4-18-2011

Dealer: Dynamic Request Splitting for Performance-Sensitive Applications in Multi-Cloud Environments

Mohammad Hajjat
School of Electrical and Computer Engineering, Purdue University, hajjat@purdue.edu

Shankaranarayanan Narayanan
School of Electrical and Computer Engineering, Purdue University

David Maltz
Microsoft Research

Sanjay Rao
Electrical and Computer Engineering, Purdue University, sanjay@purdue.edu

Kunwadee Sripanidkulchai
NECTEC Thailand

Dealer: Dynamic Request Splitting for Performance-Sensitive Applications in Multi-Cloud Environments

Mohammad Hajjat

Shankaranarayanan Narayanan

David Maltz

Sanjay Rao

Kunwadee Sripanidkulchai

TR-ECE-11-10

April 18, 2011

School of Electrical and Computer Engineering

1285 Electrical Engineering Building

Purdue University

West Lafayette, IN 47907-1285

Dealer: Dynamic Request Splitting For Performance-Sensitive Applications in Multi-Cloud Environments

Mohammad Hajjat†, Shankaranarayanan Narayanan†, David Maltz‡
Sanjay Rao†, and Kunwadee Sripanidkulchai⋆
†Purdue University, ‡Microsoft Research, ⋆NECTEC Thailand

ABSTRACT

Enterprises are increasingly deploying their applications in the cloud given the cost-saving advantages, and the potential to geo-distribute applications to ensure resilience and better service experience. However, a key unknown is whether it it is feasible to meet the stringent response time requirements of enterprise applications using the cloud. We make several contributions. First, we show through empirical measurement studies that (i) there is significant short-term variability in application workload and response times of individual components; however (ii) the response times of the same component in different data-centers are often uncorrelated. This leads us to argue that there are potential latency savings if work related to a poorly performing component is dynamically reassigned to a replica in a remote data-center. We leverage this insight to build a system that we term *Dealer* which for each component, dynamically splits transactions among its replicas in different data-centers. In doing so, *Dealer* seeks to minimize user response times, and takes component performance, as well as intra-data-center and inter-data-center communication latencies into account. We have implemented *Dealer* in a way that it can be added to any multi-tier application. Evaluations of our approach on two multi-tier applications on actual Azure cloud deployments indicates the importance and feasibility of our mechanisms. For instance, the $90\%ile$ of application response times could be reduced by as much as 6 times under natural cloud dynamics.

1 Introduction

Cloud computing promises to reduce the cost of IT organizations by allowing them to purchase just as much compute and storage 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 located close to them. These attractive advantages of cloud computing are motivating a number of enterprises to explore how their applications could be deployed using the cloud [1, 16, 24, 9].

While cloud computing offers several advantages, a key challenge for enterprises is meeting service level agreements (SLAs) associated with their applications. Enterprise applications often have stringent requirements in terms of availability and response times. However, little is known about the variability in performance of cloud components and data-center links given that a cloud is a shared multi-tenant infrastructure on which users have very little control over what other tenants are doing.

In this paper, we take a first step towards understanding and addressing challenges associated with deploying latency-sensitive multi-tier enterprise applications in the cloud. Such applications are composed of multiple components often arranged as a pipeline (with potentially complex interactions). Each component is provisioned with enough servers to achieve acceptable SLA. The applications are typically geo-distributed with replicas of each component present in multiple data-centers.

We begin by presenting a measurement study of workloads seen in real web-services, and a study of performance seen by multi-tier applications when deployed in the cloud. Our results indicate that (i) there is significant short-term variability on load seen by application components both due to variations in workload, as well as types of transactions; and (ii) there is much variability in response times of individual components at shorter time-scales. Dynamic provisioning of new cloud resources [21, 4], is inadequate to tackle these shorter-term variations given the process could take tens of minutes.

Our measurements also reveal that the latencies of the same component in different data-centers are often uncorrelated. This leads us to argue that there are potential latency savings if work related to a poorly performing component is dynamically reassigned to a replica in a remote data-center. Such dynamic reassignment is further facilitated given that typical application deployments in data-centers must operate with enough extra servers to handle short-term variations in workload. This approach is distinguished from conventional schemes that load-balance application traffic across entire data-centers as a whole in a coarse-grained fashion (for e.g., Akamai [14, 23]), in that only the processing related to poorly performing components is directed to alternate data-centers, and the resources of other components is utilized to the extent possible.

Based on these insights, we present the design of a system which we term Dealer. For each component, *Dealer* dynamically splits transactions directed at a particular component across its replicas in different data-centers. In doing so, *Dealer* seeks to minimize user latencies, and takes component performance and loads, as well as intra-data-center and inter-data-center communication latencies into account. *Dealer* seeks to be responsive to poor performance, while ensuring stability. *Dealer* includes algorithms to dynamically discover the load that can be handled by components, and can automatically adapt these capacity estimates to changes in the mix of application transactions.

We implemented *Dealer* in a fashion that can be integrated with any enterprise application. We have extensively evaluated our approach on two multi-tier applications on actual Azure cloud deployments. The first application is data-intensive, while the second application involves interactive transaction processing. Overall, the results indicate the importance and feasibility of our mechanisms.

2 Measurement and Implications

In this section, we begin by characterizing variability in workloads and transactions mixes of multi-tier applications. This work is based on an analysis of web server traces of a large campus university. We next 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.

2.1 Workload variability in multi-tier applications

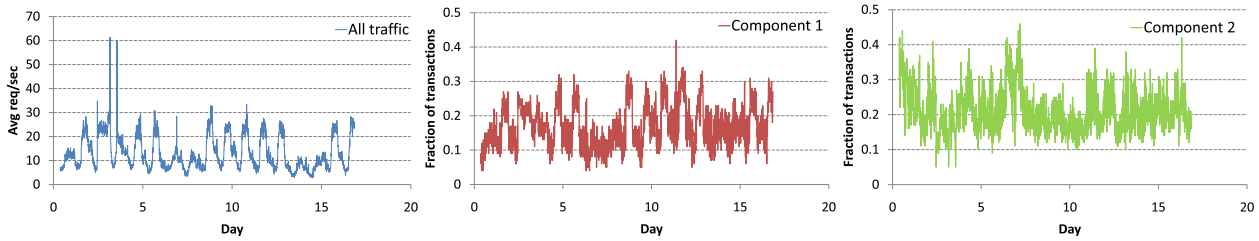
We begin by presenting insights on the variability of workloads of multi-tier applications, and the implications for cloud deployments. Our insights have been gleaned by collecting logs from a web-service of a large campus network. The web-service includes a front-end and multiple back-end components. All requests enter through a front-end, which are then directed to different back-end component based on the type of request. For instance, a separate component is in charge of mail requests, another component in charge of mailing list related transactions, a third component in charge of web requests associated with a subset of departments, a fourth component in charge of web requests associated with another subset of departments, and so on. The logs we collected were at the front-end, but had sufficient information to identify which back-end component was involved in serving the requests.

We now summarize some of the key insights obtained through our analysis:

There is much variability not only in workload but also in the mix of transactions: Figure 1(a) shows the request rate at the front-end server averaged over 10 second intervals, as a function of time. The workload exhibits significant variability, and diurnal effects. Figure 1(b) and Figure 1(c) show the fraction of transactions going to each component as a function of time. The graphs exhibit significant variability with the fractions ranging from 5% to over 30% for each component. These results indicate that in addition to variability in overall workload, changes in composition of transactions can lead to significant variability of loads seen by individual components.

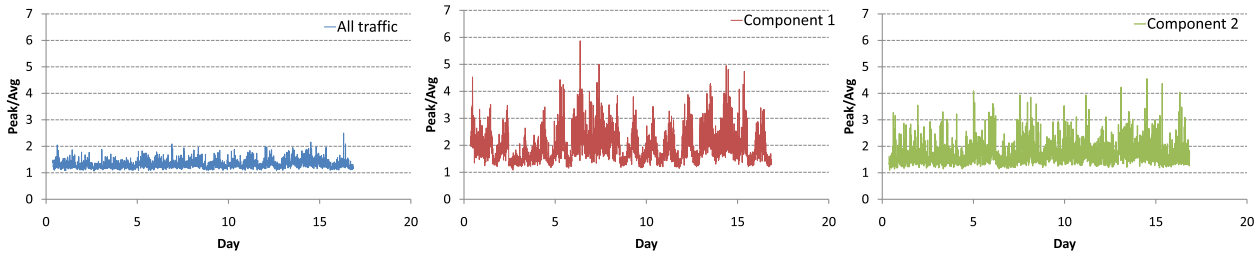
Short-term variability necessitates large margins even in cloud deployments: To handle fluctuations in workload when deploying applications in data-centers, application architects must maintain margins, i.e., pools of servers beyond the expected load [8, 26, 25]. Cloud deployments allow for dynamic invocation of resources during peak periods, which can result in reduced margin requirements. However, cloud deployments may still require margins to cope with *short-term* fluctuations in workload. This is because dynamic invocation of new server instances in the cloud typically 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 make sure a newly booted server is warmed up, for e.g., a server may not be able to meet SLA requirements until its caches are filled with relevant data.

Figure 2 depicts the short-term variability for both the web front-end, and two back-end components. Each 10 minute period is considered, the average and peak request rate during this period is determined, and the peak to average ratio is then computed. Figure 2(a) depicts the short-term variability for the front-end, while (b) and (c) depict the variability for two back-end components. While the ratio is around 1.5 for the front-



(a) Request rate at front-end (b) Fraction of transactions for Component 1 (c) Fraction of transactions for Component 2

Figure 1: Variability in overall workload and transaction mix. (a) shows the over-all request rate at the front-end. (b) and (c) respectively show the fraction of all transactions directed to two back-end components.



(a) Front-end (all traffic) (b) Component 1 (c) Component 2

Figure 2: Short-term variability in workload for front-end and two back-end components. The peak and average rates are computed during each 10 minute window and the ratio of peak to average over each window is plotted as a function of time.

end, it is much higher for each of the two back-end components, and can be as high as 3 or more during some time periods. Overall, these results indicate that a significant margin may be needed even in cloud deployments to handle shorter-term workload fluctuations.

Margins requirements across different tiers of multi-tier applications are heterogeneous and exhibit much variability: Figure 2 not only illustrates the need for margins with cloud deployments, but also illustrates 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 back-end components during some time periods. Figure 2 also illustrates that the exact margin required is highly variable over time. These factors make it difficult to simply over-provision a cloud component since (i) it is complicated to exactly estimate the extent of over-provisioning required; and (ii) over-provisioning for the worst-case scenario could be expensive.

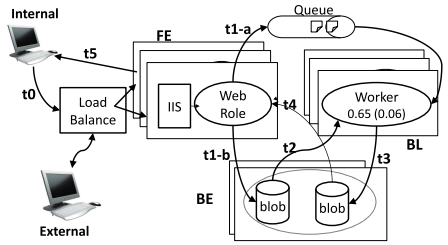
2.2 Performance variability in the cloud

In this section, we characterize variability in performance experienced by applications when deployed on the cloud. Our experiments have been conducted using two applications on the Windows Azure platform. The first application, $Thumbnails$, is a typical 3-tier application provided as part of the Windows Azure SDK. The application involves users uploading a picture to a server and receiving thumbnail versions in turn. The second application, *StockTrader*, is a tiered enterprise web application that allows a user to buy and sell stocks, view her portfolio information, modify her profile, and perform other tasks like viewing a stock quote or her recent transactions. We used the version of *StockTrader* from Apache Stonehenge Interoperability Project [2] and re-wrote it to be deployed on Azure cloud. Figure 3(a) and Figure 3(b) respectively show the component architecture and data-flow for each application.

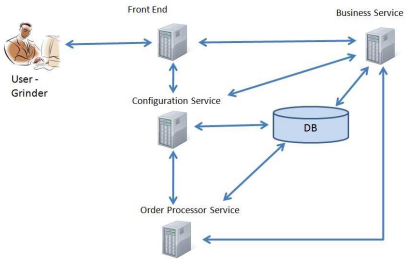
We ran each application simultaneously in two separate data-centers, both located in the United States. The users were assumed to be located in a campus network, also in the United States. Each application was carefully configured with enough instances of each component so they could at least handle the average load along with additional margins. More details of how we configured the deployment 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:



(a) Thumbnails application architecture and data-flow. The application is composed of a Front-End (FE), Back-End (BE), and Business-Logic (BL). Users upload pictures (t0). The FE writes the image to the BE (t1-b) and notifies the BL(t1-a). The BL in turn creates a thumbnail, and stores it in BE (t3). The FE retrieves the thumbnail (t4) and sends it to the user (t5).



(b) *StockTrader* architecture and data-flow. Components include a user facing front-end (FS), a business logic server (BS) that handles computation associated with most requests, the Order Service (OS) that handles buy and sell operations, a Database (DB), and a Config Service (CS) that binds these components. The precise data-flow depends on the type of transaction.

Figure 3: Applications Testbed.

There is significant variation in performance of cloud components: Figure 4 considers the *Thumbnail* application and presents a box plot for the total response time and each of the individual contributing delays for each data-center. The X-axis is annotated with each of the delays being measured, with the number in parenthesis represents the data-center being measured. The first two box plots on the left are the total response time for the two data-centers. The other box plots correspond to delays of the components contributing to the total response time. For e.g., BL-BE(1) represents the delay between the Business-Logic (BL) and the Back-End (BE) components, at the first data-center.

Several interesting observations can be made from Figure 4. 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 the first data-center (DC1) is smaller than the second data-center (DC2), DC1 shows significantly more variability. Third, while the Internet upload delays are

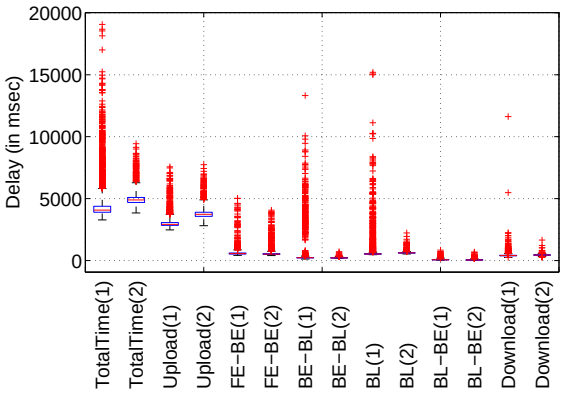


Figure 4: Box plot for total response time, and contributing processing and communication delays for Thumbnail application. The bottom and top of each box represent the 25^{th} and 75^{th} percentiles, and the line in the middle represents the median. The vertical line (whiskers) extends to the highest datum within 3*IQR of the upper quartile, where IQR is the interquartile range. Points larger than this value are considered outliers and shown separately.

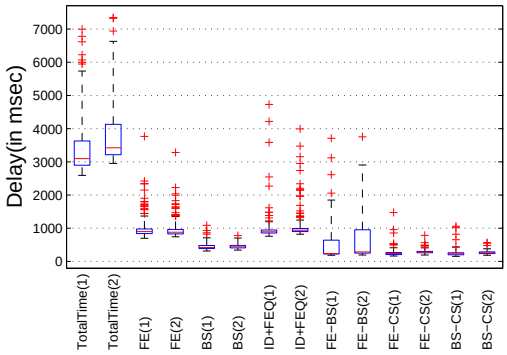


Figure 5: Box plot for total response time, and contributing processing and communication delays for *StockTrader*.

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.

Figure 5 presents a similar box plot for *StockTrader*. Note that delays involving the OS component have not been shown for space reasons. The figure shows a similar trend, indicating significant variability in both total response time, and each of the contributing delays for both data-centers. While the median delay with DC1 is lower, both DCs do show significant variability. The variation is particularly high for communication between the front-end (FE) and Business Service (BS) (which includes queuing at BS). The delay

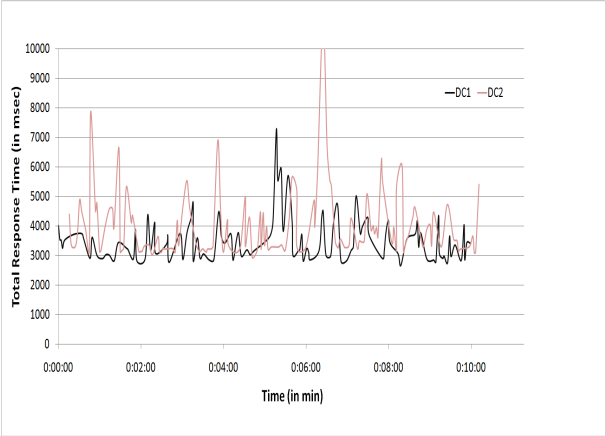


Figure 6: The total response time for the two data-centers for the *StockTrader* for a 10 minute snapshot.

ID+FEQ denotes the sum of Internet transfer related delays and queuing at the front-end. We notice this term also shows noticeable variability, which we found typically arises due to queuing at the front-end.

The performance of component replicas in multiple data centers is not correlated: While our results so far indicate that there is significant variability in response times for both data-centers, Figure 6 next considers the correlation between the response times of the two data-centers. The figure shows the response time with the *StockTrader* for a 10 minute snapshot, at each of the data-centers. We see that while there is variability in response times of both DCs, they are not correlated with each other, and different DCs perform differently at different times.

3 *Dealer* Design Rationale

In this section, we present the rationale underlying the design of *Dealer*. *Dealer* is designed to enable applications deployed in multi-cloud settings adapt to short-term variability in workloads, and to performance variations of cloud components. Such shorter term variability has been found to be common from our measurement studies in Section 2. *Dealer* is meant to complement rather than replace other adaptation mechanisms, such as redirection based on the Domain Name System (DNS), and dynamic invocation of cloud resources. While such mechanisms are suitable for adaptation over longer time-scales, they may be unable to cope with shorter term variability, as we discuss in this section. We next describe key ideas behind *Dealer*:

Exploit margins in remote data-centers of multi-cloud deployments: Our measurement studies in Section 2 show that cloud deployments must operate with margins of over 300%, to handle short-term variability in workload, given the time-scales involved in invoking new cloud resources. This in turn presents several opportunities. First, rather than provisioning the margin for each data-center in isolation, there is potential for cost reductions by sharing margins across data-centers. Such an approach is promising since the margin requirements are highly variable over time (Figure 2), and given both the margin requirements and latencies of cloud components across data-centers may not be correlated at any given instant(Figure 6). Second, tapping into margins in remote DCs may enable the application to adapt to short-term workload variability and performance problems, which may not otherwise be possible. Transactions could be quickly redirected from a temporarily overloaded or poorly performing component to a replica in another data-center. Temporary overloads may be particularly common during maintenance of a data-center, when it is likely to be operating with reduced margin levels.

Split transactions at the granularity of components rather than data centers: The notion of redirecting user traffic to remote data centers for reasons such as load-balancing or avoidance of Internet congestion between a user and a certain data center is well known [23, 7]. However, most mechanisms used today are coarse-grained, and move entire user requests to alternate data centers. In large multi-tier applications (with potentially hundreds of components), it is possible that only a small number of components are impacted by a temporary surge in requests (Figure 2) or performance problems (Figure 4). Redirecting all application traffic to an alternate data center does not make effective use of other components (which have already been paid for). Further, since there is much heterogeneity in component margins (e.g., Figure 2), it isn't clear the redirected requests can be accommodated by all components in the alternate data center. For these reasons, *Dealer* focuses on a new design point, involving splitting transactions at a finer per-component granularity.

Complement rather than replace DNS-based redirection mechanisms: DNS-based redirection is a commonly used coarse-grained approach to redirect user traffic to alternate data centers today [23, 7]. This approach works by providing clients the IP addresses of front-end servers in alternate data centers. However, in many web services, including the service studied in Section 2.1, the front-end IP address contacted is the same, though the back-end components contacted might be different based on the type of transaction. Thus, redirection based on DNS is not feasible for the finer-grained component-level transaction redirection targeted by *Dealer*. Further, there have been studies that have quantified the responsiveness of DNS [20], which have suggested that DNS is a coarser-grained mechanism that may be poorly suited for applications which require quick response to link failures or perfor-

mance degradations. A primary reason for this is that over 47% of clients and local DNS (LDNS) servers, may violate time-to-live (TTLs) values that determine how long an earlier DNS mapping must be cached, and in some cases the violations are as large as two hours. In contrast, *Dealer* is designed for fast response to shorter-term variability and performance fluctuations.

That said, *Dealer* is intended to complement DNS-based redirection. In particular, *Dealer* relies on DNS to correctly map users to front-end servers in appropriate data-centers, and relies on DNS to adapt to performance problems between users and their front-ends. *Dealer* is targeted at handling performance problems associated with back-end components, and communication between components, and does not target problems between users and front-end servers, or the front-ends themselves.

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

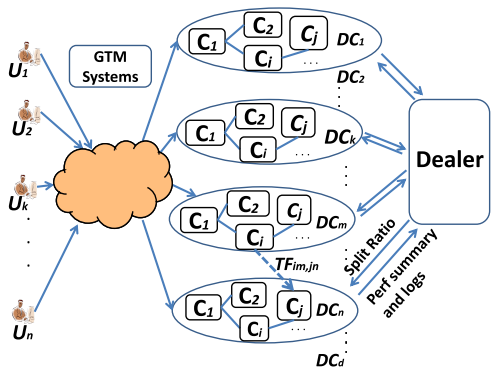


Figure 7: System overview

Consider an enterprise application with multiple components $\{C_1..C_l\}$. We consider a multi-cloud deployment where the application is replicated across d data-centers, with instances corresponding to each application 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 instances of component C_i in data-center m as C_{im}.

Traffic from users is mapped to each of the data-centers using standard mapping services that are used today based on metrics such as geographical proximity, or latencies [23]. 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 of a data-center is the additional load that can be served by that component 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 components.

In making its determination, *Dealer* estimates several parameters including (i) delay of processing user requests in individual components, and data-center links; (ii) available capacity of components in each data-center, i.e., the load that each component can handle; and (iii) application communication patterns, i.e., the fraction of requests that involve communication between each pair of application components, and the average size of transactions between each component pair. 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 both by the research community, and in the industry [10, 19, 15, 6]. While any of these techniques may be applied, in our current implementation, each application is instrumented using knowledge of the application to capture the delays incurred by user transactions on individual components and links. This information is periodically reported 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 reasonable.

Smoothing delay estimates: It is important to trade-off the agility of *Dealer* 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. Considering N(t) ensures a higher weight is given to windows with more transactions compared to windows with fewer ones. The use of a WMA scheme ensures that *Dealer* reacts to prolonged performance episodes that last several seconds, while not aggressively reacting to extremely short-lived performance problems within a time window. W determines the number of windows for which a link/component must perform poorly (well) for it to be avoided(reused) by *Dealer*. 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 whether 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., [3]). To bound the overhead of such probes, 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:

$$P_i = \frac{CR_i}{\sum_j CR_j} \quad (2)$$

Here, CR_i is the compliance ratio, or the fraction of requests that use a given path which have a response time lower than the Service Level Agreement (SLA). The intuition is that a path that has generally been good might temporarily suffer poor performance. Biasing the probing algorithm 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. In addition, *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 phased based on coarse estimates of link delays. Such coarse estimates can be obtained based on the size of transactions exchanged between the components (obtained through monitoring application traffic), and the bandwidth between data-centers. While each individual application may measure the bandwidth between every pair of data-centers, cloud providers could provide such bandwidth estimation services in the future amortizing the overheads across all applications.

4.3 Determining transaction split ratios

In the last section, we discussed how *Dealer* estimates the processing delays of components, and communication times of links. In this section, we discuss how *Dealer* uses this information 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 pair of $<$component,data-center $>$ choices. 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 below, the term combination refers to an assignment of each application component to exactly one data-center. For example, in Figure 7, 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 operates by iteratively assigning 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 choice of combinations. The mean delay is computed like in [17]. 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:

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

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 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:

$$DR(L_i,t) = \frac{W(L_i,t)}{\sum_k W(L_k,t)}, where$$
$$W(L_i,t) = \sum_{\ell=0}^{W-1} Rank(L_i,t-\ell) * Req(L_i,t-\ell) \quad (4)$$

Here, $Rank(L,t)$ is the ranking of combination L at the end of time window t (with the lowest mean delay combination assigned the highest ranking), and $Req(L,t)$ is the number of requests sent on combination L during that time window. 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 have found that choosing values of W 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).

Algorithm 1 Determining transaction split ratios.

1: **procedure** COMPUTESPLITRATIO()
2: 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
3: Let $AC[i,m]$ be the available-capacity matrix for C_{im} Initialized as $AC[i,m] \leftarrow C[i,m]$
4: 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
5: Let T_i be the load on each component ($\sum_j T_{ji}$)
6: Let *FA* be fraction of transactions that has been assigned to combinations. Initialized as FA $\leftarrow 0$
7: **Goal**: Find $TF[im,jn]$: the number of transactions that must be directed between C_{im} and C_{jn}
8: Foreach combination L, sorted by mean delay values
9: For each C_{im} in L
10: $f_i \leftarrow \frac{AC[i,m]}{T_i}$
11: $min_f \leftarrow \min_{\forall i}(f_i)$
12: $ratio = \min(min_f, DR(L,t))$
13: Rescale damping ratios if necessary
14: For each C_{im} in L
15: $AC[i,m] \leftarrow AC[i,m] - ratio * T_i$
16: $TF[id(i),jd(j)] \leftarrow TF[id(i),jd(j)] + ratio * T_{ij}$, $\forall i,j$
17: $FA \leftarrow FA + ratio$
18: *Repeat until FA = 1*
19: **end procedure**

Algorithm 2 Dynamic capacity estimation.

procedure COMPUTECAPACITYTHRESHOLD(T, D)
  if $D > 1.1 * DelayAtThresh$ **then**
    if $T <= Thresh$ **then**
      $LowerThresh \leftarrow 0.8 * T$
      $ComponentCapacity \leftarrow Thresh$
    else
      $Thresh \leftarrow unchanged$
      $ComponentCapacity \leftarrow Thresh$
    end if
  else if $D <= DelayAtThresh$ **then**
    if $T >= Thresh$ **then**
      $Thresh \leftarrow T$
      $ComponentCapacity \leftarrow T + 5\% of T$
    else
      $Thresh \leftarrow unchanged$
      $ComponentCapacity \leftarrow Thresh$
    end if
  end if
end procedure

4.4 Estimating capacity of components

We next discuss how *Dealer* determines the capacity of components in terms of the user load that each component can handle. Typically, application delays are not impacted by an increase in load upto a point which we term as the *threshold*. Beyond this point, application delays increase gradually with load, until we enter a breakdown region 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 sensi-

tive to the types of application transactions, and may change dynamically as the mix of application transactions 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 the worst-case load (i.e., transactions that are most expensive for that component to process). Alternately, *Dealer* may learn an initial threshold during the boot-up phase of an application in the data-center, given application traffic typically ramps up slowly before production workloads are handled.

Dealer dynamically updates the threshold in response to application behavior using the algorithm summarized in Algorithm 2. The parameter *DelayAtThresh* represents the delay in the flat region learnt from 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 the threshold, *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 whether *Dealer* is operating too conservatively.

The algorithm begins by checking whether the delay is unacceptably high (line 2). In such a scenario, if $T \leq Thresh$, (line 3) it is an indication that the threshold must be lowered. Otherwise (line 6), the algorithm leaves the threshold unchanged, and lowers the component capacity to the threshold. If the delay D is comparable to *DelayAtThresh* (line 10), then, it is an indication the component can take on more load. If $T \geq Thresh$ (line 11), this is an indication that the threshold estimate is too conservative, and hence the threshold is increased. Further *ComponentCapacity* is set to slightly higher than the threshold to experiment whether the component can in fact absorb more transactions. If however $T < Thresh$, (line 14), then *ComponentCapacity* is set to *Thresh*, to allow more transactions to be directed to that component. Finally, while we have used component delays as a means of estimating if the component is saturated, we note that one could also use other metrics such as CPU and memory utilization of components as well as sizes of queues being processed.

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 Section 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 mixes. (§ 5.3-§ 5.5).

5.1 Evaluation Methodology

We study and evaluate the design of *Dealer* by conducting experiments on the two applications, *Thumbnail* and *StockTrader*, that we had introduced in § 2. All experiments were conducted on the Azure cloud platform, and by deploying a given application simultaneously in two Azure data-centers located geographically apart within the United States. In all experiments, application traffic to one of the data-centers (referred to as the primary data-center) was subjected to a range of application workloads, and the performance compared with and without *Dealer*. Application traffic to the other data-center (referred to as the secondary data-center) on the other hand was maintained at a steady rate, and was run without *Dealer*. The objective was to not only study the effectiveness of *Dealer* in enhancing performance of traffic to the primary data-center, but also ensure that *Dealer* did not negatively impact performance of traffic to the secondary data-center.

Application traffic to both the primary and secondary data-center was generated using a Poisson arrival process. Spikes in workload were achieved through a higher mean arrival rate for the Poisson process. The *Thumbnail* application was relatively simple since it had only one type of transaction. However, a key workload parameter that we did vary was the size of pictures that were uploaded by users. The *StockTrader* application was more complex as it involved a variety of transactions (requests that involve viewing the homepage, buying or selling stocks, fetching quotes etc.) To generate a realistic mix of transactions, we used the publicly available DaCapo benchmarks [13]. A set of user sessions were generated using a Poisson arrival process, with each session consisting of series of requests as specified in the benchmark.

The applications were deployed in both data-centers with enough instances of each component so that they

could handle typical loads along with additional margins. We estimated the capacities of the various components deployed on each data-center separately through a series of stress-tests. For instance, the *Thumbnail* application was provisioned to handle an average load of 2 requests per second (typical rates in our experiments) along with a 100% margin (typical of real deployments as shown in § 2. We found empirically that this required 2 instances of the front-end (FE), and 5 instances of the business-logic servers (BL). Likewise, for the *StockTrader* application, we found that handling a load of 0.5 user sessions per second (each session consisting of a series of 4 requests) required provisioning 5 instances of the front-end (FE), 2 instances of the Business Service (BS), and 1 instance of the Order Processing Service (OS). The *StockTrader* application involves a database that could only be located in one data-center (primary in our setup). We found that this resulted in slightly higher transaction processing times for requests entering the secondary data-center, consequently resulting in more active sessions and queuing at all components. To compensate for this, we used a lower request rate of 0.35 sessions per second at the secondary data-center.

When deploying applications across data-centers, it is important to honor natural application constraints. As described above, in the *StockTrader* application, the database is deployed only in the primary data-center. Further, each component can only contact the Config Service (CS) in its local data-center, since all components (FE, BS or OS) bind themselves to their local CS for obtaining the communication credentials of the other components. Finally, in *StockTrader*, all requests belonging to a user session must use the same set of components given the stateful nature of the application.

5.2 *Dealer* under natural cloud dynamics

In this section, we evaluate the effectiveness of *Dealer* in responding to dynamics that occur naturally in a real cloud deployment. Our goal is to both explore the inherent performance variability in cloud environments, and evaluate the ability of *Dealer* to cope with such variability.

Our experiments are conducted using the *Thumbnail* application configured as described in § 5.1. The focus of these experiments was to compare the performance of the application with and without *Dealer*. Ideally it is desirable to compare the two schemes under identical conditions. Since this is not feasible on a real cloud, we ran a large number of experiments with and without *Dealer*, alternating between the two approaches. In particular, the duration of our experiment was 48 hours, with each hour split into two half-hour runs; one

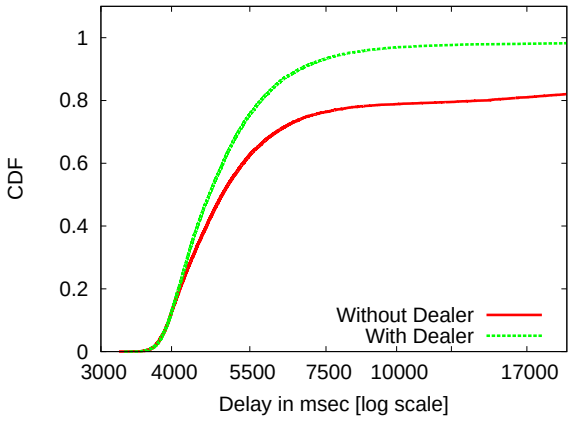


Figure 8: CDF of total response time for 48 hours with and without *Dealer* under natural cloud dynamics. X-axis trimmed at 20 seconds.

without activating *Dealer*, and another with it.

We deployed a mix of PlanetLab nodes and a set of hosts within a campus network to generate traffic to both data-centers. A total of 66 PlanetLab users, spread across the US, were used to send requests to the primary data-center. Furthermore, another set of users, all located inside a campus network, were used to generate traffic to the secondary data-center. Requests had an average size of 1.4 MB (in the form of an image), and an average request rate of 2 requests per second at each data-center generated using a Poisson process.

Figure 8 shows a CDF of the total response time when operating with *Dealer* and without it, for the whole experiment. The X-axis is in milliseconds, and is trimmed at 20 seconds for better visualization. The figure shows *Dealer* performs significantly better. The 50^{th}, 75^{th}, 90^{th}, and 99^{th} percentiles with *Dealer* were 4.6, 5.4, 6.6 and 12.7 seconds respectively. In contrast, the corresponding values without *Dealer* were 4.9, 6.8, 43.2 and 90.9 seconds. The reduction was more than a factor of 6.5x for the top 10 percentiles.

Figure 9 helps better understand why *Dealer* performs better. The figure shows a box-plot of total response time for each run of the experiment. The X-axis indicates the run number, and the Y-axis shows the total response time, in milliseconds. Figure 9(a) shows the runs with *Dealer* enabled, and 9(b) shows the runs with *Dealer* disabled (i.e., all traffic going to the primary data-center 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:

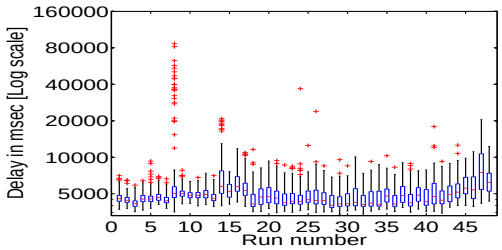
• First, in the absence of *Dealer*, most of the runs had a normal range of total response time with a median value close to 5 seconds. However, the delays

were much higher in runs 13-16 and 43-48. Further investigation showed these high delays were caused by the BL components in the primary data-center, which seemed to have lower capacity to absorb requests during those periods, and consequently experienced significant queuing. Such a sudden dip in the component capacity is an example of the kind of event that may occur in the cloud, and highlights the need for *Dealer*.

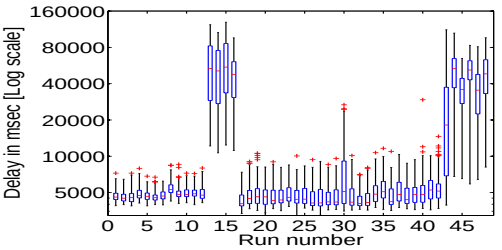
• Second, *Dealer* too experienced the same performance problem with the BL component in the primary data-center during runs 13-16 and 43-48. Figure 9(a) shows this trend where total response time has a median at about 8 seconds during these bad periods. However, *Dealer* was able to mitigate the problem by tapping into the margin available at the secondary data-center. Figure 10 shows the fraction of requests that were directed to one or more components in the secondary data-center by *Dealer*. Each bar corresponds to a run and is split according the the combination of components chosen by *Dealer*. For example, for run 0 around 9% of all requests handled by *Dealer* used one or more components from the secondary data-center. Further, for this run, 5% of the requests used the path PPS (primary FE, primary BE, and secondary BL), while 1% used PSP, and 3% used the path PSS. We see that *Dealer* directs a much larger fraction of requests to the secondary data-center in runs 13-16 and 43-48. Further, most of the requests directed to the secondary DC take the path PPS, which indicates the BL component in the secondary DC is used.

• Third, we have compared the performance with and without *Dealer*, 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, while the 99^{th} percentile was reduced from 18.1 to 8.9 seconds. Most of these benefits were due to *Dealer* being able to handle short-term spikes in workload by directing transactions to the BL component of the secondary data-center. There were also some instances of congestion in the blob of the primary data-center which led *Dealer* to direct transactions to the blob of the secondary data-center.

• Finally, Figure 9(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 requests directed to the secondary data-center, and were caused by high upload delays of requests going to the secondary data-center. This was likely due to Internet congestion between the users and the secondary data-center. 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 [29, 14].



(a) With *Dealer*.



(b) Without *Dealer*.

Figure 9: **Box-plots showing the total response time for all runs in the experiment.**

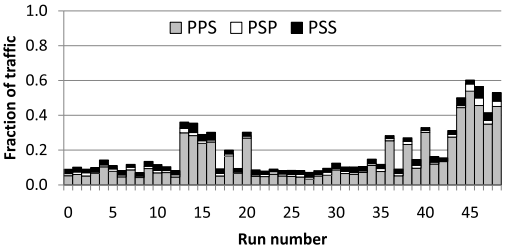
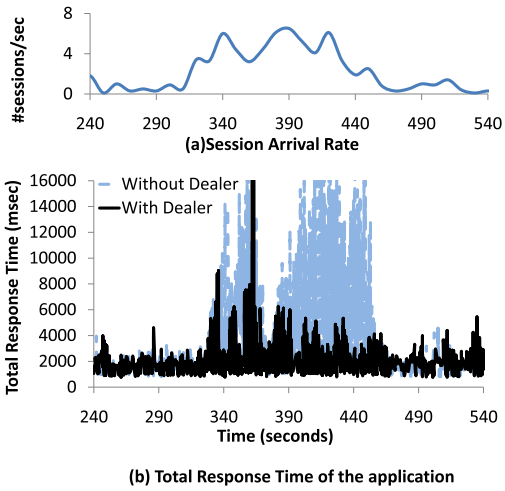


Figure 10: **Fraction of *Dealer* traffic sent from the primary to the secondary data-center.**

5.3 Reaction to surges in user load

In this section, we evaluate *Dealer*'s effectiveness in reacting to sudden increases in the application workload and present our results for this scenario using the *StockTrader* application. Figure 11(a) shows the number of user sessions per second that arrive at the front-end server in the primary data-center. The user sessions are generated with Poisson arrivals having a mean rate of 0.5 per second under normal conditions and at a mean rate of 2 per second during the spike. Figure 11(b) compares the total response time seen by the requests issued to the application at the primary data-center deployed with and without *Dealer* for the same workload. We can clearly see that the performance of the application with *Dealer* (the solid curve) is substantially better during the spike than without it (dotted curve). This is because *Dealer* is quickly able to redirect some of the excess traffic over to the BS in the secondary data-center and prevent the degradation of the application's response time.



(b) Total Response Time of the application

Figure 11: Performance under change in workload(*StockTrader*).

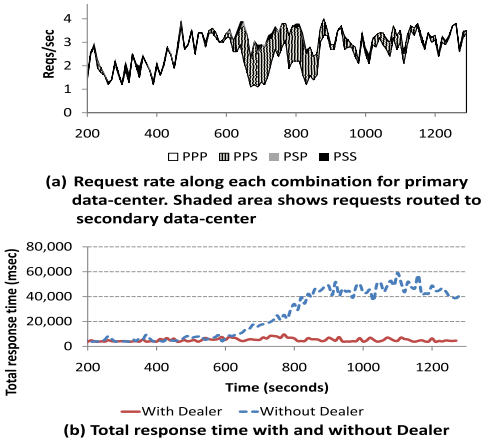
5.4 Reaction to component failures

Applications deployed in the cloud may see failures of components both due to actual physical failures, and due to maintenance and upgrades. For instance, Windows Azure's SLA states that an application's component has to to have two or more instances to get 99.95% availability, as instances can be taken off for maintenance and platform upgrades at any time [5]. Maintenance and upgrade operations make the application work with lower margins, and render the application susceptible to even modest workload spikes.

In this experiment, we test *Dealer*'s capability to adapt to such component failures using the *Thumbnail* application. We use the same setup as in § 5.1 with two additional modifications: i) at time 400, we introduce a slight increase in user load at the primary data-center (from 2 reqs/sec to 3.5 reqs/sec), thus decreasing the margin from 100% to 25%; and ii) at time 600, we reboot one of the BL instances in the primary data-center to reproduce the case of an instance that is taken down for maintenance, upgrade or physical failure. After rebooting, the instance becomes available at around time 900.

Figure 12(a) shows the request rate to the primary data-center. The shaded area under the curve shows the number of these requests at each time snapshot that were serviced by at least one component in the secondary data-center. Different shades are used to represent the different paths used by *Dealer*. Figure 12(b) shows the total response time of all requests, comparing the performance with and without *Dealer*. The x-axis in all figures represent the time in seconds, and is aligned in all figures.

The figure clearly shows the benefits of *Dealer*. Around time 600, *Dealer* detected an increase in total response



(a) Request rate along each combination for primary data-center. Shaded area shows requests routed to secondary data-center

(b) Total response time with and without Dealer

Figure 12: Performance under component failures (*Thumbnail*).

time and starts redirecting transactions that arrive at the front-end of the primary data-center to the business logic of the secondary data-center. After the instance was brought back up around time 900, *Dealer* returned to using its original path.

5.5 Reaction to change in transaction mix

Multi-tier applications show a lot of variability not only in the arrival rate of user requests but also in the mix of transactions, as we discussed in §2. In this section, we evaluate the effectiveness of *Dealer* in adapting to changes in transaction mix using both *Thumbnail* and *StockTrader*.

The *Thumbnail* application is relatively simple with just one type of transaction. However, the performance does depend on the size of user images. Using the same configuration described in § 5.1, we increase the size of images that users upload to the primary data-center's from 860 KB to 1.4 MB during time 400 to 800, and reduce it to 860 KB after that. Figure 13 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 of 60%. 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 the 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 the secondary data-center.

We next present our evaluation of *Dealer*'s response to a similar scenario using the *StockTrader*. While we make use of sessions from the DaCapo benchmark for the normal workload, we stress the system by increasing the $heaviness$ of the Fetch Quotes request for a short duration. In particular, we increase the number of quotes to be fetched from the default number of 5

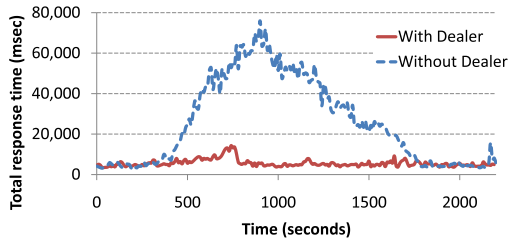


Figure 13: Performance under varying transaction size (*Thumbnail*).

to 30 quotes per session. The heavier transactions arrive at around time 200 and lasts for about 100 seconds after which the transactions go back to the normal mix. Figure 14(a) compares the Total Response Time of all the requests seen by the application when deployed with and without *Dealer* on the primary data-center. It can be seen that the application with *Dealer* performs much better than without it.

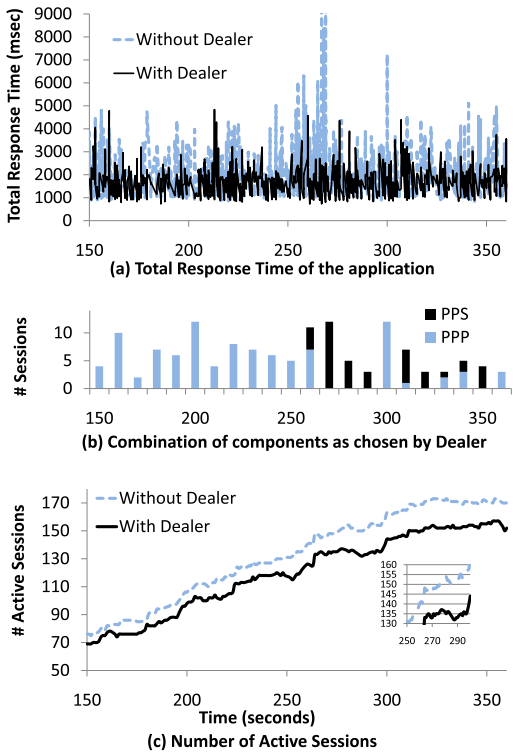


Figure 14: Performance under change in transaction mix(*StockTrader*).

Figure 14(b) shows the number of user sessions seen over time along with the corresponding combination of components that was chosen by *Dealer* for those sessions. For instance, at time 250, when *Dealer* saw an increase in the response time of the requests along the combination PPP, i.e. FE (primary), BS (primary) and OS (primary), it decided to direct some of the sessions along the combination PPS, which is FE (primary), BS (primary) and OS (secondary). While it is clear from figure 14(a) that this decision helped the application, we now explain why and how the decision was made by *Dealer*. As we mentioned earlier in § 5.1, all the components need to communicate frequently with CS. To process a Fetch Quotes request, BS has to contact the CS and DB as many times as the number of quotes fetched, which results in a large number of connections made to the CS. This increase causes queue formation at the CS resulting in bad performance of the application. When *Dealer* chooses to route the Order processing requests to the OS in the secondary data-center, all connections from OS (secondary) remains local to its corresponding CS (secondary) and therefore do not contribute to the queue build up in the primary. The benefit of this reaction can be observed from Figure 14(c) which shows the number of active sessions over time for both the cases. During the period (250-300) when *Dealer* decides to route requests along the PPS combination, the number of active sessions remains fairly constant (around 135) as can be observed from the solid curve. An interesting observation here is that *Dealer* did not choose to direct requests to the BS (secondary), i.e. along the PSS combination. This is because a large number of requests from BS (secondary) to the DB (located only at the primary data-center) would result in a higher amount of data transfer across the data-centers thereby incurring higher latencies. *Dealer* determines this through the probes, and therefore decides to direct only the order processing traffic to the secondary data-center.

6 Related Work

Several works have studied the problem of mapping users to appropriate data-centers [29, 14, 27]. 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. In contrast, our focus is on splitting transactions across data-centers at the granularity of individual components. The goal is to alleviate short-term performance problems or transient overloads on particular components inside a data-center, while still utilizing those components inside the data-center that perform well. To our knowledge, this is the first work that explores component-level performance-aware transaction splitting in multi-cloud deployments.

The cloud industry already provides mechanisms to scale up or down the number of servers in a particular component in a particular data center based on workload [21, 4]. However, it takes tens of minutes to invoke new cloud instances and ensure the servers are warmed up. Our focus is on faster adaptation at

shorter time-scales, and is intended to complement solutions for dynamic resource invocation.

Recent work [17] has developed algorithms for planning hybrid cloud deployments of enterprise applications. This work is limited to a static snapshot of application workload, and network conditions. In contrast our focus is on dynamically adapting multi-cloud layouts to short-term dynamics in cloud performance and application workload.

Several researchers have pointed out the presence of performance problems with the cloud [28, 18, 12, 11]. In contrast our focus is on designing systems to adapt to short-term variability in application workload and performance. Other researchers have started looking at support that can be provided by cloud providers to achieve performance isolation in data-centers in the presence of multiple tenants [22]. Our work is complementary in that we take an enterprise-centric view, and focus on ways to adapt applications to performance variability that may occur in the cloud.

7 Conclusions

In this paper, we have shown that it is critical for multi-tier enterprise applications to adapt to short-term variability in the performance of cloud components and application workloads when deployed in the cloud. We have shown the importance and feasibility of adapting to such variability by dynamically splitting transactions across data-centers at the granularity of individual application components in a performance-aware fashion. Our work is in contrast to traditional approaches that employ coarse-grained load-balancing at the granularity of data-centers. We have presented *Dealer*, a system built around this new design point. We have shown that it is easy to integrate *Dealer* with two contrasting multi-tier applications. Evaluations on actual Azure cloud deployments indicate the benefits of *Dealer*. Under natural cloud dynamics, the $90th$ and higher percentiles of application response times were reduced by more than a factor of 6 with *Dealer*. Our controlled experiments show *Dealer* ensures good application performance under a variety of controlled experiments including abrupt spikes in workload, changes in transaction mixes, and component failures. While the results are promising, they are a start. As future work, we plan to explore the performance of *Dealer* with a wider range of applications and cloud environments, as well as evaluate the performance under scale.

8 References

[1] Animoto - Scaling Through Viral Growth. `http://aws.typepad.com/aws/2008/04/animoto---scali.html`.

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

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

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

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

[6] Microsoft Corp., Event Tracing for Windows (ETW). `http://msdn.microsoft.com/en-us/library/aa363668(v=VS.85).aspx`, 2002.

[7] B. Ager, W. Mühlbauer, G. Smaragdakis, and S. Uhlig. Comparing DNS resolvers in the wild. In *IMC 2010*.

[8] F. Ahmad and T. Vijaykumar. Joint optimization of idle and cooling power in data centers while maintaining response time. *ASPLOS 2010*.

[9] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[10] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: Online modelling and performance-aware systems. In *HOTOS 2003*.

[11] S. Barker and P. Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. In *MMSys 2010*.

[12] T. Benson, S. Sahu, A. Akella, and A. Shaikh. A first look at problems in the cloud. In *HotCloud 2010*.

[13] S. M. Blackburn and et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA 2006*.

[14] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. Globally distributed content delivery. *Internet Computing, IEEE*, 2002.

[15] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *NSDI 2007*.

[16] D. Gottfrid. The New York Times Archives + Amazon Web Services = TimesMachine. `http://open.blogs.nytimes.com/2008/05/21/`.

[17] M. Hajjat and et al. Cloudward bound: Planning for beneficial migration of enterprise applications to the cloud. *SIGCOMM 2010*.

[18] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: comparing public cloud providers. In *IMC 2010*.

[19] M. Massie, B. Chun, and D. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing 2004*.

[20] J. Pang, A. Akella, A. Shaikh, B. Krishnamurthy, and S. Seshan. On the responsiveness of DNS-based network control. In *IMC 2004*.

[21] RightScale Inc. Cloud computing management platform. `http://www.rightscale.com`.

[22] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the Data Center Network. In *NSDI 2011*.

[23] A. Su, D. Choffnes, A. Kuzmanovic, and F. Bustamante. Drafting behind akamai. *SIGCOMM 2006*.

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

[25] B. Urgaonkar and P. Shenoy. Cataclysm: Handling extreme overloads in internet services. In *PODC 2004*.

[26] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. *OSDI 2002*.

[27] V. Valancius, N. Feamster, J. Rexford, and A. Nakao. Wide-area route control for distributed services. In *USENIX 2010*.

[28] G. Wang and T. S. E. Ng. The impact of virtualization on network performance of Amazon EC2 data center. In *IEEE INFOCOM 2010*.

[29] P. Wendell, J. Jiang, M. Freedman, and J. Rexford. DONAR: decentralized server selection for cloud services. In *SIGCOMM 2010*.