The Patia Autonomic Webserver: Feasibility Experimentation

Julie A McCann, Gawesh Jawaheer
Department of Computing
Imperial College London
SW7 2BZ, UK
{jamm, gawesh}@doc.ic.ac.uk

Abstract

This paper introduces the Patia Autonomic webserver, which has been designed to be self-monitoring and adaptive to not only improve webserver performance but robustness in terms of helping with flash crowd situations. This paper describes the Patia architecture, highlighting how a traditional computing system can be redesigned to become reflective and adaptive. In doing so we examine and report initial performance results concerning the extra functionality required of autonomic software, highlighting the advantages and problems of adding autonomicity.

1 Introduction

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 [2]. Further, there have been claims that 40% of Web delays are in fact due to webservers [5]. Consequently, webservers are under pressure to perform and when they fail to perform, end-users experience increased access latency. Performance failure has been further acerbated by the phenomenon known as *flash crowds* -- a situation where very large numbers of users simultaneously access a Website [6]. Flash crowding is becoming more prevalent and technologies such as Web caches and CDNs (Content Distribution Networks), previously employed to improve webserver performance, have been proven to not be very effective in flash crowd situations [6].

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-autonomic decentralized system. The Patia system consists of essentially webserver agents (each called a *FLY*) that carryout 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.

The Patia meta-architecture is introduced in section 2 with some initial performance study results presented in section 3. Finally we give a brief overview of related work in section 4 and conclude in section 5.

2 Patia webserver Meta-Architecture

Patia's system architecture follows the general adaptive component-based architecture combining agent-based technology for component autonomy and migration. Here both the data and system's functionality is componentised.

2.1 Patia Data Componentisation

Adaptivity in webserver 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 mid delivery, then a less bandwidth hungry codec is swapped in, as described in [8]. These examples illustrate how adaptivity helps performance, however adaptivity can also 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 using migration to help with events like flash crowds.

In Patia data is partitioned and distributed over a number of dedicated and non-dedicated Patia webservers. Each unit of data is known as an *Atom*. We define the Atom as the smallest web object that is best to not be sub-divided[1]. 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 system and some may be replicated. 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 what conditions under which the Atom is used.

2.2 Patia Component Architecture

Like data, the webserver code is also componentised. Essentially when a request comes into the system, it is received by a component, which we call the *FLY Constructor* that takes the request and instantiates the most appropriate *service-agent component,* which we call the *FLY*[2]*,* to serve it based on the requested Atom, the rules associated with the latter and current system state. That is, 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, which is maintained by a set of environment monitors; *internal, external and session* respectively. This data is stored in an *environmental database* (e-database).

An external monitor, namely the *P-probe* samples bandwidth between the client and the webserver at regular intervals and stores this in the 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 database with up-to-date statistics as well as monitoring trends using regression analysis to predict very near future performance[3]. When the *FLY Constructor* component initiates a FLY, it essentially gathers the performance data relevant to the rules described with that particular Atom from the e-database, and copies it to the FLY's local performance information cache (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 handed-off and the FLY's session manager then takes responsibility for the client-server connection from then onwards. This is described in more detail in [9].

The *resource server* is basically the component in the FLY that retrieves the Atom from the disk and serves it to the client. The core FLY component is the *session manager,* which supervises the run-time delivery of the Atom to the client, periodically checking that rules have not been broken by examining the P-table data. When a rule has been broken, the session manager decides the best course of action based on the Atom's *action* associated with the condition and hands over the adaptation to an *Adaptivity Manager*, which unbinds the current session and rebinds the new session to implement the adaptivity. The condition/action rules associated with each Atom have the BNF format described in [9].

Switching functionality is defined in the rules through the *instructions* and *resources*. The *resource* is simply a representation of a predefined system resource (e.g. PROC_UTIL processor utilisation for a given processor). The *instruction* 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 ranked list of processors. 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. The different node could be an under-utilised machine in the typing pool, which contains a replica. An 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.

3 Feasibility Study Performance results

The primary objectives of this project is to build an autonomic webserver which:

- must not have a bottleneck (i.e. for both performance and reliability – fault-tolerance)

[1] 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.

[2] 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.

[3] This technique was used successfully in the Kendra project which inspired the Patia project [8]

- must be extensible (i.e. adapt to future technologies)
- must be adaptive to changing environmental circumstances
- must show acceptable performance, utilisation and scalability.

Many of the components of Patia have been built and we are beginning to carry out tests to measure its effectiveness given the objectives listed above. The major design trade-offs have been concerned with the componentisation of both the code and the data and how that affects performance. Further, the Patia meta-architecture is augmented with monitors and condition/action rules that drive adaptation and this comes with added costs. Therefore this section presents the results of two preliminary studies focusing on both the costs of monitoring and decision-making in Patia and the costs of designing all web documents as atoms.

This initial study is used to ascertain whether autonomicity is feasible or useful for webserver systems. In particular we focus on the costs of data partitioning and the addition of self-monitoring.

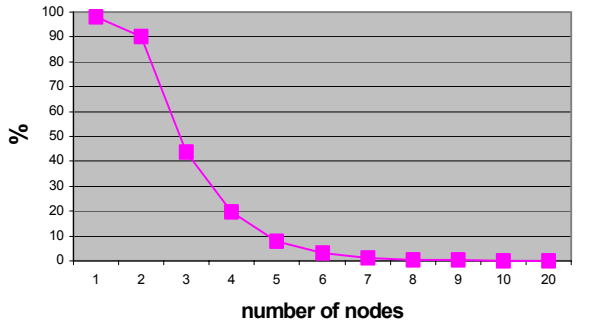

Figure 1. %queueing time - full replication

3.1 Is Data Componentisation and Placement Feasible?

A data placement study was carried out to estimate the potential costs/benefits of distribution and replication of web documents over the Patia server. The web data simulated was that of the university's own webpages and care was taken to ensure generality by checking that this data matched general characteristics and distribution patterns of non-university websites found in [1].

Queueing delay is important; as it is the time a request is waiting to be served on a node in Patia. For a single node we found that the queueing delay was 98%. The first experiment examined replication and scale and found that by fully replicating documents on as little as four nodes we see a significant improvement in performance i.e. queueing delays to the servers reduced significantly (see Figure 1). Note that when the data was fully replicated the average queueing delay is 1%. However when the data is then fully partitioned over 10 nodes (randomly), the queueing delay increased to 11.4%. If one of the 10 nodes is selected as a backup or hot node (i.e. data is randomly distributed on nodes 2-10 and node one holds a replica) this improves the queueing delay by just under 2%.

We also looked at the numbers of replications of a given document (randomly placed on the 10 nodes). Here we found again that only four replicas are necessary to achieve a significant improvement in performance (see figure 2). However the cost of 100% replication of all documents required just under 20 times the storage requirements.

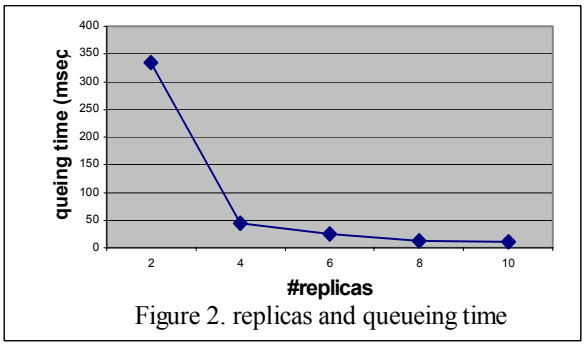

Figure 2. replicas and queueing time

The main reason that 4 replicas are enough to achieve significant reduction in queueing delays is because typically web access requests are heavily tailed and follow a Zipf like distribution. This means that a very small percentage of files (e.g. 10%) are accessed by far the most (e.g. 90% of the time). To estimate how popularity affects performance we carried out an experiment showing that the top 63% of requests go to 12% of the files only and by fully replicating these on all 10 nodes in the system we can achieve an improvement in queueing delay of 62%.

These experiments assumed that the level of granularity was that of a web page. However for added flexibility Patia sub-divides pages into their atomic parts. This experiment placed a given type of atom to a specialized server. That is there was image, text, PDF, zip, html servers etc. This caused the average queueing delay for all request to increase to 38%, which is by far the worst result. This was due to two servers' usage being heavily skewed (image/html). Therefore obviously distributing these popular atoms would overcome that performance problem. Consequently, we believe that the cost of decomposition-reconstitution can therefore be overcome by replicating the top 10% most popular atoms over at least 4 nodes with a possible overall performance improvement of around 25% (62%-38%) at an estimated cost of only 5 times the storage space consumed (for this

particular suite of data). Further experimentation will confirm this.

3.2 Is Self-monitoring feasible?

Recall, monitors feed a database of environment metadata describing the Patia architecture (both code and data) and its current state. A second suite of experiments was carried out to find the costs associated with these system monitors. Four PIII 1.4 GHz PCs were used to send HTTP requests over a switched 100 MBbps Ethernet to a system of three heterogeneous Web server nodes each running Apache. All machines were running under Linux. An internal monitor ran on each server node, collecting performance statistics and transmitting the latter (using UDP transport protocol) every 5 seconds to the environment monitoring sub-system. A utility called *sysstat* [12] was used to collect the CPU usage, memory usage and interface traffic on each server node. The experiments consisted in randomly sending HTTP requests to the server nodes with and without monitoring processes running. The results are shown in Table 1.

System information	Experiment[4]	Server#1 PIII 1.4 GHz 256 MB RAM	Server#2 PIII 650 MHz, 128 MB RAM	Server#3 P Pro 200MHz, 256 MB RAM
% Cpu usage	Rnd No Mon	28.8	17.0	7.2
	Rnd Mon	29.2	17.0	7.4
Memory usage	Rnd No Mon	19.0	89.4	34.5
	Rnd Mon	60.0	97.6	98.3
Tx Bytes/sec	Rnd No Mon	6313698	714078	441330
	Rnd Mon	6282343	704817	429611
Throughput (req. per sec)	Rnd No Mon	384.9	192.3	25.6
	Rnd Mon	381.1	190.8	25.4

Table 1. Results of experiments for costs of monitors

As shown in Table 1, the monitors marginally increased the server node CPU usage. However, they did consume a substantial amount of memory and network bandwidth. The latter is reflected by an overall decrease in server throughput. The relatively high memory usage on the servers shows the system to be memory bound. The decrease in server throughput caused by the monitors is due to the consumption of bandwidth by the monitor traffic. This can be mitigated by decreasing the frequency at which performance statistics are transmitted at the expense of having less up-to-date state information or by decoupling the performance statistics traffic from the HTTP traffic. The latter can be done using a separate network to link the internal monitors to the environment monitoring subsystem. However, this adds to the cost of the hardware as it requires separate network cards and thus reduces the flexibility of the system. Further experiments are being carried out to find the optimum performance statistics transmission frequency and the performance of the session manager.

[4] Rnd No Mon = Random server selection without any monitoring processes running
Rnd Mon = Random server selection with monitoring processes running

4 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. However, most of the webserver adaptivity work focuses on either QoS or tailoring web documents for adaptive presentation [3, 7, 10]. That is, their focus is on adapting content to differing output devices based on individual user profiles and is not on performance or fault tolerance.

Also similar to Patia is the FLEX web system [4], which uses adaptive load balancing for 'content-based' routing (using DNS). 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. 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. [11]). Typically the users are categorised into classes and the scheduling or resource management function decides the level of service that that class will receive. Here adaptivity is relatively restricting in that classes of users are given classes of service. Alternatively Patia's adaptivity is tailored to each request

5 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. A more cost effective solution is to build a webserver that can grow and shrink based on demand. To do this the webserver must be reflective and self-adapting. This autonomicity requires that traditional systems software has added monitoring, intelligence and ability to reconfigure on demand.

However, flexibility of an autonomic system comes at a cost.

In this paper we present an adaptive webserver architecture, Patia, designed to adapt the content and processing of the content on demand to ensure a given level of QoS. Patia is interesting in that the addition of self-awareness is not only to help with its ability to be more robust in situations such as flash crowds etc but to generally improve performance. Given the cost of autonomicity this is a tall order. In this paper we briefly examine extra architectural components that are required in reflective flexible systems to see if it is feasible to make a distributed webserver autonomic. These show that there is indeed a cost in partitioning web pages into atoms and reconstructing them, and that self-monitoring (alone, without intelligence even being measured) is costly in terms of bandwidth and memory consumption. However we have shown that some of the costs can be balanced by for example replicating the most popular atoms over the nodes and using specialist-monitoring communications respectively.

Further work on Patia will examine how effective and at what speed the system will learn to adapt and how much this costs in terms of resources.

6 Acknowledgements

This work has been sponsored by the EPSRC of the United Kingdom; grant number GRN38008. Thanks to Linxue Sun who designed the performance benchmarking suite and Adamantia Alexandraki, an MSc student, who helped us with some of the data placement aspects.

7 References

1. 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
2. Barford P, Crovella M; Critical path analysis of TCP transactions, IEEE/ACM Transactions on Networking Vol. 9 No. 3 (Jun) 2001, pp 238-248
3. Chen J, Zhou B, Shi J., Zhang H., Wu Q.; Function-based object Model Towards Website Adaptation', Microsoft Research China.
4. Cherkasova L., Ponnekanti 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).
5. 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.ccwebhost.com/PAWS2000/huitema_keynote.ppt
6. 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
7. 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
8. 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.
9. McCann J.A., Jawaheer G., Sun L., Patia: Adaptive Distributed Webserver (a Position Paper), proc ISADS 2003 - The Sixth International Symposium on Autonomous Decentralized Systems, April, 2003
10. Perkowitz M, Etzioni O; Towards adaptive Web sites: conceptual framework and case study, Elsevier Artificial Intelligence, 2000 (118), pp 245-275.
11. 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.
12. sysstat http://freshmeat.net/projects/sysstat/