

The Versatile Plugin System

by Frank Zhang on Apr 4, 2015

(/) (/) (/) (/) (/) (/) |
 Share (https://www.addtoany.com/share_save?url=http%3A%2F%2Fzstack.org%2Fblog%2Fplugin.html&title=ZStack%20-%20The%20Versatile%20Plugin%20System&description=ZStack%20%3A%20ZStack%20is%20open%20source%20IaaS%20software%20managing%20cloud%20resources%20with%20a%20plugin%20system&from=www.addtoany.com&type=button&style=button&size=small&color=blue&text=Share%20on%20AddToAny.com)

Current IaaS software are more like cloud controller software that still lack lots of features to be complete cloud solutions. As an evolving technology, predicting what features will make up a complete solution is hard, so an IaaS software cannot build all features it will need from the very beginning. Given those facts, the architecture of an IaaS software must be able to keep core orchestration stable from adding new features. ZStack, with a versatile plugin system, allows features to be implemented as plugins(both in-process or out-of-process) which can not only extend ZStack's functionality, but also hook into business logic to change default behaviours.

The motivation

eBay's Chief Engineer in charge of the OpenStack private cloud, Subbu Allamaraju says:

However, OpenStack is a cloud controller software. Though the community did a nice job at putting together this software, an instance of an OpenStack installation does not make a cloud. As an operator you will be dealing with many additional activities not all of which users see. These include infra onboarding, bootstrapping, remediation, config management, patching, packaging, upgrades, high availability, monitoring, metrics, user support, capacity forecasting and management, billing or chargeback, reclamation, security, firewalls, DNS, integration with other internal infrastructure and tools, and on and on and on. These activities are bound to consume a significant amount of time and effort. OpenStack gives some very key ingredients to build a cloud, but it is not cloud in a box.

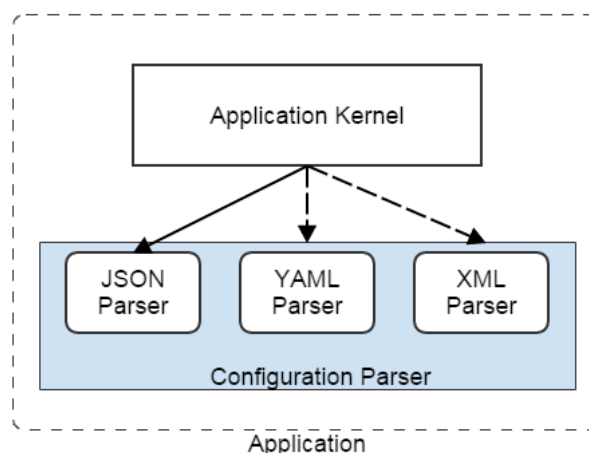
This speaks out the situation of current IaaS software that except some well-established IaaS(e.g. AWS), most IaaS software(including our ZStack) are still not complete cloud solution. As pioneers like Amazon that have been exploring for years, the public cloud has a more mature model of what a public cloud solution should look like. As still in the stage of developing, the private cloud has not had a proven model of a complete solution. Unlike proprietary public cloud software that can be specifically made for vendor's infrastructure and services, open source IaaS software have to give consideration to requirements of both public and private cloud, which makes building a complete solution much harder. As we cannot predict what a complete solution looks like, the only way for us is to provide a pluggable architecture that can easily add new features without impacting the stability of core orchestration.

The problem

Most software claims themselves as pluggable, but most of them are not really pluggable or at least not fully pluggable. Before demonstrating the reason, let's see two major forms of pluggable architecture; though there are tons of articles discussing this topic, from our experiences, we conclude all plugins into two forms that can be precisely described by two famous GoF design patterns (http://en.wikipedia.org/wiki/Design_Patterns): Strategy Pattern (http://en.wikipedia.org/wiki/Strategy_pattern) and Observer Pattern (http://en.wikipedia.org/wiki/Observer_pattern).

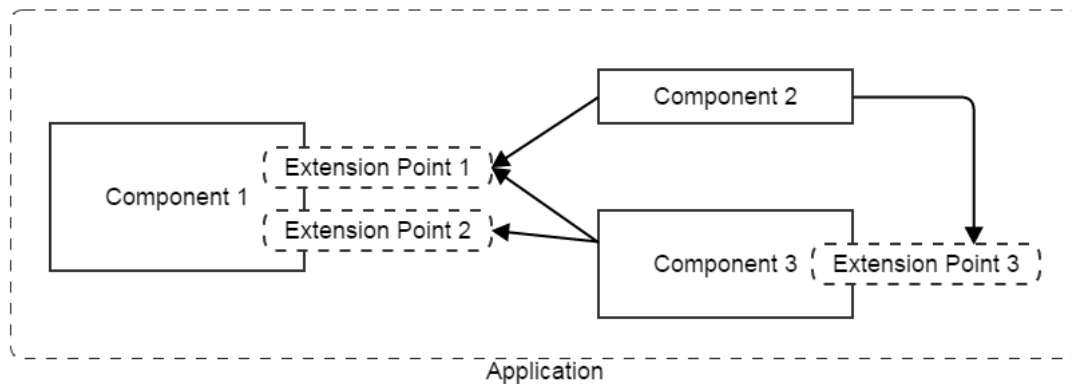
Plugins originated from strategy pattern

Plugins of this form usually extend specific functionality of software by providing different implementations, or add new functionality to the software by filling narrow plugin APIs. Lots of software we are familiar with are constructed in this pattern, for example, drivers of operating system, extensions of web browsers. This form of plugin works in the way that **allows the application to access plugins by a well-defined protocol**.



Plugins originated from observer pattern

Plugins of this form usually hook into application's business logic for specific events; once an event happens, plugins hooking on it will be called to execute a piece of code that can even change the execution flow, for example, throwing an error to stop the execution flow when the event meets some condition. Plugins built on this pattern are normally transparent to end user but purely internal implementation, for example, a listener listening on database insert event. This form of plugin works in the way that **allows plugins to access the application by well-defined extension points.**



Most software claimed as pluggable either implement one of those forms or have part of code implementing those forms. To be fully pluggable, the software must be conceived with the idea that all business logic should be implemented using those two forms, which means the entire software is made up of a number of small plugins, just like the Lego toys.

The Plugin System

An important design principle runs through all ZStack components: **every component should be designed with minimum knowledge, be self-contained, and be ignorance to other components.** For example, to create a VM, it seems that allocating volume, applying DHCP, setting up SNAT are all necessary steps that the component in charge of creating VM should know; but does it really need to know that much? Why can't this component be as simple as allocating VM's CPU/memory then sending startup request to the host, but letting other components like storage, network take care of their own businesses? You may have guessed the answer: no, in ZStack, the component doesn't need to know that much, and yes, it can be that simple. We fully realize the fact that **the more your components know, the tighter your application will be, and finally you end up building a cumbersome software that is hard to modify.** So we offer below forms of plugin to keep our architecture loosely coupled while allowing us to add features easily in order to be a complete cloud solution.

1. Strategy pattern plugin

The most general plugins in IaaS software are drivers that integrate different physical resources; for example, NFS primary storage, ISCSI primary storage, L2 network based on VLAN, L2 network based on Open vSwitch (<http://openvswitch.org/>); those plugins are in the form of strategy pattern we have stated above. ZStack has abstracted cloud resources to hypervisors, primary storage, backup storage, L2 networks, L3 networks and so on. Each resource has a reference driver that is an individual plugin. To add a new driver, developers only need to implement three components: a type, a factory, and a concrete resource implementation, all of which are enclosed in a single plugin that is usually built as a JAR file. Citing Open vSwitch as an example, let's say we will create a new L2 network that uses Open vSwitch as backend, and then the developer needs to:

1.1 define an L2 network type of Open vSwitch which will automatically register itself into ZStack's type system of L2 network.

```
public static L2NetworkType type = new L2NetworkType("Openvswitch");

/* once the type is declared as above, there will be a new L2 network type called 'Openvswitch' that can be retrieved by API */
```

1.2 create an L2 network factory that is responsible for returning a concrete implementation to L2 network service.

```

public class OpenvswitchL2NetworkFactory implements L2NetworkFactory {
    @Override
    public L2NetworkType getType() {
        /* return type defined in 1.1 */
        return type;
    }

    @Override
    public L2NetworkInventory createL2Network(L2NetworkVO vo, APICreateL2NetworkMsg msg) {
        /*
         * new resource will normally have own creational API APICreateOpenvswitchL2NetworkMsg that
         * usually inherits APICreateL2NetworkMsg, and own database object OpenvswitchL2NetworkVO that
         * usually inherits L2NetworkVO, and a java bean OpenvswitchL2NetworkInventory that usually inherits
         * L2NetworkInventory representing all properties of Openvswitch L2 network.
         */

        APICreateOpenvswitchL2NetworkMsg cmsg = (APICreateOpenvswitchL2NetworkMsg)APICreateL2NetworkMsg;
        OpenvswitchL2NetworkVO cvo = new OpenvswitchL2NetworkVO(vo);
        evaluate_OpenvswitchL2NetworkVO_with_parameters_in_API(cvo, cmsg);
        save_to_database(cvo);
        return OpenvswitchL2NetworkInventory.valueOf(cvo);
    }

    @Override
    public L2Network getL2Network(L2NetworkVO vo) {
        /* return the concrete implementation defined in 1.3 */
        return new OpenvswitchL2Network(vo);
    }
}

```

1.3 create a concrete implementation of Open vSwitch L2 network that will talk to the backend Open vSwitch controller.

```

public class OpenvswitchL2Network extends L2NoVlanNetwork {
    public OpenvswitchL2Network(L2NetworkVO self) {
        super(self);
    }

    @Override
    public void handleMessage(Message msg) {
        /* handle Openvswitch L2 network specific messages(both API and non API) and delegate
         * others to the base class L2NoVlanNetwork; so the implementation can focus on own business
         * logic and let the base class handle things like attaching cluster, detaching cluster;
         * of course, the implementation can override any message handler if it wants, for example,
         * override L2NetworkDeletionMsg to do some cleanup work before being deleted.
         */
        if (msg instanceof OpenvswitchL2NetworkSpecificMsg1) {
            handle((OpenvswitchL2NetworkSpecificMsg1)msg);
        } else if (msg instanceof OpenvswitchL2NetworkSpecificMsg2) {
            handle((OpenvswitchL2NetworkSpecificMsg2)msg);
        } else {
            super.handleMessage(msg);
        }
    }
}

```

Putting these three components together into a Maven (<http://maven.apache.org/>) module with necessary Spring (<https://spring.io/>) configuration file and compiling it to a JAR file, you create a new type of L2 network for ZStack. All ZStack resource drivers are implemented in this way(type, factory, and implementation); once you learned how to create a driver for one resource, you learned for all resources. And as we have mentioned in The In-Process Microservices Architecture ([microservices.html](#)), the driver can have own API and configuration methods.

2. Observer pattern plugin

The strategy pattern plugins(drivers) allow you to extend existing ZStack functionality; however, to make the architecture loosely coupled, plugins must be able to hook into the application's business logic and even other plugins' business logic; the key of this kind of observer pattern plugin is *extension point* that allows a piece of code from plugins to be called during a code flow is executing. Current ZStack defines about 100 extension points, exposing plenty of spots that plugins can hook to receive an event or to alter the behavior of a code flow. Creating a new extension point is nothing but defining a JAVA interface, components can easily create extension points to allow other components to hook into its business logic. To see how it works, let's continue our Open vSwitch example; assuming the Open vSwitch L2 network needs to hook into VM's creational process to prepare GRE tunnel before the VM is created, the plugin can implement `PreVmInstantiateResourceExtensionPoint` :

```

public class OpenvswitchL2NetworkCreateGRETunnel implements PreVmInstantiateResourceExtensionPoint {
    @Override
    public void preBeforeInstantiateVmResource(VmInstanceSpec spec) throws VmInstantiateResourceException {
        /*
         * you can do some check here; if any condition makes you think the VM should not be created/started,
         * you can throw VmInstantiateResourceException to stop it
         */
    }

    @Override
    public void preInstantiateVmResource(VmInstanceSpec spec, Completion completion) {
        /* create the GRE tunnel, you can get all necessary information about the VM from VmInstanceSpec */
        completion.success();
    }

    @Override
    public void preReleaseVmResource(VmInstanceSpec spec, Completion completion) {
        /*
         * in case VM fails to create/start for some reason, for cleanup, you can delete the prior created GRE tunnel here
         */
        completion.success();
    }
}

```

and when ZStack connects to a KVM host, the Open vSwitch L2 network wants to check and bring up the Open vSwitch daemon on the host, then it can implement `KVMHostConnectExtensionPoint`:

```

public class OpenvswitchL2NetworkKVMHostConnectedExtension implements KVMHostConnectExtensionPoint {
    @Override
    public void kvmHostConnected(KVMHostConnectedContext context) throws KVMHostConnectException {
        /*
         * you can use various methods like SSH Login, HTTP call to KVM agent to check the Openvswitch daemon
         * on the host, using information in KVMHostConnectedContext. If any condition makes you think the
         * host cannot provide Openvswitch L2 network function, you can throw KVMHostConnectExtensionPoint to
         * stop the host from being connected.
         */
    }
}

```

Finally, you need to advertise that you have two components implementing those extension points, ZStack's plugin system will ensure owners call your components at a proper time. The advertisement is done in plugin's Spring configuration file:

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx" xmlns:zstack="http://zstack.org/schema/zstack"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
        http://zstack.org/schema/zstack
        http://zstack.org/schema/zstack/plugin.xsd"
    default-init-method="init" default-destroy-method="destroy">

    <bean id="OpenvswitchL2NetworkCreateGRETunnel" class="org.zstack.network.l2.ovs.OpenvswitchL2NetworkCreateGRETunnel">
        <zstack:plugin>
            <zstack:extension interface="org.zstack.header.vm.PreVmInstantiateResourceExtensionPoint" />
        </zstack:plugin>
    </bean>

    <bean id="OpenvswitchL2NetworkKVMHostConnectedExtension"
        class="org.zstack.network.l2.ovs.OpenvswitchL2NetworkKVMHostConnectedExtension">
        <zstack:plugin>
            <zstack:extension interface="org.zstack.kvm.KVMHostConnectExtensionPoint" />
        </zstack:plugin>
    </bean>

</beans>

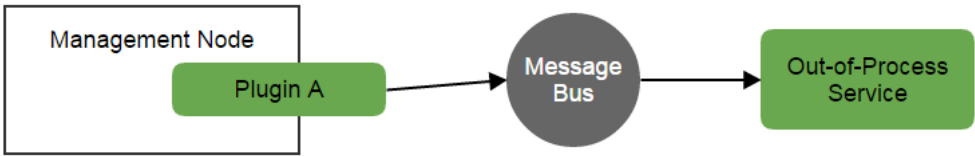
```

That's all you need. The new type of L2 network is created without even one line code change to any other ZStack components. It is the basis that ZStack keeps its core orchestration stable.

No OSGI: People familiar with Eclipse (<https://eclipse.org/home/index.php>) and OSGI (<http://www.osgi.org/Main/HomePage>) may have noticed our plugin system is very similar to what Eclipse has and what OSGI offers. Someone may ask why we don't directly use OSGI as it's made for building plugin system for Java applications. Actually we did spend decent time trying out OSGI; however, our feeling is it's overkill. We don't like another container in our application, don't like separate class loaders, and don't like the complexity of making a plugin. It seems OSGI put lots of efforts to make plugins isolated, but ZStack wants to make plugins flat. We have seen that many projects introduce unnecessary restrictions in code to make the overall architecture apparently layered, isolated, but because of the poorly designed interfaces, plugins have to write a lot of ugly codes to overcome those restrictions, which messes up the real architecture instead. ZStack considers all plugins, which have equal privileges to the core orchestration, as part of its kernel. We are not building a consumer application like browsers that users may mistakenly install malicious plugins; we are building an enterprise software that every corner needs to be rigorously tested. A flat plugin system makes our code simple and healthy.

3. The out-of-process service (plugin)

Besides above two forms, developers do have the third way to extend ZStack -- the out-of-process service. Though ZStack encompasses its orchestration services into a single process, functions that are independent to orchestration services can be implemented as individual services that run in separate processes even in separate machines. The ZStack web UI, a Python application that communicates to ZStack orchestration services through RabbitMQ, is a good example. ZStack has a well-defined message specification, out-of-process services can be written in any languages as long as they can communicate through RabbitMQ. ZStack also has a mechanism called `canonical event` that exposes internal events like VM created, VM stopped, volume created and so on to the message bus, software like billing system can be totally built as an out-of-process service by listening those events. If a service wants to be out-of-process but still needs to access some core orchestration data structures that haven't been disclosed or needs to access database, it can take a hybrid form that a small piece of plugin in the management node is responsible for collecting data and sending them over the message bus, and the out-of-process service receives those data and carries out its business.



Summary

In this article, we demonstrated ZStack's plugin architecture. Though ZStack has not been a complete cloud solution yet, it offers the architecture that can build all future needed features into plugins (both in-process and out-of-process), which provides the potential for rapidly evolving into a mature, complete cloud solution, while keeping the core orchestration robust.

0 Comments

zstack.org

1 Login

Recommend

Share

Sort by Best

Start the discussion...

Be the first to comment.

Community

Mailing List (<https://groups.google.com/forum/#!forum/zstack>)
Community (<http://www.zstack.org/community>)
Gitter


Resources

Intallation (<http://www.zstack.org/installation>)
Tutorials (<http://www.zstack.org/tutorials>)
Blog (<http://www.zstack.org/blog>)
Documentation (<http://www.zstack.org/documentation>)

Connect Us

 (https://twitter.com/zstack_org)  (<https://www.facebook.com/zstackorg>)  (<https://github.com/zstackorg/zstack>)
 ([../misc/wechat.html](https://misc/wechat.html))  (<http://weibo.com/zstack>)

ZStack is open source IaaS software provided under the Apache 2.0 license.

Your feedback is invaluable, please let us know your thoughts.  (<mailto:info@zstack.org>)