WebSphere Application Server

WebSphere Liberty Batch Topologies

This document can be found on the web at:
www.ibm.com/support/techdocs
Search for document number **WP102626** under the category of "White Papers"

Version Date: September 29, 2017
See "Document change history" on page 20 for a description of the changes in this version of the document

IBM Software Group
Application and Integration Middleware Software

Written by:
David Follis
IBM Poughkeepsie
845-435-5462
follis@us.ibm.com

Don Bagwell
IBM Advanced Technical Sales
301-240-3016
dbagwell@us.ibm.com

Many thanks go to the WebSphere Batch team for getting all this work done.

Table of Contents

Introduction

Starting with the 8.5.5.6 maintenance level, WebSphere Liberty supports the Java EE7 standard. Part of that standard is JSR-352, the open standard for Java Batch. In addition to supporting the core batch programming model, the IBM implementation in Liberty includes numerous operational extensions. One of those extensions is the ability to involve multiple servers in a single batch environment. For example, you can configure one server to act as a 'dispatcher' with a collection of other servers configured as 'executors'. When you want to run a Java Batch job, you contact the dispatcher and it sends the job to one of the executors which actually runs the job.

There are a lot of possible ways to configure a set of servers as part of a Liberty Batch environment. In this paper we'll look at what configuration options influence the nature of the batch server topology. Once we understand our options, we'll consider some different possibilities and why they might be good (or bad) choices for your needs. And we'll try to steer you away from topology choices that simply won't work at all.

Configuration and Topology

The Job Repository

We'll be looking at several pieces of configuration that influence the nature of your batch topology but the most important one is the Job Repository. The Job Repository is a set of tables in a database that is used by the Liberty Batch code to keep track of what is going on. It knows what jobs have been submitted, where they are running, what state they are in, and for completed jobs it knows if they worked or failed. For those familiar with traditional batch environments on z/OS (JES2 and JES3), this is the spool (sort of).

In a development environment you can choose to use an in-memory Job Repository. This saves some fuss in setting up the environment, but any time the server restarts the contents of the Job Repository are lost. This is usually fine for development because you really don't care if the job you ran last Tuesday failed or not. In fact, being able to wipe out the Job Repository by just restarting the server is pretty cool. But it is just terrible for a production environment where you need to be able to restart failed jobs after server restarts.

Real production topologies will use a database to store the Job Repository. Several databases are supported and since this is a paper about topologies we're not going to get into which database product is best suited for this role (although it being an IBM paper you might suspect we have a preference...).

The configuration of each server in the environment will point to the location of the database where the tables live. You can find an example in the WebSphere Knowledge Center's “rwlp_batch_persistence_config” article.

In a multi-server environment how many Job Repositories do you need? It depends of course. Every job in a given repository will be visible to every server that is using that repository. So the number of repositories you need depends on how many servers need to share that information.

For example, if you have the dispatcher/executor configuration we mentioned in the introduction then those servers will need to share the same repository. The dispatcher will create entries in the repository that will be needed by the executor when it runs the job.

On the other hand, if you just have stand-alone batch servers, where the dispatcher and executor function is all performed in one server, then no other servers need to access its repository. Could

you share a repository across multiple stand-alone batch servers? Technically yes, but it might get confusing. We'll take a look at some of the weird things you can do with this when we start exploring topology choices.

The Job Repository database largely defines the scope of a Liberty Batch environment. All the servers sharing one set of tables should be considered part of that environment. Servers using separate Job Repositories are really separate environments.

The Queue(s)

The Job Repository is the easy part. Things get really interesting with the queue (or queues).

As we said earlier, you can configure a Liberty server as a batch Dispatcher or an Executor. Both bits of configuration point to a queue (either in MQ or in the integral messaging engine in Liberty). It should (hopefully) be fairly obvious that if you want the Executor to get messages the Dispatcher puts on the queue, then they both need to be configured to use the same queue.

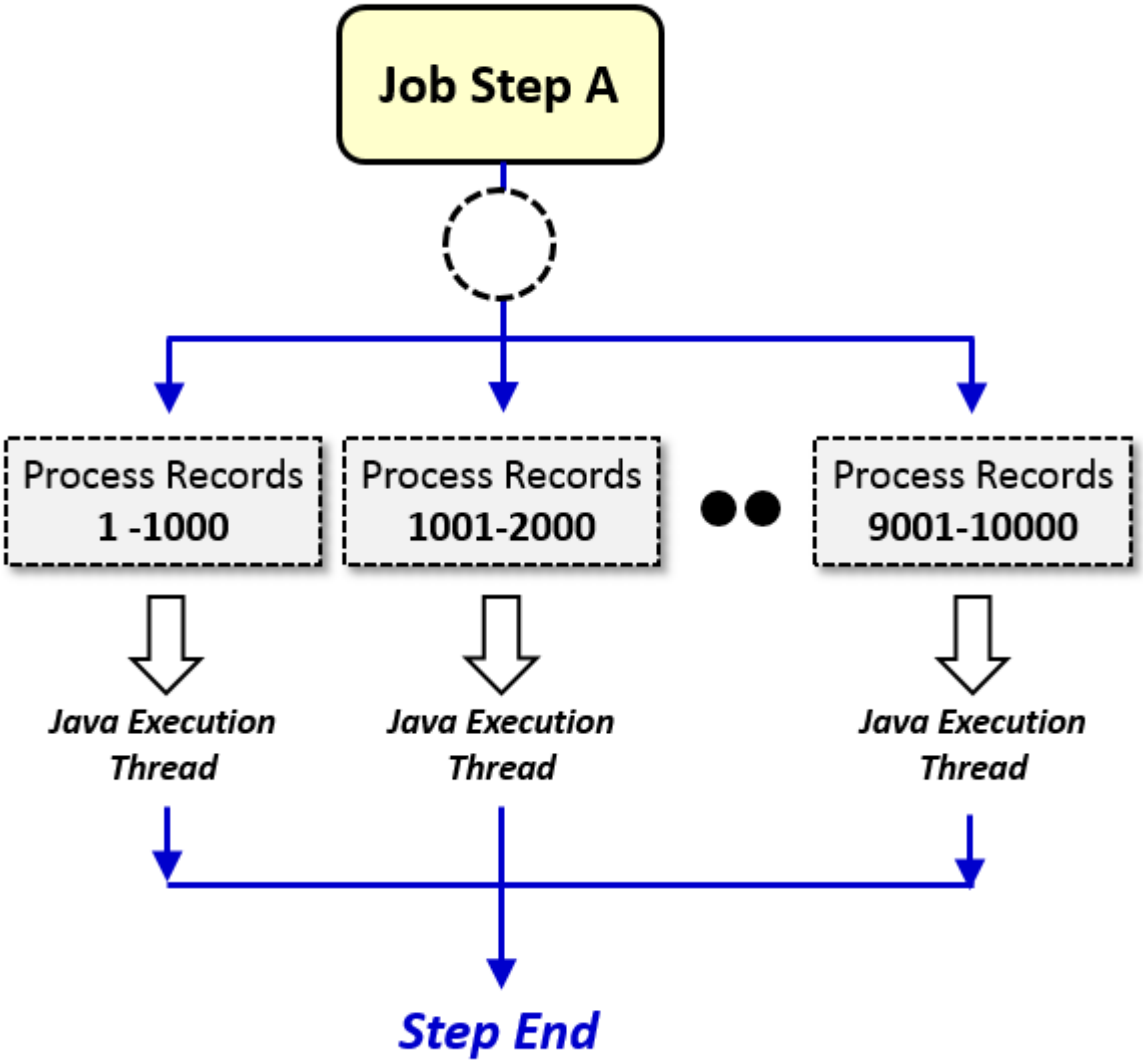
And you can have multiple Dispatchers and multiple Executors. Do the Dispatchers all need to point to the same queue? Well they do if they want the same set of Executors to run the jobs. Do the Executors all need to point to the same queue? They do if they all want to run the jobs dispatched through the same Dispatcher.

You can make use of the SELECTOR string in the configuration of the Executor to control which jobs are picked up from a given queue by different Executor servers. This is a topic we covered in WP102600.

But it gets more complicated. You can also configure an Executor to act as a Dispatcher. Why would you do that? To spread the execution of partitions across multiple servers.

Part of the JSR-352 specification allows for the definition of a partitioned step. In this scenario the execution of a single step of the job is broken up into several partitions. Each partition gets its own set of properties (to control what it does) and then runs the same code on separate threads. This allows you to potentially process a large volume of data more quickly. Instead of one thread processing ten thousand records, you can have ten threads running concurrently, each processing one thousand records.

Version Date: Friday, September 29, 2017



If you configure an executor that is running batch jobs to also be a Dispatcher, then when it encounters a partitioned step it can act as a Dispatcher and dispatch the partitions to run across a set of servers. That way, instead of running a bunch of threads in one server, you can run threads spread across multiple servers, possibly on multiple systems. The process of doing this is much like a regular Dispatcher giving work to Executor servers. The Executor creates a message for each partition and puts those messages on the queue pointed to by the Dispatcher configuration in the Executor. This dispatches the partitions to run on servers that are listening on that queue.

Can the regular dispatch queue and the queue used by an Executor to dispatch partitions be the same queue? Yes. Can they be different queues? Yes. What should you do? We'll get there, hang on.

Batch Events

The last configuration piece is batch events. Any server running Liberty Batch jobs can be configured to publish messages at interesting points in the lifecycle of a batch job. These messages are published to a topic tree and can be used to monitor what's going on in the environment. Each server has its own configuration that tells it where to publish the messages.

This doesn't really affect topology because it has nothing to do with how the work flows through the servers in your environment. But if you are monitoring your batch jobs by watching for events it is important that it align with the topology in a way that makes sense.

We won't talk about batch events in our topology considerations later on. Suffice it to say here that all the servers involved in the processing of a given job, whether Dispatcher, Executor, or Partition Executor, should all be publishing event messages to the same place.

If you have several environments that could be described that way then you should probably have separate places for the batch events. The key is really the Job Repository. The way you tell which job an event is about is by looking at the identifiers assigned to the job. Each job instance and execution is assigned an identifier (a number). Those numbers are unique WITHIN THE JOB REPOSITORY. If you are getting messages from servers using different Job Repositories, then those identifiers aren't unique and you might confuse an event about one job for an event about another job from a server using another repository.

The simplest thing to do is align the location of the batch event messages with the set of servers using the same Job Repository.

Other Configuration

What about collectives and clusters? The servers in your batch topology can certainly be a collective. You might want to do that for the advantages the collective brings you. But it has nothing to do with the batch features. Batch servers, whether Dispatchers or Executors, don't really care if they are in the same collective or not.

Similarly, you may wish to configure a set of Executor servers (or Dispatcher servers) as members of a cluster. But the batch features in those servers don't care if they are clustered or not. You might want to cluster them for the administration and management benefits, but batch itself doesn't care.

For this reason none of the topologies we'll go through will make any reference to collectives or clusters. You might imagine them drawn in with dashed lines around sets of servers if you like though.

Topologies

In this section we'll consider a variety of possible topologies. We'll look at how the Job Repository is (or isn't) shared and which message queue (or queues) are used. And we'll consider some of the pros and cons of each topology as we go. Remember that some of these might be very bad or even unworkable configurations.... Hopefully it will be clear which ones those are...

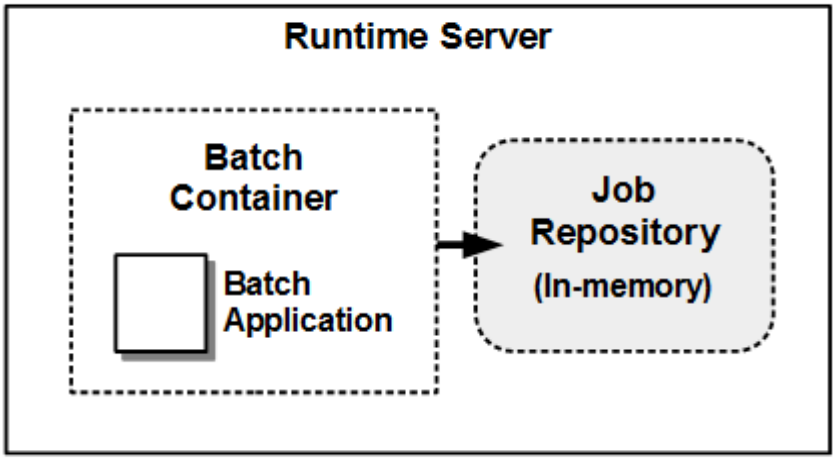
One quick thing before we get started... All of these configurations assume that the REST interface is being used to submit the jobs, either directly or via the Java or Native Command Line Interfaces. If you have an application that is calling the JSR-352 defined JobOperator API then those jobs will always run in the server where JobOperator was called, regardless of other topology things you may have set up.

I tried to make up interesting names for the configurations. This was really just a probably failed attempt to make this paper more interesting to read. If I've collided with some real name of something it's entirely an accident.

The Developer

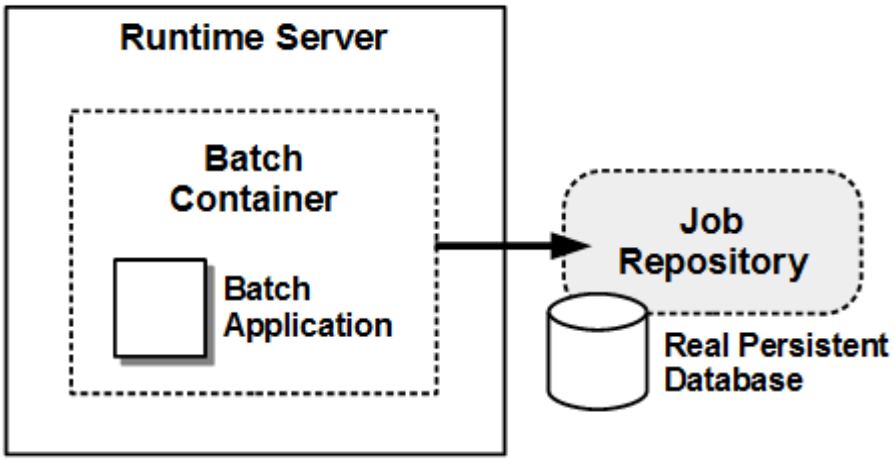
This is the simplest possible configuration. Just a single server, possibly running inside of Eclipse, and using the in-memory Job Repository. For just playing around with the batch capabilities or doing some serious batch application development this is a great choice. For any sort of real production work it is simply awful. The loss of any memory of what was going on with jobs on any server restart might be helpful to clean up a mess in development, but for production it is a disaster.

Start here, but for production, look elsewhere.



The Lone Wolf

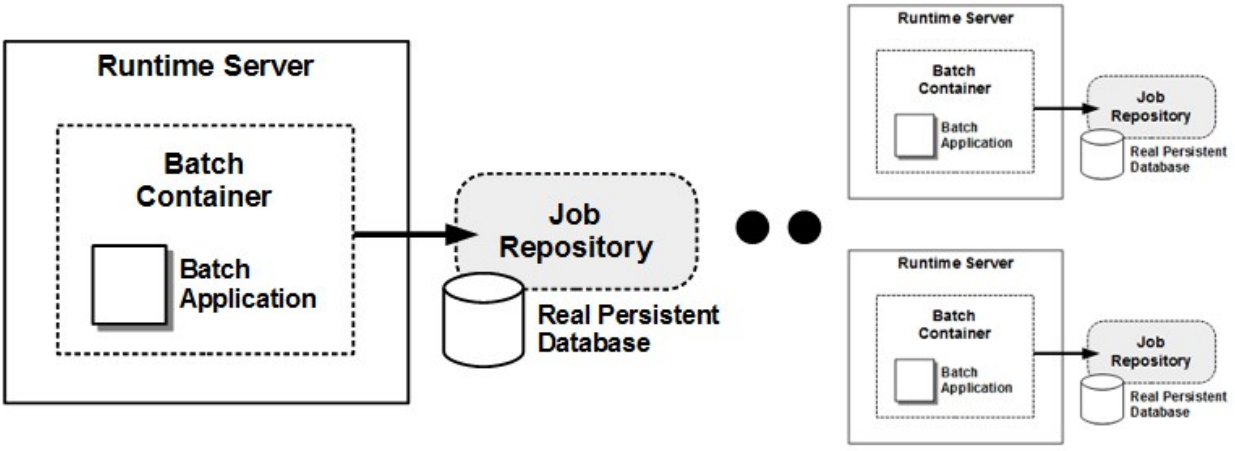
This is the simplest production capable environment. Essentially it is our Developer environment kicked up a notch to use a real persistent database for the Job Repository. Depending on your production requirements some databases might be better at this than others, but that's outside our scope.



This is a good place to go for a starter production environment. You don't have any HA capabilities or load balancing or anything else like that. But it is a start.

Wolf Pack - Independent

In this configuration we just have lots of Lone Wolf servers. Each server has its own Job Repository. Different applications are potentially installed in each server. Users have to know which server to go to in order to run each batch application. You could have some limited high availability coverage by having the same application in multiple servers, but users would need to handle switching to an alternate server themselves and there is no capability to restart a failed job in a different server.

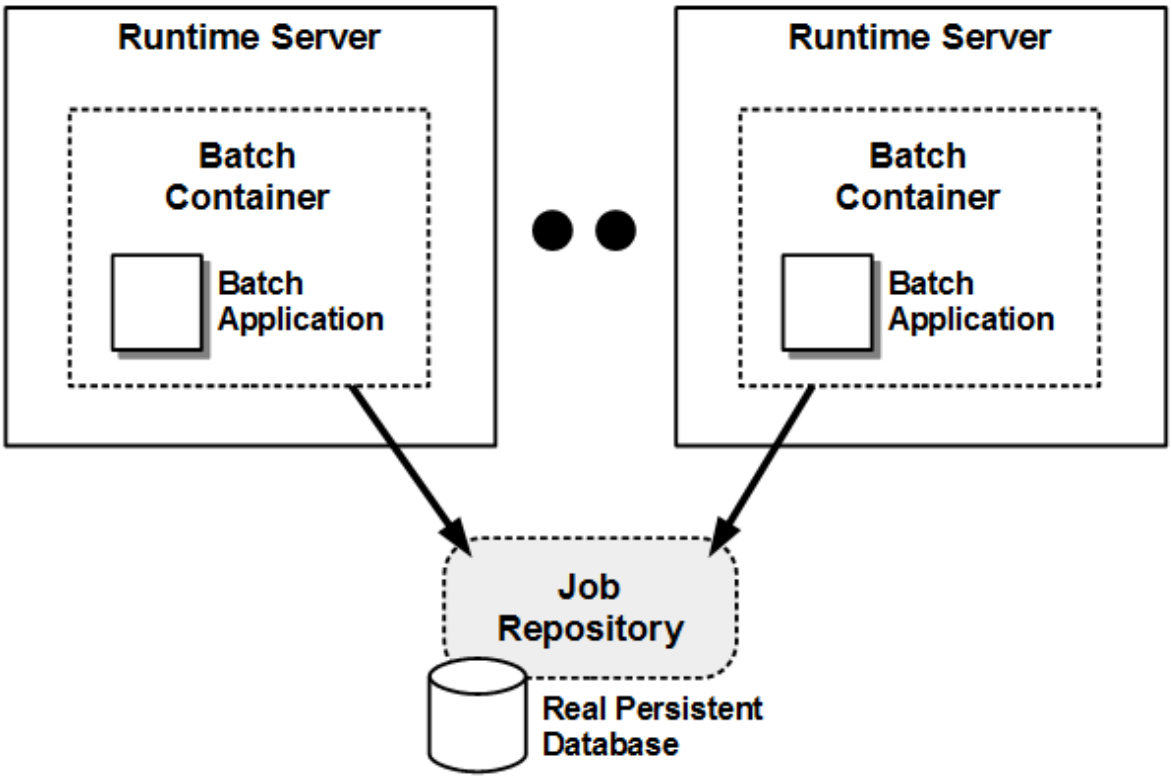


Wolf Pack - Group

This is similar to the Independent Wolf Pack configuration described just above. The difference in this scenario is the servers all share a Job Repository. You still have to know which servers have which applications installed in order to know where to go to run a particular job. You can get a little bit better HA by being able to restart a failed job in a different server if you have multiple servers with the same application installed.

But it can get a bit confusing. For example, you can ask any server about the status of a job, even if it isn't running in that server, because they all share the same Job Repository. But if the job fails and you want to restart it, you have to talk to the server where the application lives (assuming the servers don't all have the same applications).

In short, a shared repository between multiple Lone Wolf servers doesn't really buy you many benefits and greatly increases the chances for confusion.



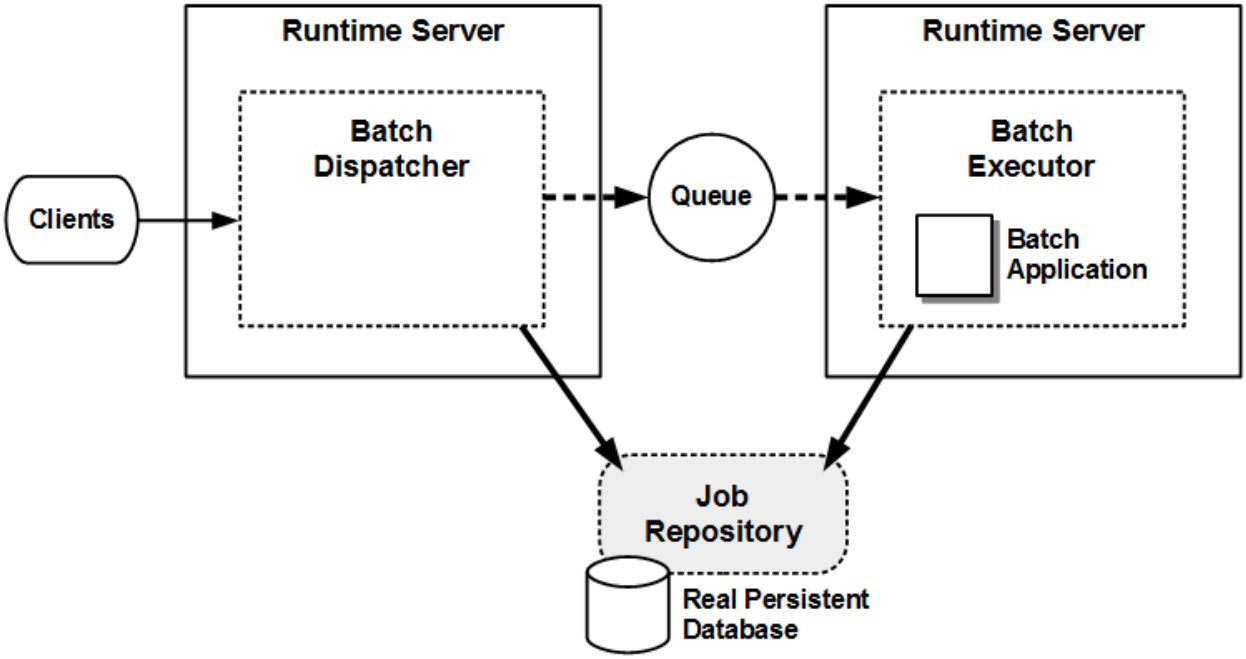
The Split Brain (one Dispatcher, one Executor)

Now things start to get interesting. We're going to take advantage of the messaging capabilities of the batch implementation in Liberty. For this configuration we just need two servers. One of them we configure to be a Dispatcher and the other is configured as an Executor. When the Dispatcher gets a job to run, instead of just running it right there in the server, it will put a message on a queue. The queue is part of the configuration of the Dispatcher.

The Executor is configured to process messages from that same queue (it would be an erroneous configuration to have them using different queues).

And of course both servers need to be using the same Job Repository.

Why would you want to use this configuration instead of The Lone Wolf? There really isn't a reason. You still have one server running all the jobs. You just have a different server acting as the contact point for the REST interface. It is a good starting point for moving to some of the more complicated multi-server configurations. But there isn't much point in running with just one Executor. Well...perhaps if the Executor server crashed you could still ask the Dispatcher about the state of the job. In the Lone Wolf configuration if the server goes down you can't ask it anything.



More Power (one Dispatcher, multiple Executors)

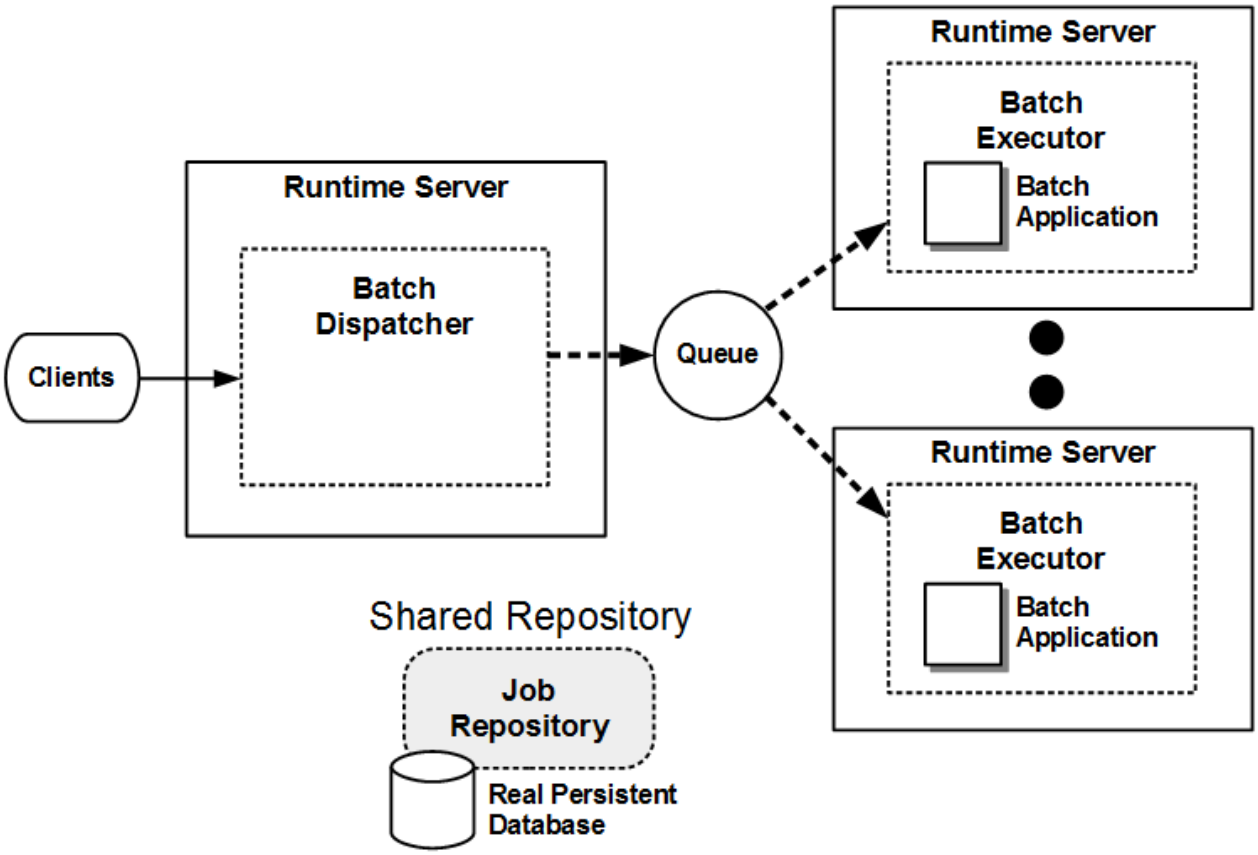
In the immortal words of Tim “The Tool Man” Taylor(TM), we need more power. We'll define multiple Executor servers all configured to get messages from the same message queue that the Dispatcher is using. But which server gets which jobs? If you do no additional configuration then any Executor can run any job.

However, you can include in the executor configuration a selector string. The selector is essentially an SQL Select that defines what messages will be processed from the queue by this server. The selector can select messages based on message properties like application name, or by user-defined properties that were provided when the job was submitted.

With knowledge of how the jobs are being submitted you can know which server (or set of servers) will run those jobs. It is also possible to not have any executor servers running for a particular job when it is submitted.

Suppose you have two types of jobs. Both can be submitted at any time of the day. One type needs to run immediately when it is submitted. The other type should only be allowed to run during a 'batch window'. You could define two sets of Executor servers with a Selector chosen to differentiate between the jobs. The servers for the run-immediately jobs are always up. The servers for the other jobs are only started during the batch window and shut down when the window ends. The messages for those other jobs will wait in the message queue until a server starts up to select them.

The down side of this flexibility is that if you aren't careful with your selector definitions it could be possible to submit a job that no server will ever be eligible to run.



And, of course, like the basic Split Brain configuration all the servers involved have to be using the same Job Repository.

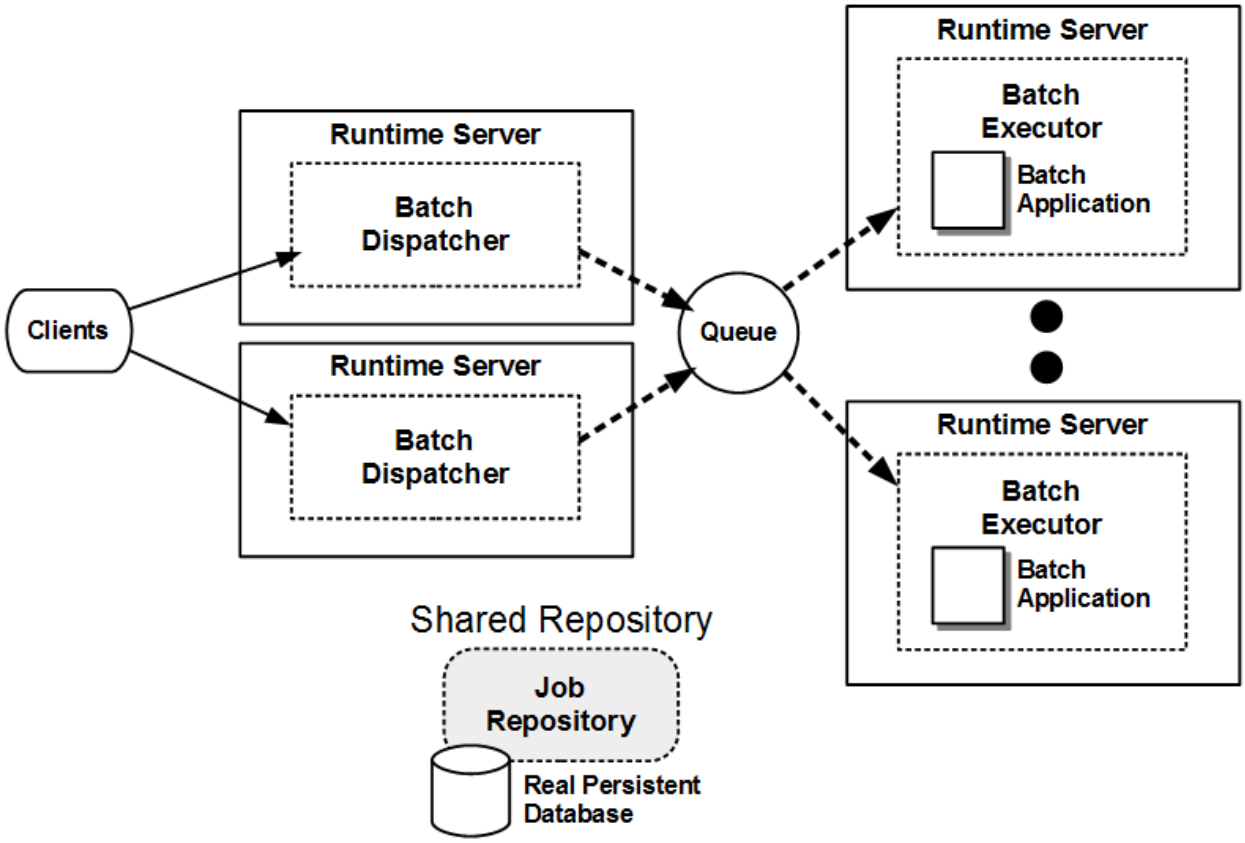
HA Power (multiples of both)

The “More Power” configuration is pretty cool, but that one Dispatcher is a single point of failure. If the Dispatcher server goes down, there is no way to submit jobs or do other job management stuff. Can't we have more than one?

Of course! Just define another server as a Dispatcher using the same message queue and the same Job Repository as the first one. Clients can contact either Dispatcher over the REST interface (or perhaps some request routing mechanism in front selects where it goes) and manage their jobs. The jobs have no affinity to which Dispatcher got things started. You can start a job with one Dispatcher and use the other one to check the job's status. They share a Job Repository so what one Dispatcher knows, the other one also knows.

Could you have more than two Dispatchers? Sure. As many as you like.

For a production environment this is probably the best topology to use. At least until you start using partitioned steps heavily....

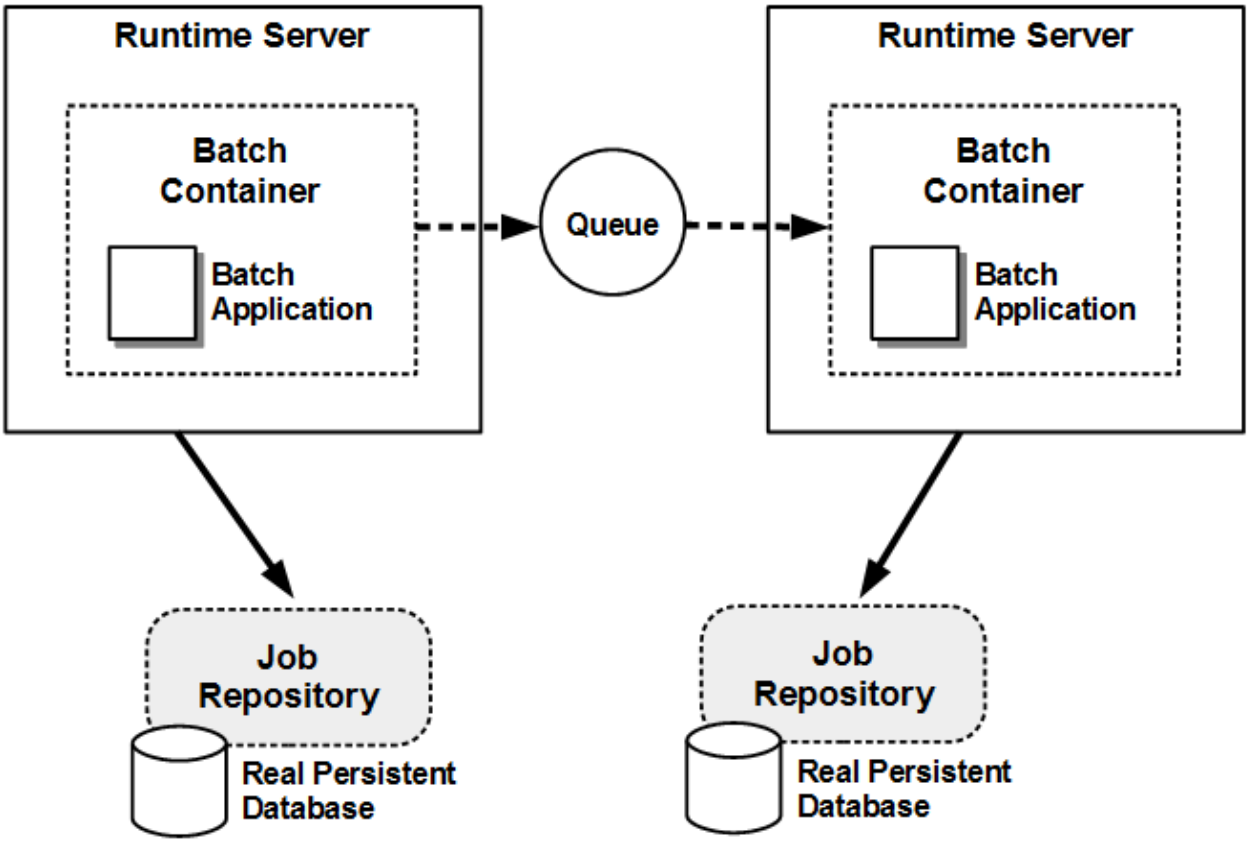


Broken Brain (one Dispatcher, one Executor, Two Repositories)

We've made this point a few times already, but I promised to include some configurations that don't work and this is one of them. In this configuration we have a Dispatcher and an Executor, both configured to use the same message queue. But each server is using a separate Job Repository. This just doesn't work. What will happen?

A client will contact the Dispatcher to submit a job. The Dispatcher will create entries in the Job Repository about the job and create a message to represent the job. The Dispatcher will put the message on the queue. The Executor will get the message and begin to process it. Part of that processing involves finding the information in the Job Repository put there by the Dispatcher. But since the Executor is using his own repository, he can't find any of the information he needs. The job will fail. When the client uses the REST interface to ask the Dispatcher about the status of the job, the Dispatcher will look in his Job Repository. From that information it appears the job is still waiting for an Executor to pick it up, because the Executor couldn't update the right repository.

It is just a big mess. Don't do this.

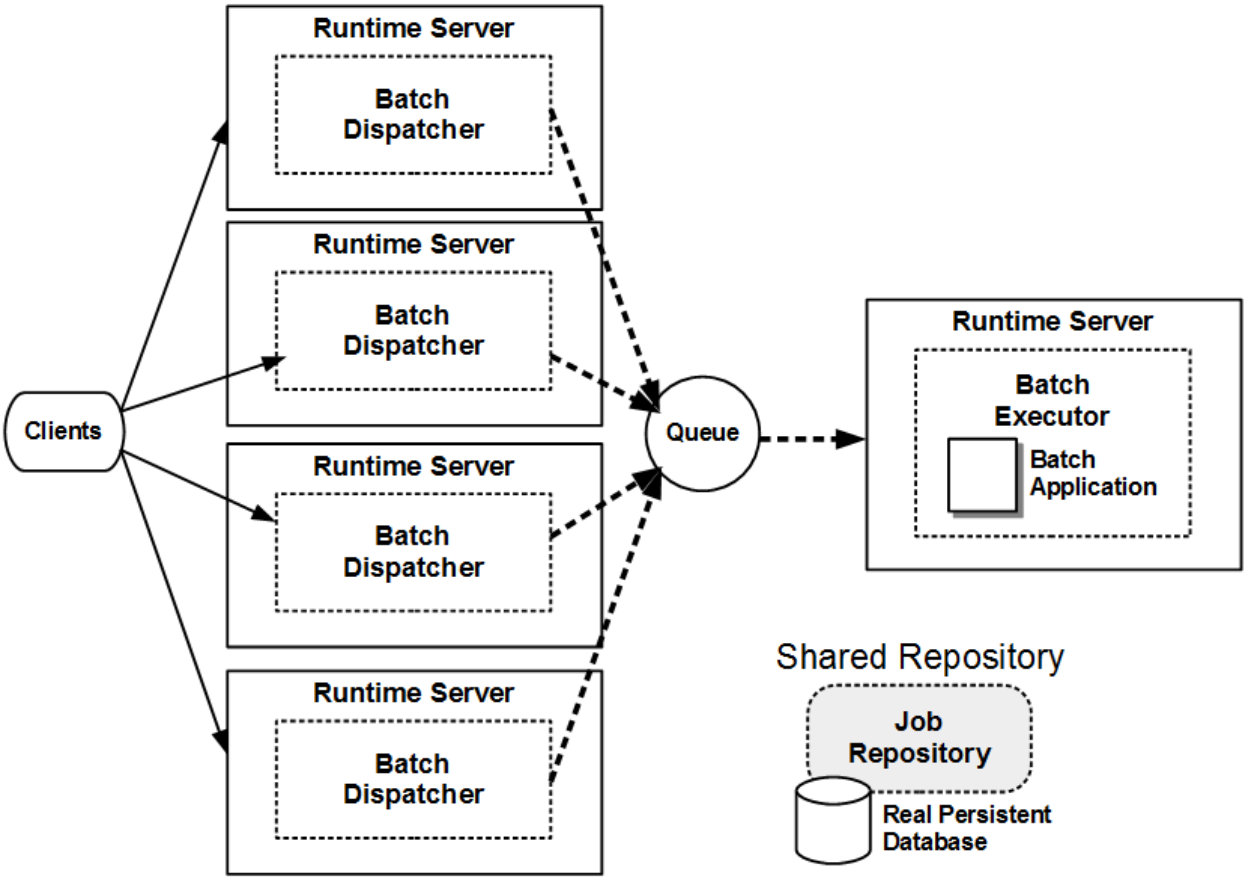


The Hydra (multiple Dispatchers, one Executor)

Back to configurations that work, we'll go to another weird one. This, again, is a workable configuration that doesn't have any apparent benefits to it. We'll start off like the HA Power configuration with multiple Dispatchers. But instead of also configuring a lot of Executors, we'll just have one.

In this configuration the client has a list of possible end points to contact to work with batch jobs, but all the jobs run in one server. They would, of course, have to share a Job Repository.

This configuration works, but I can't think of any reason why you'd want to do it.



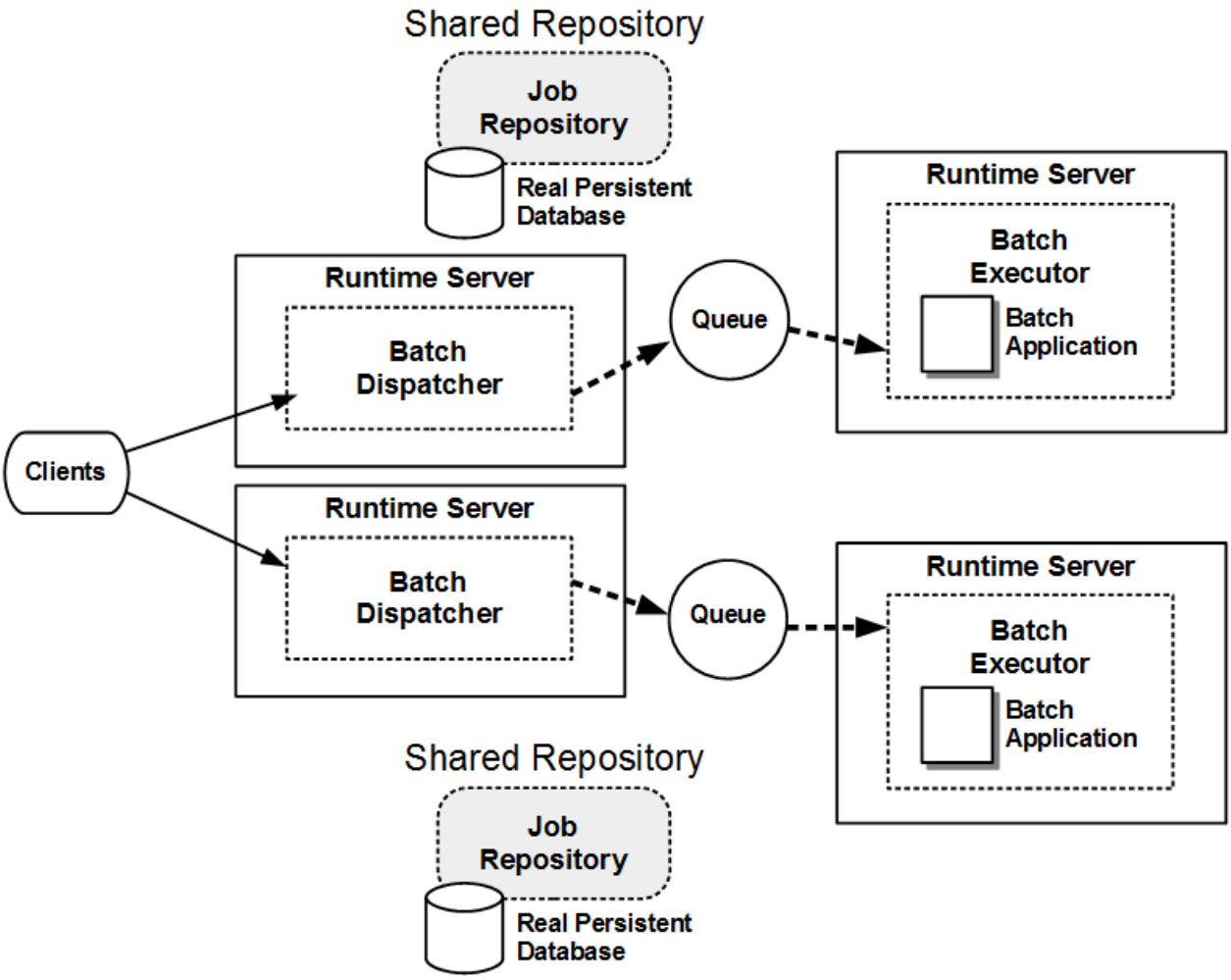
Parallel Power (Two Dispatchers, 2X Executors, Two Repositories)

This one sounds like a really cool, really complicated configuration. But it isn't. For this configuration we have multiple Dispatchers (say two to keep it simple) and multiple Executors (let's say two). Feels a lot like the HA Power configuration. But.... they aren't all using the same Job Repository.

In this configuration one Dispatcher/Executor pair is using one repository and the other Dispatcher/Executor pair is using another repository. What does that mean?

It means they are completely separate configurations. It's really just two of the basic Split Brain configurations running side by side.

In order for it to work they can't both be sharing the same message queue. If the two dispatchers are feeding messages into the same queue and the two Executors are getting messages from that same queue... Well, if you are very very careful with the job properties and the Selector strings so you are absolutely sure that jobs end up in the right Executor you could get away with it. It hardly seems worth the risk.



So what is the point of this configuration? Well, as we said it is really just two of the same configuration running side by side. In truth you can run all of these configurations (well, the ones that work) side by side in the same environment as long as you carefully respect the boundaries of the Job Repository and message queue. All these different things may be part of what you consider your batch topology. But the individual pieces of it just see themselves in their own little topology. There is no larger awareness of your grand plan.

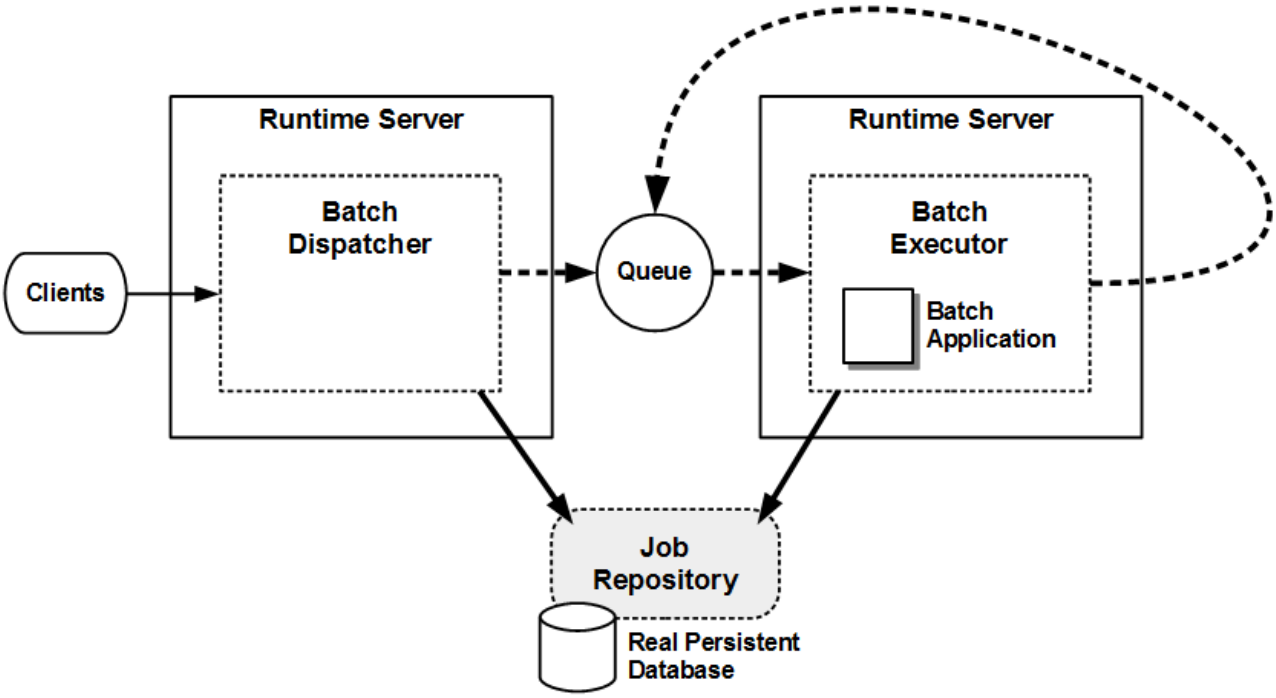
Partition Madness (Dispatcher, Executor handling both regular and partition jobs)

We talked about partitions earlier but let's review quickly. In the JSL that defines the job you can create a partitioned step. This causes the step to break into some number of pieces and some or all of those pieces may run concurrently. When all the pieces are complete, the step is complete and flow moves to the next step.

Basic partition support will run those partitions on threads in the server running the job, while the main job waits on the thread executing the job itself. But you can configure the Executor server to create messages for those jobs and push the messages onto a queue to be run by Executor servers listening for messages on the queue.

In this configuration the same servers that are running the main jobs are also allowed to take partitions. If you don't do anything special in the selectors (which we'll get to in our next configuration) then partitions are just like regular jobs and get routed by the application name property or others depending on our selector setup.

This is an easy configuration to set up as an extension to any of our Dispatcher/Executor configurations. Updating the Executors to also act as Dispatchers enables multi-JVM partition execution. But you have to be sure there is somewhere for those partitions to run. Planning to run them in the same servers as the main jobs might not work out well for you as loopbacks and resource contention may lead to deadlocks.



Three Layer Tree (Dispatcher, Executor, Partition Executor, one Queue)

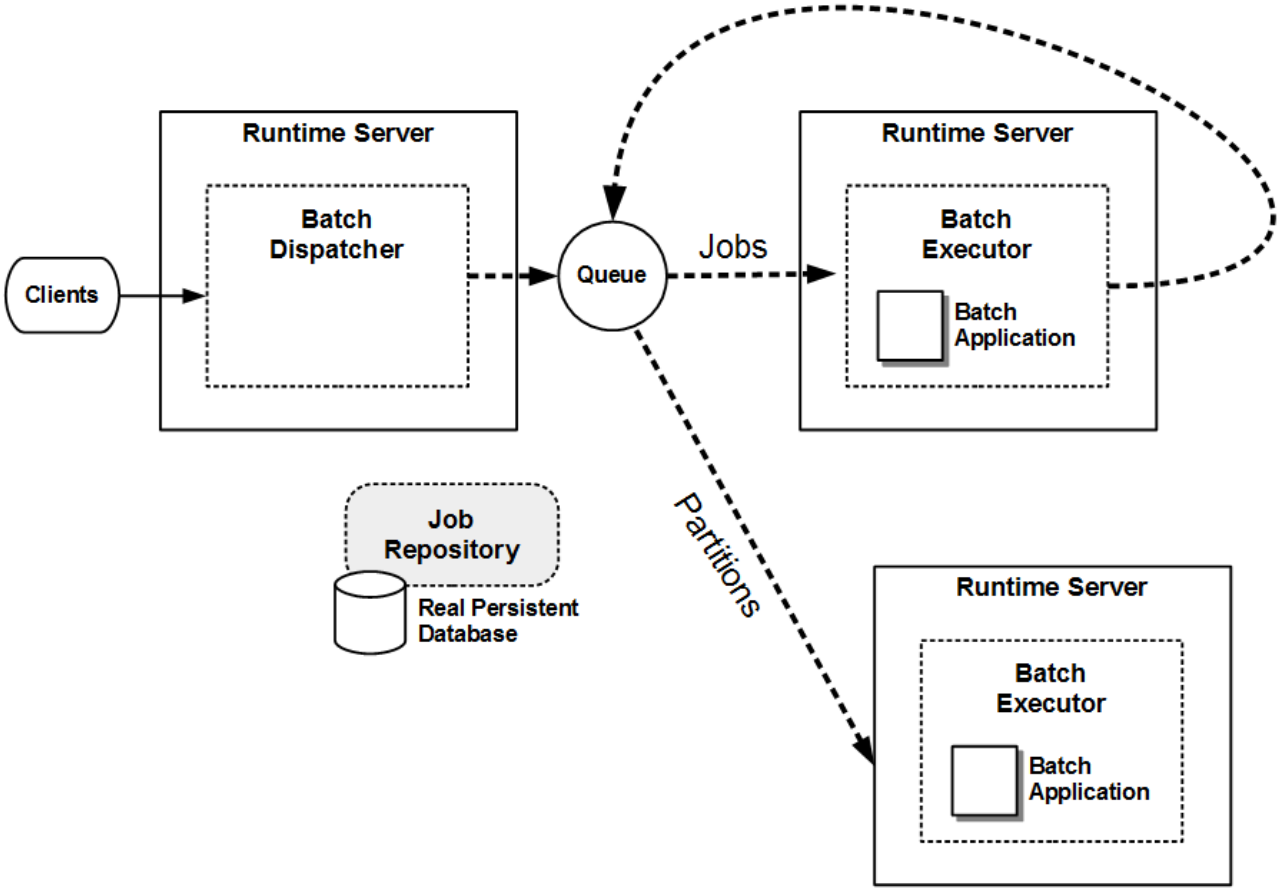
If you'd like to distinguish between messages for real jobs and messages for partitions, can you do that? Fortunately there is a property set on every batch message that indicates whether it is for a job or a partition (`com_ibm_ws_batch_work_type`). You can use that property in a Selector string to control whether an Executor runs jobs or partitions, in addition to managing messages by application or whatever else you like.

This allows us to create a three tier architecture. In the first layer we have the Dispatcher (or Dispatchers). In the second layer we have Job Executors. These specifically request messages that are work_type=Job. You may have many different Executors that process different applications or jobs that have other specific characteristics (e.g. JobClass, etc). The third layer consists of Partition Executors. These Executors specifically request messages that are work_type=Partition.

Each Job Executor server (or set of servers) that process the same jobs (e.g. jobs from the same application) will have a Partition Executor server (or set of servers) behind it that process partitions from jobs running in those Job Executors. If you know that a particular application never has partitions (or you haven't configured it to Dispatch messages for partitions) then you don't need Partition Executors behind it.

Remember as you think about this that nothing is special about a Partition Executor. It is all determined by the properties/values mentioned in the Selector statement.

Of course all these servers have to use the same Job Repository and they can quite safely use the same message queue for messages relating to both jobs and partitions.

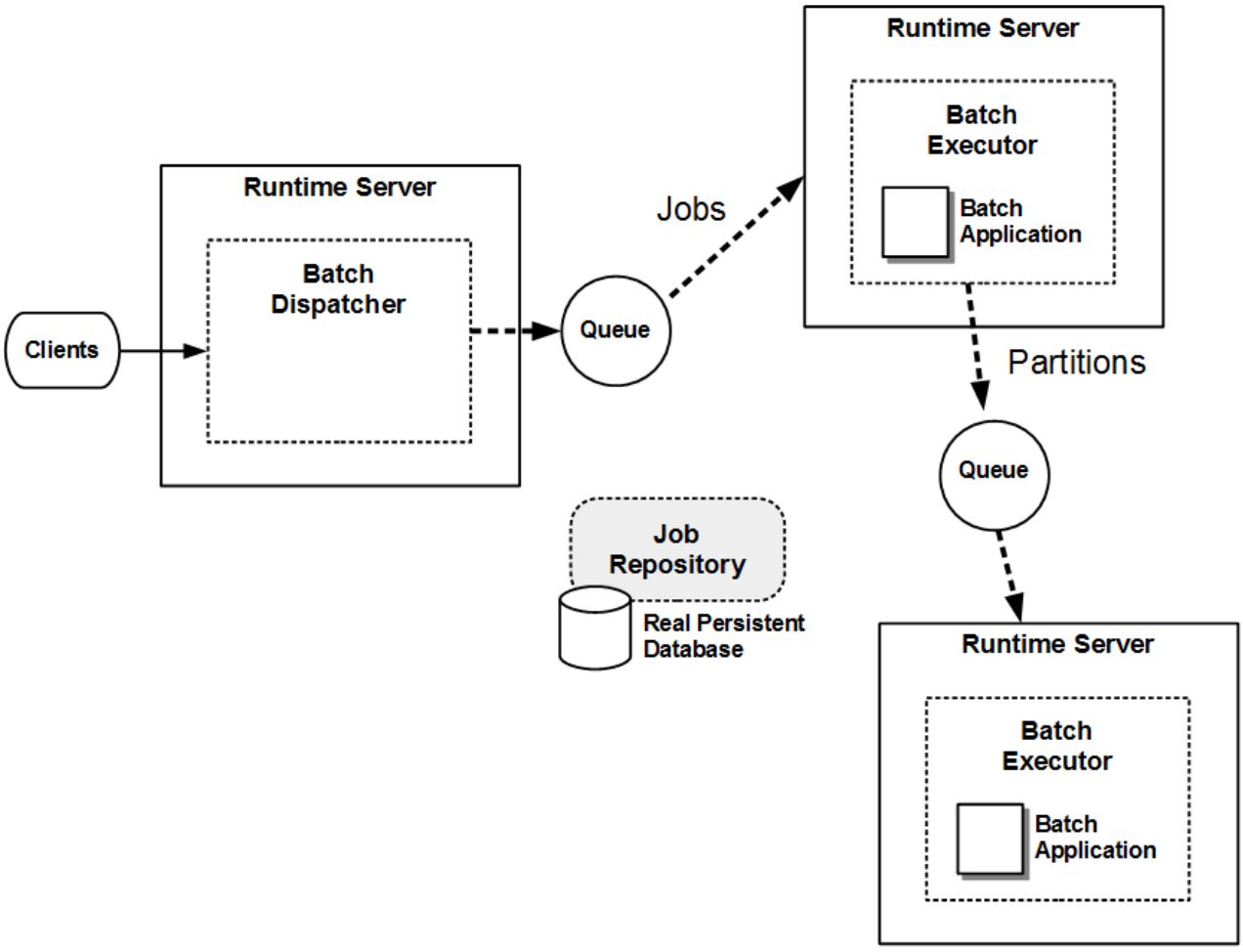


Multi-Q-Tree (same but with separate queues for partitions)

This configuration is another one of those things you CAN do, but it isn't clear why you'd want to. If we consider the Three Layer Tree (I wanted to call it a Three Layer Cake but I've never seen a cake that branched out..) you'll see that the messages between the Dispatchers and Job Executors and the messages between the Job Executors and their Partition Executors all go through the same queue. It is the message properties that determine where the messages go.

If for some reason that makes you uncomfortable, then it is entirely possible to configure the Job Executors to Dispatch their partitions by putting messages into a queue only used by their Partition Executors. This could make for quite a few queues.

It isn't clear that this buys you anything, but it is possible to do if you want.



Document change history

Check the date in the footer of the document for the version of the document.

February 15, 2016	Initial Version
February 16,2016	Add document number
March 11, 2016	Fixed document properties title
September 29, 2017	Correct typos (thanks Carl!)

End of WP102626