Lokad.Cloud

.NET execution framework for Windows Azure

Joannes Vermorel ([joannes.vermorel@lokad.com](mailto:joannes.vermorel@lokad.com))  
Rinat Abdullin ([rinat.abdullin@lokad.com](mailto:rinat.abdullin@lokad.com))

Last updated: May 2009

**Project manifesto:** Cloud computing is still a fairly recent paradigm. In our experience, building apps for the cloud is still a very rough process lacking many ingredients that are usually taken for granted when developing classical desktop or web apps.

At Lokad.com, we rely on the cloud for intensive computing operations. When we started migrating our technology toward the cloud, we quickly realized that most of our migration problems would be common to virtually any software company migrating too toward Windows Azure.

Thus, we decided to release Lokad.Cloud, our framework as open source. This framework implements basic scalable components such as cloud services but also provide guidance with patterns & practices for the cloud.

Lokad.Cloud is an execution framework for cloud back-end processing. This framework is intended as a **structural component to organize processes that happen within worker roles**.

Key orientations of Lokad.Cloud

* Strong typing.
* Scalable by design.
* No scheduler, queue messages are the schedule.
* Smooth scalability, each extra worker gradually reduces latency.
* Cloud storage abstraction.
* Implicit storage initialization.
* Decoupling entirely (de)serialization from business logic.

Key features of Lokad.Cloud

* **Queue Services** as a scalable equivalent of Windows Services.
* **Scheduled Services** as a cloud equivalent of the task scheduler.
* **BlobSet** as strongly typed scalable collections.
* **Locks** and **Cuid** to deal with synchronization when it can’t be avoided.
* **Scalable logs** and monitoring.
* **Service administration console** to start/stop cloud services.

# Scalable services

Through **Queue Storage**, Windows Azure offers a very scalable processing pattern where N producers can fill the queue, and M consumers can read the queue; N and M being arbitrarily large and everything happening concurrently. This pattern looks very robust to us, but used as such, it suffers from a couple of issues:

* No strong-typing.
* Large messages (i.e. larger than 8kb) must be manually handled through the Blob Storage.
* Queue initialization code is dead-code (run only once).

***Disclaimer****: Our intent is not to criticize the cloud abstractions shipped with Windows Azure. Quite the opposite, we believe that Microsoft made an excellent job in providing very meaningful scalable resources for Windows Azure; exactly the way it should be. The whole point of Lokad.Cloud is to leverage those resources in the most productive way from a software engineering viewpoint.*

## QueueService

In order to overcome those limitations, we introduce the notion of **cloud services**, as a rough yet scalable equivalent of the classical Windows Service but tailored for the cloud. Let’s have a look at the QueueService abstract class (only the relevant methods are included):

public abstract class QueueService<T> : CloudService  
{  
 public abstract void Start(IEnumerable<T> messages);  
 public IEnumerable<T> GetMore(int count);  
 public void Delete(IEnumerable<T> messages);  
 public void Put<U>(IEnumerable<U> messages);  
}

Implementing a scalable cloud service basically means inheriting QueueService, overriding the Start method, and that’s all. The Start method is expected to process incoming messages. When messages are processed, Start should return. This method will be called again if new messages become available.

The method Delete is used to mark messages as processed (and to remove them from the queue). The method Put<U> is used to put messages to the queue implicitly associated to the type U.

When Start returns (without thrown exceptions) all messages retrieved through Start or GetMore are automatically deleted – unless Delete has already been called.

Based on this implementation, the Lokad.Cloud automation unfolds with:

* Dynamic service instantiation through .NET reflection.
* Auto-initialization of a queue dedicated to processing messages of type T.
* Transparent blob wrapping of large messages that don’t fit into the Queue Storage.

Note that Lokad.Cloud provides ways to get more control over the name of the queue being plug into the QueueService and over other service settings, such as the service priority or the suggested number of messages to retrieve. Those settings can be applied through QueueServiceSettingsAttribute.

A key idea of Lokad.Cloud is to avoid worker specialization: **all workers are kept logically identical**. Each worker instantiates local instances of the services and then starts pulling messages from the corresponding queues. For the app developer, it means that all cloud services are deployed as a single WorkerRole without bothering with multiple specialized roles.

**Lokad.Cloud ensures than all workers are kept busy** as long as messages could be found in a queue.

The QueueService comes with a couple of subtleties related to processing delays of messages. In particular, Lokad.Cloud makes sure than a single message does not end up being processed concurrently twice because the message timeout expired (and consequently the message was made available again in the queue).

Also, Lokad.Cloud makes sure that a single heavy queue does not end up starving all the allocated cloud resources. Service priorities are provided to tune the overall latency of your cloud components.

**Technical notes:** In order to reduce cloud resources wasted in pulling empty queues, Lokad.Cloud is using a *smart* local scheduler that routinely probes queues but focusing on the queues that are the most likely to have a message ready to be processed. This problem, pulling the right queue at the right time, is similar to the multi-armed bandit (see <http://bandit.sourceforge.net/>) where a gambler devises a strategy to get the most of a gambling machine.

Then, overflowing messages, that is to say messages that are larger than 8kb, are automatically put in the Blob Storage and transparently passed to their corresponding services. Yet, messages on Windows Azure are silently garbage collected after 7 days in the queue. Thus, Lokad.Cloud applies a similar treatment to the overflowing items stored in the Blob Storage.

## Guidance for QueueService

Due to the nature of the cloud, **workers could not be assumed as reliable**. Simply put, workers are going to fail because of hardware failures or because cloud maintenance operations and, for the app developer, there is nothing that that can be done to avoid this issue altogether. Yet, Queue Storage provides a reliable processing pattern: **if a worker fails at processing a message, then the message later reappears in the queue** to be processed again by another worker.

According the Wikipedia: *Idempotence describes the property of operations in mathematics and computer science which means that multiple applications of the operation do not change the result.*

**If your QueueService logic is made idempotent** then no worker failure will have any impact on the ultimate consistency of your data (ultimate meaning here *after a non-specified but potentially large amount of time*). Thus, we suggest, whenever possible, to make your QueueService logic idempotent. In particular, the **message deletion** should always be the last call of your logic.

Then, it is important to use the **right granularity** for your queue services:

* If messages are too small (think a single Int32 value) and too numerous, computing resources will be wasted in network and framework overhead.
* If messages are too large (think 10GB blobs) then processing each message is likely to take a lot of time, and will lead to a poor overall scalability of your app.

The same considerations naturally apply to processing delays, independently of the size the messages.

## ScheduledService

A feature frequently found missing from Windows Azure is the availability of a **Task Scheduler** to automatically execute operations at certain times. Thus, Lokad.Cloud provides the ScheduledService that exposes a simple Start method:

public abstract class ScheduledService : CloudService

{

public abstract void Start();  
 public void Put<U>(IEnumerable<U> messages);  
}

Then, inheritors can be decorated with the ScheduledServicesSettings attribute to specify the execution frequency of the service (other settings are also available, but they are skipped here for the sake of clarity).

public class ScheduledServiceSettingsAttribute

{

public TimeSpan TriggerInterval { get; set; }  
}

The framework Lokad.Cloud ensures that the service is executed only once, in a single worker, for each scheduled execution time; eventually spreading the execution among many workers if there are frequent coarse grained scheduled operations.

Although, Lokad.Cloud is distributing the schedule executions among workers, we suggest **keeping scheduled operations as lightweight as possible**: the call to Start() must return as fast as possible. If the operation is heavy, then we suggest the ScheduledService to put a message in a queue for a later processing through a distinct QueueService. Following this guidance will improve the overall load balancing of your cloud app because it will help the framework to better distribute the workload.

# Scalable collections

The **Blob Storage** offered by Windows Azure, when directly tackled by the app developer, suffers from somewhat similar issues than the Queue Storage, in particular:

* No strong-typing.
* No simple way for scalable iterations and processing.
* Storage initialization is dead code.

Lokad.Cloud provides collections referred as *scalable* not because they can store a lot of data, which is already a basic feature delivered by the cloud storage itself, but because those collections support **fast parallelized processing** of their content.

## BlobSet

The BlobSet is a scalable typed collection, somehow the cloud equivalent of the HashSet generic collection. The BlobSet is based on the Blob Storage and provides basic Add / Remove / Clear methods to populate the collection, but also **map** and **reduce** operations inspired from the MapReduce algorithm (see <http://en.wikipedia.org/wiki/MapReduce> for the detail).

The code snippet below illustrates key methods available for the BlobSet. More methods are actually available, but they have been excluded here for the sake of clarity.

public class BlobSet<T> : IEnumerable<T>  
{  
 public BlobSet(string blobSetId);

public T this[BlobId blobId];  
 public BlobId Add(T item);  
 public void Remove(BlobId blobId);  
 public void Clear();

public void MapToQueue<U,M>(Func<T,U> mapper, M onCompleted); public void ReduceToQueue<U>(Func<U,U,U> reducer);  
}

The BlobSet is instantiated with an identifier and **the collection is persistent**: objects added to the collection are still present when the collection is re-instantiated anywhere else in the cloud – considering that the same identifier is used. The Blob Storage is used by the BlobSet to store the items.

Then, two methods are provided – MapToQueue and ReduceToQueue – as simple yet fast parallelized processes over the collection.

The method MapToQueue maps all the items of the collection through the **mapper** and injects results to the queue implicitly associated with the type U. Items are mapped in parallel: no order is guaranteed in the resulting queue. Once the mapping is completed, that is to say that all items of type U have successfully been queued; the message onCompleted is put into the queue implicitly associated to the type M.

The method ReduceToQueue is using a **reducer** to aggregate the whole input collection into a single item of type U. The final resulting item is put into the queue implicitly associated to type U.

In particular, the reducer must be **associative and commutative** that is to say and where is the reduction operator. Those two properties are guarantying that the reduction can be performed in any order, as the order won’t impact the final result.

Also, if the mapping represents the memory footprint associated to the item and if we assume that then the whole reduction operation can be performed in less than of I/O processing with a number proportional to the total footprint of the collection.

Lokad.Cloud assumes that which simply means that the reducer is expected to, well, reduce the amount of data. Violating this property will not cause any particular failure from Lokad.Cloud, but it will most likely hinder the actual performance of the reduction process.

**Technical note**: the underlying implementation leverages the capacity of the Blob Storage to behave like a pseudo-hierarchical file system with directories through the use of prefixes. Each BlobSet is associated to a prefix based on the blobSetId.

# Synchronization helpers

Although cloud-wide synchronization should be avoided at all cost in order to achieve truly scalable design, synchronization is very hard to avoid entirely. In particular, external resources (possible 3rd party resources) outside the cloud may require non-concurrent accesses, and may not offer any way to organize the logic into idempotent operations.

For those reasons, Lokad.Cloud offers synchronization helpers. In particular, the Lock class which is best described with the following example:

using(new Lock("mylock"))  
{  
 // no concurrent execution possible here  
}

Lokad.Cloud deals with the details of ensuring the proper synchronization through a dedicated blob for each lock identified by its name.

Then, when it comes to global identifiers, System.Guid provides a very nice fully scalable and fully distributed identifier implementation. Yet, Guid are long, unreadable and basically user hostile. For example, using Guid as account identifiers for a multitenant application can be a real pain for the end-user if this account identifier ends up in his URLs.

Thus, when performance is not such a critical issue, we propose the **Cuid – Compact Unique Identifier** that provides an alternative way of generating unique Int64 over the cloud. Also those identifiers are **nearly incremental**, thus if you don’t need that many identifiers, identifiers will stay very compact. The code snippet below illustrates the Cuid class:

public static class Cuid  
{  
 public static long Next(string counterId);  
}

The naïve implementation for Cuid would use a blob to store the current index and would require synchronized blob operation for identifier allocation. Obviously, **the naïve implementation does not scale much.** Yet, the implementation offered by Lokad.Cloud is very scalable through a simple trick: the counter is not exactly incremental but nearly incremental, that is to say, **we tolerate certain values to be skipped**. The current implementation is somehow wasting one identifier value out of two in order **to achieve a very good scalability**.

Basically, Lokad.Cloud is still using a blob to store the counter index, but the local instance of the Cuid, when called multiple times, ask for exponentially increasing identifier ranges from the blob storage. As a result, in order to allocate 220 unique identifiers (a bit more than 1 million calls), only 20 synchronous operations will be carried on the underlying blob storage.

# Logs, Monitoring and management

A cloud app needs logs, monitoring and management like any web app; but in addition, those components are expected to be scalable too. Lokad.Cloud features:

* **Scalable logs**, based on Blob Storage, designed for fast writing, but also for narrow log investigation when date ranges are provided.
* **Monitoring**, that estimates of the remaining workload lying in each queue, but that also gathers sample data such as average message processing latency and overall queue latency.
* **Management**, to start or stop cloud services.

Monitoring and management are exposed to the application administrator though a dedicated WebRole bundled with the Lokad.Cloud package.

## Scalable logs

(Draft) The framework provides a plain logger using the Blob Storage for persistence. The logger is basically following the spirit of the ELMAH (<http://code.google.com/p/elmah/>) framework, but tuned for the cloud.

In particular, write operations are made concurrently through the following naming pattern for each entry:

/container/yyyy/MM/dd/hh/mm/ss/cuid/message-title

where yyyy-MM-dd hh:mm:ss is the date of the logged message. This naming convention allows fast retrieval of specific date range through the use of prefixes while querying Blob Storage.

## Monitoring

(Draft) Lokad.Cloud essentially relies on QueueServices to balance the workload between Windows Azure workers. As a result, monitoring the execution of an app build on top of Lokad.Cloud essentially means monitoring the queues with:

* The average processing time per message.
* The number of message currently in queue.
* The overall queue latency.
* …

All those indicators are reported directly through the WebRole that comes bundled with Lokad.Cloud.

## Management

(Draft) The Lokad.Cloud services web console is an administration utility that let you start or stop individual cloud services in your app. In particular, service management can be handy for migration purposes or intermittent interactions with 3rd party services outside the cloud.

The service administration console also features prioritization settings for the services.