SULTAN: A Composite Data Consistency Approach for SaaS Multi-Cloud Deployment

Islam Elgedawy
Computer Engineering Department,
Middle East Technical University, Northern Cyprus Campus
Guzelyurt, Mersin 10, Turkey,
Email: elgedawy@metu.edu.tr

Abstract—**Migrating business services to the clouds creates many high business risks such as "cloud vendor lock-in". One approach for preventing this risk is to deploy business services on different clouds as SaaS (i.e., Software as a Service) services. Unfortunately, such SaaS multi-cloud deployment approach faces many technical obstacles such as clouds heterogeneity and ensuring data consistency across different clouds. Cloud heterogeneity could be easily resolved using service adapters, but ensuring data consistency remains a major obstacle, as existing approaches offer a trade-off between correctness and performance. Hence, SaaS providers opt to choose one or more of these approaches at design time, then create their services based on the limitations of the chosen approaches. This approach limits the agility and evolution of business services, as it tightly couples them to the chosen data consistency approaches. To overcome such problem, this paper proposes SULTAN, a composite data consistency approach for SaaS multi-cloud deployment. It enables SaaS providers to dynamically define different data consistency requirements for the same SaaS service at run-time. SULTAN decouples the SaaS services from the cloud data stores, enabling services to adapt and migrate freely among clouds without any SaaS code modifications.**

I. Introduction

Moving business services to global clouds as SaaS services has many benefits (such as flexibility, scalability, reducing capital expenditure on capacity investment), however many businesses are still reluctant to do the move due to its high business risks such as "cloud-vendor lock-in". The cloud-vendor lock-in risk makes businesses vulnerable to price increase and/or changes to the cloud-vendor services. As a result, businesses do not have the freedom to change their cloud-vendors whenever they like as their services and data are locked-in due to clouds heterogeneity and data migration costs. One approach for preventing such risk is to deploy business services on different clouds. SaaS multi-cloud deployment provides better performance and lower costs compared to the usage of a single cloud, as it provides better availability, responsiveness, and resources utilization [1]. However, SaaS multi-cloud deployment approach faces many technical challenges such as cloud heterogeneity, ensuring data correctness, and security.

Cloud heterogeneity results from having different PaaS (i.e., Platform as a Service) and IaaS (i.e., Infrastructure as a Service) service offerings provided by cloud-vendors. Hence, SaaS services cannot freely move between different clouds due to the need for code customizations. In addition to the data migration costs. There are many initiated research efforts to overcome this problem, we summarize and classify them as follows:

- **The Cloud Unification Approaches:** Efforts in this approach (such as the works in [2] [3] [4]) are dedicated to standardize and unify the APIs of all PaaS and IaaS offerings for all cloud-vendors. Unfortunately, existing efforts for cloud standardization are still in their first stages and we are not expecting to have a mature solution soon. Moreover, cloud-vendors themselves are reluctant to unification approaches as they prefer to keep their competitive edge and diversity to attract customers.
- **The Cloud Orchestration Approaches:** Efforts in this approach (such as the works in [1] [5] [6]) are dedicated to create an orchestration layer between SaaS providers and cloud-vendors that can provide many services for SaaS providers such as portability, migration, and monitoring services. However, such approach is still not mature enough, furthermore it uses a centralized global control over WAN connections, which leads to bad performance as indicated in [7] [8], as global locks are used to ensure data correctness.
- **The Cloud Adaptation Approaches:** Efforts in this approach (such as the works in [9] [10] [11]) use service adapters to decouple SaaS services from the cloud vendors APIs. In this approach, adapters are used to provide a unified interface for SaaS services hiding any cloud PaaS and IaaS services heterogeneity. However, such adapters are created manually, and require adopting global data concurrency control approaches to ensure data correctness.

We believe the cloud adaptation approach is the most practical approach to overcome cloud heterogeneity, as first it does not require much efforts to build adapters for the cloud storage services due to their simple data access interfaces. Second, there exist many new cloud data store systems (such as spanner [12]) that are build for global clouds and adopt distributed global control approach to ensure data correctness. We summarize and classify the existing distributed global concurrency control approaches as follows:

- **The Non-Serializable Approaches:** Such approaches adopt weaker forms of data consistency such as eventual consistency, where replicas are not synchronized at all times. Such approach is widely adopted by NoSQL databases (such as Dynamo [13] and Cassandra [14]). However, when a conflict is detected, conflict resolution approaches should be carried out by the SaaS services. For example, undo transactions and/or compensating transactions could be performed. Or the last-write-wins strategy could be applied, which leads to loss of some data versions. Therefore, the SaaS service functionality must be able to tolerate such data loss/inconsistency (e.g., Facebook status notifications). These non-serializable approaches are fast, scalable but not necessarily correct.

- **The Optimistic Serializable Approaches:** Such approaches (such as the works in [15] [16]) require each datacenter to ensure its local data consistency using classical approaches, and to try to maintain global data consistency using variations of the Paxos protocol. Such approaches minimizes conflicts but cannot prevent them. Therefore, the SaaS service functionality must be able to tolerate such conflicts, as conflict resolution operations are needed. These optimistic approaches are fast, scalable but not necessarily correct.
- **The Pessimistic Serializable Approaches:** Such approaches (such as the Spanner [12] and Scatter [17] systems) guarantee data correctness, and allow no conflicts to occur. For example, the Spanner system [12] uses 2PC and 2PL to provide atomicity and isolation, running on top of a Paxos-replicated log, which provides a fault-tolerant synchronous replication process across datacenters. Such pessimistic approaches suffer from bad performance when the executed workload creates hot spots, as global locks are established over the slow WAN connections. SaaS services adopting these approaches guarantees their data correctness, as all the replicas are eagerly synchronized, however they may suffer from bad performance due to hot spots.
- **The Escrow-Based Serializable Approach:** This is a novel approach (i.e., NASEEB [18]) that guarantees data correctness as in the pessimistic approaches, and still ensures good performance as in the optimistic approaches. It uses the the notion of a datacenter escrow, in which every datacenter will have a specific non-overlapping capacity quota (i.e., the escrow) for each data object (i.e., escrow assigned at the attribute level) that it can consume independently without checking with other instances. For example, in a flight reservation application deployed in a global cloud with four datacenters, if a given flight object has a capacity of a given number of available seats attribute (e.g. 100). Instead of globally locking such flight object every time a booking is made within a given datacenter, we simply divide such attribute capacity (i.e. 100) among the four datacenters such that every datacenter will have a specific quota (e.g. 25 for each if we distribute equally). Hence, datacenters could process their incoming requests locally without the need to have global locks or consensus protocols, which drastically improves performance. Therefore, data correctness is guaranteed without the need for global locks, as any transaction committed locally is guaranteed to commit globally. NASEEB allows quota borrowing among the datacenters. However, in this approach, the total value of a given attribute has to be aggregated from the values of all replicas' quotas, which are replicated using a lazy replication approach. Hence, the aggregate attributes' values in this approach are timed-tamped, and not necessarily the freshest values. Hence, the functionality of SaaS services adopting such approach should be able to tolerate such data unfreshness issue, or could ask for data refreshing operation, which might take few seconds to occur.

As we can see every approach has its pros and cons, and requires different handling from the SaaS services. Currently SaaS multi-cloud deployment could be easily achieved by using service adapters to communicate with different storage services, and by choosing storage services that support one or more of the preferred data correctness approaches. Then design the SaaS service according to the capabilities of the chosen PaaS services. For example, the spanner system enables business service providers to choose from two options: a pessimistic option with bad performance during contention periods, or an optimistic option with a possibility of data inconsistency. We argue that such approach for realizing SaaS multi-cloud deployment is limiting and inflexible for the following reasons:

- First, it tightly couples the SaaS service design to the chosen data correctness approach. Hence, any changes for the consistency requirements and/or PaaS services' interfaces will require SaaS code modification. To improve business agility and responsiveness, we believe SaaS services should only focus on the business logic issues and should be decoupled from handling the logic of the adopted data consistency approaches.
- Second, it forces all the services' data items to have the same level of consistency. We find this approach inefficient, as not all data items have the same importance for the SaaS services' operations. For example, strong consistency requires high costs in terms on time and resources, and it is not wise to spend these costs over data items not crucial for the SaaS services' operations. Hence, we need to have different consistency requirements for different data items of the same SaaS service.

To overcome these limitations, we propose SULTAN, a composite data consistency approach for SaaS multi-cloud deployment. It enables SaaS providers to have different consistency requirements for different data items. This is done via a Data Consistency Plan (DCP) provided by the SaaS providers. SULTAN executes such DCPs using different PaaS storage services that support different consistency levels, and uses service adapters to overcome the PaaS services heterogeneity. SaaS services submit their data access requests directly to SULTAN, which coordinates all the data requests in a distributed global manner to ensure data correctness. Experimental results show that SULTAN handles consistency requirements' changes in a realistic practical timing, also improves services performance when compared with the existing pessimistic serializable data concurrency approaches.

The rest of the paper is organized as follows. Section II explains the proposed SULTAN architecture and assumptions, while Section III shows how to create cloud adapters. Section IV explains the concepts of DCP, while Section V gives an overview over some of the important SULTAN protocols. Section VI shows the performed validation experiments and discusses results. Finally, Section VII concludes the paper.

II. SULTAN Architecture and Assumptions

As maintaining strong data consistency is a costly process, we argue that it should be only used for data objects that their correctness is crucial for the SaaS services' correctness, while for less important data we could go for weaker consistency notions. Hence, SaaS services should have different consistency requirements for different data objects. To achieve such vision, we encapsulate the SULTAN approach as a PaaS service that currently supports the following levels of consistency:

- **Strong-Eager:** It implies that the global correctness of the data object is maintained such that any SaaS service instance will be accessing the up-to-date correct value of the object. This requires expensive eager replication techniques between datacenters. This consistency level is most suitable for non-aggregatable attributes with a single atomic quantity and low concurrent update frequency such as `Name`, `SSN` and `Credit-Card-Number` attributes. Strong-Eager objects will be stored in a serializable data store (known as the SQL store), which is accessed via a corresponding PaaS storage service.
- **Strong-Escrow:** It implies that the escrow-based approach is adopted, where the global correctness of the data objects' quotas is maintained such that any SaaS service instance will be accessing the correct aggregate value of the object with a data freshness threshold. This requires cheap lazy replication techniques between datacenters. This consistency level is most suitable for aggregatable attributes with aggregatable quantities and high concurrent update frequency such as `Quantity-On-Hand`, `Number-Of-Available-Seats`, `Total-Cash-Received` attributes. Strong-Escrow objects will be stored in eventual data store (known as the NoSQL store), which is accessed via a corresponding PaaS storage service.
- **Eventual:** It implies that object correctness is locally maintained (i.e. within a datacenter) but not globally (i.e. between all datacenters). However, if there are no global conflicts between datacenters, and no more new updates are made to the object, eventually all database accesses will return the same last updated value. This requires lazy replication techniques between datacenters. This consistency level is most suitable for attributes that have low concurrent update frequency and their correctness is not crucial for service operations such as `Address`, `Telephone`, `fax-number` attributes. Eventual objects will also be stored in the NoSQL store.
- **Session** It implies that the SaaS services read their own writes only. Hence, those data will be lost after the session terminates. Session objects will be stored in a special cache memory accessed via the SULTAN service.

SaaS service providers should select a consistency level for each of their services' data objects, by defining a Data Consistency Plan (DCP) for each SaaS service, then submit the defined DCPs to SULTAN. To execute such DCPs, SULTAN requires a group of PaaS storage services that can support one or more of the above consistency levels. Every datacenter could use a different group of PaaS Services, as shown in Figure 1. The figure depicts the architecture of the SULTAN approach, and shows an example of three datacenters adopting the SULTAN approach. Every datacenter has its own PaaS storage services, and the SULTAN service will be deployed on top of them. All datacenters use the NASEEB service [18] to handle the strong-escrow consistency. For strong-eager and eventual consistency, datacenters 1 and 2 use the Spanner service, while datacenter 3 uses an Oracle service. SULTAN is the coordinator between all the PaaS storage services, and communicates with them via a SULTAN adapter.

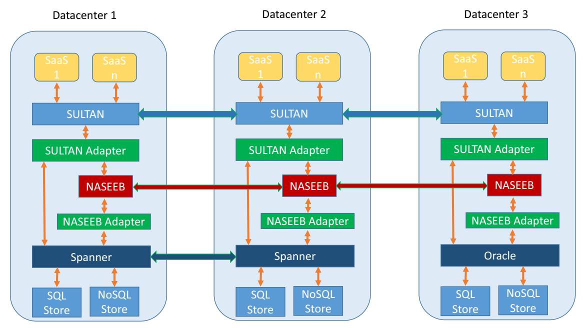


Fig. 1. SULTAN Architecture

We require the following design constraints and assumptions to realize the SULTAN approach:

- Each SaaS service instance must be connected only to one SULTAN service instance at a given time, but a given SULTAN service instance could be connected to multiple SaaS service instances (i.e., SULTAN supports multi-tenancy).
- Each datacenter should have at least one SULTAN service instance. Every SULTAN service instance keeps a list of all other SULTAN service instances in all datacenters; known as the SULTAN-Peer list. Instances in the list should send periodic heartbeat signals to each other in different rates to detect failing instances.
- When different strong-eager consistency PaaS services

are used in different datacenters (as in in Figure 1, where Spanner and Oracle services are used), all locking operations are done via SULTAN, otherwise write access could be directly delegated to the corresponding PaaS service. For example, when all datacenter uses the Spanner service, SULTAN can delegate to it all the write operations without getting involved.

- Objects with weak consistency requirements should be kept in a separate NoSQL storage, different from the SQL storage of strong objects to avoid any indirect effects, as shown in Figure 1.
- SULTAN uses adapters to overcome the cloud interoperability problem. Such adapters are used to decouple the SaaS service instances from the cloud vendors PaaS services. SULTAN service instances provide a unified data access interface for all SaaS instances in all datacenters, while a special adapter is used for every cloud. The adapter contains the APIs required for data accessing and locking operations.
- Each SULTAN service instance has it is own view of the data objects, we call this view as the SULTAN-Instance-View (SIV), and it is defined as a set of objects projections, where an object projection (denoted as $\prod_{(i,j)}$) is defined as the tuple $< SULTAN-Instance_j$, O_i, A_j, $ConsistencyLevel$, $OldValue$, $LastValue$, $TimeStamp >$, which represents reference to the SULTAN service instance accessing the data object, reference to the data object O_i and its attribute A_j, the object's attribute consistency level, old and last values on that service instance, and finally the time stamp for the last value provided, which is needed to for the replication process.
- SULTAN is purely distributed and does not depend on a centralized common master to coordinate updates and to perform consistency checks. All communication between service instances are done in a Peer-to-Peer (P2P) fashion adopting asynchronous communication messages.

SULTAN overhead is limited, as at design-time, DCP definition and adapters creation are needed, which is done once before deployment. While, at run-time, the time consumed by the created extra layer(s) between the SaaS service and the cloud storage is small (i.e., (0.3ms-1ms) [20]), as all communications occur within the datacenter.

A. SULTAN Data Management

Data management in SULTAN is quite simple. SULTAN exposes simple `read(obj o)` and `write(obj o, value v)` interfaces for the SaaS services. Hence, SaaS providers write their code in terms of these two operations. Any other administration interfaces such as `SubmitDCP(DCP p, Service s)` are done independently from the SaaS service code. When a SULTAN service instance receives a read or write request, it searches the corresponding service DCP and retrieves the required consistency level, then invokes the corresponding PaaS storage service. Once it received the results and acknowledgements, it passes the results to the SaaS service. In case of strong-escrow consistency read, SULTAN needs to know the freshness threshold and the aggregation scope to retrieve the correct values. Write operations are straight forward, as they will be mapped to a simple call for the PaaS service write operations. Only when the strong-eager PaaS services are different, SULTAN takes charge of the distributed concurrency control process using a lightweight version of the Scatter [17] approach. However, any pessimistic distributed concurrency approach could be used.

III. Cloud Adapters

SaaS, PaaS, and IaaS services communicate via exchanging messages. A message indicates the operation to be performed by the service receiving the message, while a sequence of messages constitutes what is known by a service conversation. As service providers could use different concepts, and semantics to generate their service conversations, a conversation incompatibility problem might arise due to signature and protocol incompatibilities [9]. One way to overcome this problem is by using conversation adapters. A conversation adapter is the intermediate component between the interacting services that facilitates service conversations by converting the exchanged messages into messages understandable by the interacting services, as shown in Figure 1, in which SULTAN adapter is used to interact with NASEEB and Spanner services, also the NASEEB service used another adapter to interact with the Spanner service.

SULTAN service provide a unified data access interface for all SaaS instances in all datacenters, hence SaaS services are decoupled from the PaaS services. However, to decouple the SULTAN service from the PaaS services as well, we use a special SULTAN-adapter for every cloud. Such adapter will map the data access requests passed to the SULTAN service to the cloud vendors' PaaS storage services, and returns the obtained results. SULTAN mainly requires read, write, locking and replication interfaces to be supported by the adopted cloud vendor storage services. Such basic interfaces are supported by all existing cloud vendors storage services. To create SULTAN adapters, SaaS providers should follow the following steps:

- First, they should support the SULTAN generic APIs for read and write operations in their services code such that data is accessed only via the SULTAN service instances.
- Second, they should determine the mappings between the SULTAN generic APIs and the actual APIs of the PaaS services offered by the cloud-vendor(s), and define the corresponding conversion functions.
- Third, they should create a class for the conversation adapter with methods corresponding to the messages to be mapped (i.e., a message is an API invocation), where a method body has the corresponding conversion function.
- Finally, the created adapter class should be deployed as a platform service on the datacenters of the chosen cloud-vendors.

The structure of the SULTAN-Adapter constituted from the read and write operations of every supported consistency level, in addition to the control operations. For example, for the strong-escrow consistency level, SULTAN requires the following read interface `EscrowRead(DataObject X, String AggregateScope, String FreshnessThreshold)` to be in all the adapters. The code of conversion function of this operation inside the adapter is just invoking the `EscrowRead` operation from the corresponding NASEEB Service. Another example, is the strong-eager consistency level, the corresponding read operation interface will be `EagerRead(DataObject X)`, which will call the read operation from the corresponding Spanner service. More details about adapter creation and resolving semantic incompatibilities could be found in [9]. We argue that creating the SULTAN adapters manually is practical, as it is done once for every SULTAN cloud deployment, and the efforts needed are small, as they are just simple calls of stable PaaS services' interfaces, which do not change frequently. However, in case of a PaaS service interface is changed, only the adapter will be updated.

IV. Data Consistency Plan (DCP)

SaaS providers need to capture their consistency requirements for their services in a form of DCPs, and submit these DCPs to SULATN via `SubmitDCP` operation. A DCP contains the required consistency level for every data object as well as its corresponding semantics. A DCP could contain the consistency requirements at the object level (i.e., this implies all the attributes of the object should have the same specified consistency level), or at the attribute level (i.e., this implies only the concerned attribute should have the specified consistency level, hence an object could have a mix of consistency levels for its attributes). In what follows, we explain the required semantics of the supported consistency levels, and provide the formal definition of a DCP supported by an example.

A. Consistency Level Semantics

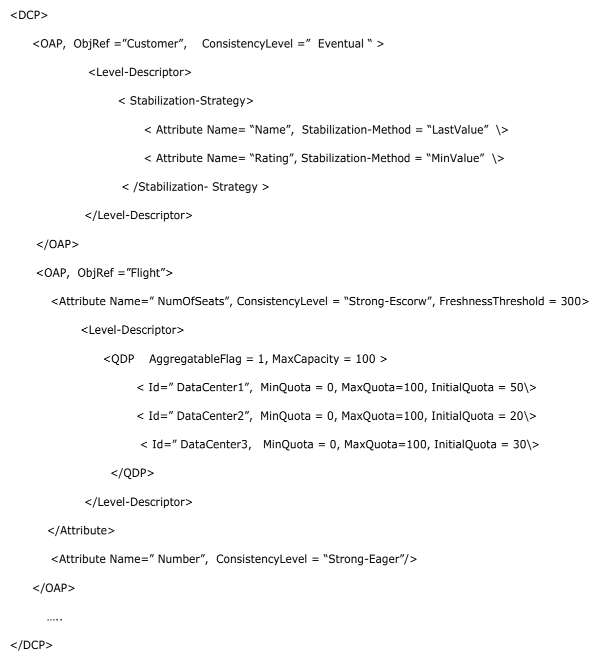
SULTAN supports four levels for consistency: Strong-Eager, Strong-Escrow, Eventual, and Session. Every level has its own corresponding semantics that need to be defined as follows.

1) Strong-Eager Consistency Level Semantics: For this consistency level, SaaS providers just need to specify the object/attribute reference.

2) Strong-Escrow Consistency Level Semantics: For this consistency level, SaaS providers need to specify the attribute reference, and its Quota Distribution Plan (QDP) among the datacenters as well as its computation semantics. Such semantics are specified by the NASEEB service [18], we summarize the QDP semantics for a given attribute as follows:

- A_j: is the attribute unique reference.
- $Aggregatable$: is a flag equals to 1 if the attribute value is aggregatable and equals to 0 otherwise.
- $Incremental$: is a flag equals to 1 if the attribute maximum capacity can be increased during run time (as in account balance attribute), and equals to 0 if the maximum capacity can not be increased (for example, max number of available seats cannot exceed plane capacity).
- $MaxCapacity$: is the attribute maximum capacity which will be distributed among datacenters. Attributes with no capacity semantics, their maximum capacity will equal to 1 (such as the `Name` attribute).
- $FeshnessThreshold$: is the maximum amount of time (in seconds) defines how fresh should be the replicated data. That any replicated data older than the specified threshold has to be re-fetched from the corresponding SULTAN service instance.
- $QuotaDistribution$: is defined as a set of allocations tuples $\langle DatacenterRef$, $MinQuota$, $MaxQuota$, $InitQuota\rangle$, where the allocation tuple elements are defined as follows:
  - $DatacenterRef$: is a reference for involved datacenter.
  - $MinQuota$: is the lower limit of the attribute capacity that can be allocated for the attribute.
  - $MaxQuota$: is the upper limit of the attribute capacity that can be allocated for the attribute.
  - $InitQuota$: is the initial allocated quota for the attribute (which should be within the specified range).

SULTAN passes this information to the NASEEB service, which will ensure that the attributes' values will always be in the defined quota range; adopting a quota-borrowing protocol for high demands. An example for a DCP is shown in Figure 2.



```
<DCP>
  <OAP, ObjRef ="Customer",  ConsistencyLevel =" Eventual " >
        <Level-Descriptor>
            < Stabilization-Strategy>
                < Attribute Name= "Name", Stabilization-Method = "LastValue" \>
                < Attribute Name= "Rating", Stabilization-Method = "MinValue" \>
            < /Stabilization- Strategy >
        </Level-Descriptor>
  </OAP>
  <OAP, ObjRef ="Flight">
    <Attribute Name=" NumOfSeats", ConsistencyLevel = "Strong-Escorw", FreshnessThreshold = 300>
        <Level-Descriptor>
           <QDP  AggregatableFlag = 1, MaxCapacity = 100 >
               < Id=" DataCenter1", MinQuota = 0, MaxQuota=100, InitialQuota = 50\>
               < Id=" DataCenter2", MinQuota = 0, MaxQuota=100, InitialQuota = 20\>
               < Id=" DataCenter3,  MinQuota = 0, MaxQuota=100, InitialQuota = 30\>
           </QDP>
        </Level-Descriptor>
    </Attribute>
    <Attribute Name=" Number", ConsistencyLevel = "Strong-Eager"/>
  </OAP>
    .....
</DCP>
```

Fig. 2. An example for a DCP XML Representation

3) Eventual Consistency Level Semantics: For this consistency level, SaaS providers need to specify the object/attribute reference, and its stabilization strategy. A stabilization strategy is the object/attribute value computation method that needs to be adopted by SULTAN to agree on a single value when conflicts occur. SULTAN supports the following stabilization strategies (i.e. `RollBack`, `LastValue`, `MaxValue`, `MinValue`, `AvgValue`, `SumValue`, `MajorityValue`, or `Customized`). `RollBack` option implies undoing all conflicting transactions and performs the corresponding compensating transactions if any. This option is the highest in cost and could degrade the performance. Other options such as `LastValue`, `MaxValue`, `MinValue`, `AvgValue`, `SumValue`, `MajorityValue`, apply basic probabilistic basic functions over the conflicting data to create an approximate new common value. This probabilistic functions should be used only if the application business logic tolerates such approximation. Otherwise, the developer should provide a `Customized` function for resolving the conflict. Hence, SULTAN requires the SaaS providers to specify the required stabilization strategy for each object as part of the application DCP in order to execute such strategy when needed.

4) Session Consistency Level Semantics: For this consistency level, SaaS providers need to specify the object/attribute reference, and its expected life time. The object should be killed after its life time expires.

B. DCP Formal Definition

We formally define DCP as a set of Object Access Patterns (OAP), that $DCP = \{OAP_i\}$ where an object access pattern OAP_i is defined as a tuple $OAP_i = \langle ObjRef, ConsistencyLevel, LevelDescriptor\rangle$, where $ObjRef$ is the data object reference, where $ConsistencyLevel$ is chosen as $(Strong{-}Eager \mid Strong{-}Escrow \mid Eventual \mid Session)$, and $LevelDescriptor$ is the description of the adopted policies concerning the consistency level and could be described as $(\ \emptyset \mid QDP \mid StabilizationStrategy \mid LifeTime)$. Such that for data objects with strong-escrow consistency we define the required quota distribution for the object attributes, and for data objects with eventual consistency we define the required object stabilization strategy that should be adopted in case of conflicts, and for data objects with session consistency we define the life time of the objects in minutes. Figure 2 shows an XML representation for a DCP for flight reservation application. The figure shows that we have two objects, the first object is `Customer` with eventual consistency and the second object is `Flight` with mix consistency. `Customer` object has attribute `Name` that should be stabilized via the last value stabilization strategy , and the attribute `Rating` that should be stabilized via the min value stabilization strategy. The second object is `Flight`, it has two attributes: `Number` with strong-eager consistency and `NumOfSeats` with strong-escrow consistency. The `NumOfSeats` attribute with capacity 100 that is divided among three datacenters with quotas 50, 20, and 30 respectively.

One more issue is that DCP file could get large if the number of data objects are big and becomes a cumbersome to define. To overcome such problem, we assume a default consistency level for objects, and only objects with other consistency requirements are defined. For example, a common consistency level in cloud-based applications is eventual with `LastValue` stabilization method, could be used as default. However, as business requirements could change by adding new objects, removing existing objects, or changing consistency levels of existing objects, SULTAN allows DCPs to be dynamic. Hence, SaaS providers submits the new DCP plans, and SULTAN automatically handle all the changes, as shown in the following section.

V. SULTAN Protocols

SULTAN is a purely distributed approach, hence it follows many P2P distributed protocols to accomplish its functionality such as join/leave protocols, object stabilization and recovery protocols. Due to space limitation, we will not discuss all SULTAN protocols but we will focus on the important issues that ensure the approach liveness and correctness. The coming protocols are leader-based protocols, that a given task (such as consistency change) must have a leader node to orchestrate the protocol execution. However, if more than one node needs to accomplish the same task over the same object, a leader election process must be done to choose a winner. We elect the leader as the node with the least load.

A. Object Stabilization Protocol

When an eventual object has conflicting values, object stabilization is required. The object stabilization protocol works as follows:

- The leader SULTAN instance initiating the stabilization requests broadcasts a stabilization request for all SULTAN instances, and waits for their response.
- Each SULTAN instance replies back with its current value of the object, and blocks access to the object until it is stabilized.
- The leader computes the new object value using the stabilization method defined in the DCP, and broadcast it to all other instances.
- The leader waits for the acknowledgement of all the other instances. If all instances replied, it considers the request is fulfilled. In case of missing or slow acknowledgment, the leader tries back after certain timeout window, if an instance still not replying, the leader consider it as a failed node and starts the failure notification procedure,as shown later.

B. DCP Change Management Protocol

If an object is added/removed from a service set of objects. SULTAN simply adds/removes the object to its catalog. However, when a consistency requirement for an existing object is changed, it could be upgraded to a stronger level, or could be downgraded to a weaker level. Hence, SULTAN has to perform some operations to ensure the correctness of the data.

The SULTAN service instance that receives the change request, becomes the leader for this request, and it will coordinate with the other SULTAN service instances to perform the change. In what follows, we show how SULTAN performs consistency requirements' changes:

- In case of consistency level upgrade request from Session to Eventual, the SULTAN service instance leader does the following steps:
  - It updates the corresponding DCP catalog entry with the new level.
  - It stores the object value written in its cache into the NoSQL store.
  - It notifies other SULTAN service instances in its peer-list to store the object and its value into their corresponding PaaS storage services.
  - It waits for the acknowledgments of other SULTAN instances. If all instances replied, it considers the request is fulfilled. In case of missing or slow acknowledgment, the leader tries back after certain timeout window.
- In case of consistency level upgrade request from Session/Eventual to Strong (Eager or Escrow), the SULTAN service instance leader does the following steps:
  - It updates the corresponding DCP catalog entry with the new level.
  - It starts the object stabilization protocol in order to stabilize the object values in all datacenters.
  - It computes the new object value according to stabilization method. In case of strong-escrow, it computes the remaining object capacity (i.e. capacity given in the new DCP - the new object value) and distributes it among the existing instances. If the attribute new quota distribution is not given in the new DCP. It keep the remaining quota for itself and make other instances quota equals to zero.
  - It removes the object from the NoSQL store, then stores it to the SQL store.
  - It sends a object change command for each SULTAN instance with the new object value /allocated quota. Once a SULTAN instance receives a change command, it updates the object value and its DCP and unlock the object to make it ready for access, then sends to the leader a change acknowledgement.
  - It waits for SULTAN instances acknowledgement, then sends back a stabilization confirmation to all other instances that indicates stabilization has successfully performed.
- In case of consistency level downgrade request from Strong to Eventual, the SULTAN service instance leader does the following steps:
  - It updates the corresponding DCP catalog entry with the new level.
  - In case of strong-eager consistency, it removes the object from the SQL store, then stores it to the NoSQL store, and issues an object change command for each SULTAN instance.
  - In case of strong-escrow consistency, it computes the aggregate object value, then sends a object change command for each SULTAN instance with the new object aggregate value.
  - Once a SULTAN instance receives a change command, it updates the object value and its DCP and unlock the object to make it ready for access, then sends to the leader a change acknowledgement.
  - It waits for instances acknowledgement, then sends back a stabilization confirmation to all other instances that indicates stabilization has successfully performed.
- In case of consistency level downgrade request Strong/ Eventual to session, the SULTAN instance leader updates the corresponding DCP entry, and then creates an entry in its cache for the object and stop storing object updates into the data stores as all updates has to in the cache only. In both cases, SULTAN leader notifies other SULTAN instances with the change and waits for their acknowledgement, as shown in the previous cases.

In case a SULTAN instance is not replying for any of the previous steps, the leader considers it as a failed node and starts the failure notification procedure,as shown in the following section.

C. SULTAN Failure Notification Protocol

Every SULTAN service instance regularly uses the heartbeat approach to check on other SULTAN instances located within its datacenter (according to its own schedule). Regular SULTAN instances do not send such heartbeats to instance outside its datacenter, however only gateway instances do the heartbeat check with other gateway instances in other datacenters. Once an instance failure is detected, the failure discovering instance becomes the leader of the failure notification protocol, and contacts other instances within its datacenter to inform them about the discovered instance failure. The failure notification protocol works as follows:

- **Step 1:** Once a failed instance is detected, the leader sends a failure notification message to all instances in its datacenter, and waits for their acknowledgments.
- **Step 2:** Once the leader is decided, each follower instance receiving the failure notification removes the failing instance from its active instance list, then responds with an acknowledgment piggy backed with its version of the last values and their time-stamps of the failing instance.
- **Step 3:** The leader collects all acknowledgments, and accepts the values with the most recent time stamps as correct ones, then checks if the data stores of the failing instance is connected to other SULTAN instances. If yes, we have no problems, as the connected instance will update the corresponding NoSQl store when it is notified, otherwise the leader needs to synchronize its NoSQL store with the NoSQL store of the failing node, then informs other SULTAN instances with the changes to update their NoSQL stores.

- **Step 4:** Each instance receiving the change confirmation message should update its SIV entries regarding the failing instance, also updates its quota with the new amount (if any), then acknowledges the leader with the update confirmation.

- **Step 5:** The leader waits for all confirmations, if any instance did not respond to the leader after a number of trials, the leader claims its share of extra quota, then issues a failure notification to all other instance regarding the non-responding instance.

D. SULTAN Recovery Actions

Having the failure notification protocol is not enough, as failing instance could be a leader involved in other SULTAN protocols. Also failure could arise during SaaS service interaction with its SULTAN service instance. In what follows, we will discuss the SULTAN recovery approach for each case.

- **In Case of SaaS service Failure:** The corresponding SULTAN instance rollbacks any uncommitted transactions using classical DB approaches, as all the updates occurred locally.
- **In Case of SULTAN Instance Failure:** Other SULTAN instances will follow the failure notification protocol, while the corresponding SaaS service will wait until it times out, then it chooses another SULTAN service instance to redirect its requests to. This is done by providing the SaaS services with a configuration file containing a list of SULTAN service instances within its datacenter as well as within the global cloud.
- **In Case of SUTLAN Gateway Failure:** Failure of gateways is not a crucial problem, as no data is lost, as a gateway only propagates SIV projections changes to other gateway instances. Hence, when a gateway is recovered it can check with other gateways the last confirmed SIV projection updates, and continue from where it stopped.
- **In Case of Stabilization Protocol Leader Failure:** If leader failed before sending change command messages, we will have no problems as no DCPs have changed, however if it failed before receiving all change acknowledgement a problem arises. As some SULTAN instances could have successfully received the stabilization command and updated their DCPs while other instances could not do such updates. To avoid this problem, each follower node that did not receive the stabilization confirmation message from the leader within a given time, it assumes the leader has failed, and rollbacks the last changes required by such leader, then issue a failure notification about the leader node. If a follower failed during the stabilization protocol, the leader issues failure notification about it and applies the protocol.

VI. SULTAN Evaluation

To evaluate the proposed SULTAN approach, we used the Cloudsim [19] tool that enables us to simulate complex cloud environments. As SULTAN needs PaaS services for strong-eager, strong-escrow, and eventual consistencies. We extended the Cloudsim datacenter class to include the logic of the following approaches:

- **2PC:** The classical 2PC approach tries to lock the objects all over the cloud before performing any write operation. If the lock cannot be established, it just rejects the request, and aborts the transaction. Read operation is fulfilled from the local instance, and no locks are required. This approach realizes the strong-eager consistency.
- **NASEEB:** The NASEEB approach does not perform any global locks. Write operations are performed according to the quota constraints. Read operation is fulfilled from the local instance, and no locks are required (more details are in [18]). This approach realizes the strong-escrow consistency.
- **Eventual:** The eventual consistency approach does not perform any global locks, however it directly applies the update to the local instance. During the lazy replication process if conflicts are detected, object stabilization process is performed. This approach realize the eventual consistency.

We performed different simulation experiments for different scenarios that show how SULTAN accomplishes composite consistency and consistency change requests.

A. Experiments Setup

To perform our experiments, first we have to configure the Cloudsim tool, then generate our dataset and workloads. To simulate realistic cloud environment, the inter-datacenter latency must be much bigger than the intra-datacenter latency, as global clouds have a nonhomogeneous timing model resulting from the big difference between WAN and LAN latencies. The work in [20] shows that intra datacenter latency (i.e., LAN connections) is within the range (0.3ms-1ms), while the work in [21] shows the inter datacenter latency (i.e., WAN connections) is within the range (83ms-445ms). Hence, performance mainly affected by the inter-datacenter latency. We summarize the experiments setup steps as follows:

- **Inter-Datacenter Latency:** The work in [21] shows that the latency between 8 datacenters on Amazon EC2 ranges between 83ms and 445ms. As we can see the variance between the inter-datacenter latency is quite big, hence we will do our experiments for the worst case scenario (i.e., 500ms).
- **Cloud Topology:** We created a global cloud with 5 identical datacenters, where every datacenter has a corresponding user base with latency 50ms.
- **Datacenter Configuration:** We used the default settings of Cloudsim tool. That each datacenter has 5 virtual machines. Each virtual machine contains 512 MB and 1KB bandwidth. Each datacenter is build using two 4-core processors identical servers with 10000 MIPS, 200GB RAM, 10 TB storage, and IMB bandwidth. It is important to note that 5 virtual machines are enough for the generated

workload to have intra datacenter latency within the range (0.3ms-1ms), however for higher workloads, more virtual machines should be used to keep the latency within the realistic range (0.3ms-1ms) as indicated in [20].

- **Dataset Generation:** For simplicity, we assumed that we have 10,000 different objects, where every object contains one aggregatable attribute initialized with zero and has a maximum of 100 global capacity. Such capacity will be divided equally among the datacenters when NASEEB is adopted.
- **User-Base Workload Generation:** We will perform the experiments over a large user base that generates 1000 request per hour. Every generated request will contain one read and one write operations, and the accessed object is randomly chosen from the dataset.
- **Lazy Replication Configuration:** For eventual and NASEEB approaches, we perform an independent background replication operation that replicates the objects values every 20 minutes.

For our experiments, we used SULTAN in different scenarios to test consistency requirements change and composition. We run the simulation for a period of 24 hours and computed the average response time, and computed the throughput. Experiments are done as follows:

- **Scenario1: 100% Strong-Eager Objects.** We created a DCP that requests all objects to have strong-eager consistency, then submitted the DCP and the generated workload to SULTAN. SULTAN expected to store the objects in the SQL store, then passes all the requests to the PaaS service responsible for strong-eager consistency.
- **Scenario 2: 100% Strong-Escrow Objects.** We created a DCP that requests all objects to have strong-escrow consistency, then submitted the DCP and the generated workload to SULTAN. SULTAN expected to store the objects in the NoSQL store, then allocates equal quotas for the datacenters, then passes all the requests to the NASEEB service.
- **Scenario 3: 100% Eventual Objects.** We created a DCP that requests all objects to have eventual consistency, then submitted the DCP and the generated workload to SULTAN. SULTAN expected to store the objects in the NoSQL store, then passes all the requests to the PaaS service responsible for eventual consistency.
- **Scenario 4: 10% Strong-Eager Objects.** We created a DCP that requests 10% of objects to have strong-eager consistency, and the rest have eventual consistency, then we submitted the DCP and the generated workload to SULTAN. SULTAN expected to store the strong-eager objects in the SQl store, and the eventual objects in the NoSQL store, then passes all the requests to the corresponding PaaS services.
- **Scenario 5: 50% Strong-Eager Objects.** Same as scenario 4 but we increased the percentage of strong-eager objects to 50%.

In every scenario, we compute the average response time, and compute the throughput. Results are depicted in Figure 3.

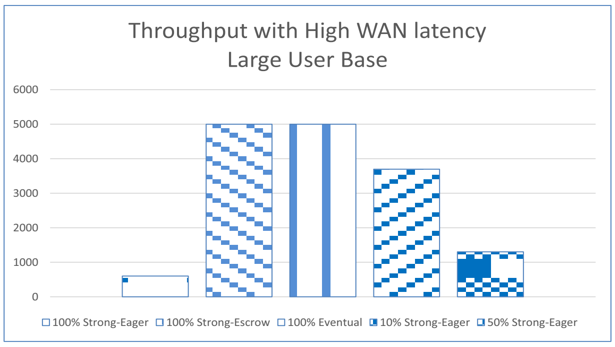


Fig. 3. Consistency Requirements Scenarios Comparison

Figure 3 shows SULTAN managed to fulfill different consistency requirements without any SaaS code modification. The obtained results are logical, as the strong-eager approach expected to be the worst, as it establishes locks over the slow WAN connections, while the eventual and escrow-based approaches are expected to be the best performing approaches, as they use no global locks. Also we can see 100% eventual and strong-escrow scenarios managed to get the max throughput (i.e., 5000, where 1000 comes from every datacenter). We argued that strong-eager consistency should be used only for the objects crucial for the service correctness, Figure 3 confirms our argument, as when having only 10% of the objects as strong-eager, it shows big performance improvement (i.e., about 500%, jumping from 600 to 3700) when compared with with the 100% of the objects are strong-eager. This means adopting a composite consistency approach improves SaaS service performance. Also when comparing this scenario to the scenario of 100% of the objects are eventual, we can see limited performance loss (i.e., 26%, dropping from 5000 to 3700), which we believe is a very acceptable price for ensuring SaaS services correctness. From results, we can see that the strong-escrow consistency is the best option for SaaS services, if they can tolerate data unfreshness, as it provides the best performance with data correctness assurance. Otherwise, a composite data consistency approach should be adopted in SaaS multi-cloud deployments to ensure the correctness of the crucial data.

For consistency requirements change evaluation, we conducted another set of experiments to measure the time taken (in msec) by SULTAN to change consistency requirements from one level to another for all objects. Results are shown in Figure 4, which shows the smallest change cost occurs when downgrading from the strong-eager consistency to eventual consistency, as it only involves objects migration from SQL store to NoSQL store. While, downgrading from strong-eager consistency to strong-escrow consistency costs more, as in addition to object migration step, a quota distribution step is required. It is important to notice that downgrading from strong-escrow consistency to eventual consistency costs more than downgrading from strong-eager consistency to eventual. This is because an object aggregation step is required, which involves access to the WAN connections. Figure 4 shows that the most expensive change is from strong-escrow to strong-eager, as this involves objects' values aggregation and propagation as well as objects migration steps.

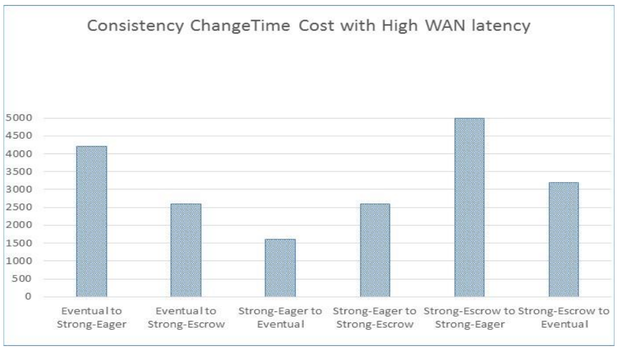


Fig. 4. Consistency Change Time Cost

The obtained latency of 5000ms is realistic, as other systems such as Albatross [22] recorded similar values. Hence, we believe SULTAN accomplishes consistency requirements' changes in a realistic practical time.

B. Threats To Validity

Simulation is performed using simple approaches' implementation, any full-fledged implementation may lead to different results. Hence, obtained metrics' values should not be used as an absolute basis for approaches judgement. However, the trend of the obtained values (i.e. values increasing, values decreasing or values unchanged) should be used instead.

VII. Conclusion

This paper proposed SULTAN, a composite data consistency approach for SaaS multi-cloud deployment. It enables SaaS providers to dynamically define different data consistency requirements for the same SaaS service at run-time; by defining different Data Consistency Plans (DCPs), which are executed by coordinating data requests among different cloud storage services. Service adapters are used to overcome the cloud heterogeneity problem, and protocols for ensuring SULTAN liveness and correctness are discussed. Experimental results show that SULTAN handles consistency requirements' changes in a realistic practical timing, and improves services performance when compared with the existing pessimistic serializable data concurrency approaches.

References

[1] J. Tordsson, R. S. Montero, R. Moreno-Vozmediano, and I. M. Llorente, "Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers," *Future Generation Computer Systems*, vol. 28, no. 2, pp. 358 – 367, 2012.

[2] L. Youseff, M. Butrico, and D. Da Silva, "Toward a Unified Ontology of Cloud Computing," in *Proceedings of the 2008 Grid Computing Environments Workshop*, 2008, pp. 1–10.

[3] N. Loutas, V. Peristeras, T. Bouras, E. Kamateri, D. Zeginis, and K. Tarabanis, "Towards a reference architecture for semantically interoperable clouds," in *Proccedings of the IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, Nov 2010, pp. 143–150.

[4] T. Kurze, M. Klems, D. Bermbach, A. Lenk, S. Tai, and M. Kunze, "Cloud federation," in *CLOUD COMPUTING 2011, The Second International Conference on Cloud Computing, GRIDs, and Virtualization*, 2011, pp. 32–38.

[5] B. P. Rimal and M. A. El-Refaey, "A framework of scientific workflow management systems for multi-tenant cloud orchestration environment," in *Enabling technologies: Infrastructures for collaborative enterprises (wetice), 2010 19th ieee international workshop on*. IEEE, 2010, pp. 88–93.

[6] C. Liu, Y. Mao, J. Van der Merwe, and M. Fernandez, "Cloud resource orchestration: A data-centric approach," in *Proceedings of the biennial Conference on Innovative Data Systems Research (CIDR)*, 2011, pp. 1–8.

[7] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Customizable fault tolerance for wide-area replication," in *In Proceedings of the 26th IEEE Symposium on Reliable Distributed Systems*, 2007.

[8] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: building efficient replicated state machines for wans," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08, 2008, pp. 369–384.

[9] I. Elgedawy, "On-demand conversation customization for services in large smart environments," *IBM Journal of Research and Development, Special issue on Smart Cities*, vol. 55, no. 1/2, 2011.

[10] A. Sampaio and N. Mendonça, "Uni4cloud: an approach based on open standards for deployment and management of multi-cloud applications," in *Proceedings of the 2nd International Workshop on Software Engineering for Cloud Computing*. ACM, 2011, pp. 15–21.

[11] C. Zou, H. Deng, and Q. Qiu, "Design and implementation of hybrid cloud computing architecture based on cloud bus," in *Mobile Ad-hoc and Sensor Networks (MSN), 2013 IEEE Ninth International Conference on*. IEEE, 2013, pp. 289–293.

[12] J. C. e. a. Corbett, "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 8:1–8:22, Aug. 2013.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007.

[14] A. Lakshman and P. Malik, "Cassandra: structured storage system on a p2p network," in *Proceedings of the 28th ACM symposium on Principles of distributed computing*, 2009.

[15] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi, "Low-latency multi-datacenter databases using replicated commit," *Proc. VLDB Endow.*, vol. 6, no. 9, pp. 661–672, Jul. 2013.

[16] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete, "Mdcc: multi-data center consistency," in *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013.

[17] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson, "Scalable consistency in scatter," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 15–28.

[18] I. Elgedawy, "NASEEB: An escrow-based approach for ensuring data correctness over global clouds," *Arabian Journal for Science and Engineering*, vol. 39, no. 12, pp. 8743–8764, 2014.

[19] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Softw. Pract. Exper.*, vol. 41, no. 1, pp. 23–50, 2011.

[20] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: scalable causal consistency for wide-area storage with cops," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011.

[21] Z. Ye, S. Li, and J. Zhou, "A two-layer geo-cloud based dynamic replica creation strategy," *Applied Mathematics and Information Sciences*, vol. 8, no. 1, pp. 431–440, Jan. 2014.

[22] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi, "Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration," *Proc. VLDB Endow.*, vol. 4, no. 8, May 2011.