Varying Resource Consumption to Achieve Scalable Web Services

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. Web service deployment is hampered by the possibility of sudden variations in request volumes. Mechanisms exist to enhance scalability in times of heavy load when the delivered content is static. However, web services typically involve dynamic content, delivered through application servers which may have little to no support for adapting to varying loads in order to ensure timely delivery. In this paper we discuss why scaling dynamic content delivery under load is difficult, we present a technique for controlled service degradation to achieve this scalability, and we present experimental results evaluating its benefits.

1 Introduction

The vast and uncertain environment of the Internet has direct consequences for service delivery, and has the potential to derail attempts to provide economic benefits via the electronic offering of services. In particular, the tardy delivery of a service may force the provider to decrease the price or increase the quality to offset the negative effects of the poor delivery.

The delivery of software as a service across the Internet carries with it significantly more risk than that of offering functionally equivalent shrink-wrapped software. By logically centralizing the software and offering it as a service, the service provider invites direct customer dissatisfaction by transferring originally client-side risks (such as the provision of an an environment capable of meeting client demand) back upon themselves. In addition there is a significant risk of insufficient capacity for client demand: a direct consequence of the unpredictable Internet environment in which a web service operates. Web services exposed to the Internet may experience huge demand fluctuations. These changes may occur rapidly, making it impossible for a human operator to respond in a timely manner. This situation is exacerbated if a client's expectations of a service remains high regardless of the problems being experienced by the service provider.

A possible way to maintain timely delivery in cases of capacity overflow, is by dynamically varying service overhead so that performance degradation is gracefully managed and user dissatisfaction minimized. Examples of situations where service degradation can be applied include: (i) choosing not to execute code for generating advertising content; (ii) skipping non-essential validations of

XML messages; and (iii) offering delayed data (e.g. stock quotes) rather than the most up-to-date version.

As a step toward a system for controlled service degradation, this paper aims:

1. to investigate the impact of high user load (such as a flash crowd) on a given service, and
2. to determine if benefits associated with a simple method for dynamically varying service resource consumption through service degradation, outweigh the overhead introduced.

The paper specifically considers Quality of Service (QoS) measured only in terms of response time and focuses on CPU consumption, as evidence indicates that this is a bottleneck when delivering dynamic content [1].

Section 2 discusses previous work on scalability of web service provision under extreme load. Section 3 discusses our approach to measuring service adequacy. Using this approach, we discuss in section 4 our initial technique for mitigating the effects of heavy load using variable CPU consumption. Section 5 presents three experiments involving large user load on a simulated service offering and discusses the results. Section 6 concludes outlining directions for future work.

2 Background

When delivering static content such as basic HTML pages, network bandwidth is typically the bottleneck [2]. Given that static content changes infrequently, basic Internet infrastructure allows caching of the content "closer" to the user via reverse-proxy caching, web browser caching, Content Distribution Networks (CDN), etc. This allows a degree of scalability by spreading bandwidth consumption amongst a set of remotely located caches.

Resource management for bandwidth under heavy load has also been investigated and typically revolves around techniques to limit certain types of bandwidth consumption (see [3] for a comparison of popular approaches) and controlled degradation of service, for example [4].

In times of extremely heavy load, traditional caching techniques can be augmented with more active approaches to better meet demand [5]. Peer-to-Peer caching schemes [6], the use of adaptive CDNs [7] and Cooperative networking models [2] where clients act as proxy cache servers to newer clients have all been investigated as approaches to mitigate the effects of heavy request loads for static content. Regardless of the caching strategy chosen, the core target of each strategy is to maximise *cache-hits*, allowing the originating server to deliver to just a minimised number of clients that do not have access to a remote cache.

When looking to scale delivery of dynamic content offered by web services, however, the application of caching is problematic. Basic Internet infrastructure offers limited support for caching results to dynamic queries, simply because it can make no guarantees on what future response might be. Considering that the delivery of dynamic content becomes CPU bound, the target for content caching shifts to saving on content construction overhead. Dynamic content caching

schemes work on the idea that some amount of the dynamic content remains static enough over time to justify the overhead of synchronizing copies with the source content as this source content changes. Data Update Propagation [1], Active Query Caching [8] and Macro-Pre-Processing of HTML [9] are examples of this approach. The applicability of dynamic content caching however, is limited as noted by Yagoub et al. [10]. Hence, the possibility of minimizing the cost of dynamic content generation during times of heavy load should be considered.

When considering the differences between static and dynamic content delivery, the bottleneck typically shifts from bandwidth to CPU. Kraiss et al. [11] note that the bottleneck for e-services resides in the application server and back-end database servers. Padmanabhan & Sripanidkulchai [2] point to the effects of September 11's tragedy on a popular news site. As dynamically generated content initially made the CPU the primary bottleneck, it was manually replaced with static content, shifting the bottleneck to network bandwidth. Challenger et al. [1] state that "a typical dynamic page may require several orders of magnitude more CPU time to serve than a typical static page of comparable size".

Graceful degradation of service in times of server-side under-capacity is not a new concept [12]. When discussing CPU as a resource however, existing discussion on techniques for resource management seem limited to queue management of a static algorithm for response generation (see [11], [13], and [14] as examples). These techniques, while useful for delivering quality service to a limited number of the total requests received, do little to increase overall service scalability. Hence, we concentrate on when a process becomes CPU bound and how judicious service overhead reduction may help us increase overall scalability.

Certain parallels exist between our proposal and the automatic Quality of Service (QoS) control mechanisms proposed by Menascé and Mason [15]. Their approach however involves modifying the configuration of the Application/Web Server, which usually cannot be done at runtime as required in case of sudden capacity overflow. In contrast, our work focuses on what the application developer may do without modifying the underlying server's configuration or API.

3 Measuring service adequacy

3.1 Service Time

Perhaps one of the biggest differences between web services and traditional distributed computing is the service's potential concurrent audience size and that audience's expectations for adequate service provision regardless of its own size. For services to be consumed by an end user in real-time (a user sitting and waiting for response), we define a *client_acceptable_time_limit* taking the following into consideration:

- Any human/computer interaction taking over a second is an obtrusive delay to the user [16].
- An end user's perception of service quality is strongly influenced by response time. Content makeup has little effect on user's perception of QoS [17].

– Though end users typically have a conceptual model of dynamic content delivery taking more effort than static content, these conceptual models are often grossly mismatched with the effort actually required by the server [18].
– So long as service response time is adequate, the requesting client will not bail-out waiting for it [19].

Therefore, we assume a value of 1 second as our *client_acceptable_time_limit*. We use this time limit as a worst-case response time a server should endeavour to deliver a response to its client within. This limit is a hard upper bound on performance (as opposed to say, an average over time, allowing some responses to go above the limit). We do this in recognition of user behaviour that will punish a service provider for a 4 second response the first time it happens, regardless of the fact that the past several responses were sub-second, and that an average over all the responses might definitely be within a sub-second bound.

This client acceptable time limit will no doubt vary with circumstance. We can imagine that a client may have varying requirements depending on the client's nature. A software client that is in turn a service composed of several other services may need a tighter client acceptable time limit if it is to eventually serve its composed service to some end human consumer.

The base metric we wish to collect is that of *end-to-end_response_time*, defined in [20] as *the amount of time that passes from when a client first starts sending a request to when the client completes receiving the response*. We view *end-to-end_response_time* as an attractive measure for controlled test environments, whilst acknowledging that it may be difficult (if not impossible) to collect in a real-world setting.

The choice of application server can severely restrict an application developer's options in attempting to deliver dynamic content in a user acceptable time-frame. We choose the Apache Tomcat 4.1.18 servlet engine [1] and deliberately limit ourselves to what we can achieve without modification of the Servlet engine itself. As we do not have access to the time a servlet resides on the request queue, we cannot use approaches based on length of time in queue.

3.2 Service Adequacy

We define the function *service_adequacy* between two time points (t_1 and t_2) as:

$$\frac{adequate\_responses(service\_requests(t_1, t_2))}{\#\ service\_requests(t_1, t_2)} \tag{1}$$

where $service\_requests(t_1, t_2)$ is the set of requests sent between times t_1 and t_2, while $adequate\_responses(\{r : ServiceRequest\})$ is the number of requests in the supplied set that return a response within the *client_acceptable_time_limit* discussed previously.

By varying effort in dynamic content delivery, we seek to maximise the service adequacy over a period of heavy load, which in turn means generating a higher

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

number of adequate responses to requests under that load than what we would have without varying effort. This measure does not take into account any user "dissatisfaction" with a less "complete" service provision.

4 Design Considerations

We assume that for processing a given type of request, there are multiple *approaches* available, each with its own expected execution time. Approaches are statically ranked during servlet initialization in order of expected system time execution from heaviest to lightest. We also assume that a service provider would prefer to generate the most costly approach they can so long as the response is timely. Thus, in times of heavy load, we choose a less costly approach; in times of light load, we choose one more costly.

Approaches are chosen via an *approach selector* illustrated in Fig. 1. The approaches for handling a service request are placed in a group and ranked by cost. Each approach in a group is either active or inactive. Active approaches are capable of processing a service request whereas inactive ones will accept no new requests for processing. Initially all approaches in a group are active.

We define two thresholds for the activation and deactivation of approaches:

***time_limit*:** The elapsed server time in which we desire a response to be sent for any received request. Once the reported cost of an approach breaches this limit, the approach is deactivated. The cheapest approach is not deactivated, regardless of the degree to which the limit is breached.

***reactivation_threshold*:** Once the reported cost of an approach drops below this threshold, the next most expensive (previously deactivated) approach will be re-activated and its reported cost reset to zero.

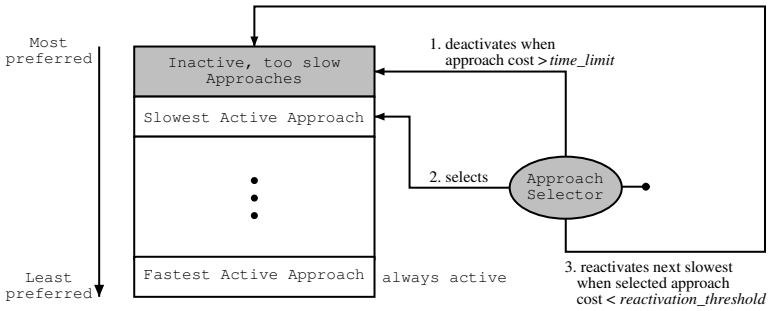


Fig. 1. The Approach Selector

The threshold *time_limit* differs from the *client_acceptable_time_limit* defined earlier, in that it takes no account for time outside the control of the server (such

as communications overhead). The *target_processing_time* should conceivably be small enough to allow reasonable communications on top of processing time to allow overall delivery within the *client_acceptable_time_limit*. The focus of our research, however, is not in having the service consider network latency in its delivery so we model the time difference as a simple constant value.

Every request for a service is passed through the approach selector, implemented as a servlet filter. This filter can be plugged without requiring any alteration to the servlet engine itself. Each time that the processing of a service request is completed using a given approach, the approach selector gets the cost. If the cost has fallen below the *reactivation_threshold*, the next most expensive deactivated approach is reactivated, and its cost reset to zero. The next service request will be processed using the reactivated approach. The reported cost of a single approach is the worst elapsed time recorded of the last w calls to the approach. In the experiments presented below, we have set w at 20 for the experiment after some initial trials on the stability of approach cost reporting.

Note that for service degradation to be applicable, several versions of the application providing the service should be available: a "full" version and one or more "degraded" versions. In order to apply controlled degradation, the need for multiple versions should thus be imposed as a requirement in the application development process. In order to reduce the costs of handling this requirement in terms of analysis, design, coding, and testing, (semi-)automated support can be provided. In particular, design-level support could be introduced that would allow designers to provide meta-data about the cost and importance of various processing steps. At the coding level, special directives could be inserted in the dynamic content generation scripts (e.g. JSP, ASP, or PHP) to indicate for example not to perform a full XML message validation, or not to generate certain document portions in the degraded version of the service. As the space of possibilities is large, this issue desserves a separate treatment in future work.

5 Experiments

5.1 Simulation Environment

Our experiments are carried out on a set of eight dedicated Sun boxes running Debian Linux on an isolated 100 megabit/second LAN. The testbed relies on a single testing server and a single target server. The testing server sends broadcast UDP messages to synchronise the activity of clients in order to generate differing request patterns. The target server, a Sparc Ultra-5 with 192 Mb of memory, contains the application server and approach selection algorithm to be tested.

The remaining six boxes act as test clients. Two of these machines act simply to generate sufficient request traffic to tax the target server. These two machines have been configured to generate enough traffic to ensure the server is close to, but not actually at the point where, the server will refuse an incoming connection. The other four machines send requests and wait for responses before sending new requests. The figures reported later have been derived from the data collected on this second set of boxes.

The request processing component for each service delivery approach is an instance of a "workload simulation" servlet, pre-configured to run a set number of floating-point calculations of 3000, 1000, 500 or 100 loops. A loop of 3000 floating-point calculations is used to represent the baseline approach. A response from a service returns processing time as measured by the server.

Each experiment is conducted using two request traffic patterns. The "steady" pattern corresponds to a server under heavy, constant request pressure, as might result from the arrival of a flash crowd. The "bursty" pattern alternates periods of high arrivals of requests which go beyond server capacity, with periods of no arrivals. In future work, we look at other traffic patterns including the transition between the steady and bursty request patterns. Note that for both patterns, the total amount of requests over the period of one experiment is the same.

As a result of the non-threaded nature of the sampling clients, the number of requests that these data sampling clients can make is bound by the response times delivered by the server. Our expectation with introducing approach selection is that we should be able to reduce end-to-end response times, and allow significantly more timely service responses reported from the sampling machines than when run against the baseline.

Tests are run for one hour each to ensure the receipt of enough sampled data in the worst case. Statistics are collected for the last 50 minutes of each experiment to ensure the results are not optimistically skewed by warm-up time. We track the *end_to_end_response_time* of each client request and report the percentage that fail to fall within our *client_acceptable_time_limit*.

The time thresholds *time_limit* and *reactivation_threshold* are set at 800 and 400ms respectively. The *time_limit* threshold of 800ms was chosen as a number within our one second target with 200ms set aside to receive and transmit request and response messages. The 400ms *reactivation_threshold* was simply chosen as half of the *time_limit*.

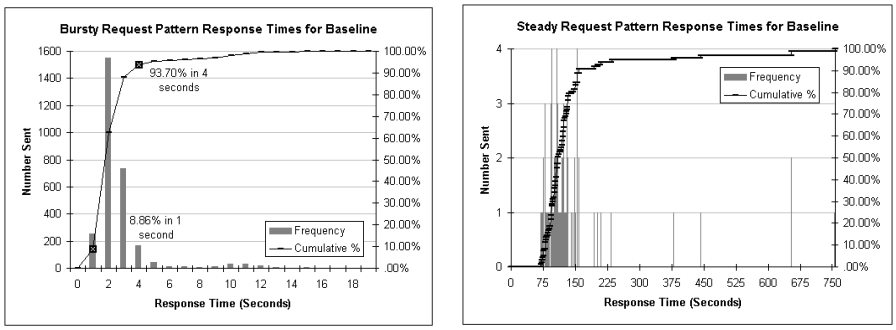
We performed three sets of experiments. In the first experiment we run the baseline approach against both the steady and bursty request patterns. The second experiment replaces the baseline approach with our approach selection algorithm and was run against the bursty request pattern. The third experiment uses the approach selection algorithm against the steady request pattern.

Results are displayed as a histograms, with the number of responses returned to the sampling clients within a given timeframe represented as gray bars along the x axis. A line showing the cumulative percentage of responses is also supplied. Service adequacy is the percentage of responses returned within one second.

5.2 Experiment 1 - Baseline approach for both request patterns

Figure 2(a) shows that most responses were received by the sampling clients within two seconds when tested against the bursty request pattern. Via our definition of service adequacy, only around 9% of the responses received were adequate, however. The steady request pattern results in Fig. 2(b) shows that the server was taxed heavily as a result of the request traffic generated. The

minimum response time recorded was slightly under 69 seconds. As a result, the baseline delivered zero service adequacy against the steady request pattern.



(a) Baseline: Bursty Request Pattern (b) Baseline: Steady Request Pattern

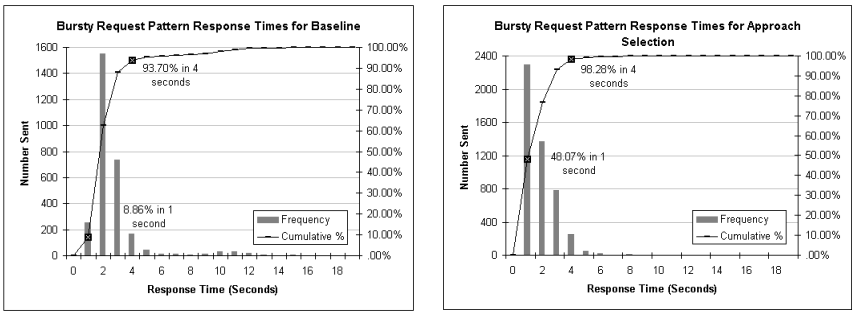
Fig. 2. Effect of running baseline against bursty and steady request patterns

The experiments show that the request arrival pattern has a strong effect on service adequacy. By simply allowing a minute between bursts of requests, the baseline response times are 20 times better with the bursty request pattern. This is predictable given that the bursty arrival pattern allows the service to use the period of no requests between bursts to focus on response delivery, and therefore achieve some "adequate" responses. However, it is surprising that almost no request took more than 20 seconds to process under the bursty pattern, which shows that the system does perform request processing during the bursts. Note that the same total amount of requests were processed under both approaches.

5.3 Experiment 2 - Approach Selection for bursty pattern

Here we compare the bursty baseline presented in Fig. 2(a) (repeated in Fig. 3(a)), with approach selection in Fig. 3(b) The number of adequate responses has moved from a little under nine percent to around 48 percent. Again, most messages were received within four seconds of sending the request. However, approach selection has resulted in a much smaller worst case response time, and significantly more responses within one second. We conclude that the approach selection overhead does not outweigh its benefits.

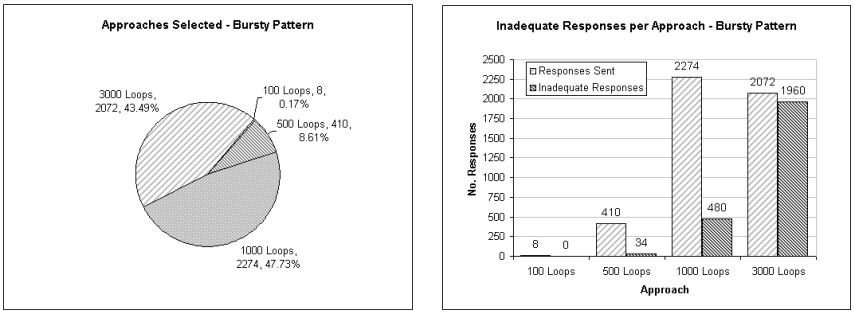
Figure 4(a) shows the breakdown of approaches selected for the bursty request pattern. Most requests were balanced between either the most costly, or second most costly approach. Very few attempts were made to use the lightest two approaches. Figure 4(b) shows that most of the inadequate responses generated were a result of attempting the most costly approach, thereby showing a deficiency in in the approach selection algorithm. To generate better service adequacy for the bursty pattern, we would expect far less attempts at the 3000 loop algorithm and more on the 1000 and 500 loop algorithms instead (see section 6).



(a) Baseline (b) Approach Selection

Fig. 3. Bursty request pattern using baseline and approach selection

The results in Fig. 4(b) suggest that the decision-making algorithm for approach selection can be poor when determining when to switch between the most costly and second-most costly approaches with the bursty pattern. We believe a contributor to this result is the number of threads that can be simultaneously running. As the number of threads running a given approach increase, it allows for larger numbers of in-progress threads that still need to complete using a given approach once that approach is deactivated. The bursty traffic pattern is likely to be compounding the issue, as we expect it to exercise the switching of approaches more frequently than the steady request pattern.



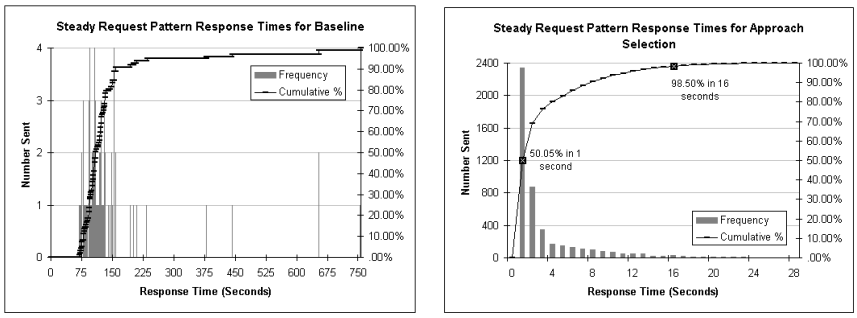
(a) Approaches Selected (b) Inadequate Responses per Approach

Fig. 4. Breakdown of approaches selected for bursty request pattern

5.4 Experiment 3 - Approach Selection for steady pattern

In this experiment we repeat the baseline results for a steady request pattern (from Fig. 2(b)) in Fig. 5(a) and compare them against running the same pat-

tern with approach selection in Fig. 5(b). Most responses were received between 75 and 150 seconds of transmission from the client in the baseline experiment and the worst-case response time took over 755 seconds to return. No adequate responses to baseline requests were received by the sampling clients.



(a) Baseline (b) Approach Selection

Fig. 5. Steady request pattern using baseline and approach selection

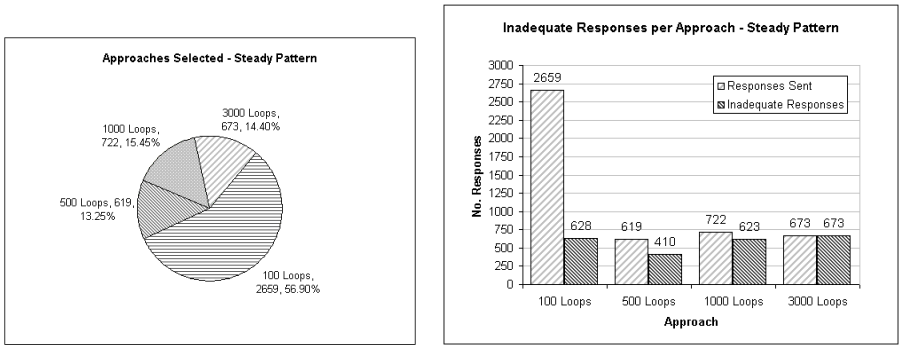
In contrast to the baseline, approach selection saw over 50% of responses return in an adequate timeframe under the bursty pattern, with most responses returned within few seconds. In addition, approach selection proved even more beneficial to response times under the steady pattern than the bursty one.

Figure 6(a) shows the breakdown of approaches selected for the steady request pattern. Most of the requests were processed using the least costly approach. This is in contrast to approach selection for the bursty request pattern, which used the 3000 and 1000 loop approaches mostly. We conclude that the accuracy of using elapsed time as a measure for determining when to switch approaches is yields better results with heavier request loads.

A relatively even distribution of attempts to use more costly approaches was recorded. Figure 6(b) shows that most of the requests processed with the least costly approach returned adequate responses, whereas, most requests attempting heavier approaches returned inadequate responses. The number of attempts at more costly approaches is surprisingly large. We theorise that a combination of the occasional break in steady request traffic, sufficient enough to retry the heaviest of approaches, and concurrent threads still running recently deactivated approaches combined to produce the numbers observed.

6 Concluding Remarks & Future Work

We have shown that despite limitations imposed by the implementation technology chosen, the principle of generating cheaper responses to a request under times of load can deliver significant performance improvement. We wish to automate the ranking of approaches to remove the limitation of hard-coding it.



(a) Approaches Selected (b) Inadequate Responses per Approach

Fig. 6. Breakdown of approaches selected for steady request pattern

The choice of cost calculation algorithm, *time_limit* and *reactivation_threshold* may also have an impact on optimal approach selection. The experiments described used a simple worst-case elapsed time calculation method with a relatively small window and time limit values that were not varied. Future work will focus on investigating the behavioural change evidenced when using the same algorithm with differing sample window sizes, varied cost sampling algorithms, differing time limit values and differing degrees of multi-threading.

We have shown that the type of request pattern has a strong effect on service provision adequacy without approach selection, and that approach selection serves to smooth the differences in response times between patterns. Though the method employed works to decrease response times and increase scalability for the request patterns tested, it produces better responses when used with heavy, constant request load. We intend to test approach selection against a wider range of request patterns to gain a better understanding of its strengths and weaknesses and help synthesize better selection algorithms. Also, we plan to study the behavioural effects of varying the two time thresholds of the proposed method, namely *reactivation_threshold* and *time_limit*, in order to undertand how they can be tuned to achieve greater scalability.

Our current experimental setup involves CPU-intensive services requiring no RAM or disk access. At present, we are working towards extending the experimental setup first to services involving frequent RAM access (e.g.to simulate XML parsing and DOM tree-traversals), and then to services involving disk access (e.g. relying on a database system).

Acknowledgments Thanks to Alex Delis for his valuable comments on a draft of this paper.

References

1. Challenger, J., Iyengar, A., Witting, K., Ferstat, C., Reed, P.: A Publishing System for Efficiently Creating Dynamic Web Content. In: INFOCOM (2). (2000) 844–853

2. Padmanabhan, V., Sripanidkulchai, K.: The Case for Cooperative Networking (2002)
3. Lau, F., Rubin, S.H., Smith, M.H., Trajovic, L.: Distributed Denial of Service Attacks. In: IEEE International Conference on Systems, Man, and Cybernetics. Volume 3., Nashville, TN, USA (2000) 2275–2280
4. Singh, S.: Quality of service guarantees in mobile computing. Computer Communications **19** (1996)
5. Iyengar, A., Rosu, D.: Architecting Web sites for high performance. In: Scientific Programming. Volume 10. IOS Press (2002) 75–89
6. Stading, T., Maniatis, P., Baker, M.: Peer-to-Peer Caching Schemes to Address Flash Crowds. In: 1st International Peer To Peer Systems Workshop (IPTPS 2002), Cambridge, MA, USA (2002)
7. Jung, J., Krishnamurthy, B., Rabinovich, M.: Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites. In: Proceedings of the International World Wide Web Conference, ACM Press New York, NY, USA (2002) 252–262
8. Luo, Q., Naughton, J.F., Krishnamurthy, R., Cao, P., Li, Y.: Active Query Caching for Database Web Servers. In Suciu, D., Vossen, G., eds.: WebDB (Selected Papers). Volume 1997 of Lecture Notes in Computer Science., Springer (2001) 92–104
9. Douglis, F., Haro, A., Rabinovich, M.: HPP: HTML Macro-Preprocessing to Support Dynamic Document Caching. In: USENIX Symposium on Internet Technologies and Systems. (1997)
10. Yagoub, K., Florescu, D., Issarny, V., Valduriez, P.: Caching Strategies for Data-Intensive Web Sites. In Abbadi, A.E., Brodie, M.L., Chakravarthy, S., Dayal, U., Kamel, N., Schlageter, G., Whang, K.Y., eds.: VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt, Morgan Kaufmann (2000) 188–199
11. Kraiss, A., Schoen, F., Weikum, G., Deppisch, U.: Towards Response Time Guarantees for e-Service Middleware. Bulletin on the Technical Committee on Data Engineering **24** (2001) 58–63
12. Hutchison, D., Coulson, G., Campbell, A.: 11. In: Quality of Service Management in Distributed Systems. Addison Wesley (1994)
13. Garg, A., Reddy, A.L.N. In: Mitigating Denial Of Service Using QoS Regulation. Volume 1270 of Lecture Notes in Computer Science. Springer (1997) 90–101
14. Leiwo, J., Zheng, Y.: A Method to Implement a Denial of Service Protection Base. In: Australasian Conference on Information Security and Privacy. Volume 1270 of Lecture Notes in Computer Science., Berlin, Germany, Springer (1997) 90–101
15. Menascé, D.A., Mason, G.: Automatic QoS Control. IEEE Internet Computing **7** (2003) 92–95
16. Miller, R.: Response Time in Man-Computer Conversational Transactions. In: Proc. AFIPS Fall Joint Computer Conference. Volume 33. (1968) 267–277
17. Ramsay, J., Barbesi, A., Peerce, J.: A psychological investigation of long retrieval times on the World Wide Web. In: Interacting with Computers, Elsevier (1998)
18. Bhatti, N., Bouch, A., Kuchinsky, A.: Integrating user-perceived quality into Web server design. Computer Networks (Amsterdam, Netherlands: 1999) **33** (2000) 1–16
19. Zona Research: The Need for Speed (1999) Whitepaper.
20. Menascé, D.A., Almeida, V.A.F.: Capacity Planning for Web Services. Prentice Hall (2001)