Zend Platform™

White Paper:

Scalability and Responsiveness with Zend Platform's Job Queue

By **Dotan Perry and Shie Erlich**

June 2007

Zend
The php Company

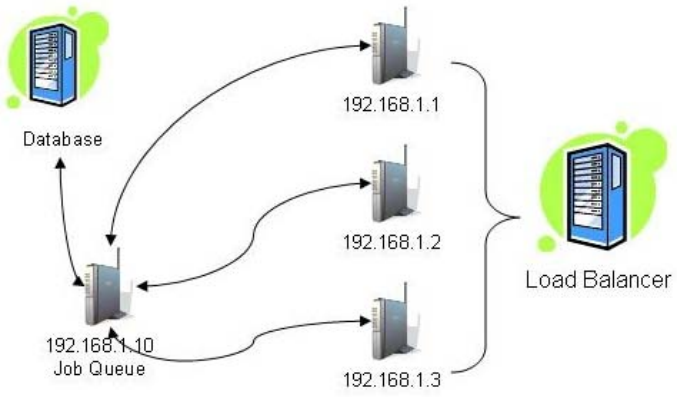
Scalability and Responsiveness with Zend Platform's Job Queue

Job Queue is used to make applications more responsive by allowing tasks to be performed on other machines (e.g.: dedicated machine), and scheduling tasks to a future time (usually to low-traffic hours). Moreover, Job Queue supports more complex application-level logic such as recurring jobs, job dependencies, job priorities, different scheduling algorithms (by priority, application etc.), easy detection and handling of failed jobs. Job Queue offers both a web-based management UI, and a programmer-oriented rich API, which allows embedding sophisticated logic into applications. This article will demonstrate basic usage of Job Queue, through an imaginary company wesellalot.com.

wesellalot.com is a large and successful site, selling all kinds of merchandise to a very large number of users. In order to support the large operation, *wesellalot.com* has a large cluster of web servers, stationed behind a high-end load balancer. The site's responsiveness is usually good, but is largely affected by the load from the checkout process, as it involves heavy database activity (verifying credit card's validity, charging, processing order, sending confirmation emails etc). During peak customer activity, the delays increase, and the site's responsiveness is at its worst.

The Simple Scenario

A common layout of cluster web servers consists of a group of machines behind a load balancer, probably communicating with a database machine, storage devices, and remote services. In this article, we'll demonstrate a simple Job Queue deployment, by installing it on a single dedicated machine in the cluster[1].



A simplified checkout process consists of the following stages:

1. Verification of the credit card using a remote credit company site
2. Order processing
3. Sending a confirmation email

Each of the steps depends on the successful completion of the previous step, and a failure in any of the steps requires sending an appropriate email to the customer.

We will optimize the process using the following methods:

1. Offloading the task to the dedicated machine, thereby reducing the load on the front end web servers
2. Postpone tasks to low-traffic hours, evenly distributing the load throughout the day

[1] If needed, Job Queue can be installed on multiple machines, which allows more fine-grained control over the way jobs are distributed.

The original *checkout.php* script looks like the following:

```
<?php
$transaction = TxManager::instance()->getTx($_GET['tx_id']);
If (!verifyCreditCard($transaction->creditCard)) {
      sendFailureEmail($transaction->error(), $transaction->customerEmail());
      gracefulExit();
}
If (!processOrder($transaction)) {
      sendFailureEmail($transaction->error(), $transaction->customerEmail());
      gracefulExit();
}
sendSuccessEmail($transaction->customerEmail());
displayOrderConfirmedPage();
?>
```

This script runs on each of the front end servers during checkout, and the customer's browser waits for the long operation end before displaying a message.

Step 1: Off Load to a Dedicated Server

The first example will simply off load the original *checkout.php* task to the dedicated machine. Instead of the original *checkout.php* script (which will now reside on the dedicated machine).

The front-end server will now run *add_checkout_job.php*, which looks like:

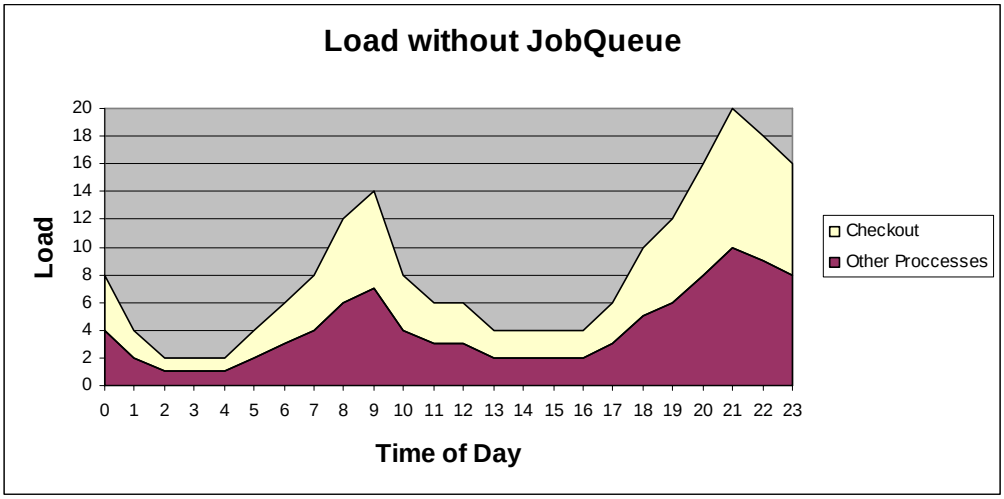
```
<?php

$queue = new ZendApi_Queue("192.168.1.10:10003");

$queue->login("queuePassword"); // log into the dedicated job queue machine,
using its password

$checkoutJob = new ZendApi_Job("checkout.php");

$id = $queue->addJob($checkoutJob);

echo "Your order has been added to the queue. The order ID is $id.
Confirmation email will be sent later.";

?>
```

This script will be very fast, and the actual *checkout.php* task will be performed asynchronously on the dedicated machine as soon as possible[2].

[2] In this example we didn't use any scheduling and priority properties for the job. By default, a job will be performed as soon as possible.

Step 2: Scheduling to low-traffic hours

Step 1 was beneficial by increasing the responsiveness of the front-end web server; however, there is still a very high load on the dedicated machine and database during peak-hours, which also hurts performance of other processes, such as adding items to a shopping cart. Various strategies can be used when scheduling tasks and processes with Job Queue, depending on the load characteristics of the system as a whole. As an example for the possible benefit of JQ, we'll take a system where 50% of the load comes from the checkout process and 50% from other processes. The system load graph exhibits a peak at 09:00-10:00 and another one at 21:00-23:00.



The effects of two possible strategies will be demonstrated:

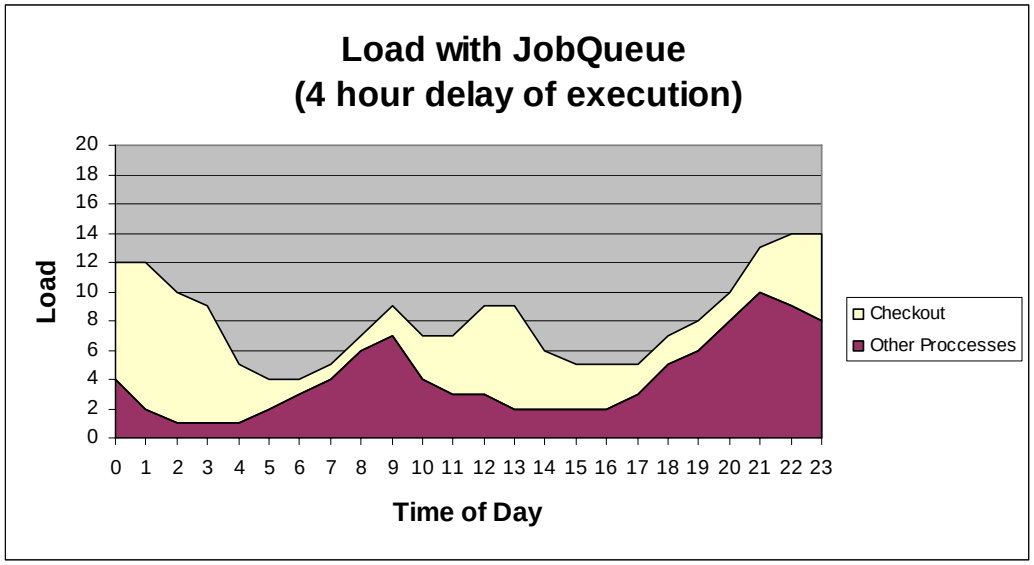
1. Delaying the execution of all checkout related tasks by 4 hours.
2. Distributing the execution of checkout related tasks randomly across the low traffic hours.

*Note that in all cases the total load on the system remains the same.

To simply delay the execution of the job by 4 hours, we would change the *add_checkout_job.php* script to look like this:

```
<?php

$queue = new ZendApi_Queue("192.168.1.10:10003");

$queue->login("queuePassword"); // log into the dedicated job queue machine,
using its password

$checkoutJob = new ZendApi_Job("checkout.php");

$checkoutJob->setScheduledTime(time() + 4*3600);

$id = $queue->addJob($checkoutJob);

echo "Your order has been added to the queue. The order ID is $id.
Confirmation email will be sent later.";

?>
```

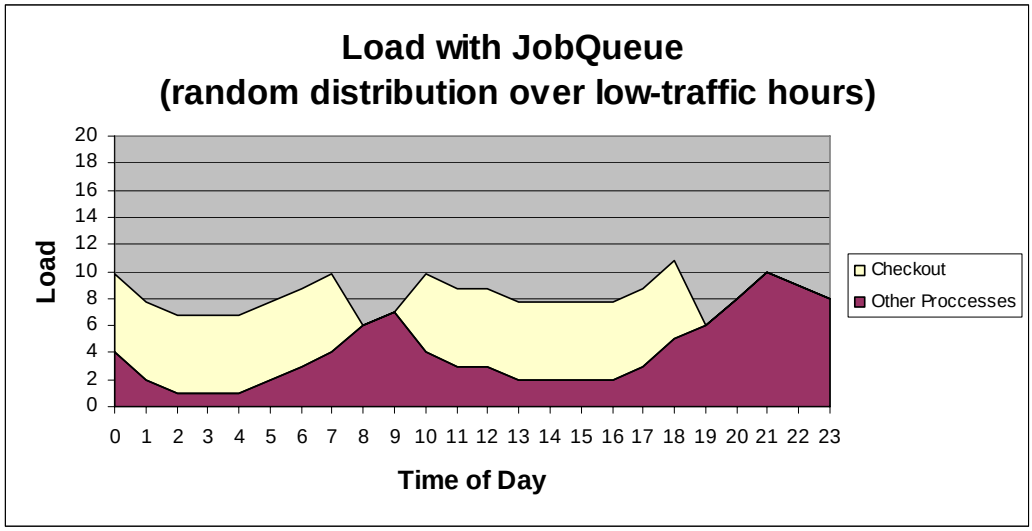
After this modification, the load graph would look like:



Notice that using this simple strategy the peak load dropped by 30% and the load standard deviation dropped by 41%.

It isn't much more difficult to implement the second strategy, and distribute tasks across low traffic hours.

The resulting graph would look like:



In this case we got a 46% reduction in peak load, while standard deviation dropped by 76%!

The lower peak loads mean better responsiveness to the user during the times when it matters the most. The lower standard deviation means resources can be fully utilized throughout the day.

Step 3: Breaking down tasks using dependencies

Step 2 accomplished the goal of moving the load to low-traffic hours; however, users usually want to know if their credit card is invalid quickly, and not the day after – which is exactly what happens when the task is scheduled for the night. In this step, we will break down the checkout process into several steps, and handle their scheduling separately. The goal is that the credit card verification will be done ASAP, while other task will still be scheduled to low-traffic hours.

We'll break down *checkout.php* into several scripts: *verify_card.php*, *process_order.php*, *send_email.php*. It's important to note that the scripts are actually dependant on one another (you can't process an order without verifying the credit card first), so we'll have to introduce the concept of dependencies.
When job B is dependent on job A it means that:

A is called B's predecessor

B will not be performed until A is completed successfully (if A fails, B will not be executed)

Job Queue supplies an API function that can indicate a logical failure and can be used to control the application flow (e.g. make the job show as failed if a connection to database failed, or credit card verification failed).

The new *add_checkout_job.php* script now looks like:

```
<?php

$queue = new ZendApi_Queue("192.168.1.10:10003");

$queue->login("queuePassword"); // log into the dedicated job queue machine,
using its password



$verifyJob = new ZendApi_Job("verify_card.php"); // will be performed ASAP

$verifyJobId = $queue->addJob($verifyJob);



// processOrderJob is both scheduled and dependent on verifyJob

$processOrderJob = new ZendApi_Job("process_order.php");

$processOrderJob->setScheduledTime($time() + 4*3600);

$processOrderJob->setJobDependency($verifyJobId);

$processOrderJobId = $queue->addJob($processOrderJob);



// sendEmailJob is dependent on processOrderJob. There's no need to schedule
it, since it can't be

// performed before processOrderJob is completed successfully

$sendEmailJob = new ZendApi_Job("send_email.php");

$sendEmailJob->setJobDependency($processOrderJobId);

$queue->addJob($sendEmailJob);

?>
```

This script allows the *verify_card.php* script to be performed ASAP, thereby allowing a quicker response to the customer, but the rest of the process is scheduled to low-traffic hour, thereby reducing the load.

Step 4: A recurring cleanup job

When the system is running for a long time, failed jobs will start to accumulate (successful jobs can be configured to be automatically cleaned-up), burdening the database. In this step, we'll introduce the concept of a recurring job, and demonstrate use of Job Queue's query abilities. Let's add a cleanup job, which runs every hour, and cleans up all failed jobs.

Here's a simple snippet of adding a recurring job:

```
<?php

$queue = new ZendApi_Queue("192.168.1.10:10003");

$queue->login("queuePassword"); // log into the dedicated job queue machine,
using its password

$cleanupJob = new ZendApi_Job("cleanup.php");

$cleanupJob->setRecurrenceData(60*60, time()); // job is performed every hour,
starting immediately

$queue->addJob($cleanupJob);

?>
```

The code of *cleanup.php* looks like:

```
<?php

$queue = new ZendApi_Queue("192.168.1.10:10003");

$queue->login("queuePassword"); // log into the dedicated job queue machine,
using its password

$jobs_status = array(status=>JOB_QUEUE_STATUS_LOGICALLY_FAILED);

$all_failed_jobs = $queue->getJobsInQueue($jobs_status);

foreach ($all_failed_jobs as $failed_job) {

      echo "removing job $failed_job from queue…";

      $queue->removeJob($failed_job);

}

?>
```

That was easy, wasn't it? Job Queue offers a rich API that allows for much more fine-grained control over the jobs in the queue, and more advanced applications like:

- Querying the status of a job that already completed, including the HTML output it produced
- Advanced queries (i.e.: jobs with status failed, high priority which originated in an accounting application)
- Allowing the job to be performed within the same context as the script inserting the job
- Different scheduling algorithms

I can do all this with *Cron*, right? Wrong!

A frequently asked question is "why is Job Queue better than simply using a cron job to schedule PHP scripts?". The main advantage of Job Queue, is that it isn't a "fire and forget" mechanism like cron, where once a job has been started, there is no way to follow up on its status, completion etc. Job Queue is a job management system, which allows doing a lot more:

- Keeping track of previous jobs, including all their attributes (output, date, errors etc.)
- Dependencies between jobs
- Different prioritization schemes
- Implementing complex application logic based on job attributes
- Different kinds of relevant statistics
- Suspend/resume individual jobs as well as the entire queue
- Easy to use web-based UI

Conclusion

As this article demonstrated, Job Queue provides an easy way to improve user experience by performing tasks asynchronously, thereby keeping the page a lot more responsive. Moreover, it allows offloading tasks to backend machines, thereby relieving the load from the front-end web server. Job Queue also allows utilization of unused resources by scheduling resource-intensive tasks to hours where the site is relatively idle. Since Job Queue supplies a PHP client API, it is very easy to embed it into existing applications, both for simple routine tasks, and using the advanced features for building complex applications.

About the Authors:
Dotan Perry is a software engineer in the Zend Platform group, and is Job Queue's lead developer.
Shie Erlich is a software team leader in the Zend Platform group, co-designer of Job Queue, and co-author of the Krusader open source project.