Specifying and Guaranteeing Quality of Service for Web Services through Real Time Measurement and Adaptive Control

Akhil Sahai, Jinsong Ouyang, Vijay Machiraju, Klaus Wurster
E-Service Management Project
E-Services Software Research Department
HP Laboratories, 1501 Page Mill Road, Palo-Alto, CA 94034
{asahai, jinsongo, vijaym, kwurster}@hpl.hp.com

Abstract

Web based services are becoming increasingly prevalent. These web services are accessed not only by end users but also by other web services. A stage has arrived at which availability and performance guarantees are expected of these web services. In many cases web services are hosted by web hosting or aggregator sites making it even more important for the businesses that build these web services to receive certain service level guarantees from the web hosting sites. Specifying, measuring and guaranteeing in real time such service level agreements is a non-trivial task. This paper describes an adaptive control mechanism based on real-time measurements for guaranteeing service level agreements. It will allow the service provider to establish specific performance guarantees for individual transaction of the service, thus letting him decide which of the transactions are most relevant for his business goals. It will also address dynamic activation of standby web server resources and will enable smoth degradation of the overall service when the load can not be handled in time anymore.

Keywords: End-to-End QoS, Web, e-Service management, service monitoring and control

1. Introduction

A Web based service or e-Service can be described broadly as a service available via the Internet that allows the processing of certain business transactions. E-Services are set up for clients and other e-Services to make use of the offered set of functionality. They have a Uniform Resource Locator at which they can be accessed and have a set of Interfaces that can be utilized to access them. E-services are web based applications that are created and hosted by the developing e-business itself or might be hosted frequently by web hosting enterprises or on aggregator sites.

In either case, there is a need for specifying certain Service Level Agreements (SLA) and measuring whether the service is in compliance or not. The e-Service provider and its clients will agree upon the Service Level Objectives (SLO) that detail the SLA. In the hosting case the e-Service provider will use the SLOs as a measure against the hosting business, but the hosting organization will be ultimately responsible for fulfilling the agreed-upon conditions. In the other case the e-Service provider himself will be responsible for delivering the service according to the contracts.

The work for Quality of Service (QoS) that has been done in the area or Web-based e-Services is centered around either providing the QoS at the

- Web Server level, by making sure that the http requests/responses are prioritized and queued separately and handled by the web server according to their priority [4]

- System level, by providing resource allocation at the system kernel level, along with web server request differentiation/prioritization[10][11]

- Network level, the IntServ [5][6] and DiffServ [8][9] approaches at the network level enable resource reservation and in performing service packet prioritization /differentiation respectively.

E-service providers are interested in higher-level metrics, which measure transaction throughput, and ultimately on the number of products sold and the profit made. They are interested in maximizing their revenue while clients are interested in receiving a good or at least satisfactory service and measure a service using the perceived Quality of Experience.

The most natural way to specify service level agreements for both the e-Service provider and its clients should be based on the underlying business transactions that are conducted between the two parties. This way, the e-Service provider can discriminate certain transactions over others, e.g. can establish faster execution times for buying transaction over browsing functions. The client can understand the behavior of the system and does not have to speculate on the service he will receive.

We will describe our approach termed BizQoS based on guaranteeing transaction execution times for individual transactions that will allow the e-Service provider to establish his individual preferences for certain transactions of the offered service over others while complying to the established SLAs.

Clients accessing the e-Service have expectations towards it in the areas of availability, performance and reliability. They want the e-Service to be accessible all the time, offering good response times in the execution of transactions and the presentation of the results. They are also expecting that transactions be executed consistently and that they do not have to reenter the same information over and over again.

Client requirements
Performance
Availability
Reliability

An e-Service provider will be motivated by the service level considerations as well. It will strive to fulfill the established service level goals towards its consumers. Additionally it will try to utilize resources optimally, while not provisioning too many stand-by resources that are unused most of the time. It will try to optimize revenues, and thus depending on an analysis of the client behavior prefer certain transactions over others, e.g. revenue creating transactions over revenue-neutral ones. It will set up measures to adapt to changes in the load on the e-Service. Finally it will implement means to allow the e-Service to degrade gracefully when all the standby resources have been activated and the incoming load is still increasing.

E-service requirements	
Client motivated	Self motivated
Performance Reliability Availability	Transaction differentiation Throughput guarantees Resource allocation and load balancing Dynamic capacity allocation Smooth degradation on overload

3. Service level QoS

The overall definition of the QoS of the e-Service will be determined by consolidating the compliance with the detailed SLA and ultimately with the derived Service Level Objectives (SLO). To allow a machine to compare the SLOs with the real world behavior the SLOs need to be mapped into measurements that can be taken from the running system.

3.1. Service level transactions

Most e-Services offer a well-defined set of interactions. We will measure the execution times of individual instantiations of these interactions and compute aggregates to control the prioritization of subsequent requests.

For example a book selling e-Service will implement all the interactions to enable customers to shop for books. These customers will be real people, but in some cases can also be other e-Services that include the shopping for books into their own service offering. The process of selling a book through this book seller's web site is broken down in a series of sub-interactions like 'login', 'browsing', 'searching', 'adding a book to the shopping cart', 'entering the payment and delivery information' and ultimately the 'buying of the selected books'. This list will be slightly different for other e-Services even in the same business. We will now assign individual service level objectives to all the transactions.

3.1.1 Identifying transactions.

To be able to measure the execution time of individual transactions that are implemented by the e-Service they need to be identified or demarcated at the application level. This can be done either invasively or in a non-invasive manner.

Non-invasive instrumentation can be used when the application was developed without manageability in mind or if changing the application code is not an option.

This requires to discretely monitor the http requests and responses and to map them to service level transactions. This could also involve studying http web server logs or in the most extreme case a person to establish a mapping between http requests and business transactions.

Invasive instrumentation typically requires the insertion of certain function calls into the source code that establish beginning and the corresponding end of transactions.

Software Development Kits (SDK) exist that provide APIs to enable invasive identification of transaction start and end time by inserting function calls into the normal application flow. They also enable the application developer to provide additional context information, e.g. 'number of items bought' for a 'checkout' transaction, to be passed to the management system. ARM[13] and XARM [12] can be used for doing transaction type, transaction instance and service instance level identification and allowing a receiving measurement system to do the necessary correlation. The raw data collected consists of start and stop time, failure and abort counts for transaction(s) and breakdown for its component transactions, at the type and instance level.

3.1.2 Differentiating transactions

An e-Service provides a well-defined set of transactions. A bookstore e-Service for example will structure its offer using transactions like 'login', 'search', 'addToShoppingCart', an 'checkout'. In order to maximize revenue, an e-Service may like to assign different priorities to these transactions. Transactions like 'addToShoppingCart' and 'checkout' transactions could be given higher priority over 'search' requests so as to maximize revenue. There might be other cases where an analysis of the

customer behavior shows that preferring 'search' transactions over 'checkout' might be the appropriate approach, with the underlying logic, that the customer that has made up his mind and has already finished shopping will tolerate slight delays, while the shopper that is still looking for the most interesting offer might potentially abort his shopping and move to another site.

3.2 QoS Management

An infrastructure that is used often to implement e-Services is a J2EE compliant Application Server [1][3]. A typical J2EE compliant application server consists of a set of web servers with a network load balancer in front of them. These web servers receive all requests initially. They prioritize the requests and select the appropriate application server instance in the application server cluster to ultimately serve the request. The user session state is typically stored in a state server maintained in a database.

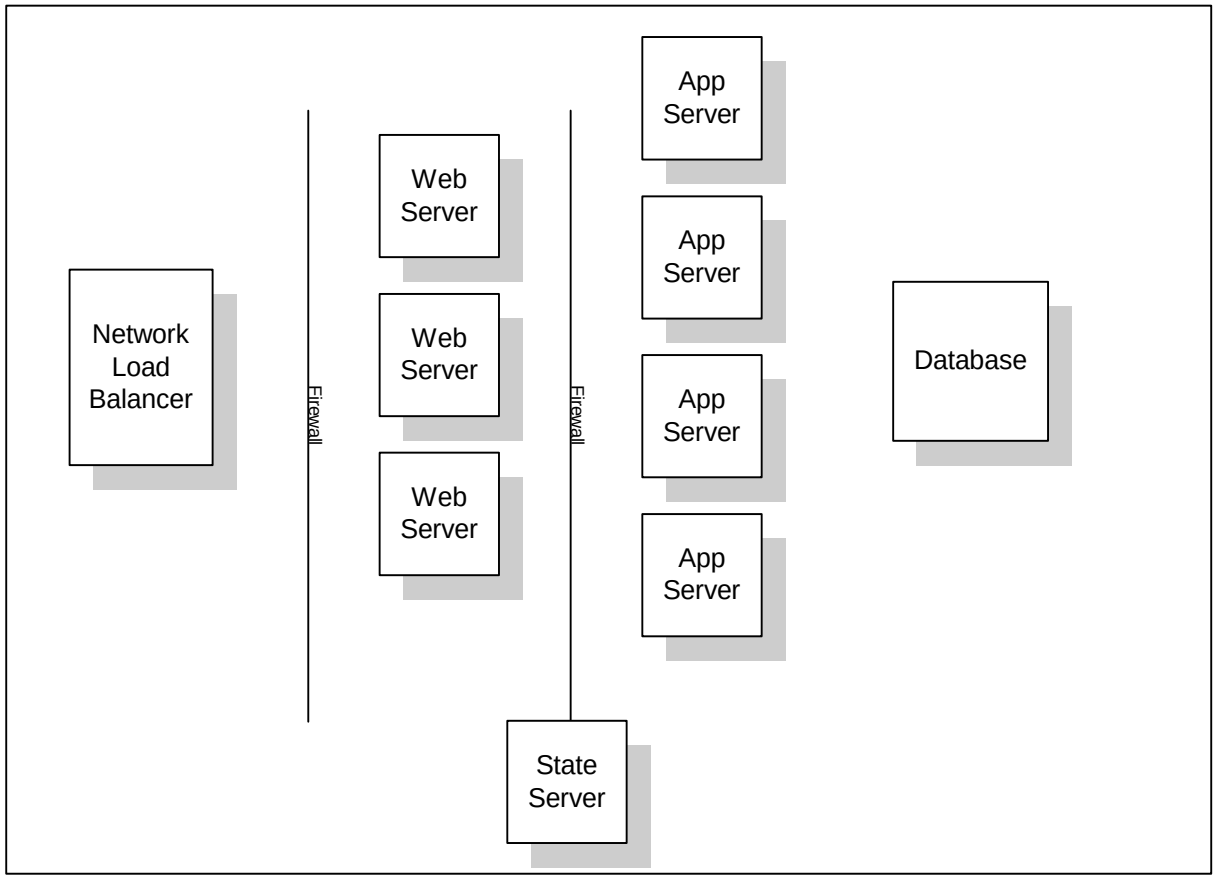

Figure 1. J2EE Compliant e-Service Architecture

We will initially establish our SLOs and define any prioritization that we want to implement for certain transactions. We will then compute some initial weights for the request queues based on expected average transaction times, the execution capacity of the different machines and the above preferences.

We will measure the execution times of instantiations of all transactions and feed them with the service level agreements into a measurement engine. The measurement engine will keep track of individual transaction execution times and will also aggregate the times for all transaction types. This consolidated

information is continuously fed into the controller. The controller analyses the data and from time to time adjusts the parameters on the e-Service by changing the weights on the different queues.

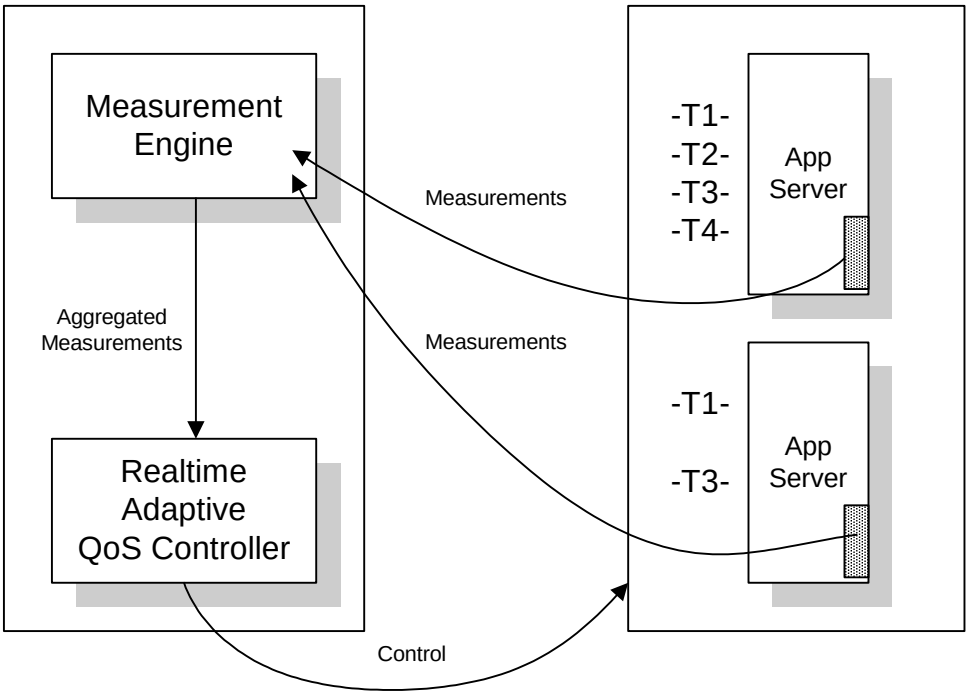

Figure 2. Adaptive control using real time measurements

The QoS management layer we propose, will be implemented between the load balanced web servers and the application servers. We will implement a set of logical queues representing the individual application servers. We will collect response time information in real time and aggregate them for the different transactions. We will compare the average response time for the individual transactions, the arrival density of new requests and the logic queue length to determine changes in the weights for sending different transactions to the queues. It will compare the calculated numbers with the specified QoS requirements. If necessary it will perform adaptations to the algorithm used to assign the requested transactions to the application servers. For example, there are two application servers in the architecture illustrated in figure 2. The upper server hosts transaction types T_1, T_2, T_3, T_4 while the lower one hosts T_1 and T_3. Suppose that the response time objectives on the upper server cannot be satisfied, and the measurement module finds out the problem was due to the higher-than-expected volumes of T_1 and T_2. One the other hand, the transactions on the lower server perform well. Then the measurement engine sends the aggregated metrics to the QoS controller. Based on the metrics, the controller executes the real-time load-balancing algorithm (see details in next section) and re-adjusts the weights for T_1 on each server. The weights should be adjusted in such a way, if possible, that enough amount of transaction load of T_1 will be shifted from the upper server to the lower one so that, given the current transaction loads, there will be no response time violations.

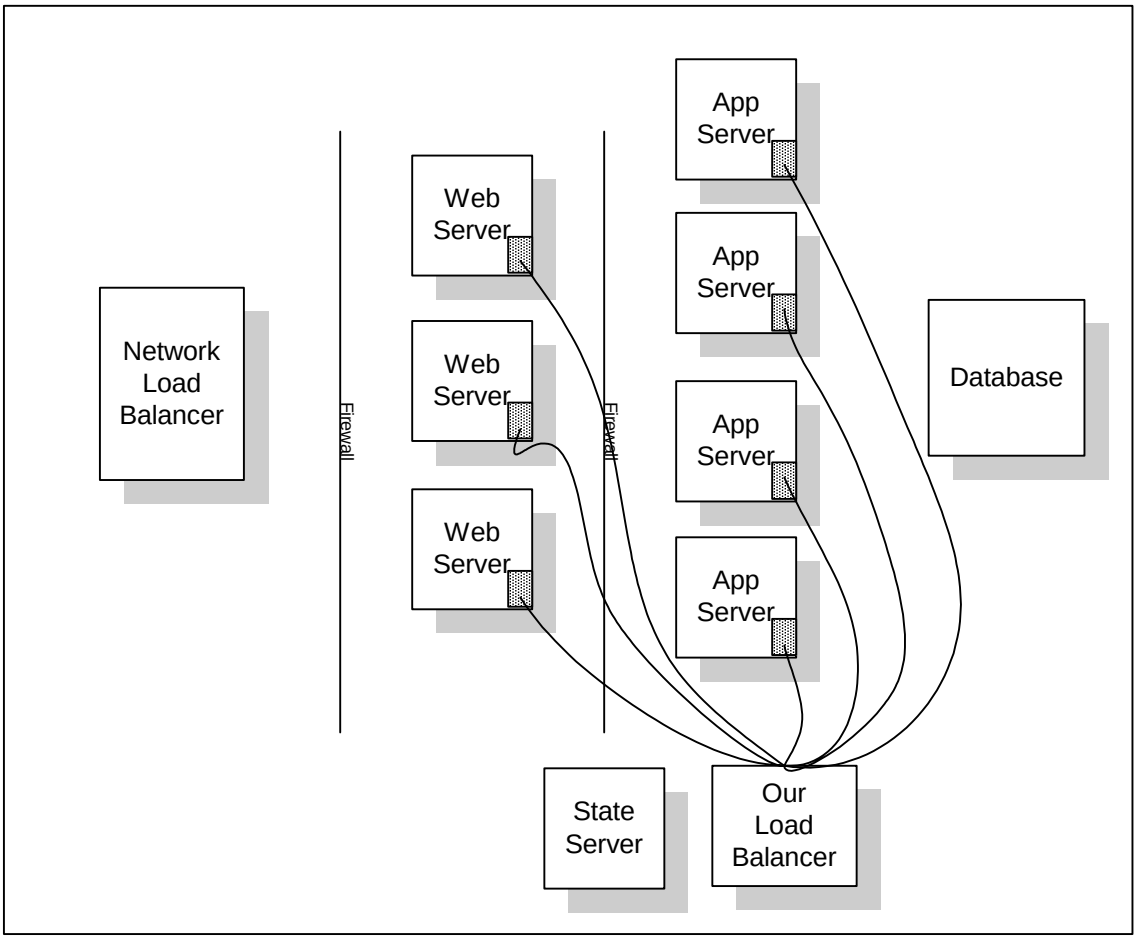

Figure 3. J2EE Compliant e-Service with measurement infrastructure

The possible control actions are:

- Sending transaction requests to different queues and thus to different application servers for processing based on their current capacities. Adjusting weights for particular transaction types on the queues controls this. By initially setting the weights in a specific fashion transaction differentiation becomes possible, e.g. it allows to send 'important' transactions to multiple servers while the 'normal' transactions might have to share a single server.

- Adding new application (nodes) servers from a pool of standby/failover servers to take the increased load. Nodes (servers) can also be retired/removed for planned or unplanned shutdown. This will enable handling of situations when particular nodes are failing or are over-saturated.

- Graceful degradation in case of a load that cannot be handled even after activating all the standby resources. Prioritizing transactions and controlling the number of transactions of a particular type being executed on a particular node allows for discrimination of transactions to be delayed or to be dropped completely in case of overload.

Each of these control actions have to be motivated by certain decision-making algorithms based on real time measurements. We describe an analytical model in the next section to make the above-mentioned control decisions.

3.3 BizQoS Analytical Model

We define a theoretical model on which the QoS management layer is built. The model is used for specifying, measuring, and guaranteeing quality of service in distributed systems. We assume the following characteristics in the distributed system.

- The system consists of a number of homogeneous or heterogeneous computing nodes.
- An application program can be replicated on one or more nodes, and the running of each replica will produce the same result.
- Applications are running to service requests. Requests are either from the outside world or from within the distributed system[1].
- Each node logically has a queue to buffer the incoming requests for potentially different services.

We define the nodes in the distributed system as $\{N_0, N_1, \ldots, N_{k-1}\}$ where k is the number of nodes. The queue associated with node N_i is defined as Q_i $(0 \le i \le k-1)$. Application programs hosted in the system are termed services, and we define the set of hosted services as $\{S_0, S_1, \ldots, S_{l-1}\}$ where l is the number of services currently hosted in the system. For each service S_i $(0 \le i < l)$, it can be replicated on the multiple nodes, and a replica is defined as S_i^n where n is the index of the hosting node. A service may provide a number of functions, and each function may consist of one or more processing units. We term, a transaction, the processing units corresponding to a service request. For each service S_i $(0 \le i < l)$, its supported transactions are defined as $\{T_0^i, T_1^i, \ldots, T_{m-1}^i\}$ where m is the number of transactions implemented in the service. For each transaction T_j^i $(0 \le j < m)$, its incarnation on node N_n is defined as $T_j^{i,n}$ where j identifies the type of the transaction (T_j^i), i represents the service implementing the transaction (S_i), and n is the index of the node where the transaction/service is replicated.

While node, service, and transaction represent three different levels of granularity for modeling quality of service in a distributed system, specification, measurement, and control are three interdependent phases for the QoS management. We describe in the following these sub modules.

3.3.1 Specification

Quality of service specification is a set of agreements between client and server or peer-to-peer, which can contain performance, scalability, availability, and reliability objectives. These specifications are used by servers for monitoring compliance. While monitoring compliance the servers take certain control actions to match up to the specifications agreed to with clients. Clients on the other hand use the specification to expect certain level of service from the servers. Failure to adhere to the specification should lead to some kind of compensation for the clients.

The set of agreements as mentioned in the specification are negotiated at both service and transaction levels. The QoS specification for a service consists of the following parameters.

Service priority. It is described by the equation:

$$S.priority = P, 0 \le P \le 1$$

[1] In the architecture requests are from the outside world (e.g., clients, or other remote e-services).

More computing resources will be allocated for a service with a higher priority than that for a lower priority service. For instance, it could mean the service will have a higher degree of replication, or be hosted on more powerful nodes.

Availability. It is described by the formula:

$$S.availability = \frac{U}{I}100\%$$

where I is a specific time interval, and U is the least amount of uptime expected from a service during an interval. This parameter specifies the agreement on the percentage of the uptime for a service.

Reliability. It is described by the formula:

$$S.reliability = \frac{C_c}{C_t}100\%$$

This definition means that, for every C_t number of transactions initiated by a service, it is expected that at least C_c number of transactions will successfully be finished. That is the rate of successfully finished transactions for a service.

As described above, a service consists of a list of transactions it supports. The QoS specification for a transaction in a service contains the following parameters.

Transaction priority. The priority of a transaction is defined as:

$$T.priority = P, 0 \le P \le 1$$

T.priority has the same semantics with *S.priority.* The priorities for transactions should be set up to meet the following requirements: 1. The priority of any transaction in a service is no higher than the service's priority. 2. The priority of any transaction in the first service is lower than that of any transaction in the second service if the first service has a lower priority.

$$T_j^i.priority \le S_i.priority$$

and

$$If \quad S_i.priority < S_k.priority \ \ then \ T_j^i.priority < T_l^k.priority$$

User priorities. For each type of transaction, there may be different types of users with different priorities. The user priorities for a type of transaction can be specified as:

$$\{T.user_{u.}priority = P, 0 \le P \le 1 \mid u = 0,1,\cdots,U-1\}$$

The priorities for user types should be set up to meet the following requirements: 1. The priority of any user type with a transaction type is no higher than the transaction's priority. 2. The priority of any user type with the first transaction type is lower than that of any user type with the second transaction type if the first transaction type has a lower priority.

$$\forall u, T_j^i.user_u.priority \le T_j^i.priority$$

and

$$If \;\; T_j^i.priority < T_l^k.priority \;\; then \;\; \forall u, T_j^i.user_u.priority < T_l^k.user_u.priority$$

Transaction response time, transaction density, and compliance rate. For each type of transaction with a specific service and a user type, its performance requirement can be described using these three parameters. That is, under a certain transaction density on a node, there should be at least the percentage of transactions, specified by the compliance rate, which do not violate the response time requirement. The three parameters are related to each other, and are given by

$$T.resptime = R, 0 < R < \infty$$

and

$$T.density = \frac{C}{I}$$

and

$$T.compliance = \mu, 0 < \mu \le 1$$

where C is the number of T's instances initiated during a measurement interval I. *T.resptime* specifies the average response time expected from a type of transaction. It is the waiting time in the queue plus the service time on the node. *T.density* specifies the maximum transaction density (i.e., the number of concurrent transactions per time unit) to be allowed on a specific node such that at least *T.compliance* percent of transactions whose response times are no greater than *T.resptime*. *T.resptime*, *T.density*, and *T.compliance* can usually be obtained through capacity planning on the computing nodes in the system. As the nodes may be heterogeneous, the values of *T.density* depend on the capacity of the hosting nodes—each hosting node has its own agreement on transaction density.

We will describe in next section how the density for a type of transaction can be calculated on a specific node hosting multiple types of transactions.

Availability. It is described by the formula:

$$T.availability = \frac{U}{I}100\%$$

where I is a specific time interval, and U is the least amount of uptime expected from a type of transaction during an interval. This parameter specifies the agreement on the percentage of the uptime for a type of transaction. We term that a service is not available if each of its hosted transactions is unavailable.

Reliability. It is defined as:

$$T.reliability = \frac{C_c}{C_t}100\%$$

This definition means that, for every C_t number of T's instances being initiated, it is expected that at least C_c number of instances will successfully be finished. That is the rate of successfully finished

transactions for a transaction type. The reliability requirement of a service should be the lower bound of that of its hosted transactions.

3.3.2 Measurement

We describe in this section the set of metrics that need to be collected and calculated at transaction, service, and node levels. They are used to check if there are any violations on the QoS agreements or irregularities in the system. There are two types of metrics: raw and aggregate data. We formally present the metrics in the following, and will describe in next section how they can be used to provide better quality of service, or reinforce the QoS agreements in the event of violations.

3.3.2.1 Raw metrics

Transaction registration time. It is the time when a type of transaction is registered/initiated in a service. It is described by $T_j^{i,n}.regtime$.

Transaction failed time. It is the time when a type of transaction becomes unavailable in a service. It is described by $T_j^{i,n}.failedtime$.

Transaction start and stop times. Transaction start/stop time marks the start/stop time of a transaction instance. They are given by $t_{j,k}^{i,n}.startime$ and $t_{j,k}^{i,n}.stoptime$. After its start, the status of the instance is marked "in-progress" until it stops, either successfully ("committed") or in failure ("failed").

Transaction duration. For a finished transaction instance, it is the duration between its start and its stop. For a transaction that is still in-progress at the end of a measurement interval, it is the duration between its start and the end of the interval. $t_{j,k}^{i,n}.stime$ is described by:

$$t_{j,k}^{i,n}.stime = \begin{cases} t_{j,k}^{i,n}.stoptime - t_{j,k}^{i,n}.startime, t_{j,k}^{i,n}.status = committed \text{ or } failed \\ intvl.end - t_{j,k}^{i,n}.startime, t_{j,k}^{i,n}.status = inprogress \end{cases}.$$

3.3.2.2 Aggregate metrics

Aggregate metrics are derived from the basic raw metrics as described in the previous section. The aggregate metrics are used by the BizQoS controller to monitor the node, transaction, and service level details and to take certain control actions. The aggregate metrics calculated for this purpose are:

Transaction down time. The down time of a type of transaction is given by:

$$T_j^{i,n}.dtime = T_j^{i,n}.regtime - T_j^{i,n}.failedtime$$

$T_j^{i,n}.failedtime$ represents the time when the resource (i.e., the software implementing the transaction or the node hosting the transaction) fails. $T_j^{i,n}.regtime$ is the time when the resource is back up (i.e., when the type of transaction is re-registered).

Transaction residence count. It represents the number of transactions of a specific type existing during a measurement interval. It consists of the transactions starting before and within the interval. The residence count of a service transaction on a node is defined as:

$$T_j^{i,n}.residence = T_j^{i,n}.prev\_start + T_j^{i,n}.curr\_start$$

Transaction residence time. It is the average time for which a type of transaction resides in a measurement interval. It is described by:

$$T_j^{i,n}.etime = \frac{Sum(t_{j,k}^{i,n}.etime)}{T_j^{i,n}.residence}$$

$t_{j,k}^{i,n}$ is a transaction instance that exists during the measurement interval. $t_{j,k}^{i,n}.etime$ represents the amount of time that $t_{j,k}^{i,n}$ spends in the interval, and is described by:

$$t_{j,k}^{i,n}.etime = \begin{cases} t_{j,k}^{i,n}.stoptime - t_{j,k}^{i,n}.startime, [t_{j,k}^{i,n}.startime, t_{j,k}^{i,n}.stoptime] \subset intvl \\ t_{j,k}^{i,n}.stoptime - intvl.start, t_{j,k}^{i,n}.startime < intvl.start \\ intvl.end - t_{j,k}^{i,n}.startime, t_{j,k}^{i,n}.status = inprogress \text{ and } t_{j,k}^{i,n}.startime \geq intvl.start \\ intvl.end - intvl.start, t_{j,k}^{i,n}.status = inprogress \text{ and } t_{j,k}^{i,n}.startime < intvl.start \end{cases}.$$

Transaction waiting time. The waiting time of a transaction indicates the amount of time for which the corresponding request waits in the queue before being serviced. The transaction waiting time on node N_n is calculated in the following where $Q_n.length$ represents the average length of Q_n.

$$T_j^{i,n}.wtime = \frac{(intvl.end - intvl.start) * Q_n.length}{Sum(T_k^{l,n}.curr\_start)}$$

Transaction commit count. $T_j^{i,n}.commit$ represents the number of transactions of a specific type that finished successfully during a measurement interval.

Transaction violated commit count. $T_j^{i,n}.violation$ counts the number of transactions of a specific type that finished successfully, but violate the response time requirement, which is,

$$t_{j,k}^{i,n}.stime + T_j^{i,n}.wtime > T_j^i.resptime$$

Transaction in-progress count. $T_j^{i,n}.inprog$ counts the number of on-going transactions of a specific type by the end of a measurement interval.

Transaction violated in-progress count. $T_j^{i,n}.vinprog$ counts the number of on-going transactions of a specific type whose durations, by the end of a measurement interval, violate the response time requirement, which is,

$$t_{j,k}^{i,n}.stime + T_j^{i,n}.wtime > T_j^i.resptime$$

Transaction failed count. $T_j^{i,n}.failed$ counts the number of transactions of a specific type that failed during a measurement interval.

Transaction service time. It is the average duration for the set of transaction instances with a specific type during a measurement interval. Each instance in the set meets the following condition:

$$\forall i, j, k, t_{j,k}^{i,n}.status = committed, \text{ or } t_{j,k}^{i,n}.status = inprogress \text{ and } t_{j,k}^{i,n}.stime + T_j^{i,n}.wtime > T_j^i.resptime$$

$t_{j,k}^{i,n}$ is a transaction instance that either finishes successfully within the interval, or is still in-progress at the end of the interval whereas it has violated the response time agreement. The average transaction service time is thus given by.

$$T_j^{i,n}.stime = \frac{Sum(t_{j,k}^{i,n}.stime)}{T_j^{i,n}.commit + T_j^{i,n}.vinprog}$$

Transaction response time is the sum of transaction service time and transaction waiting time. That is the average time spent by a type of request on a node. It is defined as:

$$T_j^{i,n}.rtime = T_j^{i,n}.stime + T_j^{i,n}.wtime$$

Transaction violation rate. It represents the percentage for a type of transaction whose instances violate the response time agreement during a measurement interval.

$$T_j^{i,n}.vrate = \frac{T_j^{i,n}.violation + T_j^{i,n}.vinprog}{T_j^{i,n}.commit + T_j^{i,n}.vinprog}$$

Absolute transaction density for a type of transaction is the average number of concurrent transaction per time unit. It is defined as:

$$T_j^{i,n}.adensity = \frac{T_j^{i,n}.residence}{intvl.end - intvl.start}$$

Absolute transaction load. The absolute transaction load of a type of transaction is the total residence time for such type of transaction during a time interval. It is described by:

$$T_j^{i,n}.aload = T_j^{i,n}.residence * T_j^{i,n}.etime$$

Relative transaction load of a transaction type represents the ratio of its transaction load to all types of transaction load on the hosting node during the measurement interval.

$$T_j^{i,n}.rload = \frac{T_j^{i,n}.aload}{Sum(T_k^{l,n}.aload)}$$

Relative transaction density represents the density of a type of transaction with respect to its relative transaction load. It is described by:

$$T_j^{i,n}.rdensity = \frac{T_j^{i,n}.adensity}{T_j^{i,n}.rload}$$

Transaction saturation rate. This metrics indicates if and how much a specific type of transaction is overloaded on a node. It is the ratio of the relative transaction density to the transaction density specified in the QoS agreements.

$$T_j^{i,n}.saturation = \frac{T_j^{i,n}.rdensity}{T_j^{i,n}.density}$$

Node saturation rate. The metrics indicates if and how much a node is overloaded. That is the ratio of the sum of the density of each type of transaction on a node to the sum of the weighted density agreements, according to the measured transaction load distributions. The metrics is defined as:

$$N_n.saturation = \frac{Sum(T_j^{i,n}.adensity)}{Sum(T_j^{i,n}.density * T_j^{i,n}.rload)}$$

Transaction weight. It indicates how the workload for a type of transaction should be distributed among the hosting nodes. The initial value is the capacity of the hosting node. Next section will describe how this metrics can be changed and used by the control mechanism.

$$T_j^{i,n}.weight = \varpi, 0 \le \varpi \le 1$$

3.3.3 Control

The BizQoS control module takes necessary actions, when performance, availability, and reliability issues arise, to guarantee quality of service. There are two types of actions: automated and non-automated. If a performance, availability or reliability agreement for a service/transaction is violated on either a single node or the whole cluster, it usually means that some of the following problems occurred.

- The workload on a node or the cluster is overloaded, thus load redistribution or adding new resources is needed.
- The quality of the application software is not good enough, thus needs to be re-engineered.
- The system is not configured or set up properly.
- Some system components are malfunctioning, and need to be repaired or replaced.

In this section we focus on what automated actions can be taken when potential performance or availability/reliability problems occur. In our model, transaction requests are dispatched to different hosting hosts in a weighted round-robin fashion. In implementation, some type of request mapping table can be built, where each type of transaction with a specific service and a user type has an entry. The entry contains a node index, and the transaction weight on each of the hosting nodes. The node index is computed using the transaction weights. It indicates the node to which the next request for this

type of transaction will be dispatched, unless the corresponding queue is full. When a performance bottleneck occurs or a QoS agreement is violated, the transaction loads are re-adjusted among the hosting nodes, or distributed to new nodes if necessary and possible. Then the corresponding transaction weights are re-calculated. The new weights will be used to rout transaction requests for the new time interval. In our model, there are several ways to detect potential performance or availability/reliability problems.

- The average response time for a type of transaction measured on a node is greater than that specified in the agreement.
- The transaction compliance rate is not satisfied, i.e., the transaction violation rate is higher than expected.
- Transaction/node saturation rates are unbalanced in the system even if there are no QoS violations.

There are basically three causes to the problems: transactions on a node are overloaded, transaction loads are unbalanced among the hosting nodes, or a node (or some of its components) is malfunctioning (e.g., system-hang, low-memory due to memory leaks, etc). In order to diagnose the cause to the problem, the transaction saturation rate for each type of transaction on each hosting node is checked, starting from higher priority transactions to lower priority ones.

3.3.3.1 Transaction level guarantees

If $T_j^{i,n}.saturation > 1,$ it indicates that transactions with type T_j^i are overloaded on node N_n. The overload, in terms of transaction density and transaction count, is calculated as:

$$\partial = \frac{T_j^{i,n}.rdensity - T_j^{i,n}.density}{T_j^{i,n}.rdensity}$$

and

$$\Delta = \min(T_j^{i,n}.curr\_start,\ \partial * T_j^{i,n}.residence)$$

It indicates that the transaction density for $T_j^{i,n}$ should be reduced by ∂ for the new time interval so that the response time agreement could be satisfied. The absolute transaction load for $T_j^{i,n}$ should be modified as:

$$T_j^{i,n}.aload = T_j^{i,n}.aload * \theta$$

where

$$\theta = 1 - \frac{\Delta}{T_j^{i,n}.residence}$$

As a result, the relative transaction load for each type of transaction hosted on N_n needs to be adjusted, using the formula for relative transaction load. Further, $T_j^{i,n}$'s absolute density should be changed as:

$$T_j^{i,n}.adensity = \frac{T_j^{i,n}.residence - \Delta}{intvl.end - intvl.start}$$

Then the relative density and saturation rate for each type of transaction hosted on N_n needs to be adjusted, using the density and saturation formulas respectively. Finally, $N_n.saturation$ is re-calculated using its formula.

To reduce $T_j^{i,n}$'s load on N_n, the transaction weight for $T_j^{i,n}$ must also be adjusted accordingly, and it is calculated as:

$$T_j^{i,n}.weight = T_j^{i,n}.weight(1 - \frac{\Delta}{T_j^{i,n}.curr\_start})$$

After quantifying the overload, we need to find out if any other nodes have extra capacity to handle the extra load while not compromising the QoS agreements of the existing transactions.

There are three possible scenarios: 1. Other hosting nodes can absorb the load (load balancing); 2. Some nodes that did not previously host the type of transaction can absorb the load (or adding new nodes); 3. No other nodes can handle this overload without sacrificing the performance of their hosted transactions (graceful degradation).

3.3.3.1.1 Redistribution among existing nodes

In the first scenario, we assume there are *m* nodes each of which meets the following conditions: each of these nodes is not saturated, and T_j^i is not saturated on each of these nodes.

$$\forall k \in \Omega(m), N_k.saturation < 1 \text{ and } T_j^{i,k}.saturation < 1$$

The overload is distributed among the *m* nodes such that the resulting transaction and node saturation rates on each node would not be greater than 1 (one simple strategy would be to equally distribute the load if possible). Suppose that the quota N_k receives is Δ' out of Δ, the absolute transaction load for $T_j^{i,k}$ should be modified as:

$$T_j^{i,k}.aload = T_j^{i,k}.aload * \theta'$$

where

$$\theta' = 1 + \frac{\Delta'}{T_j^{i,k}.residence}.$$

As a result, the relative transaction load for each type of transaction hosted on N_k needs to be adjusted, using the formula for relative transaction load. Further, $T_j^{i,k}$'s absolute density should be changed as:

$$T_j^{i,k}.adensity = \frac{T_j^{i,k}.residence + \Delta'}{intvl.end - intvl.start}$$

Then the relative density and saturation rate for each type of transaction hosted on N_k needs to be adjusted, using the density and saturation formulas respectively. Finally, $N_k.saturation$ is re-calculated using its formula.

To increase $T_j^{i,k}$'s load on N_k, the transaction weight for $T_j^{i,k}$ must also be adjusted accordingly, and it is calculated as:

$$T_j^{i,k}.weight = T_j^{i,k}.weight(1+\frac{\Delta'}{T_j^{i,k}.curr\_start})$$

After re-distributing the overload, the transaction weight for T_j^i on each hosting node is normalized as follows.

$$\forall k, T_j^{i,k}.weight = \frac{T_j^{i,k}.weight}{\max(\Omega(T_j^{i,k}.weight))}$$

where $\Omega(T_j^{i,k}.weight)$ is the set of transaction weights, and function *max* returns the maximum value.

3.3.3.1.2 Adding new nodes

In the second scenario, suppose that there are *K* nodes that did not previously host T_j^i and meet the following conditions: each node and its hosted transactions if any are not saturated (for a new node, its saturation rates are zero).

$$\forall k \in \Omega(K), N_k.saturation < 1 \text{ and } \forall l, m, T_m^{l,k}.saturation < 1$$

The overload is distributed among the *K* nodes such that the resulting saturation rate of each node would not be greater than 1, and be as close as possible. Suppose that the quota N_k receives is Δ' out of Δ, the absolute transaction load for $T_j^{i,k}$ on N_k would be:

$$T_j^{i,k}.aload = \Delta' * \frac{(T_j^i.resptime - T_m^{l,k}.wtime)T_j^{i,n}.etime}{T_j^{i,n}.stime}, T_j^i \neq T_m^l$$

For a new empty node, $T_m^{l,k}.wtime$ is not available, and $T_j^{i,n}.wtime$ is used instead. As a result, the relative transaction load for each type of transaction hosted on N_k needs to be adjusted, using the formula for relative transaction load. Further, $T_j^{i,k}$'s density should be calculated as:

$$T_j^{i,k}.adensity = \frac{\Delta'}{intvl.end - intvl.start}$$

Then the relative density and saturation rate for each type of transaction hosted on N_k needs to be adjusted, using the density and saturation formulas respectively. Finally, $N_k.saturation$ is re-calculated using its formula.

As N_k takes Δ' out of the total overload Δ, the transaction weight for T_j^i on N_k should be set to:

$$T_j^{i,k}.weight = T_j^{i,n}.weight \frac{\Delta'}{T_j^{i,n}.curr\_start - \Delta}$$

After re-distributing the overload, the transaction weight for T_j^i on each hosting node is normalized as described in the first scenario.

3.3.3.1.3 Graceful degradation

In the third scenario where each node in the system is saturated, the only thing that could be done is to squeeze the resources for the lower-priority transactions. Therefore, the performance of these low-priority transactions will be degraded. The idea is to push an equivalent amount of load for some lower-priority transactions to other hosting nodes, if each of these transactions, $T_m^{l,n}$, meets the following conditions.

1. It has a lower priority.
2. There exists another hosting node N_k for T_m^l where the priority of each hosted transaction is less than that of T_j^i.

To simplify the discussion, we suppose that $T_m^{l,n}$ is the only transaction satisfying the above conditions. We pull out the equivalent amount of load for $T_m^{l,n}$ from N_n, and push it to N_k. That is:

$$\Delta' = \frac{\Delta * T_j^{i,n}.etime}{T_m^{l,n}.etime}$$

The same algorithms used in the first scenario can be used here to adjust the relevant metrics on N_n and N_k accordingly. Note that the "push" strategy can also be used as an alternative to the algorithms described in the second scenario. The only difference is, each pushed $T_m^{l,n}$ should meet the condition: There exists at least another hosting node N_k for T_m^l where neither N_k nor $T_m^{l,k}$ is saturated, $N_k.saturation < 1$ and $T_m^{l,k}.saturation < 1.$

Consider the time complexity of the above algorithms redistributing T_j^i's load. Let *n* be the number of nodes where T_j^i is overloaded, and *m* be the number of nodes that can absorb extra load. Let *k* be the maximum number of transactions hosted on a node. As transaction load, density, and saturation rate, for each type of transaction on both source and destination nodes, need to be re-computed, the time complexity of redistributing T_j^i's load among the *n* and *m* nodes is $O(nmk)$. Let *l* be the number of

transaction types in the system whose loads need to be redistributed, the time complexity of the algorithms is $O(nmkl)$. Let N be the total number of nodes in the system, and T be the total number of transaction types in the system. As $n \le N, m \le N,$ and $k \le T, l \le T,$ the complexity of the algorithms is bounded by $O(N^2T^2)$.

3.3.3.2 Dealing with failing nodes

If $T_j^{i,n}$'s response time agreement is violated and $T_j^{i,n}.saturation < 1,$ it indicates that the violation is not caused by the overload. Instead, there might be some related system and/or software components that are not performing normally (e.g., system-hang, low-memory due to memory leaks, etc). So when the above situation occurs, an alarm should be generated—the problematic component, depending on if it is a temporary or permanent problem, may need to be repaired or replaced. Besides, $T_j^{i,n}$'s weight should be adjusted so that the situation will not deteriorate (at least not as fast). The adjustment may not be accurate because it is almost impossible to quantify the impact a fault imposes on the resources.

If $T_j^{i,n}.inprog \neq 0,$ it indicates that there are instances of $T_j^{i,n}$ which are still in-progress. $T_j^{i,n}$'s weight should temporarily be set to 0. If the in-progress transactions can finish successfully, $T_j^{i,n}$'s weight will be set to an empirical value lower than its previous weight (e.g., $T_j^{i,n}.weight = T_j^{i,n}.weight/2$). Otherwise, the problem must be fixed before $T_j^{i,n}$ can service any new requests.

If $T_j^{i,n}.inprog = 0$ and $T_j^{i,n}.failed \neq 0,$ it indicates that some instances of $T_j^{i,n}$ have failed during the current measurement interval. $T_j^{i,n}$ should not service any new request before the problem is fixed. Thus its weight is set to zero.

If $T_j^{i,n}.inprog = 0,$ and $T_j^{i,n}.failed = 0,$ it indicates that no failures have occurred during the current measurement interval. But the hosting node is not performing normally as indicated by the response time violation. Thus, before the problem is fixed, $T_j^{i,n}$'s weight should be reduced to a lower empirical value.

After adjusting T_j^i's weight, the transaction weight for T_j^i on each hosting node should be normalized as described before.

Consider the time complexity of the above algorithm. Let *m* be the number of nodes where T_j^i's response time agreement is violated, and *n* be the number of nodes hosting T_j^i. The time complexity of adjusting and normalizing T_j^i's weights on its hosting nodes is $O(m+n)$. Let *l* be the number of troubled transaction types in the system, the complexity of the algorithm is $O((m+n)l)$, and it is bounded by $O(NT)$ where N is the total number of nodes in the system, and T is the total number of transaction types in the system.

Conclusion

BizQoS Management Layer is used to specify, measure, and guarantee quality of service at service level. The model is based on real-time measurement and historical metrics analysis, and it presents a quantitative method to identify and remedy performance, availability, and reliability problems in order to guarantee Quality of Service. The approach is unique in the sense that it proposes a real-time close-loop adaptive control mechanism based on service and transaction level data.

References

[1] HP Bluestone Total E-Server. http://www.bluestone.com

[2] J2EE specification. http://java.sun.com/j2ee/download.html

[3] BEA Web Logic http://www.bea.com

[4] T.F. Abdelzaher and N. Bhatti. Web Server QoS Management by Adaptive Content Delivery. In Seventh International Workshop on Quality of Service (IWQoS'99), May 1999.

[5] R. Braden, L.Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP). RFC 2205, IETF, September 1997

[6] D. Clark, S. Shenkar, and L. Zhang. Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism. IN SIGCOMM Symposium on Communications Architectures and Protocols, pages 14-26, August 1992.

[7]C. Dovrolis and P. Ramanathan. A Case for Relaqtive Differentiated Services and the Proportional Differentiation Model. IEEE Network, October 1999.

[8] L. Eggert and J. Heidemann. Application-Level Differentiated Services for Web Servers, World Wide Web Journal, 2(3):133-142, August 1999

[9] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. RFC 2475, IETF, December 1998.

[10] J. Almeida, M.Dabu, A. manikutty, and P.Cao. Providing Differentiated Levels of Service in Web content Hosting. In Proceedings of the 1998 SIGMETRICS Workshop on Internet Server Performance, March 1998.

[11] M. Aron, D. Sanders, P. Druschel, W. Zwaenepoel. Scalable Content-aware Request Distribution in Cluster-based Network Servers. In Proceedings of USENIX 2000 Annual Technical Conference, San Diego, CA, June 2000.

[12] A. Sahai, J. Ouyang, V. Machiraju. End-to-End Transaction Management for Composite Web Services. In proceedings of Third International Workshop on Advanced Issues of E-Commerce and Web based Information Systems. San Jose, CA, June 2001 (to appear).

[13]. ARM Working group. Application Response Measurement API. http://www.opengroup.org/onlinepubs/009619299/