Dealer: Application-aware Request Splitting for Interactive Cloud Applications

Mohammad Hajjat[†] Shankaranarayanan P N[†] David Maltz[‡]
Sanjay Rao[†] Kunwadee Sripanidkulchai[⋆]
[†]Purdue University, [‡]Microsoft Corporation, [⋆]NECTEC Thailand

ABSTRACT

Deploying interactive applications in the cloud is a challenge due to the high variability in performance of cloud services. In this paper, we present *Dealer*– a system that helps geo-distributed, interactive and multi-tier applications meet their stringent requirements on response time despite such variability. Our approach is motivated by the fact that, at any time, only a small number of application components of large multi-tier applications experience poor performance. *Dealer* abstracts application structure as a component graph, with nodes being application components and edges capturing inter-component communication patterns. *Dealer* continually monitors the performance of individual component replicas and communication latencies between replica pairs. In serving any given user request, *Dealer* seeks to minimize user response times by picking the best combination of replicas (potentially located across different data-centers). While *Dealer* does require modifications to application code, we show through integration with two multi-tier applications that the changes required are modest. Our evaluations on two multi-tier applications using real cloud deployments indicate the $90\%ile$ of application response times could be reduced by a factor of 3 under natural cloud dynamics compared to conventional data-center redirection techniques which are agnostic of application structure.

Categories and Subject Descriptors

C.4 [**Performance of systems**]: Design studies; Reliability, availability, and serviceability; Modeling techniques; C.2.3 [**Computer communication networks**]: Network operations—*Network management; Network monitoring*

Keywords

Cloud Computing, Interactive Multi-tier Applications, Request Redirection, Geo-distribution, Service Level Agreement (SLA), Performance Variability

1 Introduction

Cloud computing promises to reduce the cost of IT organizations by allowing them to purchase as much resources as needed, only when needed, and through lower capital and operational expense stemming from the cloud's economies of scale. Further, moving to the cloud greatly facilitates the deployment of applications across multiple geographically distributed data-centers. Geo-distributing applications, in turn, facilitates service resilience and disaster recovery, and could enable better user experience by having customers directed to data-centers close to them.

While these advantages of cloud computing are triggering much interest among developers and IT managers [40, 21], a key challenge is meeting the stringent *Service Level Agreement (SLA)* requirements on availability and response times for interactive applications (e.g. customer facing web applications, enterprise applications). Application latencies directly impact business revenue [13, 9]– e.g., Amazon found every $100ms$ of latency costs 1% in sales [9]. Further, the SLAs typically require bounds on the $90th$ (and higher) percentile latencies [29, 12].

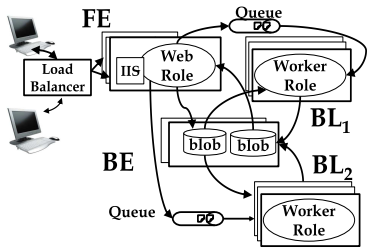
Meeting such stringent SLA requirements is a challenge given outages in cloud data-centers [1, 10], and the high variability in the performance of cloud services [44, 33, 23]. This variability arises from a variety of factors such as the sharing of cloud services across a large number of tenants, and limitations in virtualization techniques [44]. For example, [33] showed that the $95\%ile$ latencies of cloud storage services such as tables and queues is 100% more than the median values for four different public cloud offerings.

In this paper, we argue that it is critically important to design applications to be intrinsically resilient to cloud performance variations. Our work, which we term *Dealer*, is set in the context of geo-distributed, multi-tier applications, where each component may have replicas in multiple data-centers. *Dealer* enables applications to meet their stringent SLA requirements on response times by finding the combination of replicas –potentially located across multiple data-centers– that should be used to serve any given request. This is motivated by the fact that only a small number of application components of large multi-tier applications experience poor performance at any time.

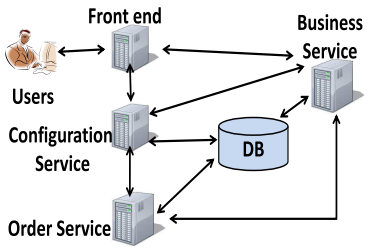
Multi-tier applications consist of potentially hundreds of components with complex inter-dependencies and hundreds of different transactions all involving different subsets of components [28]. Detailed knowledge of the components involved in every single type of transaction is hard to obtain. Instead, *Dealer* abstracts application structure as a component graph, with nodes being application components and edges capturing inter-component communication patterns. To predict which combination of replicas can result in the best performance, *Dealer* continually monitors the performance of individual component replicas and communication latencies between replica pairs.

Operating at a component-level granularity offers *Dealer* several advantages over conventional approaches that merely pick an ap-

CoNEXT'12, December 10–13, 2012, Nice, France.



(a) Thumbnails application architecture and data-flow. The application is composed of a Front-End (FE), Back-End (BE), and two Business-Logic components BL_1 (creates thumbnail) and BL_2 (creates Gray-scale, rotating images).



(b) *StockTrader* architecture and data-flow. Components include a front-end (FE), Business Server (BS), Order Service (OS) (handles buys/sells), Database (DB), and a Config Service (CS) that binds all components. The precise data-flow depends on transaction type.

Figure 1: Applications Testbed.

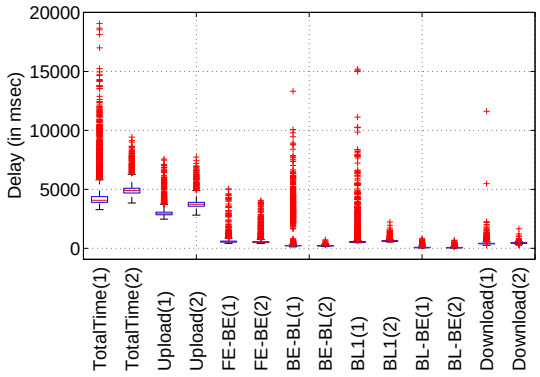


Figure 2: Box plot for total response time, and contributing processing and communication delays for Thumbnail application.

propriate data-center to serve user requests [26, 39, 45, 35]. Modern web applications consist of many components, not all of which are represent in each data-center, and the costs are extremely high to over-provision each component in every data-center to be able to handle all the traffic from another data-center. *Dealer* is able to redistribute work away from poorly performing components by utilizing the capacity of all component replicas that can usefully contribute to reducing the latency of requests.

While much of the *Dealer* design is independent of the particular application, integrating *Dealer* does require customization using application-specific logic. First, stateful applications have constraints on which component replicas can handle a given request. While *Dealer* proposes desired split ratios (or probabilities with which a request must be forwarded to different downstream component replicas), the application uses its own logic to determine which component replicas can handle a given request. Further, application developers must properly instrument their applications to collect the per-component performance data needed for *Dealer*. However, in our experience, the work required by application developers is modest.

We have evaluated *Dealer* on two stateful multi-tier applications on Azure cloud deployments. The first application is data-intensive, while the second application involves interactive transaction processing. Under natural cloud dynamics, using *Dealer* improves application performance by a factor of 3 for the 90^{th} and higher delay percentiles, compared to DNS-based data-center-level redirection schemes which are agnostic of application structure. Overall, the results indicate the importance and feasibility of *Dealer*.

2 Performance and Workload Variability

In this section, we present observations that motivate *Dealer*'s design. In §2.1, we characterize the extent and nature of the variability in performance that may be present in cloud data-centers. Our characterization is based on our experiences running multi-tier applications on the cloud. Then, in §2.2, we characterize the variability in workloads of multi-tier applications based on analyzing web server traces of a large campus university.

2.1 Performance variability in the cloud

We measure the performance variability with two applications. The first application, *Thumbnail* [18] involves users uploading a picture to the server (FE) and getting back either a thumbnail (from BL1) or a rotated image (from BL2). The second application, *StockTrader* [3], is an enterprise web application that allows users to buy/sell stocks, view portfolio, etc. Figure 1(a) and Figure 1(b) respectively show the component architecture and data-flow for each application.

We ran each application simultaneously in two separate data-centers (DC1 and DC2), both located in the United States, and subjected them to the same workload simultaneously. More details of how we configured the deployments are presented in §5.1. We instrumented each application to measure the total response time, as well as the delays contributing to total response time. The contributing delays include processing delays encountered at individual application components, communication delay between components (internal data-center communication delays), and the upload/download delays (Internet communication delays between users and each data-center).

We now present our key findings:

Performance of component replicas in multiple data centers is not correlated: Figure 3 shows a two hour snapshot from an experiment comparing the latency of database(DB) transactions for *StockTrader* across two consecutive days. The figure shows that the DB latency for DC1 on Day1 is significantly higher than on Day2, and has more prominent variation. The figure also shows that on Day1, the DB in DC2 performed significantly better than the DB in DC1 on the same day. This illustrates that the performance of similar components across multiple data-centers is not correlated. Further investigation revealed that the performance variability was due to high load on the data-center during a 9 day period [15]. Our interaction with the cloud providers indicated that during this period, different subsets of databases were impacted at different time snapshots.

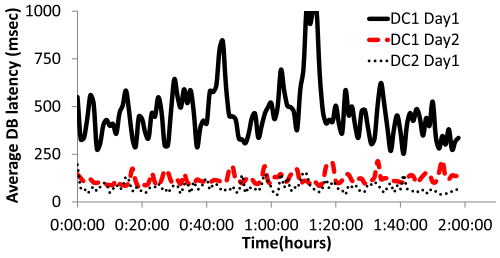


Figure 3: Comparing the latency of DB transactions in DC1 and DC2 across two consecutive days. The curve for DC2 Day2 is very similar to DC2 Day1 and is therefore omitted.

All application components show performance variability: Figure 2 considers the *Thumbnail* application and presents a box plot for the total response time (first two) and each of the individual contributing delays for each data-center. The X-axis is annotated

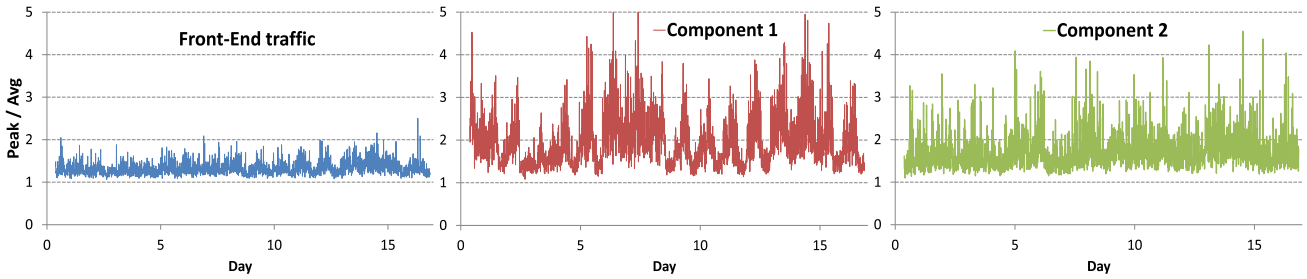


Figure 4: Short-term variability in workload for three components in a multi-tier web-service deployed in a large campus network. The peak and average rates are computed during each 10 minutes window and the *peak-to-average* ratio over each window is plotted as a function of time.

with the component or link whose delay is being measured and the number in parenthesis represents the data-center to which it belongs (DC1 or DC2). For example, BL-BE(1) represents the delay between the Business-Logic (BL) and the Back-End (BE) instances, at DC1. The bottom and top of each box represent the 25^{th} and 75^{th} percentiles, and the line in the middle represents the median. The vertical line (whiskers) extends to the highest datum within 3*IQR of the upper quartile, where IQR is the inter-quartile range. Points larger than this value are considered outliers and shown separately.

The figure shows several interesting observations. First, there is significant variability in all delay values. For instance, while the $75\%ile$ of total response time is under 5 seconds, the outliers are almost 20 seconds. Second, while the median delay with DC1 is smaller than DC2, DC1 shows significantly more variability. Third, while the Internet upload delays are a significant portion of total response time (since the application involves uploading large images), the processing delays at BL, and the communication delays between the BE and BL show high variability, and contribute significantly to total response times. Our experiments indicate that the performance of the application components vary significantly with time, and is not always correlated with the performance of their replicas in other data-centers.

2.2 Workload Dynamics

We now show the nature and extent of short-term variability in workload for multi-tier applications and the implications on cloud deployments.

While cloud computing allows for dynamic invocation of resources during peak periods, starting up new server instances takes several minutes (typically 10 minutes) in many commercial cloud deployments today. Further, besides the time to provision new instances, it may take even longer to warm up a booted server, e.g., by filling its cache with relevant data to meet its SLA. Therefore, applications typically maintain *margins* (i.e., pools of servers beyond the expected load [30, 20, 41]) to handle fluctuations in workload.

To examine workload variability and margin requirements for multi-tier applications, we collect and study the workload of a web-service in a large campus network. In the service, all requests enter through a front-end, which are then directed to different downstream components based on the type of request (e.g., web, mail, department1, department2, etc.). Figure 4 illustrates the variability in workload for the front-end and two downstream components. While the *peak-to-average* ratio is around 1.5 for the front-end, it is much higher for each of the other components, and can be as high as 3 or more during some time periods. The figure indicates that a significant margin may be needed even in cloud deployments to handle shorter-term workload fluctuations.

Furthermore, the figure not only illustrates the need for margins with cloud deployments, but also shows the heterogeneity in margin that may be required for different application tiers. While the margin requirement is about 50% for the front-end, it is over 300% for the other components during some time periods. In addition, the figure also illustrates that the exact margin required even for the same component is highly variable over time.

The high degree of variability and heterogeneity in margins make it difficult to simply over-provision an application component on the cloud since it is complicated to exactly estimate the extent of over-provisioning required, and over-provisioning for the worst-case scenario could be expensive. Moreover, failures and regular data-center maintenance actions make the application work with lower margins and render the application vulnerable to even modest workload spikes.

3 *Dealer* Design Rationale

In this section, we present the motivation behind *Dealer*'s design, and argue why traditional approaches don't suffice. *Dealer* is designed to enable applications meet their SLA requirements despite performance variations of cloud services. *Dealer* is motivated by two observations: (i) in any data-center, only instances corresponding to a small number of application components see poor performance at any given time; and (ii) the latencies seen by instances of the same component located in different data-centers are often uncorrelated.

Dealer's main goal is to dynamically identify a replica of each component that can best serve a given request. *Dealer* may choose instances located in different data-centers for different components, offering a rich set of possible choices. In doing so, *Dealer* considers performance and loads of individual replicas, as well as intra- and inter-data-center communication latencies.

Dealer is distinguished from DNS-based [26, 39, 45] and server-side [35] redirection mechanisms, which are widely used to map users to appropriate data-centers. Such techniques focus on alleviating performance problems related to Internet congestion between users and data-centers, or coarse-grained load-balancing at the granularity of data-centers. *Dealer* is complementary and targets performance problems of individual cloud services inside a data-center. There are several advantages associated with the *Dealer* approach:

• *Exploit heterogeneity in margins across different components:* In large multi-tier applications with potentially hundreds of components [28], only a few services might be temporarily impacted in any given data-center. *Dealer* can reassign work related to these services to other replicas in remote data-centers if they have sufficient margins. For instance, *Dealer* could tackle performance problems with storage elements (e.g., a blob) by using a replica in a remote data-center, while leveraging compute instances locally. Complete request redirection, however, may not be feasible since instances of other components (e.g., business-logic servers) in the remote data-center may not be over-provisioned adequately over their normal load to handle the redirected requests. In fact, Figure 4 shows significant variation in workload patterns of individual

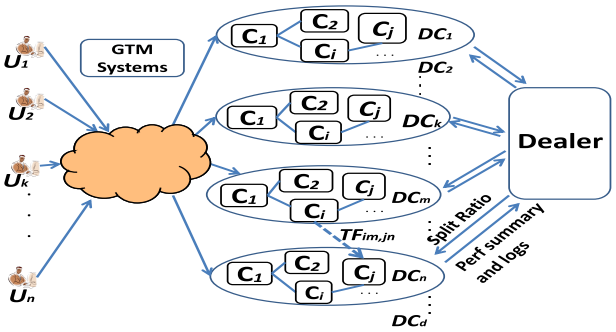


Figure 5: System overview

components of multi-tier applications, indicating the components must be provisioned in a heterogeneous fashion.

• *Utilize functional cloud services in each data-center:* *Dealer* enables applications to utilize cloud services that are functioning satisfactorily in all data-centers, while only avoiding services that are performing poorly. In contrast, techniques that redirect entire requests fail to utilize functional cloud services in a data-center merely due to performance problems associated with a small number of other services. Further, the application may be charged for the unutilized services (for example, they may correspond to already pre-paid reserved compute instances [2]). While *Dealer* does incur additional inter data-center communication cost, our evaluations in §5.5.4 indicate these costs are small.

• *Responsiveness:* Studies have shown that DNS-based redirection techniques may have latencies of over 2 hours and may not be well suited for applications which require quick response to link failures or performance degradations [34]. In contrast, *Dealer* targets adaptations over the time-scale of tens of seconds.

4 System Design

In this section we present the design of *Dealer*. We begin by presenting an overview of the design, and then discuss its various components.

4.1 System Overview

Consider an application with multiple components $\{C_1..C_l\}$. We consider a multi-cloud deployment where the application is deployed in d data-centers, with instances corresponding to each component located in every one of the data-centers. Note that there might be components like databases which are only present in one or a subset of data-centers. We represent all replicas of component C_i in data-center m as C_{im}.

Traffic from users is mapped to each data-center using standard mapping services used today based on metrics such as geographical proximity or latencies [39]. Let U_k denote the set of users whose traffic is mapped to data-center k. We refer to data-center k as the primary data-center for U_k, and to all other data-centers as the secondary data-centers. The excess capacity of each component replica is the additional load that can be served by that replica which is not being utilized for the primary traffic of that data-center. Traffic corresponding to U_k can use the entire available capacity of all components in data-center k, as well as the excess capacity of components in all other data-centers.

For each user group U_k, *Dealer* seeks to determine how application transactions must be split in the multi-cloud deployment. In particular, the goal is to determine $TF_{im,jn}$, that is the number of user transactions that must be directed between component i in data-center m to component j in data-center n, for every pair of <component,data-center > combinations. In doing so, the objective is to ensure the overall delay of transactions can be minimized. Further, *Dealer* periodically recomputes how application transactions must be split given dynamics in behavior of cloud services.

Complex multi-tier applications may have hundreds of different transactions all involving different subsets of application components. Detailed knowledge of the components involved in every single type of transaction is hard to obtain. Instead, *Dealer* dynamically learns a model of the application that captures component interaction patterns. In particular, *Dealer* estimates the fraction of requests that involve communication between each pair of application components, and the average size of transactions between each component pair. In addition, *Dealer* estimates the processing delays of individual components replicas, and communication delays between components, as well as the available capacity of component replicas in each data-center, (i.e., the load each replica can handle). We will discuss how all this information is estimated and dynamically updated in the later subsections.

4.2 Determining delays

There are three key components to the estimation algorithms used by *Dealer* when determining the processing delay of components and communication delays between them. These include: (i) passive monitoring of components and links over which application requests are routed; (ii) heuristics for smoothing and combining multiple estimates of delay for a link or component; and (iii) active probing of links and components which are not being utilized to estimate the delays that may be incurred if they were used. We describe each of these in turn:

Monitoring: Monitoring distributed applications is a well studied area, and a wide range of techniques have been developed by the research community and industry [22, 27, 7] that can be used for measuring application performance. Of these techniques, *X-Trace* [27] is the most suitable for our purposes, since it can track application performance at the granularity of individual requests. However, integrating the monitoring code with the application is a manual and time consuming process. To facilitate easy integration of *X-Trace* with the application, we automate a large part of the integration effort using *Aspect Oriented Programming (AOP)* techniques [5]. We write an *Aspect* to intercept each function when it is called and after it returns, which constitutes the *pointcuts*. We record the respective times inside the Aspect. The measured delay values are then reported periodically to a central monitor. A smaller reporting time ensures greater agility of *Dealer*. We use reporting times of 10 seconds in our implementation, which we believe is reasonable.

Smoothing delay estimates: It is important to trade-off the agility in responding to performance dips in components or links with potential instability that might arise if the system is overly aggressive. To handle this, we use a weighted moving average (WMA) scheme. For each link and component, the average delay seen during the last W time windows of observation is considered. The weighted average of these values is then computed according to the following formula:

$$D(t) = \frac{\sum_{i=1}^{W}(W-i+1)*D(t-i)*N(t-i)}{\sum_{i=1}^{W}(W-i+1)*N(t-i)} \quad (1)$$

Briefly, the weight depends on the number of samples seen during a time window, and the recency of the estimate (i.e., recent windows are given a higher weight). $D(t)$ is the delay seen by a link/component in Window t, and $N(t)$ is the number of delay samples obtained in that window (to ensure higher weight for windows with more transactions). The use of WMA ensures that *Dealer* reacts to prolonged performance episodes that last several seconds, while not aggressively reacting to extremely short-lived problems within a window. W determines the number of windows for which

a link/component must perform poorly (well) for it to be avoided (reused). Our empirical experience has shown choosing W values between 3 and 5 are most effective for good performance.

Probing: *Dealer* uses active probes to estimate the performance of components and links that are not currently being used. This enables *Dealer* to decide if it should switch transactions to a replica of a component in a different data-center, and determine which replica must be chosen. Probe traffic is generated by test-clients using application workload generators (e.g., [8]). We restrict active probes to read-only requests that do not cause changes in persistent application state. While this may not accurately capture the delays for transactions involving writes, we have found the probing scheme to work well for the applications we experiment with. We also note that many applications tend to be read-heavy and it is often more critical to optimize latencies of transactions involving reads.

To bound probes' overhead, we limit the probe rate to 10% of the application traffic rate. *Dealer* biases the probes based on the quality of the path. In particular, the probability P_i that a path is probed is given as:

$$P_i = \frac{CR_i}{\sum_j CR_j} \tag{2}$$

Here, CR_i is the compliance ratio– the fraction of requests that use a given path with a response time lower than its SLA. The intuition is that a path that has generally been good might temporarily suffer poor performance. Biasing the probing ensures that such a path is likely to be probed more frequently, which ensures *Dealer* can quickly switch back to it when its performance improves. Also, *Dealer* probes 5% of the paths at random to ensure more choices can be explored. In the initialization stage, *Dealer* probes paths in a random fashion. As an enhancement, *Dealer* can bias probing during the initialization phase based on coarse estimates of link delays. Such coarse estimates can be obtained based on the size of transactions exchanged between components (obtained through monitoring application traffic) and the bandwidth between data-centers.

While probing may add a non-negligible overhead on applications, we are investigating ways to restrict our use of active probing to only measuring inter-data-center latency and bandwidth. The key insights behind our approach are to (i) use passive user-generated traffic to update component processing delays and inter-component link latencies [1]; and (ii) limit active probes to measuring inter-data-center latency and bandwidth. These measurements can then be combined, along with passive measurements on transaction sizes observed between components, to estimate the performance of any combination. Further, rather than having each application measure the bandwidth and latency between every pair of data-centers, cloud providers could provide such services in the future, amortizing the overheads across all applications. We leave further exploration of this as future work.

4.3 Determining transaction split ratios

In this section, we discuss how *Dealer* uses the processing delays of components and communication times of links to compute the split ratio matrix **TF**. Here, $TF_{im,jn}$ is the number of user transactions that must be directed between component i in data-center m to component j in data-center n, for every <component, data-center > pair. In determining the split ratio matrix, *Dealer* considers several factors including i) the total response time; ii) stability of the overall system; and iii) capacity constraints of application components.

In our discussion, a *combination* refers to an assignment of each component to exactly one data-center. For e.g., in Figure 5, a mapping of C_1 to DC_1, C_2 to DC_k, C_i to DC_m and C_j to DC_m represents a combination. The algorithm iteratively assigns a fraction of transactions to each combination. The split ratio matrix is easily computed once the fraction of transactions assigned to each combination is determined. We now present the details of the assignment algorithm:

Considering total response time: *Dealer* computes the mean delay for each possible combination like in [28]. It is the weighted sum of the processing delays of nodes and communication delay of links associated with that combination. The weights are determined by the fraction of user transactions that traverse that node or link. Specifically, consider a combination where component i is assigned to data-center $d(i)$. Then, the mean delay of that combination is:

$$\sum_i \sum_j f_{ij} * D_{id(i),jd(j)} \tag{3}$$

Here, $D_{id(i),jd(j)}$ denotes the communication delay between component i in data-center $d(i)$, and component j in data-center $d(j)$. When $i = j$, D represents the processing delay of component i. Further, f_{ij} denotes the fraction of transactions that involve an interaction between application components i and j, and f_{ii} denotes the fraction of transactions that are processed at component i. The fractions f_{ij} may be determined by monitoring the application in its past window like in § 4.2. Once the delays of combinations are determined, *Dealer* sorts the combinations in ascending order of mean delay such that the best combinations get utilized the most, thereby ensuring a better performance.

Ensuring system stability: To ensure stability of the system and prevent oscillations, *Dealer* avoids abrupt changes in the split ratio matrix in response to minor performance changes. To achieve this, *Dealer* limits the maximum fraction of transactions that may be assigned to a given combination. The limit (which we refer to as the damping ratio) is based on how well that combination has performed relative to others, and how much traffic was assigned to that combination in the recent past. In particular, the damping ratio (DR) for each combination is calculated periodically as follows:

$$\begin{aligned} DR(L_i,t) &= \tfrac{W(L_i,t)}{\sum_k W(L_k,t)}, where \\ W(L_i,t) &= \textstyle\sum_{\ell=0}^{W-1} Rank(L_i,t-\ell) * Req(L_i,t-\ell) \end{aligned} \tag{4}$$

Here, $Rank(L,t)$ is the ranking of combination L at the end of time window t (the lower the value of mean delay, the higher the ranking). $Req(L,t)$ is the number of requests sent on combination L during t. The algorithm computes the weight of a combination based on its rank and the requests assigned to it in each of the last W windows. Similar to §4.2, we found that W values between 3 and 5 results in the best performance.

Honoring capacity constraints: In assigning transactions to a combination of application components, *Dealer* ensures the capacity constraints of each of the components is honored as described in Algorithm 1. *Dealer* considers the combinations in ascending order of mean delay (line 8). It then determines the maximum fraction of transactions that can be assigned to that combination without saturating any component (lines 9-11). *Dealer* assigns this fraction of transactions to the combination, or the damping ratio, whichever is lower (line 12). The available capacities of each component and the split ratio matrix are updated to reflect this assignment (lines 14-16). If the assignment of transactions is not completed at this point, the process is repeated with the next best combination (lines 17-18).

[1]We expect each data-center to continually receive some traffic which would ensure such passive observations are feasible.

Algorithm 1 Determining transaction split ratios.

procedure COMPUTESPLITRATIO()
  Let $C[i,m]$ be the capacity matrix, with each cell (i,m) corresponding to capacity of component C_{im} (component i in data-center m), calculated as in §4.4
  Let $AC[i,m]$ be the available-capacity matrix for C_{im} Initialized as $AC[i,m] \leftarrow C[i,m]$
  Let $T[i,j]$ be the transaction matrix, with each cell (i,j) indicating the number of transactions per second between application components i and j
  Let T_i be the load on each component ($\sum_j T_{ji}$)
  Let *FA* be fraction of transactions that has been assigned to combinations. Initialized as FA $\leftarrow 0$
  Goal: Find $TF[im, jn]$: the number of transactions that must be directed between C_{im} and C_{jn}
  Foreach combination L, sorted by mean delay values
    For each C_{im} in L
      $f_i \leftarrow \frac{AC[i,m]}{T_i}$
    $min_f \leftarrow \min_{\forall i}(f_i)$
    $ratio = \min(min_f, DR(L,t))$
    Rescale damping ratios if necessary
    For each C_{im} in L
      $AC[i,m] \leftarrow AC[i,m] - ratio * T_i$
    $TF[id(i), jd(j)] \leftarrow TF[id(i), jd(j)] + ratio * T_{ij}, \forall i,j$
    $FA \leftarrow FA + ratio$
    Repeat until FA = 1
end procedure

Algorithm 2 Dynamic capacity estimation.

1: **procedure** COMPUTETHRESH(T, D)
2: **if** $D > 1.1 * DelayAtThresh$ **then**
3: **if** $T <= Thresh$ **then**
4: $LowerThresh \leftarrow 0.8 * T$
5: $ComponentCapacity \leftarrow Thresh$
6: **else**
7: $Thresh \leftarrow unchanged$
8: $ComponentCapacity \leftarrow Thresh$
9: **end if**
10: **else if** $D <= DelayAtThresh$ **then**
11: **if** $T >= Thresh$ **then**
12: $Thresh \leftarrow T$
13: $ComponentCapacity \leftarrow T + 5\% of T$
14: **else**
15: $Thresh \leftarrow unchanged$
16: $ComponentCapacity \leftarrow Thresh$
17: **end if**
18: **end if**
19: **end procedure**

4.4 Estimating capacity of components

We now discuss how *Dealer* determines the capacity of components in terms of the load each component can handle. Typically, application delays are not impacted by an increase in load up to a point which we term as the *threshold*. Beyond this, application delays increase gradually with load, until a breakdown region is entered where vastly degraded performance is seen. Ideally, *Dealer* must operate at the threshold to ensure the component is saturated while not resulting in degraded performance. The threshold is sensitive to transaction mix changes. Hence, *Dealer* relies on algorithms for dynamically estimating the threshold, and seeks to operate just above the threshold.

Dealer starts with an initial threshold value based on a conservative stress test assuming worst-case load (i.e., transactions that are expensive for each component to process). Alternately, the threshold can be obtained systematically (e.g., using *knee* detection schemes [37]) or learnt during boot-up phase of an application in the data-center, given application traffic typically ramps up slowly before production workloads are handled. Since the initial threshold can change (e.g., due to changes in transaction mix), *Dealer* dynamically updates the threshold using Algorithm 2. The parameter *DelayAtThresh* is the delay in the flat region learnt in the initialization phase, which is the desirable levels to which the component delay must be restricted. At all times, the algorithm maintains an estimate of *Thresh*, which is the largest load in recent memory where a component delay of *DelayAtThresh* was achieved. T and D represent the current transaction load on the component, and the delay experienced at the component respectively. The algorithm strives to operate at a point where D is slightly more than *DelayAtThresh*, and T slightly more than *thresh.* If *Dealer* operated exactly at *thresh,* it would not be possible to know if *thresh* has increased, and hence discover if *Dealer* is operating too conservatively.

The algorithm begins by checking if the delay is unacceptably high (line 2). In such case, if $T \leq Thresh$, (line 3) the threshold is lowered. Otherwise (line 6), the threshold remains unchanged and the component capacity is lowered to the threshold. If D is comparable to *DelayAtThresh* (line 10), it is an indication the component can take more load. If $T \geq Thresh$ (line 11), then the threshold is too conservative, and hence it gets increased. Further, *ComponentCapacity* is set to slightly higher than the threshold to experiment if the component can absorb more requests. If however $T < Thresh$, (line 14), then *ComponentCapacity* is set to *Thresh* to allow more transactions be directed to that component.

We note that the intuition behind the choice of parameters is to increase the load the component sees by only small increments (5%) but back-off more aggressively (by decreasing the load in each round by 20%) in case the delay starts increasing beyond the desired value. We believe the choice of parameters is reasonable; however, we defer testing the sensitivity of the algorithm to these parameters as a future work. Finally, while component delays were used as a mean of estimating if the component is saturated, one could also use other metrics such as CPU, memory utilization and queues sizes.

Algorithm 3 Integration with stateful applications.

Original code:

```
procedure SENDREQUEST(Component cmp, Request req)
    Replica replica ← cmp.Replica
    replica.Send(req)
end procedure
```

With *Dealer*:

```
procedure SENDREQUEST(Component cmp, Request req)
    Replica replica ← metaData[req.ID][cmp]
    if replica is null then                          ▷ Not in meta-data.
        replica ← GetDealerReplica(cmp) ▷ Use Dealer suggestion.
        if cmp is stateful then   ▷ Cmp is stateful but its information
has not been propagated yet in meta-data.
            metaData[req.ID][cmp] ← replica
        end if
    end if
    replica.Send(req)
end procedure
```

4.5 Integrating *Dealer* with applications

We integrated *Dealer* with both *Thumbnail* and *StockTrader*, and we found that the overall effort involved was small. Integrating *Dealer* with applications involves: i) adding logic to re-route requests to replicas of a downstream component across different data-centers; and ii) maintaining consistent state in stateful applications.

Re-routing requests. To use *Dealer*, application developers need to make only a small change to the *connection logic* – the code segment inside a component responsible for directing requests to downstream components. *Dealer* provides both push and pull API's for retrieving split ratios (§4.3). Instead of forwarding all requests to a single service endpoint, the connection logic now allocates requests to downstream replicas in proportion to the split ratios provided by *Dealer*.

Integration with stateful applications. While best practices emphasize that cloud applications should use stateless services whenever possible [6, 4], some applications may have stateful components. In such cases, the application needs to affinitize requests to component replicas so that each request goes to the replicas that hold the state for processing the request. Integrating *Dealer* with such applications does not change the consistency semantics of the application. *Dealer* does not try to understand the application's policy for allocating requests to components. Instead, it proposes the desired split ratios to the application, and the application uses its own logic to determine which replicas can handle a request.

In integrating *Dealer* with stateful applications, it is important to ensure that related requests get processed by the same set of stateful replicas due to data consistency constraints. For instance, the *StockTrader* application involves session state. To integrate *Dealer*, we made sure all requests belonging to the same user session use the same combination, and *Dealer*'s split-ratios only determine the combination taken by the first request of that session. *StockTrader* persists user session information (users logged in, session IDs, etc.) in a database. We modified the application so that it also stores the list of stateful replicas for each session. We also note that some web applications maintain the session state in the client side through session cookies. Such information could again be augmented to include the list of stateful replicas.

To guarantee all requests within the same session follow the same combination, the application must be modified to propagate *meta-data* (such as a unique session ID and the list of stateful replicas associated with it) along all requests between components. Many web applications (such as *StockTrader*) use SOAP and RESTful services that provide *Interceptors* which can be easily used to propagate meta-data with very minimal modifications. In the *StockTrader* application, we used *SOAP Extensions* [16] to propagate meta-data. In other cases where Interceptors cannot be used, endpoint interfaces can be changed or overloaded to propagate such data.

The propagated meta-data is used by components to guide the selection of downstream replicas. Algorithm 3 illustrates this. A component initiating a request must first check if the downstream component is stateful (by examining the meta-data), and if it is, it picks the replica specified in the meta-data. Otherwise, it picks the replica suggested by *Dealer*. If a downstream stateful component is visited for the first time, it picks the replica that *Dealer* suggests and saves this information into the meta-data which gets propagated along requests to the front-end.

While handling such state may require developer knowledge, we found this required only moderate effort from the developer in the applications we considered. As future work, we would like to integrate *Dealer* with a wider set of applications with different consistency requirements and gain more experience with the approach.

5 Experimental Evaluation

In this section, we evaluate the importance and effectiveness of *Dealer* in ensuring good performance of applications in the cloud. We begin by discussing our methodology in §5.1. We then evaluate the effectiveness of *Dealer* in responding to various events that occur naturally in a real cloud deployment (§5.2). These experiments both highlight the inherent performance variability in cloud environments, and evaluate the ability of *Dealer* to cope with them. We then evaluate *Dealer* using a series of controlled experiments which stress the system and gauge its effectiveness in coping with extreme scenarios such as sharp spikes in application load, failure of cloud components, and abrupt shifts in application transaction sizes.

5.1 Evaluation Methodology

We study and evaluate the design of *Dealer* by conducting experiments on *Thumbnail* and *StockTrader* (introduced in §2).

Cloud testbed and application workloads: All experiments were conducted on Microsoft Azure by deploying each application simultaneously in two data-centers located geographically apart in the U.S. (North and South Central). In all experiments, application traffic to one of the data-centers (referred to as DC_A) is controlled by *Dealer*, while traffic to the other one (DC_B) was run without *Dealer*. The objective was to not only study the effectiveness of *Dealer* in enhancing performance of traffic to DC_A, but also ensure that *Dealer* did not negatively impact performance of traffic to DC_B.

Application traffic to both data-centers was generated using a Poisson arrival process when the focus of an experiment is primarily to study the impact of cloud performance variability. Furthermore, to study the impact of workload dynamics, we also use real campus workload traces (described in §2.2) and conduct experiments that involve abrupt changes of the rate of the Poisson process. In *Thumbnail*, we set the transaction mix (fraction of requests to BL_1 and BL_2) according to the fraction of requests to Component1 and Component2 in the trace. Another key workload parameter that we did vary was the size of pictures uploaded by users. Requests in *Thumbnail* had an average upload size of 1.4 MB (in the form of an image) and around 3.2 (860) KB download size for BL_1 (BL_2) transactions. *StockTrader*, on the other hand, had a larger variety of transactions (buying/selling stocks, fetching quotes, etc.) with relatively smaller data size. To generate a realistic mix of transactions, we used the publicly available DaCapo benchmark [24], which contains a set of user sessions, with each session consisting of a series of requests (e.g., login, home, fetch quotes, sell stocks, and log out). A total of 66 PlanetLab users, spread across the U.S., were used to send requests to DC_A. Further, another set of users located inside a campus network were used to generate traffic to DC_B.

Application Deployments: Applications were deployed with enough instances of each component so that they could handle typical loads along with additional margins. We estimated the capacities of the components through a series of stress-tests. For instance, with an average load of $2\ \frac{req}{sec}$ and 100% margin (typical of real deployments as shown in §2), we found empirically that 2/5/16 instances of FE/BL_1/BL_2 components were required. Likewise, for *StockTrader*, handling an average load of $1\ \frac{req}{sec}$ ($0.25\ \frac{session}{sec}$) required 1/2/1 instances of FE/BS/OS.

In *StockTrader*, we deployed the DB in both data-centers and configured it in master-slave mode. We used SQL Azure Data Sync [14] for synchronization between the two databases. We note that *Dealer* can be integrated even if the application uses sharding or has weaker consistency requirements (§4.5) – the choice of master-slave is made for illustration purposes. While reads can occur at either DB, writes are made only at the master DB (DC_B). Therefore, transactions involving writes (e.g., buy/sell) can only occur through the BS and OS instances in DC_B. Thus, the BS component would see a higher number of requests (by $\approx 20\%$) than the FE and therefore requires higher provisioning than FE.

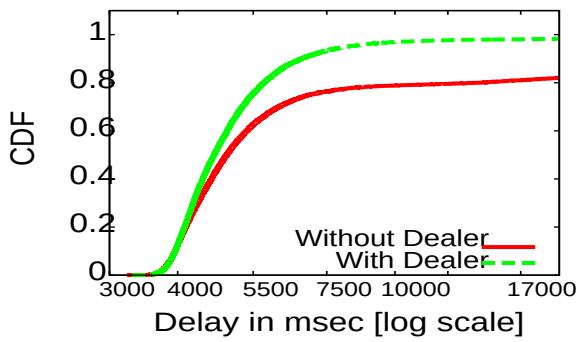


Figure 6: CDF of total response time under natural cloud dynamics.

Further, each component can only connect to its local CS and DB to obtain communication credentials of other components. Finally, all requests belonging to a user session must use the same set of components given the stateful nature of the application.

Comparison with existing schemes: We evaluate *Dealer* against two prominent load-balancing and redirection techniques used today:

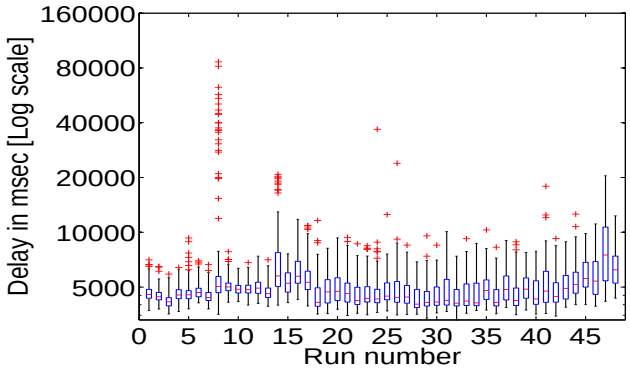
• *DNS-based redirection*: Azure provides *Windows Azure Traffic Manager (WATM)* [19] as its solution for DNS-based redirection. WATM provides *Failover*, *Round-Robin* and *Performance* distribution policies. Failover deals with total service failures and sends all traffic to the next available service upon failure. Round-robin routes traffic in a round-robin fashion. Finally, Performance forwards traffic to the closest data-center in terms of network latency. In our experiments, we use the Performance policy because of its relevance to *Dealer*. In WATM, requests are directed to a single URL which gets resolved through DNS to the appropriate data-center based on performance tables that measure the round trip time (RTT) of different IP addresses around the globe to each data-center. We believe WATM is a good representative of DNS-based redirection schemes for global traffic management. However, its redirection is based solely on network latency and is agnostic to application performance. We therefore compare *Dealer* with another scheme that considers overall application performance.

• *Application-level Redirection*: We implemented a per-request load-balancer, that we call *Redirection*, which re-routes each request as a single unit, served completely by a single data-center. *Redirection* re-routes requests based on the overall performance of the application, calculated as the weighted average of total response time (excluding Internet delays) across all transactions. If it finds the local response time of requests higher than that of the remote data-center, it redirects clients to the remote data-center by sending a `302 HTTP` response message upon receiving a client request. It re-routes requests as long as the remote data-center is performing better, or until capacity limits are reached remotely (limited by the capacity of lowest margin component). Similar to *Dealer*, re-routing in *Redirection* does not depend on transaction types. We use the same monitoring and probing infrastructure described in §4.2.

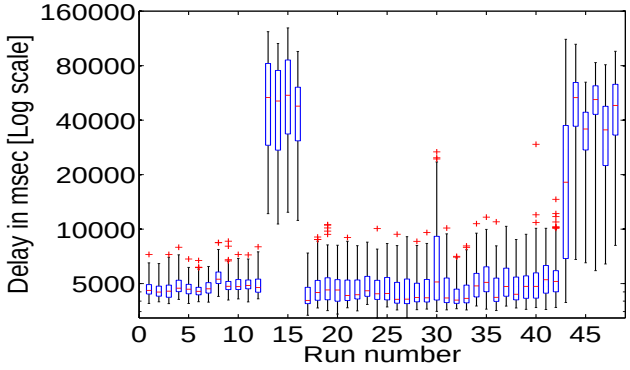
5.2 *Dealer* under natural cloud dynamics

In this section, we evaluate the effectiveness of *Dealer* in responding to the natural dynamics of real cloud deployments. Our goal is to explore the inherent performance variability in cloud environments and evaluate the ability of *Dealer* to cope with such variability.

We experiment with *Thumbnail* and compare its performance with and without *Dealer*. Ideally it is desirable to compare the two



(a) With *Dealer*.



(b) Without *Dealer*.

Figure 7: Box-plots of total response time under natural cloud dynamics.

schemes under identical conditions. Since this is not feasible on a real cloud, we ran a large number of experiments alternating between the two approaches. The experiment was 48 hours, with each hour split into two half-hour runs; one without activating *Dealer*, and another with it. Traffic was generated using a Poisson process with an average request rate of 2 $\frac{req}{sec}$ to each data-center.

Figure 6 shows the CDF of the total response time for the whole experiment. *Dealer* performs significantly better. The 50^{th}, 75^{th}, 90^{th}, and 99^{th} percentiles with *Dealer* are 4.6, 5.4, 6.6 and 12.7 seconds respectively. The corresponding values without *Dealer* are 4.9, 6.8, 43.2 and 90.9 seconds. The reduction is more than a factor of 6.5x for the top 10 percentiles.

Figure 7 helps understand why *Dealer* performs better. The figure shows a box-plot of total response time for each run of the experiment. The X-axis shows the run number and the Y-axis shows the total response time in milliseconds. Figure 7(a) shows the runs with *Dealer* enabled, and 7(b) shows the runs with *Dealer* disabled (i.e., all traffic going to DC_A stay within the data-center). In both figures, runs with the same number indicate that the runs took place in the same hour, back to back. The figures show several interesting observations:

• First, without *Dealer*, most runs had a normal range of total response time (median $\approx$ 5 seconds). However, the delays were much higher in runs 13-16 and 43-48. Further investigation showed these high delays were caused by the BL instances in DC_A, which had lower capacity to absorb requests during those periods, and consequently experienced significant queuing. Such a sudden dip

in capacity is an example of the kind of event that may occur in the cloud, and highlights the need for *Dealer*.

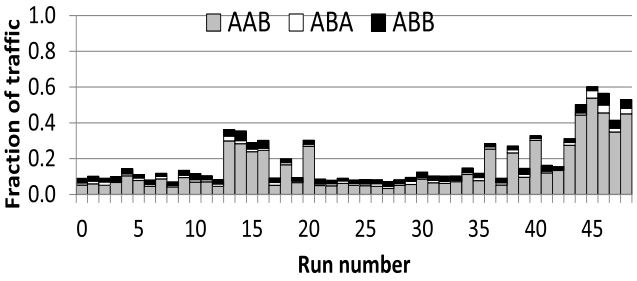


Figure 8: Fraction of *Dealer* traffic sent from DC_A to DC_B.

• Second, *Dealer* too experienced the same performance problem with BL in DC_A during runs 13-16 and 43-48. However, *Dealer* mitigated the problem by tapping into the margin available at DC_B. Figure 8 shows the fraction of requests directed to one or more components in DC_B by *Dealer*. Each bar corresponds to a run and is split according to the combination of components chosen by *Dealer*. Combinations are written as the location of FE, BE, BL_1 and BL_2 components[2] respectively, where A refers to DC_A and B to DC_B. For example, for run 0 around 9% of all requests handled by *Dealer* used one or more components from DC_B. Further, for this run, 5% of requests used the combination AAB, while 1% used ABA, and 3% used ABB. Further, most requests directed to DC_B during the problem take the path AAB, which indicates the BL component in DC_B is used.

• Third, we compared the performance when runs 13-16 and 43-48 are not considered. While the benefits of *Dealer* are not as pronounced, it still results in a significant improvement in the tail. In particular the 90^{th} percentile of total response time was reduced from 6.4 to 6.1 seconds, and the 99^{th} percentile was reduced from 18.1 to 8.9 seconds. Most of these benefits come from *Dealer*'s ability to handle transient spikes in workload by directing transactions to the BL replica in DC_B. There were also some instances of congestion in the blob of DC_A which led *Dealer* to direct transactions to the blob of DC_B.

• Finally, Figure 7(a) shows that the performance is not as good in run 8. Further inspection revealed that the outliers during this run were all due to the high upload delays of the requests directed to DC_B. This was likely due to Internet congestion between the users and DC_B. We note that such performance problems are not the focus of *Dealer*, and should rather be handled by schemes for Global Traffic Management such as DNS-based redirection [45, 26].

5.3 Reaction to changes in transactions size

Multi-tier applications show a lot of variability not only in request rates but also in the mix and size of transactions, as we discussed in §2. In this section, we evaluate the effectiveness of *Dealer* in adapting to changes in transactions size using the *Thumbnail* application. Using the same configuration described earlier, we change the size of images that users upload to DC_A from 860 KB to 1.4 MB during time 400 to 800, and reduce it back to 860 KB after that. Figure 9 shows the total response time, comparing the performance with and without *Dealer*. The performance without *Dealer* is significantly affected even by a moderate increase in image size. Further, although the problem lasted for only 400 seconds (6.6 minutes), it took the application without *Dealer* around 960 seconds (16 minutes) to recover after transaction sizes returned to normal due to the large build-up of queues. However, the performance with *Dealer* is good as the application could dynamically direct transactions to DC_B.

[2]Since all transactions in this experiment were of type BL_1, we drop the 4^{th} tuple.

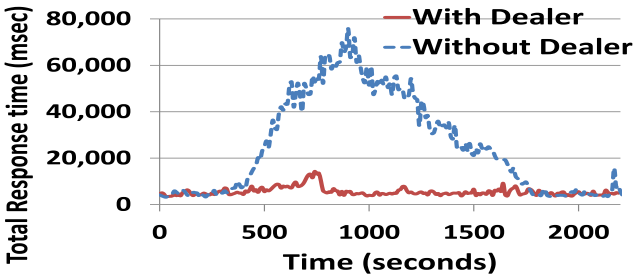


Figure 9: Performance under varying transaction size (*Thumbnail*).

5.4 *Dealer* vs. DNS-based redirection

Global Traffic Managers (GTM) are used to route user traffic across data-centers to get better application performance and cope with failures. We conducted an experiment with the same setup mentioned in §5.2 to compare *Dealer* against WATM (§5.1). Figure 10 shows that *Dealer* achieves a reduction of at least 3x times in total response time for the top 10 percentiles. Like before, we found the BL instances had lower capacity in some of the runs leading to a higher total response time in GTM. Since the GTM approach only takes into account the network latency and not the application performance, it was unable to react to performance problems involving the BL instances.

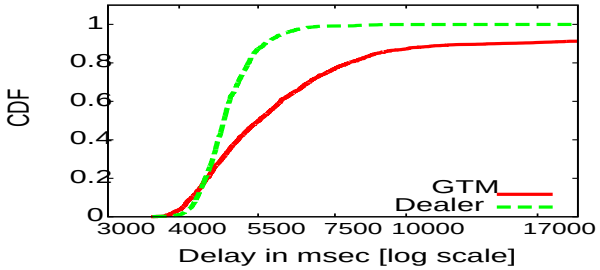


Figure 10: CDF of total response time for GTM vs. *Dealer* (*Thumbnail*).

5.5 *Dealer* vs. application-level redirection

In this section, we evaluate the effectiveness of *Dealer* in adapting to transient performance issues and compare its performance with application-level redirection described in §5.1.

5.5.1 Reaction to transient performance problems

We present our evaluation of *Dealer*'s response to performance variation in the cloud by deploying *StockTrader* at both data-centers, using the master-slave mode as described in §5.1. We emulate a performance degradation in the database (DB) at DC_A using the traces we collected during the DB performance issue in §2.1 by taking a 10 minutes period with high DB latency and using the corresponding data points to induce delay at the DB.

Figure 11 shows that during the period of performance degradation at the DB (9-18th and 27-36th min), the average response time of *Dealer* is significantly better than that of *Redirection*. Figure 11(b) shows that *Dealer* takes ABB and switches requests over

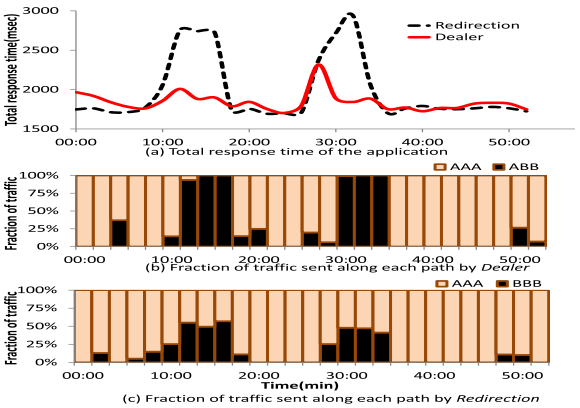


Figure 11: Performance of *Dealer* vs. *Redirection* using traces collected during the DB performance issue. A combination (FE, BS, OS) is represented using the data-center (DC_A or DC_B) to which each component belongs. 20% of transactions perform DB writes (combination ABB), hence we exclude them for better visualization.

to the BS and OS at DC_B to avoid the high latency at DB. Similarly, Figure 11(c) shows the path (BBB) taken by *Redirection* and how this scheme switches a fraction of the requests entirely to the data-center, DC_B. The fraction of traffic redirected to BBB in (c) is less than the fraction of traffic sent through ABB in (b). This is because *Dealer* is able to better utilize the margin available at the BS by switching a larger fraction of requests to the BS in DC_B. On the other hand, *Redirection* is constrained by the available capacity at the FE (DC_B) and hence is not able to completely utilize the margin available at the BS (DC_B).

5.5.2 Reaction to transient component overload

In this section, we evaluate *Thumbnail* under natural cloud settings using a real workload trace from §2. We use two intervals, each around 30 minutes long, and replay them on DC_A and DC_B simultaneously. The two intervals are about 4 hours away from each other, allowing us to simulate realistic load that may be seen by data-centers in different time-zones. We ran the experiment in a similar fashion to §5.2 for 5 hours alternating between *Dealer* and *Redirection*. We subjected BL_1 and BL_2 to the same request rate seen by Component1 and Component2. A total of 55 VM's were used to deploy the application in both data-centers. We picked the margin for each component as the average peak-to-average ratio during each interval. Margins ranged between 190% and 330%.

Figure 12 shows that the 90^{th}(99^{th}) percentiles for *Dealer* were 11.9 (14.4) seconds, compared to 13.3 (19.2) seconds for *Redirection*– a reduction of over 10.5% in response times for the top 10 percentiles. Further investigation revealed that this was due to a short-term overload affecting the BL_1 replica in DC_A. *Dealer* was able to mitigate the problem by splitting requests to BL_1 between its replicas in both data-centers. *Redirection*, on the other hand, could not re-direct all excess traffic to DC_B since BL_2 did not have sufficient capacity in the remote data-center to handle all the load from DC_A. Figure 13 shows that at times 400-500, BL_1 in DC_A experienced a surge in request rate exceeding its available margin. At the same time, BL_1(BL_2) in DC_B had a request rate that is lower(higher) than its average. These results highlight the importance and effectiveness of *Dealer*'s fine-grained component level mechanism in adapting to transient overload of individual components.

5.5.3 Reaction to failures in the cloud

Applications in the cloud may see failures which reduce their margins, making them vulnerable to even modest workload spikes.

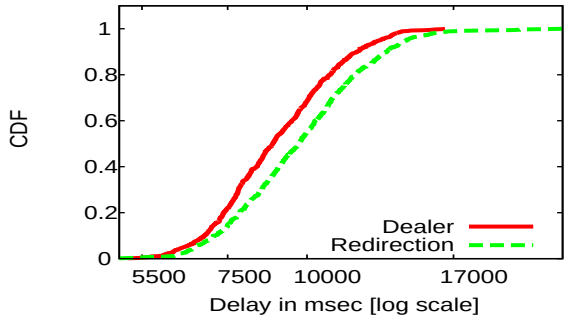


Figure 12: CDF of total response time of *Dealer* vs. *Redirection* using real workload trace under natural spikes and transaction mix changes (*Thumbnail*). Latencies with both schemes are higher than Figure 6 because transactions to BL_2 involve heavier processing (image rotation).

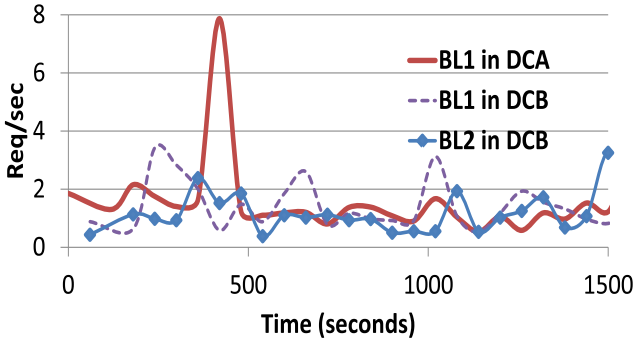


Figure 13: Request rate for each component in both data-centers. BL_2 in DC_A not shown for better visualization.

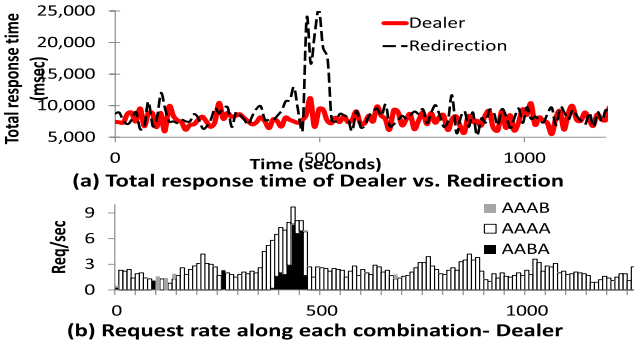


Figure 14: Performance of *Dealer* vs. *Redirection* using real workload trace with cloud failures (*Thumbnail*).

Failures can happen due to actual physical outages or due to maintenance and upgrades. For example, Windows Azure's SLA states that a component has to have 2 or more instances to get 99.95% availability [17] as instances can be taken off for maintenance and upgrades at any time.

In Figure 14, we reproduced the case of a single fault-domain failure at time 300 affecting BL_2 instances in DC_B[3]. The combination AABA represents requests which were served by FE, BE, BL_2 at DC_A and BL_1 at DC_B. For the same reasons described in §5.5.2, *Dealer* maintained a significantly lower response time during the surge in workload (130% lower). The results show that *Dealer* is effective in handling failures in the cloud.

[3]This involved bringing 4 BL_2 VM's offline since Azure deploys each component's VMs on 2 or more fault-domains.

5.5.4 Inter data-center bandwidth costs

A potential concern arises due to wide-area traffic that *Dealer* introduces in re-routing requests across data-centers. In this section, we compute the cost percentage increase for *Thumbnail* and *StockTrader* based on the experiments described in §5.5.1 and §5.5.2.

We consider the bandwidth, storage and compute (small instances) costs based on Microsoft Azure tariffs in January, 2012. The bandwidth cost is based on all transactions exiting each data-center (incoming transactions do not incur bandwidth costs in Azure). The average size of each request in *Thumbnail* (*StockTrader*) is 1.5MB (2 KB). *StockTrader* uses SQL Azure DB (Web Ed.) and *Thumbnail* uses Azure blobs for storage. We calculate the storage cost for *Thumbnail* based on the number of storage transactions and storage size consumed. The cost of the DB and compute instances is normalized to the duration of the experiments.

The cost percentage increase for *Thumbnail* and *StockTrader* were found to be 1.94% and 0.06% respectively. This shows that the cost introduced due to inter data-center bandwidth is minimal, even for data-intensive applications such as *Thumbnail*. We have repeated our calculations using the Amazon EC2 pricing scheme [2], and we have found similar results. Finally, we note that in our evaluations we assume compute instances in both data-centers cost the same. However, in practice, application architects are likely to provision *reserved instances* in each data-center [2] (i.e., instances contracted over a longer period for a lower rate). Under such scenarios, *Dealer* has the potential to incur lower costs than *Redirection* by leveraging reserved instances in each data-center to the extent possible.

6 Related Work

Several researchers have pointed out the presence of performance problems with the cloud (e.g., [44, 33, 23]). In contrast, our focus is on designing systems to adapt to short-term variability in the cloud.

The cloud industry already provides mechanisms to scale up or down the number of server instances in the cloud (e.g., [36, 11]). However, it takes tens of minutes to invoke new cloud instances in commercial cloud platforms today. Recent research has shown the feasibility of starting new VMs at faster time scales [32, 43]. For instance, [32] presents a VM-fork abstraction which enables the cloning of a VM into multiple replicas on-the-fly. While such schemes are useful for handling variability in performance due to excess load on a component, they cannot handle all types of dynamics in the cloud (e.g., problems in blob storage, network congestion, etc.). Further, ensuring the servers are warmed up to serve requests after instantiation (e.g., by filling caches, running checks, copying state, etc.) demands additional time. In contrast, *Dealer* can enable faster adaptation at shorter time-scales, and is intended to complement solutions for dynamic resource invocation.

DNS-based techniques [26, 39, 45] and server-side redirection mechanisms [35] are widely used to map users to appropriate data-centers. However, such techniques focus on alleviating performance problems related to Internet congestion between users and data-centers, and load-balance user traffic coarsely at the granularity of data-centers. In contrast, *Dealer* targets performance problems of individual cloud components inside a data-center, and may choose components that span multiple data-centers to service an individual user request. This offers several advantages in large multi-tier applications (with potentially hundreds of components [28]) where possibly only a small number of components are temporarily impacted. When entire user requests are redirected to a remote data-center as in [26, 39, 45, 35], not all components in the remote data-center may be sufficiently over-provisioned to handle the redirected requests. Further, redirecting entire user requests does not utilize functional resources in the local data-center that have already being paid for. For instance, the local data-center may have underutilized *reserved* instances [2], while the remote data-center might require the use of more expensive *on-demand* instances. The cost could be substantial over a large number of components. Finally, studies have shown that the use of DNS-based redirection techniques may lead to delays of more than 2 hours and thus may not be suitable for applications which require quick response to failures [34]. We note that [35] does mention doing the redirection at the level of the bottleneck component; however, *Dealer* is distinguished in that it makes no a priori assumption about which component is the bottleneck, and dynamically reacts to whichever component or link performs poorly at any given time.

Several works [38, 42, 46] study utility resource planning and provisioning for applications. [38] studies resource planning for compute batch tasks by building predictive models in shared computing utilities. Further, [42, 46] build analytic models for handling workload variability (changing transaction mix and load) in multi-tier applications. For example, [42] aims at handling peak workloads by provisioning resources at two levels; predictive provisioning that allocates capacity at the time-scale of hours or days, and reactive provisioning that operates at time scales of minutes. While such techniques are complementary to *Dealer*, their focus is not applications deployed in public clouds. *Dealer* not only deals with workload variability, but also handles all types of performance variability (e.g., due to service failures, network congestion, etc.) in geo-distributed multi-tier applications, deployed in commercial public clouds. *Dealer* provides ways to avoid components with poor performance and congested links via re-routing requests to replicas in other data-centers at short time scales.

Other works [31, 25] study the performance of multi-tier applications. [31] tries to control the performance of such applications by preventing overload using self-tuning proportional integral (PI) controller for admission control. Such a technique can be integrated with *Dealer* to control the load directed to each component replica. Further, [25] combines performance modeling and profiling to create analytical models to accomplish SLA decomposition. While SLA decomposition is outside the scope of *Dealer*, component profiling may be incorporated with *Dealer* to capture component's performance as a function of allocated resources (e.g., CPU) to achieve performance prediction.

7 Conclusions and Future Work

In this paper, we have shown that it is important and feasible to architect latency-sensitive applications in a manner that is robust to the high variability in performance of cloud services. We have presented *Dealer*, a system that can enable applications to meet their SLA requirements by dynamically splitting transactions for each component among its replicas in different data-centers. Under natural cloud dynamics, the $90th$ and higher percentiles of application response times were reduced by more than a factor of 3 compared to a system that used traditional DNS-based redirection. Further, *Dealer* not only ensures low latencies but also significantly outperforms application-level redirection mechanisms under a range of controlled experiments. Integrating *Dealer* with two contrasting applications only required a modest level of change to code.

As future work, we plan to explore and gain more experience integrating *Dealer* with a wider set of cloud applications with various consistency constraints. Further, we intend to study ways for reducing probing overhead by limiting active probes to measuring inter-data-center bandwidth and latency only. Finally, we will evaluate the performance of *Dealer* under scale and explore more cloud infrastructures (such as Amazon EC2 and Google App Engine).

8 Acknowledgments

This material is based upon work supported in part by the National Science Foundation (NSF) under Career Award No. 0953622 and NSF Award No. 1162333, and was supported by cloud usage credits from Microsoft Azure under NSF/Microsoft's Computing in the Cloud program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or Microsoft.
Finally, we thank the reviewers and our shepherd, Prashant Shenoy, for their feedback which helped us improve the paper.

9 References

[1] Amazon cloud outage. http://aws.amazon.com/message/2329B7/.

[2] Amazon EC2 pricing. http://aws.amazon.com/ec2/pricing/.

[3] Apache, Project Stonehenge. http://wiki.apache.org/incubator/StonehengeProposal.

[4] Architecting for the Cloud: Best Practices. http://jineshvaria.s3.amazonaws.com/public/cloudbestpractices-jvaria.pdf.

[5] Aspect Oriented Programming. http://msdn.microsoft.com/en-us/library/aa288717%28v=vs.71%29.aspx.

[6] Coding in the Cloud. Use a stateless design whenever possible. http://www.rackspace.com/blog/coding-in-the-cloud-rule-3-use-a-stateless-design-whenever-possible/.

[7] Event Tracing for Windows (ETW). http://msdn.microsoft.com/en-us/library/aa363668.aspx.

[8] Grinder Load Testing Framework. http://grinder.sourceforge.net/index.html.

[9] Latency - it costs you. http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it.

[10] Microsoft Live outage due to DNS corruption. http://windowsteamblog.com/windows_live/b/windowslive/archive/2011/09/20/follow-up-on-the-sept-8-service-outage.aspx.

[11] Microsoft Windows Azure. http://www.microsoft.com/windowsazure/.

[12] Response Time Metric for SLAs. http://testnscale.com/blog/performance/response-time-metric-for-slas.

[13] Slow pages lose users. http://radar.oreilly.com/2009/06/bing-and-google-agree-slow-pag.html.

[14] SQL Azure Data Sync. http://social.technet.microsoft.com/wiki/contents/articles/sql-azure-data-sync-overview.aspx.

[15] SQL Azure Performance Issue. http://cloudfail.net/513962.

[16] Using SOAP Extensions in ASP.NET. http://msdn.microsoft.com/en-us/magazine/cc164007.aspx.

[17] Windows Azure SLA. http://www.microsoft.com/windowsazure/sla/.

[18] Windows Azure Thumbnails Sample. http://code.msdn.microsoft.com/windowsazure/Windows-Azure-Thumbnails-c001c8d7.

[19] Windows Azure Traffic Manager (WATM). http://msdn.microsoft.com/en-us/gg197529.

[20] AHMAD, F., ET AL. Joint optimization of idle and cooling power in data centers while maintaining response time. *ASPLOS 2010*.

[21] ARMBRUST, M., ET AL. Above the Clouds: A Berkeley View of Cloud Computing. Tech. rep., EECS, University of California, Berkeley, 2009.

[22] BARHAM, P., ET AL. Magpie: Online modelling and performance-aware systems. In *HOTOS 2003*.

[23] BARKER, S., AND SHENOY, P. Empirical evaluation of latency-sensitive application performance in the cloud. In *MMSys 2010*.

[24] BLACKBURN, S. M., AND ET AL. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA 2006*.

[25] CHEN, Y., IYER, S., LIU, X., MILOJICIC, D., AND SAHAI, A. SLA decomposition: Translating service level objectives to system level thresholds. In *ICAC'07*.

[26] DILLEY, J., ET AL. Globally distributed content delivery. *Internet Computing, IEEE* (2002).

[27] FONSECA, R., PORTER, G., KATZ, R., SHENKER, S., AND STOICA, I. X-trace: A pervasive network tracing framework. In *NSDI 2007*.

[28] HAJJAT, M., ET AL. Cloudward bound: Planning for beneficial migration of enterprise applications to the cloud. *SIGCOMM 2010*.

[29] HASTORUN, D., ET AL. Dynamo: amazons highly available key-value store. In *In Proc. SOSP* (2007).

[30] HONG, Y.-J., ET AL. Dynamic server provisioning to minimize cost in an iaas cloud. In *ACM SIGMETRICS, 2011*.

[31] KAMRA, A., MISRA, V., AND NAHUM, E. Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites. In *IWQOS 2004*.

[32] LAGAR-CAVILLA, ET AL. SnowFlock: rapid virtual machine cloning for cloud computing. In *ACM EuroSys, 2009*.

[33] LI, A., YANG, X., KANDULA, S., AND ZHANG, M. CloudCmp: comparing public cloud providers. In *IMC 2010*.

[34] PANG, J., ET AL. On the responsiveness of DNS-based network control. In *IMC 2004*.

[35] RANJAN, S., KARRER, R., AND KNIGHTLY, E. Wide area redirection of dynamic content by Internet data centers. In *INFOCOM 2004*.

[36] RIGHTSCALE INC. Cloud computing management platform. http://www.rightscale.com.

[37] SATOPAA, V., ET AL. Finding a 'Kneedle' in a Haystack: Detecting Knee Points in System Behavior. In *SIMPLEX Workshop, 2011*.

[38] SHIVAM, P., BABU, S., AND CHASE, J. Learning application models for utility resource planning. In *ICAC'06*.

[39] SU, A., ET AL. Drafting behind Akamai. *SIGCOMM 2006*.

[40] SYMANTEC. 2010 State of the Data Center Global Data. http://www.symantec.com/content/en/us/about/media/pdfs/Symantec_DataCenter10_Report_Global.pdf.

[41] URGAONKAR, B., AND SHENOY, P. Cataclysm: Handling extreme overloads in internet services. In *PODC 2004*.

[42] URGAONKAR, B., SHENOY, P., CHANDRA, A., AND GOYAL, P. Dynamic provisioning of multi-tier internet applications. In *ICAC 2005*.

[43] VRABLE, M., ET AL. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *ACM SOSP, 2005*.

[44] WANG, G., AND NG, T. S. E. The impact of virtualization on network performance of Amazon EC2 data center. In *IEEE INFOCOM 2010*.

[45] WENDELL, P., ET AL. DONAR: decentralized server selection for cloud services. In *SIGCOMM 2010*.

[46] ZHANG, Q., CHERKASOVA, L., AND SMIRNI, E. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *ICAC'07*.