WebSphere Application Server for z/OS V7

WebSphere Application Server and the z/OS Workload Manager

This document can be found on the web at:
`www.ibm.com/support/techdocs`
Search for document number **WP101740** under the category of "White Papers"

Version Date: October 22, 2012
See "Document change history" on page 17 for a description of the changes in this version of the document

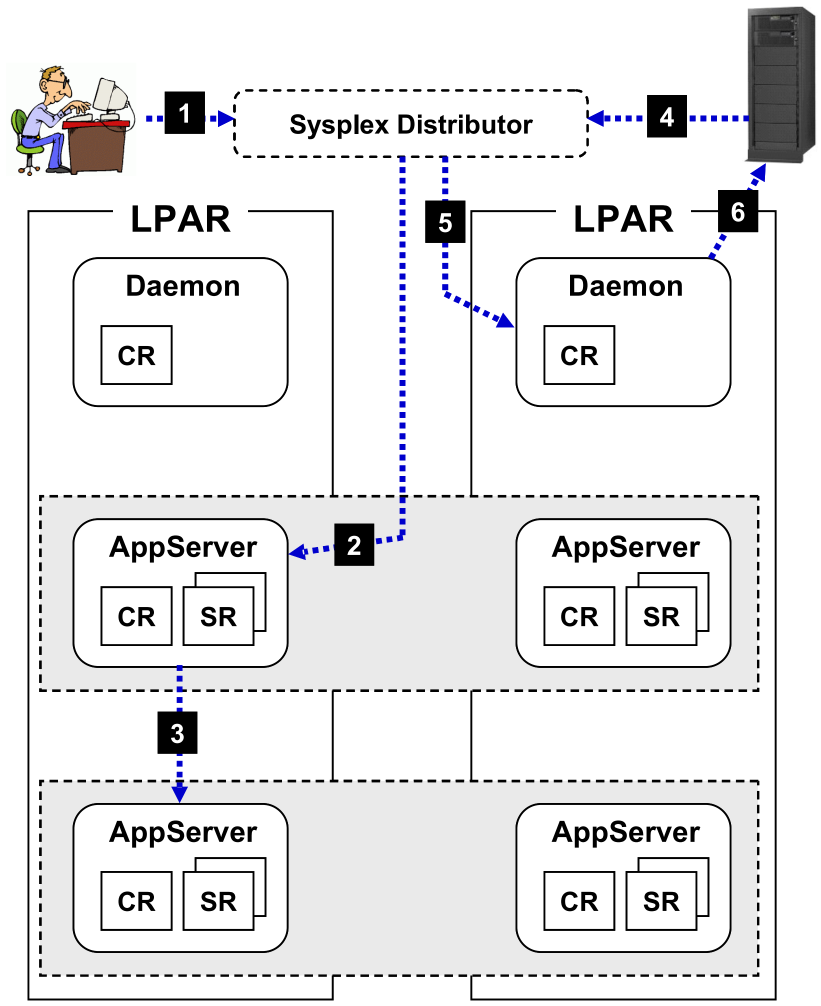
IBM Software Group
Application and Integration Middleware Software

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

Many thanks go to ...Don Bagwell, John Hutchinson, Mike Loos, and Robert Vaupel.

Introduction

Suppose you have installed WebSphere Application Server for z/OS into one of your LPARs. You followed the install instructions for classifying the WebSphere started tasks into an appropriate WLM service class. The servers start up without problems. The application that needed WebSphere got installed and seems to run pretty well. Your configuration looks something like this:



But now there are rumors about more applications and more servers. The application you have now uses HTTP, but the new stuff might include MDBs or Async Beans. The applications come with service level agreements for expected response time goals. How do you explain these goals to WLM? How can WLM even tell all this stuff apart? WebSphere must be involved somehow.

This paper will explain a little bit about the z/OS Workload Manager and a little bit about WebSphere Application Server, but focus mostly on how they interact and the configuration options that affect that interaction. We will go through some examples and look at how you can monitor your system to observe the consequences of changes to your configuration.

The Basics

All application work that runs inside WebSphere Application Server runs under a Workload Manager (WLM) enclave. WebSphere creates the enclave for each request that gets dispatched. An enclave is associated with a WLM service class. WebSphere uses information it has about the request to help WLM decide which service class to associate with the enclave. The service class tells WLM what level of service the work requires (think response time goal).

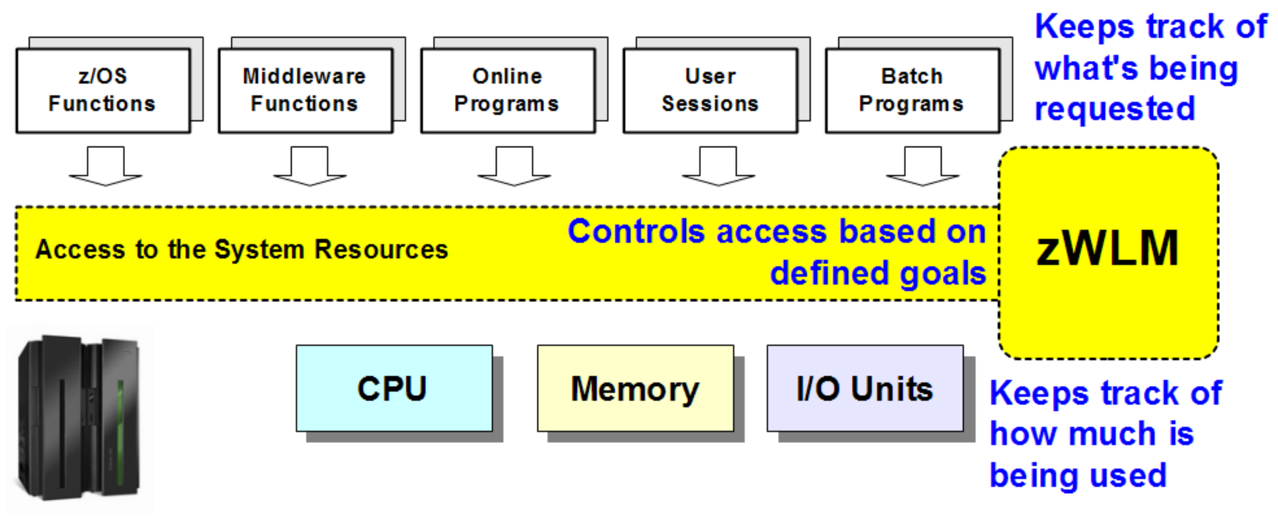
That is pretty much all there is to it. In the rest of this paper we will define what some of these terms mean and go through some of the options you have to help WebSphere and WLM understand what behavior you want.

What is Workload Management?

Workload Management consists of categorizing, prioritizing, and routing work. Resources, such as CPU or memory, are provided to those units of work which are more important in order to achieve specific goals. These goals are often referred to as a Service Level Agreement (SLA). Work is routed to systems or processes which best enable that work to meet its goals.

What is WLM on z/OS?

On z/OS the Workload Manager (WLM) component allows the system programmer to configure the rules which will be used to categorize requests. These are called classification rules. Once classified, WLM can determine what the goals (SLA) are for that work. For example, a goal for one workload might be a response time of less than 1 second for 95% of the requests. WLM will assign system resources to units of work from that workload making its best attempt to allow all work to meet its specified goals. Less important work will be sacrificed to meet the goals of more important work if necessary. WLM also provides routing advice to help direct units of work to a system or server in the sysplex most suited to meeting the goals of that work.


z/OS Functions
Middleware Functions
Online Programs
User Sessions
Batch Programs
Keeps track of what's being requested
Access to the System Resources
Controls access based on defined goals
zWLM
CPU
Memory
I/O Units
Keeps track of how much is being used

Defining Terms

Before getting started with how everything works, we should make sure we understand all the terms we will be using. This section will define and discuss some common terms used by the z/OS Workload Manager (often stealing from the book MVS Planning: Workload Management (SA22-7602).

Another good source of basic information on z/OS WLM is a paper by Robert Vaupel which you can find here (those are underscores, not spaces):

http://www-03.ibm.com/systems/resources/servers_eserver_zseries_zos_wlm_pdf_zWLM.pdf

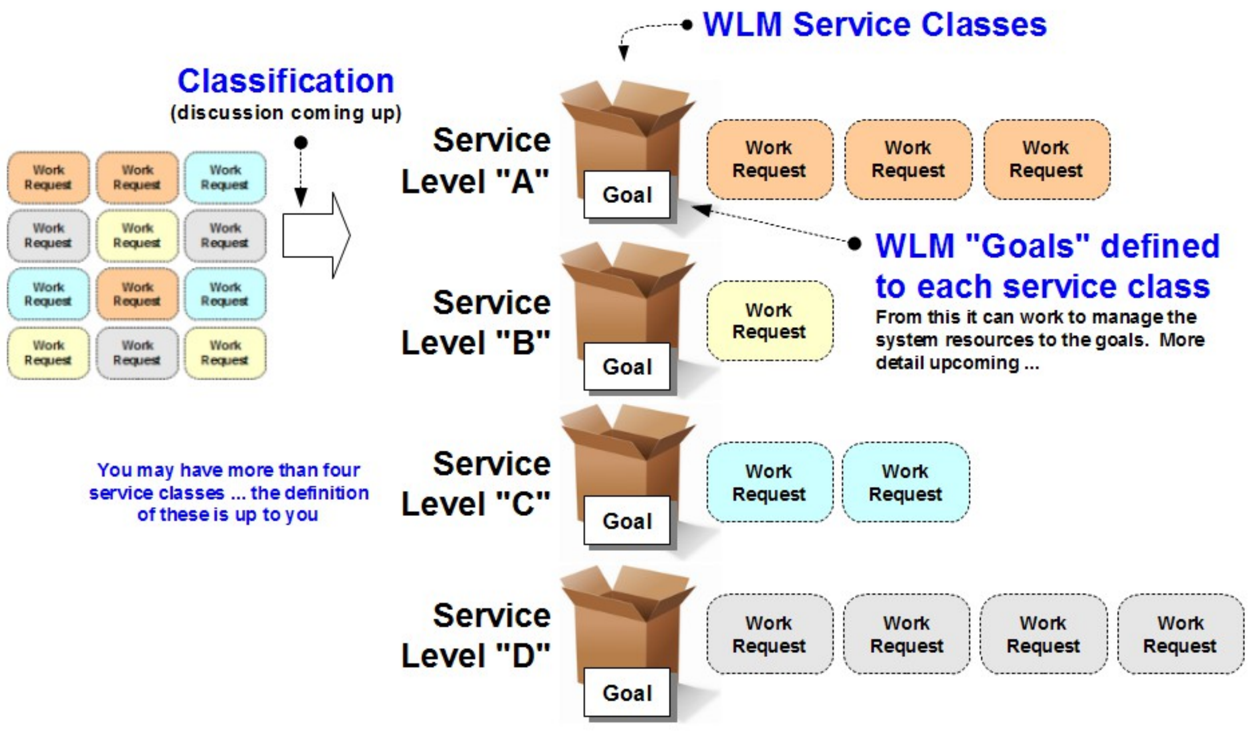
Service Class

The WLM documentation defines a service class as:

> Service classes, which are subdivided into periods, group work with similar performance goals, business importance, and resource requirements for management and reporting purposes. You assign performance goals to the periods within a service class.

Defining different periods allow a unit of work to become progressively less (or more) important and to change goals as it uses more resources. For example, a unit of work could have an importance of '1' until it uses 5 seconds of CPU time after which its importance drops to '2'. Short requests get more attention, but longer running requests to get less.

We will define 'importance' and types of 'goals' in a little bit.

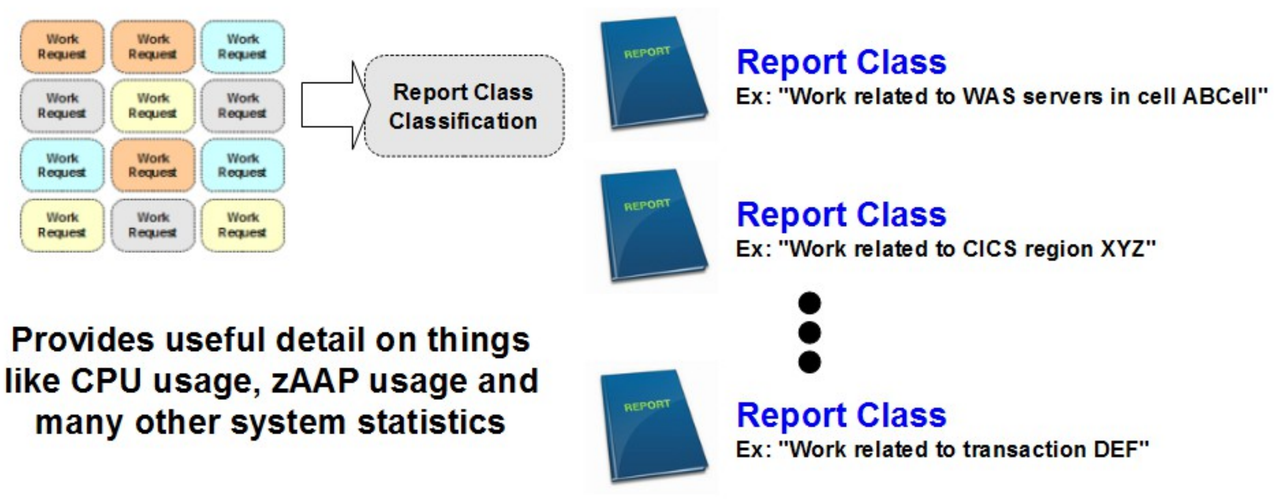

WLM Service Classes
Classification
(discussion coming up)
Work Request
Service Level "A"
Goal
WLM "Goals" defined to each service class
From this it can work to manage the system resources to the goals. More detail upcoming ...
Service Level "B"
Service Level "C"
Service Level "D"
You may have more than four service classes ... the definition of these is up to you

Report Class

The WLM documentation defines a report class as:

> Report classes […] group work for reporting purposes. They are commonly used to provide more granular reporting for subsets of work within a single service class.

As you define the goals for different types of work you will probably find that different, unrelated, work may have the same goals. This allows you to assign these different types of work into the same service class. However, you probably want to know what is happening with each different type of work. Assigning different report classes to work in the same service class allows you to generate separate reports for work that just happens to share the same goals.


Work Request
Report Class Classification
Provides useful detail on things like CPU usage, zAAP usage and many other system statistics
REPORT
Report Class
Ex: "Work related to WAS servers in cell ABCell"
Report Class
Ex: "Work related to CICS region XYZ"
Report Class
Ex: "Work related to transaction DEF"

Enclave

The WLM documentation defines an enclave as:

> Enclaves are transactions that can span multiple dispatchable units in one or more address spaces.

In order to manage something running in z/OS as a single unit of work, WLM has to have a way to know which service class is associated with each piece of work. Inside WebSphere a work request is associated with a WLM structure called an enclave. The enclave has an associated service class and report class. This allows WLM to know what goals to use when managing the thread where the request is dispatched and how to report the work it does.

Do not confuse WLM enclaves with LE enclaves. They are completely unrelated.

Classification Rules

How does an enclave get associated with a report and service class? Classification rules defined in the WLM policy are used when an enclave is created to determine which report and service class are appropriate for this enclave. There are several attributes that can be used to classify a unit of work. WebSphere just uses a few of them.

Subsystem Type

This classification field scopes the other fields to the subsystem interacting with WLM. It could be a value like 'CICS' or 'DB2' or 'ASCH'. For classification rules that apply to enclaves created by WebSphere Application Server, use a subsystem type of 'CB'. Why 'CB'? Because what we presently know as WebSphere Application Server began life as a product called Component Broker. The subsystem type was created in WLM during those early days and as the product evolved and changed names the subsystem type remained the same.

Collection Name

What WLM knows as the collection name is the WebSphere cluster name (also known as the short cluster name or the cluster transition name). This is the name that groups together servers which are part of a cluster. Having this name in your classification rules allows you to assign a service class and report class based on which server is running the work. We use the cluster name instead of the server name because we assume that servers in the same cluster are running the same types of work. Many customers separate their applications into different servers and simply use the cluster name/collection name to assign service and report classes to their application work.

Transaction Class

The transaction class is an eight character name used by WebSphere to group related/similar application work. The default value is all blanks. However, WebSphere can provide a value to help identify the work it is running. You can create an XML file, called the classification XML file, which provides rules for WebSphere to use in determining the transaction class name.

This file is essentially a second layer of classification. You use the classification XML file to group similar/related WebSphere work into the same transaction class. Then you can use the WLM classification rules to assign work into service and report classes. You can also separate out similar but different work.

For example, you could assign different parts of a single application to different transaction classes (based on the URI of different servlets, for example). Then assign the two transaction classes to the same service class (because everything in the application has the same goals) but different report classes. Then you can monitor the two parts of the application separately.

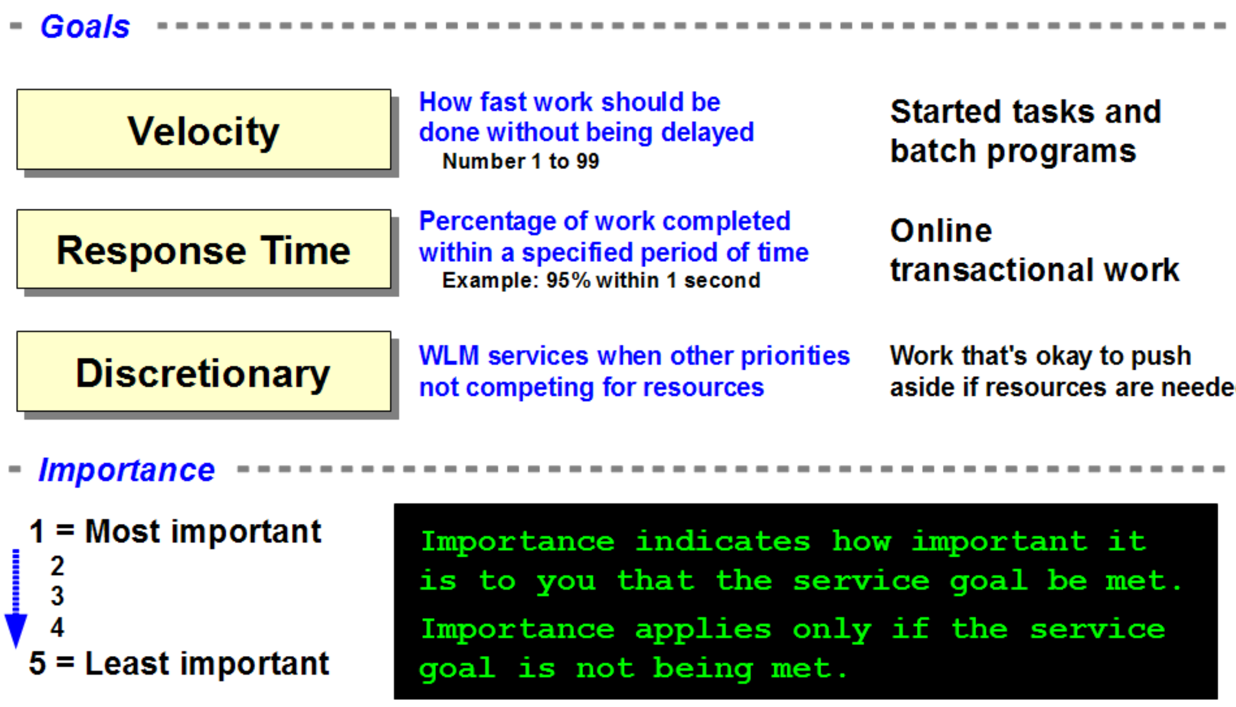
We will talk a lot more about how to set up the classification XML file in later sections.

Transaction Name

This is another eight character name that can be used to classify the request. The transaction name is only used when propagating WLM classification from a CICS transaction using the WebSphere Optimized Local Adapter (WOLA). We will describe how this works later when we cover setting up the classification XML file.

Goal Types

Each period in each service class has a goal. Different types of work will have different goals. WLM allows you to describe that goal in three different ways.



Goals

Velocity	How fast work should be done without being delayed Number 1 to 99	Started tasks and batch programs
Response Time	Percentage of work completed within a specified period of time Example: 95% within 1 second	Online transactional work
Discretionary	WLM services when other priorities not competing for resources	Work that's okay to push aside if resources are needed

Importance

1 = Most important
2
3
4
5 = Least important

Importance indicates how important it is to you that the service goal be met.

Importance applies only if the service goal is not being met.

Velocity Goal

First, the official WLM definition:

> A measure of how fast work should run when ready, without being delayed for WLM-managed resources. Velocity is a percentage from 1 to 99.

Velocity goals are usually used for longer running tasks like server address spaces.

Response Time Goal

From the book:

> The expected amount of time required to complete the work submitted under the service class, in milliseconds, seconds, minutes and hours. Specify either an average response time, or response time with a percentile. Percentile is the percentage of work in that period that should complete within the response time.

Response time goals are usually used for more transactional work such as an HTTP request. We will use response time goals in defining service classes for WebSphere Application Server requests.

Discretionary

This is a simple concept, but WLM provides a long definition:

> With discretionary goals, workload management decides how best to run this work. Since workload management's prime responsibility is matching resources to work, discretionary goals are used best for the work for which you do not have a specific performance goal. For a service class with multiple performance periods, you can specify discretionary only as the goal in the last performance period. Discretionary work is run using any system resources not required to meet the goals of other work. If certain types of other work are overachieving their goals, that work may be "capped" so that the resources may be diverted to run discretionary work.

So, in summary, discretionary work will generally only run if WLM has nothing better to do (mostly… as with most things, it can get pretty complicated when you get into the details). Assign work a discretionary goal if you are in no rush for it to complete.

Business Importance

First, the official WLM definition of 'importance':

> When there is not sufficient capacity for all work in the system to meet its goals, business importance is used to determine which work should give up resources and which work should receive more. You assign an importance to a service class period, which indicates how important it is that the goal be met relative to other goals. Importance plays a role only when a service class period is not meeting its goal. There are five levels of importance: lowest (5), low, medium, high, and highest (1).

The tricky part with assigning importance is not making everything have an importance of '1'. Something has to be less important. If you give everything the same importance then all goals are equal and WLM is free to just pick something when deciding which work will meet the goals you set and which work will not. Of course WLM does not just randomly select something. It follows complicated rules to make a decision that will probably look random to you. Perhaps instead of asking yourself how important the work is, instead ask yourself, "Would I be willing to not meet the goals for this work in order to meet the goals of something else?"

How WebSphere Application Server Classifies Work

What do we mean by 'Work'?

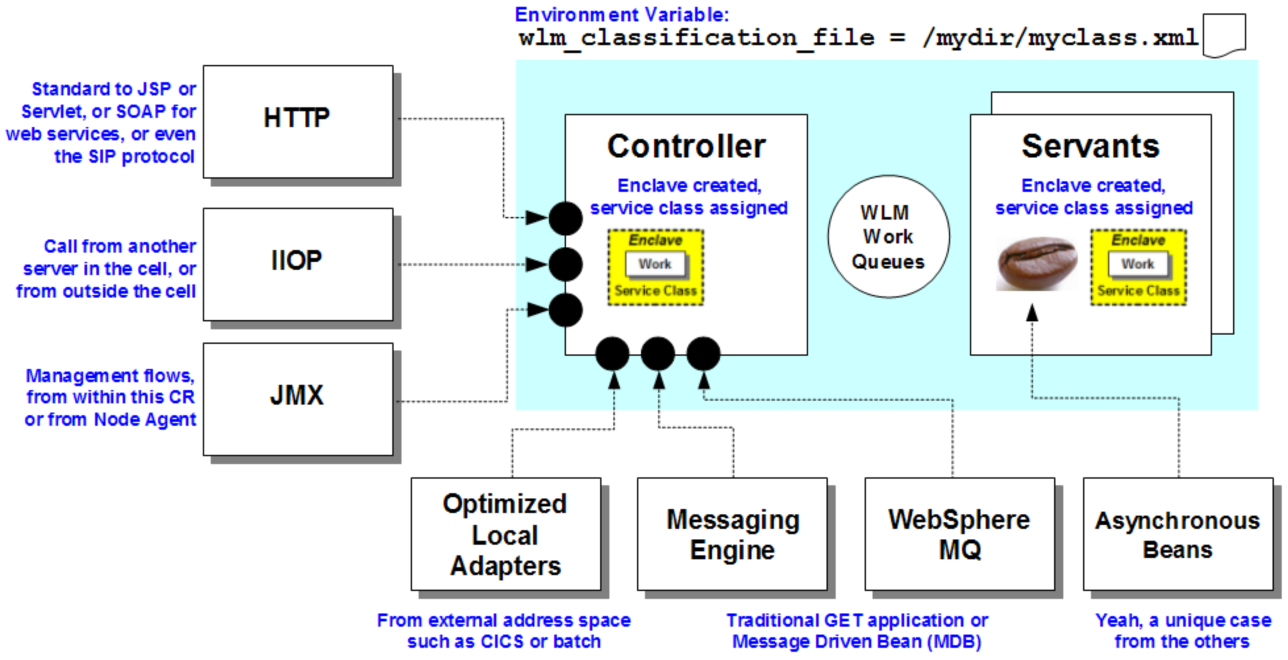
There are a lot of things going on inside the WebSphere address spaces. What we are concerned about here are those things for which WLM enclaves are created. Enclaves are created for anything that gets queued from the controller to run in a servant. Mostly that consists of what we call inbound requests.

An HTTP request arriving over a TCP/IP connection is an inbound request. An IIOP request received over TCP/IP or locally from another server is too. An inbound request could be a message delivered from MQ or a Messaging Engine (ME) in the Control Region Adjunct (CRA) to drive an MDB (Message Driven Bean).

An administrative request that will drive a JMX MBean will also have an enclave created for it. Some of these may be generated internally from within the controller; others will come from external sources like a Node Agent.

Recently we have added the Optimized Local Adapter (OLA) as a source for inbound requests that can generate an enclave.

It is also possible for an enclave to be generated within the servant. This is the case for Asynchronous Beans, which we will discuss separately in a later section.


Environment Variable:
wlm_classification_file = /mydir/myclass.xml
Standard to JSP or Servlet, or SOAP for web services, or even the SIP protocol
HTTP
Call from another server in the cell, or from outside the cell
IIOP
Management flows, from within this CR or from Node Agent
JMX
Controller
Enclave created, service class assigned
Enclave
Work
Service Class
WLM Work Queues
Servants
Enclave created, service class assigned
Enclave
Work
Service Class
Optimized Local Adapters
Messaging Engine
WebSphere MQ
Asynchronous Beans
From external address space such as CICS or batch
Traditional GET application or Message Driven Bean (MDB)
Yeah, a unique case from the others

The classification process

How it works

To classify a request with WLM, WebSphere provides the three values described earlier: Subsystem Type (always 'CB'), Collection Name (the short cluster name), and a transaction class. The first two of these are always the same for every request classified by a particular server. The only thing WebSphere needs to determine for each request is the transaction class.

If there is no classification XML file configured for the server, then a transaction class name is not provided to WLM. You can define a classification XML file by setting the variable wlm_classification_file to the path and file name of the XML file WebSphere should use. We will discuss the specific contents of this file a little later. For now just remember that WebSphere examines the contents of the XML file and the attributes of the request being classified to determine what transaction class to use for this request.

After WebSphere has decided on a transaction class name, WLM is called to classify the request. WLM looks through the classification rules defined in the currently active policy and makes the best match it can.

Default Behavior

Often customers will define rules with the cluster name and transaction class name specified for every combination they expect. However, if a request comes along that does not have a transaction class defined, or gets set to an unexpected value then no rule will match. This will cause the default values for the subsystem type to be used. Because this should represent unexpected work, the default is often set to a service class with a Discretionary goal.

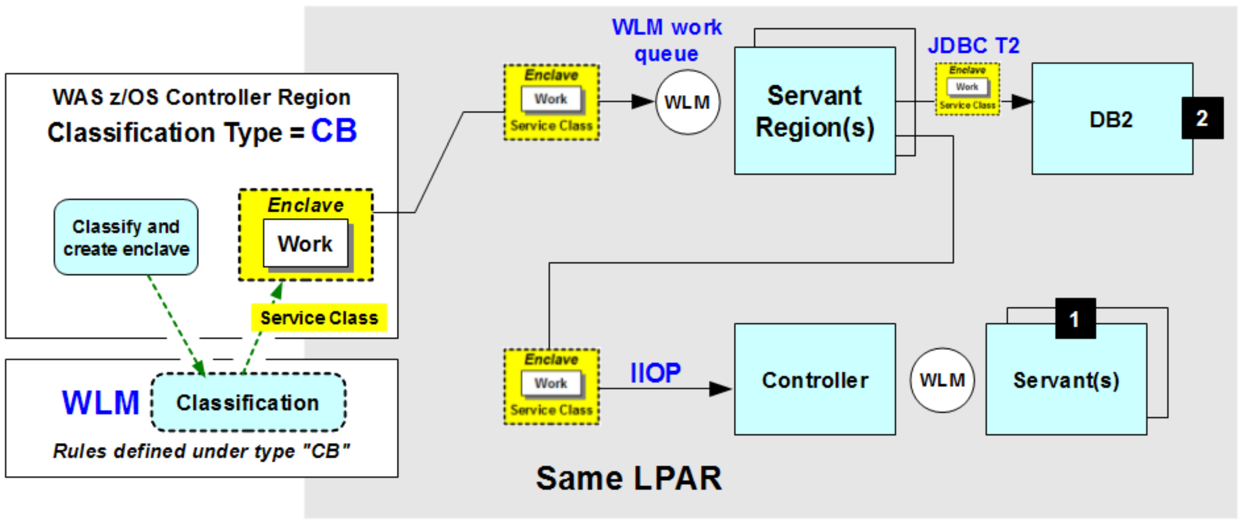
Therefore, one reason for very slow response times might be that that work is not being classified the way you think. This can result in WLM treating the work as very unimportant. Later in this paper we will look at ways to monitor how work is being classified to ensure WebSphere and WLM are handling requests the way you expected.

Enclave Propagation

Suppose an application request is dispatched in a servant region and the application makes a call to an EJB which is installed in another server on the same z/OS image. WebSphere will flow an IIOP request between the two servers to drive the EJB. The enclave will be carried to the second server as part of the flow. In this case the controller for the second server will not create its own enclave. Instead it will use the enclave created by the first server.

This can be good because it causes work in the second server to be managed according to the importance of the original request that started things. If the original request is important, perhaps the work done on its behalf in another server should be important too. Furthermore, CPU time used by the dispatched request in both servers will be reported together for the enclave.

On the other hand, if the second server just has servant regions to support running service classes it expects, work showing up in a different service class can cause problems. Because this feature can sometimes cause problems, there is an option (protocol_iiop_local_propagate_wlm_enclave) to turn it off.



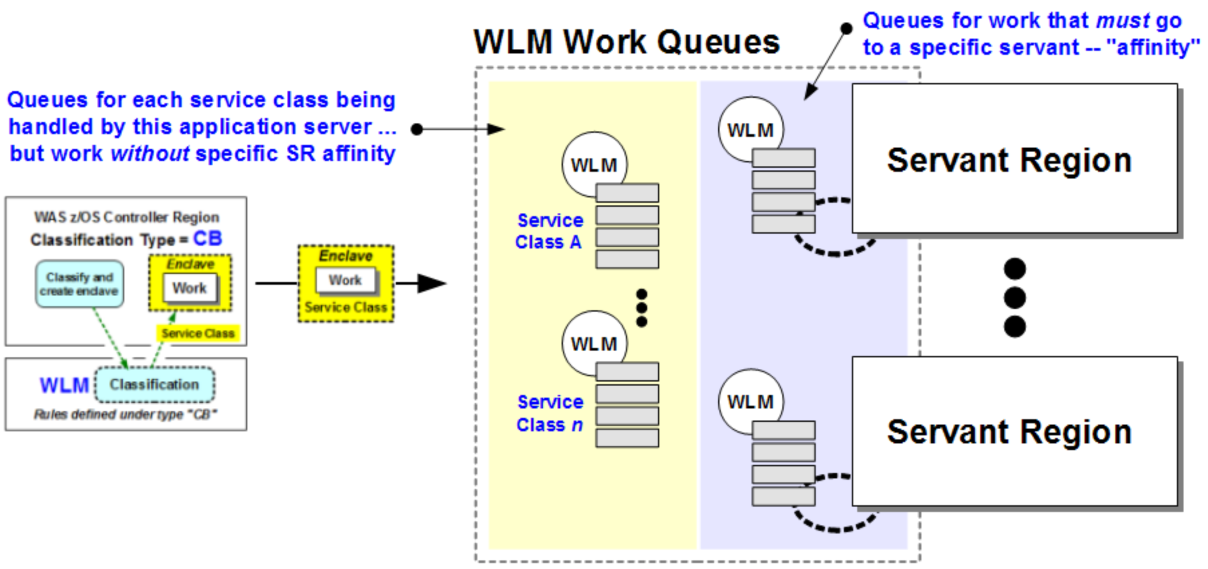
WebSphere Application Server and WLM

How WLM picks a servant region

The WebSphere code in the controller calls a WLM interface to queue a request to be dispatched in a servant region. WebSphere provides two hints to WLM to help decide which servant region should receive the request. The first hint is the WLM service class associated with the enclave that was created for this request. The second hint is an affinity. To understand what WLM does with these hints, first we need to look at the queues WLM keeps between the controller and the servants.

WLM work queues

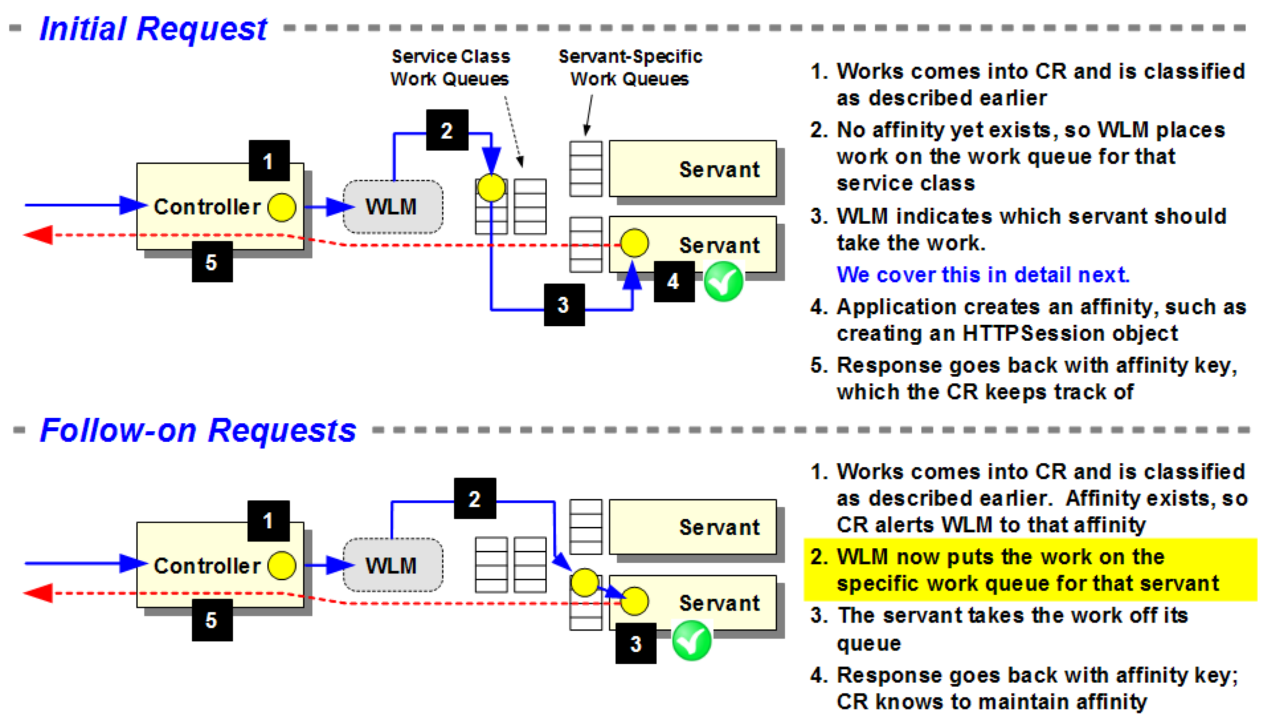
The picture we always draw of the WebSphere server on z/OS shows a single queue between the controller and servants. Really there are multiple queues. Each servant region has its own queue for work that must run in that servant region. Additionally, there are separate queues for each service class being handled by this server. Servant regions are 'bound' to a service class and will, generally, take work from the queue for that specific servant region and then from the queue of the service class to which it is bound.


WLM Work Queues
Queues for work that *must* go to a specific servant -- "affinity"
Queues for each service class being handled by this application server ... but work *without* specific SR affinity
WAS z/OS Controller Region
Classification Type = CB
Classify and create enclave
Enclave
Work
Service Class
WLM
Classification
Rules defined under type "CB"
Enclave
Work
Service Class
WLM
Service Class A
WLM
Service Class *n*
WLM
WLM
Servant Region
Servant Region

Work that has an affinity

An affinity is created by the application. For example, a servlet may create an HTTPSession object, such as a 'shopping cart'. A 'key' to this session object is returned as part of the response to the HTTP request that caused the servlet to be dispatched. A subsequent HTTP request from the same client (e.g. the user clicked on something in the browser) will contain this key. This helps the server dispatch another request from the same user into the servant where his shopping cart is being kept in memory. To do this the controller uses the key to determine in which servant region the HTTPSession object exists. When calling WLM to queue the request, WebSphere will indicate that the request has an affinity to that servant region.

This causes WLM to place the request on the queue for that specific servant region.



Initial Request

1. Works comes into CR and is classified as described earlier
2. No affinity yet exists, so WLM places work on the work queue for that service class
3. WLM indicates which servant should take the work.
   We cover this in detail next.
4. Application creates an affinity, such as creating an HTTPSession object
5. Response goes back with affinity key, which the CR keeps track of

Follow-on Requests

1. Works comes into CR and is classified as described earlier. Affinity exists, so CR alerts WLM to that affinity
2. WLM now puts the work on the specific work queue for that servant
3. The servant takes the work off its queue
4. Response goes back with affinity key; CR knows to maintain affinity

Work that has no affinity

When the request does not have an affinity to a particular servant region, then WLM gets to choose. Since there is no affinity, the only clue WLM has is the service class associated with the request. How WLM uses this information depends on how the WLM application environment was defined.

WebSphere defines the application environment to use either the "First Available" algorithm or the "Round Robin" algorithm. Which one depends on the setting of the WebSphere configuration variable *wlm_stateful_session_placement_on* (Note that despite the name, this variable has nothing to do with stateful session EJBs; it works the same way for any request that does not have affinity to a particular servant region). This variable defaults to zero which means WLM is told to use the "First Available" algorithm. Setting the variable to one tells WebSphere to define the application environment to use the "Round Robin" algorithm. How do these two algorithms work?

First Available

Remember that WLM groups requests by service class. Therefore the servant regions associated with a server will be grouped together and each group ‘bound’ to a particular service class. WLM determines which service class is associated with the enclave for the request. This determines the set of servant regions which can process the work. WLM may choose to give the request to an idle dispatch thread in one of those servant regions. If all threads in the servant regions bound to the service class are busy the work will be placed in a queue. The threads in the group of servant regions will take work from the queue.

Round Robin

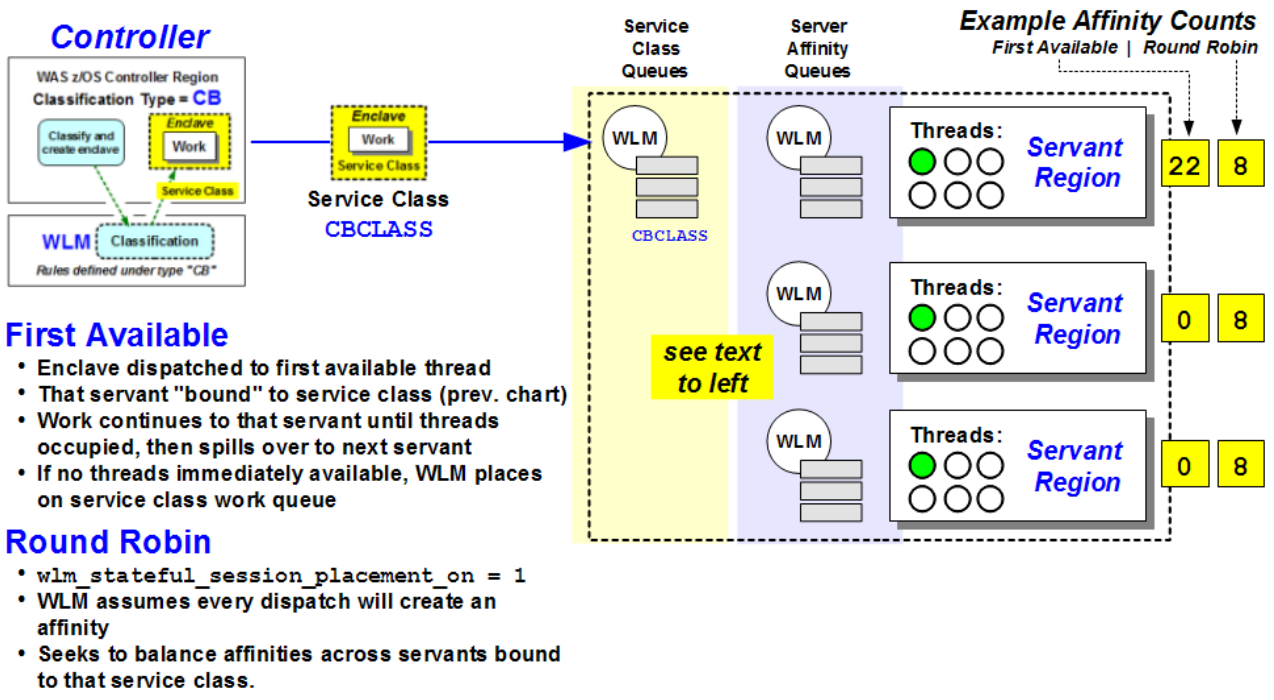
This algorithm is really only kind of like a round robin. What WLM is really doing is trying to keep the affinity count even across the servant regions handling a particular service class. To understand this, first remember that WLM groups the servant regions based on the service class of the work they are handling. For example, if a server is processing two different service classes of work and the application environment is defined to have 4 servant regions WLM will assign two servant regions to each service class.

For this algorithm WLM assumes that any request which is dispatched without an affinity is going to create a new affinity. WLM wants to keep the affinity count the same across servants handling the same service class. In our example, suppose the affinity counts for two of the servants handling a particular service class are 45 and 47. WLM will assign a request that is queued without affinity to the first of these servant regions, hoping to bring that servant’s affinity count up by one.

The general effect of this algorithm is to balance work evenly across a set of servant regions. However, it can have unexpected consequences in some situations. For example, if a servant region abends (for any reason) a new servant region will be started to replace it. The new servant region will have an affinity count of zero. Therefore, for a while, all requests without affinity will go to the new servant as WLM attempts to help it ‘catch up’ with other servants.

Another odd behavior that may be observed with this algorithm is WLM apparently ignoring a servant region. This is usually caused by the server dispatching more service classes than you think it does. Suppose again that you have four servant regions defined. If all the work received is in the same service class, then the Round Robin algorithm will spread the work across all four servant regions, keeping the affinity count as even as possible. However, if even a few requests without affinity (or even one) have been processed with a different service class, WLM will divide the servants evenly among the service classes (generally, see the discussion on ae_spreadmin coming up in a few pages). This will cause two of the four servant regions to be bound to this other service class. The Round Robin of the bulk of the requests will then be balanced between the two other servant regions instead of using all four.

Yet another odd behavior occurs with low utilization and no actual affinities being created. WLM assumes each request without affinity will create one. However if the requests aren't actually creating affinities then no matter how WLM routes the work the affinity counts will stay at zero. If requests aren't coming in very fast then WLM might route them all to one servant region, trying desperately to get its affinity count to increment to one.



First Available

- Enclave dispatched to first available thread
- That servant "bound" to service class (prev. chart)
- Work continues to that servant until threads occupied, then spills over to next servant
- If no threads immediately available, WLM places on service class work queue

Round Robin

- `wlm_stateful_session_placement_on = 1`
- WLM assumes every dispatch will create an affinity
- Seeks to balance affinities across servants bound to that service class.

How threads are managed in a servant region

As we have mentioned earlier, WLM binds each servant region to a particular service class. The dispatch threads in the servant region receive work for the service class to which it is bound. However, a request with affinity to a servant region might have a different service class than the one to which the servant is bound. That could cause a mixture of service classes in the servant region. How does WLM handle that? And how does WLM manage threads in the servant region that do not have an associated enclave?

Enclave threads

The dispatch threads are generally the only threads in a servant region which have associated enclaves (except the Async Bean thread pool which we will discuss a little later). A thread with an associated enclave is managed to the goals of the service class of the enclave. This means that if different dispatch threads in the same servant region are running requests that have different service classes WLM will manage the threads differently. A thread associated with a higher priority service class might receive more CPU than another thread in the same address space whose associated enclave has a service class that is lower in priority.

Remember that WLM tries to keep all the threads in a servant region dispatching requests with the same service class and thus the same goals. However, an affinity requirement could force a request into a servant region bound to a different service class than the one associated with the request. WLM will manage each dispatch thread to the service class of the request being dispatched.

Non-enclave threads

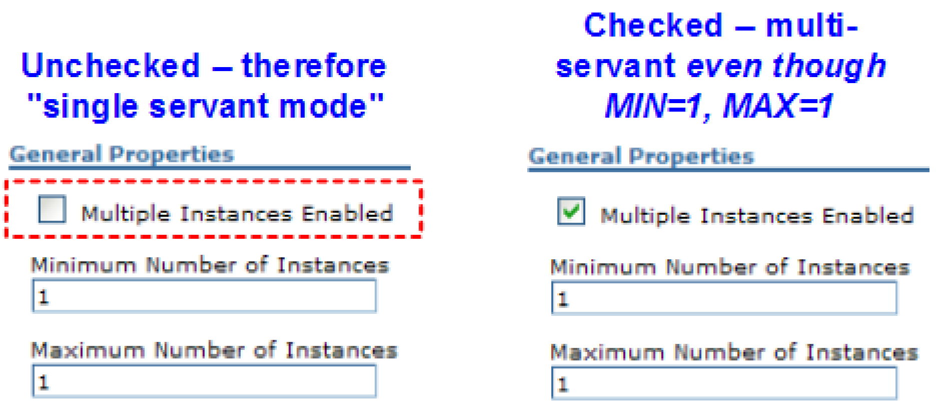
What about all those threads in the servant that do not have an associated enclave? Since WLM knows what service class this servant region is supposed to be processing, it knows how important the work done by supporting tasks should be (for example, garbage collection threads). WLM will manage these threads to the same goals as the service class to which the servant region is bound.

However, this can cause some problems for WLM resulting in servant regions that stay swapped in or swapped out incorrectly. And it might be nice to have a little more control over how these threads are managed. To address these issues, in z/OS 1.12 WLM introduced the ManageNonEnclaveWork parameter in the IEAOPTxx member of SYS1.PARMLIB. When set to 'NO' (the default) things work as we have described above.

However, if you set this parameter to YES non-enclave threads in the servant region will be managed to the goals of the servant region started task. This goal normally only applies to non-enclave threads during initialization; that is, until the servant region is bound to a service class queue. This new option applies the goal based on the started task classification to all the non-enclave threads for the life of the servant. If you decide to set ManageNonEnclaveWork to YES you should carefully consider the goals from the started task classification and whether that is appropriate for the life of the servant. The goals of started tasks are typically higher than WebSphere applications.

Single Servant mode

WLM manages much of this magic by separating work with different service classes into different servant regions. However, it is possible to configure the application environment to have only a single servant region. This is done by un-checking the “Multiple Instances Enabled” checkbox in the admin console just above where you configure the minimum/maximum number of servants. Single servant is a special configuration choice and is not the same as specifying ‘one’ as the value for the minimum and maximum number of servants.


Unchecked – therefore "single servant mode"
General Properties
Multiple Instances Enabled
Minimum Number of Instances
1
Maximum Number of Instances
1
Checked – multi-servant *even though MIN=1, MAX=1*
General Properties
Multiple Instances Enabled
Minimum Number of Instances
1
Maximum Number of Instances
1

So what does WLM do if work is queued with different service classes and the application environment is defined as ‘single servant’? WLM has no choice. It has to mingle work with different service classes in the same servant region. This is essentially the same situation as work with affinity being placed in a servant region bound to a different service class. WLM manages each dispatch thread to the goals of the service class of the work currently dispatched on that thread. The difference in this case is WLM will allow work with different service classes and no affinity to mix together in the same servant region – because it has no other choice.

Threads without an enclave (garbage collection threads, etc.) are managed to the goals of the highest goal of any enclave this servant has joined or the goal of the servant started task (as noted in the previous section).

By contrast, specifying a minimum and maximum of one servant region can cause problems. In this case, which is not single-servant, WLM binds each servant to a service class. Therefore the one servant region the configuration allows will be bound to whatever service class of work is being processed by the server. Any work that is classified into a different service class will wait in the WLM work queue until timed out by the controller. The configuration does not allow WLM to start another servant and in the min=max=1 configuration WLM will not mix service classes in the servant. If you are sure you only want one servant region, it is probably safer to configure the server as single-servant.

How Many Servant Regions Do I Get?

After the last section that would seem to be pretty easy. You tell WLM the minimum and maximum number and WLM will start the minimum and possibly grow to the maximum if WLM thinks it needs them to meet goals. Not so fast...

If all your work is in one service class then the simple explanation still works. But if you have more than one service class involved, it gets complicated. Remember that with multiple servant regions WLM has to decide how to divide up the servant regions among the different service classes. This seems easy, but WLM does not know up-front how many service classes it will have. It only knows the server is classifying work to a particular service class when work shows up for it.

Consider an example. We configure a servant with minSRs of 3. You start the server and WebSphere puts a dummy piece of work into the queue to get things started. This work is classified to some service class (say ClassA). WLM starts one servant and binds it to ClassA. But we also tell WLM we want three servants. So WLM will start the next two servants (probably one by one, but not necessarily). Are those servant regions also bound to ClassA? For now we will say 'yes'. Now we have three servant regions, all bound to ClassA.

But then real work comes in and some if it gets classified into another service class (ClassB). WLM has to reassign at least one servant region from ClassA to ClassB to handle the other service class. Do we end up with one servant bound to ClassB and two bound to ClassA? Or perhaps WLM binds one servant to ClassB, leaves one bound to ClassA and makes the third servant 'unbound' so it doesn't handle any work. WLM holds it in reserve for a possible third service class or maybe if WLM gets behind with ClassA or ClassB work it can assign the servant to one of those service classes.

It turns out to all depend on a parameter called AE_SPREADMIN that WebSphere specifies to WLM. For all releases of WebSphere prior to Version 8 this parameter was always hard-coded as 'YES'. This means that WLM will distribute the servant regions evenly across the service classes. If AE_SPREADMIN is set to 'NO' then WLM can distribute service classes as it sees fit in order to meet goals.

By specifying 'YES' WebSphere is telling WLM to balance the servants among the service classes evenly. WLM will bind one servant to ClassA and one servant to ClassB and hold the third servant as 'unbound'.

Starting in Version 8 you can specify the wlm_ae_spreadmin environment variable which will affect the value WebSphere passes to WLM on this parameter. If you set it to '1' this tells WebSphere to set AE_SPREADMIN=YES and set to '0' tells WebSphere to set AE_SPREADMIN=NO.

There is one more interesting thing to note about AE_SPREADMIN=YES. If the number of service classes does not divide evenly into the number of servant regions (minSRs) then WLM might decide it does not actually need all the servant regions you asked for. Basically, WLM will not maintain the minimum number of servants specified if it does not need them all to balance the number of service classes.

Suppose in our example with three servant regions and two service classes one of the servant regions abends. Suppose it is the servant bound to ClassA. WLM will take the unbound servant region and assign it to ClassA. Now it has one servant for ClassA and one servant for ClassB and no unbound servants. The minimum is three, but if WLM starts it the servant will just be unbound, which means WLM does not really need it right now. Therefore, WLM will not start it, even though it is required to keep the minimum you specified.

Seem odd to you? WLM agrees. They took FIN APAR OA35546 to eventually change this behavior and always honor the minimum number of servants. We have also seen odd behavior when work is allowed to be dispatched while the minimum number of servants are starting. If the work shows up at the right time and in different service classes WLM might never actually even start the minimum number. This can cause problems if other things are waiting for the minimum number to start. The best solution to this is to configure the server to not take any work at all until the minimum number of servants are started.

Just one more thing to be aware of as you consider placing work in different service classes into one server. When you get to Version 8 you might also consider the consequences of changing the wlm_ae_spreadmin environment variable to let WLM distribute servant regions between service classes as it wants to, rather than forcing WLM to try to keep them evenly balanced.

Dynamically Changing minSRs and maxSRs

In Version 7 we introduced the Modify command WLM_MIN_MAX. You can use this to dynamically change the values given to WLM for the minimum and maximum number of servant regions. This command has no effect if your server is running in single-servant mode.

You specify both the new minimum and new maximum value on the command, like this:

```
MODIFY server,WLM_MIN_MAX=(2,4)
```

This sets the minimum and maximum values to two and four respectively. :There is also a display command you can use to verify the current values are what you expect:

```
MODIFY server,DISPLAY,WLM
```

Remember that dynamically changed values only stay in effect until the controller is restarted.

So how does this really work? If you change the minimum and maximum values will WLM immediately react and add or remove servant regions? Maybe. It depends which value you changed and whether you made it bigger or smaller than the current value.

Generally speaking, if you increase the minimum value WLM will try to add a new servant region as soon as it can in order to maintain the minimum number you specified.

If you decrease the minimum value below the current value, then WLM just has an extra servant region now and it may or may not get rid of it depending on whether or not it thinks it needs it. If there is session data pinned to all the servant regions then WLM can't get rid of any of them until the affinities count goes to zero. So lowering the minimum may have no immediately visible affect at all.

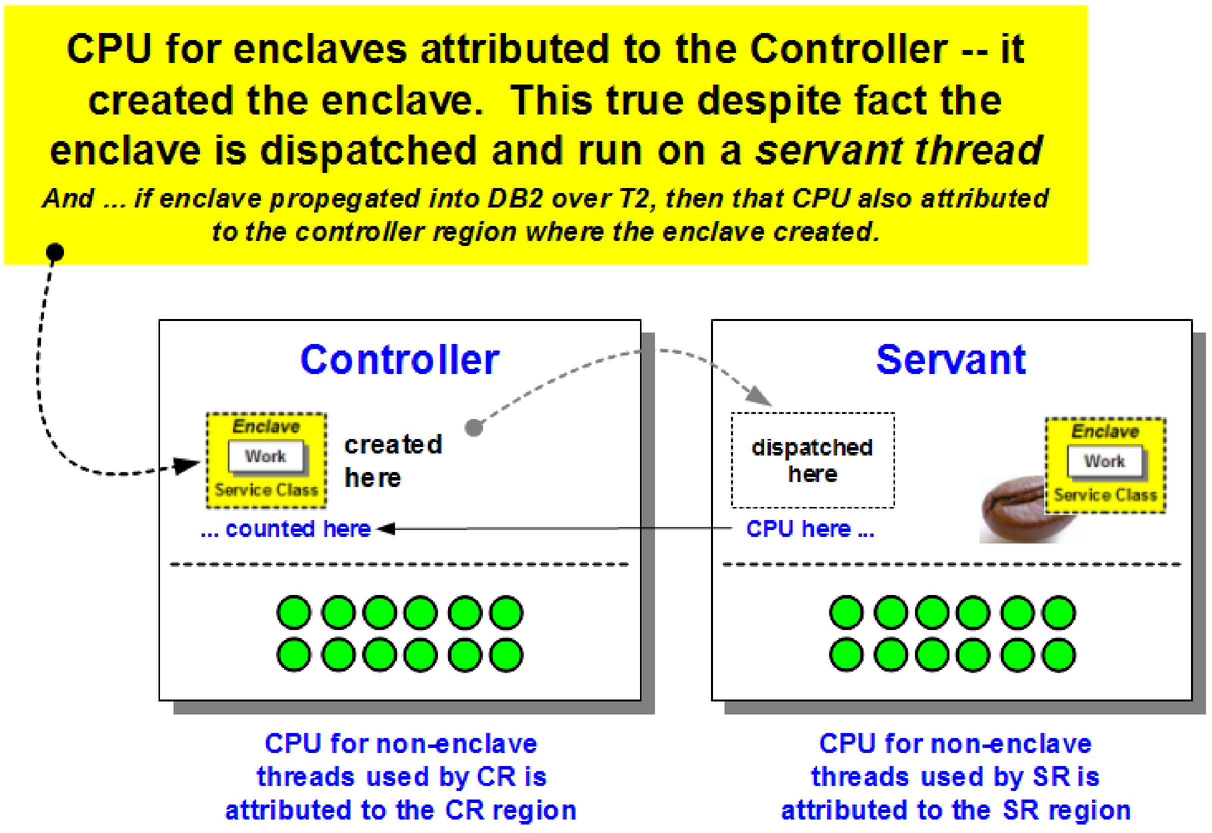
If you increase the maximum value you are just giving WLM extra room in case it needs it. Unless WLM is desperate for a new servant region because it is already up against the old maximum value it is unlikely to create a new one right away just because you gave it permission.

And finally, if you decrease the maximum value, even below the current value, WLM may not react to immediately remove a servant region. Again, depending on pinned session data, it might not even be allowed to remove a servant if it wanted to.

The most likely scenario where you would want to dynamically change the minimum and maximum would be in a configuration where min and max are configured to the same value and you feel another servant region would be useful. Dynamically changing the min and max values (say from 1,1 to 2,2) will likely force WLM to create another servant region. Depending on configuration (e.g. round-robin or not) WLM may use the new servant or not. In this scenario you have chosen to manually control the number of servants yourself instead of setting a maximum greater than the minimum and allowing WLM to create new servants when it thinks it needs them. Depending on your system resources and your own requirements this may or may not be a better solution.

Reporting CPU usage

With a WebSphere server started and processing requests, you might want to know how much CPU is being used. If you have RMF, you can look at the RMF reports to see what is going on. We will look at this a bit later, but while we are talking about enclaves we should probably talk about how they show up in RMF.


CPU for enclaves attributed to the Controller -- it created the enclave. This true despite fact the enclave is dispatched and run on a *servant thread*
And ... if enclave propegated into DB2 over T2, then that CPU also attributed to the controller region where the enclave created.
Controller
Enclave
Work
Service Class
created here
... counted here
Servant
dispatched here
Enclave
Work
Service Class
CPU here ...
CPU for non-enclave threads used by CR is attributed to the CR region
CPU for non-enclave threads used by SR is attributed to the SR region

Enclaves owned by the controller

Application requests that are dispatched in the servant region run with an enclave associated with the dispatch thread. You might think that running in the servant region would cause the CPU used to be reported as part of CPU consumed by the servant region. However, since the work is done under an enclave, the CPU used is reported as enclave CPU time from the address space that created the enclave.

For normal requests, an enclave is created by the controller. Therefore, to view CPU used by application requests you need to look at enclave CPU time in the controller. It gets more complicated if enclaves are propagated, as discussed above.

Non-enclave threads

Other threads in the servant region that are not used to dispatch requests, and this includes any threads created by an application, do not have an associated enclave. CPU used by these threads will be reported by RMF as CPU used by the servant address space.

Reporting RMF Queued Wait Time

RMF will also report the time work spent waiting on the queue. That seems fairly straight forward. You would think wait time would start when the work is put on the queue between the controller and the servants and stop when the work was picked up by a servant for dispatch. But that would be far too easy to understand and not nearly complicated enough.

The beginning of the queue time is always a timestamp saved by WebSphere when the request first arrives in the server. Generally this is when TCP/IP gives us the request. For large requests it might be the time when the last fragment is received. So it gets complicated already.

How the end of the queue time is determined depends on a value supplied to WLM when the enclave is created. This parameter is called EXSTARTDEFER. If WebSphere specifies 'YES' then queue time ends when the work is picked up by a servant region and the enclave is actually put on the thread. If WebSphere specifies 'NO' then queue time ends when the enclave is created. In this second case the queue time measured is actually the time it took the controller region to process the request and create the enclave. If the controller is backed up enough there might be some measurable time spent waiting (on a queue!) in the controller.

In all likelihood it is the first value (YES) that you would prefer to use. So what does WebSphere specify? Well, it depends. Historically it was very complicated and some times we specified 'YES' and sometimes we specified 'NO'. To clear things up we took APAR PM24445. This allows you to set the environment variable wlm_enclavecreate_exstartdefer and tell WebSphere what value to pass to WLM. The default is 'YES' and you should probably leave it that way.

Asynchronous Beans

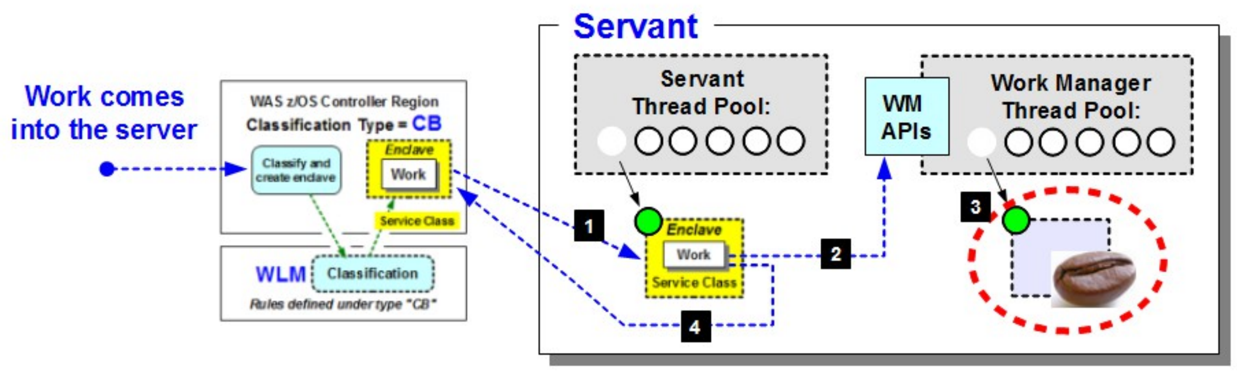
Several times earlier in this paper we have deferred the discussion of Asynchronous beans and indicated that they are often special exceptions. In this section we will finally explain what they are and how they behave differently from ‘normal’ requests.

What are they?

Asynchronous Beans are an IBM extension to the Java EE programming model. For an application that wants to run work asynchronously to the dispatch of a request, or that wants a long running thread that is present for the life of a server, asynchronous beans are the way to go. This is the preferred way for an application to get its own thread inside the server. Just using the Java APIs to create a thread is usually a bad idea.

When an application wants to schedule an asynchronous bean it needs to use a work manager. For this paper we will assume you are using the WLM enabled work manager.

Remember that no matter how the asynchronous work gets classified with WLM it ALWAYS runs in the servant region where it was scheduled. If you have one servant bound to Service Class A and another bound to Service Class B and a servlet in the first servant starts an async bean that winds up in Service Class B, it will run in that same first servlet. Asynchronous work is not routed. It runs in the threads owned by the work manager accessed by the application.


Work comes into the server
WAS z/OS Controller Region
Classification Type = CB
Classify and create enclave
Enclave
Work
Service Class
WLM
Classification
Rules defined under type "CB"
Servant
Servant Thread Pool:
WM APIs
Work Manager Thread Pool:
1
2
3
4
Enclave
Work
Service Class

How are they managed?

When an asynchronous bean is dispatched the thread it runs on will join an enclave. The thread will be managed to the goals of the service class associated with the enclave. The report class associated with the enclave will be used to report on it. What enclave is used? How does it get created? It depends how the asynchronous bean was scheduled.

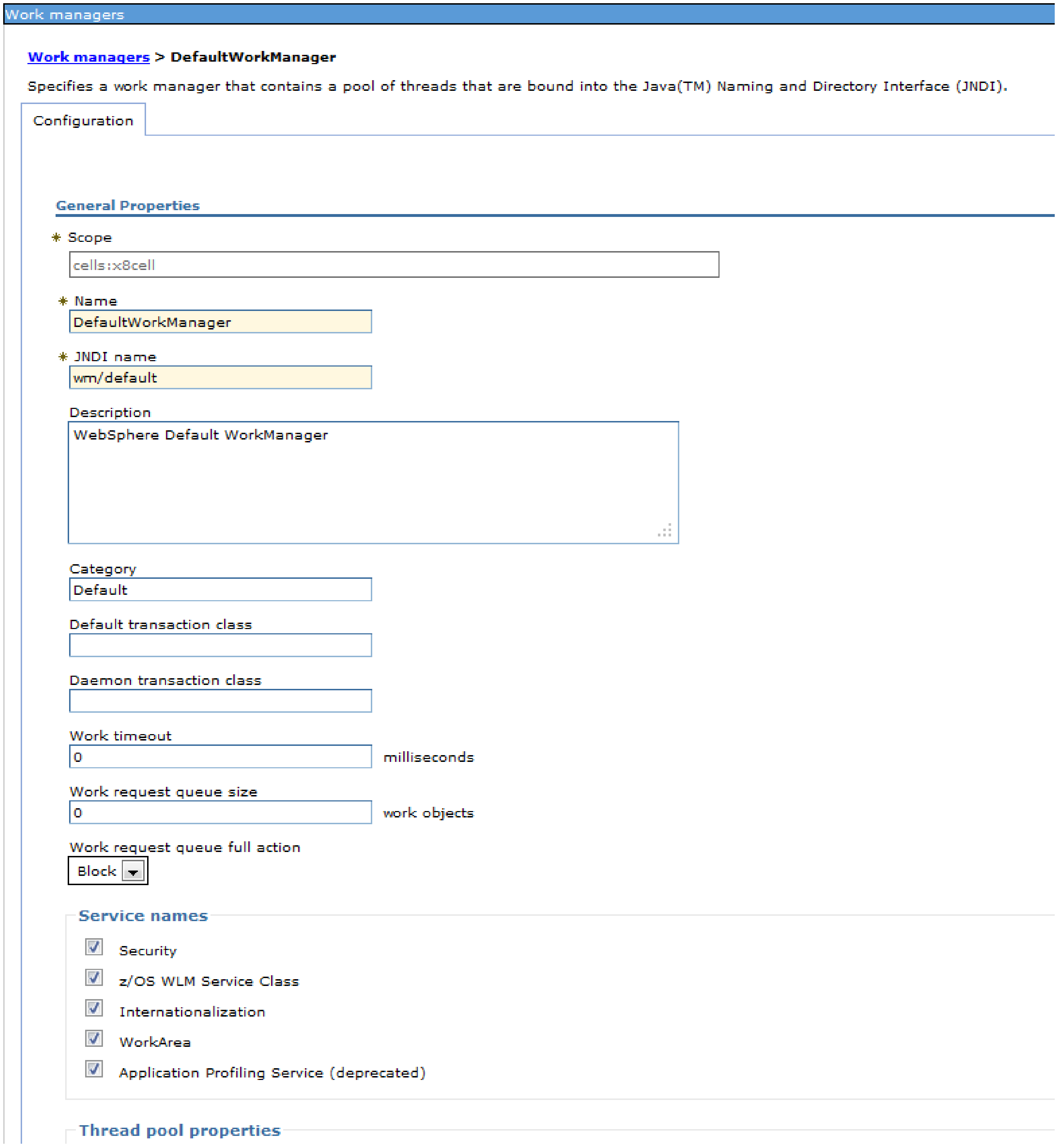
Long running Asynchronous Beans

Asynchronous beans are usually used to run work in parallel with a dispatched request or to run something at a later time. However, sometimes you may want to establish a permanent or long running thread. For this situation you can specify 'true' for the Boolean isDaemon parameter on the startWork Workmanager API.

This causes the work manager to create a new thread instead of using one from the work manager pool. This avoids tying up a thread from the pool for something you expect to be around for a long time. A new enclave is created for this asynchronous work. But how is it classified with WLM? The configuration of the work manager allows you specify a Daemon transaction class. This transaction class will be used, along with the cluster name and the 'CB' subsystem type, to classify the work using the rules defined in WLM. The enclave created will be used by the thread which dispatches the asynchronous bean. If you do not specify a Daemon transaction class for the work manager, the value ASYNCDMN is used.

Work Manager Configuration

Before we get into some other scenarios we need to look at the configuration panel for the work manager. We have already mentioned the Daemon Transaction class. You can also specify a default Transaction Class. We will see how that is used soon. There is also a section labelled "Service Names". In this section is a checkbox labelled "z/OS WLM Service Class". The effect of checking or unchecking this box is, as you might guess, complicated and depends on what the application is doing. The only easy thing to remember is that the checkbox has no effect on work scheduled with is-Daemon set to 'true'. That work always creates a new enclave using the Daemon Transaction class. Here is a shot of the work manager configuration screen for reference.



Work managers

Work managers > **DefaultWorkManager**

Specifies a work manager that contains a pool of threads that are bound into the Java(TM) Naming and Directory Interface (JNDI).

Configuration

General Properties

* Scope
cells:x8cell

* Name
DefaultWorkManager

* JNDI name
wm/default

Description
WebSphere Default WorkManager

Category
Default

Default transaction class

Daemon transaction class

Work timeout
0 milliseconds

Work request queue size
0 work objects

Work request queue full action
Block

Service names

- ✓ Security
- ✓ z/OS WLM Service Class
- ✓ Internationalization
- ✓ WorkArea
- ✓ Application Profiling Service (deprecated)

Thread pool properties

Asynchronous Beans scheduled without an execution context and WLM Service enabled

Another case to consider involves an asynchronous bean scheduled by an application running on a thread with an associated enclave, but without specifying an execution context to the work manager (we will get to execution contexts in a moment).

First consider the case where the WLM service is enabled (check box is checked). In this case the enclave associated with the original dispatched request will be used by the work manager. The dispatch thread in the work manager pool will be joined to the enclave before the asynchronous bean is dispatched. When the asynchronous bean dispatch is complete the thread will leave the enclave.

Ideally the original request will stick around and wait for the asynchronous bean to complete before it returns. This pattern is useful when an application wants to do several things in parallel, but needs to wait for them all to complete before responding to its client. In this case WebSphere does not need to create a new enclave because the asynchronous bean is not really a separate piece of work, but is instead just an asynchronous piece of the original dispatched request.

However, it is possible the application could complete and return to the client before the asynchronous bean completes dispatch. Normally WebSphere will delete the enclave associated with the dispatched request when the request finishes. How can we prevent that from happening if there are still asynchronous beans running that use the same enclave?

WLM provides an API to Register as a user of an enclave. The work manager will register with WLM for the dispatched request enclave as part of scheduling the work. This prevents WLM from deleting the enclave until the work manager Deregisters. The deregister will not happen until after the asynchronous bean has finished and the thread has left the enclave. Therefore, even if the application request dispatch finishes and the controller deletes the enclave, it will not really go away until any asynchronous beans using that enclave have completed.

Asynchronous Beans scheduled without an execution context and WLM Service NOT enabled

But what if the WLM service is not enabled for the work manager? In this case a new enclave will be created to run the asynchronous work. The “Default Transaction Class” from the work manager configuration will be used to create the enclave. The enclave will be deleted when the work finishes. If no Default Transaction class is specified in the work manager configuration a value of ASYNCBN will be used.

Asynchronous Beans scheduled with an execution context

When an application defines work that it wants a work manager to execute as an asynchronous bean it can specify a execution context on the startWork API. The application is allowed to create the execution context and then, maybe later, use it to schedule the asynchronous work. What happens depends on the configuration of the work manager used to create the execution context and on the configuration of the work manager used to run the asynchronous work. Yes, it could be a different work manager, apparently just to make things confusing.

We thus have four scenarios depending on whether the WLM service is enabled or not in the two work managers. We will consider all four scenarios and then have a table at the end to summarize. We will start with the easy cases where the two work managers have the same configuration (probably because you used the same work manager to both extract the context and run the work).

Suppose the WLM service is enabled for the work manager(s). When the execution context is created Websphere will look for a WLM enclave associated with the thread. If an enclave is present the classification information (Collection Name, Transaction Class) used to create the enclave will be extracted and placed in the execution context. When the asynchronous work runs a new enclave will be created using the classification information saved in the execution context. The enclave will be deleted when the asynchronous work ends.

But suppose the WLM service is disabled for the work manager(s). Then no information is saved in the execution context when it is created. When the work runs the work manager will create a new enclave using the Default Transaction Class name from the work manager used to run the work.

Now we consider some of the odd cases. Suppose the work manager used to create the execution context has the WLM service enabled and the work manager used to run the asynchronous work has the WLM service disabled. Since the WLM service is enabled when we create the execution context we extract the classification information from the enclave on the thread. However, when we run the asynchronous work that work manager has the WLM service disabled so it ignores the classification information in the execution context and creates a new enclave using the Default Transaction Class from that work manager.

And finally, what if the work manager used to create the execution context has the WLM service disabled and the work manager used to run the asynchronous work has the WLM service enabled? The execution context will not contain any classification information so the work manager that runs the work will create a new enclave using the Default Transaction Class from that work manager.

	WLM Service enabled to run	**WLM service disabled to run**
WLM service enabled for extract	Determine classification information from current enclave and use it to create a new enclave	Determine classification information from current enclave but ignore it and use the Default Transaction class to create a new enclave
WLM service disabled for extract	No classification information used and Default Transaction class used to create a new enclave	No classification information used and Default Transaction class used to create a new enclave

Table 1: Enclave Creation Rules for Async Work with an Execution Context

What if we try to determine classification information from the current enclave but there is no current enclave? That can happen if the application is not running on a dispatch thread, for example during application initialization. In this case the Default Transaction Class from the configuration of the work manager used to run the work will be used to create a new enclave.

Some other things to keep in mind about asynchronous work

Remember that these enclaves are created by the servant region and will show up there in RMF. Enclaves created for the dispatch of normal requests are created by the controller and show up there in RMF.

Also note that in Version 8 (8.0.0.1 to be specific) asynchronous work can now write SMF 120-9 records. This record contains information that can help you determine whether a new enclave was created for the work and, if so, what classification information was used to create it.

Hopefully now you understand why we chose to leave asynchronous beans for later.

WLMless queuing

One of the many WebSphere configuration variables is *server_use_wlm_to_queue_work*. So far we've only talked about how WebSphere uses WLM to manage work. What could this variable mean? Which value is the best for you? Like most IBM answers, it depends.

What it does

Specifying *server_use_wlm_to_queue_work* as zero (false) causes WebSphere to use its own internal queuing mechanism to get work to the servant regions instead of using WLM. WebSphere has its own queues and the dispatch threads call WebSphere code instead of WLM when they are ready for work. A WLM application environment is still used to start the servants and restart them if they end unexpectedly. However, as far as WLM is concerned, the servants are not getting any work to do.

Because WLM does not see any work flowing to the servant regions it will simply maintain the minimum configured number of servant regions. WLM will never dynamically add an extra servant region above the minimum in order to help meet its goals because it doesn't see any work being executed.

When to use it

This paper has discussed many of the benefits of the interaction between WebSphere and z/OS WLM. Why would you want to let WebSphere manage the queuing of requests instead of WLM?

For applications which are largely stateless, each request can be routed to a servant region based on the goals associated with the request (via classification) and how the servant regions and z/OS in general are doing towards meeting the goals of all the work on the system. Applications which are heavily state driven take away that decision. An application that has state usually requires each subsequent related request to run in the same servant region as the initial request. This binds a series of requests into the same servant region, taking away WLM's ability to choose and balance work.

WLMless queuing recognizes that a request without state is likely to be followed by a long series of requests that will need to run in the same servant. Therefore, the number of servant regions remains fixed and we focus instead on how we spread the initial requests among the servants. WLM provides a way to do that, which we have discussed earlier (controlled by *wlm_stateful_session_placement_on*). With WLMless queuing we pay no attention to the service class of the requests and treat every request equally. Therefore, WLMless queuing works best if all requests to the server have the same goal.

Routing options

There are presently three different algorithms you can choose to spread work across the servant regions when using WLMless queuing. You select the algorithm by setting the variable *server_work_distribution_algorithm*. The default value is zero which selects the Hot Thread algorithm.

Hot Thread

The Hot Thread algorithm simply looks for an available thread. The servant regions are placed in a list. The code starts with the first servant region in the list, looking for an idle dispatch thread. If one is found it gets the work. If no idle threads are found in the first servant region it proceeds to the next servant in the list. If no idle dispatch threads are found in any servant region the request is placed on a global queue and selected by the first available thread. This approach tends to keep all the work in the fewest number of servant regions possible while letting every request get a thread as quickly as possible.

This is probably the best choice for applications which are lightly used and probably don't need more than one servant region except under occasional heavy load.

Round Robin

The second algorithm, configured by setting *server_work_distribution_algorithm* to one, is called Round Robin. This is more like a true Round Robin algorithm than WLM's Round Robin algorithm. Since WLMless queuing does not consider the service class of each request it simply assigns each request to a different servant in a round robin fashion. If a thread is available in the selected servant region it is given the request to process. If no threads are available, the request is placed in a queue dedicated to that servant region. The work will be processed by the first available thread in the selected servant region.

This algorithm is better suited to spreading the work across the all the servant regions you have configured. For requests with a long chain of affinity-driven work this algorithm will spread the affinities across the servant regions. This leads, usually, to a better balance of requests with affinity across the servants. Contrast this with Hot Thread which tends to run everything in the first servant region. Subsequent requests with affinity will be forced to run in the first servant regardless how overloaded that servant becomes.

Hot Robin

The third algorithm you can configure for WLMless queueing is called Hot Robin. Select this algorithm by setting *server_work_distribution_algorithm* to two. The Hot Robin algorithm is sort of a combination of the Hot Thread and Round Robin algorithms. This algorithm is only available in WAS Version 7 at maintenance level 7.0.0.7 and above.

In the Hot Robin algorithm we begin by selecting a servant region in a round robin manner, just as we did in the Round Robin algorithm. If a dispatch thread is available we give the work to the thread. However, if no thread is available to dispatch the request we do not just put the work in the servant queue. Instead we 'spill' to another servant region. Which servant region we spill to is controlled by a second round robin. We look in the first 'spill' servant region for an available thread. If no thread is found we proceed to the next servant region in the list looking for an available thread, just as we did in the Hot Thread algorithm. If no threads are found in any servant region we put the work on a global queue to be selected by the first available thread, just as we did in the Hot Thread algorithm.

The result of all of this is better spreading of work across the servant regions, especially if one servant region gets bogged down for some reason. For example, suppose a problem causes one servant region to slow down while a dump is taken. In the Round Robin algorithm work will continue to pile up in that servant region's queue while it takes the dump. The Hot Robin algorithm will notice the slow-down and spread requests that would 'normally' go to that servant among the other servants. In net you get the benefits of spreading affinity around combined with a tendency to give work without affinity to the servant most able to handle it. Of course once all the threads are busy processing work Hot Robin actually behaves exactly like Hot Thread, putting all the work in the global queue.

Using the Classification XML File

In this section we will talk about the contents of the XML file you can use to classify application requests. This classification results in a transaction class name. As described earlier, the transaction class name will be used as part of the WLM classification process to determine the service class and report class that will be associated with the request when it is dispatched.

What we are not going to do in this section is talk about the required syntax used to create the file. The file contents and format are already very well described in the WebSphere Information Center in the article located here:

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm.websphere.zseries.doc/info/zseries/ae/rrun_wlm_tclass_dtd.html

IIOP Request

The IIOP section of the XML file allows you to classify requests for EJBs that arrive at the server using the CORBA IIOP protocol. You can group together requests from the same .ear file, or requests that are in different .jar files within the same .ear file. Or separate requests for different EJBs inside a .jar file. You can even specify the actual method name used in the request. All this allows you to place requests for different applications or parts within an application into different service classes (because they have different goals) or different report classes (allowing you to monitor them separately).

If you decide to classify requests at the method level, remember that the method name used is the RMI mangled method name. Because an EJB can have multiple methods with the same name but different signatures the method name used for classification is a string that merges the parameter list into the name. You can usually find the mangled name in the generated Stub or Tie.

HTTP Request

The most common way to classify an HTTP request is to use the URI. The URI is the part of the URL after the host name. It is unusual to specify the complete URI for each possible request to an application. Instead wild carding is used to specify the context root for the application. If you want to classify based on different parts of the application, the directory structure that forms the rest of the URI is a good way to do it.

The classification XML file also allows you to classify the request based on the target host and port. This allows you to separate HTTP requests that get to the server in different ways, using different target host names. Note the host name used for classification is retrieved from a header in the HTTP request and, in some unusual circumstances, might not actually match the host name used to reach the server.

SIP Request

There aren’t really any distinguishing characteristics of a SIP request that we can use to classify them with. Just being a SIP request is all we really have. Therefore you can specify a transaction class to be used for all SIP requests.

MDB Request

This clause applies to Message Driven Beans that are driven by messages received from MQ via the Message Listener Port (MLP) listener in the controller. It only applies if the controller can be given a reference to the message. This is a characteristic of persistent, durable message queues. Being able to get a reference to the message allows the controller to take that reference and queue work to be dispatched in the servant. The servant then calls MQ with the message reference and gets the actual message. This is sometimes referred to as ‘Plan A’ MDBs.

By contrast a non-persistent, non-durable queue can not provide a message reference. Only the message itself can be provided. In this case the listener is in the servant and the message is received and processed in the servant region. In this case the ‘Internal’ clause (below) of the WLM classification file is used. This is sometimes referred to as “Plan B’ MDBs.

For Plan A MDBs you can use this clause of the WLM classification file to specify a transaction class for different MDBs. At the highest level you can specify the name of the Message Listener Port used to listen for messages. You can also specify a selector clause which has to match the selector specified in the deployment descriptor for the MDB. This allows you to specify different WLM transaction classes for different MDBs.

If you use MQ and Activation Specifications the messages are routed through the Control Region Adjunct (CRA) and classification is handled using the WMQRAClassification section of the XML file. You can specify a selector, a destination, and a queue manager to control the selection of a transaction class.

SIB Request

The Service Integration Bus (SIB) handles messages in the Control Region Adjunct (CRA). There are two clauses in the XML file for these requests. Both clauses use the SibClassification tag. Within one clause you can specify a type of "jmsra". This applies to MDBs which are deployed against JCA resources for use with the default messaging provider. You can also specify a type of "destinationmediation". This applies to mediations assigned to a destination on the Service Integration Bus.

In both cases you can specify the bus name, a selector, and a destination to qualify which requests you want to match. For each clause you can specify a different transaction class to be used in classifying the requests with Workload Manager.

Internal Work

Sometimes the WebSphere code that runs in the controller needs to communicate with the WebSphere code running in the servants. This might be MBean processing or perhaps transaction commit or rollback processing, or something else entirely. For all these cases the controller drives special requests into the servant that run on threads in a different pool. You might want to classify these requests separately from your application request. These requests are usually important to server processing so do not put them into a service class that is very low in priority. Putting them into their own report class allows you to see how many requests make up the internal work overhead.

Optimized Local Adapter Inbound Requests

Classification for WOLA requests is handled in a very special way. How this works and the syntax for the XML file are described in a separate Information Center article found here:

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm.websphere.zseries.doc/info/zseries/ae/tdat_olawlm.html

To summarize what happens:

- A transaction running in CICS has an associated WLM Performance Block (like an enclave, but different).
- The transaction uses the WOLA API to drive a request into WebSphere. The JNDI name of the target EJB is provided in that request.
- The WOLA request flows into WebSphere. The transaction name from the WLM Performance Block is extracted and carried with the request.
- The WAS WLM Classification XML file is examined for an entry matching the JNDI name of the request.

- If the XML entry is found and indicates that the transaction name should be propagated, the extracted transaction name is used.
- If the XML entry is found and provides a transaction class name it will be used when classifying the request.
- Otherwise no transaction name or transaction class is provided when WebSphere calls WLM to classify the request.

These two options allow you to classify WOLA work in WebSphere with either a transaction class, like other WebSphere requests, or with the transaction name, like you are already doing in CICS. You have to decide which 'type' of classification you want to be consistent with. Remember that the CICS classification rules in WLM won't be used by WebSphere because the subsystem name in the rules is different.

An example

Explanation of the example

In this example we have a single WebSphere Application Server instance. The server handles HTTP requests for two different applications. We want the work for the applications to be managed to the same goal. However, we want RMF to report on the applications separately.

How it works

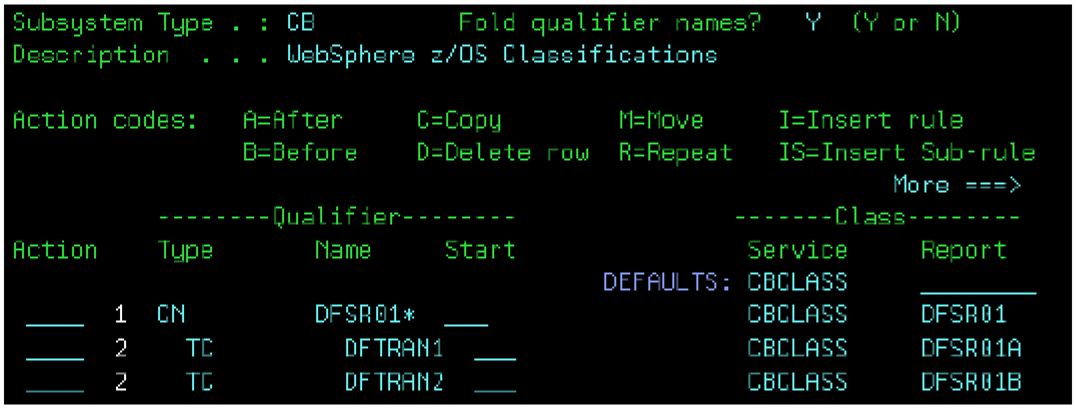
To configure this we just need to take two simple steps. The first step involves setting up a classification XML file that recognizes the two applications and assigns them to two different transaction class names.

The second step requires us to update our WLM policy to include classification rules that assign WebSphere work from this server into a defined service class and two different report classes.

The XML file looks like this:

```
<InboundClassification type="http" schema_version="1.0"
default_transaction_class="DFTRAN3">
   <http_classification_info uri="/gcs/applicationA/*"
transaction_class="DFTRAN1" />
   <http_classification_info uri="/gcs/otherApplication/*”
transaction_class="DFTRAN2”/>
</InboundClassification>
```

The WLM classification rules look like this:



```
Subsystem Type . : CB          Fold qualifier names?   Y  (Y or N)
Description  . . . WebSphere z/OS Classifications

Action codes:   A=After     C=Copy        M=Move     I=Insert rule
                B=Before    D=Delete row  R=Repeat   IS=Insert Sub-rule
                                                            More ===>
          --------Qualifier--------              -------Class--------
Action    Type       Name     Start                Service     Report
                                       DEFAULTS: CBCLASS     ________
 ____  1  CN         DFSR01*  ___                 CBCLASS     DFSR01
 ____  2    TC         DFTRAN1  ___               CBCLASS     DFSR01A
 ____  2    TC         DFTRAN2  ___               CBCLASS     DFSR01B
```

Next Steps

What is your system doing now?

Once you have your system all configured and work is being run, you might reasonably ask yourself, "What is happening?" Are your goals being met? How is work being classified? Is everything going the way you hoped? There are numerous tools you can use to help figure out what is going on. In this section we will look at just a few.

WLMQUE

The z/OS Workload Manager development team has provided a simple, TSO-based tool to help monitor the queue of work between the controller and servant (or any other user of WLM's application environment technology). This tool, called WLMQUE, can be downloaded from the following website:

https://www-03.ibm.com/servers/eserver/zseries/zos/wlm/tools/wlmque.html

This tool displays each application environment and information about the servant regions associated with it. You can see how many requests are in the queues, which service class the servant regions are bound to, and other useful information. You should see the documentation that comes with the tool (including the interactive help built into the tool) for more information.

RMF

The Resource Measurement Facility (RMF) is an IBM product which reads SMF records produced by z/OS and formats them into easy-to-view summaries. A lot of different information about your z/OS system can be monitored using RMF. The most relevant information to this paper is the Workload Activity Report which shows, on a service class or report class basis, how WLM is doing at meeting the defined goals.

RMF will provide a graph which shows the percentage of requests received during an interval which exceeded, met, or failed to meet the goals specified in the service class. Looking at this report you can tell how much work is ending up in the different classifications you provided and how well WLM is doing at meeting your goals.

There is a lot of documentation, presentations, and education available about using RMF.

SMF 120-9

WebSphere Application Server Version 7 writes an SMF type 120, subtype 9 record for each request dispatched in a servant region. These SMF records contain the information used to classify requests (e.g. the URI for an HTTP request). The transaction class name which was given to WLM is provided. Therefore these records could be used to validate that your classification XML file is working as you expect.

You can also extract other useful information, such as elapsed time to dispatch a request, the amount of time each request spent on the WLM queue, etc. These values can be helpful in determining how you are doing in meeting your goals. You can also find out how much CPU time was used by each request.

For more information about the SMF 120-9 record, please see the WebSphere Information Center or white paper WP101342.

Building your own XML file

If you have decided that you need to create your own WLM classification file you might wonder where you should begin. If you have HTTP requests to classify you need to know what URIs are arriving at the server that need to be classified. By knowing the root specified for the application and the various pages available within it, you can determine the URIs you need to specify in the XML file. However, in some cases it might be simpler to just run for a while and see what requests come in.

The SMF 120-9 records will tell you the URIs of the HTTP requests processed by the server. The record can be turned on and off dynamically via MVS console Modify commands. This allows you to just turn the record on for a little while to get a sample of the requests being processed. This should give you a start.

SMF Browser Plug-in

The Java browser provided by the WebSphere Application Server on z/OS development team includes a plug-in that will examine the classification information provided in the SMF 120-9 records and produce a template XML file as a starting point.

Getting the Browser

The browser can be downloaded from the website below. Please remember that this browser is not officially part of the WebSphere product and is, therefore, not officially supported.

https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=zosos390

Running the Browser

To generate the XML template you need to run the browser specifying the template generation plugin. The syntax of the command is this:

```
java com.ibm.ws390.sm.smfview.SMF "INFILE(smf.dataset.name)"
"PLUGIN(com.ibm.ws390.sm.smfview.ClassificationXMLFilter,output-
destination)"
```

For example:

```
java com.ibm.ws390.sm.smfview.SMF "INFILE(HUTCH.SMFDATA.SYSB.D080822)"
"PLUGIN(com.ibm.ws390.sm.smfview.ClassificationXMLFilter,/u/hutch/smf.txt
)"
```

Thinning out the output

The plug in will generate a node in the XML tree for every different request that it sees. This is probably far more granularity than you want. Most of the different requests within a single application can probably be grouped together into very few transaction class names (perhaps just one). To do this you will need to 'thin' the output by removing overly-specific nodes and perhaps making use of some wild-cards to group similar things together.

Remember that each request received by the server traverses the tree you provide in this XML file. Do not make it any more complicated than necessary.

Is it working?

How do you know if the XML file is doing what you wanted?

Display,work,clinfo

WebSphere Application Server on z/OS provides a display command to show how the XML file is being used. By directing an MVS console Modify command to the controller region you can get a breakdown about the IIOP and HTTP sections of the file. The command is:

MODIFY server,DISPLAY,WORK,CLINFO

This command will issue a WTO for each node in the HTTP and IIOP sections of the tree. The WTO will show you how many times that node has been used. It will also show how many times the node has been examined to determine if it is a match for the request being processed. This information may be useful in 'turning' the XML file so that the most frequently used nodes are earlier in the tree and thus match more quickly.

SMF 120-9

As stated earlier, you can examine the SMF 120-9 records to view the classification information and the resulting transaction class provided to WLM. These records can be used to verify your XML file is behaving the way you expected.

WebSphere Liberty Profile and WLM

In WebSphere Version 8.5 an entirely new server runtime was introduced. Liberty is a light-weight, composable server. You can read much more about it at http://wasdev.net as well as in the WAS V8.5 Information Center (of course!). The purpose of this section is just to talk about the interactions between Liberty and z/OS WLM.

The most important thing to understand about this interaction is that it is optional and, by default, turned off. If you do nothing in the server.xml to ask for interactions with WLM, then there will be none. This is deliberate in order to make the Liberty server work exactly the same on z/OS as it works on other platforms unless configured specifically to work differently.

If you don't enable the WLM feature then your Liberty server is just an address space consuming resources to WLM. It has no visibility to the work running inside the server. The address space will be managed as a USS process or as a started task, depending on how you start the server. RMF will not be able to report on transactions processed by the server because WLM doesn't see the beginning and end of any transactions. This is all probably just fine for development and limited test. In fact, it makes development and test easier because those groups don't need to get involved in how their servers are interacting with WLM to just run a few things.

However, for larger scale testing or production you will probably want to let WLM know what is going on inside your server.

To get any interactions with WLM from Liberty you need to configure the server to enable the WLM feature. Do that by adding the following line to the featureManager section of the configuration:

```
<feature>zoswlm-1.0</feature>
```

This alone won't cause the server to connect to WLM or classify work. You need to tell the server how you want the work classified. This is a big difference between WLM classification with Liberty compared to traditional WAS. In traditional WAS every piece of work that runs in the servants gets classified with WLM. In Liberty only those work requests that match a classification rule in the server configuration will have an enclave created. That means it is possible to have some work in the server running with enclaves and some without. How do you tell Liberty how to classify work? With the wlmClassification XML tag:

```
<wlmClassification>
   <httpClassification transactionClass=”CLASS001” resource=”/test/*” />
</wlmClassification>
```

This will cause any requests for URI's under the 'test' context root to be classified with WLM using the CLASS001 transaction class. Any requests for other URIs will not be classified with WLM. If you don't want a mix of some work with and some work without an enclave, its a good idea to include an extra 'default' rule to catch any requests that slip through, like this:

```
<wlmClassification>
   <httpClassification transactionClass=”CLASS001” resource=”/test/*” />
   <httpClassification transactionClass=”WLPDFLT” />
</wlmClassification>
```

Any requests that don't match the previous rule(s) will match that final one. Note that you can also classify using the target host/port as well as the method name (e.g. GET, POST).

But what about the collection name? In Traditional WAS we always classified with the collection name (really the cluster short name) and the transaction class, if we had one. In Liberty we don't even do classification with WLM if we don't have a transaction class. But what about the collection name?

We classify using the collection name too. We get it from the directory name where the server.xml file lives. Its probably in something like $WLP_USER_DIR/servers/myServer. In that case Liberty will use the string 'myServer' as the collection name (note that WLM ignores the case differences).

Well that's all fine, but what if the name is longer? The default server name is defaultServer. That is much longer than 8 characters. Can WLM handle that? Actually it can, but the WLM panels make it a little awkward to define the rules for classification with longer strings. You have to set up nested rules to match each 8 character section of the string. If your server directory name is longer than 8 characters we recommend overriding the collection name in the server configuration like this:

```
<zosWorkloadManager collectionName=”MYSERVER” />
```

This will cause the server to use a collection name of MYSERVER instead of defaultServer or whatever your server directory name is.

The last thing to talk about is authorization. Anybody with access to the binaries can configure and start a server. You probably don't want them just configuring anything they want for WLM classification and work running under whatever service class it matches. You only want servers that are authorized to classify work with WLM to be allowed to do that.

There are two different ways to do this. If you have an Angel started then authorize the server to use the Angel and then authorize it to access the WLM functions via the Angel. Please see the documentation in the information Center about what the Angel is and how it is used. The entity to allow the server userid access to for WLM functions is:

bbg.authmod.bbgzsafm.zoswlm

However, if you haven't started an Angel you can still classify work with WLM. The Liberty server will try to use the native LE interfaces which are provided by WLM. These interfaces don't require code to run in an authorized key, but instead just need access to the Facility class entity BPX.WLMSERVER.

Either approach will work. Chose the one which fits best with how you want to manage access to WLM services by Liberty servers. The most important thing is to be consistent.

That's all there is. Remember that the Liberty profile is a completely separate runtime from the Traditional WAS server. The things discussed in this section only apply to Liberty.

Appendix: Configuration Variables Referenced

wlm_classification_file

This variable specifies the name of the XML file containing information used by WebSphere to help WLM classify the request. Attributes of each request will be matched against the contents of this file to determine a transaction class name. This name is used by WLM, along with other attributes, to classify the request and assign a service class and report class. The XML file has to be in ASCII and is located in the Unix System Services file system.

protocol_iiop_local_propagate_wlm_enclave

When an IIOP request is used to flow work from one WebSphere server to another server on the same z/OS image the WLM enclave used by the originating server is carried along with the request. This can sometimes cause an work in an unexpected service class to show up in the second server. To stop this behavior and have the target server create its own enclave, set this variable to zero.

wlm_stateful_session_placement_on

By default WLM tries to keep all requests assigned to the same service class in the same servant region. Setting this variable to one will cause WLM to instead spread the work around. This works best when each request without an existing affinity to a particular servant region creates such an affinity to be used by subsequent requests from the same client. WLM will balance the number of affinities to each servant region which will keep the work evenly spread across the set of servants.

server_use_wlm_to_queue_work

This variable controls whether WLM or internal WebSphere code should be used to manage how work is delivered to the servant regions. The default value of one for this variable tells WebSphere to use WLM to deliver work to the servants.

server_work_distribution_algorithm

When WebSphere code is used to deliver work to the servant regions there are several algorithms that can be used. They algorithms are called: Hot Thread, Round Robin, and Hot Robin. Please see the descriptions of these algorithms earlier in the paper for details.

wlm_enclavecreate_exstartdefer

Determines if the RMF measured queue time will end when work is picked up in the servant region (YES) or when the enclave is created in the controller (NO). Probably best to leave this one as the default of 'YES'.

wlm_ae_spreadmin

Controls whether WLM evenly divides the servant regions among the service classes for a given server or if WLM is allowed to divide them up as it wants. The default and value used prior to the introduction of this variable is '1'. Set to '0' to let WLM decide how to divide things up.

Appendix: Common Problems

Work Not Classified As Expected

Things not being classified the way you expect can lead to all sorts of interesting problems. Some of these are described in the following sections. But generally it pays to be sure things are ending up managed by WLM the way you wanted. You can use the WLMQUE tool to monitor how the servant regions are being bound to service classes. Reports from RMF will also show you how much work is running in each report and service class. If you do not have problems and these tools show what you expect, then you are probably fine. If things look odd, then the SMF 120-9 record can help you determine what transaction class was used by WebSphere to help WLM classify the request. The most common cause of unexpected classification is work turning up that does not match expected patterns. Either an unexpected default is taken in the WebSphere classification XML file or else an unexpected default is taken in the WLM classification rules. The trick is to realizing this is happening and then determine what work request is causing it.

Enclave Propagation Causes an Unexpected Service Class

One way things can take an unexpected turn is an enclave propagating from another local server. If a request is dispatched in Server A and makes an RMI-IIOP request to an EJB located in Server B in the same z/OS image WebSphere will flow the enclave with the request. This means that Server B will have to run work under the service class of the enclave created by Server A. It is possible that no work that comes directly to Server B will be classified to this service class. This leads to an ‘extra’ service class that WLM has to accommodate as it assigns servant regions to handle different service classes.

In this case the SMF 120-9 record can help. There is a bit in the record (SM1209DU) that is set when the server created the enclave for the request described by the record. If the bit is not set, then the enclave was propagated to this server from somewhere else.

That, by itself, is not necessarily a problem. It is only an issue if the propagated enclaves are associated with a different service class than other work handled by this server. Even then it might not be bad, unless you were not expecting it.

If you do not want enclaves to be propagated, but would prefer to let each server classify it’s own work and create it’s own enclaves just tell WebSphere not to do it. Setting the variable protocol_iiop_local_propagate_wlm_enclave to 0 tells WebSphere not to propagate enclaves. Remember to set this in the originating server (Server A in our example).

WLM Round Robin Behaves Oddly

First, remember that WLM Round Robin is not really a round robin. WLM is trying to balance the affinities owned by each servant region bound to a particular service class. If the server is managing multiple service classes, then the servants are divided between the service classes and WLM will try to keep affinities balanced within each set. A common reason for unexpected behavior with the Round Robin function is that the server is handling more service classes than you think it is. The WLMQUE tool is the best way to find out what service classes the servant regions are handling. If you have two servant regions and WLM Round Robin is only using one of them, chances are the other servant is bound to some other service class. You need to find out which one (WLMQUE) and then determine what work ran that required WLM to assign a servant to that service class.

Only One Servant Region with Multiple Service Classes

A common problem with workload classification is having more service classes in use than you think you do. This is especially a problem if you only have WebSphere configured to use a single servant region by specifying a min=max=1.

WLM will always try to keep requests with different service classes in different servant regions. If you only allow WLM to create one servant region then all requests must end up classified to the same service class, even requests generated internally by the server itself (such as Mbeans).

The one servant region will be bound to the service class of the first request that is dispatched. Any requests without affinity that are associated with another service class will just be stuck in the WLM queue waiting for a servant region that the configuration will not allow it to start. WebSphere will eventually time out the request and clean it up.

There are two possible solutions to this problem. The first is to change the configuration to specify single-servant instead of setting min=max=1. This configuration change tells WLM that it is ok to mix multiple service classes in the single servant region. By limiting WLM to one servant region using min=max you are telling WLM you are sure that you only need one service class and anything else is in error and should not be allowed to run. If you are ok with mixing service classes in a single servant region and are you sure you only want one servant, then configure the server as single-servant.

The other possible solution to this problem is to cause all work to be classified into the same service class. Take this approach if you thought you only had one service class and thus configuring min=max=1 servant was correct. To resolve this problem you need to determine what requests are ending up in a different service class. First you should use the WLMQUE tool described earlier to determine what service classes are being used. With that knowledge you can examine your WLM classification rules to conclude how work is end up in in this service class. Often this is caused by a 'default' rule in your classification that is being used unexpectedly. The SMF 120-9 records can also be helpful in determining which work requests are using a classification rule you didn't expect would be used.

Defaulting to Discretionary

Another common problem caused by defaults and unexpected requests is very very slow responses. This happens when a default classification rule puts work requests into the discretionary service class. Discretionary means WLM should run this work when it gets a chance, but it is not very important. It makes sense that requests classified as discretionary would run very slowly.

Carefully examine the WLM classification rules for a path that could lead to work being classified as discretionary. If you have a wlm classification XML file you should also examine it for paths or defaults that might lead to the WLM classification yielding a discretionary goal. Finally, examine the requests being executed by the server by looking at the SMF 120-9 records. Hopefully you can find the requests that led to a classification of some work running as discretionary and either eliminate it or classify it properly.

Document change history

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

08/18/10	Initial Version
08/23/10	Assigned document number
09/01/10	Corrected link to Robert Vaupel's WLM document
5/26/11	Added information about IEAOPTxx ManageNonEnclaveWork parameter
6/10/11	Restored the picture on page 5 which mysteriously vanished and tried yet again to have a link to Robert Vaupel's WLM document that works
7/20/211	Corrected the hyper link to Robert's document (text was right, link behind it was wrong..)
7/25/11	Corrected picture of WLM classification rules in the Example and added some text discussing the affect of the AE_SPREADMIN WLM parameter
11/23/11	Rewrote the section on asynchronous work to consider whether the WLM Service is enabled on the work manager(s) or not.
10/19/12	Various minor edits, talk about dynamic min/max Srs, WLM round-robin low-utilization effects, and ExStartDefer. Also added the section on work classification with Liberty.

End of WP101740