acmqueue

Condos and Clouds

Constraints in an environment empower the services

Pat Helland, Salesforce.com

Living in a condominium (commonly known as a condo) has its constraints and its services. By defining the lifestyle and limits on usage patterns, it is possible to pack many homes close together and to provide the residents with many conveniences. Condo living can offer a great value to those interested and willing to live within its constraints and enjoy the sharing of common services.

Similarly, in cloud computing, applications run on a shared infrastructure and can gain many benefits of flexibility and cost savings. To get the most out of this arrangement, a clear model is needed for the usage pattern and constraints to be imposed in order to empower sharing and concierge services. It is the clarity of the usage pattern that can empower new PaaS (Platform as a Service) offerings supporting the application pattern and providing services, easing the development and operations of applications complying with that pattern.

Just as there are many different ways of using buildings, there are many styles of application patterns. This article looks at a typical pattern of implementing a SaaS (Software as a Service) application and shows how, by constraining the application to this pattern, it is possible to provide many concierge services that ease the development of a cloud-based application.

USAGE PATTERNS

Over the past 50 years, it has become increasingly common for buildings to be constructed for an expected usage pattern. Not all buildings fit this mold. Some buildings have requirements that are *so* unique, they simply need to be constructed on demand—steel mills, baseball stadiums, and even Super Walmarts are so specialized you can't expect to find one using a real estate agent.

Such custom buildings are becoming increasingly rare, however, while more and more buildings—whether industrial parks, retail offices, or housing—are being constructed in a common fashion and with a usage pattern in mind. They are built with a clear idea of *how* they will be used but not necessarily *who* will use them. Each has standard specifications for the occupants, and the new occupants must fit into the space.

CONSTRAINTS AND CONCIERGE SERVICES

A building's usage pattern may impose constraints but, in turn, offers shared concierge services. A condominium housing development, for example, imposes constraints on parking, noise levels, and barbequing. Residents can't work on garage projects or gardening projects. In exchange, someone is always on hand to accept their packages and dry cleaning. They may have a shared exercise facility and pool. Somebody else fixes things when they break.

An office building may have shared bathrooms, copy rooms, and lobby. The engineering for the building is typically common for the whole structure. To get these shared benefits, tenants may have a fixed office layout, as well as some rules for usage. Normally, people can't sleep at work, can't have pets at work, and may even have a dress code for the building.

A retail mall provides shared engineering, parking, security, and common space. An advertising budget may benefit all the mall tenants. In exchange, there are common hours, limits on the allowable retail activities, and constraints on the appearance of each store.

Each of these building types imposes constraints on usage patterns and offers concierge services in exchange. To enjoy the benefits, you need to accept the constraints.

Similarly, cloud computing typically has certain constraints and, hence, can offer concierge services in return. What can the shared infrastructure do to make life easier for a sharing application? What constraints must a sharing application live within to fit into the shared cloud?

WHAT IS CLOUD COMPUTING?

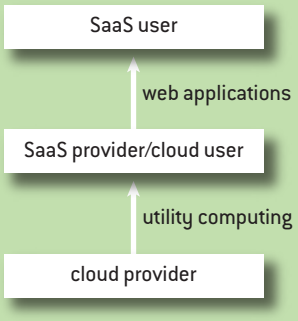
Cloud computing delivers applications as services over an intranet or the Internet. A number of terms have emerged to characterize the layers of cloud-computing solutions.

- **SaaS.** This refers to the user's ability to access an application across the Internet or intranet. SaaS has been around for years now (although the term for it is more recent). What's new is the ability to project the application over the Web without building a data center.
- **PaaS.** This is a nascent area in which higher-level application services are supplied by the vendor with two goals: first, a good PaaS can make developing an application easier; second, a good PaaS can make it easier for the cloud provider to share resources efficiently and provide concierge services to the app. Today, the leading examples of PaaS are Salesforce's Force.com[5] and Google's App Engine.[2]
- **IaaS** (infrastructure as a service). Sometimes called *utility computing*, this is virtualized hardware and computing available over the Web. The user of an IaaS can access VMs (virtual machines) and storage to accompany them on demand.

Figure 1 shows the relationship between the cloud and SaaS providers and users. (The figure was derived from a technical report from the University of California at Berkeley, "Above the Clouds: A Berkeley View of Cloud Computing."[3]) As observed in "Above the Clouds," cloud computing has three new aspects: the illusion of infinite computing resources on demand; the elimination of upfront commitment by cloud users; and the ability to pay for computing resources on a short-term basis.

FIGURE 1

Cloud Computing, Utility Computing, and Software as a Service



Cloud computing allows the deployment of SaaS—and scaling on demand—without having to build or provision a data center.

PUBLIC AND PRIVATE CLOUDS

Clouds are about sharing. The question is whether you share *within* a company or go to a third-party provider and share *across* companies.

In a public cloud, a cloud-computing provider owns the data center. Other companies access their computing and storage for a pay-as-you-go fee. This has tremendous advantages of scale, but it is more challenging to manage the trust relationship. Trust ensures that the computing resources are available when they are needed (this could be called an SLA, or service-level agreement). In addition, there are issues of privacy trust in which the subscribing company needs to have confidence its private data will not be accessed by prying eyes. Demonstrating privacy is easier if the company owns the data center.

A public cloud can project its shared resources as VMs and low-level storage requiring the application to build on what appears to be a pool of physical machines (even though they are really virtual). This would be a public-cloud IaaS. Amazon's AWS (Amazon Web Service)[1] is a leading example of this.

Alternatively, in a public-cloud PaaS, higher-level abstractions can be presented to the applications that allow finer-grained multi-tenancy than a VM. The shape and form of these abstractions are undergoing rapid evolution. Again, Force.com and App Engine are emerging examples.

In a private cloud the data center, physical machines, and storage are all owned by the company that uses them. Sharing happens within the company. The usage of the resources can ebb and flow as determined by different departments and applications. The size of the shared cloud is likely to be smaller than within a public cloud, which may reduce the value of the sharing. Still, it is attractive to many companies because they do not need to trust an outside cloud provider. So, far, we have seen only private-cloud IaaS. The new PaaS offerings are not yet being made available for individual companies to use in their private clouds.

THE FORCES DRIVING US TO THE CLOUD

A number of forces are prompting increased movement of applications to the cloud:

Data-center economics. Very large data centers can offer computation, storage, and networking at a relatively cost-effective price. Power is an ever-increasing portion of data-center costs, and it can be obtained more effectively by placing the data center near inexpensive sources of electricity such as hydroelectric dams. Internet ingress and egress is much cheaper near Internet main lines. Containerized servers with thousands of machines delivered in a shipping container offer lower cost for computation and storage. Shared administration of the servers offers cost savings in operations. All of this is included in the enormous price tag for the data center. Few companies can afford such a large investment. Sharing (and charging for) the large investment reduces the costs. This provides economic drive for both the cloud providers and users.

Shared data. Increasingly, companies are finding huge (and serendipitous) value in maintaining a "big-data" store. More and more, vast amounts of corporate data are placed into one store that can be addressed uniformly and analyzed in large computations. In many cases the value of the discoveries grows as the size of the data store increases. It is becoming a goal to store *all* of an enterprise's data in a common store, allow analysis, and see surprising value.

Shared resources. By consolidating computation and storage into a shared cloud, it is possible to provide higher utilization of these resources while maintaining strong SLAs for the higher-priority work. Low-priority work can be done during slack times while being preempted for higher-priority work during the busy times. This requires that resources are fluid and fungible so that the lower-priority work can be bumped aside and the resources reallocated to the higher-priority work.

SaaS: FRONT END, BACK END, AND DECISION SUPPORT

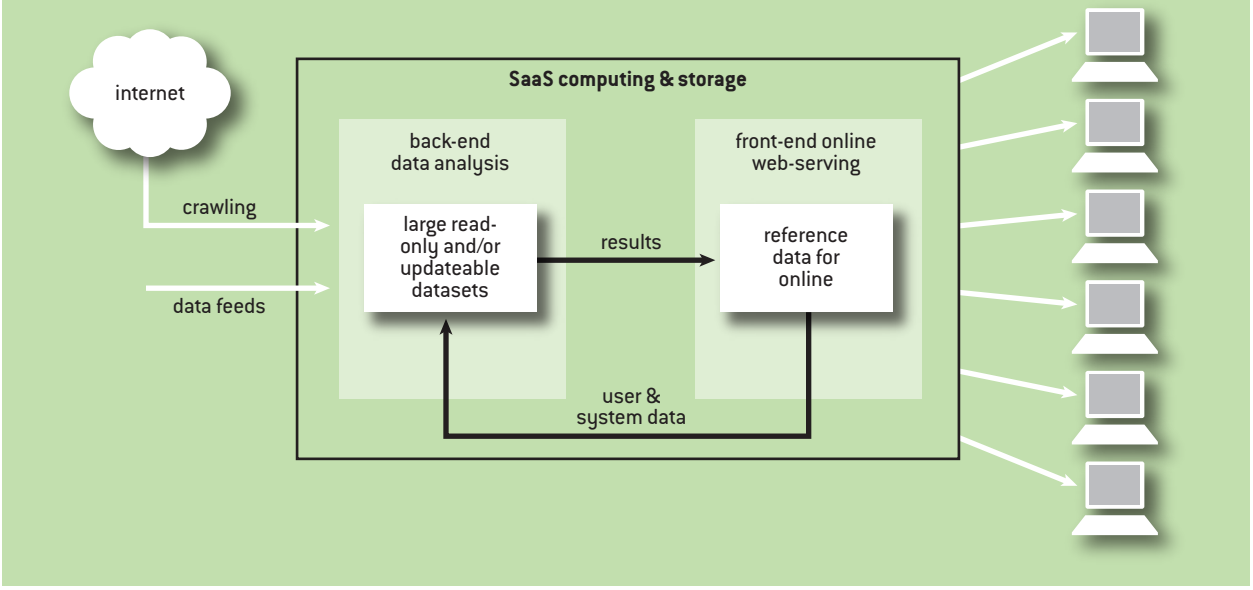
Let's look more closely at a typical pattern seen in a SaaS implementation. In general, the application has two major sections: the front end, which handles incoming Web requests; and the back end, which performs offline background processing to prepare the information needed by the front end. In addition to its work preparing data for the front end, the back-end application is usually shared with decision-support processing (see figure 2).

In a typical SaaS implementation, the front end offers user-facing services dealing with Web service or HTML. It is normal for this Web-serving code to have aggressive SLAs, typically of only 300-500 ms, sometimes even tighter. The back-end processing consumes crawled data, partner feeds, and logged information generated by the front end and other sources, and it generates reference data for use by the front end. You may see product catalogs and price lists as reference data, or you may see inverted search indices to support systems such as Google or Bing search. In addition to the generation of reference data, the back-end processing typically performs decision-support functions for the SaaS owner. These allow "what-if" analyses that guide the business.

PATTERNS IN SaaS APPS: THE FRONT END

This section explores a common pattern used in building the front-end portion of SaaS applications.

FIGURE 2 **Software as a Service Computing and Storage**



By leveraging the pattern used by these applications, a number of very useful concierge services can be supplied by the PaaS plumbing.

MANY SERVICE APPS FIT A PATTERN

Many service applications fit nicely within a pattern of behavior. The goal of these applications is to implement the front end of a SaaS application. Incoming Web-service requests or HTML requests arrive at the system and are processed with a request-response pattern using session state, other services, and cached reference data.

CONCIERGE SERVICES FOR THE FRONT END

When a front-end application fits into the constraints of the pattern just described, a lot of *concierge services* may be supplied to the application. These services simplify the development of the app, ease the operational challenges of the service, and facilitate sharing of cloud resources to efficiently meet SLAs defined for the applications. Some possible concierge services include:

Auto-scaling. As the workload rises, additional servers are automatically allocated for this service. Resources are taken back when load drops.

Auto-placement. Deployment, migration, fault boundaries, and geographic transparency are all included. Applications are blissfully ignorant.

Capacity planning. This includes analysis of traffic patterns of service usage back to incoming user workload. Trends in incoming user workload are tracked.

Resource marketplace. The concierge plumbing automatically tracks a service's cost as it consumes resources directly and indirectly (by calling other services). This allows the cost of shared services to be attributed and charged back to the instigating work.

A/B testing and experimentation. The plumbing makes it easy to deploy a new version of a service on a subset of the traffic and compare the results with the previous version.

Auto-caching and data distribution. The back end of the SaaS application generates reference data (e.g., product catalog and price list) for use by the front end. This data is automatically cached in a scalable way, and changes to items within the reference data are automatically distributed to the caches.

Session-state management. As each request with a partner is processed, the request has the option to record session state that describes the work in progress for that partner. This is automatically managed so that subsequent requests can easily fetch the state to continue work. The session-state manager works with dynamically scalable and load-balanced services. It implements the application's policy for session-state survival and fault tolerance.

Each of these concierge services depends on the application abiding by the constraints of the usage pattern as described for the typical front-end SaaS application.

STATELESS REQUEST PROCESSING

Incoming requests for a service are routed to one of many servers capable of responding to the request. At this point, no state is associated with the session present in the target server (we'll get it later if needed). It is reasonable to consider this a stateless request (at least so far) and select any available server.

The plumbing keeps a pool of servers that can implement the service. Incoming requests are

dynamically routed and load balanced. As demand increases and decreases, the concierge services of the plumbing can automatically increase and decrease the number of servers.

COMPOSITE REQUEST PROCESSING

Frequently, a service calls another service to get the job done. The called service may, in turn, call another service. This composite call graph may get quite complex and very deep. Each of these service calls will need to complete to build the user's response. As requests come in, the work fans out, gets processed, and then is collected again. In 2007, Amazon reported that a typical request to one of its e-commerce sites resulted in more than 150 service requests.[4] Many SaaS applications follow the pattern shown in figure 3a:

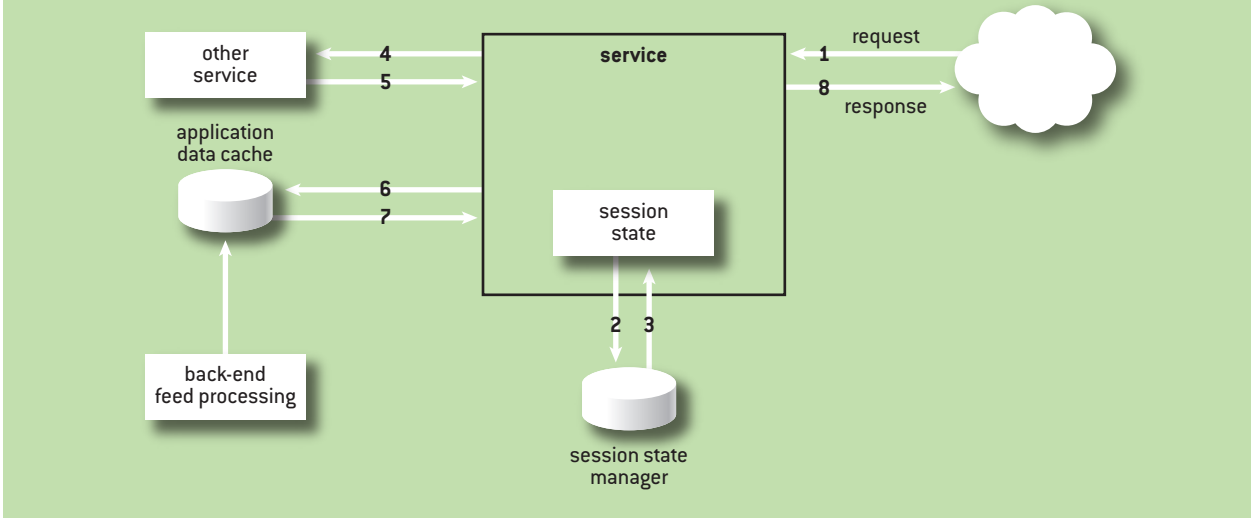
1. A request arrives from outside (either Web service or HTML).
2. The service optionally requests its session state to refresh its memory about the ongoing work.
3. The response comes back from the session-state manager.
4. Other services are consulted if needed.
5. The other service responds.
6. The application data cache (curated by the back-end processing) is consulted.
7. Cached reference data is returned to the service for use by its front-end app.
8. The response is issued to the caller.

SLAS AND REQUEST DEPTH

Requests serviced by the SaaS front end will have an SLA. A typical SLA may be a "300-ms response for 99.9 percent of the requests assuming a traffic rate of 500 requests per second."

It is common practice when building services to measure an SLA with a percentile (e.g., 99.9 percent) rather than an average. Averages are much easier to engineer and deploy but will lead to user dissatisfaction because the outlying cases are typically very annoying.

FIGURE 3A **The Typical Front-End SaaS Application Pattern**



POUNDING ON THE SERVICES AT THE BOTTOM

Implementing a system-wide SLA with a composite call graph puts a lot of pressure on the bottom of the stack. Because the time is factored into the caller's SLA, deeper stacks mean more pressure on the SLAs.

In many systems, the lowest-level services (such as the session-state manager and the reference-data caches) may have SLAs of 5 to 10 milliseconds 99.9 percent of the time. Figure 4 shows how composite call graphs can get very complex and put a lot of SLA pressure down the call stack.

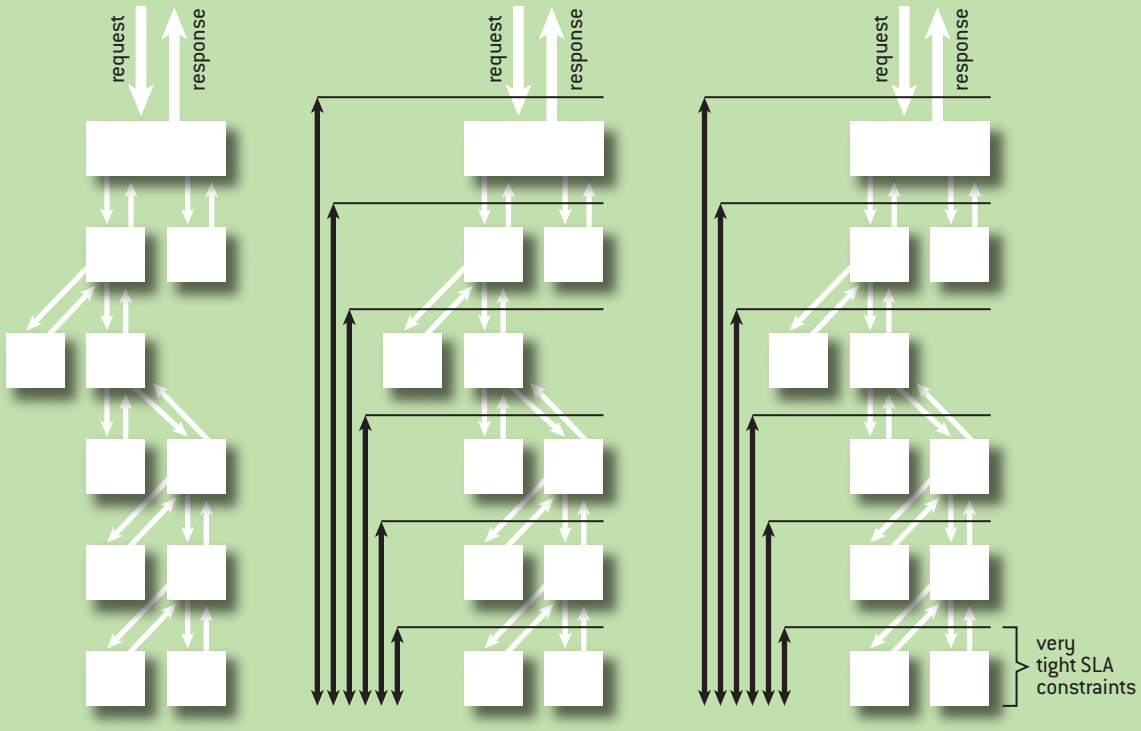
A QUICK REFRESHER ON SIMPLE QUEUING THEORY

The expected response time is dependent on both the minimum response time (the response time on an empty system) and the utilization of the system. Indeed, the equation is:

$$Expected\ Response\ Time = Minimum\ Response\ X\ \frac{Time}{1 - Utilization}$$

This makes intuitive sense. If the system is 50 percent busy, then the work must be done in the slack, so it takes twice the minimum time. If the system is 90 percent busy, then the work must get done in the 10 percent slack and takes 10 times the minimum time.

FIGURE 4 **Composite Call Graphs**



a composite call graph in an SaaS front-end can get very complex

to meet a system-wide SLA, each service deeper in the call stack must meet an ever-tighter SLA

The bottom of the call stack can be under enormous pressure to meet tight SLAs

AUTOMATIC PROVISIONING TO MEET SLAS

When the SLA for a service is slipping, one answer is to reduce the utilization of the servers providing the service. This can be done by adding more servers to the server pool and spreading the work thinner.

Suppose each user-facing or externally facing service has an SLA. Also, assume the system plumbing can track the calling pattern and knows which internal services are called by the externally facing services. This means that the plumbing can know the SLA requirements of the nested internal services and track the demands on the services deep in the stack.

Given the prioritized needs and the SLAs of various externally facing services, the plumbing can increase the number of servers allocated to important services and borrow or steal from lower-priority work.

ACCESSING DATA AND STATE

When a request reaches a service, it initially has no state other than what arrived with the request. It can fetch the session state and/or cached reference data if needed.

The session state provides information from previous interactions that this service had over the session. It is fetched at the beginning of a request and then stored back with additional information as the request is completing.

Most SaaS applications use application-specific information that is prepared in the background and cached for use by the front end. Product catalogs, price lists, geographical information, sales quotas, and prescription drug interactions are examples of reference data. Cached reference data is accessed by key. Using the key, the services within the front end can read the data. From the front end, this data is read only. The back-end portion of the application generates change to (or new versions of) the reference data. An example of read-only cached reference data can be seen on the Amazon.com retail site. Look at any product page for the ASIN (Amazon Standard Identification Number), a 10-character identifier usually beginning with “0” or “B”. This unique identifier is the key for all the product descriptions you see displayed, including images.

MANAGING SCALABLE AND RELIABLE STATE

The session state is keyed by a session-state ID. This ID comes in on the request and is used to fetch the state from the session-state manager. An example of session state is a shopping cart on an e-commerce site.

The plumbing for the session-state manager handles scaling. As the number of sessions grows, the session-state manager automatically increases its capacity.

This plumbing is also responsible for the durability of the session state. Typically, the durability requirements mandate that the session state survive the failure of a single system. Should this be written to disk on a set of systems in the cloud? Should it be kept in memory over many systems to provide acceptable durability? Increased durability costs more to implement and may require increased latency to ensure that the request is durable.

Typically, a session-state manager is used so frequently that it must provide a very aggressive SLA for both reads and writes. A 5- or 10-ms guarantee is not unusual. This means it is not practical to wait for the session state to be recorded on disk. It is common for the session state to be

acknowledged as successfully written when it is present on two or three replicas in memory. Shortly thereafter, it will likely be written to disk.

APPLYING CHANGES TO THE BACK END

Sometimes, front-end requests actually "do work" and apply application changes to the back end. For example, the user pushes Submit and asks for the work to be completed.

Application changes to the back end may be either synchronous, in which the user waits while the back end gets the work done and answers the request, or asynchronous, in which the work is enqueued and processed later.

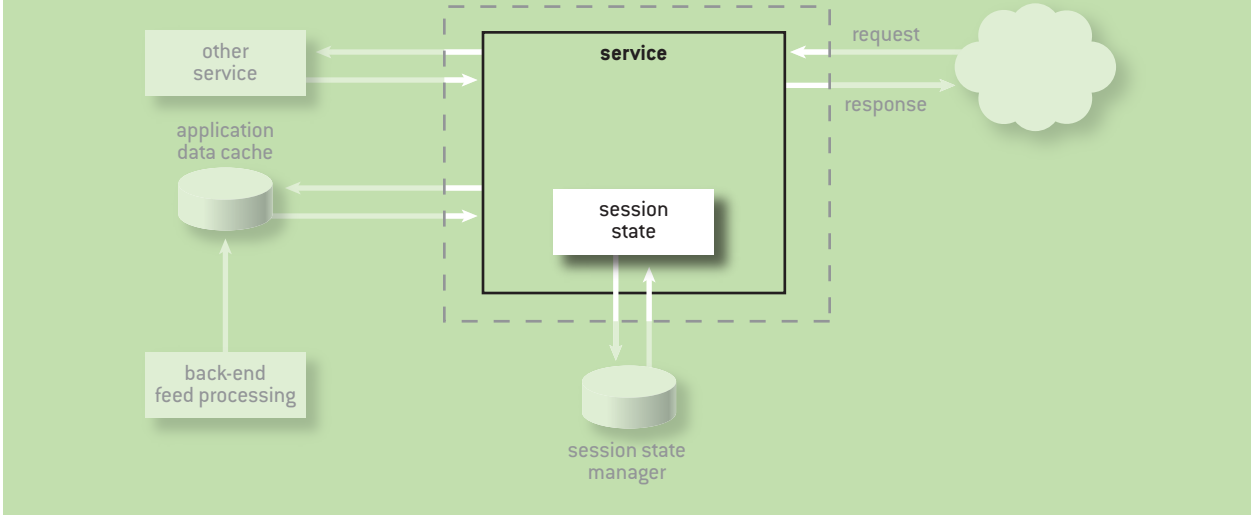
Amazon.com provides an example of asynchronous back-end app changes. When the user presses Submit, a portion of the front end quickly acknowledges the receipt of the work and replies that the request has been accepted. Typically, the back end promptly processes the request, and the user receives an e-mail in a second or two. Occasionally, the e-mail takes 30 minutes or so when the asynchronous processing at the back end is busy.

AUTOMATIC SERVICES, STATE, AND DATA

By understanding the usage pattern of a SaaS application, the platform can lessen the work needed to develop an application and increase its benefits. As suggested in figure 3b, the application should simply worry about its business logic and not about the system-level issues. Interfaces to call other services, access cached data, and access session state are easy to call. This provides support for a scalable and robust SaaS application.

The application session state, application reference-data cache, and calls to other services are available as concierge services. The platform prescribes how to access these services, and the application need not know what it takes to build them. By constraining the application functionality, the platform can increase the concierge services.

FIGURE 3B **The Application's Focus on Business Logic**



PATTERNS IN SaaS APPLICATIONS: THE BACK END

This section explores the patterns used in the back-end portion of a typical SaaS application. What does this back end do for the application? How does it typically do it?

FEEDS, CRAWLING, LOGGING, AND ONLINE WORK

The back end of a SaaS application receives data from a number of sources:

- **Crawling.** Sometimes the back end has applications that look at the Internet or other systems to see what can be extracted.
- **Data feeds.** Partner companies or departments may send data to be ingested into the back-end system.
- **Logging.** Data is accumulated about the behavior of the front-end system. These logs are submitted for analysis by the back-end system.
- **Online work.** Sometimes, the front end directly calls the back end to do some work on behalf of the front-end part of the application. This may be called synchronously (while the user waits) or asynchronously.

All of these sources of data are fed into the back end where they are remembered and processed.

DATA PUBLICATION AND UPDATES OF REFERENCE DATA

Many front-end applications use reference data that is periodically updated by the back end of the SaaS application. Applications are designed to deal with reference data that may be stale. The general model for processing reference data is:

1. Incoming information arrives at the back end from partners' feeds, Web crawling, or logs from system activity. Online work may also stimulate the back-end processing.
2. The application code of the back end processes the data either as batch jobs, event processing with shorter latency, or both.
3. The new entries in the reference-data caches are distributed to the caching machines. The changes may be new versions made by batch updates or incremental updates.
4. The front-end apps read the reference-data caches. These are gradually updated, and the users of the front end see new information.

AUTOMATING THE REFERENCE-DATA CACHE

The reference-data cache is a key-value store. One easy-to-understand model for these caches has partitioned and replicated data in the cache. Each cache machine typically has an in-memory store (since disk access is too slow). The number of partitions increases as the size of the data being cached increases. The number of replicas increases initially to ensure fault tolerance and then to support increases in read traffic from the front end.

It is possible to support this pattern in the plumbing with a full concierge service. The plumbing on the back end can handle the partition for data scale (and repartitioning for growth or shrinkage). It can handle the firing up of new replicas for read-rate scale. Also, the plumbing can manage the distribution of the changes made by the back end (either as a batch or incrementally). This distribution understands partitioning, dynamic repartitioning, and the number of replicas dynamically assigned to partitions.

Figure 5 illustrates how the interface from the back end to the front end in a SaaS application is typically a key-value cache that is stuffed by the back end and read-only by the front end. This clear pattern allows for the creation of a concierge service in a PaaS system, which eases the implementation and deployment of these applications.

Note that this is not the only scheme for dynamic management of caches. Consistent hashing (such as implemented by Dynamo[4], Cassandra[6], and Riak[8]) provides an excellent option when dealing with reference data. The consistency semantics of the somewhat stale reference data, which is read-only by the front end and updated by the back end, are a very good match. These systems have excellent self-managing characteristics.

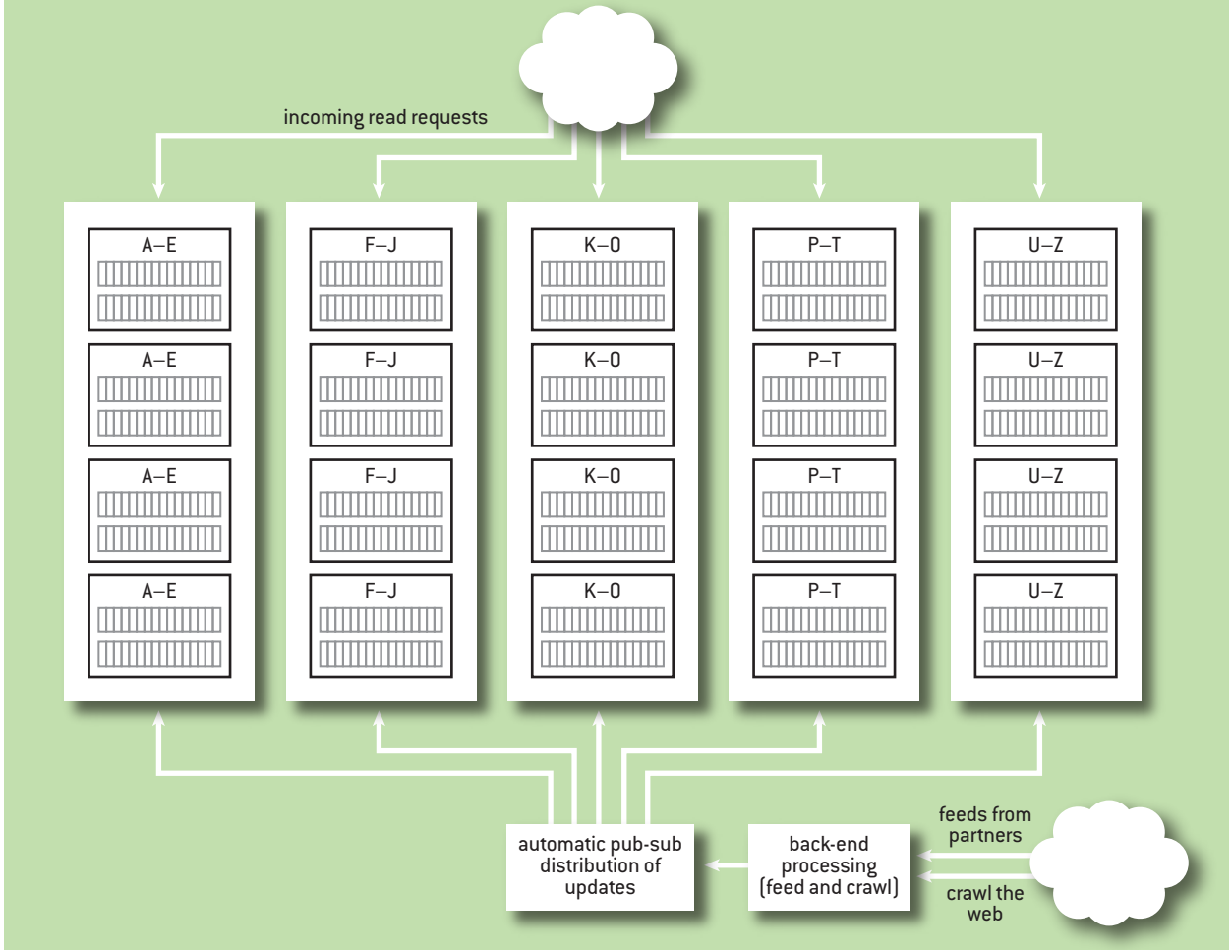
STYLES OF BACK-END PROCESSING

The back-end portion of the SaaS app may be implemented in a number of different ways, largely dependent on the scale of processing required. These include:

Relational database and normal app. In this case, the computational approach is reasonably

FIGURE 5

SaaS Application Interface from the Back-End to the Front-End



traditional. The data is held in a relational database, and the computation is done in a tried-and-true fashion. You may see database triggers, N-tier apps, or other application forms. Typically in a cloud environment, the N-tier or other form of application will run in a VM. This can produce the reference data needed for the front end, as well as what-if business analytics. This approach has the advantage of a relational database but scales to only a few large machines.

Big data and MapReduce. This approach is a set-oriented massively parallel processing solution. The underlying data is typically stored in a GFS (Google File System) or HDFS (Hadoop Distributed File System), and the computation is performed by large batch jobs using MapReduce, Hadoop, or some similar technology. Increasingly, higher-level languages declaratively express the needed computation. This can be used to produce reference data and/or to perform what-if business analytics. Over time, we will see MapReduce/Hadoop over *all* of an enterprise's data.

Big data and event processing. The data is still kept in a scalable store such as GFS or HDFS, and batch processing is available for both the production of reference data and for what-if analysis. What is interesting here is the emergence of the ability to recognize changes from feeds or from crawling the Web within seconds and perform transactional updates to the same corpus of back-end data that is available to the MapReduce/Hadoop batch jobs. A noteworthy example of this is the Google Percolator project.[7] Rapid processing of events into fresh reference data offers a vibrant Web experience.

CONCIERGE SERVICES FOR THE BACK END

When the back-end portion of the SaaS application follows the pattern described, it is possible to create a PaaS offering that can make it much easier to build, deploy, and manage the application. By living within the constraints of the pattern, many concierge services are made available such as:

Big-data unified data access. With unified enterprise-wide (and controlled cross-enterprise) data, anything may be processed with anything (if authorized).

Fault-tolerant and scalable storage. Cloud-managed storage for both big data and relational databases is available, with automatic intra-data-center and cross-data-center replication.

Massively parallel batch processing. High-level set operations perform large-scale computation.

Event processing. Low-latency operations with extremely high update rates for independent events are available. Transactional updates occur with more than tens of petabytes of data (see Percolator[7]). Event processing over the same corpus of data is available for batch processing.

Automatically scaling reference-data caches. The back end supplies the front end with dynamically updated application-specific data. The management and operations of the key-value cache are automatic. The plumbing ensures read SLAs by the front end are maintained by increasing replication of the caches as necessary.

Multitenanted access control. Both inter-enterprise (public cloud) and intra-enterprise (private cloud) require access control to the data contained in the shared stores.

Prioritized SLA-driven resource management. Relational databases, big-data batch, and big-data event processing compete for the same computational resources. Different organizations have different workloads, all competing for resources. The various workloads are given an SLA and a priority, and the tradeoffs are then automated by the plumbing.

THE DRIVE TOWARD COMMONALITY IN COMPUTING

Just as buildings have commonality, as mentioned previously, there is commonality within different

classes of computing environments. The different classes must hook up to the community, just as happens in buildings. Increasingly, the cloud environment is facilitating these changes.

SILOS AND SOA

Today's enterprises glue together silos of applications using SOA (service-oriented architecture). Silos are collections of applications that work on a shared database. SOA is the application integration that subsumes both EAI (enterprise application integration) and B2B (business-to-business) communication using messaging and data feeds. Figure 6 shows how applications with shared traits will congregate in the SOA service bus. Current enterprise applications will leverage their commonality as they migrate to the cloud. Certainly, existing applications that have not been designed for cloud deployment cannot get all the same benefits as those that follow the new and emerging patterns. Still, there is a way of leveraging the existing commonality and gain new benefits via cloud deployment, within either a public or private cloud.

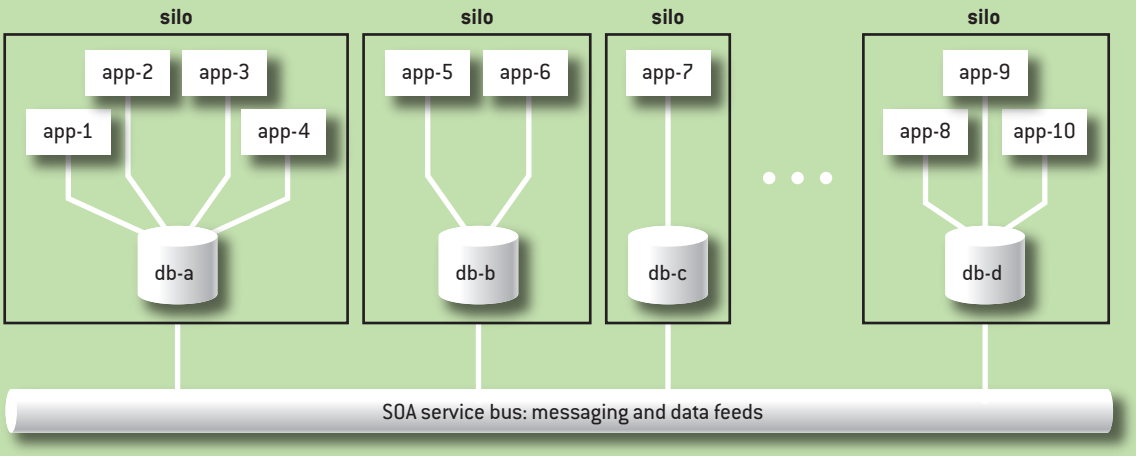
Cloud computing will help drive silos and SOA to a common representation:

Relational databases. Relational databases will be supported on standardized servers with standardized storage running over SANs (storage area networks). This will allow robust storage to underlie restartable database servers.

Existing apps using VMs. Because existing applications have their own expectations of the processing environment, those expectations are most easily met in a general fashion using a VM. The VM looks just like a physical machine to the application software but can be managed within the cloud's cluster. The application may get one machine or a pool of machines.

SOA messaging through the big-data store. The large cloud-based store can receive the messages and data feeds used to connect the enterprise's silos. By plumbing the SOA service bus through the big-data store, the information about the interaction of the silos is available for analysis. This class of analysis has been shown to be invaluable in the insight it brings to the enterprise.

FIGURE 6 **Silos and SOA**



Standardized monitoring and analysis. As the pieces of the enterprise's computing move to common hosting in a private or public cloud, the behavior of the applications, interactions across them, and usage patterns of the apps can be tracked. Capturing this in the big-data store allows for analysis in a fashion integrated with the application's messaging data and other enterprise knowledge.

DATABASES AND BIG DATA

Relational databases offer tightly controlled transactions and full relational semantics. They do, however, face challenges scaling beyond a handful of servers. Still, relational databases have more than 30 years of investment in applications, operations, and skills development that will survive for many years.

The emerging big-data stores as represented by MapReduce and Hadoop offer complementary sets of advantages. Leveraging massive file systems with highly available replicated data, these environments offer hundreds of petabytes of data that may be addressed in a common namespace. Recently, updatable key-value stores have emerged that offer transaction-protected updates.[7] These enormous systems are optimized for sharing with multiple users accessing both computational and storage resources in a prioritized fashion.

Increasingly, these benefits of the big-data environments will be applied to copies of the relational database data used within existing applications. The integration of the line-of-business relational data with the rest of the enterprise's data will result in a common backplane for data.

IT DEPARTMENTS AND LINE-OF-BUSINESS: A NATURAL TENSION

The line-of-business department in an enterprise drives the development of new applications. This is the department that needs the application and the solutions it provides to meet a business requirement. The department funds the application and, typically, is not too concerned with how the application will fit into the rest of the enterprise's computational work.

The IT department, on the other hand, has to deal with and operate the application once it is deployed. It wants the application, database, and servers to be on common ground. It needs to integrate the application into enterprise-wide monitoring and management.

This natural tension is similar to that seen between property developers and the city planning commission. Developers want to construct and sell buildings and aren't too fussy about the quality. The city planners have to make sure the developers consider issues such as neighborhood plans and whether the sewage-treatment plant has enough capacity.

As we move to cloud-computing environments in which the application, database management system, and other computing resources are hosted on a common collection of servers, we will see an increase in the standards and expectations of how they will fit into the enterprise.

WHAT ABOUT THE FORCES DRIVING US TO THE CLOUD?

The forces that are driving us to the cloud will have their way as they focus on a common and shared basis for computation and storage. First, the cost savings from cheaper electricity, networking, and server technology that are available in concentrated and expensive data centers will continue. While there are issues of managing trust, the economics are a powerful force. Second, enterprises will continue to find serendipitous value in the analysis of data. The more data available for analysis,

the more often valuable surprises will happen. Finally, both computational and data-storage resources will be increasingly shared across applications and enterprises. Existing applications will get some value from sharing and new cloud-aware applications will get even more.

These forces will encourage applications to work together atop new abstractions for sharing. When applications stick to their old models, they will get some advantages of the commonality of the cloud but not as many as those seen by new applications.

CONCLUSION

The constraints in an environment are what empower the services. The usage pattern allows for supporting infrastructure and concierge services.

Shared buildings become successful by constraining and standardizing their usage. Building designers know *how* a building will be used, even if they don't know *who* will be using it. Not everyone can accept the constraints, but for those who do, there are wonderful advantages and services.

The standardization of usage for computational work will empower the migration of work to the shared cloud. With these usage patterns, supporting services can dramatically lower the barriers to developing and deploying applications in the cloud. Lower-level standards are emerging with VMs. These support a broad range of applications with less flexibility for sharing. Higher-level PaaS solutions are nascent but offer many advantages.

We must define and constrain the usage models for important types of cloud applications. This will permit enhanced sharing of resources with important supporting services. The new PaaS offerings will bring tremendous value to the computing world.

REFERENCES

1. Amazon Web Services; http://aws.amazon.com/.
2. App Engine; http://code.google.com/appengine/.
3. Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R. H., Konwinski, A., Lee, G., Patterson, D. A., Rabkin, A., Stoica, I., Zaharia, M. 2009. Above the clouds: a Berkeley view of cloud computing; http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf.
4. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W. 2007. Dynamo: Amazon's highly available key-value store. ACM Symposium on Operating Systems Principles; http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf.
5. Force.com; http://www.force.com/.
6. Lakshman, A., Malik, P. 2009. Cassandra – a decentralized structured storage system. Large-scale Distributed Systems and Middleware; http://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf.
7. Peng, D., Dabek, F. 2010. Large-scale incremental processing using distributed transactions and notifications. Ninth Usenix Symposium on Operating Systems Design and Implementation; http://research.google.com/pubs/pub36726.html.
8. Riak; http://basho.com/products/riak-overview/

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

PAT HELLAND has been working in distributed systems, transaction processing, databases, and similar areas since 1978. For most of the 1980s, he was the chief architect of Tandem Computers' TMF (Transaction Monitoring Facility), which provided distributed transactions for the NonStop System. With the exception of a two-year stint at Amazon, Helland worked at Microsoft from 1994 to 2011 where he was the chief architect for Microsoft Transaction Server, SQL Service Broker, and a number of features within Cosmos, the distributed computation and storage system underlying Bing. Pat recently joined Salesforce.com and is leading a number of new efforts working with very large-scale data.

This paper was written before Pat joined Salesforce.com and, while there are many similarities, this is not intended to be a description of Salesforce's architecture.