ZStack's Scalability Secrets Part 1: Asynchronous Architecture

by Frank Zhang on Apr 4, 2015

ZStack offers an architecture that 99% tasks are executed asynchronously, which is a key design that a single management node can manage hundreds of thousands of physical servers, millions of virtual machines, and handle tens of thousands of concurrent tasks.

The motivation

For public clouds that usually manage massive hardware and virtual machines, scalability is one of the key problems that IaaS software need to solve. For a medium-sized data center, which may have 50,000 physical servers, may run about 1,500,000 virtual machines that belong to, for example, 10000 users; though it's unlikely that users will start/stop their virtual machines as frequent as they update Facebook pages, it's still possible that the IaaS system is crowd of thousands of tasks either from API or internal components at any given moment. In some worse case, for example, a user may wait for 1 hour for his or her new virtual machine to be created, just because the system is stuck with other 5000 tasks and the poor thread pool has only 1000 threads that have been busy for a while.

The problem

First of all, we'd like to show our opposition to an opinion about IaaS scalability issue we have seen in some articles, which claim **"the supporting infrastructure, especially databases and message brokers are the culprits of poor IaaS scalability"**. No, that is wrong! First, for the database, the data size of IaaS software is rather small or medium at most; web giants like Facebook and Twitter, are still using MySQL as their main database; can an IaaS's data size beat Facebook's or Twitter's? No way, they are at the billions level, IaaS are at the millions level (super large data center). Second, for the message broker, the RabbitMQ which ZStack uses is a moderate scalable message broker, compared to others like Apache Kafka or ZeroMQ; but it can still sustain average 50,000 messages per second(see RabbitMQ Performance Measurements, part 2 (http://www.rabbitmq.com/blog/2012/04/25/rabbitmq-performance-measurements-part-2/)). Isn't that good enough for communications inside IaaS software? That is enough.

The cause of IaaS scalability issue is: **slow tasks**. Yes, tasks in IaaS software are very slow, they usually take a couple of seconds even a few minutes to finish; so when the system is full of slow tasks, it's unsurprised that new tasks get a huge delay before getting served. Slow tasks are originated from long task path; for instance, to create a virtual machine, it normally needs to go through identity service --> scheduler --> image service --> storage service --> network service --> hypervisor; each service may take a few seconds or minutes to conduct operations on external hardware, which causes the amount of task time to be very long.

Synchronous vs. Asynchronous

Traditional IaaS software does tasks synchronously; they are usually thread pool based which offer a thread for each task and the thread can serve the next task only after the previous task is finished. As tasks are slow, when encountering a peak of concurrent tasks, the system can be ridiculously slow because the thread pool is running out of capacity, and incoming tasks are queued to be served.

To tackle the issue, the straight idea is to increase the thread pool's capacity; however, even if modern operating systems allow an application has tens of thousands of threads, no operating system can efficiently schedule them. So people start scaling things out, distributing workloads to analogous software running on different operating systems; because each software has own thread pool, it eventually increases the thread capacity of the entire system. However, the scaled out solution comes with a cost; it augments the difficulty of management, and clustering software is still a challenge in software design. Finally, the IaaS software itself becomes the bottleneck of the cloud, while the rest of infrastructures including databases, message brokers, and external systems (e.g. thousands of physical servers) still have adequate capacity to serve more concurrent tasks.

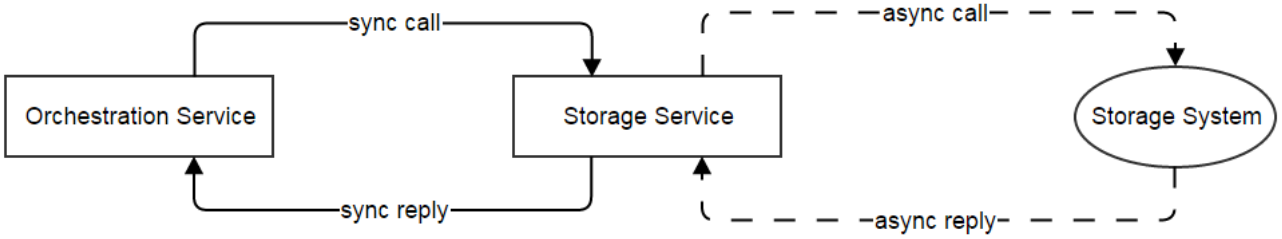
ZStack tackles this problem by an asynchronous architecture. If we look at the relationship between IaaS software and facilities of data center, the IaaS software is actually playing a mediator role; it coordinates external systems but does not do real time costly tasks; for example, volumes are created by storage system, templates are downloaded by image system, virtual machines are created by hypervisor and so on. **The real task the IaaS software does is to make decisions then distributes sub-tasks to different external systems**. For example, for KVM, many sub-tasks like preparing volume, preparing network, and creating virtual machine are carried out by KVM hosts; it may take 5 seconds to finish spawning a virtual machine; however, the actual time cost in the IaaS software may be only 0.5s and the rest of 4.5s are spent on KVM host. The truth of ZStack's asynchronous architecture is the IaaS management software doesn't need to wait that 4.5s, but only spends 0.5s choosing which host to do the task then simply delegates it to that host. Once the host completes its assignment, it notifies the IaaS management software the result. By the asynchronous architecture, a thread pool with 100 threads can easily serve thousands of concurrent tasks.

ZStack's asynchronous way

Asynchronous operations are very common in computer science; async I/O, AJAX are well-known use cases. However, to build all business logic based on asynchronous operations, especially for IaaS that is typically an integration software, there are still many challenges.

The biggest challenge is you must make all components asynchronous, not just a part of components asynchronous; for example, if you build an asynchronous storage service while other services are synchronous, the entire system gets no bonus; because when calling the storage service, even it's asynchronous, the invoking service has to wait it to complete before moving on to the next step, which makes the whole workflow still

synchronous.



The thread that orchestration service calls storage service has to block until the storage service returns, though the storage service communicates to the external storage system asynchronously

ZStack's asynchronous architecture consists of three parts: asynchronous message, asynchronous method, and asynchronous HTTP call.

1. Asynchronous message

ZStack uses RabbitMQ as message bus connecting various services. When a service calls another service, the source service sends a message to the destination service and registers a callback, then returns immediately; once the destination service finishes the task, it replies a message triggering the callback registered by source service to notify the result. For example:

```
AttachNicToVmOnHypervisorMsg amsg = new AttachNicToVmOnHypervisorMsg();
amsg.setVmUuid(self.getUuid());
amsg.setHostUuid(self.getHostUuid());
amsg.setNics(msg.getNics());
bus.makeTargetServiceIdByResourceUuid(amsg, HostConstant.SERVICE_ID, self.getHostUuid());
bus.send(amsg, new CloudBusCallBack(msg) {
    @Override
    public void run(MessageReply reply) {
        AttachNicToVmReply r = new AttachNicToVmReply();
        if (!reply.isSuccess()) {
            r.setError(errf.instantiateErrorCode(VmErrors.ATTACH_NETWORK_ERROR, r.getError()));
        }
        bus.reply(msg, r);
    }
});
```

A service can also send a list of messages to other services and wait for replies asynchronously:

```
final ImageInventory inv = ImageInventory.valueOf(ivo);
final List<DownloadImageMsg> dmsgs = CollectionUtils.transformToList(msg.getBackupStorageUuids(), new Function<DownloadImageMsg, String>() {
    @Override
    public DownloadImageMsg call(String arg) {
        DownloadImageMsg dmsg = new DownloadImageMsg(inv);
        dmsg.setBackupStorageUuid(arg);
        bus.makeTargetServiceIdByResourceUuid(dmsg, BackupStorageConstant.SERVICE_ID, arg);
        return dmsg;
    }
});

bus.send(dmsgs, new CloudBusListCallBack(msg) {
    @Override
    public void run(List<MessageReply> replies) {
        /* do something */
    }
}
```

moreover, it's even possible to send a list of messages with a certain parallelism level; for example, with a list of 10 messages, it can send 2 messages each time, which means messages 3,4 are sent only after replies to messages 1,2 are received.

```
final List<ConnectHostMsg> msgs = new ArrayList<ConnectHostMsg>(hostsToLoad.size());
for (String uuid : hostsToLoad) {
    ConnectHostMsg connectMsg = new ConnectHostMsg(uuid);
    connectMsg.setNewAdd(false);
    connectMsg.setServiceId(serviceId);
    connectMsg.setStartPingTaskOnFailure(true);
    msgs.add(connectMsg);
}

bus.send(msgs, HostGlobalConfig.HOST_LOAD_PARALLELISM_DEGREE.value(Integer.class), new CloudBusSteppingCallback() {
    @Override
    public void run(NeedReplyMessage msg, MessageReply reply) {
        /* do something */
    }
});
```

2. Asynchronous method

Services, as the first class citizen in ZStack, communicate through asynchronous messages; however, inside services, there are a bunch of components, plugins that still interface with each other using method calls, which are also asynchronous:

```
protected void startVm(final APIStartVmInstanceMsg msg, final SyncTaskChain taskChain) {
    startVm(msg, new Completion(taskChain) {
        @Override
        public void success() {
            VmInstanceInventory inv = VmInstanceInventory.valueOf(self);
            APIStartVmInstanceEvent evt = new APIStartVmInstanceEvent(msg.getId());
            evt.setInventory(inv);
            bus.publish(evt);
            taskChain.next();
        }

        @Override
        public void fail(ErrorCode errorCode) {
            APIStartVmInstanceEvent evt = new APIStartVmInstanceEvent(msg.getId());
            evt.setErrorCode(errf.instantiateErrorCode(VmErrors.START_ERROR, errorCode));
            bus.publish(evt);
            taskChain.next();
        }
    });
}
```

Still, callbacks can carry return value:

```
public void createApplianceVm(ApplianceVmSpec spec, final ReturnValueCompletion<ApplianceVmInventory> completion) {
    CreateApplianceVmJob job = new CreateApplianceVmJob();
    job.setSpec(spec);
    if (!spec.isSyncCreate()) {
      job.run(new ReturnValueCompletion<Object>(completion) {
          @Override
          public void success(Object returnValue) {
            completion.success((ApplianceVmInventory) returnValue);
          }

          @Override
          public void fail(ErrorCode errorCode) {
            completion.fail(errorCode);
          }
      });
    } else {
        jobf.execute(spec.getName(), OWNER, job, completion, ApplianceVmInventory.class);
    }
}
```

3. Asynchronous HTTP call

ZStack uses a couple of agents to manage external systems, for example, agent managing KVM host, agent managing console proxy, agent managing virtual router and the like. Those agents are all lightweight web servers built on Python CherryPy (http://www.cherrypy.org/). As it's impossible to do bidirectional communication without HTML5 technologies like web sockets, ZStack puts a callback URL in the HTTP header of every request; therefore, agents can send responses to caller's URL when tasks are done.

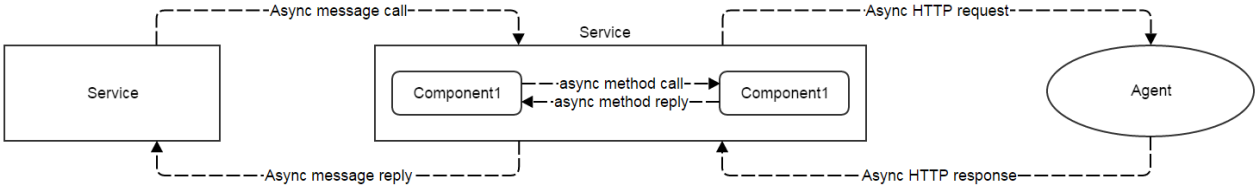
```
RefreshFirewallCmd cmd = new RefreshFirewallCmd();
List<ApplianceVmFirewallRuleTO> tos = new RuleCombiner().merge();
cmd.setRules(tos);

resf.asyncJsonPost(buildUrl(ApplianceVmConstant.REFRESH_FIREWALL_PATH), cmd, new JsonAsyncRESTCallback<RefreshFirewallRsp>(msg, completion) {
    @Override
    public void fail(ErrorCode err) {
        /* handle failures */
    }

    @Override
    public void success(RefreshFirewallRsp ret) {
        /* do something */
    }

    @Override
    public Class<RefreshFirewallRsp> getReturnClass() {
        return RefreshFirewallRsp.class;
    }
});
```

By those three methods, ZStack has built a hierarchy that all components can behave asynchronously:



Summary

In this article, we demonstrated the asynchronous architecture that ZStack relies on to solve IaaS's scalability issue caused by enormous slow concurrent tasks. In our testing, using simulator, a single ZStack management node with 1000 threads can easily handle 10,000 concurrent tasks of creating 1,000,000 virtual machines. Despite a single management node is scalable enough to deal with workloads in most clouds, for the sake of high availability (HA) or super large workload (e.g. 100,000 concurrent tasks), a setup of multiple management nodes is necessary. Please check out ZStack's stateless services in ZStack's Scalability Secrets Part 2: Stateless Services (stateless-clustering.html).

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) (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)