A Simulation Analysis of Dynamic Server Selection Algorithms for Replicated Web Services

Marco Bernardo
Università di Torino, Dipartimento di Informatica
Corso Svizzera 185, 10149 Torino, Italy
E-mail: bernardo@di.unito.it

Abstract

A practical approach to the provision of responsive Web services is based on introducing redundancy in the service implementation by replicating the service across a number of servers geographically distributed over the Internet. In this paper we compare the user perceived performance of three dynamic server selection algorithms operating at the client side, in order to provide some guidelines for adopting an algorithm that is appropriate for a given scenario. All the three algorithms use small probes to assess the network congestion and the server load before making a decision. The first algorithm downloads the whole requested Web page from a single server, while the other two algorithms concurrently download different pieces of the same requested Web page from different servers. The analysis is conducted via simulation with ÆMPA/TwoTowers.

1 Introduction

The success of a Web service is largely dependent on its responsiveness, i.e. the timeliness and availability with which the service is provided to its users. From the timeliness viewpoint, several techniques have been proposed in the literature, which share the idea of constructing a Web service out of replicated servers forming a cluster of workstations and distributing the client request load among those servers. Service availability is achieved through the redundancy inherent in the service implementation.

In [8] it is however argued that the techniques above can only partially meet the availability requirement, because they suffer e.g. from the vulnerability to failures of the router/gateway that interfaces the service cluster with the rest of the network. Moreover, the techniques above are not able to control the client latency time over the network, hence they cannot be deployed in such a way that they achieve a full timeliness.

In order to overcome the limitations above, in [8, 4] it has been proposed to replicate the servers across the Internet rather than in a cluster of workstations. In the Internet framework, a successful deployment depends on the ability of binding the client to the most convenient replicas and maintaining data consistency among the replicas. In the following we assume that data consistency is guaranteed by some dedicated mechanism.

Previous work in the field of task assignment policies for distributed server systems (see, e.g., [7, 5, 9] and the references therein) study different classes of server selection algorithms at the client side. The investigated algorithms differ for the selection criteria, which essentially are based on either exploiting past performance data or sending small probes to assess the current conditions about the network congestion and server load, but share the characteristic of selecting a single server out of the set of available servers.

In this paper we focus on the class of probe based algorithms, which we call dynamic algorithms, and we extend previous studies by comparing the performance of the classical dynamic algorithm, which selects a single server, with the performance of two novel dynamic algorithms [6], which select several servers from which different pieces of the same requested Web page are concurrently downloaded. The performance metrics considered in the comparison is the user response time, intended as the time elapsed between the generation of a HTTP request and the rendering at the client side of the requested Web page. This study should provide some guidelines for the adoption of a dynamic algorithm that is appropriate for a certain configuration.

The analysis of the three dynamic server selection algorithms is conducted via simulation, by assuming the size of the requested Web pages to follow a heavy tailed distribution in order to fit the degree of variability of empirically measured workloads. However, instead of writing a conventional simulation program, we have resorted to formal methods. More precisely, we have modeled the replicated Web service and the three algorithms with ÆMPA [2], an architectural description language based on the stochastic

process algebra $\text{EMPA}_{\text{gr,vp}}$ [3, 1], and we have carried out the simulation experiments with the related software tool TwoTowers [1]. The reason for this is that ÆMPA allows one to model systems in a compositional, hierarchical and visual way and is equipped with system architecture level checks and functional verification techniques that helps to detect the presence of modeling errors and possibly pinpoint their causes. All these features enhance the confidence in the correctness of the developed models, thereby constituting a more adequate framework than conventional simulation programs.

This paper is organized as follows. In Sect. 2 we illustrate the three dynamic server selection algorithms. In Sect. 3 we give a brief overview of the technology used to conduct our analysis, i.e. ÆMPA and TwoTowers. In Sect. 4 we present the ÆMPA model of the scenario being examined. In Sect. 5 we provide and discuss the simulation results about the responsiveness of the three algorithms. Finally, Sect. 6 contains some concluding remarks.

2 Dynamic Server Selection Algorithms

In this section we illustrate the three client side, server selection algorithms under examination by informally describing their behavior. From a theoretical viewpoint, such algorithms can be regarded as task assignment policies for a distributed server system. A task assignment policy is a rule that establishes how the service requests must be assigned to the servers of the system. The problem of finding a task assignment policy that improves the system performance perceived by the system users has been extensively investigated analytically, empirically, and via simulation (see, e.g., [7, 5, 9] and the references therein). This has resulted in a classification of the task assignment policies into static, statistical, and dynamic. Static policies select a server on the basis of the resource capacity, such as number of hops, connection bandwidth, and server architecture. Statistical algorithms, instead, reflect typical levels of resource availability or contention as they consider past performance data, such as latencies and bandwidths, to choose a server. Finally, dynamic or run time algorithms use small probes to detect the current network and server conditions in order to make a decision based on the current response times.

In this paper we concentrate on dynamic policies because, besides adapting automatically to changes in the network and server conditions, previous studies (see, e.g., [5]) show that dynamic probes are easy to implement and add little traffic or delay overhead. On the contrary, good estimators for statistical policies are difficult and cumbersome to obtain and update, because: bandwidth and latency related data must be collected for each server, bandwidths and latencies depend upon time-of-day and day-of-week, repeated measurements exhibit large skew, long tails, and high variability, and estimators go stale when network or server resources are upgraded.

Each of the three dynamic algorithms that we consider operates in two phases. In the first phase, after receiving a HTTP request, the algorithm sends a HTTP HEAD to every HTTP server supporting the service. This is obtained by having the DNS converting the service URL into the related server IP addresses and then opening a TCP/IP connection with every server. These HTTP HEADs are used to estimate the response time for each HTTP server, which depends on the current network traffic and the current server load. If properly set, the HTTP HEAD also returns the size of the Web page to download. On the basis of the HTTP HEAD responses, in the second phase the algorithm makes a decision and sends a HTTP GET. The contribution of our study is to compare the performance of the classical dynamic algorithm, which downloads the whole Web page from a single server, with the performance of two novel dynamic algorithms, which concurrently download different pieces of the same Web page from different servers.

2.1 Algorithm $A1$

The first algorithm [8, 4], hereafter referred to as $A1$, is the classical dynamic algorithm. For every HTTP request, $A1$ sends a HTTP HEAD to each of the HTTP servers in order to test their response times. As soon as one of the servers responds, $A1$ starts downloading the whole page from that server by sending it a HTTP GET.

From the timeliness viewpoint, $A1$ does not take into account the fact that the congestion of the connection to the chosen server and the load of the chosen server can increase while downloading the page. From the reliability viewpoint, $A1$ is vulnerable to failures of the connection to the chosen server or of the server itself.

2.2 Algorithm $A2$

The second algorithm [6], hereafter referred to as $A2$, tries to improve on $A1$ by concurrently downloading different pieces of the same Web page from different HTTP servers. Like $A1$, $A2$ initially sends a HTTP HEAD to each of the HTTP servers in order to test their response times. As soon as all of the servers have responded, $A2$ fragments the Web page to be downloaded into as many subpages as there are available servers, based on the principle that the faster a server is, the greater the size of the subpage that will be downloaded from that server. Formally, it there are n available servers with speeds $s_1, \ldots, s_n$ Kbytes/msec, respectively, and the size of the Web page to be retrieved is z Kbytes, then the size of the fragment requested to server i, $1 \leq i \leq n$, is given by

$$z_i = z \cdot s_i / \sum_{j=1}^{n} s_j \text{ Kbytes}$$

After the computation above, $A2$ concurrently downloads different pieces of the requested Web page from all of the available servers by sending to each of them a HTTP GET.

Although $A2$ fully realizes a distribution of the load among the available servers, which should reduce the user response time w.r.t. $A1$, $A2$ suffers from problems similar to those of $A1$ because it is vulnerable to increasing load or failures of the connection to the initially available servers or of the servers themselves.

2.3 Algorithm $A3$

The third algorithm [6], hereafter referred to as $A3$, tries to improve on $A2$ by periodically recomputing the size of the Web subpages in order to adapt to the changing conditions of the connections and the HTTP servers. $A3$ operates in $1 + m$ phases. In the first phase, $A3$ sends a HTTP HEAD to each of the HTTP servers in order to test their response times. As soon as all of the servers have responded, $A3$ fragments the Web page to be downloaded into as many subpages as there are available servers, based on the same principle as $A2$, with the difference that their sizes are computed according to the duration p of a monitoring period. Formally, m monitoring periods of p msec are necessary to concurrently donwload a Web page of z Kbytes from all of the available servers. If at the beginning of monitoring period k, $1 \leq k \leq m$, there are n_k available servers with speeds $s_{k,1}, \ldots, s_{k,n}$ Kbytes/msec, respectively, then the size of the fragment requested to server i, $1 \leq i \leq n$, via a HTTP GET is given by

$$z_{k,i} = s_{k,i} \cdot p \text{ Kbytes}$$

where each $s_{k,i}$ is measured on the basis of the actual number of Kbytes downloaded from server i in the previous period. If server i provides the requested $z_{k,i}$ Kbytes before monitoring period k terminates, i.e. after $0 < p' < p$ msec have elapsed, then further

$$z'_{k,i} = (z_{k,i}/p') \cdot (p - p') \text{ Kbytes}$$

are requested to it via a HTTP GET, where $z_{k,i}/p'$ is the new speed of server i and $p - p'$ is the remaining duration of monitoring period k. If instead server i does not provide the requested $z_{k,i}$ Kbytes by the end of monitoring period k, at the beginning of monitoring period $k + 1$ no new HTTP GET is sent to it; this will be sent when server i terminates, say $0 < p' < p$ msec after the beginning of period $k' > k$, and at that point further

$$z'_{k',i} = (z_{k,i}/(p \cdot (k' - k) + p')) \cdot (p - p') \text{ Kbytes}$$

are requested to it. The number m of monitoring periods necessary to download the whole Web page depends upon z, p, and the number and responsiveness of the servers.

From the timeliness viewpoint, $A3$ realizes an adaptive distribution of the load among the available servers by taking into account the fact that the congestion of the connection to the available servers and the load of the available servers can vary while downloading the page. From the reliability viewpoint, $A3$ can redistribute the download tasks assigned to failed servers (which do not respond within a certain number of monitoring periods) among the remaining available servers.

3 ÆMPA/TwoTowers Technology

In this section we give a brief overview of the ÆMPA/TwoTowers technology employed to analyze the performance of the three dynamic server selection algorithms introduced in the previous section.

ÆMPA [2] is an architectural description language built on top of the compositional algebraic language $\text{EMPA}_{\text{gr,vp}}$ [3, 1]. A description in ÆMPA is an architectural type (AT). As shown in Table 1, each AT is defined as a function of its architectural element types (AETs), its topology, its architectural interactions (AIs), and its generic parameters. An AET is in turn defined as a function of its behavior, specified either as a family of $\text{EMPA}_{\text{gr,vp}}$ sequential terms or through an invocation of a previously specified AT, and its interactions, specified as a set of $\text{EMPA}_{\text{gr,vp}}$ action types. The architectural topology consists of a fixed set of architectural element instances (AEIs) related by a fixed set of attachments among their interactions. AIs are interactions of AEIs that act as interactions for the whole AT, thus supporting hierarchical architectural modeling. Finally, generic parameters are values for parametric rates and weights.

To clarify how ÆMPA works, we provide the architectural description of a pipe-filter system. Such a system consists of three processing elements with identical structure – a server with a one position buffer – but different service rates. After leaving the first processing element, an item is immediately forwarded by the pipe to one of the other two processing elements with a certain routing probability.

The first part of the ÆMPA description of the system defines the name of the AT and its parameters:

$$\begin{aligned} \textbf{archi\_type} \quad & PipeFilter(FilterT, PipeT; \\ & PipeFilterI; \\ & \text{rate } service\_rate_0, \\ & \text{rate } service\_rate_1, \\ & \text{rate } service\_rate_2, \\ & \text{weight } routing\_prob) \end{aligned}$$

The specification above indicates that the AT named $PipeFilter$ relies on two AETs, $FilterT$ and $PipeT$, a set $PipeFilterI$ of AIs, and four generic parameters that represent the service rates of the three processing elements and the routing probability.

archi_type	⟨name and parameters⟩
archi_elem_types	⟨architectural element types: behaviors and interactions⟩
archi_topology	⟨architectural topology: architectural element instances and attachments⟩
archi_interactions	⟨architectural interactions⟩
end	

Table 1. ÆMPA textual notation

The second part of the description defines the AETs that have been specified to be parameters of the AT in the first part. Each AET is defined through a family of $\text{EMPA}_{\text{gr,vp}}$ sequential terms (or an invocation of a previously specified AT) expressing its behavior and a set of $\text{EMPA}_{\text{gr,vp}}$ action types expressing its interactions:

archi_elem_types
  elem_type $FilterT(Filter;$
    $FilterI;$
    rate $\mu)$
    behavior
      $Filter \triangleq <accept\_item, *>.Filter'$
      $Filter' \triangleq <accept\_item, *>.Filter'' +$
        $<serve\_item, \mu>.Filter$
      $Filter'' \triangleq <serve\_item, \mu>.Filter'$
    interactions $FilterI = accept\_item,$
      $serve\_item$
  elem_type $PipeT(Pipe;$
    $PipeI;$
    weight $p)$
    behavior
      $Pipe \triangleq <accept\_item, *>.$
        $(<forward\_item_1, \infty_{1,p}>.Pipe +$
        $<forward\_item_2, \infty_{1,1-p}>.Pipe)$
    interactions $PipeI = accept\_item,$
      $forward\_item_1,$
      $forward\_item_2$

The specification above indicates that AET $FilterT$ has the behavior described by term $Filter$, which represents a processing element with capacity two and service rate μ, and the interactions described by action type set $FilterI$. Initially, the filter can only accept an item from the outside, thus evolving into $Filter'$. If an item is in the buffer ($Filter'$), then either another item from the outside is accepted or the item in the buffer is served. If two items are in the buffer ($Filter''$), no more items can be accepted; the filter can only serve one of the waiting items. Similarly, AET $PipeT$ has the behavior described by term $Pipe$ and the interactions described by action type set $PipeI$. The pipe repeatedly accepts an item and forwards it along one of two different routes, with probabilities p and $1-p$ respectively.

The third part of the description defines the topology of the AT in terms of instances of the previously introduced AETs and related attachments among their interactions:

archi_topology $F_0 : FilterT(service\_rate_0)$
  $F_1 : FilterT(service\_rate_1)$
  $F_2 : FilterT(service\_rate_2)$
  $P : PipeT(routing\_prob)$
  $F_0.serve\_item$ **to** $P.accept\_item$
  $F_1.accept\_item$ **to** $P.forward\_item_1$
  $F_2.accept\_item$ **to** $P.forward\_item_2$

The attachments above specify that an instance P of $PipeT$ is connected to an upstream instance F_0 of $FilterT$ and two downstream instances F_1, F_2 of $FilterT$.

Finally, the fourth part defines the AIs. They are interactions of the previously defined AEIs that are used when defining the behavior of an AET of another AT by invoking the AT being specified, i.e. when plugging the AT at hand in the context of a larger system:

archi_interactions $PipeFilterI = F_0.accept\_item,$
  $F_1.serve\_item,$
  $F_2.serve\_item$

The AIs above support hierarchical modeling, and may be exploited e.g. in the description of a client-server system in which the server has the pipe-filter architecture above.

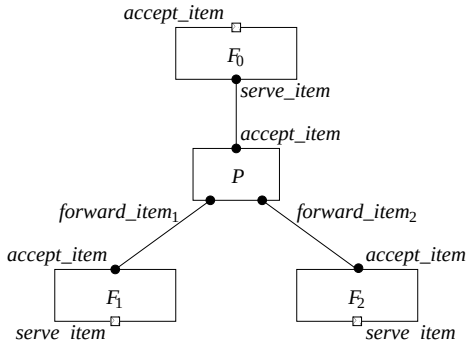

Figure 1. Graphical description of $PipeFilter$

$PipeFilter$ can be pictorially represented through the graphical notation of ÆMPA. This results in the flow graph of Fig. 1, where boxes denote AEIs, black circles/white squares denote interactions/AIs, and edges between ports denote attachments.

The semantics for ÆMPA is formally defined by translation into $\text{EMPA}_{\text{gr,vp}}$, by composing in parallel the sequential terms describing the behavior of the AEIs, with the synchronization sets being defined according to the declared attachments. Based on such a semantics, ÆMPA is equipped with all the functional verification and performance evaluation techniques of $\text{EMPA}_{\text{gr,vp}}$. Additionally, specific system architecture level checks have been introduced – architectural compatibility, architectural interoperability, and architectural conformity – to detect mismatches and possibly identify their causes.

ÆMPA specifications can be analyzed by means of an enhanced version of the software tool TwoTowers [1]. TwoTowers requires the user to provide: (i) a specification at the architectural level of the system to be designed/studied in ÆMPA, (ii) a description of the functional requirements to be met by the system in an ÆMPA companion language based on temporal logic, and (iii) a representation of the QoS metrics of interest for the system in a further ÆMPA companion language based on rewards. Besides a graphical user interface and a compiler for ÆMPA to $\text{EMPA}_{\text{gr,vp}}$ and then to state transition graphs, from the analysis standpoint TwoTowers is endowed with a kernel for conducting architectural coordination checks, a kernel for the verification of functional requirements (mostly based on the model checking technique), and a kernel for the evaluation of performance metrics (through both Markovian analysis and simulation).

4 Modeling the Three Algorithms

The three dynamic server selection algorithms for replicated Web services introduced in Sect. 2 have been modeled through the graphical notation of ÆMPA as shown in Fig. 2. In the scenario of Fig. 2 the dynamic server selection algorithm is implemented in the client browser. There are ten AEIs: the client, the algorithm, four Internet connections between the algorithm and the servers, and four HTTP servers.

The client has one parameter represented by the size distribution of the Web page to be downloaded. The client browser forwards each received Web page request to the algorithm and waits for the page to be retrieved. The algorithm, which has a parameter represented by the monitoring period only in the case of $A3$, sends a HTTP HEAD to each HTTP server and on the basis of the responses it decides how to download the page by sending the necessary HTTP GETs. The network connections have the same structure and may differ only for the distribution of the transmission delay. Likewise, the HTTP servers have the same structure and may differ only for their utilization. Due to lack of space, the behavior of each AET is not reported here.

It is worth pointing out that the behavior of the network connection AET and the behavior of the server AET have been modeled just once and then invoked four times each in the scenario above, thus achieving a high degree of specification reuse. Even more important from the point of view of saving time during the modeling process is the fact that the architectural scenario has been represented only once for each of the three algorithms. We also observe that ÆMPA has allowed us to easily represent the combination of two architectural styles, which are the layered organization composed of the client browser and the algorithm and the overall client-server organization. We conclude by mentioning the fact that all of the three ÆMPA specifications have passed the architectural compatibility check, which guarantees that they are deadlock free. In particular, such a check has allowed us to discover a mismatch in an early ÆMPA model of *A1*, which caused unexpected responsiveness results.

5 Responsiveness Comparison

The ÆMPA specifications of the three dynamic server selection algorithms illustrated in the previous section have been analyzed through the simulation routine of TwoTowers, in order to assess the responsiveness perceived by a single client in different configurations. Every simulation experiment for a given configuration has consisted of 20 simulation runs, each considering 100 Web page requests issued by the client. The simulation results have been obtained with 90% confidence level, where the extremes of each confidence interval are within 10% of the corresponding estimated value. The 96 simulation experiments have been conducted under the Solaris operating system on a PC with a 800 MHz CPU and a 768 Mbyte RAM.

In the scenario of Fig. 2 four HTTP servers are considered because we wanted to contrast our simulation results with responsiveness values measured on the field during some experiments in which the client was located at the Department of Computer Science of the University of Bologna, the first server was located at the Laboratory for Computer Science of the University of Bologna in Cesena (4 hops from the client), the second server was located at the International Center of Theoretical Physics in Trieste (9 hops), the third server was located at the Department of Computing Science of the University of Newcastle (15 hops), and the fourth server was located at the Computer Science Department of the University of California at San Diego (19 hops) [6]. Note that the four considered HTTP servers constitute a good combination of short, middle and long distances. In the simulation experiments, it is assumed that no link/server failure occurs.

For the four HTTP servers above, the network delays and the server utilizations have been set as follows. As far as transmission delays are concerned, we have chosen for all the Internet connections to the HTTP servers a Gaussian

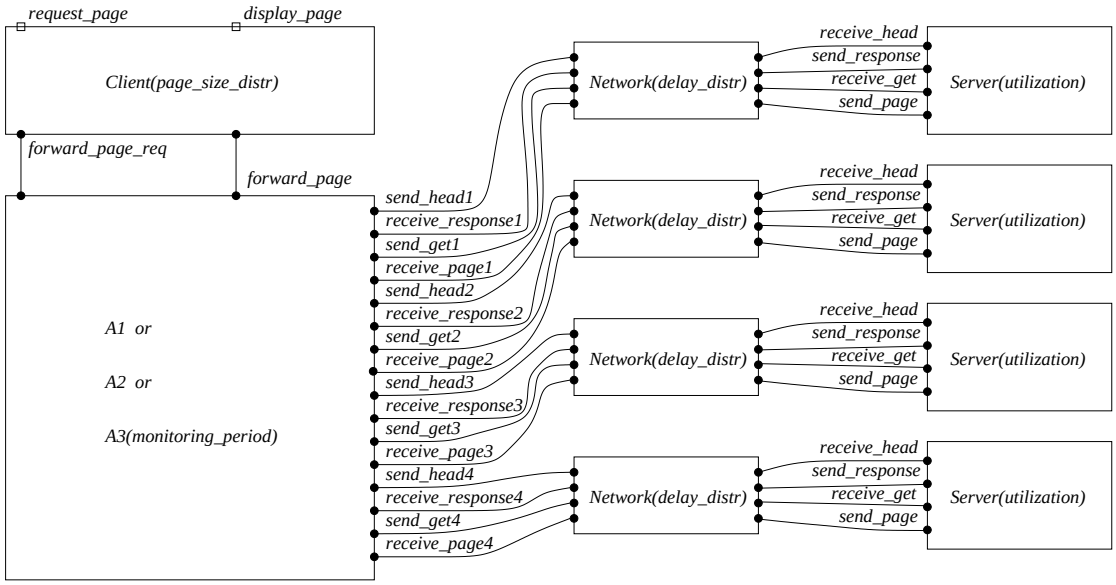

Figure 2. ÆMPA graphical description of the replicated Web service

distribution with mean μ and standard deviation σ. The two parameters of such a distribution vary from server to server and have been estimated on the field through the ping command for a MTU of 1.5 Kbytes: $\mu = 9$ msec and $\sigma = 1$ msec for Cesena, $\mu = 27$ msec and $\sigma = 8$ msec for Trieste, $\mu = 35$ msec and $\sigma = 5$ msec for Newcastle, and $\mu = 100$ msec and $\sigma = 10$ msec for San Diego. Given these figures for the basic transmission delay of each server, the transmission delay at the HTTP level for x Kbytes from a given HTTP server is obtained from the basic transmission delay of that server by assuming an increment equal to 50% of the basic transmission delay for each additional slot of 1.5 Kbytes of x [10]. The utilization of each HTTP server has been fixed to 80%. Assuming an exponential arrival process with rate 0.01 reqs/msec, the utilization above determines a service rate of 0.0125 reqs/msec. The inverse of the service rate is taken to be the service time for an average size Web page request, so the service time for an actual Web page request is determined by properly scaling the previous value.

The size of the requested Web pages is assumed to follow a heavy tailed distribution in order to fit the degree of variability of empirically measured workloads. More precisely, we have used a bounded Pareto distribution and we have conducted simulation experiments for two pairs of bounds – 50..5000 Kbytes, with an average size of 150 Kbytes (small pages), and 500..5000 Kbytes, with an average size of 1000 Kbytes (large pages) – for three different values of the parameter of the distribution – 1.8 (low variability), 1.5, and 1.2 (high variability). Moreover, we have considered ten different values for the monitoring period of $A3$: 300, 600, 900, 1200, 1500, 1800, 2100, 2400, 2700, and 3000 msec.

In Fig. 3 we report the responsiveness of the three dynamic server selection algorithms for 50..5000 Kbyte bounds (small pages). For the sake of completeness, the first figure shows the responsiveness achieved by using a single server: each curve is labeled with the name of the corresponding server. As expected, Cesena achieves the best performance, followed by Trieste, Newcastle, and San Diego. The second figure shows that if algorithm $A1$ is used, we obtain a responsiveness that is worse than that of Cesena alone, because of the overhead due to the initial HTTP HEADs, but better than that of the other three servers when considered in isolation. If $A2$ is used instead, we observe a responsiveness comparable to that of Newcastle in isolation from 1.8 to 1.5, with the response times dramatically increasing from 1.5 to 1.2. The reason for this is that the overhead introduced by $A2$ (which has to wait for the probe response from all the servers) cannot be amortized in this configuration, in which most of the requested Web pages are very small. The last two figures show the responsiveness of $A3$ for different durations of the monitoring period. The performance of $A3$ improves when increasing the duration of the monitoring period, as too frequent checks result in a big overhead, and tends to the performance of $A2$ ($A2$ can be viewed as a version of $A3$ in which the duration of the monitoring period is infinite).

In Fig. 4 we report the responsiveness of the three algorithms for 500..5000 Kbyte bounds (large pages). In this configuration we see that $A2$ and $A3$ perform much better as here their overhead can be amortized over bigger requested Web pages. In particular, $A2$ outperforms $A1$ and $A3(3000)$ is outperformed only by Cesena. Here the reason

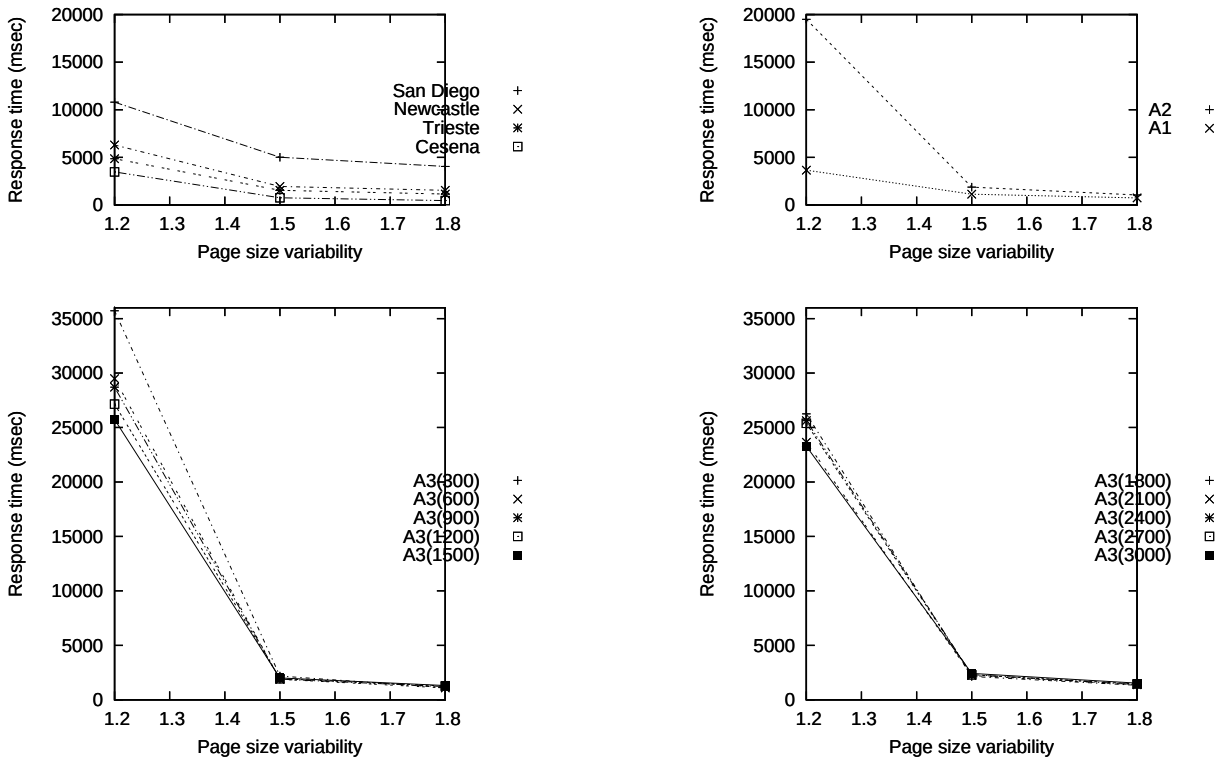

Figure 3. Responsiveness measured for 50..5000 Kbyte bounds

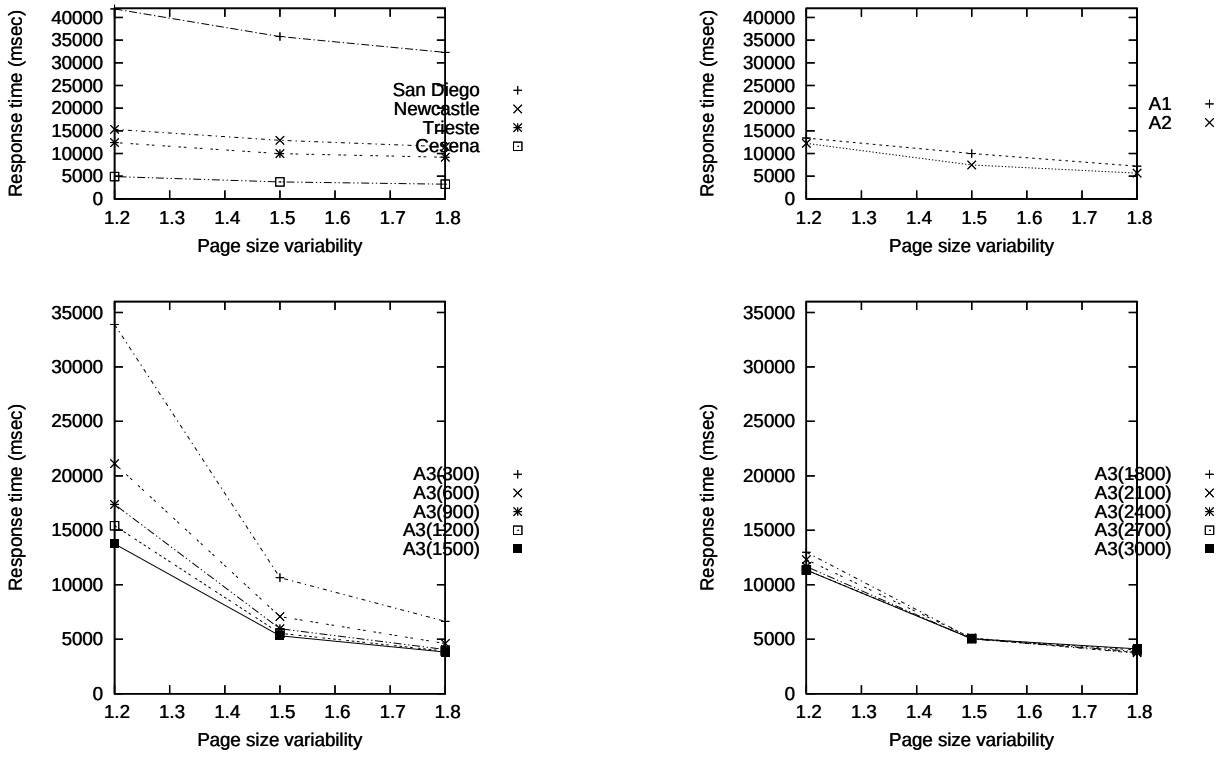

Figure 4. Responsiveness measured for 500..5000 Kbyte bounds

is that the Internet connection between Bologna and Cesena has a bandwith of 34 Mbps, i.e. it is essentially as if the client and the server were connected by a LAN within the same building. This suggests that, in the situation in which one of the servers is at least one order of magnitude faster than the others, it is more convenient – from the timeliness viewpoint – to resort to such a server only. On the contrary, when the servers have comparable speeds, the load distribution algorithms are effective.

A reason for resorting to $A3$ also in the case of a fast server is to achieve a greater service availability. In fact, our simulation experiments have been conducted under the assumption of absence of link/server failures. Should such failures be taken into account in the simulation experiments, the results concerning the responsiveness of the single servers, $A1$, and $A2$ would be much worse than those reported in the figures above, while the degradation of the performance of $A3$ would be more limited, as the available servers could dynamically take care of the downloading tasks assigned to the unreachable/failed servers. We also note that the responsiveness of $A3$ critically depends on the duration of the monitoring period, which must be set according to the speed of the slowest server in order to avoid the overhead due to exceedingly frequent checks.

We conclude by mentioning the fact that some of the curves in Fig. 3 and Fig. 4 have been compared with the corresponding curves in [6], which describe the results obtained by conducting some experiments on the field. The outcome of the comparison is a full qualitative agreement, in the sense that the relationships among the responsiveness of the contrasted curves are preserved. On the quantitative side, the figures in this paper show better results than those achieved on the field, the reason being that the experiments described in [6] were conducted some months earlier with a much slower connection between Bologna and Cesena.

6 Conclusion

In this paper we have conducted a case study on the performance of three dynamic server selection algorithms for responsive Web services, with two of them sharing the novel characteristic of concurrently downloading different fragments of the same Web page from different servers.

From the point of view of the examined algorithms, first we have found out that they are effective – for timeliness related purposes – only in the case in which none of the servers is much faster than the others, because in such a case the overhead introduced by the algorithms is not compensated for by the load distribution. Second, the simulation results have confirmed that $A3$ is the algorithm that achieves the best responsiveness among the considered ones, but only in the case of large requested Web pages. Third, we have discovered that the performance of $A3$ heavily depends on the duration of its monitoring period.

From the point of view of the employed technology, the modeling process has been greatly simplified by the use of a compositional, hierarchical and visual formalism like ÆMPA. In particular, it is worth mentioning the high degree of specification reusability, which has allowed us to model the replicated Web service only once for the three algorithms. Furthermore, the architectural coordination kernel of TwoTowers has helped us in verifying the correctness of the ÆMPA specifications.

Acknowledgements

We would like to thank Vittorio Ghini, Fabio Panzieri, and Marco Roccetti for the valuable discussions. This research has been funded by Progetto MURST Cofinanziato SALADIN.

References

[1] M. Bernardo, *"Theory and Application of Extended Markovian Process Algebra"*, Ph.D. Thesis, University of Bologna (Italy), 1999 (`http://www.di.unito.it/~bernardo/`)

[2] M. Bernardo, P. Ciancarini, L. Donatiello, *"ÆMPA: A Process Algebraic Description Language for the Performance Analysis of Software Architectures"*, in Proc. of WOSP 2000, ACM Press, pp. 1-11, Ottawa (Canada), 2000

[3] M. Bravetti, M. Bernardo, *"Compositional Asymmetric Cooperations for Process Algebras with Probabilities, Priorities, and Time"*, in Proc. of MTCS 2000, ENTCS 39(3), State College (PA), 2000

[4] M. Conti, E. Gregori, F. Panzieri, *"Load Distribution among Replicated Web Servers: A QoS-Based Approach"*, in WISP 1999, ACM Press, Atlanta (GA), 1999

[5] S.G. Dykes, K.A. Robbins, C.L. Jeffery, *"An Empirical Evaluation of Client-Side Server Selection Algorithms"*, in IEEE INFOCOM 3:1361-1370, 2000

[6] V. Ghini, F. Panzieri, M. Roccetti, *"Client-Centered Load Distribution: A Mechanism for Constructing Responsive Web Services"*, in HICSS 34, IEEE-CS Press, Maui (Hawaii), 2001

[7] M. Harchol-Balter, M.E. Crovella, C.D. Murta, *"On Choosing a Task Assignment Policy for a Distributed Server System"*, in Parallel and Distributed Computing 59:204-228, 1999

[8] D. Ingham, S.K. Shrivastava, F. Panzieri, *"Constructing Dependable Web Services"*, in IEEE Internet Computing 4:25-33, 2000

[9] M. Sayal, Y. Breitbart, P. Scheuermann, R. Vingralek, *"Selection Algorithms for Replicated Web Servers"*, in Performance Evaluation Review 26:44-50, 1998

[10] W.R. Stevens, *"TCP/IP Illustrated"*, Addison-Wesley, 1994