Multi-application multi-layer configuration in an AWS environment

Victor S. Denisov

Abstract – **When designing and implementing complex multi-application systems for cloud environments, developers often run into issues with managing run-time configuration for each application in different deployments (such as development, staging and production), as well as configuration information common between deployments and applications. This paper proposes one approach to solving this problem for Java applications running in Amazon Web Services cloud platform.**

Keywords — **application configuration management, application settings management, Java, cloud, options-util, Amazon Web Services.**

I. Introduction

Consider a typical data processing system running on Amazon's cloud infrastructure. More likely than not, this system involves at least the following chain of applications:

- Set of ingress applications, which actively (via polling) or passively (via pushing/streaming) consume, filter and transform incoming data, making it available to applications further down the chain. This can take form of a servlet container [1] cluster running behind an Elastic Load Balancer [2] receiving batches of tweets via DataSift Push API [3], or a set of GELF-compatible [4] servers receiving event streams from your other applications. Data captured by ingress applications is put into some sort of persistent or ephemeral storage, such as a database or a distributed queue.
- Set of data processing applications, which process data captured by ingress applications and extract business value from it (like, for example, measuring consumers' sentiment towards a brand from a raw stream of tweets mentioning the brand). The resulting information is stored in a persistent storage for later use and analysis by business applications.
- Set of business intelligence/business analytics/decision support applications, which help business users to act upon information extracted from incoming data.

This typical application stack can be further supplemented by various support applications, such as management/monitoring systems, centralized logging systems, etc.

This leads to a problem of managing runtime configuration information for all the applications involved in a consistent, transparent and maintainable manner. The problem is made significantly more complex by several important considerations:

- often, applications have to run in different deployments (such as development, staging and production), with certain configuration settings common to some deployments, but not others;
- in larger deployments, different application versions (with different configuration information sets) have to coexist for reasonably long periods of time;
- configuration information should be, to the extent possible, separate from application code, for both security and practical issues (re-deploying most applications takes significantly longer than simply restarting them to pick up changes in configuration – even if they don't handle such changes on-the-fly);
- using local config files is impractical if Auto-Scaling Group [5] handles instance creation/termination (which is extremely common for larger deployments); additionally, not all cloud computing services support a notion of a filesystem at all (AWS Lambda [6] being a good example), which makes using local files flat out impossible.

II. Application Model

To design a library solving the above-mentioned problem, it would be helpful to define our application model in greater detail (illustrated on Fig. 1 below):

- there is a set of application groups – deployments – reflecting specific operating environment (such as cloud or on-premises) and/or specific stage in the system's lifecycle (development, staging, etc);
- each deployment consists of one or more applications, and each application can have one or more versions running concurrently in the deployment at any point in time;
- each application runs on one or more AWS EC2 instances (and, perhaps, on other AWS computing services and on local development workstations).

Victor S. Denisov is with the Lomonosov Moscow State University (e-mail: vdenisov@plukh.org).

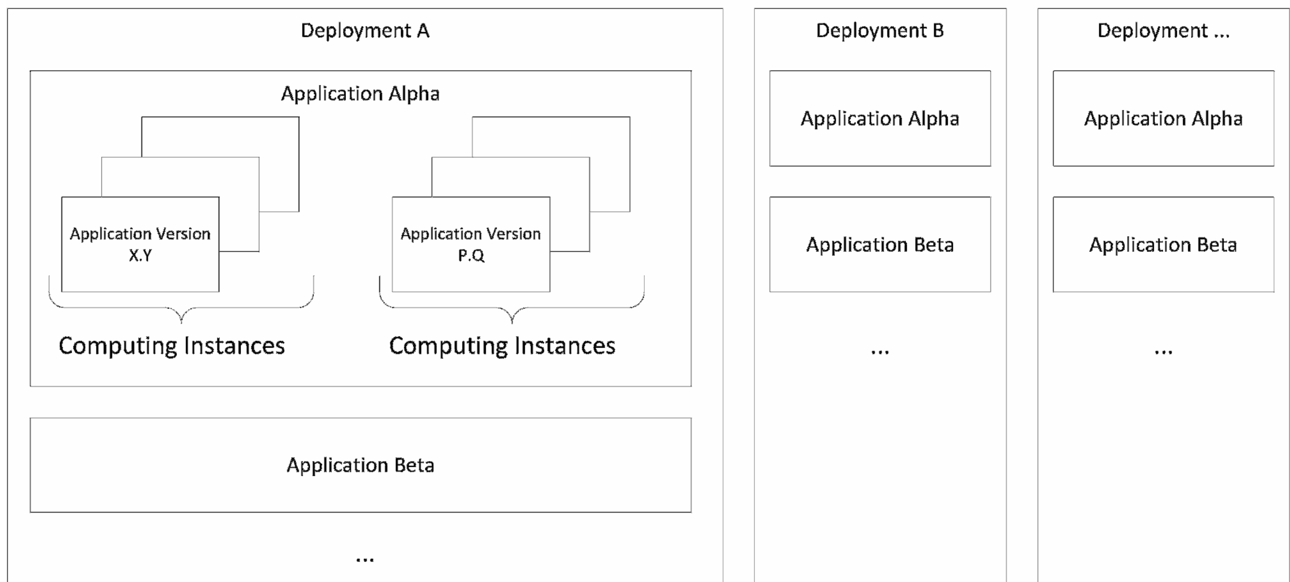

Fig. 1. Application Model

To match this model, several layers of configuration information should be supported when configuring an application:

- global configuration information (common for all deployments and applications);
- deployment configuration information (common for all applications within a deployment);
- application configuration information;
- configuration information for specific application version.

Note that this particular model does not support instance-specific configuration information. This is by design – for most cloud services, there is either no notion of a tangible instance at all, or instances are being constantly recycled by instance governors, such as the one managing AWS auto-scaling groups. However, the model can be easily extended to support instance-specific configuration, if it is absolutely required by the system.

III. Proposed Solution

It is proposed to develop the configuration library with the following primary design points in mind:

- should use Amazon S3 service [7] as a central backing store for all configuration information;
- should extend `java.util.Properties` to keep familiar and simple interface and file format;
- should store configuration information in a set of files and directories inside a single S3 bucket, to simplify upload and access control;
- should automatically build a layered configuration, with configuration options in more specific files overriding same options from less specific configuration files[1];
- should, to the extent possible, automatically determine identifying information about deployment, application and application version; should provide extension points to allow for different information providers.

IV. Implementation Details

Following the above requirements, a simple Java library, named *S3Properties* – after its primary class – was developed as an open-source project under the Apache License, Version 2.0 (ASLv2, [8]), available at https://github.com/options-util/s3properties. It was originally developed as a cloud module prototype for *options-util* configuration framework ([9], [10]), but turned out to be quite useful without *options-util* support as well.

The library assumes that each application can provide three main identifying elements:

- deployment id;
- application id;
- version id;

By default, *S3Properties* will try to provide two out of three values automatically:

- read deployment id from *deploymentId* tag on the EC2 instance of the application;
- read version id from *Implementation-Version* property in the application's `MANIFEST.MF` file.

1 For example, if the same option named *my.option* is defined in both deployment config file and application config file, value from application config should be used; similarly, an option from application config file will be overridden by version-specific config file option.

Of course, application can supply its own classes to provide this information (more on that below), and it is always responsible for providing its own name to the library.

Using the library is straightforward. First, application must obtain an instance of `org.plukh.s3properties.config.S3Properties` class. In the simplest case, no parameters have to be supplied at all, library will substitute reasonable defaults:

```
S3Properties props = new S3Properties();
```

More parameters can be specified in constructor, such as a set of default properties or a custom S3 bucket name. Alternatively, S3Properties can be configured via fluent interface methods:

```
S3Properties props = new S3Properties()
      .withBucket("my.bucket")
      .withDefaultProperties(defaults)
      .withGlobalFileName("my.global.properties");
```

If default implementations for instance and version information providers have to be replaced, they can either be configured programmatically:

```
S3Properties props = new S3Properties()
      .withInstanceInfoProvider(
         new DummyInstanceInfoProviderImpl())
      .withVersionInfoProvider(
         new DummyVersionInstanceProvider());
```

or via Java system properties – for example, at application startup via the command-line switch:

```
java -Dorg.plukh.s3properties.version.
versionInfoProviderClass=org.plukh.s3properties.
config.DummyVersionInstanceProvider -jar
myapp.jar
```

Loading properties is also straightforward:

```
//Load properties for application "ingress-app"
props.load("ingress-app");
```

Optionally, a local file name, or an `InputStream` can be passed to load a local properties instance, which is useful when running on a development workstation. If properties can be read successfully from a local file or from a stream, S3 isn't accessed at all, which allows to run a local copy of the application without Internet access:

```
//Try to load local properties first
//If successfull, don't load S3 properties
props.load("ingress-app",
   "ingress-app-local.properties");
```

After `S3Properties` have been instantiated and loaded, application can use them just like any other `Properties` instance. Note that, to keep the layered configuration contract intact, standard `Properties#load()` methods will throw an exception when accessed.

On the AWS S3 side of things, the following file structure has to be suppported:

- all configuration information is stored inside a common bucket, which by default should be named "XXX-config", where XXX is the current AWS account number[2];
- at the top level of a bucket, a file with global properties can be created (by default, it is aptly named "global.properties"); this file will be loaded by all applications in all deployments;
- also at the top level, a directory for each deployment id (i.e., "staging", "production", etc) can be created;
- inside deployment directory, a deployment properties file (named "<deploymentId>.properties") can be created; this file will be loaded for all applications running in this specific deployment;
- also inside deployment directory, directories for all application names can be created (i.e., "ingress-app", "processing-app", etc);
- inside application directory, a file named "<applicationName>.properties" (i.e., "ingress-app.properties") can be created; this file will be loaded for all application versions of this application;
- finally, a file named "<applicationName>-<versionId>.properties" (i.e., "ingress-app-1.0.properties") can be created inside the application directory; this will be loaded for a specific version of a specific application.

All files and directories are optional, but the library expects to find at least one valid configuration file along the configuration file hierarchy. Global properties file name can be overridden, but deployment and application file naming scheme is fixed.

S3Properties is available from Maven Central, at the following coordinates:

```
groupId   : org.plukh
artifactId: s3properties
version   : 1.0
```

V. CONCLUSION

S3Properties library provides a simple but powerful interface to a multi-application multi-layer configuration information for applications running in the Amazon Web Services environment. The library has reasonable coverage via unit tests and has reached stable 1.0 status, so it can be (and is) used in actual production environments. This library will serve as a solid foundation for any future work focused on managing run-time configuration of cloud-based applications.

VI. FUTURE WORK

An obvious extension would be to integrate this library as an optional module for *options-util* configuration

2 S3 bucket names are globally unique, so the static default name, like simply "config", can't be used in this case.

management framework. This involves implementing a `PersistenceProvider` to handle configurations options loading (and, possibly, saving as well), and making it available to user applications. This work is well underway.

Additionally, current rigid reliance on file format (`.properties`) and file provider (Amazon S3) can be replaced with more generic interfaces, allowing to implement support for additional file formats and configuration sources, with or without *options-util* support.

REFERENCES

[1] Web container [Online]. Available: https://en.wikipedia.org/wiki/Web_container

[2] Elastic Load Balancing [Online]. Available: https://aws.amazon.com/elasticloadbalancing/

[3] Push Delivery [Online]. Available: http://dev.datasift.com/docs/deliver/push

[4] Graylog Extended Log Format [Online]. Available: https://www.graylog.org/resources/gelf/

[5] Auto Scaling [Online]. Available: https://aws.amazon.com/autoscaling/

[6] AWS Lambda [Online]. Available: https://aws.amazon.com/lambda/

[7] Amazon Simple Storage Service (Amazon S3) [Online]. Available: https://aws.amazon.com/s3/

[8] Apache License Version 2.0 [Online]. Available: http://www.apache.org/licenses/LICENSE-2.0

[9] V. Denisov. Annotations-driven configuration framework for Java applications. International Journal of Open Information Technologies ISSN: 2307-8162 3(10), 2015. Available: http://injoit.org/index.php/j1/article/view/237

[10] Options-util: annotation-based Java configuration helper library. Available: https://github.com/options-util/options-util