Azure presentation

Contents

1 Introduction

With the rise of advanced internet technologies, an alternative mode of software consumption has been proposed. This alternative mode, referred as Software as a Service (SaaS), provides the use of applications that are not hosted on the users hardware but on a distant server. The user is accessing to this distant server through the internet, and most of the resources required by the application are provided on the server side. This server/client approach is a kind of renew with the old server/terminals design. This SaaS approach is rising a lot of interests and concerns. To name a few, SaaS lowers maintenance labor costs through multi-tenancy, it requires a smaller customer commitment since the customer is renting the usage of a software instead of buying it, it partially outsources IT for companies for which it is not their core business, etc. The SaaS approach has also its detractors. For example, Richard Stallman has incarnated the voice of those for whom SaaS rise many concerns, e.g. data privacy.

No matter how it turns out, SaaS market will probably experience significant growth in the following years and cloud computing will be a competitive hosting platform for theses software applications. This partial software paradigm shift has a lot of impact on software development companies. Among them, Microsoft is an illuminating example. According to its annual report for the fiscal year ended June 30, 2011, more than 83% of Microsoft revenues are split among three of its divisions: Windows & Windows Live Division ($ 18,778 millions), Server and Tools Division ($ 17,107 millions), and Microsoft Business Division ($ 21,986 millions). These 3 divisions all correspond to standard desktop software or operating system: Windows & Windows Live Division is mainly composed of Microsoft Windows, the Server and Tools Division mainly deals with Windows Server and SQL-Server, and the Business Division is related to desktop software such as Office, SharePoint, Exchange, Lync, Dynamics, etc. While one can therefore see that the Microsoft revenues were and still are mainly driven by the licence selling of their desktop software applications and of their operating system Windows, an important migration has been initiated toward cloud computing solutions to provide these applications as SaaS. This situation has made Microsoft both a cloud provider and a cloud consumer. This introduction gives a brief overview of the different Microsoft cloud computing solution layers.

A first key aspect of the Microsoft cloud computing solutions is this effective shift operated by Microsoft from desktop software to Software as a Service (SaaS). An increasing part of the software applications presently sold by Microsoft is indeed migrated to the cloud and is offered alternatively as a SaaS solution. Microsoft Office 365 for example, is a commercial software service offering a set of

Microsoft products. It is publicly available since 28 June 2011 and includes the Microsoft Office suite, Exchange Server, SharePoint Server, etc. Another important Microsoft cloud SaaS solution is the search engine Bing, which is run (at least partially) on their cloud computing platform (see [9]). The cloud SaaS version of these applications are targeted to any customer of the desktop version of the same applications and the use of these cloud applications do not require any software development knowledge. The consumer just use these applications as if everything was processed locally whereas the computations are actually run on the cloud.

A second aspect of the Microsoft cloud computing system is Microsoft cloud High Performance Computing (HPC) solution. This solution is mainly based on Dryad and Dryad-LINQ (see [**?**] and [13]). A new Azure component available since november 2011, called Windows Azure HPC Scheduler includes modules and features that enable to launch and manage high-performance computing (HPC) applications and other parallel workloads within a Windows Azure service. The scheduler supports parallel computational tasks such as parametric sweeps, Message Passing Interface (MPI) processes, and service-oriented architecture (SOA) requests across the computing resources in Windows Azure. The Azure HPC solution is targeted to be the corresponding cloud version of the Microsoft HPC solution on Windows Server.

The purpose of this chapter is to describe the Microsoft cloud components that are the elementary building blocks upon which are build cloud applications. These elementary building blocks are used by Microsoft to run its SaaS applications like Office 365 or its HPC solutions (Azure HPC), but are also used by external companies to build other cloud applications. For example, Lokad has migrated its forecasting engine on the cloud to benefit from the scalability and elasticity provided by Azure. The Azure elementary building blocks are gathered in the form of a Platform as a Service (PaaS) solution (see section **??**). This solution, referred as Windows Azure Platform, is marketed for software developers to store data and to host applications on Microsoft's cloud. Azure let developers deploy and run robust and scalable applications. Built upon geographically distributed datacenters all over the world, Azure also provides low-latency responsiveness guarantees for the hosted web applications and a solid framework for CPU-intensive processing tasks. On the contrary to the SaaS part of their cloud computing solution, Windows Azure is targeted to developers only.

Windows Azure Platform is composed of a persistent storage (Azure Storage) and of a cloud operating system (Azure Compute) that provides computing environment for applications. In addition to these two core blocks, many tools are available to help developers during the application building, deployment, monitoring and

maintenance. This chapter provides a short overview of the Azure Platform. It is organized as follows. The first following section presents Azure Compute. We give there a definition of the main components of a Windows Azure application. The section 3 defines the different parts of Azure persistent storage system and the architecture proposed to build a scalable storage system. The section 4 is dedicated to raw performances of the storage and of the computing instances. We report for example bandwidth and Flops we manage to get. These performances have a crucial impact on algorithms since they determine their design and potential performances. Last section 5 introduce some key numbers to estimate the total price of running algorithms on Azure.

2 Windows Azure Compute

Windows Azure Compute is exposed through hosted services deployable to an Azure datacenter. Each hosted service corresponds to a specific web application and is composed of roles, each of this role corresponding to a logical part of the service. The different roles that can be part of a hosted service are: the **web roles**, the **worker roles** and the **Virtual Machines (VM) roles**. A single hosted service is composed of at least one web role and one worker role. Each role is run on at least one virtual machine, referred as a role instance or a worker, so a single hosted service requires at least two role instances. We now briefly describe the different roles:

- Web roles are designed for web application programming. Web Roles allow public computers to connect to the hosted service over standard HTTP and HTTPS ports. VM running a given web role are pre-configured with IIS7 (Internet Information Services) and specifically designed to run Microsoft web-programming technologies as ASP.NET or Windows Communication Foundation (WCF), but they also support native code such as PHP or Java to build web applications.

- Worker roles are designed to run general background process. These processes can be dependant of a web role (handling the computation required by the web role) or independent. One of the difference between web and worker roles is that worker roles don't come with a pre-installed IIS. Worker roles execution code can be defined using the .NET framework.

- VM roles are designed to provide developers a much wider scope of possibilities and especially to control the operating system image. VM roles should

not be used unless worker and web roles do not fit the developer purpose, as it is the case for example when one have long and complicated installations in the operating system or a setup procedure that cannot be automated. In VM roles, the developer will upload his own virtual hard drive (VHD) that holds a custom operating system (more specifically a Windows Server 2008 R2 image) that will be run on the different VM running the VM role. This role will not be used in our cloud algorithm implementations.

The role is the scalability unit of a given hosted service. For each role, the number of role instances that are run is a user-defined quantity that can be dynamically and elastically modified by the Azure account owner through the Monitoring API or the Azure account portal. Azure Compute manages the lifecycle of role instances. More specifically, Azure by itself monitors that all the role instances are alive and available. In the event of a failure, the failing role instance is automatically restarted on a different virtual machine.

By consequence, developers are not expected to handle the individual virtual machines by themselves: they just implement the logic for each role and upload the code to Azure through a package from the Azure administration portal, or directly from Visual Studio. An Azure-side engine called Azure Fabric handles the package deployment on as many virtual machines as requested. Similarly, the role update requires the developer to only update the role and not each of the role instances, which are automatically handled by Azure.

While the role instances have not been designed to be manually controlled, it is still possible to directly access them. More specifically, an internal endpoint is exposed by each role instance so that the other role instances that are related to the same role could access it. These internal endpoints are not visible outside the hosted service.

3 Windows Azure Storage

The Windows Azure Storage (WAS) is the storage component of the Windows Azure Cloud Platform. It is a public cloud service available since November 2008 and presently (in January 2012) holds 70PB of data. It is used as an independent storage service but also as the persistency storage for the applications run on Windows Azure Cloud Platform. According to [9], the WAS is used internally by Microsoft for applications such as social networking search, serving video music

and game content but also outside Microsoft by thousands of customers.

The WAS has been designed to be highly scalable, so that a single piece of data can be simultaneously accessed by multiple computing instances and so that a single application can persist terabytes of data. For example, the ingestion engine of Bing used to gather and index all the Facebook and Twitter content is reported to store around 350 TeraBytes of data in Azure (see [9]).

The WAS provides various forms of permanent storage components with differing purposes and capabilities. The following section describes these components.

3.1 WAS components

The WAS is composed of four different elements: Windows Azure BlobStorage, Windows Azure TableStorage, Windows Azure QueueStorage (respectively referred in the following by BlobStorage, TableStorage and QueueStorage), and Windows Azure SQL.

- The BlobStorage is a simple scalable Key-Value pairs storage system. It is designed to store serialized data items referred as blobs.

- The TableStorage is also an alternative Key-Value pairs storage system. It provides atomicity guarantees in a constrained scope, as detailed below.

- The QueueStorage is a scalable queues system designed to handle very small objects. It is used as a delivery message mechanism across the computing instances.

- Azure SQL is a relational cloud database built on SQL-Server technologies. It is designed to be an on-demand RDBMS and requires no setup, installation and management.

A common usage pattern for the different storage elements is as follows: I/O data transfert is hold through the BlobStorage, overall workflow instructions and job messages are hold by the QueueStorage and describes how the blobs need to be processed, and intermediate results or services state are kept in the TableStorage or the BlobStorage.

3.1.1 Azure BlobStorage

Azure BlobStorage is a large-scale key-value pair storage system. It is designed to be the main storage component of most azure applications.

The objects stored in the BlobStorage are stored in Binary Large OBjects (referred as blobs). The size of a blob is bounded to 50GBytes. The blobs are stored on multiple nodes, and a complex load-balancing system (partially described in section 3.2) ensures the system is at the same time scalable and strongly consistent. The strong consistency is an important aspect of the BlobStorage since it has an impact on overall latency and scalability of the BlobStorage, as explained in **??**. This property is a key characteristic of the WAS, as many other No-SQL storage (such as Cassandra) do not provide such a guarantee. The strong consistency implies that each blob is immediately accessible once it has been added or modified, and that any subsequent read from any machine will immediately see the changes made by the previous write operations.

Each object stored in the BlobStorage is accessed through its key. The key is a simple two-level hierarchy of strings referred as containerName and blobName. ContainerName and blobName are scoped by the account, so two different accounts can have same containerNames or blobNames. The BlobStorage API provides methods to retrieve and push blobs for a given key, and a method to list all the blobs whose key share a given prefix.

A key feature of data storages is the way multiple data can be accessed and modified simultaneously. As detailed in section **??**, a single transaction may indeed involve multiple pieces of data at once. In the context of RDBMS, transactions are guaranteed to be atomic, so each piece of data concerned by a transaction is updated accordingly or none is (if an update fails). The Azure BlobStorage does not give any primitive to atomically modify **multiple** blobs. Because of the internal design of the WAS, the different blobs are stored on multiple machines distributed on separate storage stamps (we refer the reader to more in depth details in section 3.2). This physical distance between the storage nodes of different blobs involved in a transaction preclude low latency atomic multi-blob transactions. This is why there are no primitive on Azure side to run transactions over multiple blobs atomically.

Blobs can be stored in two different blobs formats : block blobs and page blobs.

- Block blobs are optimized for streaming. Each block blob consists of a sequence of blocks, each block being identified by a block ID. The size of

a block is bounded to 4MB. Writing to a block blob is a two-step process relying on a commit-based update semantics. Firstly, a block is uploaded to Windows Azure using the *PutBlock* method of BlobStorage API, but it does not become part of the blob until it is commited. The second step consists in a commit-phase : when all the blocks have been uploaded, the user calls the *PutBlockList* that specifies the list of block IDs that make up the blob. The list of blocks that form the new blob version is commited atomically.

- Page blobs are targeted for random writes. Each blob consists of an array of pages, each page being a range of data that is identified by its offset from the start of the blob. Pages blobs update is an immediate update semantics: as soon as a write request for a page or a sequential set of pages succeeds on the storage side, the write is committed, and client is notified back of write success.

3.1.2 Azure TableStorage

Azure TableStorage is the second key-value pairs storage component of the WAS. It is designed to provide a more adapted storage for structured data used in Azure applications. The structured storage is provided in the form of tables. A single application may create one or multiple tables, following a data-driven design. Each table contains multiple entities. To each entity is associated two keys in the form of strings: the first key is called the PartitionKey and can be shared with some other entities of the same table, while the second key is specific to each entity and is called the RowKey. A given entity is therefore uniquely referenced as the combination of its corresponding PartitionKey and RawKey. Each entity holds a set of $< name, typed\ value >$ pairs named Properties. A single table can store multiple entities gathered into multiple different PartitionKeys.

In a given table, each entity is analogous to a "row" of a RDBMS and is a traditional representation of a basic data item. This item is composed of several values, filled in the entity's properties. Each table is schema-free to the extent that two different entities of a given table can have very different properties. Yet, a common design pattern consists in adopting a fixed schema within a given table so all the entities have the exact same set of properties.

Because of the absence of a mandatory fixed schema, even in the case of the previous design pattern where the schema is *de facto* fixed, the TableStorage does not provide any way to represent relationships between the entities' properties

on the contrary to other structured storage as RDBMS. Another consequence of this flexible schema is the lack of "secondary" indices in a given table: the only way to enumerate entities in a table rely on the entity key, namely the RowKey. Therefore, selecting entities according to a criterion based on a given property will lead to execution time linear in the number of entities in the table (on the contrary to RDBMS, where secondary indices may lead to requests time proportional to the logarithm of the number of entities).

An additional feature of the TableStorage is to allow reads, creations or updates of multiple entities in a single command. These commands, referred to entity group transactions, support executing up to 100 entities in a single batch, provided the fact the entities are in the same table and share the same PartitionKey. In addition to atomicity guarantees described in section 3.1.5, the entity group transactions are of prime interest since they provide a way to significantly lower (up to a factor 100) the price of storage I/O in the case of small objects being stored.

3.1.3 Azure QueueStorage

Azure Queues provide a reliable asynchronous message delivery mechanism through distributed queues to connect different components of a cloud application. Queues are designed to store a large amount of small messages (with maximal individual size of 8 KB, see for example [6]). Using queues to communicate helps building loosely coupled components and mitigates the impact of individual component failure.

Azure queues are not supposed to respect FIFO logic (First In First Out) as standard queues. At a small scale, the Azure queue will behave like a FIFO queue, but if it is more loaded, it will adopt a different logic so it can better scale. Therefore, the FIFO assumption cannot be assumed when designing a cloud application that uses queues.

Messages stored in a queue are guaranteed to be returned at least once, but possibly several times: this requires one to design **idempotent** tasks. When a message is un-queued by a worker, this message is not deleted but it becomes invisible for other workers. If a worker fails to complete the corresponding task (because it throws some exception or because the worker dies), the invisibility timer happens to time-out and the message becomes available again. If the worker processes the message entirely, it notifies the queue that the processing has been completed and that the message can be safely deleted. Through this process, one can make sure no task is lost because of e.g., a hardware failure.

3.1.4 Synchronization Primitives

Item updates mechanism for BlobStorage and TableStorage is achieved through an optimistic nonblocking atomic read-modify-write (RMW) mechanism. A nonblocking update algorithm consists in updates executed speculatively, assuming no concurrent machine is updating the data at the same time. The nonblocking update algorithms do not imply synchronization or locking mechanisms when an update is executed. However, a check is performed at the end of the update to make sure that no conflicts have occurred. In the case of a conflict, the update is aborted and need to be redone. Nonblocking update algorithms are often called optimistic because they bet on the fact conflicts are statistically unlikely to happen. If concurrent writes happen, then the update mechanism is likely to take more time than a simple locking update process. Yet on average, the optimistic update algorithms are much more efficient than locking algorithms.

The optimistic RMW mechanism is allowed in Azure through timestamp referred as etag, following Azure terminology. This timestamp indicates the exact date of the last successful write operation applied to the item inside the BlobStorage or the TableStorage. The Azure item update is performed as follows: the item to be updated is downloaded by the computer in charge of the item update in addition to the corresponding etag of the item. The actual value of the item is locally updated by the computer, then pushed back to Azure BlobStorage or TableStorage with the previous etag. If the returned etag matches the present etag of the item version stored in the WAS, then the item is actually updated. If the two etags do not match, then a distinct machine has been concurrently updating the same item, and the updating process is relaunched.

The RMW mechanism is a classical synchronization primitive (see for example [11]). Implementing the timestamp system and an efficient conditionnal write mechanism is a difficult task on architectures without hardware shared-memory like Azure since a single piece of data is replicated multiple times on different machines. More specifically, special attention on Azure side is paid to lower the overall write latency. A part of the adopted design is described in section 3.2.

3.1.5 Partitions, Load-Balancing and multi-items transactionality

To load-balance the objects access, the WAS is using a partitioning system. This partitioning system is build upon a partition key in the form of a string. For the BlobStorage, the partition key of a blob is the concatenation of its containerName and of its blobName strings. In the case of the BlobStorage, each partition is

therefore only holding a single blob, and it is impossible to gather several blobs into a single partition. For the TableStorage, the partition key is the PartitionKey string already introduced in section 3.1.2. In this case, multiple entities can be gathered in a single partition provided the fact they are in the same table and share the same partition key.

The load-balancing system is separated into two load balancing levels. The first load-balancing level is achieved by Azure itself and provides an automatic inter-partitions load-balancing. Nodes dedicated to handle data requests are called partition servers. The inter-partitions load-balancing system maps partitions to partition servers. A single partition server can handle multiple partitions. But for a given partition, a single partition server is in charged of addressing all the requests to this partition, while the effective data of the partition can be stored on multiple nodes (we refer the reader to the section 3.2 for more in depth explanations). The WAS monitors the usage pattern of the request on the partitions and can dynamically and automatically reconfigure the map between partitions and partition servers.

Since all the requests related to a single partition are addressed by a single partition server, the partition is therefore the smallest unit of data controlled by Azure for load-balancing. The second load-balancing system is handled by the user. This second system refers to the user's ability to adjust the partition granularity so that a given partition handles more or less data. More specifically, the WAS consumer needs to tune the partition granularity with respect to a atomicity/scalability trade-off. The bigger a partition is, the more transactions on multiple data pieces can be done atomically. Yet, since all the requests to this partition are addressed by a single partition server, the bigger a partition is the more busy the partition server will be.

For the BlobStorage, since each object is stored in a different partition key, access to different blobs (even with the same containerName) can be load-balanced across as many servers as needed in order to scale access, but no atomicity for transactions is possible on Azure level. On the contrary, the TableStorage provides atomicity for transactions run on a group of entities sharing the same partition key.

3.2 Elements of internal architecture

The internal implementation of the WAS is rather complex and leverages a multi-layer design. This design lets the storage be at the same time strongly consistent, highly available and partition tolerant. To provide at the same time these three

properties is a challenge known to be difficult, at least theoretically, due to the CAP theorem. This section provides a short insight of the underlying infrastructure and shows how the previous guarantees are achieved despite of the difficulty mentioned above. We refer the reader [9] for a more in depth presentation of the WAS underlying architecture.

3.2.1 Storage stamps

Following the terminology of [9], the computing instances in a datacenter are divided into storage stamps. Each storage stamp is a cluster of about 10-20 racks, each rack holding about 18 disk-heavy storage nodes. While the first generation of storage stamps hold about 2PB of raw data in each stamp, the next generation hold up to 30PB of raw storage in each stamp[1]. To lower the cost of cloud storage, Azure team keeps the storage stamps highly utilized. But to keep sufficient throughput and high availability even in the presence of a rack failure, each stamp is not used above 70% of its capacity. When a storage stamp is filled up to this bound, the location service migrates some accounts to different stamps, to keep the storage stamp on a capacity usage ratio of 70%.

Each rack within a storage stamp is supported on isolated hardware: each rack is supplied in bandwidth and energy with independent and redundant networking and power, to be a separate fault domain.

The datacenters hosting WAS services are spread in several regions of North America, Europe and Asia. In each location, a datacenter holds multiple storage stamps. When creating a new cloud application on Azure, customers are asked to choose a location affinity among these three continents. While the cloud consumer cannot choose the exact datacenter location being primarily used by its application, this location affinity choice lets the user lower its application latency and improve its responsiveness.

[1]These numbers may seem to be surprisingly high in view of the total amount of data stored in Azure presented in section 3. All these numbers yet come from the same source: [9]. A first response comes from all the data replicas that are created for each data piece. A second key aspect may be the Content Delivery Network (CDN), a special Azure service devoted to serve content to end users with high availability and high performance. Finally, the quantity of machines dedicated to the storage system may be small in comparison to the total amount of machines required by computations.

3.2.2 Front End Layer

The Front-End layer is composed of multiple servers processing the client incoming requests and routing them to the partition layer. Each front-end server holds in cache the mapping between partition names (as defined in section 3.1.5) and partition servers of the partition layer. When receiving a client request, the Front-End layer server authenticates and authorizes the request, then routes it to the partition server that primarily manages the partition name related to the request. The most frequently accessed data are cached inside the Front-End servers and are directly returned to corresponding requests to partially unload the partition layer servers.

3.2.3 Partition Layer

The Partition layer is composed of multiple servers managing the different partitions. As already stated in 3.1.5, one partition server can manage multiple partitions, but a given partition is managed by only one partition server. For the BlobStorage, this is obvious since the partition key is down to the blobName. A partition master monitors each partition server's load and tunes the load-balancing process.

3.2.4 Stream Layer

The Stream layer is responsible of persisting data on disk and replicating the data across multiple servers within a storage stamp. The stream layer is a kind of distributed file system within a storage stamp.

Following the terminology of [9], the minimum unit of data for writing and reading is called a Block. A typical Block is bounded to 4 MBytes. An extent is a set of consecutive and concatenated blocks. Each file stored by the stream layer is referred as a stream. Each stream is a list of pointers referring to extent locations.

The target extent size used by the partition layer is 1 GBytes. Very large objects are split by the partition layer into as many extents as needed. In the case of small objects, the partition layer appends many of them to the same extent and even in the same block. Because the checksum validation are performed at the block level, the minimum read size for a read operation is a block. It does not mean that when a client requests a 1 kilobyte object the entire block is send back to the customer, but that the stream layer reads the entire block holding the object before returning the actual small object.

Because a stream is only an ordered collection of pointers to extents, a new stream can be very quickly created as a new collection of pointers of existing extents.

A key feature of the stream layer is that all writes are append-only. When a write occurs, the last extent is appended with one or multiple blocks. This append operation is atomic: either all the entire blocks are appended, or none are. The atomicity is guaranteed by a two-step protocol. Blocks are firstly written into the disk and appended after the last block of the last extent of the stream. When these write operations are completed, the stream block and extent pointers list are updated accordingly.

An extent has a target size, defined by the partition layer. When an append enlarge an extent to the point it exceed this target size, the extent is sealed and a new extent is added to the end of the stream. Therefore, all the extents but the last one are immutable in a stream and only the last one can be appended.

Partition servers and stream servers are co-located on the same storage nodes.

3.2.5 Extent replication and strong consistency

A key feature of data storage is data persistency, durability and availability. Since data servers are expected to fail from time to time, data are duplicated several times, so a data server can die without any data loss. Yet, in the case of a networking partitioning, availability and consistency are two properties that hard to met at the same time, as stated by the CAP theorem (see **??**).

The default azure policy is to keep three replicas for each extent within the same storage stamp (according to [9]). This policy is referred as the intra-stamp replication policy and is intended to guarantee no data loss, even in the event of several disks, nodes or rack failures. A second policy designed to prevent data loss in the rare event of a geographical disaster is called inter-stamp replication policy. This replication process ensures that each data extent is stored in two different datacenters, geographically distant. This section details how the inter and intra-stamp replication policies fit into the CAP theorem constraints.

The stream layer is responsible of the intra-stamp replication process. When an extent is created, the Stream Manager promotes one of the three extent nodes (EN) into a primary EN. The partition layer is told of the primary EN, and all the write requests of the partition layer are send to this primary EN. The primary EN is in charge of applying the write request to its extent but also to make sure the two other replicas are updated the same way. It is only once all the three extents have

been updated that the partition layer is notified by the primary EN of the success of the write request. Therefore, this update process is kept as quick as possible since it impacts customer's request latency as no write success is returned before all the extents are updated. Because of all the three extents are located in the same storage stamp, this update process can be kept quick. This update mechanism ensures strong consistency among the three extent replicas, and each read request can be addressed by any of the EN since all the three replicas are always guaranteed to be the same.

During a writing event, if an extent node is unreachable or the write operation fails on this node, the partition layer is notified the write operation has failed. The Stream Manager then sealed the extent in the state it was before the write failure, adds a new unsealed extent in the end of the stream and promotes one of the three extent nodes as a primary node for this extent. This operation is reported in [9] to be completed on average within 20ms. The partition layer can therefore resume the write operation in a short time without waiting for the failing nodes to become available or operational again.

The partition layer is responsible of the inter-stamp replication process. It is a low priority *asynchronous* process scheduled so that it does not impact customer's requests performance. On the contrary to the intra-stamp replication process run by the stream layer, the inter-stamp replication process is not designed to keep data durable in the event of hardware failures. Rather, it is designed to store each data chunk in two distinct datacenters to obviate the rare event of a geographical disaster happening in one place. It is also used to migrate accounts between stamps to preserver the 70% usage ratio of each storage stamp. This replication process does not guarantee the strong consistency.

3.3 BlobStorage or TableStorage

BlobStorage and TableStorage are both No-SQL storages, designed to store and persist data through a key-value pairs format. Both of them are well integrated into the Azure Platform solution. We now investigate some elements that need to be taken into account while choosing one storage or the other:

- When dealing with really large objects, BlobStorage requests are easier to express, since each blob can be arbitrarily large (up to 50 GB per blob). We do not need to break an object into a lot of entities, making sure each entity conforms to the size constraint on entities.

- When manipulating objects than can be stored both in TableStorage and BlobStorage, BlobStorage is reported by [14] to be more effective in insert operations on blobs larger than 4KBytes.

- As already expressed in section 3.1.5, data locality management and atomicity concerns can also impact the choice between the BlobStorage and the TableStorage. While it is still possible to design custom primitives for multi-blobs atomic transactions (see section**??**), these primitives are not native and less efficient.

- Price is another factor that might affect our choice. Using Azure Storage, three things are charged : the storage used size, the I/O communications, and the requests to the storage. A single query on the TableStorage can load up to 100 elements at a time and costs as much as a single query on the blobStorage (see [2]). When dealing with many small objects, using TableStorage will thus allow to run requests on objects by groups of 100 entities, so it would divide up to a factor 100 the bill of storage requests in comparison of running the corresponding requests one by one on the BlobStorage. If we do not need to store the objects on different places, we could store them in one bigger blob, bounding the request to one. Such a design choice would lead to significant improvements when all the items thus grouped are all read or updated at the same time. In the case of random read and write accesses to a pool of items, grouping them into a single item leads to significant I/O communication increase. The usage of the TableStorage while dealing with small objects therefore seems to be more adequate but special attention is required.

All the cloud experiments run on chapter **??** and chapter **??** have been implemented using the BlobStorage. Atomic updates on multiple objects at once will not be mandatory for most of our storage requests. Besides, the BlobStorage is easier to manipulate and due to its complexity, the TableStorage suffered some bugs when we started our experiments (in June 2010).

4 Azure Performances

4.1 Performance Tradeoffs, Azure Positionning

As outlined in [8], no one cloud system can be best for all application requirements, and different storage systems have made different choices while facing some tradeoffs. Before providing raw numbers of Azure performances, we try to position the Azure framework in comparison of these tradeoffs:

Read/Write tradeoff: There is a natural tradeoff between read throughput and write throughput. A piece of data cannot be at the same time highly available for read operations and write operations: having a highly available item for read operations requires the item to be duplicated multiple times; write operations must then change all replicas, which slows down the write operations. Azure let the traffic arbitrate this tradeoff: the more a blob is requested per time unit, the more the blob is duplicated and the more it becomes available. In return, the more the blob is duplicated the more time it takes to run write operations. Azure also implements an in-memory mechanism that mitigates this tradeoff: the blobs the most requested (the hot blobs) are kept in cache instead of on hard-drive to lower read and write latencies. Such a mechanism is adopted in many other storage systems. The impact of both these mechanisms must however be tempered by the fact they rely on requests spread over time: to detect that a blob is "hot" and that it needs to be both duplicated and put into cache, Azure Storage needs to first notice a significant amount of requests before launching these mechanisms. This design makes the mechanism ineffective in the case of a blob which is once read by multiple workers at the same time, then no more read ever. Such a scenario is frequent in the algorithms presented in chapter **??** and **??**.

Consistency/Persistency/Latency tradeoff: Data persistency is a mandatory feature for a storage service such as Azure Storage. To prevent client data loss, Azure stores each single piece of data multiple times in different places to guarantee data persistency[2]. As explained in section 3.2.5, such a replication mechanism induce longer write operations when strong consistency is also guaranteed. To lower the overall write latency, Azure implements a two-level replication mechanism. The first one, referred in section 3.2.5 as intra-stamp mechanism, is a synchronous

[2]Such guarantees need to be viewed with caution. Despite Microsoft commitments and their Service Level Agreement (SLA), the duplication mechanism can only protect data from hardware failures, not from software bugs or human manipulation errors (as it is the case for all the technologies). This has been evidenced several times, for example by Amazon EC2 outage started April 21th 2011, or Azure outage started February 29th 2012 (for which Lokad has suffered from web-site downtime for several days and temporary loss of a part of its data).

process: the write operations returns success only when all the replicas have been updated. Since all those replicas are stored in a single storage-stamp, all the writes are likely to be completed in a small amount of time. This mechanism guarantees strong consistency among all these replicas. The second replication process, referred as the extra-stamp mechanism, is an asynchronous mechanism: when the storage is not stressed, the data piece is replicated in another geographically distant storage-stamp. Such a mechanism does not guarantee strong consistency, but is only used as back-up in the very unlikely event of a complete loss of the first whole storage-stamp.

CAP guarantees / latency tradeoff: This tradeoff has been partly already discussed in the Consistency/Persistency/Latency tradeoff. As stated by the CAP theorem, a distributed storage cannot be 100% strongly consistent, 100% available and 100% partition tolerant. Azure has chosen to guarantee none of these properties at a 100% rate. The strong consistency is guaranteed provided that the whole primary storage stamp of a given data chunk is not made completely unavailable (which is an event very unlikely to occur and which didn't occur to our knowledge during all our months of experiments). The system is also neither 100% available nor 100% partition tolerant. Yet, Azure framework is performing well enough in overall for these lack of guarantees to be hardly noticed.

Synchronous actions / Responsiveness tradeoff : Many asynchronous methods are designed to improve responsiveness. In the special case of an API, it is advised that methods that internally are implemented synchronously should be exposed only synchronously and methods that internally are implemented asynchronously should be exposed only asynchronously (see for example [7]). To our knowledge, this is the design adopted by Azure Storage. More specifically, all the List, Get, Push and Create methods are all exposed synchronously. A single method is exposed asynchronously: it is the container deletion method. The main priority performance targets of the WAS are low latencies for read and write operations in addition to throughput. The container deletion method, likely to be used very infrequently, is not designed to be efficient. This method is therefore made asynchronous and cloud consumers should not recreate a freshly deleted container.

4.2 Benchmarks

Benchmarking a storage system is a challenging task involving a lot of experiments and measurements in multiple contexts. Some previous works have already been made to evaluate some cloud storage solutions. For example, the Yahoo! Cloud Serving Benchmark ([8]) provides an open-source framework to evaluate storage

services. Other measurement works have been made on specific cloud storage solutions, such as [10] or [12] for Amazon and [14] or the AzureScope website [5] for Azure.

In the context of our work, we do not aim to produce such a rigorous and comprehensive evaluation of the Azure services performances. We have yet observed that the performances of Azure are very dependent of the context in which the requests are applied (in particular, the size of the data that is uploaded or downloaded, the API used to communicate with the BlobStorage, the serializer/deserializer that is used, etc. are impacting performances). As a consequence, even the previously mentioned papers that have been run in a rigorous context might be inaccurate in our specific situation. Because of this, we provide in this section naive measurements made in the specific context of our clustering experiments (see chapter **??** and chapter **??**). In particular, the storage bandwidth as well as the CPU performance are investigated as they are of prime interest for the design of our algorithms.

4.2.1 Bandwidth between the storage and the role instances

On the contrary to [14], we investigate the read and write bandwidth for small data chunks (several MBytes). To this extent, we pushed 8 blobs of 8 MBytes into the storage, and we measured the time spent to retrieve them. For a single worker using a single thread, we retrieved the blobs in 7.79 seconds on average, implying a 8.21 MB/sec read bandwidth. We tried to use multiple threads on a single worker as advised by AzureScope ([5]) to speedup the download process, and we found the best read bandwidth was obtained using 5 threads: we retrieved the 8 blobs in 6,13 seconds on average, implying a 10.44 MB/sec read bandwidth. The multi-threads read bandwidth we observed differs from what is achieved in AzureScope. This may be explained by two potential factors: we observed average bandwidth whereas AzureScope observed peak performances and we did not have the same experiment environment.

This benchmark is very optimistic compared to the downloading context of our clustering experiments run in chapter **??** because of a phenomenon referred as the aggregated bandwidth boundaries discussed in section **??**. In addition to this aggregated bandwidth boundaries phenomenon, multiple concurrent I/O operations run in parallel can sometimes overload a specific partition layer server, resulting in slower request responses: when running experiments with a large number of workers reading and writing in parallel, we sometimes experienced blobs already pushed into the storage becoming available only after 1 or 2 minutes.

4.2.2 Workers Flops performances

In the same way than for the storage bandwidth, the workers CPU cadency highly depends on the kind of tasks the workers are assigned to. All our experiments were run on Azure small VM that are guaranteed to run on 1.6 GHz CPU. Yet because of virtualization we do not have any warranty in term of effective Floating point operation per second (Flops). On any architecture, this Flops performance highly depends on the nature of the computation that are run and especially of the code implementation (see for example [**?**]). Therefore, to fit our predictive speedup model developed in chapter **??**, we ran some intensive L2 distance computations to determine how fast could our algorithm be run on these VM. As a control experiment, the code was first run on a desktop Intel Core 2 Duo T7250 2*2GHz using only one core. We noticed that our distance calculations were performed with a performance of 750 MFlops on average. We then ran the same experiment on Azure small VM and we got for the same code a performance of 669 MFlops.

It is worth mentioning that differences in processing time between workers have been observed. These observations are developed in the section **??** of chapter **??**.

4.2.3 Personnal notes

During all the experiments that we have been running (from September 2010 to August 2011), we have noticed various Azure oddities. The following remarks do not come from rigorous experiments designed to prove these oddities, but come as side observations made while running our clustering algorithms.

- **Inter-day Variability of Azure performances**: during the months of our experiments, we have noticed several days for which the Azure storage performances were significantly below average performances (from 30 to 50% below). [14] reports that "the variation in performance is small and the average bandwidth is quite stable across different times during the day, or across different days". This different results may come from the different environments where the measurements took place. More specifically, our experiments have been run on the Azure account of Lokad, account for which other hosted services were sometimes run concurrently to execute heavy computations. It is also noteworthy to mention the date of experiments. The experiments run by [14] have been run from October 2009 to February 2010 (during the Community Technology Preview and before the commercial availability of Azure), while our experiments have been run from September 2010 to August 2011. Such variations make performance measurements of

distributed algorithms harder. Especially, experiments need to be performed several times before conclusions could be drawn.

- **Sensitivity to the serialization mechanism**: different serializers (to name a few of them available in .NET: Binary, DataContract, XML, zipped-XML, JSON, ProtocolBuffers, etc.) are available to serialize an in-memory objet into a stream then stored in disk or send through the network. The choice of serializer has an impact on the overall communication latency and with storage duration.

- **Temporary unavailability of a blob**: we have been experiencing several times the temporary (around 15 minutes) total unavailability of a given blob when this blob was read by about 200 machines simultaneously. We have not managed to isolate the exact reason for this phenomenon.

- **Temporary unavailability of the storage**: we have been experiencing several times the temporary (around 15 minutes) total unavailability of the BlobStorage. We have not managed to isolate the exact reason for this phenomenon.

- **Lower throughput for smaller blobs**: while the experiments in [14] have been run on a blob of 1GBytes, a lot of interactions with the storage are made through much smaller blobs. The throughput for these smaller blobs is sometimes reduced.

- **Code redeployment, VM pool resizing**: During our experimenting year, we have been noticing impressive improvements in code redeployment and dynamic worker resizing. More specifically, our first code redeployments (December 2009) were taking hours (with failing redeployments failing after more than four hours). In August 2011, code redeployment was systematically taking less than 10 minutes. This redeployment mechanism is still improvable: for example, Lokad.Cloud provides a code redeployment within several seconds through a clever trick of separate AppDomain usage (see [1]). During the same period, dynamic worker resizing has also been significantly improved, with pending reduced from approximately one hour to several minutes (at least when requiring less than 200 small role instances).

- **Queues pinging frequency**: Workers are pinging queues on a regular basis to detect if there are messages to process. To limit the number of network pings, a standard schedule policy consists in pinging the different queues only once per second. This policy introduces a small additional latency.

5 Prices

The pricing of Microsoft Azure is driven by the pay-as-you-go philosophy of cloud computing. It requires neither upfront costs nor commitments [3]. Cloud consumers are charged for the compute time, for data stored in the Azure Storage, for data queries, etc. This section gives a short overview of the Azure pricing system. We refer the reader to [3] to an accurate pricing calculator and to [4] for more in-depth pricing details.

CPU consumption is measured in hours of usage of VM. The figure 1 presents the Azure pricing for the different VM sizes. The use of Azure Storage is charged following three criterion: the quantity of data stored by Azure storage($ 0.125 per GB stored per month), the quantity of data which is transferred from Azure storage to other machines ($ 0.12 per GBytes for North America and Europe regions) (all the inbound data transfers or between 2 Azure instances are at no charge), and the number of requests addressed to the storage ($1 per 1,000,000 storage transactions).

This pricing can be lowered through subscriptions and are expected to be cut in near future: indeed, Azure pricing is in general close to Amazon pricing that has been cut nineteenth time in just six years.

Instance	CPU Cores	Memory	Disk	I/O Perf.	Cost Per Hour
Extra Small	Shared	768 MB	20 GB	Low	$0.02
Small	1 x 1.6 GHz	1.75 GB	225 GB	Moderate	$0.12
Medium	2 x 1.6 GHz	3.5 GB	490 GB	High	$0.24
Large	4 x 1.6 GHz	7 GB	1,000 GB	High	$0.48
Extra Large	8 x 1.6 GHz	14 GB	2,040 GB	High	$0.96

Table 1: Compute instance price details provided by Microsoft on April 2012. http://www.windowsazure.com/en-us/pricing/details/.

The costs described above seem to traduce the fact that storing data is much cheaper than to use computing units. For many applications hosted by Azure, most of the

[3] No commitments are required, but customers can receive significant volume discounts when signing for commitment offers for several months. In addition, the development cost to migrate an application on a Cloud Computing Platform is often significant, which is a form of strong commitment

costs will therefore come from the VM renting.

References

[1] Appdomain trick. http://code.google.com/p/lokad-cloud/wiki/ExceptionHandling read the 26/04/2012.

[2] Azure pricing. http://www.microsoft.com/windowsazure/pricing/.

[3] Azure pricing calculator. http://www.windowsazure.com/en-us/pricing/calculator/advanced/ read the 25/04/2012.

[4] Azure pricing details. http://www.windowsazure.com/en-us/pricing/details/ read the 25/04/2012.

[5] Azure scope. http://azurescope.cloudapp.net/.

[6] Azure storage resources. http://blogs.msdn.com/b/windowsazurestorage/archive/2010/03/28/windows-azure-storage-resources.aspx.

[7] Should i expose synchronous wrappers for asynchronous methods? http://blogs.msdn.com/b/pfxteam/archive/2012/04/13/10293638.aspx read the 26/04/2012.

[8] Erwin Tam Raghu Ramakrishnan Russell Sears Brian F. Cooper, Adam Silberstein. Benchmarking cloud serving systems with ycsb.

[9] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM.

[10] T. S. Eugene Ng Guohui Wang. The impact of virtualization on network performance of amazon ec2 data center. Technical report, Dept. of Computer Science, Rice University.

[11] Maurice Herlihy and Nir Shavit. The art of multiprocessor programming, 2008.

[12] Edward Walker. Benchmarking amazon ec2 for high-performance scientific computing. Technical report.

[13] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar, and Gunda Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language.

[14] Ming Mao Arkaitz Ruiz-Alvarez Zach Hill, Jie Li and Marty Humphrey. Early observations on the performance of windows azure.