Experience using a Coordination-based Architecture for Adaptive Web Content Provision*

Lindsay Bradford, Stephen Milliner, and Marlon Dumas

Centre for Information Technology Innovation
Queensland University of Technology, Australia
{l.bradford, s.milliner, m.dumas}@qut.edu.au

Abstract. There are many ways of achieving scalable dynamic web content. In previous work we have focused on dynamic content degradation using a standard architecture and a design-time "Just In Case" methodology. In this paper, we address certain shortcomings witnessed in our previous work by using an alternate coordination based architecture, which has interesting applicability to run-time web server adaptation. We first establish the viability of using this architecture for high-volume dynamic web content generation. In doing so, we establish it's ability to maintain high throughput in overload conditions. Then we go on to show how we used the architecture to achieve a "Just in Time" adaptation to achieve dynamic web content degradation in a running web application server.

1 Introduction

Researchers have recently discussed the need for adaptable web-provision technologies, particularly in terms of architectures that cater to varying degree of adaptability [1]. This focus on architectures is perhaps due to a growing realisation that the architecture chosen is one of the key factors to successful system deployment [2]. If we want an adaptable system, its architecture must first support adaptation.

Architectures that offer a coordinated model of interaction (for example, JavaSpaces [3]) provide certain characteristics that are attractive to achieving adaptation. These architectures separate components from how the components interact via some form of coordination, and in turn, should make run-time component replacement and changes to component interaction easier to accomplish. Such architectures introduce decoupling across space (allowing distributed behaviour), time (allowing asynchronous communications) and interface (allowing easier replacement and interaction of components), in turn, allow a wide range of choice in the types of adaptation that can be implemented.

Many modern web-centric architectures take a somewhat coarse-grained/static approach where components and their interactions are fixed at design time. As a

* This research is funded in part by SAP Research Centre, Brisbane.

result, developers can be locked into limited types of adaptability, and at worst, could be forced to construct adaptation techniques at design time along with the core deliverables and in ways that may not be appropriate. Adaptive systems that take this "Just In Case" (JIC) approach run the risk of not being able to adapt key components to environmental changes. Even if the components to adapt are correctly identified and alterable, the system designer needs to also successfully guess the right type and amount of adaptation to apply when designing their systems. That is, they must have complete a priori knowledge of all possible situations.

JIC adaptation techniques are highly predictive, whereas "Just in Time" (JIT) adaptation techniques are highly reactive, even to the point of being manually constructed for specific short-lived circumstances. More formally, we describe JIT adaptation as the introduction of behavioural change into a system once some event has occurred that requires such change, and that the adaptation made is targeted specifically to this event. By breaking HTTP content delivery of a web application server into a number of components interacting via architectural coordination, the degree of predictiveness required for adaptation can be minimised, or even removed, by altering just those components and interactions that need changing at run-time.

In this paper, we aim to establish whether our coordination architecture, with its optimisations for localised coordination of network deliverable components, is capable of web content delivery with sufficient latency and throughput make it viable for use as a web application server. We also aim to establish whether the architecture can significantly and rapidly adapt via a JIT delivery of new behaviour, even under extreme load conditions.

Our focus in web application adaptation is on achieving scalability at a single web application server via behavioural change. In contrast, other popular techniques for supplying adequate web scalability involve the over-provision of a service-provider's computing resources, typically by supplying several duplicate machines that share requests through some load balancer. This solution is not only costly, but also requires increased configuration and administration effort.

In previous work [4], we constructed a dynamic web content degradation system based on elapsed-time measured at the server. Elapsed-time is a critical factor of *user-perceived quality of service* on the Web [5] [6] [7], and in human-computer interactions more generally [8]. Aspects such as layout, graphics and such are far less likely to effect user-perceived quality of service, suggesting that degrading such aspects for better elapsed-time responses is an area of web adaptability deserving further investigation.

We used a mainstream web application server, namely Tomcat[1], to supply several approaches to generating web content for a given URI. An elapsed-time based algorithm was used to decide when to degrade the web content delivered by choosing between these approaches. For example, a baseline approach might be a complete web-page portal collating results from several other web-pages. Under load, the base approach generating this complete portal might be replaced

[1] http://jakarta.apache.org/tomcat/

by a lightweight approach that returns only those portlet images that the server has cached. The algorithm and alternate approaches required are an example of JIC web adaptation, and is typical of the extra pre-emptive overhead inherent in such schemes.

What is uncertain in using coordination architectures such as ours for web content delivery is firstly, how they would behave under load conditions, and secondly, how they should be used to achieve high-performance web content delivery. To that end, we offer two contributions in this paper, being i) we establish that our coordination architecture can be used to serve high-demand dynamic web content fast enough to be considered viable and show that it exhibits good throughput characteristics under load, and ii) we present a method for exploiting our architecture to seamlessly adapt to very different behavioural patterns in a JIT fashion. To illustrate out second contribution, we introduce automated content degradation into an overloaded web application server at run-time.

Section 2 discusses the design of the system by first describing the base architecture (Sec. 2.1), then the adaptation of the base architecture into a web application server at run-time (Sec. 2.2), and finally the adaptation this web application server into one capable of automated content degradation (Sec. 2.3). Section 2.4 discusses some of the lessons learned with early attempts. Section 3 establishes the viability of our design via experiments. Section 4 discusses related work and section 5 concludes the paper.

2 Design

In Section 2.1, we discuss our coordination architecture and the localised optimisations that make this architecture a viable candidate for delivering both high load web service provision and for flexible service adaptation. In Section 2.2, we discuss how we used this architecture to deliver our JIT adaptable web application server.

2.1 Service provision with ActiveObjectSpaces

ActiveObjectSpaces (AOS) is a distributed coordination middleware drawing on three main predecessors: Blackboard Architectures (for example, Hearsay [9]), Linda [10] and Sun's JavaSpaces [3]. It supports three main primitives `read()`, `write()`, and `take()`. The AOS has a extended notification API compared to JavaSpaces (notification can trigger on arrival, existence and on deletion). Like JavaSpaces, coordination is achieved via notification templates describing essential aspects of objects on the Object Space that clients wish to interact with. Perhaps the most novel aspect of the AOS is the notion of *Active Objects*.

Active Objects are simply AOS-aware objects that execute in their own thread of control within the AOS middleware itself. By sending the appropriate active objects to a running AOS, and incorporating a coordination protocol for their interaction, we can dynamically deploy the components of a stand-alone server. By constructing a service via such a collection of loosely coupled co-ordinated components we can achieve adaptability at the level of granualarity required for a single server.

An AOS server initially knows nothing about receiving HTTP requests or delivering web content. Our HTTP module (section 2.2) is delivered to the AOS which the AOS then executes. Once the module is running, the AOS has dynamically been converted into a web application server. Later, when the server exhibits poor response times, a content degradation module (section 2.3) is delivered to the AOS, converting the AOS into a server capable of selecting degraded content based on elapsed server response-times.

2.2 Dynamic Web Content Delivery via ActiveObjectSpaces

A number of active objects and object class definitions are delivered to the AOS via the HTTP module (see figure 1). The class definitions and active objects are removed from the object space and installed for use by the AOS. For most of the active objects, this simply means informing the server to execute a callback method on them when data objects matching notification templates they specify are delivered to the object space. Later, when the AOS starts accepting client requests, the class definitions delivered via the module will allow the active objects to create and manipulate objects needed for HTTP content delivery.

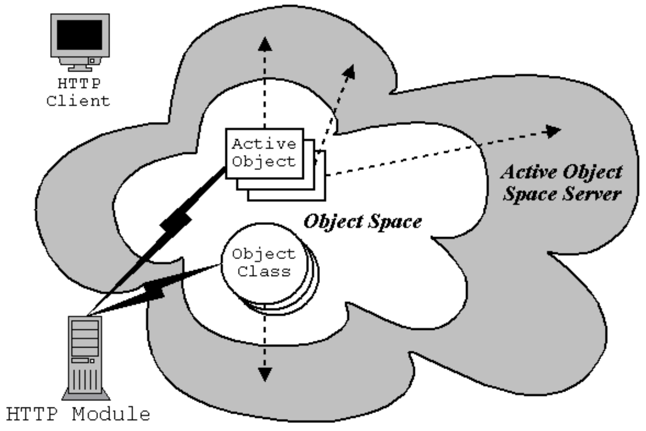

Fig. 1. Turning the AOS into a HTTP Server

Most of the content delivered in the HTTP module is generic HTTP handling code, and much like 'out of the box' application servers, is expected to remain unchanged over time. However, we do have the option of making quite radical changes even at the HTTP layer if required. The content degradation module discussed later, alters behaviour, and is roughly analogous to delivering a new WAR file to a running Tomcat application server. What makes our approach novel with respect to more contemporary application servers is that both the

HTTP generic behaviour as well as content delivery behaviour can be adapted, and at a very fine level of granularity. By using the space as our core state repository, we can also draw on looser, more interactive ways of generating web content that are not possible with call and return style architectures.

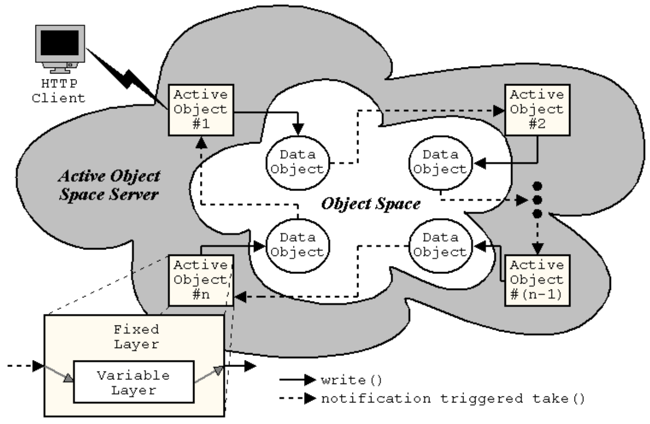

Fig. 2. HTTP Server Design

Figure 2 gives an overview of the design principles employed when building our server, supporting a subset of the HTTP/1.1 standard [11]. The active object that communicates with HTTP clients first asks the object space to deliver it any completed responses that may be deposited there. It then establishes a server socket and falls into a continuous loop, accepting client HTTP requests that it converts into request objects and delivers to the object space. Whenever a completed response is placed on the object space, the response is delivered back to this active object and the object transmits the response back to the client. Response delivery is managed via notification template processing.

In early experiments, we discovered that the active object responsible for communicating with the HTTP clients needed to contain a mapping of open client sockets to requests as part of its internal state. Java Socket objects represent underlying Operating System resources, a class of object that cannot be sensibly delivered into the object space. It was therefore necessary to have this active object retain open socket state to support the HTTP/1.1 requirement for using the same socket for request receipt and response delivery.

With careful usage of notification templates, the active objects cooperate to produce HTTP responses in a way reminiscent of a “Pipes & Filters” architecture [12], where each active object acts as a filter and enacts some self-contained transformation of the HTTP content. However, there are some important kinds of adaptation our architecture allows that are not available to more rigidy coordinated architectures.

Firstly, the object space allows decoupling of response generation across time. A response in a partially complete state could be left that way for an arbitrary amount of time, and service other requests instead in a manner reminiscent of

Floyd and Jacobson's link-sharing [13]. We might find this desirable for certain classes of HTTP client, such as screen-scraping bots that adversely effect response-times, and in turn, displease human clients who are far more judgmental of poor web response times [6].

Secondly, the active objects are loosely decoupled across their interfaces, using template matching to move data through its life-cycle. Though we have not taken full advantage of this loose coupling, we could (as an example) have certain active objects with the same template effectively "compete" for processing a HTTP request at a given point in its life-cycle. As active objects are just a special type of client for the object board, certain behaviour and its partially complete state could be relocated to a remote location if it made sense to do so.

To make behaviour replacement easier, we deliberately minimise the state each object needs to hold by making as much use of the object space as is appropriate. Data objects represent the core state of a HTTP response at various points in its life-cycle from request receipt through to response delivery.

We separate active objects loosely into a "fixed" layer and a "variable" layer. The fixed layer typically focuses on communication with the object space. It delegates behaviour that manipulates the content retrieved from the object space to its variable layer component(s). The notification template that this layer establishes with the object space describes the nature of objects it is willing to manipulate from the object space. The fixed layer is also responsible for spotting and refreshing variable layer components when they are delivered to the object space, again by using notification templates.

The variable layer consists of one or more objects that work together to process data handed to it by the fixed layer. There is nothing stopping a variable layer object from talking directly to the object space or even containing a nested fixed/variable layering itself.

We considered an alternative to behaviour update, where active objects ask the object space to inform them of the delivery of a replacement for them. A receipt of notification that a replacement had arrived would signal their need for termination. A drawback we saw to this is that a great deal of code (especially code for marshaling data to and from the object space) would be essentially the same across two versions of an active object. We opted for an active object design that expects more frequent changes in component behaviour than changes to coordination.

2.3 Dynamic Web Content Degradation via ActiveObjectSpaces

Figure 3 shows what happens generally when we vary our HTTP Server behaviour. A number of new active objects, class definitions and variable-layer objects are delivered to the object space. The server then installs the active objects and class definitions. Finally, active objects that are informed of matching variable-layer objects will retrieve and internally install these new variable-layer objects.

In our specific case for web content degradation, a second client module connects to the server and delivers a single active object and a number of support

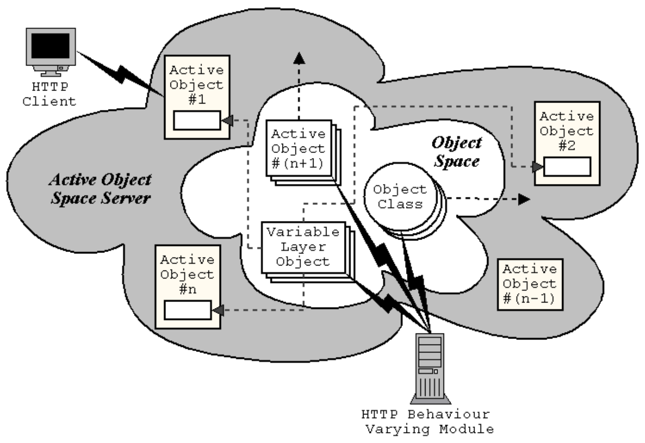

Fig. 3. Content-Degrading HTTP Server via AOS

objects and their class definitions. As this active object is being installed in the server, it first asks the server to deliver it any `ResponseTime` objects (a new type of data object the server was initially unaware of) from the object space. The active object then delivers a variable-layer object to the server space capable of generating these new data objects.

The active object responsible for HTTP request/response communications objects is informed of the existence of the new variable-layer object, and installs it as a new piece of functionality to be run each time a response is successfully delivered back to a client.

This newly delivered active object implements the algorithm for elapsed-time web content degradation that we described in [4]. We leave a more detailed discussion of the algorithm to section 3.4. Here, we will simply state that if this active object decides a new approach to generating web content is needed to ensure adequate response times, it delivers a new variable-layer object to the object space which will be picked up and installed by the appropriate active object, for use in content generation from then on.

2.4 Pitfalls discovered

In building the base server module, we learned several important lessons in ensuring high throughput and low latency using the AOS to serve web content.

One lesson learned was to minimise the number of objects on the space growing over time. An early attempt at building the server had tried to place nearly all state on the object space. Some information however, needed mutual exclusion sections to ensure the correct history was being generated. As the state requiring mutex was in the object space, the mutex behaviour was coded using AOS primitive methods. The application-level implementation of mutual exclusion was enough to ensure that `ResponseTime` objects started backing up on the object space, which in turn slowed down template matching on `take()`

operations, establishing a systemic collapse of response times as more and more new request messages mixed in with the steady growing number of `ResponseTime` objects.

In a related theme, the object space requires a great deal more object creation and destruction than a traditional call and return architecture. We learned through profiling that garbage collection on our single CPU server machine tended to contribute to poor response times when large amounts of memory were being reclaimed. Through careful configuration of the garbage collector, we were able to minimise its impact, though a more attractive proposition for future work is having a machine where garbage collection can run in parallel with service provision.

Finally, without centralised control, it became difficult to determine that we'd achieve intended behaviour until run-time. Run-time diagnostic tools such as server logs and profilers, giving real-time views into the running system became our primary method for debugging and system validation. Unit testing tended to be trivial, involving testing highly specialised, mostly stateless active objects. We couldn't verify intended overall behaviour had been achieved until actually seeing the AOS successfully marshaling request state between the loosely coupled active objects in a running environment.

3 Experiments

3.1 Experimental Setup

Experiments were carried out on a set of eight dedicated Sun boxes running Debian Linux, and a single target web server machine. The target web server machine was an 804MHz Pentium 3, with 256Mb of memory, running Fedora Core 1[2]. The AOS was run within a Sun 1.4.2-b28 JVM. The machines were connected on an isolated 100 megabit per second LAN.

One of the machines acted as a traffic synchronizer by broadcasting UDP messages to synchronise the activity of the other client machines. The remaining seven machines were used as test clients and would listen on heartbeat signals from the traffic synchroniser. For the tests we describe here, the traffic synchroniser was used only to ensure that all client machines started requesting content from the server at the same moment.

One of the test client machines was used to generate a sufficient request rate to ensure response times for the baseline approach were above one second. This client machine sent a request and once the request was received, would immediately disconnect without informing the server, generating the equivalent of a denial of service attack. Both architectural styles (Tomcat and AOS) continue to process requests and fail only when attempting to transmit responses back to the non-sampling client on a defunct client socket. The remaining six machines sent requests and waited for responses before sending new requests. The figures

[2] http://fedora.redhat.com/

reported in our third experiment have been sourced from the data collected off these six machines.

We simulated memory-intensive web content provision by generating HTML responses that ran a number of loops, accessing a random element of a 2K block of memory in each loop as they processed output. We considered four approaches for supplying a reply to a given URI that we wanted to automatically vary content generation on. Given the capacity of the target machine running the web server, we settled on four approaches doing 1,000,000, 500,000, 250,000 and 125,000 loops respectively. The 1,000,000 loop approach we name our "baseline" approach, which represents the original content being delivered before content degradation becomes necessary.

There is compelling Human/Computer interface research suggesting that we should try returning responses to web requests within one second for human clients if we want them to remain unaware of having waited for those responses [6]. When we present our results, we label responses adequate if they took under one second to return to the sampling clients. The aim we set out to achieve with our content degradation module is to maximise the number of adequate responses returned by trying to get response times back under one second once a server fails to do so for some type of web content.

3.2 Experiment 1: Sustainable Throughput per Approach

We used the tool httperf [14] to establish the sustainable throughput each approach can deliver, and used that figure to establish the number of client machines required to significantly tax the server past this rate for our baseline approach. We did this by sequentially requesting 10,000 results per approach, and recording the output from httperf. The average response time we get from this is around where a server starts degrading performance if requests arrive at rates faster than this. The key results are displayed in figure 4(a).

	Resp. Time (ms)				Std.
Loops	Avg.	Med.	Min.	Max.	Dev.
125,000	51	50.5	50	162.5	2.4
250,000	84.1	83.5	83	105.8	2.1
500,000	149.7	148.5	148.6	477.8	4.4
1,000,000	280.5	279.5	278.3	696.3	6.8

(a) AOS HTTP Module

	Resp. Time (ms)				Std.
Loops	Avg.	Med.	Min.	Max.	Dev.
125,000	47.4	47.5	47.1	58.4	0.5
250,000	91.6	91.5	91.1	104.0	0.6
500,000	179.9	179.5	179.3	192.2	0.9
1,000,000	356.5	356.5	355.5	413.3	1.6

(b) Tomcat Application Server

Fig. 4. Sequential approach response times across architectures

Using the same hardware, we ran httperf against a Tomcat 5.0.28 implementation [15] configured as closely as possible to our HTTP module with the approaches embedded in servlets. Our aim was simply to establish whether AOS could handle a similar amount of throughput for the same workloads as a contemporary web application server. The results are displayed in figure 4(b).

Minimum, average and median response times were similar in both the AOS and Tomcat. As the AOS implements a subset of the HTTP protocol, we argue only that the architectural overhead of the AOS makes it a viable candidate for delivering HTTP content. We note the large difference in maximum response times and thus standard deviation between AOS and Tomcat. A small minority of AOS responses reported much longer response times. The remainder, like Tomcat, returned much closer to their minimum response times.

Via profiling, we saw two contributors to this variability in AOS response times. Firstly, AOS requires significantly more short-lived objects than Tomcat in response processing, and is thus more prone to the *excessive dynamic allocation* anti-pattern [16]. Secondly, how the AOS uses threading has introduced extra non-determinism. There is a single thread per request for Tomcat, meaning no extra threading overhead whilst requests were being fed to it sequentially. In the AOS, each active object notification is executed in a separate thread. Even when sending requests sequentially, the AOS still invokes threading overhead as requests moves through its life-cycle.

We conclude that in non-overload conditions, the overheads of the AOS architecture do not exclude it from being used as a viable HTTP application server. We can expect occasionally longer response times than a single-thread per request architecture, but for the most part, responses will be returned with little extra evident latency.

3.3 Experiment 2: Understanding Overload Behaviour

As the architectures handle request processing very differently, we were interested in what to expect in worsening overload conditions on each server. In this experiment, we used httperf to request the baseline approach at varied request rates: from below, at around, and above the sustainable request rates derived from Experiment 1.

From Fig. 5 we see that as we increase our request rates past the sustainable point, Tomcat rapidly degraded in terms of responses per second, but the AOS settled on an average response rate at around it's sustainable request rate. We also see worsening AOS response times as request rates increases beyond sustainable rates. Either way, past the sustainable request rate, both architectures rapidly reached the point of poor adequacy.

The reason behind this difference in throughput between architectures is described by Welsh et.al [17]. Architectures that combine threading (primarily for IO and network blocking) with event-driven task scheduling offer excellent throughput characteristics for overload situations, though latency increases as request rates exceed a sustainable limit. Because of our globally shared object space amongst active objects with no inherent queuing per thread, we suspect that the AOS lies somewhere between Tomcat and SEDA [18] (the architecture Welsh et.al built from the principles described in [17]) in its ability to maintaining high throughput in overload conditions.

We conclude that the AOS will maintain high throughput rates that should slowly degrade (as a function of number of objects in the object space) in over-

Requests	Response Time (ms)				Std.	Resp. per Sec.	
per Sec.	Avg.	Median	Min.	Max.	Dev.	Avg.	Std. Dev.
2	279.1	278.5	277.8	338.6	3.2	2	0
3	287.4	286.5	277.9	301.5	2.5	3	0.1
5	9358.9	12635.5	298.9	13341.3	4986.1	2.8	0.4
10	10854.5	12722.5	319.7	13382.6	4009.8	3.3	0.3

(a) AOS HTTP Module

Requests	Response Time (ms)				Std.	Resp. per Sec.	
per Sec.	Avg.	Median	Min.	Max.	Dev.	Avg.	Std. Dev
2	357.4	357.5	356.0	467.4	2.7	2	0
3	2382.9	1048.5	381.9	9804.4	2602.3	0.4	0.9
5	3597.3	1782.5	536.1	9680.8	3288.7	0.1	0.3
10	1475.3	1606.5	1475.3	1746.9	135.8	0	0.1

(b) Tomcat Application Server

Fig. 5. Request Rates on Baseline across architectures

loaded server conditions, but that latency will continue to increase until such time as we make content cheaper to generate or offload requests elsewhere.

3.4 Experiment 3: Validating JIT Content Degradation

Our web content degradation module uses an “elapsed-time of response generation” based algorithm to choose faster or slower approaches to generating content for a given URI [4] [3]. Other content degradation algorithms could have been attempted, but are outside the scope of this research. Instead, our aim is to use the algorithm to verify that the AOS is capable of handling quite radical adaptation of both its coordination and bottleneck components in an already overloaded web application server.

We chose not to compare AOS content degradation against Tomcat. The difference between the two architectures seen in Experiments 1 and 2 led us to suspect that the behaviour of our algorithm was not readily comparable across architectures given their very different behaviours in overload conditions. Here we concentrate on the AOS architecture only.

We used the results from Fig, 4(a) to guide us in configuring the request-rate of our denial of service client. Our aim was to find a request rate, that combined with requests from the sampling clients would ensure that response-times for the AOS baseline approach would be above one second. We settled on a request rate of one request every 10 milliseconds from this client.

Experiment 3 was run for 20 minutes. The first 10 minutes used the baseline approach delivered with the first module, and the second 10 used content degradation, delivered via the second module.

[3] We pessimistically configure the algorithm’s parameters to 300ms for our *upper-time limit*, and 200ms for our *lower-time limit* in these experiments.

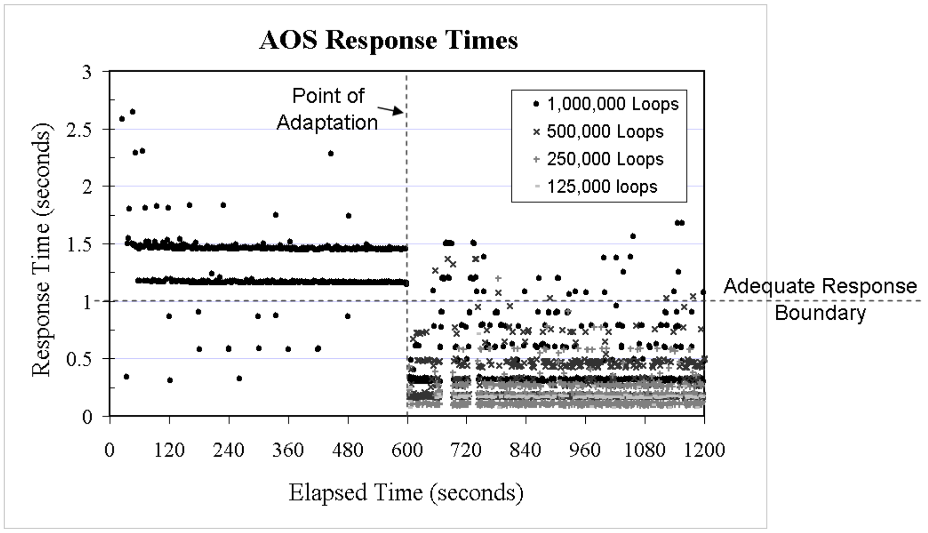

Fig. 6. AOS Responses Times

Figure 6 shows a scatter-plot diagram of response times recorded from our sampling clients. At the 600 second (10 minute) mark, the content degradation module's delivery radically altered the response-times being reported by our target server. Where most responses were definitely taking longer than one second to deliver pre-adaptation, most responses delivered post-adaptation fell below our one-second target.

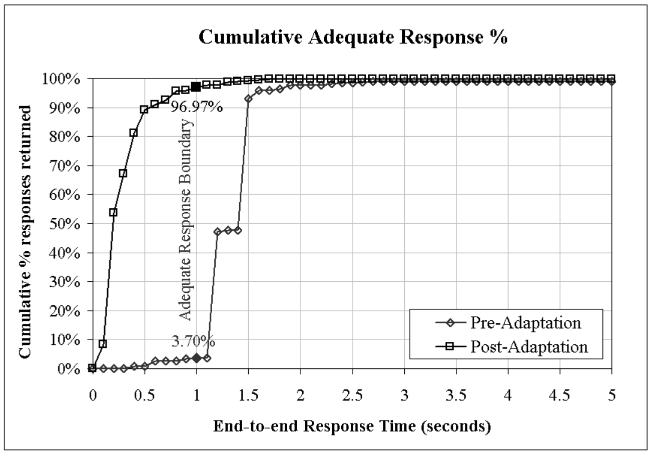

Fig. 7. Measured AOS Adequacy pre- and post-adaptation

Figure 7 shows the number of adequate responses returned pre- and post-adaptation. We achieved very high adequacy. What we draw from this result is that the AOS allows relatively complex adaptation to occur under load conditions, not only in terms of a one-off change in active-object coordination, but also in terms of automated, rapid, fine-grained changes to content-generation behaviour. We have establish that the AOS is a viable candidate for further exploration in JIT adaptation of web-service architectures, even under taxing conditions.

Of secondary importance, no special effort was made here to better match the content degradation algorithm to the AOS. Because of the the very different way in which the AOS behaves in overload conditions, there is some argument for making the algorithm less reactive to severe, short-lived changes in response times, and for it to take more consideration of how object space numbers might influence overall response latency.

4 Related Work

There are many ways to scale dynamic web content at a single server, such as caching ([19] lists several strategies), resource management (see [20] as an example), and content degradation. Caching dynamic web content is limited both in terms of when it can be applied, and how much benefit it can deliver [21]. Resource management policies often also have the undesirable "denial of service" characteristic that breaks a web user's expectations of "service on demand" from web service offerings [6].

As users seem less concerned with content makeup than with response times [5], we see web content degradation as an area deserving further exploration. Web content degradation techniques (also called transcoding, see [22], [23] and [24]) rely on the idea that web content can be degraded to a user-tolerable degree when servers are overloaded. The degraded content should require less resources to deliver, and in turn, allow more clients to consume the desired content.

Until recently, very little has been discussed in terms of degrading dynamic web content. New adaptive architectures are being introduced that look to support dynamic web content degradation, but they do not explore specific degradation techniques [1]. Chen and Iyengar described a dynamic web content degradation system involving a number of tiered servers, offering content at decreasing degrees of fidelity the further from the core server a support server is in the tier [25]. We are currently unaware of any other dynamic web content degradation techniques targeting a single server besides our own [4].

Our early work has led us to the conclusion that the architecture is the key to the degree of adaptability we can achieve. Our longer-term goal is a wide range of adaptations to suit changing situations, so maximising the range of run-time adaptability we can achieve has been a driving force in this more recent work.

To that end, space-based architectures look particularly promising to us. Early blackboard architectures such as Hearsay [9] introduced the concept of expert software components watching a blackboard, and working on parts of a problem they understood. These experts were basically self-contained components with minimal state and were loosely coupled via the blackboard. As a consequence they were easier to replace with equivalent components. Later, Gelertner's Linda project [10] on generative communication discussed the greater flexibility available across both time and space by storing and manipulating generic tuples via a tuple space.

JavaSpaces replaces tuples with Java objects, which supports the movement of behaviour in addition to state [3]. JavaSpaces however, focuses on distributed behaviour over a network. In contrast, our own interest in these architectures is in how we might take advantage of such loose coupling to introduce localised, fine-grained behaviour alterations whilst the server we are altering continues to operate. This is where the notion of active objects plays a crucial role. To best of our knowledge, there has been no previous attempt at applying coordination architectures to the problem of highly adaptable web application servers.

5 Concluding Remarks

We have shown in this paper that our coordination architecture *ActiveObjectSpaces* (AOS), can serve as a viable base for adaptive web application servers. The Active Object primitives offered by our architecture allow us to easily deliver and execute new components to a running AOS, and have them interact via localised coordination.

We described a way of using the strengths of this architecture to construct a simple HTTP server on top of the architecture, and then, to deliver significant alterations to the behaviour of the running HTTP server under load. Along the way, we learned that achieving good latency from the architecture required us to minimise the number of objects stored in the object space at any one time. We also discovered that mutual exclusion behaviour is best encapsulated inside active objects; building mutual exclusion via AOS primitives proves too costly in a busy server.

We have established that we can achieve our desired adaptability at the cost of some extra variability in server response times when compared against a more traditional web application architecture in unloaded conditions. The AOS matches the throughput and latency results of this traditional architecture close enough to make deployment viable. Overloading the architectures showed the AOS capable of maintaining high throughput, whereas the throughput of the traditional architecture rapidly degraded as request frequency was increased. Previous research has shown that a mix of threading and event-driven task scheduling (as implemented in the AOS) is the reason for the continued throughput under load witnessed in the AOS.

We used an automated web content degrading adaptation based on elapsed-time as a non-trivial example of the type of run-time adaptations we seek from the AOS. The adaptation involved the delivery of behaviour and new class definitions from a remote location as the server was suffering overloaded conditions. We have shown the AOS is capable of automated, rapid, and fine-grained changes to content-generation behaviour. Given the radically different behaviours of the architectures used under load, we've also seen that automated content degradation via elapsed-time measures should be handled differently to better suit each architecture.

From here we aim to better understand the AOS and how to best use it for adaptive, high volume HTTP service provision. Firstly, we wish to look at the granularity of work each active object performs and its impact on overall throughput and latency. We have seen in other research that thread queuing mechanisms (including the joining of two tasks into a single queue) can be of benefit and we are interested in how these concepts might carry across into our own work.

Secondly, there is still much to understand in elapsed-time based automated content degradation. With a better understanding of what elements matter in such adaptations, we might be able to supply at least partial automation. For example, we could deliver the framework for content degradation from a library of standard adaptation modules to a running application server, and allow developers to insert new versions of degraded content as they become available.

Thirdly, our content degrading adaptation adds an extra step to the end of a pipeline of coordinated tasks. This extra step supplies replacement behaviour at run-time to the bottleneck in the process. We are interested in other types of adaptation that benefit response times. Some examples might include i) altering the flow of objects by changing notification templates at run-time, ii) introducing active objects to compete with others for certain steps, or iii) automated migration of active objects and partially complete state to less loaded AOS environments.

References

1. Colajanni, M., Lancellotti, R.: System architectures for Web content adaptation services. Distributed Systems Online, Web Systems Topic (2004) http://dsonline.computer.org/was/adaptation.htm.
2. Garlan, D.: Software Architecture: a Roadmap. In Finkelstein, A., ed.: The Future of Software Engineering. ACM Press (2000)
3. Doberkat, E.E.: E. Doberkat, E. Freeman, S. Hüpfer, K. Arnold: JavaSpaces Principles, Patterns and Practice. Softwaretechnik-Trends **20** (2000)
4. Bradford, L., Milliner, S., Dumas, M.: Scaling Dynamic Web Content Provision Using Elapsed-time-based Content Degradation. In: Proceedings of the 5th International Conference on Web Information Systems Engineering (WISE 2004), Brisbane, Australia, Springer Verlag (2004) 559–571
5. Ramsay, J., Barbesi, A., Peerce, J.: A psychological investigation of long retrieval times on the World Wide Web. In: Interacting with Computers. Volume 10., Elsevier (1998) 77–86
6. Bhatti, N., Bouch, A., Kuchinsky, A.: Integrating user-perceived quality into Web server design. Computer Networks (Amsterdam, Netherlands: 1999) **33** (2000) 1–16
7. Bouch, A., Kuchinsky, A., Bhatti, N.: Quality is in the Eye of the Beholder: Meeting Users' Requirements for Internet Quality of Service. In: Proceedings of the CHI 2000 Conference on Human factors in computing systems, ACM (2000) 297–304
8. Miller, R.: Response Time in Man-Computer Conversational Transactions. In: Proc. AFIPS Fall Joint Computer Conference. Volume 33. (1968) 267–277

9. Reddy, D.R., Erman, L., Neely, R.: A model and a system for machine recognition of speech. In: IEEE Transactions on Audio and Electroacoustics. Volume 21. (1973) 229–238
10. Gelernter, D.: Generative communication in Linda. ACM Trans. Program. Lang. Syst. **7** (1985) 80–112
11. : HTTP/1.1 (RFC 2616) Standard (1999) http://www.ietf.org/rfc/rfc2616.txt.
12. Shaw, M., Garlan, D.: Software Architecture. Prentice Hall, Upper Saddle River, New Jersey (1996)
13. Floyd, S., Jacobson, V.: Link-sharing and resource management models for packet networks. IEEE/ACM Transactions on Networking **3** (1995) 365–386
14. Mosberger, D., Jin, T.: httperf-A Tool for Measuring Web Server Performance. SIGMETRICS Perform. Eval. Rev. **26** (1998) 31–37
15. The Apache Group: Tomcat web application server (2004) http://jakarta.apache.org/tomcat/.
16. Williams, L.G., Smith, C.U.: PASA^{SM}: A Method for the Performance Assessment of Software Architectures. In: WOSP 2002: Third International Workshop on Software and Performance, Rome, Italy, ACM Press New York, NY, USA (2002)
17. Welsh, M., Gribble, S.D., Brewer, E.A., Culler, D.: A Design Framework for Highly Concurrent Systems. Technical Report UCB/CSD-00-1108, UC Berkeley (2000)
18. Welsh, M., Culler, D.: Adaptive Overload Control for Busy Internet Servers. In: USENIX Symposium on Internet Technologies and Systems. (2003)
19. Shi, W., Collins, E., Karamcheti, V.: Modeling Object Characteristics of Dynamic Web Content. Technical Report TR2001-822, New York University (2001)
20. Chen, X., Heidemann, J.: Flash Crowd Mitigation via Adaptive Admission Control based on Application-level Observations. Technical Report ISI-TR-557, USC/Information Science Institute (2002)
21. Thomas M. Kroeger, Darrell D. E. Long, J.C.M.: Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In: 1st USENIX Symposium on Internet Technologies and Systems, Monterey, California, USA (1997)
22. Amir, E., McCanne, S., Katz, R.H.: An Active Service Framework and Its Application to Real-Time Multimedia Transcoding. In: SIGCOMM. (1998) 178–189
23. Chandra, S., Ellis, C.S., Vahdat, A.: Differentiated Multimedia Web Services using Quality Aware Transcoding. In: INFOCOM 2000. Proceedings of the Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Volume 2., IEEE (2000) 961–969
24. Tarek F. Abdelzaher and Nina Bhatti: Web Server QoS Management by Adaptive Content Delivery. In: Proceedings of the 8th Internation World Wide Web Conference, Toronto, Canada (1999)
25. Chen, H., Iyengar, A.: A Tiered System for Serving Differentiated Content. In: World Wide Web: Internet and Web Information Systems. Volume 6., Netherlands, Kluwer Academic Publishers (2003) 331–352