not produce excessive overheads.

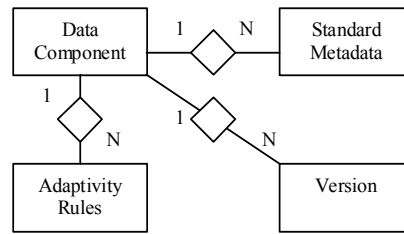


Figure 2 Data Component Structure

Figure 1 illustrates a general component-based adaptive system architecture. The main feature is that this system can self-(re)configure with the help from monitors, which provide environmental data (e.g. current performance statistics). The typical tools used to develop such systems consist of architecture description languages (ADLs) and constraint solvers² [Magee 1995]. An architecture is generally considered to consist of components and their interactions. Architectural description languages allow the software engineer to formally abstract the architecture so as to reason about it. An ADL can give a global view of the system and when augmented with constraints, the validity of change (the reconfiguration of components) can potentially be evaluated at runtime [Magee 1995, Garlan 1997].

Sophisticated adaptive systems can be made from components that in turn are composed of sub-components. In our architecture a component consists of both the application logic, the architectural description of itself (i.e. the component structure) and a copy of the adaptation rules relevant to it as well as a lightweight adaptivity manager. A set of managers monitor and detect change, when required, plan that change. We group this functionality into a *session manager*. The session manager is fed information from monitors or *probes* (which aggregate raw monitor data for more lightweight processing). At each moment the current configuration operation is being monitored by the session manager that constantly checks constraints and, if broken, consults *switching rules* to decide how best to overcome the problem. When adaptivity is triggered the component architecture model allows an alternative execution plan to be designed. The session manager decides how to instantiate the alternative component architecture and passes its alternative over to the *adaptivity manager*. The adaptivity manager then carries out the unbinding and rebinding of components (establishing any glue necessary to achieve the binding). To do this it must ensure the instantiation adheres to transactional style properties. That is, the switch can be backed-off if something goes wrong.

```

Constraint: <rule_conditional>| <rule_selection>

<rule_conditional > ::= if <boolean_expression> then
    <action_sequence>
    [ else
      <action_sequence> ]
    end_rule;

<boolean_expression> ::= <resource> <standard_boolean_expression>
<action_sequence> ::= <action > [ { ";" <action > } ]
<action > ::= <instruction> "(" <parameter_sequence> ")"
<parameter_sequence> ::= <parameter> [ { ";" <parameter> } ]
<instruction> ::= BEST, SWITCH etc...
<resource> ::= PROC_UTIL or BANDWIDTH etc...
<standard_boolean_expression> ::= AND, OR, NOT, <, >, etc...

rule_selection ::= case resource of
    { set_expr ":" <action_sequence > }
    [ else <action_sequence > ]
    end case_rule;

```

Figure 3 BNF format for Adaptivity Rules in Patia

The architecture described here is essentially a closed-adaptivity model. That is, the actual component architecture model, the constraints and switching capabilities do not themselves adapt. In a highly adaptive system the component processing can migrate, as can the data. Typically, a component migrates to a part of the system to ensure a constraint is not broken. This can help with fault-tolerance also.

5 Patia Adaptation

Patia's system architecture follows the general adaptive component-based architecture introduced above: combining agent-based technology for component autonomy and migration. Adaptivity in Webserver

² Constraint solvers ensure that the logic of the different rules is correct before run-time.

architectures can be achieved at the inter-request level and the intra-request level and can help with not only improving server performance but also network performance. An example of inter-request level adaptivity would be where a client requests a given image, the version of the image sent is one which best suits the monitored bandwidth between the server and that client. Intra-request adaptivity could be a situation where the server is delivering some streaming media (e.g. audio), the codec of the stream is chosen to best suit the bandwidth, and if the bandwidth should change during mid delivery, then a new less bandwidth hungry codec is swapped in [McCann 2000]. These examples illustrate how adaptivity helps performance, however adaptivity can help with fault tolerance. For example if a monitor detects, through some form of trend analysis, that the number of requests are beginning to peak beyond a given threshold then it can dynamically spread its processing (e.g. to non-Webserver machines like a typing-pools' word processing computers), which can help with flash crowds.

To achieve flexibility in Patia *both* the data and the Webserver applications are componentised. This means that the components that compose a webpage can be distributed over many machines. This can provide the advantage of intra-request parallelism as well as fault-tolerance where replication is used. Each unit of data is known in Patia as an *Atom*. We define the Atom as the smallest web object that cannot be sub-divided³. Examples of this would be a video stream, an image, a navigation button, a text frame etc. Webpage Atoms are distributed over the nodes in the system⁴ and some may be replicated. The Atom follows the data structure of figure 2. For each Atom there is its content (data component) and some metadata (e.g. unique identifier, name, size etc). The Atom's version is implied through its name and associated rules (see example below) and each Atom has a set of constraints representing the rules as to how and when the atom is used.

The rules (constraints) associated with each Atom have the BNF format as illustrated in figure 3. The constraint can take one of two formats: *rule_conditional* or *rule_selection* respectively. An example of *rule_conditional* is:

```
if PROC_UTIL < 70 and BANDWIDTH < 60 then
    SWITCH (node1.webpageA.html)
end_rule;
```

which means that if the processor utilisation falls below 70 and the bandwidth between the client and Webserver falls to less than 60Kbps then switch from using the current version of the **webpage** Atom to webpageA.html which is currently located on node 1. Note that the version of **webpage** is denoted in its name and node number. We can have multiple versions of an Atom on a single node and each version might represent a differing codec for example.

An example of the *rule_selection* is:

```
case BANDWIDTH of
    < 600 : SWITCH (node1.atomA)
    < 200 : SWITCH (node1.atomB)
    other: SWITCH (node2.atomB)
end case_rule;
```

which means that if the bandwidth between the client and the Webserver falls to 600Kbps then the system chooses the **atomA** from node1. If the bandwidth continues to fall then **atomB** may be selected and this could be a lower quality version (e.g. graphic), which is stored on the same node. However if the bandwidth falls further then the atom **atomB**, is delivered but from **node2**, which could be a faster computer (for example).

³ We had considered that an Atom should be an object that it is *best not further sub-divided*, which means that the Atom can be a complete web page with text and graphics and where best meant that it would perform better to not sub-divide. However since modern browser technology separately requests a web pages' subcomponent we decided that this was not necessary.

⁴ We are not considering partitioning mechanisms in this paper, but are currently running experiments to examine data partitioning policies and their performance.

Of most interest is the *instruction* and the *resource* clauses in the BNF, see figure 3. The resource is the keyword representing the identity of the resource that is being monitored e.g. PROC_UTIL represents processor utilisation of the current machine and BANDWIDTH represents the bandwidth etc. (both are discussed in more detail in the next section. The example instructions we have listed here are *BEST* and *SWITCH*. As mentioned above SWITCH means that the processing must switch to another node or atom, and that it must do this in a safe manner. BEST indicates that we wish to select the best node that contains the atom that we are to deliver. BEST is calculated using a combination of maximum capacity of that node and its current utilisation (see section 6). An example usage of BEST is:

BEST (node1.atomA.html, node2.atomA.html)

where **atomA** is stored on both **node1** and **2** and the system chooses the node that will deliver the Atom the fastest.

6 Patia Systems Architecture

Like the data the Webserver code is also componentised. Figure 4 illustrates these components.

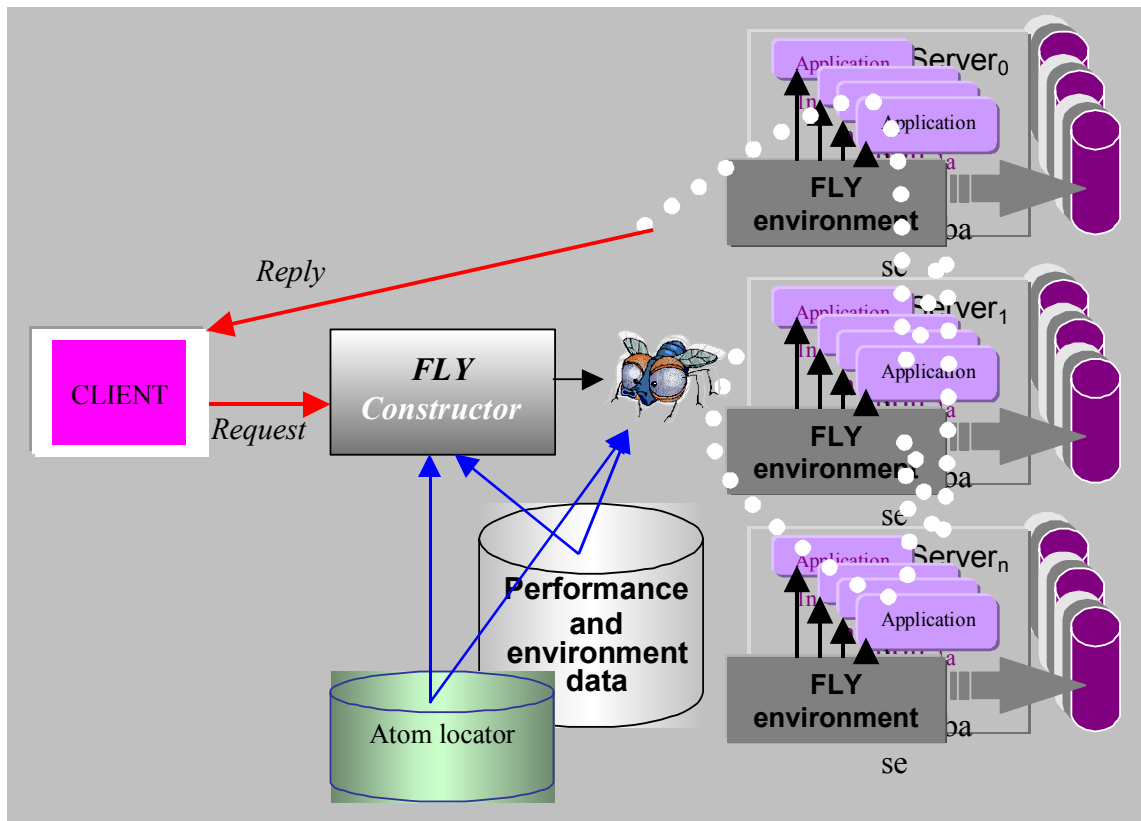


Figure 4 High-level Architecture of Patia Webserver showing FLY construction, environmental metadata, FLY migration

Essentially when a request comes into the system, it is received by a component which we call the *FLY Constructor*, that takes this request and instantiates the most appropriate *service-agent component* which we call the *FLY*⁵ to serve it based on the requested Atom, the rules associated with the latter and system state. The FLY, once constructed, becomes anonymous and can reside on any of the nodes in the system. Since, the FLY Constructor is effectively using a content-aware dispatching policy, it has to open a TCP connection with the client in order to inspect the HTTP contents of the request. Once, it has made its selection, it uses a TCP handoff mechanism to handoff the TCP connection to the selected FLY. This TCP handoff mechanism

⁵ The FLY is a mobile agent, which gets the components and has the intelligence to unbind and rebind the web object when it detects changes in the environment etc. However, the FLY is not named after ANT agent-based systems like that of [Dorigo] -- the name was given when the first author was thinking of a short but descriptive name and a FLY happened to pass the computer screen!

is implemented in the network stack of the operating system. By default, HTTP/1.1 [Fielding 1999], supports persistent connections, that is, the client can send other HTTP requests onto the same TCP connection. It can do that after receiving a full or partial response to its previous request, or before receiving any response at all. Subsequent requests onto the same TCP connection go directly to the FLY. Furthermore the FLY monitors the delivery of that Atom and if any of the Atom's constraints break, it is the job of the FLY to carry out the adaptation. The FLY constructor can be configured to run as a central Web switch or distributed onto the server nodes. In the latter case, a particular HTTP request will be received by all the FLY Constructor components, although only one of them will be allowed to act upon it.

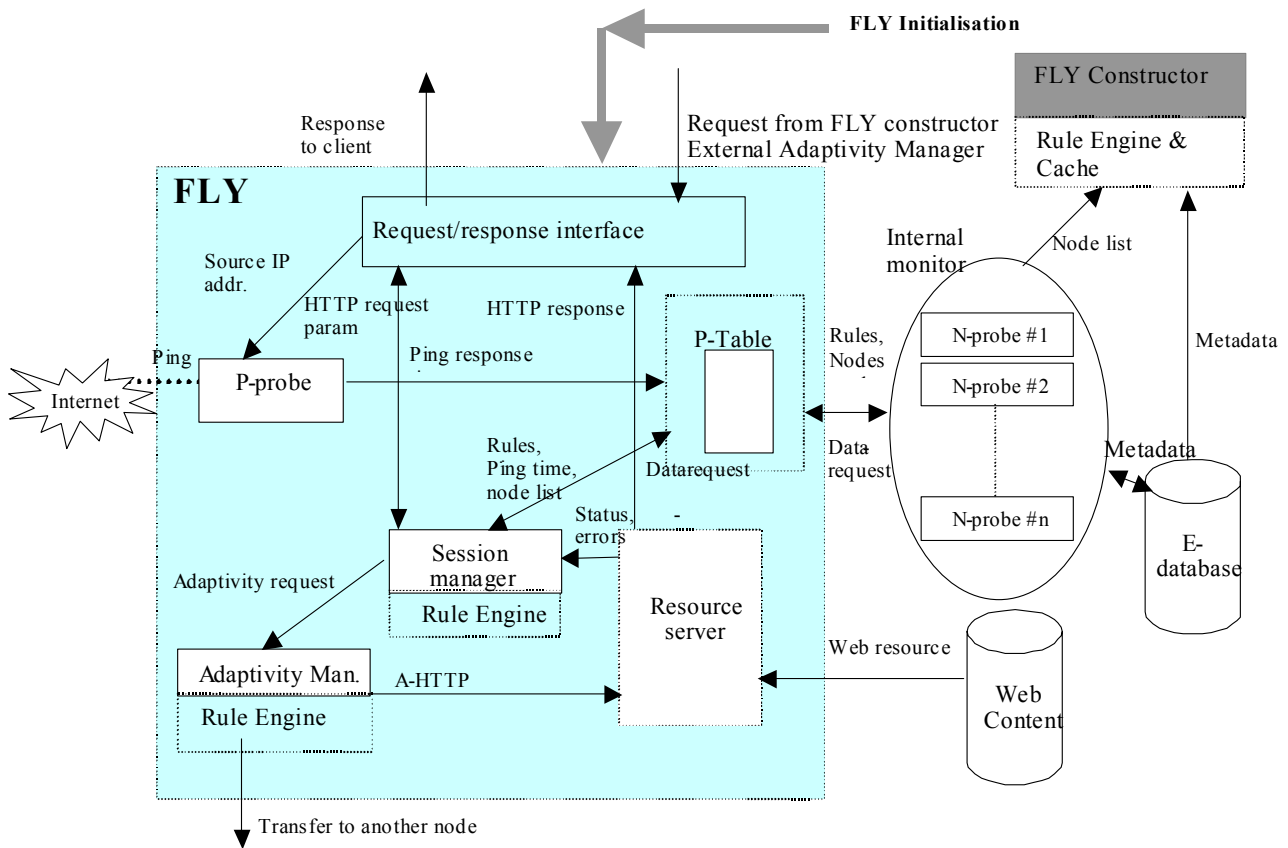


Figure 5 Overview of the Patia Webservice architecture

Figure 5 illustrates the FLY architecture. When the system receives a request from a client the *FLY Constructor* component initiates the FLY agent to serve that client. To do this it examines the current state of the system. Information about the system's state is maintained by a set of environment monitors; *internal*, *external* and *session* respectively.

An external monitor, namely the *P-probe* samples bandwidth between the client and the Webservice at regular intervals. We cannot feasibly send and measure a 'ping' request for every client that connects to the FLY. This will create undue network traffic and in an already loaded server, will load the server further. [Arlitt 2001] has shown that the distribution of requests from hosts over the Internet follows a Zipf distribution, i.e., with only a small number of very popular hosts generating the majority of requests. Hence, this invariant characteristic can be exploited in order to minimise network overhead and maximise efficiency of the 'ping'. Our approach to estimate bandwidth from a client is to map the client into one of several sets based on the popularity of the client. There will be small number of highly popular sets and a very large number of less popular ones. The distribution of the number of host IP addresses in the sets will follow an inverse-Zipf distribution, with the most popular set having only one host IP address and the least popular set having the highest number of host IP addresses. The choice of elements in each set will also be based on the geographical proximity between each other. Each set will be assigned a particular 'ping' response time (averaged over the hosts in the set), which is stored in the shared environment database (*e-database*).

The systems' state is also stored in the shared database (e-database). Likewise the *internal monitors* are a set of probes that monitor each node within the system and report performance statistics to the e-database. Both the external and internal monitors run constantly feeding the environment databases with up-to-date statistics. Furthermore, these monitors do not just report performance statistics but monitor trends using regression to predict very near future performance⁶. When the *FLY Constructor* component initiates a FLY, it essentially gathers the performance data relevant to the constraints described with that particular Atom from the environment database, and copies it to the FLY's local performance information cache (namely the P-table). Henceforth, the P-table will be updated regularly to reflect changes in the environment. Once the P-table is initially populated then the *FLY Constructor* instantiates the FLY on the node that best meets the initiation constraints essentially making the FLY autonomous from this point. The TCP connection is then handed-off and the FLY's session manager then takes responsibility for the client-server connection from then onwards. One of the characteristics of the PATIA system is that it tries to exploit various invariant characteristics that have been observed in Webserver workloads. An astute exploitation of these characteristics make the components of the system as lightweight as possible.

The *resource server* is basically the component in the FLY that retrieves the Atom from the disk and serves it to the client. It has the same functionality as that of the content-handler in Webservers such as Apache [Stein 1999]. However the core to the Patia Webserver FLY is the *session manager*, which supervises the run-time delivery of the Atom to the client, periodically checking that constraints have not been broken by examining the P-table data. When a constraint has been broken the session manager decides the best course of action based on the Atom's *action* associated with the constraints and hands over the adaptation to an *Adaptivity Manager* (see figure 5). As described in section 4, the Adaptivity Manager unbinds the current session and rebinds the new session to implement the adaptivity in a safe manner; i.e. the adaptation either happens in entirety or not at all.

Switching functionality is defined in the rules through the *instructions* and *resources* (see figure 3). The *resource* is simply a representation of a defined resource. The environment performance data must have a field of the same name in both the shared database and the P-table copy. This is so that the session manager can compare the current value for the resource (e.g. **PROC_UTIL** for a given processor) with the threshold defined in the Boolean expression (e.g. **> 90%**). The *instruction* is more complex in that it represents *what* is to happen. For example the instruction **BEST** means choose the best processor, the session manager gets the identity of the best processor from the P-table. The identity of the best processor is the one at the top of a ranked list of processors and their performance. This list takes the form:

Node = <node_id, Current_Node_Rank>

The node rank list is the current processor capacity stored in rank order and is thus maintained by the internal monitors and updated periodically (which may involve reordering). Node rank is calculated as a function of the computer's maximum capacity against its current utilisation using the following formula:

$$\mathbf{Current_Node_Rank} = (p * \mathbf{processor_rank} + m * \mathbf{memory_rank})$$

where:

$$\mathbf{Processor_rank} = (\mathbf{max_cap} * (100 - \% \mathbf{utilisation}))$$

$$\mathbf{Memory_rank} = (\mathbf{max_cap} * (100 - \% \mathbf{utilisation}))$$

and

p = tuning constant for relevance of processor (0-1)

m = tuning constant for relevance of memory (0-1)

max_cap is the maximum capacity of either the processor or memory

%utilisation is the utilisation of that hardware component as calculated by the probes

This formula was used in initial experiments and showed that it represented machine performance quite well⁷.

⁶ This technique was used successfully in the Kendra project which inspired the Patia project [McCann 2000]

⁷ However more work on the tuning constants and their effects need to be carried out.

When the session manager detects that the FLY is not performing to some minimum threshold or it predicts that the incoming requests are rising to a predefined threshold, the system can grow. The Webserver will have a number of servers/machines dedicated to it for normal day-to-day processing. On detection of an increased load the system will then start to spread out to other machines that have been allocated for this purpose. Machines that are barely used (e.g. machines in admin used for word-processing) are allocated as auxiliary nodes or overspill nodes. The system begins to gracefully expand by stealing cycles from other machines. To do this it needs to select some parts of the website to be replicated on the overspill nodes (there may be replications already set up there i.e. the overspill machine is already a mirror of some of the data). This is analogous to power plant operation. To illustrate this operation we present a small example of where BEST and SWITCH are operating together:

```
If PROC_UTIL > 90% then  
SWITCH  
(BEST ( node1.Page1.html,  
node2.Page1.html));
```

This is a constraint that could be used for fault tolerance in, say for example, the case of flash crowds. Here the session manager receives processor utilisation data from the respective environment monitor and when it detects that the utilisation rises above 90% the FLY is required to run on a different node (either node1 or node2 depending on which is the least loaded). The different node could be an under-utilised machine in the typing pool, which contains a replica of *Page1.html*. In this example the action SWITCH indicates to the session manager that not only should the Adaptivity Manager save the data state, but also the processing state, as it is this that is about to migrate. That is, essentially the whole FLY is mobile therefore making the Adaptivity Manager's task more complex.

7 Related work

Considering the growth of web usage there has been relatively little research on the actual Webserver engine architecture, yet alone infrastructures that support adaptivity. For example, systems using the Internet are often exposed to external stimuli such as changes in traffic load and internal stimuli such as changes in available resources. These conditions make such systems good candidates to benefit from some form of adaptivity. In their survey on dispatching policies, [Cardellini 2002] pointed that the kinds of open adaptation policies found in work by [Shivaratri 1992] were not being used for request dispatching in distributed Webserver systems. Our work uses similar policies to Shivaratri et al, however we are not considering the situation where our policies themselves self-adapt at this moment in time (thus Patia is currently a closed adaptive system), although we have designed the policy components such that future enhancements would be relatively straightforward to include. For example we could make the adaptivity more open if the system were to calculate the best p and m tuning constants in the *Current_Node_Rank* formula (in section 6) and automatically change these constants as the system evolves and its demands or environment change. This would mean that the emphasis on the memory or processor would change as the type of processing (memory or processor bound) changes.

Most of the Webserver adaptivity work focuses on either QoS or tailoring web documents for adaptive presentation [Chen, Ma 2000]. For example Chen et al provide static adaptivity to adapt content to differing output devices based on individual user profiles. Like Patia they decompose a web document into objects and composite objects (similar to our Atoms). They then define a function-based object model representing the intra-document object relationships, where each object is atomic or composite and has a set of semantic properties. One example property would be the extent to which the object is for decoration purposes e.g. a button icon, and where it is not necessary to use that button (e.g. Mobile phone) it is not delivered. However their focus is not on performance nor do they support dynamically generated documents, which is one of the advantages of our architecture. On the other hand, to add this functionality to Patia would simply mean creating a constraint on the button Atom that says that we either ignore or send an alternative Atom in its place once the interface e.g. mobile phone was detected. Therefore a further monitor is required to monitor the type of device being used by the client. Patia's component-based structure should make that task relatively simple. Likewise, [Perkowitz et al. 2000] are exploring adaptivity whereby the system changes its presentation based on user access patterns (which have been mined from Webserver logs). Though an interesting approach, the capabilities of this system depend heavily on how predictive the patterns in the log files are for future accesses – for example access trends sometimes are not repeated. Our system is observing

such patterns ‘live’ and adapting as patterns change. Obviously, there is a limit to the extent to which the system can accurately monitor the ‘live’ system and to which it can safely adapt in a timely fashion. This is the focus of the next stage of the project.

Also similar to Patia is the FLEX web system [Cherkasova 2000], which uses adaptive load balancing for ‘content-based’ routing (using DNS) to direct requests to a node in the cluster of machines. However unlike Patia which partitions data at the object (or Atomic) level to get more parallelism out of the cluster, they partition data at the granularity of *site* level. This means that they can use DNS to map requests to IP addresses, which has the added advantage that there is no centralised dispatcher style unit. In Patia the relatively central point is the *FLY constructor*. However by having a *FLY constructor* per computing node in the system, we also maintain a degree of distribution. Additionally, once a FLY is instantiated for a given client it assumes dispatcher responsibilities and as there are many FLYs in the system, we have a lower number of central points of failure or bottlenecks. Further the FLEX system is limited to sites that fit on a single machine.

Much of the quality of service research (QoS) adaptivity work has focused on the policies and the mechanisms to implement those policies (usually extensions to an established Webserver e.g. [Vasiliou 2001]). Typically the users are categorised into classes and the scheduling or resource management function decides the level of service that that class will receive. The adaptivity is relatively restricting in that classes of users are given classes of service. In Patia we could also implement this by having similar policies implemented in our constraints, where for each Atom the constraint indicates the version to be delivered to a particular client –class. Again environment data would be stored about the client to indicate what class it belonged to.

Scale is a significant issue in Webserver design. The key difference between the architectures discussed in section 3 i.e. Web cluster, virtual Web cluster and Distributed Web systems, is component(s) that capture the client request. To reiterate, Web cluster systems have a single Dispatcher/Web switch component, which takes the request and farms it to the processors in the cluster. In the virtual Web cluster all nodes receive all requests and only one replies to the client and in the distributed Web system the DNS carries out the distribution. The initial implementation of the Patia system will have a single FLY constructor and therefore it closely resembles the Web cluster style of architecture. As this would potentially cause bottlenecks, compromising the scalability of our architecture, future versions of Patia will have FLY constructors at each node, thus it will resemble the virtual cluster architectures. However this requires amending the lower layers of the system (i.e. the operating system) and the hardware set-up.

8 Conclusion

To avoid flash crowding and provide well performing Webserver systems large companies typically employ a Webserver with a capacity of which three quarters is not used in day-to-day processing. They purchase this large machine(s) so that the system can cope with peak requests. As seen recently, many web sites are simply unable to cope with the sudden increased demand for their servers. In this paper we present an adaptive Webserver architecture, Patia, that not only adapts the content and processing of the content to fit observed performance, but it expands itself when it detects that its performance is becoming poor. To this end the primary objectives of this project is to build a Webserver which:

- must not have a bottleneck (i.e. for both performance and reliability – fault-tolerance)
- must be extensible (i.e. adapt to future technologies)
- must be adaptive to changing environmental circumstances
- must show good performance, utilisation and scalability
- must try and avoid forcing people to use different client software or application servers (i.e. database backend servers and browsers are not specialised).

We believe the architecture presented in this paper will fulfil our aims. After successful initial experiments with client access patterns, performance monitoring and predictors, and constraint design we are now in the process of building Patia. We are also building a Webserver benchmark that will better suit modern web processing, as many in existence were too simplistic i.e. they mimicked static document processing only. Once nearing completion we hope to experiment with the FLY’s performance and amount of knowledge and the representation of that knowledge that it requires to fulfil our objectives.

9 Acknowledgements

This work has been sponsored by the EPSRC of the United Kingdom; grant number GRN38008.

10 References

- Arlitt M, Krishnamurthy D, Rolia J; Characterizing the scalability of a large Web-based shopping system, ACM Transactions on Internet Technology Vol. 1 No. 1 (Aug.) 2001, pp 44-69
- Barford P, Crovella M; Critical path analysis of TCP transactions, IEEE/ACM Transactions on Networking Vol. 9 No. 3 (Jun) 2001, pp 238-248
- Brisco T; DNS support for load balancing RFC 1794 (Apr) 1995
- Cardellini V, Casalicchio E, Colajanni M, Yu P S; The state of the art in locally distributed Web-server systems, ACM Computing Surveys Vol. 34 No. 2 (Jun) 2002, pp 263-311
- Chen J, Zhou B, Shi J., Zhang H., Wu Q.; 'Function-based object Model Towards Website Adaptation', Microsoft Research China.
- Cherkasova L., Ponnkanti S.: Optimizing "Content-Aware" Load Balancing Strategy for Shared Web Hosting Service. In Proceedings of the Eighth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'2000).
- Dorigo M., Maniezzo V. & Coloni A.; The Ant System: Optimization by a Colony of Cooperating Agents. *IEEE Transactions on Systems, Man, and Cybernetics-Part B*, 26(1), pp 29-41, 1996
- Fielding R, Irvine U C, Mogul J, Frystyk H, Masinter L, Leach P, Berners-Lee T 1999; Hypertext Transfer Protocol – HTTP/1.1 RFC 2616
- Garlan D., Monroe R. T., Wile D.; Acme: An Architecture Description Interchange Language, In Proc. of CASCON '97, November 1997.
- Gilder G; 'Fiber keeps its promise: Get ready. Bandwidth will triple each year for the next 25', Forbes (Apr) 1997
- Huitema C; Network vs server issues in end-to-end performance. Keynote speech at Performance and Architecture of Web servers Workshop (Web page) 2000 http://kkant.cwebhost.com/PAWS2000/huitema_keynote.ppt
- Jung J, Krishnamurthy B, Rabinovich M; Flash crowds and denial of service attacks: characterization and implications for CDNs and Web sites, Proceedings of the Eleventh International World Wide Web Conference WWW2002 (May) 2002
- Ma W., Bedner I., Chang G., Kuchinsky A., Zhang H.; A framework for adaptive content delivery in heterogeneous network environments Proc MMCN2000 (SPIE vol. 3969), 2000 San Jose, USA, pp 86-100
- Magee J., Dulay N., Eisenbach S. Kramer J., Specifying Distributed Software Architectures. In Proc. of the Fifth European Software Engineering Conference, Barcelona, 1995.
- McCann J.A., Howlett P., Crane J.S.; Kendra: Adaptive Internet System, Journal of Systems and Software, Elsevier Science, Volume 55, Issue 1, 5 November 2000, pp 3-17.
- Oreizy P., M. Gorlick M., Taylor R. N., Heimbigner D., Johnson G., Medvidovic N., Quilici A., Rosenblum D. S., Wolf A. L.; An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, vol. 14, no. 3, May/June 1999, pp 54-62.
- Perkowitz M, Etzioni O; Towards adaptive Web sites: conceptual framework and case study, Elsevier Artificial Intelligence, 2000 (118), pp 245-275.
- Shivaratri N G, Krueger P, Singhal M; Load distributing for locally distributed systems IEEE Computer (Dec) 1992, pp 33-44
- Stein L., MacEachern D.; Writing Apache Modules with Perl and C, O'Reilly and Associates (Mar) 1999
- Vasiliou N., Lutfiyya H.; Managing a Differentiated Quality of Service in a World Wide Web Server, *Integrated Network Management Volume VII*, May 2001, pp 309-312.
- Veloso E, Almeida V, Meira W, Bestavros A, Jin S; A hierarchical characterization of a live streaming media workload, Technical Report BUCS-TR-2002-014 Boston University (May) 2002.