ZStack's Scalability Secrets Part 3: Lock-free Architecture

by Frank Zhang on Apr 4, 2015

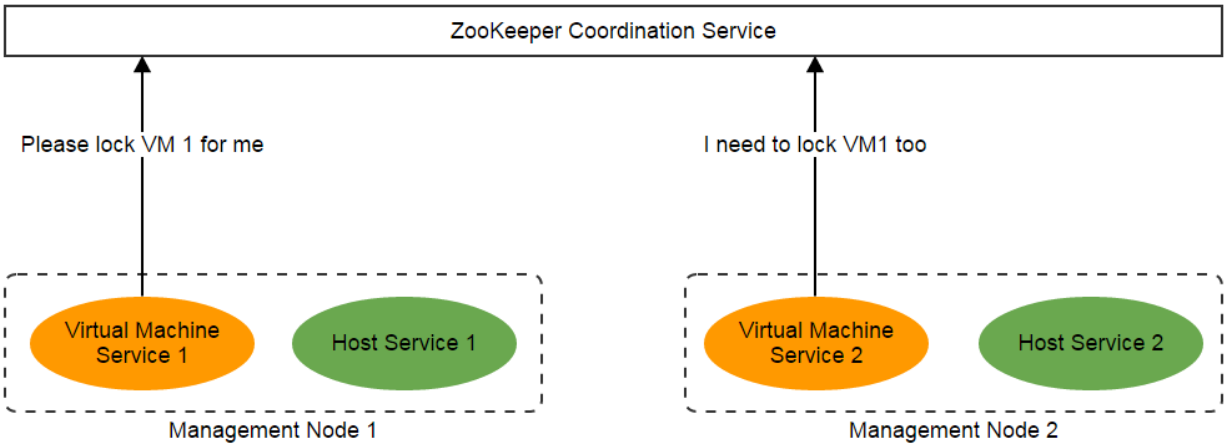
Lots of tasks in IaaS software need to execute in sequence; for example, when a task of starting a virtual machine is running, a stopping task of the same virtual machine must wait for the previous starting task to finish. On the other side, some tasks should be able to run in parallel; for example, 20 tasks of creating virtual machines on the same host can run simultaneously. Synchronization and fine-grained parallelization in a distributed system are not easy and usually require a distributed coordination software. Rising to the challenge, ZStack offers a lock-free architecture built on queues, allowing tasks to control their parallelism level easily from 1(synchronized) to N(parallel).

The motivation

A good IaaS software should have fine-grained control on synchronization and parallelization of tasks. Most of the time, tasks need to execute in certain sequence as they have dependencies to each other; for example, a task of deleting a volume can not execute if another task of taking snapshot of the same volume is still running. Sometimes, tasks should run in parallel in order to boost the performance; for example, ten tasks of creating virtual machines on the same host could run at the same time without any problem. However, parallelization can harm the system without proper control; for example, 1000 concurrent tasks of creating virtual machines on one host will undoubtedly blow it up or at least cause it long time no response. This kind of concurrent programming problem is complex in a multi-threaded environment and is more complex in a distributed system.

The problem

Textbooks tell us that lock and semaphore are answers to synchronization and parallelization; the straight idea to deal with them in a distributed system is to use some distributed coordination software like Apache ZooKeeper (http://zookeeper.apache.org/) or something similar built on Redis (http://redis.io/). A system overview of using distributed coordination software, for example, ZooKeeper, is like:

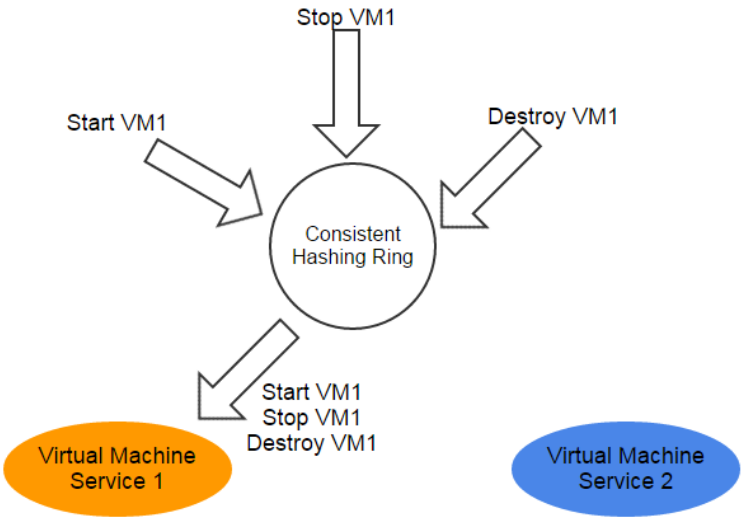


The problem here is, with lock or semaphore, **a thread needs to wait for other threads to release the locks or semaphores they have been holding**. In ZStack's Scalability Secrets Part 1: Asynchronous Architecture (asynchronous-architecture.html) we explained ZStack was an asynchronous software that no thread will be blocked to wait for others' completion; so using lock and semaphore is not a viable option. And we also concern about the complexity and scalability of using a distributed coordination software, imagining a system filled with 100,000 tasks requiring locks, that's neither easy nor scalable.

> ***Synchronous vs. Synchronized**: In ZStack's Scalability Secrets Part 1: Asynchronous Architecture (asynchronous-architecture.html), we discussed **synchronous vs. asynchronous**, in this article, we are going to discuss **synchronized vs. parallel**. Synchronous and synchronized sometimes are used interchangeably, but they are different. In our context, synchronous is talking about whether a task will block the thread while executing while synchronized is talking about whether a task must execute exclusively. If a task occupies a thread all the time until it finishes, it is a synchronous task; if a task cannot execute with other tasks at the same time, it's a synchronized task.*

The basis of lock-free architecture

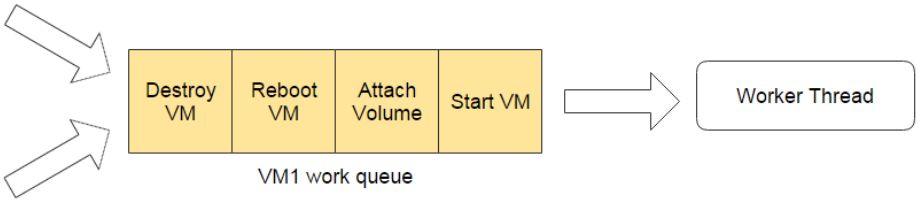
The consistent hashing algorithm, which guarantees all messages to the same resource are always handled by the same service instance, is the basis of the lock-free architecture. This way of congregating messages to the certain node reduces the complexity of synchronization and parallelization from a distributed system to a multi-threaded environment(see more details in ZStack's Scalability Secrets Part 2: Stateless Services (stateless-clustering.html))



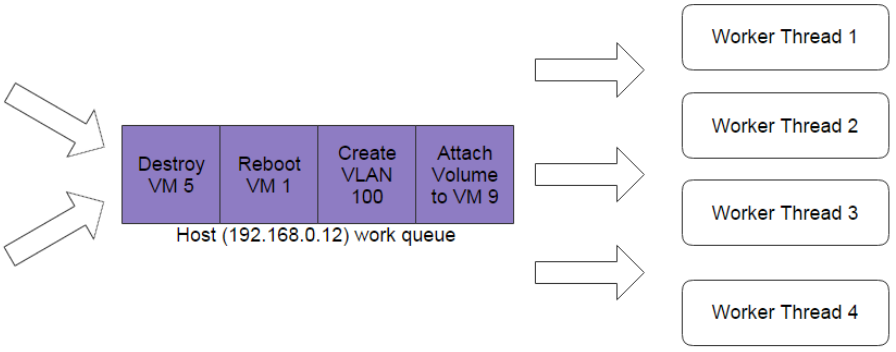
Work Queue: the traditional plain solution

> ***Note**: Before digging into details, please note the queues we are about to discuss have nothing to do with RabbitMQ message queue mentioned in ZStack's Scalability Secrets Part 2: Stateless Services (stateless-clustering.html). The message queue is a term of RabbitMQ; ZStack's queues are internal data structure.*

As tasks in ZStack are driven by messages, congregating messages also makes relevant tasks execute on the same node, relieving the challenge to a classic problem of concurrent programming with a thread pool. To avoid lock contention, ZStack uses work queue instead of lock and semaphore. Synchronized tasks execute one by one, maintained in memory based work queues:



Parallel tasks can execute in defined parallelism level, maintained in work queues with parallelism levels. The below example shows a queue with parallelism level equal to 4.



> ***Note**: A work queue can execute both synchronized and parallel task. With parallelism equal to 1, the queue is synchronized; with parallelism greater than 1, the queue is parallel; with parallelism equal to 0, the queue is unlimited parallel.*

Memory-based synchronous queue

There are two types of work queue in ZStack; one is synchronous work queue that a task(normally a Java Runnable (http://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html)) is considered as finished right after it returns:

```
thdf.syncSubmit(new SyncTask<Object>() {
    @Override
    public String getSyncSignature() {
        return "api.worker";
    }

    @Override
    public int getSyncLevel() {
        return apiWorkerNum;
    }

    @Override
    public String getName() {
        return "api.worker";
    }

    @Override
    public Object call() throws Exception {
        if (msg.getClass() == APIIsReadyToGoMsg.class) {
            handle((APIIsReadyToGoMsg) msg);
        } else {
            try {
                dispatchMessage((APIMessage) msg);
            } catch (Throwable t) {
                bus.logExceptionWithMessageDump(msg, t);
                bus.replyErrorByMessageType(msg, errf.throwableToInternalError(t));
            }
        }

        /* When method call() returns, the next task will be proceeded immediately */

        return null;
    }
});
```

> ***Emphasis**: With synchronous queues, the worker thread continues fetching next Runnable once the previous Runnable.run() returns, and will only return to thread pool until the queue is empty. Because tasks occupy the worker thread when executing, the queue is synchronous.*

Memory-based asynchronous queue

Another is asynchronous work queue that a task is considered as finished only when it issues a completion notification:

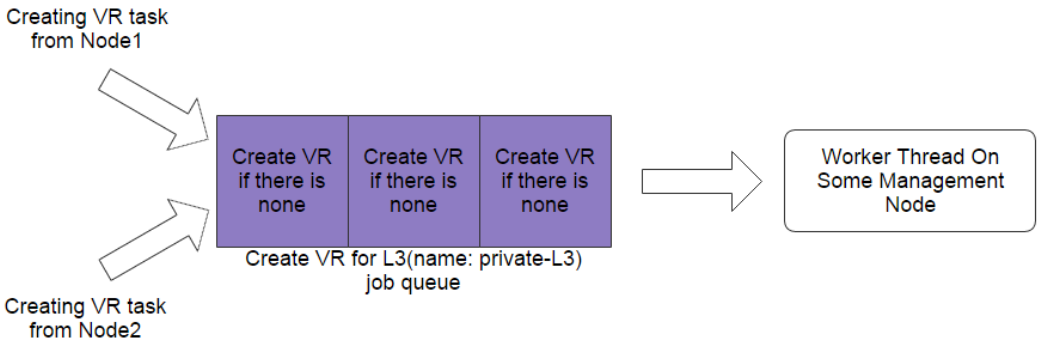
```
thdf.chainSubmit(new ChainTask(msg) {
    @Override
    public String getName() {
        return String.format("start-vm-%s", self.getUuid());
    }

    @Override
    public String getSyncSignature() {
        return syncThreadName;
    }

    @Override
    public void run(SyncTaskChain chain) {
        startVm(msg, chain);

        /* the next task will be proceeded only after startVm() method calls chain.next() */
    }
});
```

> ***Emphasis**: With asynchronous queues, ChainTask.run(SyncTaskChain chain) method may return immediately after doing some asynchronous operation; for example, sending a message with a registered callback. After the run() method returns, the worker thread gets back to the thread pool; however, the previous task is not finished yet, no task in the queue can proceed until the previous task issues a notification (e.g. calling SyncTaskChain.next()). Because tasks don't block a worker thread to wait for their completion, the queue is asynchronous.*

Database-based asynchronous queue

The memory based work queue is fast and simple, solving 99% need of synchronization and parallelization in a single management node; however, tasks related to creating resources may need to be synchronized among management nodes. The consistent hashing ring works based on resource UUID, if a resource has not been created, it has no way to work out which node should take care of the creational task. In most cases, ZStack will choose the local node the submitter of the creational task is on to carry it out if the resource to be created is not relied on by other pending tasks. Unfortunately, ongoing tasks will depend on a special resource called virtual router VM; for instance, if multiple user VMs with the same L3 network

are created by tasks running on different management nodes, and there has not been a virtual router VM on that L3 network, the task of creating a virtual router VM may be submitted by several management nodes. In this case, as it's a distributed synchronization problem, ZStack uses a database-based job queue that tasks from different management nodes can be synchronized globally.

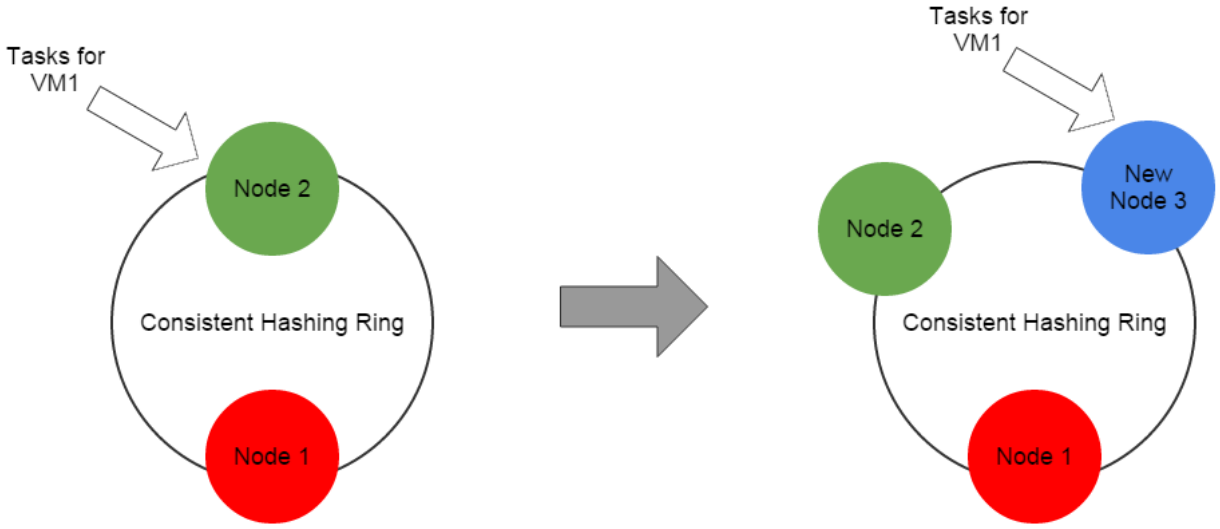


Database job queue only has the asynchronous form; that is to say, the next task executes only after the previous task issues a completion notification.

> ***Note**: As tasks are stored in the database, database job queues are slow; fortunately, the only task of creating the virtual router VM needs it.*

The limitation

Though the queue based lock-free architecture handles synchronization in 99.99% time, there is a race condition stemming from the consistent hashing algorithm: a new joining node will take a part of workload from its neighbor node as the result of expansion of the consistent hashing ring.



In this example, after the new node 3 joins, tasks previously targeted to the node 2 will go to the node 3; during a period, it can cause a race condition if an old task for a resource is still running on the node 2 but new tasks for the same resource are submitted to the node 3. However, the situation is not as bad as you may think. First, conflicting tasks rarely exist in a regular system, for example, a healthy UI should not allow you to stop a VM that is still in the process of starting. Second, every ZStack resource has state/status, a task will raise out an error if it begins with a wrong state/status; for example, if a VM is in state of stopping, a task of attaching volume will return an error immediately. Third, agents, which most tasks will be finally delivered to, have additional synchronization mechanism; for example, the virtual router agent will synchronize all requests modifying the DHCP configuration file in spite of we already have work queue for the virtual router VM at management node side. Finally, planning your operations ahead is a key to continuously managing a cloud; the operations team can spring up enough management nodes before launching the cloud; if they do need to add a new node dynamically, do it when the workload of the cloud is rather small.

Summary

In this article, we demonstrated the lock-free architecture that is built on memory-based work queue and database-based job queue. Without involving complex distributed coordination software, ZStack boosts performance as much as possible while shielding tasks from race conditions.

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)