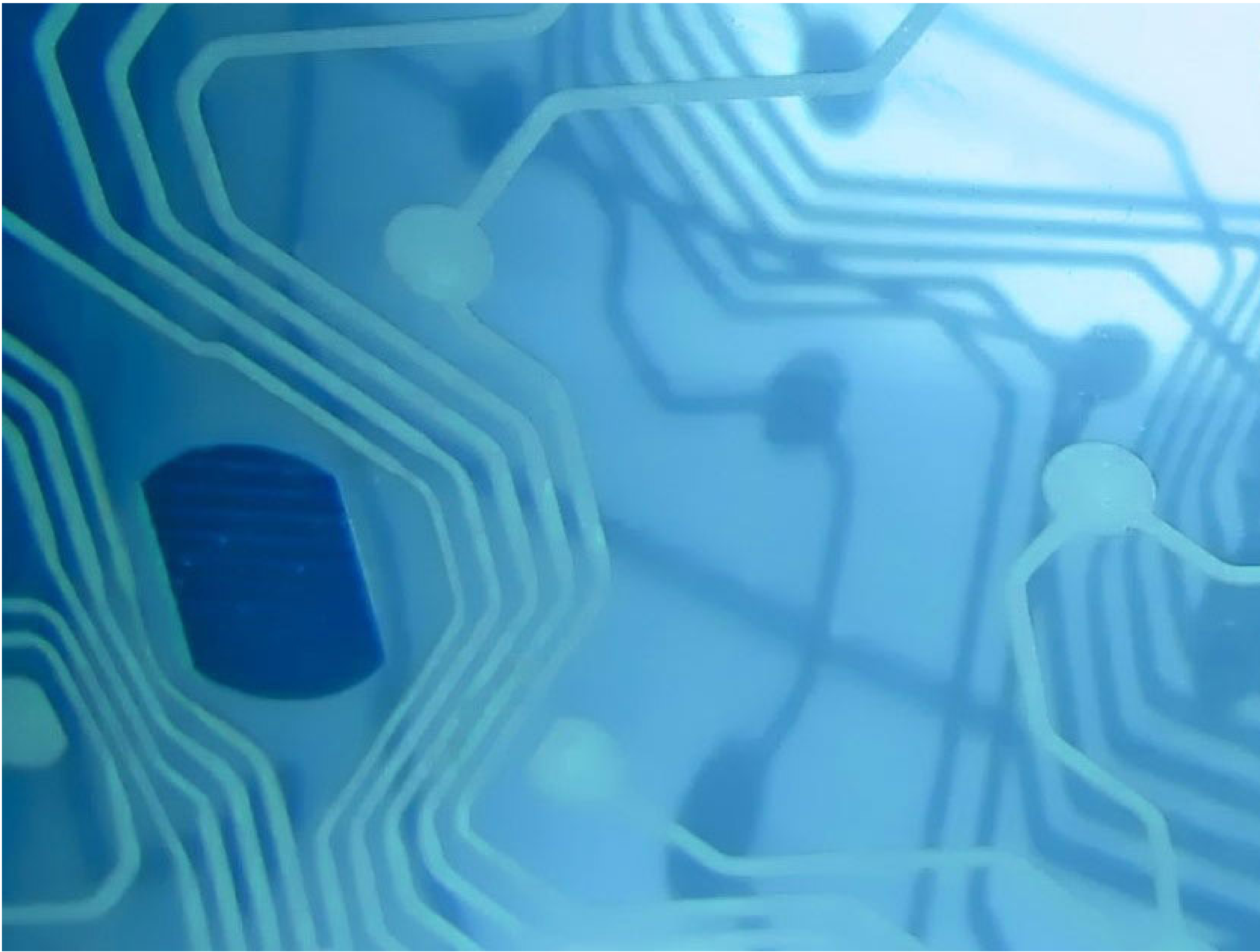
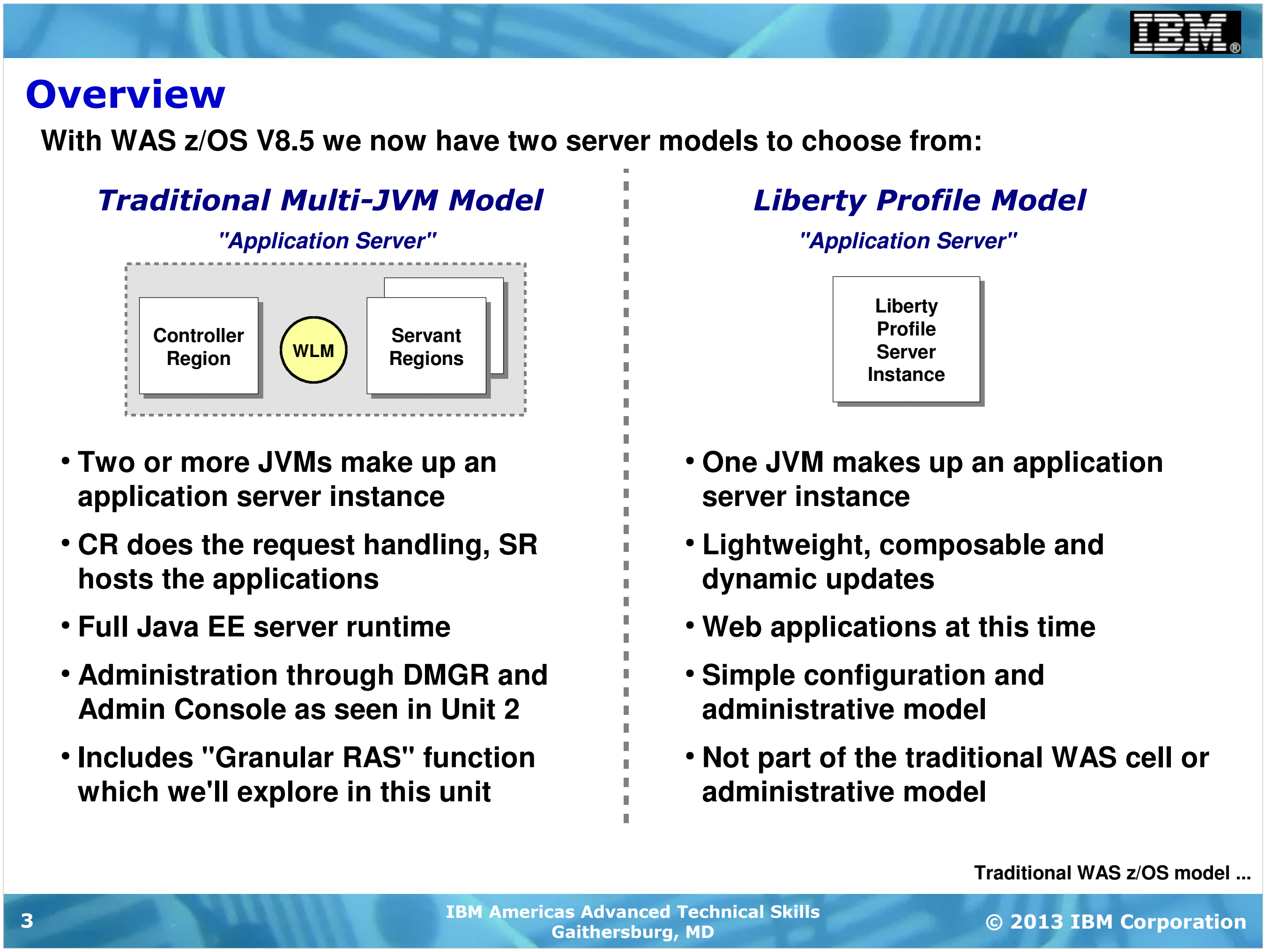
WebSphere Application Server V8.5 for z/OS

WBSR85

Unit 3 - Server Models



This page intentionally left blank



Overview

With WAS z/OS V8.5 we now have two server models to choose from:

Traditional Multi-JVM Model

"Application Server"

Controller Region
WLM
Servant Regions

- **Two or more JVMs make up an application server instance**
- **CR does the request handling, SR hosts the applications**
- **Full Java EE server runtime**
- **Administration through DMGR and Admin Console as seen in Unit 2**
- **Includes "Granular RAS" function which we'll explore in this unit**

Liberty Profile Model

"Application Server"

- **One JVM makes up an application server instance**
- **Lightweight, composable and dynamic updates**
- **Web applications at this time**
- **Simple configuration and administrative model**
- **Not part of the traditional WAS cell or administrative model**

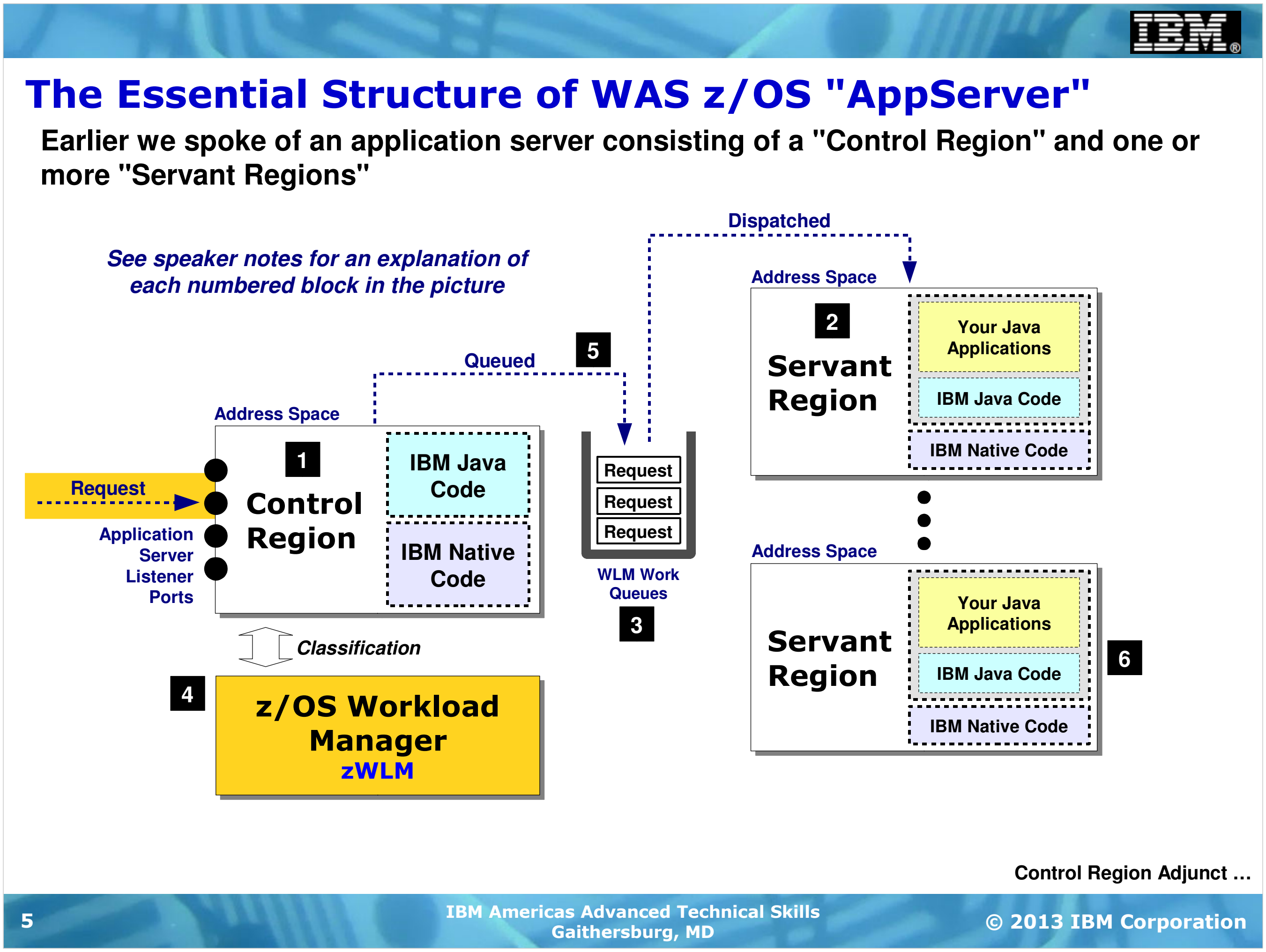
Traditional WAS z/OS model ...

With WAS z/OS V8.5 we now have two "server models" to consider -- the traditional model with the multiple JVMs and the full set of Java EE functions, and the new "Liberty Profile" model which is intended to offer a lighter, more dynamic alternative when the traditional model is deemed more than adequate.

In this unit we'll cover both, starting with the traditional model, which will include a discussion of work distribution to multiple servant regions, WLM service and reporting classes, the XML classification file and the V8 function called "Granular RAS," which allows functional behavior to be driven down to the request level. Then we'll transition to the Liberty Profile model.

Traditional WAS z/OS

The full Java EE, multiple-JVM model



The Essential Structure of WAS z/OS "AppServer"

Earlier we spoke of an application server consisting of a "Control Region" and one or more "Servant Regions"

See speaker notes for an explanation of each numbered block in the picture

Control Region Adjunct ...

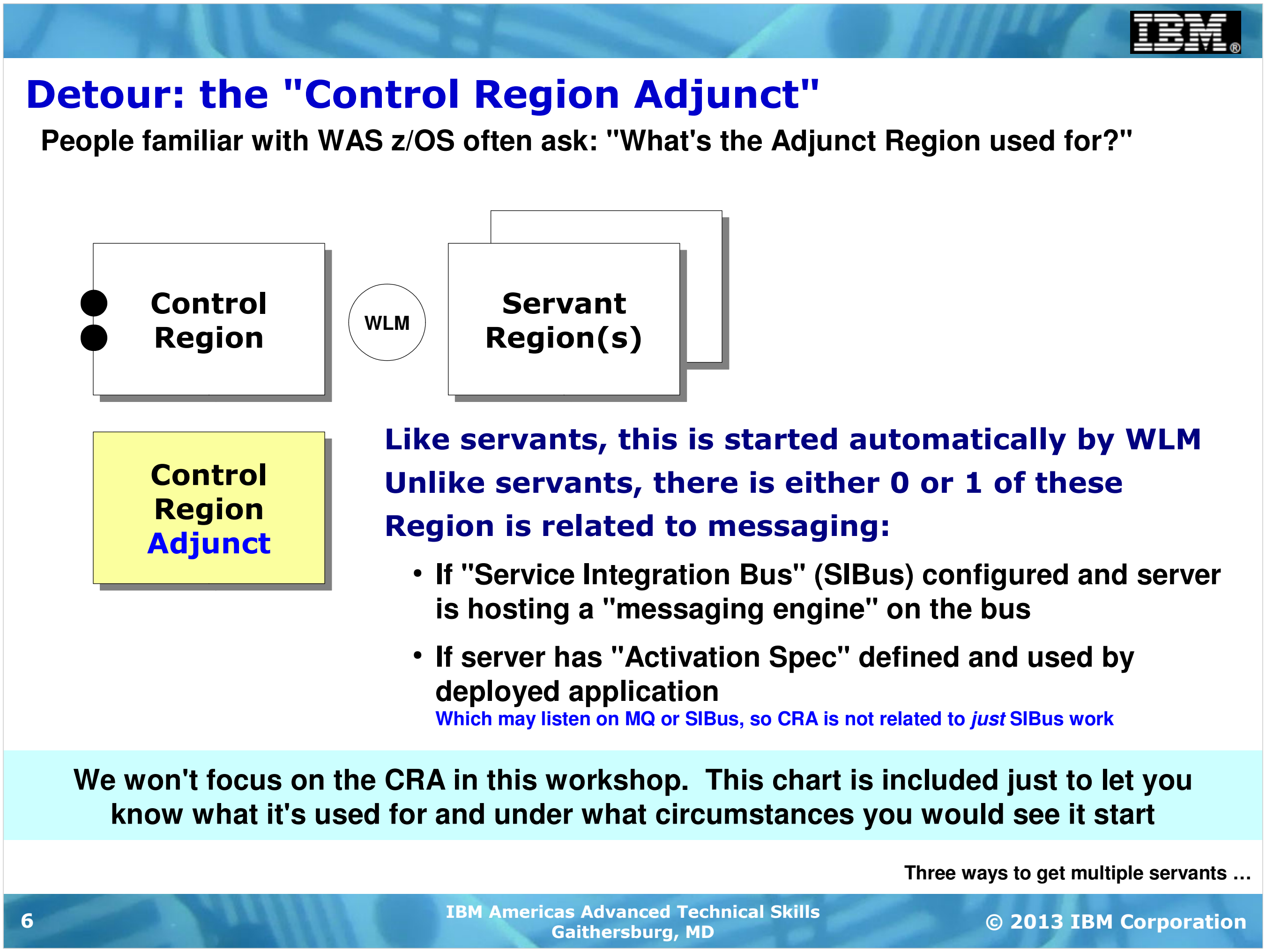
5

IBM Americas Advanced Technical Skills
Gaithersburg, MD

© 2013 IBM Corporation

This picture represents the essential architecture of a WAS z/OS "application server," which unlike distributed WAS is not one JVM but multiple. The following list corresponds with the numbered blocks on the chart:

1. The "Control Region" (or "CR" for short) is a z/OS address space that contains about half native code and half Java code. It serves as the front-end handler of inbound work. Your applications do not run here. Work is received here, processed, then passed back to the z/OS address space where your applications do run, which is the servant region.
2. The "Servant Region" (or "SR" for short) is mostly IBM Java code with a small slice of native code. This is where your applications will run. When you deploy an application to the "application server," WAS z/OS makes sure the application binaries end up in this JVM.
3. Between the CR and the SR are a set of WLM work queues. When work comes into the CR and gets processed, the CR places the work on a work queue and from there it gets dispatched to a servant region. Work is pulled from the queue by the servant region; it is never pushed into a servant. That means the servant will consume only as much work as it can process. Temporary excess work queues up and then gets worked down.
4. Under the CR sits z/OS Workload Manager, or "zWLM" for short. zWLM is a key component of the z/OS system. It manages work and work priorities across all the jobs and tasks running on the z/OS system. One of the roles zWLM plays is to classify work so zWLM can understand the relative priority of that work to other work it sees on the system. The WAS z/OS CR makes use of the zWLM APIs to classify the work and place it on a work queue. We have much more coming up on classification.
5. As noted earlier, work is queued to a work queue, then dispatched to a servant where your application runs that work.
6. WAS z/OS has the ability to host multiple servant regions behind the CR to act as a kind of "vertical cluster" to the CR. Additional servants may be configured ahead of time or they may be dynamically started by zWLM based on workload seen by zWLM. More on this coming up as well.



Detour: the "Control Region Adjunct"

People familiar with WAS z/OS often ask: "What's the Adjunct Region used for?"

Control Region

WLM

Servant Region(s)

Control Region Adjunct

Like servants, this is started automatically by WLM

Unlike servants, there is either 0 or 1 of these

Region is related to messaging:

- **If "Service Integration Bus" (SIBus) configured and server is hosting a "messaging engine" on the bus**
- **If server has "Activation Spec" defined and used by deployed application**
  Which may listen on MQ or SIBus, so CRA is not related to *just* SIBus work

We won't focus on the CRA in this workshop. This chart is included just to let you know what it's used for and under what circumstances you would see it start

Three ways to get multiple servants …

6

IBM Americas Advanced Technical Skills
Gaithersburg, MD

Another address space you may see is something called the "Control Region Adjunct," or "CRA" for short.

This region is used for messaging. It's in one way like servant regions in that it's started automatically by WLM when the control region starts, but unlike servant regions there is either zero or one of these. There can't be more than one for any given application server.

The CRA is started by WLM when one of the following conditions is met:

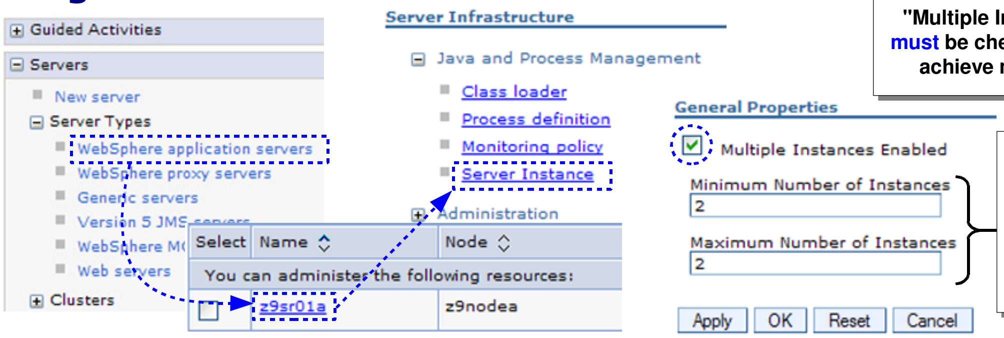
1. You have configured a "Service Integration Bus" (otherwise known as the SIBus) and an application server is connected to the bus. By doing that you tell WAS z/OS that the server will host a "messaging engine." The CRA is used as the runtime region in which the messaging engine function resides. So a server added to an SIBus signals to WAS z/OS to fire up a CRA so the messaging engine has a place to run.
2. You have configured an "Activation Spec" (as opposed to the older-style "Listener Port") for Message Driven Bean (MDB) applications, and there's a MDB application deployed that references the defined Activation Spec. That tells WAS z/OS that it needs to start the CRA so the Activation Specification function has a place to operate.

Absent those two conditions, the CRA will not be present. With one (or both) of those conditions, the CRA will appear when the server is started.

Three Ways to Achieve Multiple Servant Regions

There are three ways to achieve more than one servant region for a given appserver:

Configure Minimum > 1



☑ Multiple Instances Enabled

Minimum Number of Instances: 2

Maximum Number of Instances: 2

"Multiple Instances Enabled" must be checked in all cases to achieve multiple servants

Configure "Minimum" at some number larger than 1. When the server starts you'll achieve that number of servants

Use MODIFY to dynamically change minimum

```
F Z9SR01A,WLM_MIN_MAX=3,3
```

Controller JOBNAME

MIN,MAX

- "Multiple Instances Enabled" must already be checked
- If MODIFY minimum > configured minimum then servant is started
- If MODIFY minimum < configured minimum then excess servants will be stopped when all work flushed

Let WLM dynamically start more servants

Based on volume of work

Based on WLM Service Class assignments

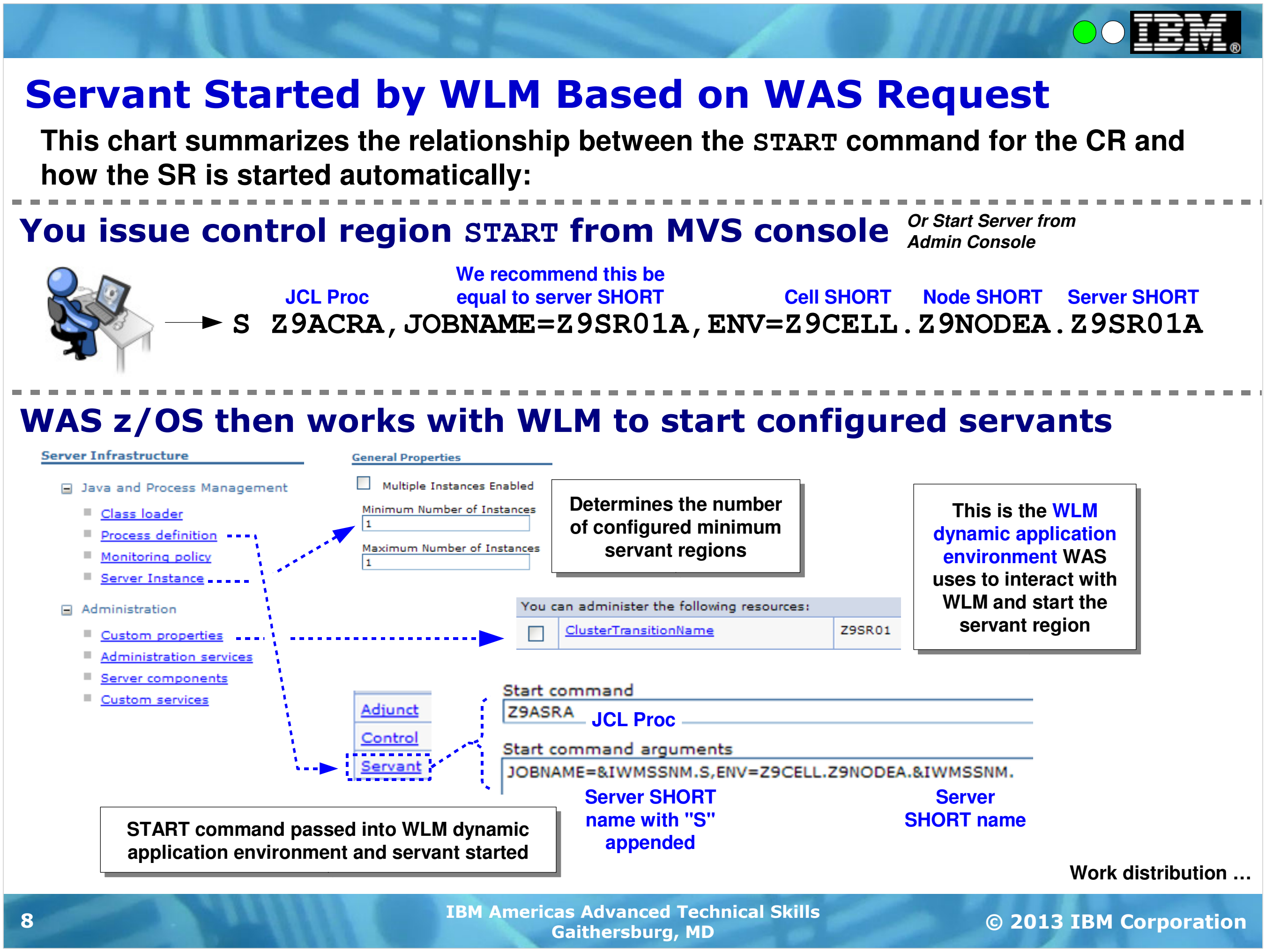
z/OS and starting servant ...

The default number of servant regions within any given application server is one. But there are three ways you may achieve multiple servant regions.

The first is to check the "Multiple Instances Enabled" checkbox and configure a minimum number greater than one. When the application server is started the next time your configured minimum servant regions will be started.

The second way is to use the z/OS MODIFY command to dynamically increase the minimum and maximum numbers. For this to work the "Multiple Instances Enabled" checkbox must have been set. Suppose your server was running with MIN=1 and MAX=3 and at the present time you had only one servant but wished for all three to be running. Issuing the MODIFY command to change the minimum tells zWLM to start additional servants. The reverse is true as well -- MODIFY to a minimum number small than presently running and zWLM will stop servants when all the work in those servants has been flushed.

The third way is to allow WLM to dynamically start more servants. Again, "Multiple Instances Enabled" must be checked and the configured maximum servants must be more than what's presently running. Then zWLM may start additional servants based on either the volume of the work or the arrival of a Service Class otherwise not placed in a servant. The role of zWLM Service Classes in all this will be explained in more detail in a bit.



Servant Started by WLM Based on WAS Request

This chart summarizes the relationship between the `START` command for the CR and how the SR is started automatically:

You issue control region `START` from MVS console

Or Start Server from Admin Console

JCL Proc | We recommend this be equal to server SHORT | Cell SHORT | Node SHORT | Server SHORT

```
S Z9ACRA,JOBNAME=Z9SR01A,ENV=Z9CELL.Z9NODEA.Z9SR01A
```

WAS z/OS then works with WLM to start configured servants

Determines the number of configured minimum servant regions

This is the WLM dynamic application environment WAS uses to interact with WLM and start the servant region

JCL Proc

Server SHORT name with "S" appended

Server SHORT name

START command passed into WLM dynamic application environment and servant started

Work distribution …

The Control Region is something you start with a z/OS command; the Servant Region is something zWLM starts automatically. (In addition to issuing a z/OS command, it's possible to start WAS z/OS control regions through the Admin Console "Start Server" button, or with the startServer.sh shell script. In both cases what ends up happening is WAS z/OS issues the z/OS START command like what's shown above.

The CR START command has the format as show -- START <proc>, then a JOBNAME value, then an ENV= string that helps WAS z/OS know which of potentially many different servers you want started. The ENV= string consists of the cell short name, the node short name and the server short name. That's a unique identifier for WAS z/OS and using that it can start the specific server you request. The JOBNAME value can be anything you wish, but we recommend it be equal to the server short name.

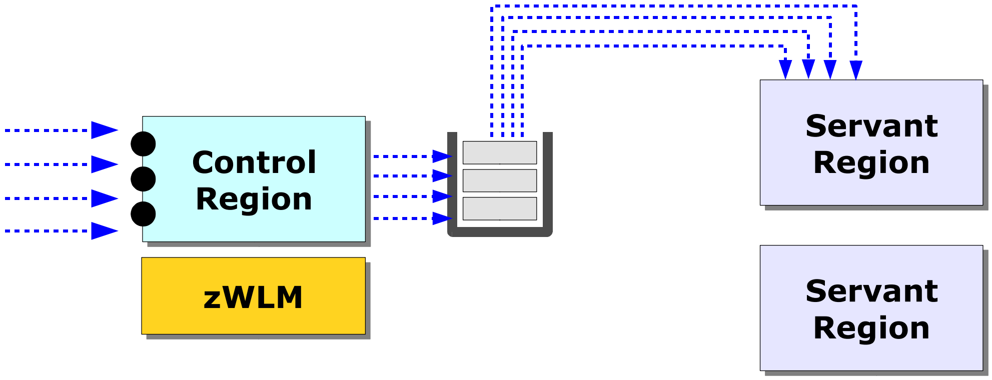
The servant region is started automatically using a combination of a "WLM dynamic application environment" and a configured START command for the servant. The dynamic application environment is an API to zWLM that allows for zWLM to start things on behalf of applications. In this case the CR is what's requesting the start of the SR through the zWLM dynamic application environment API. The WLM dynamic application environment name is configured in WAS z/OS as the "Cluster Transition Name." There's a reason for that ... but it's somewhat obscure and it's best just to leave it at that -- Cluster Transition Name = WLM dynamic application environment name.

The next thing it does is determine how many servant regions should be started initially. This is the "Minimum Number of Instances" number. The next thing it does is to locate the START command for the servant from the configuration files. The START command is similar to the CR except it uses a different JCL start procedure, and the JOBNAME and server short names are passed in as variables. Further, the JOBNAME value is appended with an "S" to indicate a "servant."

With the servant START command issued by zWLM the servant region(s) come up and when all the dust settles and the CR and its SRs are started then WAS z/OS knows the overall application server itself is started as well.

In General, WLM will Favor One Servant

This is behavior many *don't* expect ... when multiple servants are present the work tends to end up in one but not the other:



Most or all work will show up in one servant region

You *may* see some "spill over" into the other if work spike exhausts threads in first servant and WLM decides it's better to dispatch to second than hold in queue

WLM's default is to take "first available" and once chosen, stick with it

This behavior makes some sense. If all work can be handled by one servant, why not keep two swapped in and active when one will do?

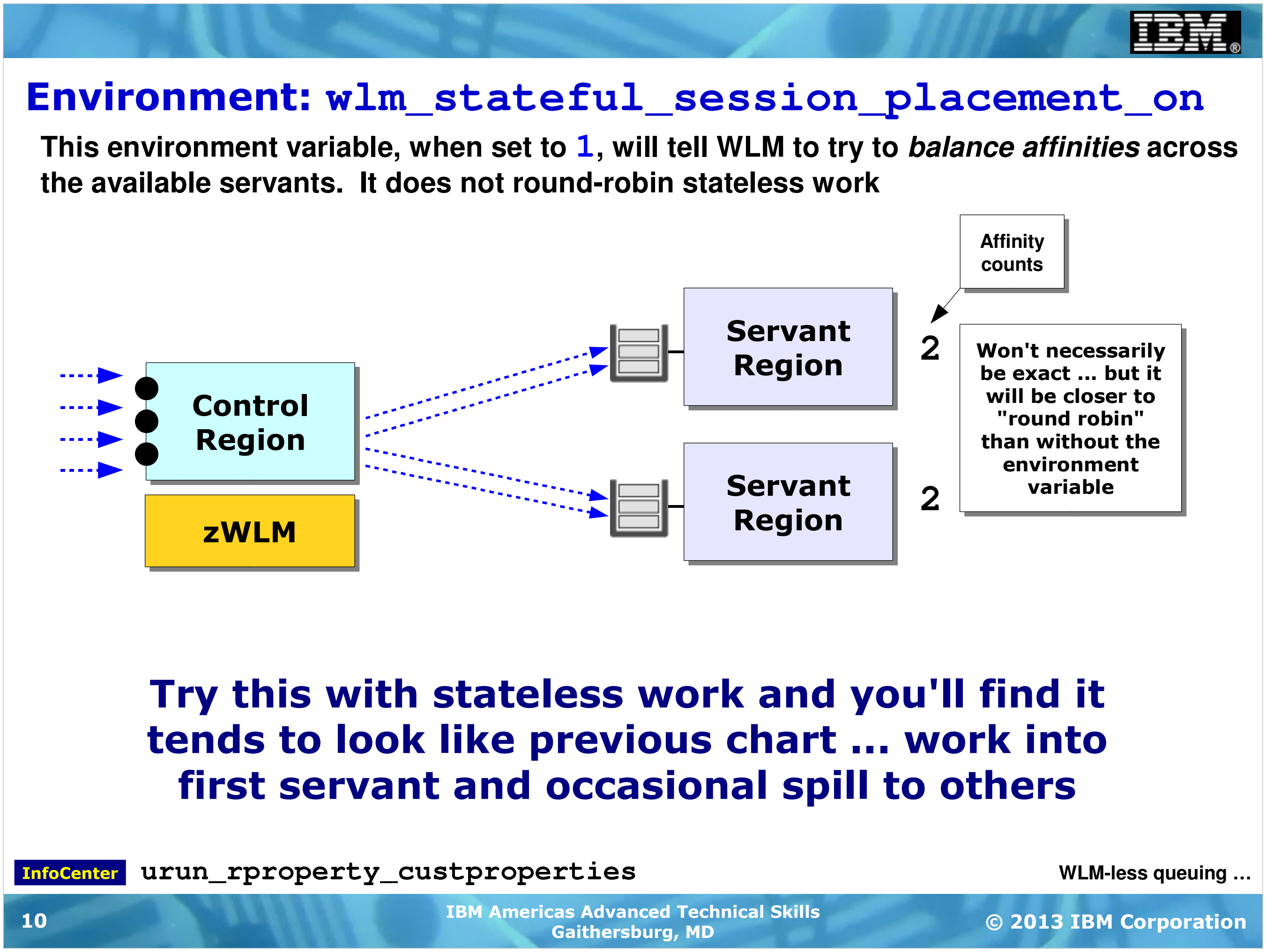
There's an environment variable to *approximate* "round-robin" ...

Stateful "round-robin" ...

Imagine a relatively simple two servant configuratino as shown. You might expect that the work would be evenly distributed between the two servers, but that's not the case. What you'll find is that WLM will favor one servant over the other. You may see all the work ending up in one servant and none in the other if WLM determines one server is sufficient. Or you may see a little "spill over" into the second servant. But in general, WLM will favor one over the other.

WLM's default behavior is to seek the first available servant and stick with that until it has reason to believe it must bring another one into play. This makes some sense if you think about it -- if you have two swappable address spaces and the work flowing in can be handled entirely by one address space, why keep both active and "hot" when one will do?

But as we said, most expect to see a balanced distribution between the two servant regions. This can be accomplished (to a degree) with an environment variable that will seek to balance the work. However, the environment variable is or stateful work. Let's see what that variable is ...



Environment: `wlm_stateful_session_placement_on`

This environment variable, when set to `1`, will tell WLM to try to *balance affinities* across the available servants. It does not round-robin stateless work

Won't necessarily be exact ... but it will be closer to "round robin" than without the environment variable

Try this with stateless work and you'll find it tends to look like previous chart ... work into first servant and occasional spill to others

InfoCenter **`urun_rproperty_custproperties`**

WLM-less queuing ...

10 IBM Americas Advanced Technical Skills Gaithersburg, MD © 2013 IBM Corporation

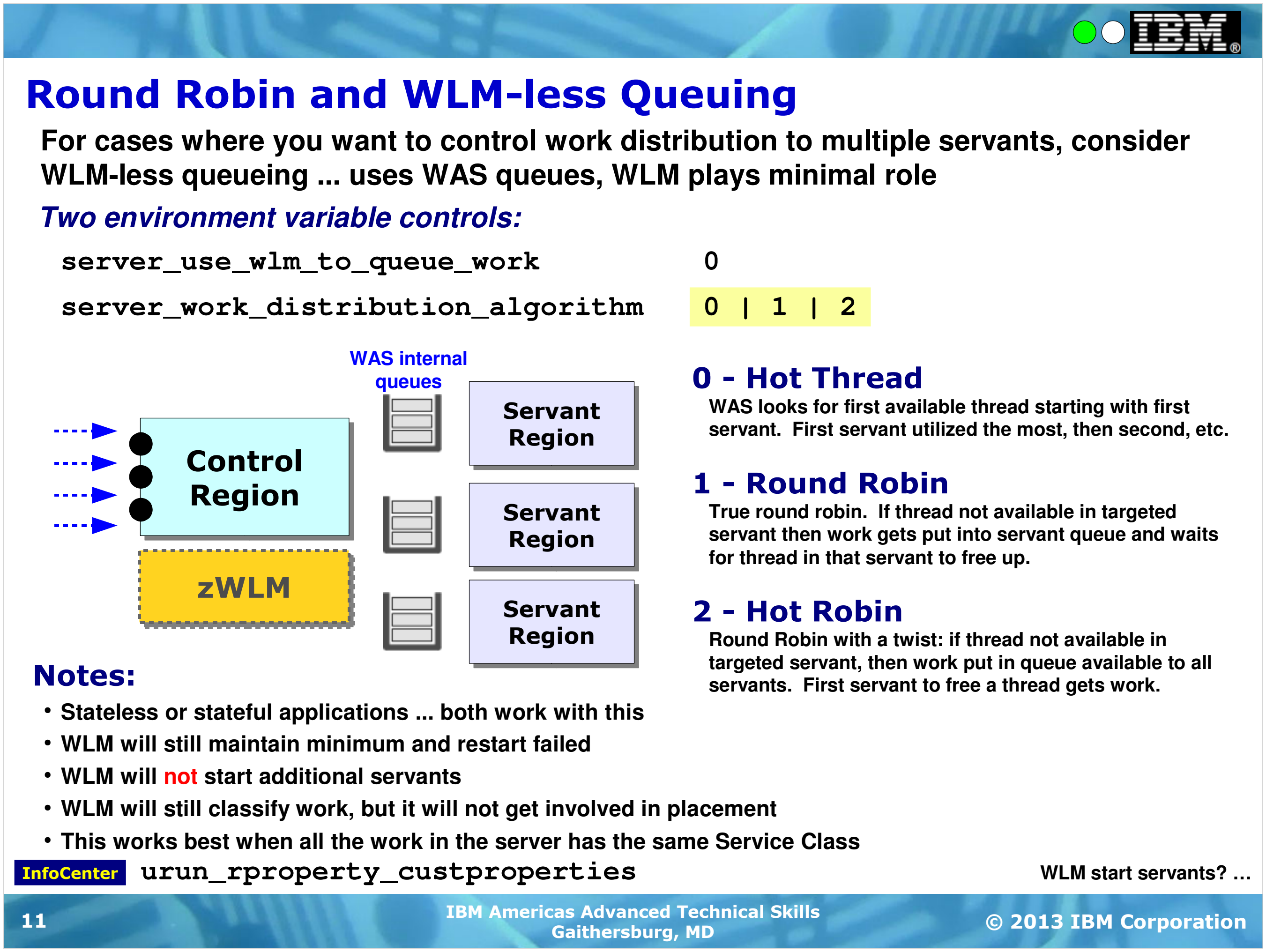
The environment variable `wlm_stateful_session_placement_on` tells WLM to try to balance *affinities* across available servant regions. An "affinity" to a servant region is created when the application creates some bit of data that is specific to the requesting client. The most common example of this is the HTTP Session Object, which many applications use as a means of maintaining some awareness of the client's "state" between requests that flow up from that client. The most common example cited is the "shopping cart" ... when a person is using some online shopping application an HTTP Session Object is often created and maintained to keep the user's shopping cart contents in an in-memory object.

Note: WAS z/OS and zWLM maintain the "affinity routing" between the CR and the SR. You do not need to worry about this. It's taken care of automatically.

This environment variable tells WAS z/OS to try to balance these affinities between the available servant regions. So if you have a stateful application and you turn this environment variable on, you'll see a rough balancing between the servant regions.

But, if you were to use this environment variable with a stateless application, you'd find that WAS z/OS and z/WLM would go back to favoring one servant over others. That's because there's no affinities to balance.

If you have a stateless application and you wish to distribute the work across available servants then the way to go is with WAS z/OS internal queuing. This takes WLM out of the picture and a more true "Round Robin" is possible.



Round Robin and WLM-less Queuing

For cases where you want to control work distribution to multiple servants, consider WLM-less queueing ... uses WAS queues, WLM plays minimal role

Two environment variable controls:

`server_use_wlm_to_queue_work`	0
`server_work_distribution_algorithm`	0 | 1 | 2

WAS internal queues
Control Region
zWLM
Servant Region
Servant Region
Servant Region

0 - Hot Thread

WAS looks for first available thread starting with first servant. First servant utilized the most, then second, etc.

1 - Round Robin

True round robin. If thread not available in targeted servant then work gets put into servant queue and waits for thread in that servant to free up.

2 - Hot Robin

Round Robin with a twist: if thread not available in targeted servant, then work put in queue available to all servants. First servant to free a thread gets work.

Notes:

- Stateless or stateful applications ... both work with this
- WLM will still maintain minimum and restart failed
- WLM will not start additional servants
- WLM will still classify work, but it will not get involved in placement
- This works best when all the work in the server has the same Service Class

InfoCenter `urun_rproperty_custproperties`

WLM start servants? ...

11

IBM Americas Advanced Technical Skills
Gaithersburg, MD

© 2013 IBM Corporation

Because of this desire to truly round-robin stateless work, there's a function in WAS z/OS to take WLM out of the middle and use a set of internal work queues instead. Some notes on this:

- This is configurable down to the server level, so you may have some servers doing this and some servers remaining with WLM
- This will work with stateful applications as well. Affinities are established and honored. It's just WLM isn't involved in that.
- If you use this feature you should understand that WLM will not start additional servants. It'll restart failed ones but not start additional ones.
- There's still work classification going on, but in this case WLM stays out of the placement of that work. To WLM it looks like the servants aren't doing anything. They are, of course ... it's just that WAS is taking care of it rather than having WLM take care of it.
- Two environment variables -- one to take WLM out of the mix, one to indicate the work placement model.
- Three options -- Hot Thread, Round Robin and Hot Robin.

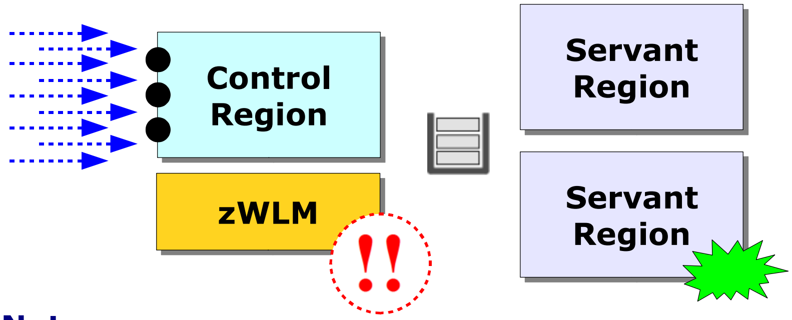
  Hot Thread - WAS looks for first available thread staring in first servant. If the work can be handled with the threads in the first servant, that'll be the only servant used. Work spills to the second only if there are no threads available in the first at the time of request arrival.

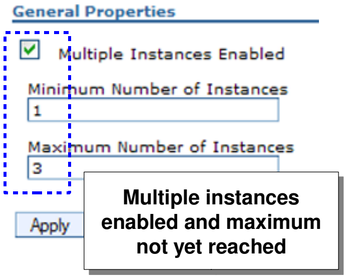
  Round Robin - True round robin format. Work is placed on the queue of the next servant in line. If there's no thread currently available in that servant the work is queued and it waits for a thread to open. This option will result in the most even distribution of work.

  Hot Robin - WAS works around the circle of servants in round robin fashion. If a work request is targeted for a servant and that servant does not have an available thread at that moment, then WAS puts the work on a queue that's available to all servants. The first servant to have a thread open up gets the work. On average this should result in fairly even distribution. The objective is to round-robin as much as possible but leave open the option to flow work to a servant that can do the work.

Will WLM Start Servant for Spike in Work?

Yes ... again provided the MAX value has not yet been met. If incoming work can't meet goals with one servant, another started:




General Properties

☑ Multiple Instances Enabled

Minimum Number of Instances: 1

Maximum Number of Instances: 3

Multiple instances enabled and maximum not yet reached

Notes:

- **WLM uses very sophisticated algorithms ... exactly when it will start the next servant region is not always easy to predict. *Not* based on simple rule of work queue depth.**
- **WLM will shut down extra servant if not needed, but it's very conservative about eliminating resources once created. You may MODIFY the MAX down to close unneeded servant regions if you wish.**
- **When second servant is up the work is *not necessarily* distributed evenly as we just saw**
- **If you're using WLM-less queueing then servant will *not* be automatically started**

Essentials of classification ...

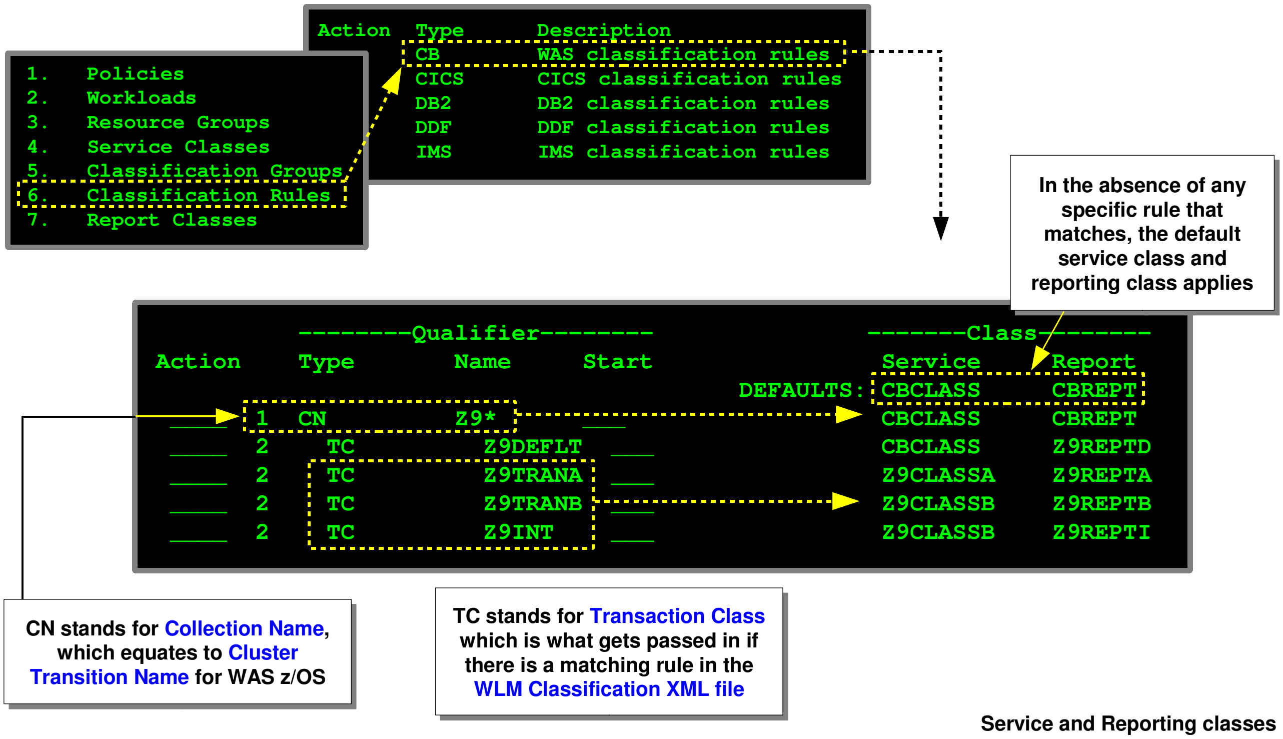
If we go back to the case where WLM **is** in the mix (in other words, the WLM-less queuing method we just spoke of is *not* configured), then WLM can dynamically start a servant when the work exceeds the ability of the available servants to process the work. But here we have to caution you -- it may not take place when you think it would.

When WLM might start a servant region is based on a set of WLM algorithms that tries to take into account the work facing a server and balance it against the "cost" to start up an additional resource. If the spike in work is very short in duration then WLM will most likely *not* start an additional servant. WLM is a bit cautious about firing up address spaces ... it wants to first see if the work spike is more than just short-term temporary. So the algorithm is more than merely the depth of the queue between CR and SR.

On the flip side of this ... if WLM sees a servant that's no longer needed it *can* shut that servant down, but again ... it's cautious about this, perhaps more cautious with eliminating resources once started than with starting them in the first place. If you have a servant you think is not needed and you'd like to eliminate it, you can MODIFY the maximum value so it's lower than the currently started servants. This sends a signal to WLM to close a servant. It will do so when it sees all the work is flushed out of that servant.

The Essentials of WLM Classification

All WAS z/OS requests get classified. How that's done depends on how the "CB Rules" are coded in WLM:



```
1.   Policies
2.   Workloads
3.   Resource Groups
4.   Service Classes
5.   Classification Groups
6.   Classification Rules
7.   Report Classes
```

```
Action  Type       Description
        CB         WAS classification rules
        CICS       CICS classification rules
        DB2        DB2 classification rules
        DDF        DDF classification rules
        IMS        IMS classification rules
```

In the absence of any specific rule that matches, the default service class and reporting class applies

```
           --------Qualifier--------                  -------Class-------
Action     Type        Name       Start               Service       Report
                                            DEFAULTS: CBCLASS       CBREPT
 ____    1  CN          Z9*         ___               CBCLASS       CBREPT
 ____    2    TC          Z9DEFLT   ___               CBCLASS       Z9REPTD
 ____    2    TC          Z9TRANA   ___               Z9CLASSA      Z9REPTA
 ____    2    TC          Z9TRANB   ___               Z9CLASSB      Z9REPTB
 ____    2    TC          Z9INT     ___               Z9CLASSB      Z9REPTI
```

CN stands for Collection Name, which equates to Cluster Transition Name for WAS z/OS

TC stands for Transaction Class which is what gets passed in if there is a matching rule in the WLM Classification XML file

Service and Reporting classes ...

As we mentioned earlier, *all work* gets classified by WLM. Now the question is *what* WLM uses to know how to do that classification. And the answer is rules found under the "CB" classification rule type.

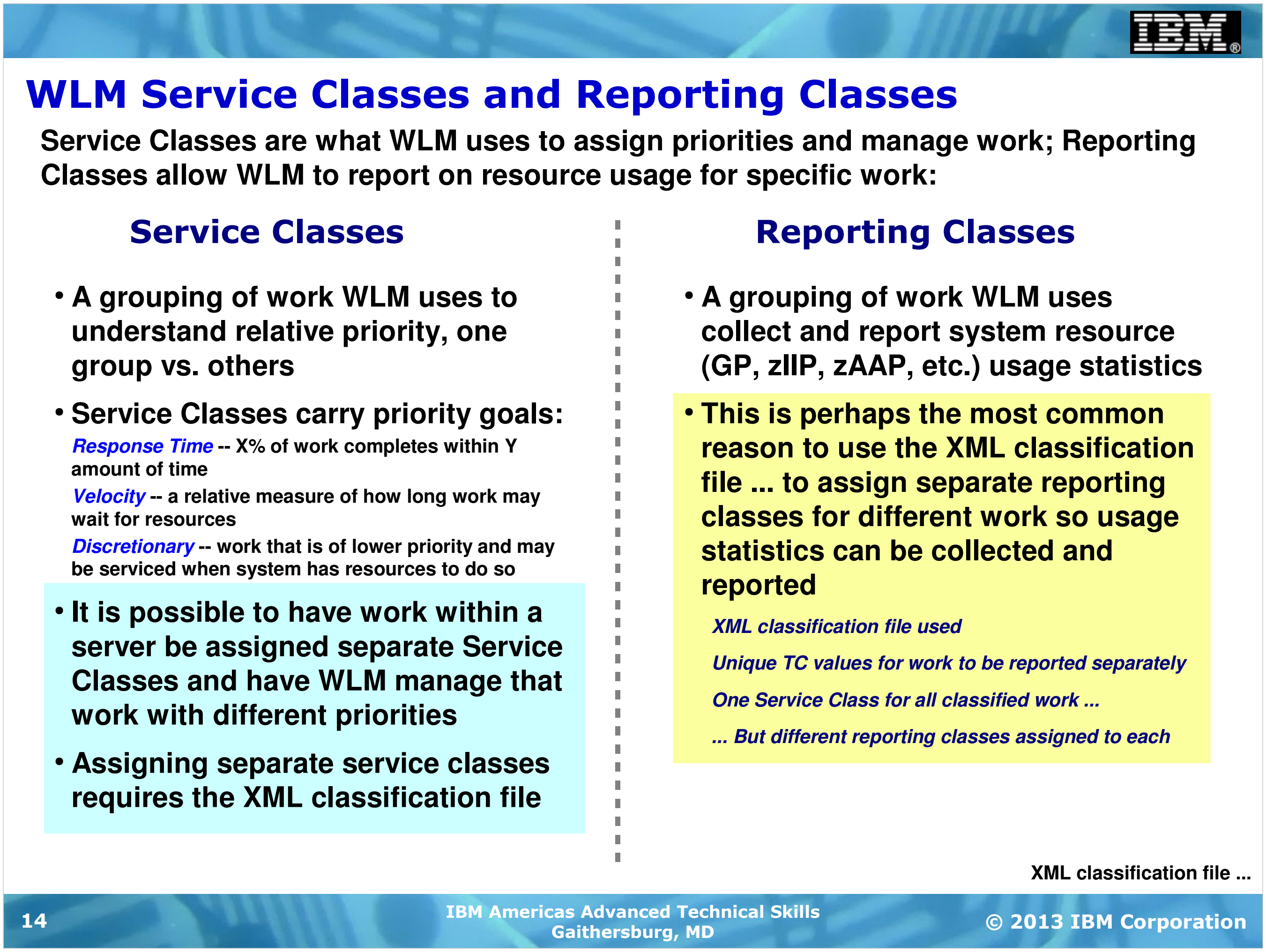
Within the CB type classification rules there's a qualifier type of "CN." CN stands for "Collection Name" and that refers to the "Cluster Transition Name" for a server. In this workshop the servers we use all start with the letters "Z9," which matches the rule shown here. So, in the absence of any transaction class information work in the Z9 servers will carry a Service Class of `CBCLASS` and a reporting class of `CBREPT`.

Imagine the `Z9*` rule was *not* there ... what then? If no rule matches then WLM falls up to the "DEFAULTS" specified at the top. In this example it's the same as the rule for `Z9*` ... `CBCLASS` and `CBREPT`.

The TC qualifier type refers to "transaction class" information passed to WLM by the server. The only way TC information is passed to WLM is when the XML classification file is in effect. We'll talk about that next. But here's a hint -- the XML classification file provides a way to identify work -- for example, with a URI matching pattern -- and then pass a transaction class name into WLM based on that.

If a TC is passed in, and it matches a rule, then that work will carry the associated service class and reporting class coded for that rule. In the example above you see that TC is subordinate to the CN ... which means the rule here is really "If CN=Z9* *and* TC=<*tc_name*>." It is possible to have TC= be a first-level rule so the match is on the TC name only with CN not playing a part. This example shows TC as a sub-rule under CN.

This is how different work within the same server might carry different service classes or different reporting classes. The XML classification file contains rules that map requests to transaction classification names. Those names are passed into WLM. WLM attempts to match them to TC= rules. If a match is found, then the associated service class and reporting class is assigned.



WLM Service Classes and Reporting Classes

Service Classes are what WLM uses to assign priorities and manage work; Reporting Classes allow WLM to report on resource usage for specific work:

Service Classes

- A grouping of work WLM uses to understand relative priority, one group vs. others
- Service Classes carry priority goals:

  Response Time -- X% of work completes within Y amount of time

  Velocity -- a relative measure of how long work may wait for resources

  Discretionary -- work that is of lower priority and may be serviced when system has resources to do so
- It is possible to have work within a server be assigned separate Service Classes and have WLM manage that work with different priorities
- Assigning separate service classes requires the XML classification file

Reporting Classes

- A grouping of work WLM uses collect and report system resource (GP, zIIP, zAAP, etc.) usage statistics
- This is perhaps the most common reason to use the XML classification file ... to assign separate reporting classes for different work so usage statistics can be collected and reported

  XML classification file used

  Unique TC values for work to be reported separately

  One Service Class for all classified work ...

  ... But different reporting classes assigned to each

XML classification file ...

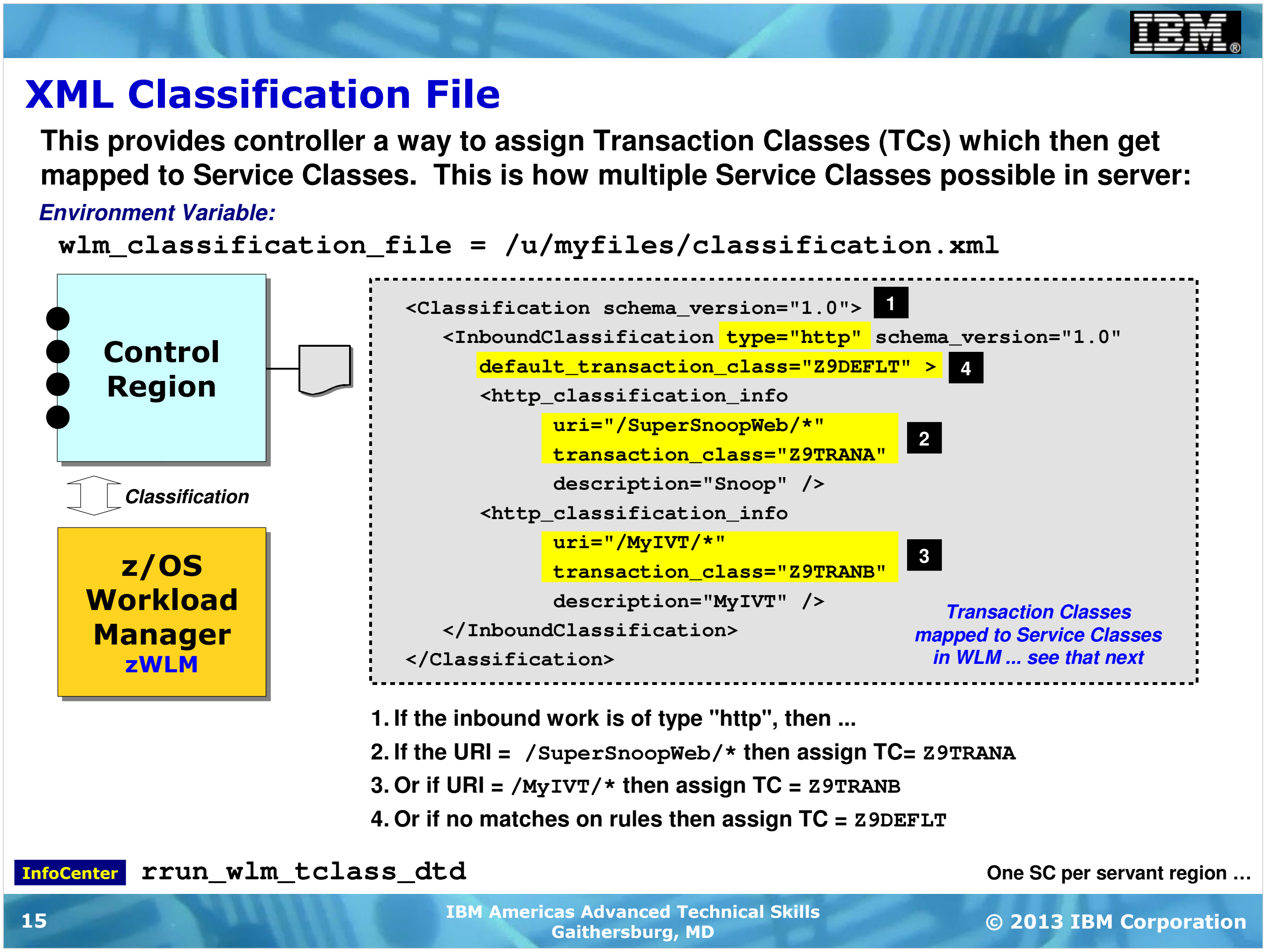
The previous chart had two terms used that requires a bit of explanation to those not already familiar with them -- Service Classes and Reporting Classes. Both are related to z/OS WLM and how it operates.

Service Classes are a way WLM groups work so it may manage system resources so the group gets what it needs to meet the goals defined for that service class. So think of a "service class" like a bucket in which work is categorized, and all the work in that bucket is deemed of similar priority. Multiple service classes may exist, which allows the administrator to provide different "buckets" of classified work.

These service classes carry a defined *goal*, which is a way WLM determines how to prioritize access to system resources. Goals may be defined in several ways -- response time, velocity or discretionary. A response time goal of "99% complete with 1/10th of a second" is a very aggressive goal; "75% within 5 seconds" less aggressive. Between those two WLM would know to prioritize the first higher than the second. Velocity is expressed as a number between 1 and 100, with 100 being the "fastest" velocity. WLM uses this number to get a handle on how long work may be allowed to wait for system resources. Velocity=100 work can't wait long; Velocity=50 can wait some time if necessary.

Note: it's important to understand that WLM gets involved with the allocation of system resources only when there is competition for the resources. If you have a wonderfully provisioned machine -- lots of CPU, lots of memory, not so much work -- then everyone is happy and WLM doesn't really need to think about relative priorities and balancing access to limited resources. Conversely, it's important to understand that if WLM is told to view everything as highest priority, then nothing has priority when things are running tight.

Reporting Classes on the other hand provide a way for WLM to categorize work so another tool, RMF, can sweep through all the usage statistics and report back the resource usage for work in the reporting class. As stated on the chart, this is the most common use for the XML classification file -- the assignment of work to different reporting classes (but only one service class) for the purpose of being to report on how much each group of work consumed.



XML Classification File

This provides controller a way to assign Transaction Classes (TCs) which then get mapped to Service Classes. This is how multiple Service Classes possible in server:

Environment Variable:

```
wlm_classification_file = /u/myfiles/classification.xml
```

```
<Classification schema_version="1.0">
   <InboundClassification type="http" schema_version="1.0"
      default_transaction_class="Z9DEFLT" >
      <http_classification_info
            uri="/SuperSnoopWeb/*"
            transaction_class="Z9TRANA"
            description="Snoop" />
      <http_classification_info
            uri="/MyIVT/*"
            transaction_class="Z9TRANB"
            description="MyIVT" />
   </InboundClassification>
</Classification>
```

Transaction Classes mapped to Service Classes in WLM ... see that next

1. If the inbound work is of type "http", then ...
2. If the URI = `/SuperSnoopWeb/*` then assign TC= `Z9TRANA`
3. Or if URI = `/MyIVT/*` then assign TC = `Z9TRANB`
4. Or if no matches on rules then assign TC = `Z9DEFLT`

InfoCenter `rrun_wlm_tclass_dtd`

One SC per servant region ...

15 IBM Americas Advanced Technical Skills Gaithersburg, MD © 2013 IBM Corporation

The XML classification file is used to introduce to the server rules it may use to map requests to a "Transaction Class" (TC), which is then used in WLM CB rules to map the TC to a Service Class. The XML Classification File is the starting point for mapping multiple Service Classes into a multi-servant server, and as you'll soon see it is the foundation for the granular control function we'll discuss at the end of this unit.

By default there is no XML classification file defined to a WAS server. You enable to that by explicitely coding an environment variable, which provides the pointer to the file. The WAS server must have READ authority to the file. The file must be stored in USS as an ASCII file.

The chart shows a valid-but-simplified example of a classification file. The numbered blocks correspond to the following explanations:

1. The `InboundClassification` tag is used to start a section for work classification. That carries with it a `type` specification. Here we're showing http because that's what most people think of when they think of work for WAS servers. Other options include `iiop`, `mdb`, `sip`, `ola` and `internal`. For example, if you had http and mdb work, the classification file would have two `InboundClassification` sections.
2. For any given piece of work received we wish to attempt to identify it. For HTTP this is commonly done with a pattern-match on the URI (URI is the portion of the URL that follows host and port). The example above is showing the case where a match on `/SuperSnoopWeb/*` (yes, wildcards acceptable) would mean the assignment of the Transaction Class `Z9TRANA` to this request. We'll see on the next chart how that TC maps to a Service Class.
3. This shows a separate work classification rule, using `uri=` but a different string to match. It results in the assignment of a TC of `Z9TRANB`.
4. In the event no classification rule applies, then a default transaction classification value of `Z9DEFLT` is assigned.

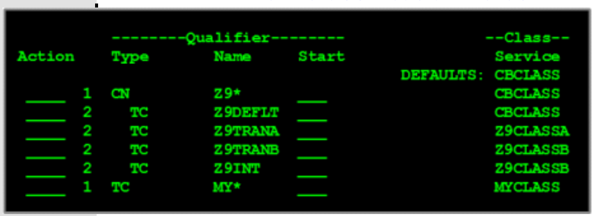
Note: there are many more possibilities than what's shown on this chart. Note the InfoCenter search string in the lower left of the chart. That article has full details on the options supported by the classification document.

One Service Class Per Servant Region

There's a subtle "gotcha" with respect to WAS internal work. This provides us a good opportunity to review planning for the number of servant regions you need:

```
<Classification schema_version="1.0">
   <InboundClassification type="http" schema_vers
      default_transaction_class="Z9DEFLT" >
      <http_classification_info
            uri="/SuperSnoopWeb/*"
            transaction_class="Z9TRANA"
            description="Snoop" />
      <http_classification_info
            uri="/MyIVT/*"
            transaction_class="Z9TRANB"
            description="MyIVT" />
   </InboundClassification>
</Classification>
```

CB Classification Rules



```
              --------Qualifier--------                --Class--
Action    Type        Name      Start                   Service
                                            DEFAULTS:  CBCLASS
____   1  CN          Z9*       ____                   CBCLASS
____   2    TC        Z9DEFLT   ____                   CBCLASS
____   2    TC        Z9TRANA   ____                   Z9CLASSA
____   2    TC        Z9TRANB   ____                   Z9CLASSB
____   2    TC        Z9INT     ____                   Z9CLASSB
____   1  TC          MY*       ____                   MYCLASS
```

The URI classification rules suggest **at least** two servant regions

Servant Region Z9CLASSA

Servant Region Z9CLASSB

What happens if another application with a different URI is in the server?

It gets the default TC and in this example gets mapped to CBCLASS

Servant Region CBCLASS

Third servant needed

WAS internal work will be classified, and in the absence of a rule it gets the CN default ... CBCLASS

Thus it might be a good idea to take internal work into account in the XML classification rules

First ... Will WLM start servant for service class? ...

If you use the XML file to assign work to service classes, WLM will seek to "bind" a servant to a service class. That means the maximum number of servants you have configured needs to be equal or larger than the maximum number of Service Classes you expect to get assigned.

Note: we'll show you how to account for the internal work in a few charts. First a review of the planning considerations that bubble up based on the configuration XML.

Take the example above. The XML has two explicit transaction classifications and one default. The two explicit TCs map to separate Service Classes in the CB rules (`Z9CLASSA` and `Z9CLASSB`). If you know for certain those two applications are going to be deployed, you know that you need *at least* two servant regions. What happens if another application sneaks into that server? Or if one of the applications has URI that's a little different from the patterns provided? Well, then it's possible the default transaction class will apply and that means a Service Class of `CBCLASS` based on the CB rules. That suggests the *possibility* of needing three servant regions.

One option would be to make sure the default TC was the same value as specified for one of the specific URI patterns. Then you'd be back to two servant regions. Another option would be to change the CB rule so the default TC of `Z9DEFLT` mapped to one of the two Service Classes (`Z9CLASSA` and `Z9CLASSB`). That too would bring you back to needing only two servant regions.

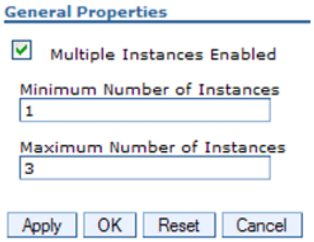
The main points here are:

- It's a good idea to think about the relationship between your classification rules and the number of servant regions that will be needed
- Be aware of the WAS internal work because if you don't specifically account for it (which we'll show you how to do shortly), it ends up picking up the default TC and that may imply binding a servant to that internal work and preventing your planned-for work having a place to go.

A question comes up ... let's say a third Service Class does sneak in. If MIN=2 and MAX=3 will WLM start another servant to handle it? Answer: yes ...

Will WLM Start Servant for a New Service Class?

Yes ... provided the MAX value has not yet been met, a Service Class that comes in without a place to go will result in the dynamic start of an additional servant:

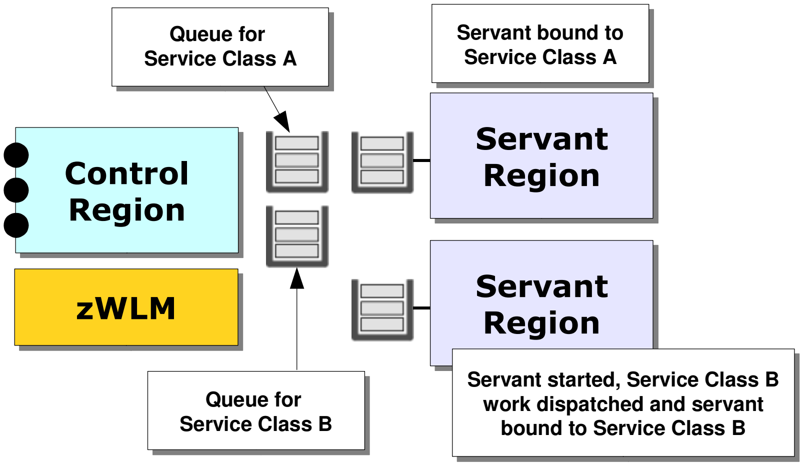


General Properties

☑ Multiple Instances Enabled

Minimum Number of Instances: 1

Maximum Number of Instances: 3



Queue for Service Class A

Servant bound to Service Class A

Control Region

Servant Region

zWLM

Servant Region

Queue for Service Class B

Servant started, Service Class B work dispatched and servant bound to Service Class B

Notes:

- **Server most likely won't have multiple Service Classes unless XML Classification File used. If you're not using that then most likely only one Service Class will be present.**
- **While servant is starting work waits in queue. Possibility exists it could timeout waiting in that queue**
- **Probably don't want to use dynamic expansion for this purpose. Understand your Service Classes and plan for MIN value accordingly.**

How to account for internal work ...

Suppose you have a server with "Multiple Server Instances" enabled and MIN=1 and MAX=3 configured. Suppose two servants are presently active with separate Service Classes bound to each. Now suppose a third Service Class comes into the picture -- either unanticipated or one you expected might show up and it finally has. In this case WLM will queue the work to a Service Class queue and start the third and final servant region. While that servant is coming active the work stays in the queue. Depending on how long it takes for that servant to come active that work in queue may time out.

Generally speakign you probably do not want to start servants dynamically for each Service Class. Better to anticipate the number of Service Classes and have the MIN=*n* value set accordingly.

The final point here is for those who may be wondering how many Service Classes may be present in their environment. The rule of thumb is this -- if you're not using the XML Classification File then the answer is most likely "one." There are obscure exceptions but it's better not to focus on them.

How to Account for Internal Work

There *will* be internal work classified. How can you account for it without it simply falling under the CN default Service Class?

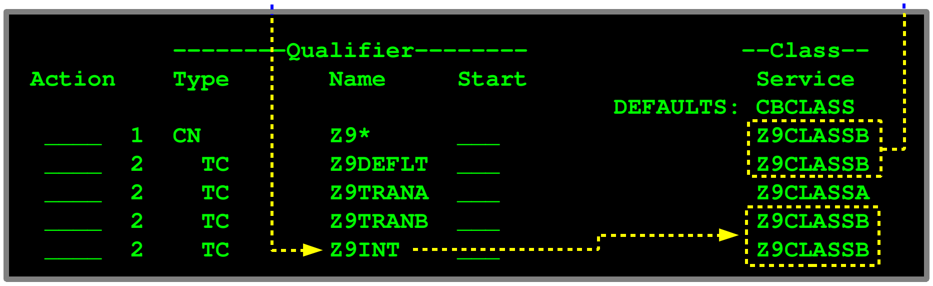
```
<Classification schema_version="1.0">
   <InboundClassification type="http" schema_version="1.0"
      default_transaction_class="Z9DEFLT" >
      <http_classification_info
            uri="/SuperSnoopWeb/*" transaction_class="Z9TRANA"
            description="Snoop" />
      <http_classification_info
            uri="/MyIVT/*" transaction_class="Z9TRANB"
            description="MyIVT" />
   </InboundClassification>
  <InboundClassification type="internal" schema_version="1.0"
     default_transaction_class="Z9INT" >
   </InboundClassification>
</Classification>
```

"http" is one of several inbound work types:

```
http    internal
iiop    mdb
sip     ola
```

Account for internal work as shown. Then map to a TC you know will be used by one of your other rules.

Do same for the default TC and the CN default and you then have all cases covered.



```
         --------Qualifier--------              --Class--
Action   Type        Name      Start             Service
                                      DEFAULTS: CBCLASS
____     1 CN        Z9*       ___              Z9CLASSB
____     2   TC      Z9DEFLT   ___              Z9CLASSB
____     2   TC      Z9TRANA   ___              Z9CLASSA
____     2   TC      Z9TRANB   ___              Z9CLASSB
____     2   TC      Z9INT     ___              Z9CLASSB
```

You can assign separate reporting classes to isolate out the internal work and get numbers on each service class

AE_SPREADMIN ...

Earlier we mentioned that the XML Classification File is capable of supporting work types other than just HTTP. We started with HTTP as the example simply because it's the most common type of work people think about with WAS. But other types are supported as well -- `iiop`, `sip`, `mdb`, `ola` and ... *`internal`*.

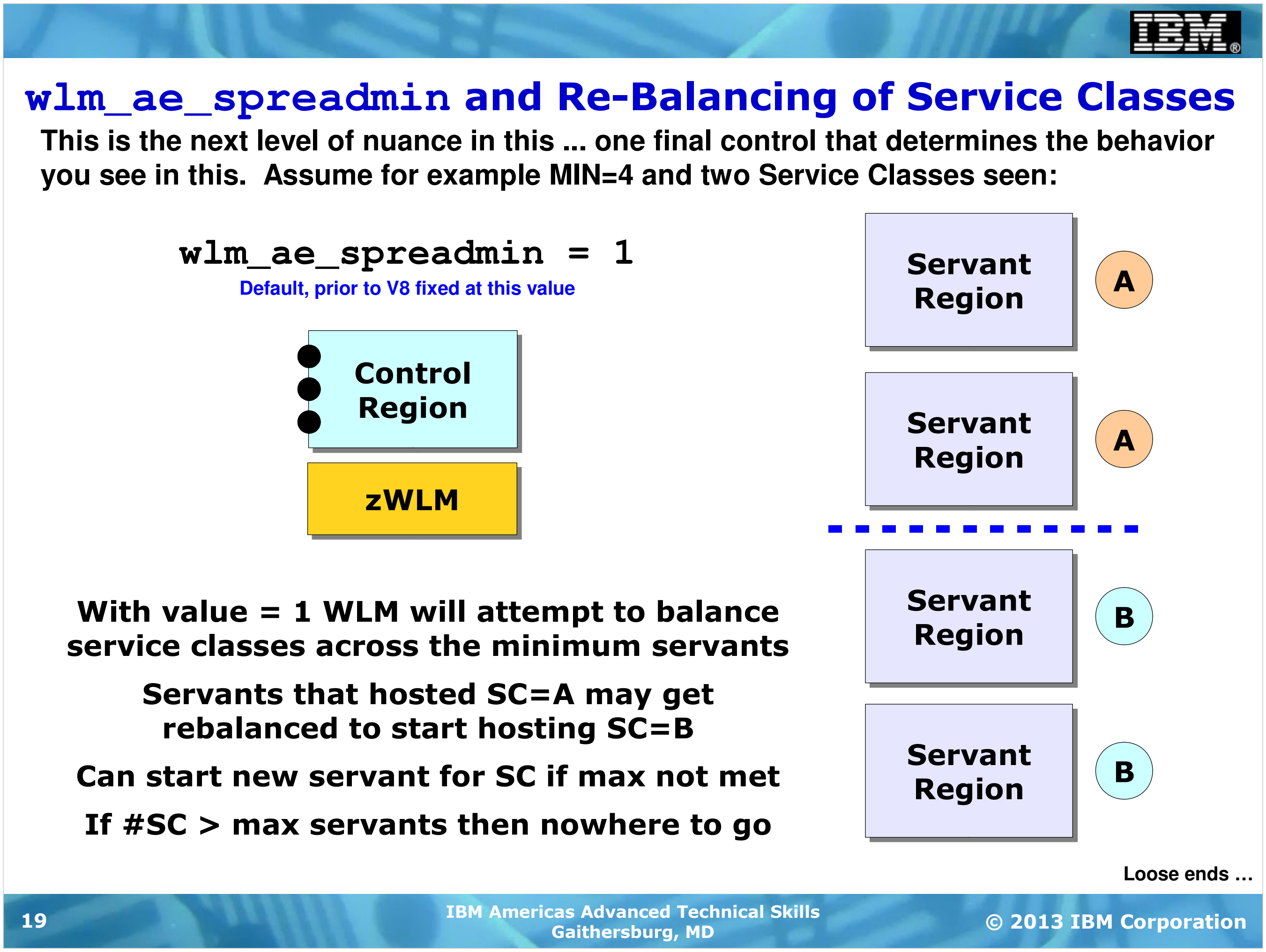
The `type="internal"` value provides us a way to explicitely account for internal work. The example shows a separate `InboundClassification` section with `type="internal"` that maps internal work to a TC of `Z9INT`. Just like other TC values this gets mapped to a Service Class in the CB rules.

You could map the internal work to a separate Service Class in the CB rules, but that would mean the internal work would get its own servant region. There's not enough internal work to justify that. Better to assign the internal work to one of your other planned-for Service Classes. The example shows assigning `Z9INT` to the `Z9CLASSB` Service Class. Yes, that means internal work and the other work carrying the `Z9TRANB` TC get mixed together. Again, there's not a lot of internal work so that should not hurt performance. If you're concerned about having clean reporting information then the answer is to assign a separate *Reporting Class* (not shown on this chart) to the `TC=Z9INT` work. That will allow `Z9TRANB` to have a separate reporting class and the reporting numbers stay clean and separate.

The example above is showing Service Class `Z9CLASSB` assigned to several things:

- `CN=Z9*` -- With an XML file in place you might think this would never be used because of the default TC in the file. But it's possible some other "type" of work (IIOP, for example) could arrive. If you don't have XML to account for the IIOP work then it would not get a TC and would thus fall back to this Service Class.
- `TC=Z9DEFLT` -- This will pick up any unaccounted-for work for the work "type" ... in this case HTTP.
- `TC=Z9TRANB` -- Our explicit TC for the work matching the URI pattern.
- `TC=Z9INT` -- For internal work, as we've noted.

With this in place you've accounted for the contingencies. Then all you'd need is a minimum of two servant regions. You could have more, of course. And it is possible to have a Service Class bind to multiple servants. There's a lot of combinations and possibilities to all of this.



wlm_ae_spreadmin and Re-Balancing of Service Classes

This is the next level of nuance in this ... one final control that determines the behavior you see in this. Assume for example MIN=4 and two Service Classes seen:

`wlm_ae_spreadmin = 1`

Default, prior to V8 fixed at this value

With value = 1 WLM will attempt to balance service classes across the minimum servants

Servants that hosted SC=A may get rebalanced to start hosting SC=B

Can start new servant for SC if max not met

If #SC > max servants then nowhere to go

Loose ends ...

19

IBM Americas Advanced Technical Skills
Gaithersburg, MD

The final bit of nuance we'll show you has to do with an environment variable called `wlm_ae_spreadmin`. As the name of that variable suggests, the value influences how WLM attempts to spread the application environments ("ae") across the minimum servants.

Assume you have a value of MIN=4 for servant regions. Assume further that for WLM sees two service classes in the server. When `wlm_ae_spreadmin` is set to 1 then WLM will split the minimum servants into two equal groups, one for each service class seen.

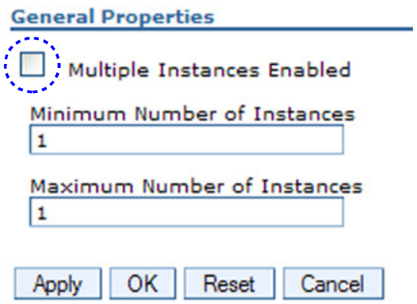
But WLM can't know in advance how many service classes it might see, so this dividing up of the minimum servant regions doesn't occur when the server starts. At first, if only one service class is seen (for example, internal work), it's possible all the servants get bound to that one service class. But when the second service class is seen then WLM divides the pool of minimum servants. At this point some rebalancing of work assignments may very well take place. And at this point there may well be two different service classes at work in the same servant. Over time WLM works to get the service classes balanced.

This is when the value for this environment variable is "1". In V8 that environment variable was opened up so you can set it to "0". If the value is "0" that tells WLM to do what it believes it must do to meet goals. That means it's possible the allocation of servants to service classes might be (in tihs example) 3 to 1 ... or whatever WLM sees is necessary.

Let's change the example. Imagine MIN=2 with the value of `wlm_ae_spreadmin` set to 1. Two service classes are in play and WLM has one servant for SC=A and one for SC=B. Then SC=C comes in. What happens? If the maximum servants is not yet met then a third servant region is started. If max has been met, then that new SC has nowhere to go. That's a condition you don't want to see happen. Which is why planning for the maximum number of service classes possible is important.

Tying up Loose End -- Multiple Servant Instances

There's a very subtle configuration scenario you should be aware of ...



General Properties

☐ Multiple Instances Enabled

Minimum Number of Instances: 1

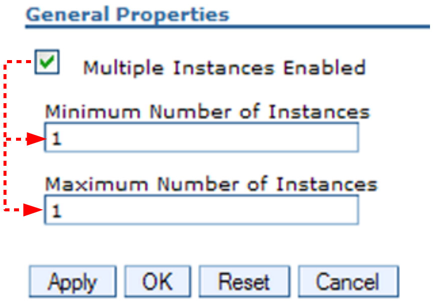
Maximum Number of Instances: 1

This is a true single servant environment

This will allow multiple Service Classes to co-mingle in the same servant

If you really want just one servant, this is the way to configure it.

However, can't use MODIFY to expand.



General Properties

☑ Multiple Instances Enabled

Minimum Number of Instances: 1

Maximum Number of Instances: 1

This is a multi-servant environment with only one servant

This restricts servant to one Service Class

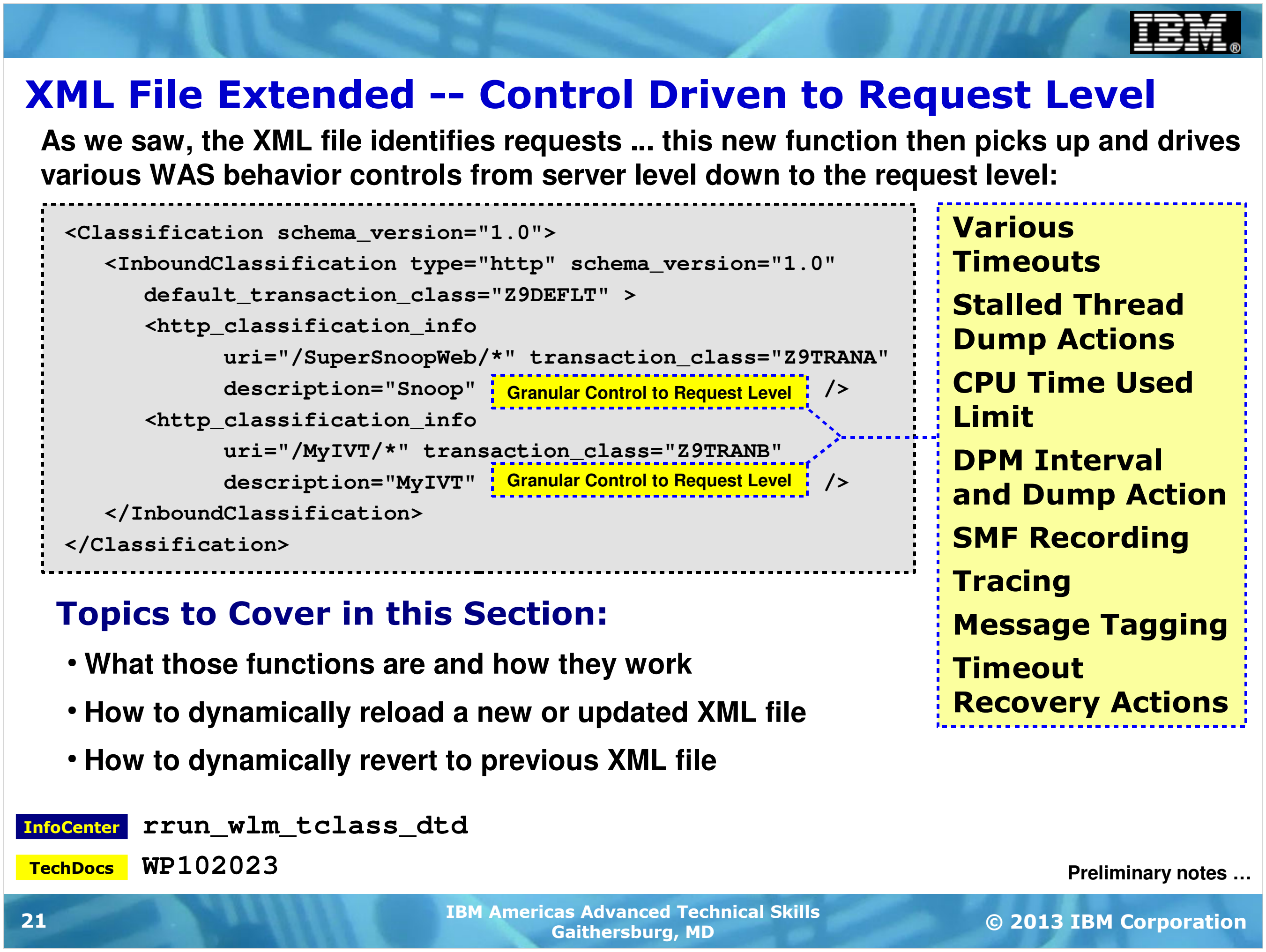
Generally not recommended unless you're very certain about the Service Classes in use. Better to specify MAX greater than MIN to give WLM ability to process other work if needed.

You do have opportunity to MODIFY MIN and MAX higher, however.

Setting stage for Granular RAS ...

We need to tie up a loose end here ... and it has to do with a little checkbox on the configuration panel for servant instances. If that box is *unchecked* then WAS z/OS sees this as a true single-server environment. In that case multiple Service Classes *can* be mapped into the single servant. WLM will allow it.

However, if the box is checked and the MIN and MAX are set to 1 then WAS z/OS does not see this as a true single servant environment. The first Service Class in will find that servant and no other Service Classes will find room to work. You could use MODIFY to expand the MIN and MAX values. In general this setting is not recommended unless you're very certain of the Service Classes you have in play.



XML File Extended -- Control Driven to Request Level

As we saw, the XML file identifies requests ... this new function then picks up and drives various WAS behavior controls from server level down to the request level:

```
<Classification schema_version="1.0">
   <InboundClassification type="http" schema_version="1.0"
      default_transaction_class="Z9DEFLT" >
      <http_classification_info
            uri="/SuperSnoopWeb/*" transaction_class="Z9TRANA"
            description="Snoop"  Granular Control to Request Level  />
      <http_classification_info
            uri="/MyIVT/*" transaction_class="Z9TRANB"
            description="MyIVT"  Granular Control to Request Level  />
   </InboundClassification>
</Classification>
```

Various Timeouts

Stalled Thread Dump Actions

CPU Time Used Limit

DPM Interval and Dump Action

SMF Recording

Tracing

Message Tagging

Timeout Recovery Actions

Topics to Cover in this Section:

- **What those functions are and how they work**
- **How to dynamically reload a new or updated XML file**
- **How to dynamically revert to previous XML file**

InfoCenter rrun_wlm_tclass_dtd

TechDocs WP102023

Preliminary notes ...

This new function in WAS z/OS V8 (it's exclusive to z/OS) is on the surface an extension of the XML Classification File. But what it does is quite different from merely assigning Transaction Classes to requests.

Think about what the XML Classification File does -- it provides WAS z/OS a mechanism to identify inbound work by criteria of your design. Well ... if you have the request identified at the point of classification then it's possible to have WAS do *other* things at the request level as well. So that's what we'll explore in this unit -- how the XML file is extended to include additional keyword values so functions such as tmeouts, CPU limits, SMF recording, and tracing can be controlled at the identified request level rather than just at the server level.

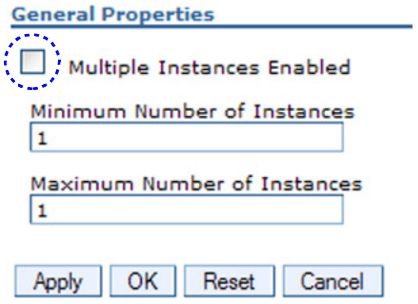
A Few Preliminary Notes

To use the granular control features implies classifying work with transaction classes as well ...

```
<Classification schema_version="1.0">
   <InboundClassification type="http"
      schema_version="1.0"
      default_transaction_class="AAA" >
      <http_classification_info
            uri="/SuperSnoopWeb/*"
            transaction_class="AAA"
            description="Snoop"
             New Function    />
      <http_classification_info
            uri="/MyIVT/*"
            transaction_class="AAA"
            description="MyIVT"
             New Function    />
   </InboundClassification>
</Classification>
```

If you don't wish to use multiple transaction classes then code all the TCs in the XML with the same value

If the CB rules in WLM don't have TC rules then other defaults will apply



General Properties

□ Multiple Instances Enabled

Minimum Number of Instances: 1

Maximum Number of Instances: 1

This new function does not require multiple servants, even if two or more Service Classes at work

WLM will place different Service Classes in servant if the server is true single server

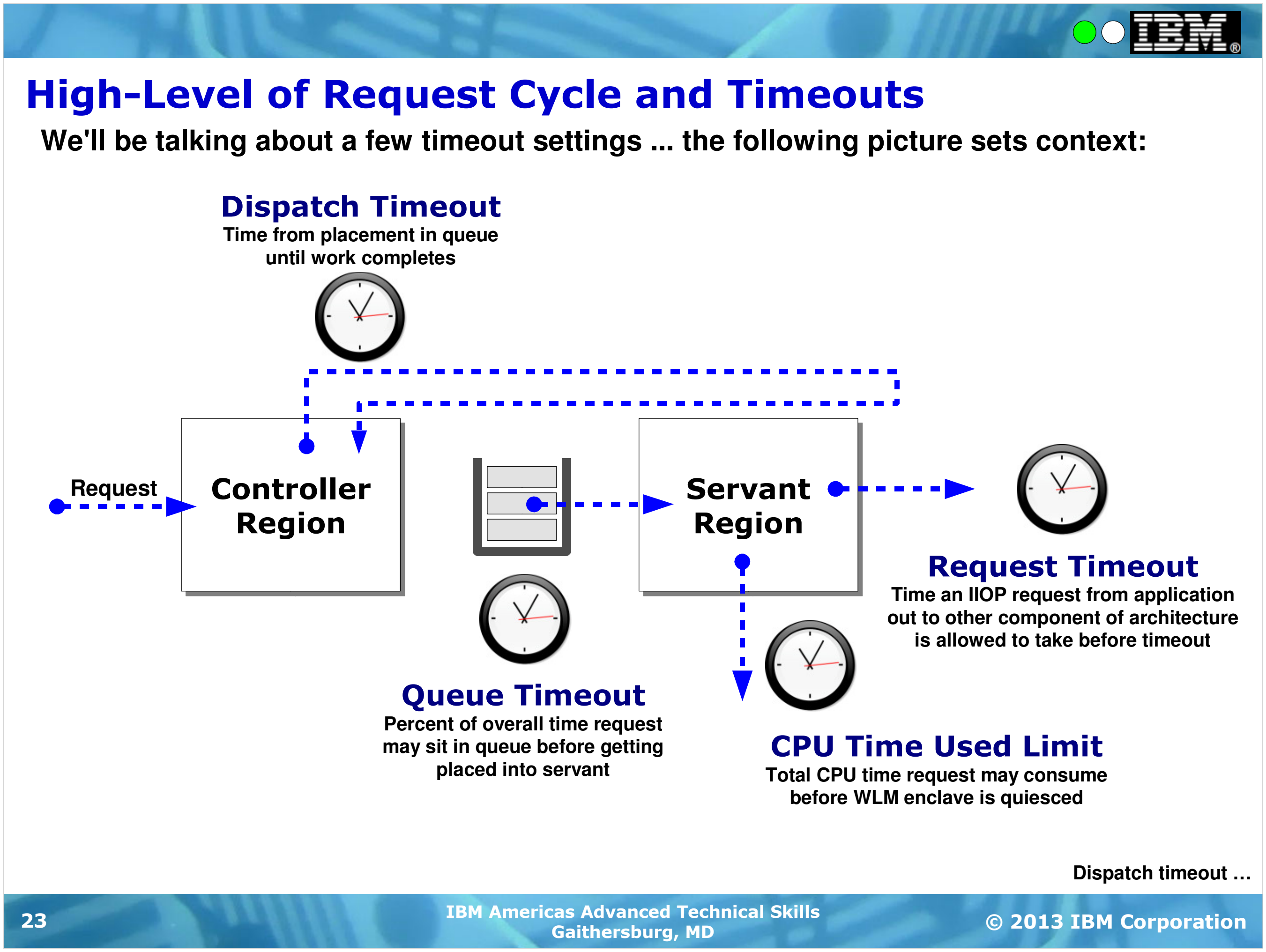
Request cycle ...

Coming right after all that talk about multiple servant regions and multiple Service Classes you might think this new function requires all of that as well ... but that would *not* be the case.

Service Classes --The XML Classification File does require that a transaction class value be specified. If your interest is in the granular function and not multiple TCs and multiple Service Classes, then you may code each transaction class in the XML file as the same thing. Then in the CB rules you can map this TC to a single Service Class. Or not code any TC rule and allow the CN= rule to apply.

Servant Regions -- This new function does not require multiple servants. It will work with multiple servants but it does not require it. That means if your standard configuration is a single servant region that's perfectly okay with this new function we're about to see.

The message here is that the granular control function is actually quite separate from all the Service Class and multiple servant stuff we just worked through. With one key exception -- this new granular control function requires the use of the XML classification file. That's because the XML file provides the key starting point -- *the identification of the request*. With the request identified, then WAS z/OS can drive the control behaviors down to that request. Before the finest level of control was server, now it's at the request level.



High-Level of Request Cycle and Timeouts

We'll be talking about a few timeout settings ... the following picture sets context:

Dispatch Timeout
Time from placement in queue until work completes

Queue Timeout
Percent of overall time request may sit in queue before getting placed into servant

CPU Time Used Limit
Total CPU time request may consume before WLM enclave is quiesced

Request Timeout
Time an IIOP request from application out to other component of architecture is allowed to take before timeout

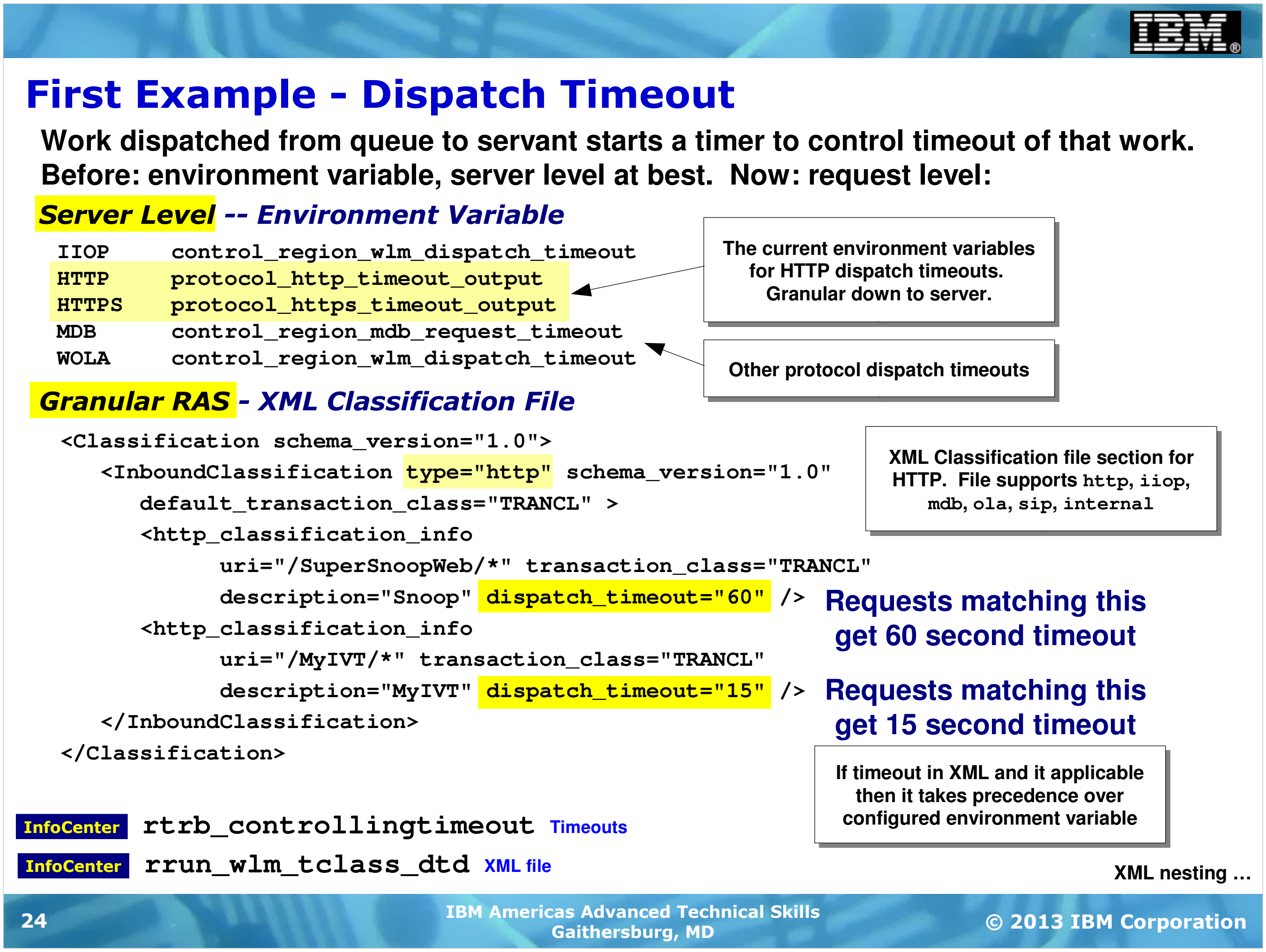
Dispatch timeout ...

Over the next several charts we're going to discussing a few different timeouts that now may be set down at the request level. A picture helps set the context for those timeouts.

Note: WebSphere Application Server has other timeouts beyond what's shown on this chart. What we're discussing here are timeouts that have been made configurable in the classification XML file as part of this new V8 "granular RAS" function.

Four timeout values will be discussed:

- **Dispatch Timeout** -- this is a timer that clocks the time from placement of request on WLM queue to completion of request. Consider this a kind of "total time" timer.
- **Queue Timeout** -- this is a timer that clocks the time a request sits in the WLM queue before getting picked up by the servant region. This is expressed as a percent of "dispatch timeout" (total time).
- **CPU Time Used Limit** -- this is not really a timer per se; it is a measure of how many CPU milliseconds a request may consume before being considered "runaway." When this value is exceeded the WLM enclave is quiesced, which results in the thread being set at the lowest possible priority.
- **Request Timeout** -- this is a timer that clocks the time an *outbound* IIOP request from your program in the servant may take.



First Example - Dispatch Timeout

Work dispatched from queue to servant starts a timer to control timeout of that work. Before: environment variable, server level at best. Now: request level:

Server Level -- Environment Variable

```
IIOP     control_region_wlm_dispatch_timeout
HTTP     protocol_http_timeout_output
HTTPS    protocol_https_timeout_output
MDB      control_region_mdb_request_timeout
WOLA     control_region_wlm_dispatch_timeout
```

The current environment variables for HTTP dispatch timeouts. Granular down to server.

Other protocol dispatch timeouts

Granular RAS - XML Classification File

```
<Classification schema_version="1.0">
   <InboundClassification type="http" schema_version="1.0"
      default_transaction_class="TRANCL" >
      <http_classification_info
            uri="/SuperSnoopWeb/*" transaction_class="TRANCL"
            description="Snoop" dispatch_timeout="60" />
      <http_classification_info
            uri="/MyIVT/*" transaction_class="TRANCL"
            description="MyIVT" dispatch_timeout="15" />
   </InboundClassification>
</Classification>
```

XML Classification file section for HTTP. File supports `http`, `iiop`, `mdb`, `ola`, `sip`, `internal`

Requests matching this get 60 second timeout

Requests matching this get 15 second timeout

If timeout in XML and it applicable then it takes precedence over configured environment variable

InfoCenter **rtrb_controllingtimeout** Timeouts

InfoCenter **rrun_wlm_tclass_dtd** XML file

XML nesting ...

24 IBM Americas Advanced Technical Skills Gaithersburg, MD © 2013 IBM Corporation

The first example of the new granular control function we'll look at is the "Dispatch Timeout" value. Many of us are familiar with this timeout -- it's the timer for how long a request takes once it has been dispatched from the WLM work queue into the servant region.

At the top of the screen we see the environment variables that control this timeout value. These environment variables have been around for several releases now. They still work if you wish to use them. But note that the lowest level of granularity you can achieve with environment variables is scope=server.

Let's take the granularity even lower. A sample XML Classification File is shown on the chart. It's showing the familiar `type="http"` ... and we ask you remind yourself that other protocols are supported and this granular function can be used with them as well.

We see two `http_classification_info` sections ... one for the SuperSnoop application, one for the MyIVT application. Highlighted in yellow we see the new feature in action. The attribute `dispatch_timeout` is provided for each, but with a different value.

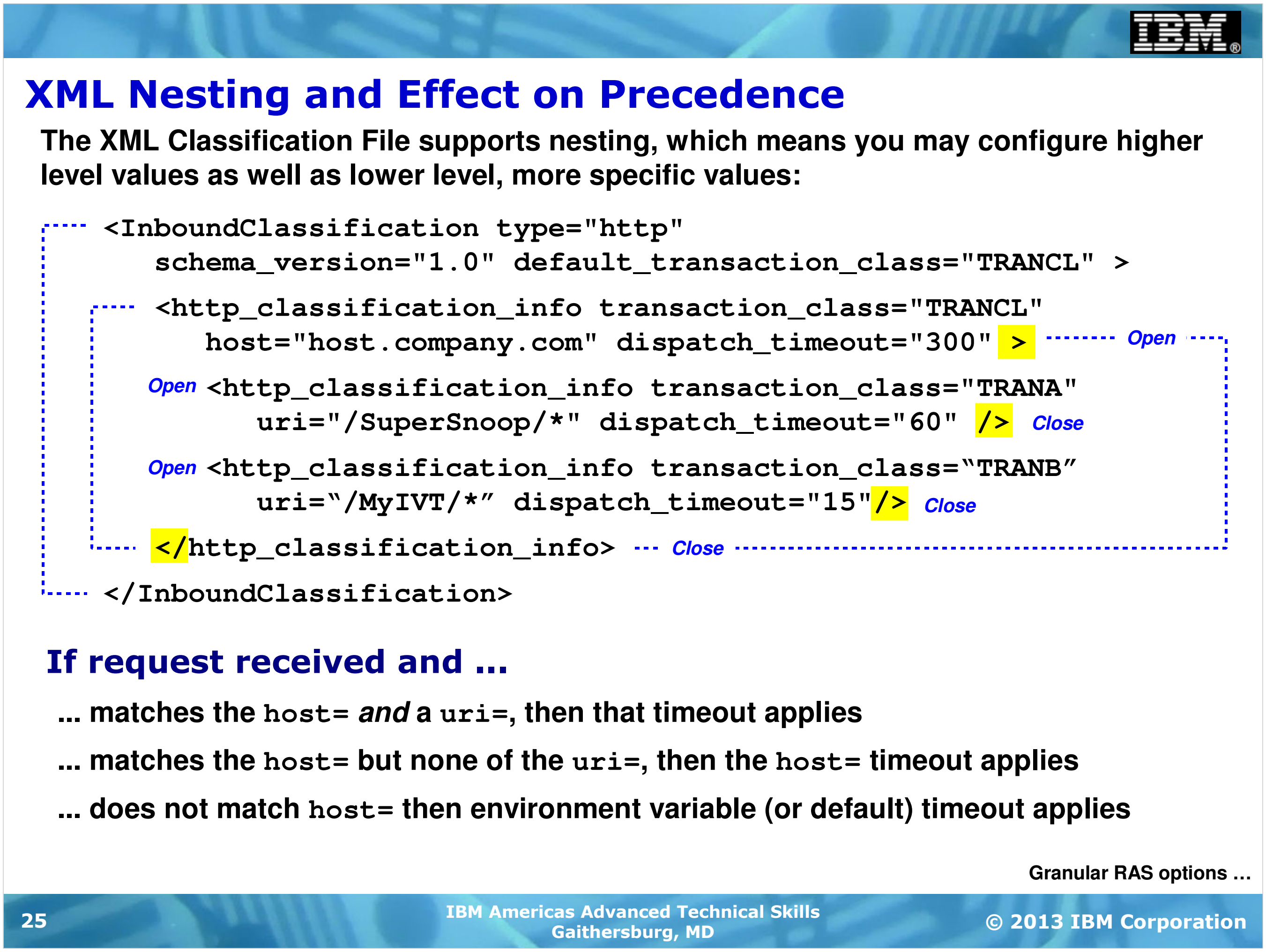
Step back and think about what's going on here:

- If a request matches the `uri=/SuperSnoopWeb/*` filter, then it will operate under a dispatch timeout of 60 seconds.
- If a request matches the `uri=/MyIVT/*` filter, then it will operate under a dispatch timeout of 15 seconds
- If a request doesn't match either then the environment variable applies, or its default value.

That's the essence of this new function -- illustrated with one of many attributes to control behavior down to a very granular level.

The chart has two InfoCenter articles noted -- one for a summary of all the timeout environment variables and one for the details on what can be coded in the XML file.

And don't forget Techdoc WP102023, written by David Follis of the IBM development organization. It provides excellent detail on this new function.



XML Nesting and Effect on Precedence

The XML Classification File supports nesting, which means you may configure higher level values as well as lower level, more specific values:

```
<InboundClassification type="http"
   schema_version="1.0" default_transaction_class="TRANCL" >
   <http_classification_info transaction_class="TRANCL"
      host="host.company.com" dispatch_timeout="300" >  Open
   Open <http_classification_info transaction_class="TRANA"
         uri="/SuperSnoop/*" dispatch_timeout="60" />  Close
   Open <http_classification_info transaction_class=“TRANB”
         uri=“/MyIVT/*” dispatch_timeout="15"/>  Close
   </http_classification_info>  Close
</InboundClassification>
```

If request received and ...

... matches the `host=` *and* a `uri=`, then that timeout applies

... matches the `host=` but none of the `uri=`, then the `host=` timeout applies

... does not match `host=` then environment variable (or default) timeout applies

Granular RAS options ...

Up to this point in our discussions on the XML Classification File we've kept the XML itself somewhat simple. In our previous examples we left out XML nesting. Nesting provides a way to provide higher level defaults and then provide more detailed filtering at individual requests.

Look at the example shown above. This is an example of the XML *like* what we've shown before, but with a difference -- this is nested. The blue dotted lines show the open / close relationships in the XML.

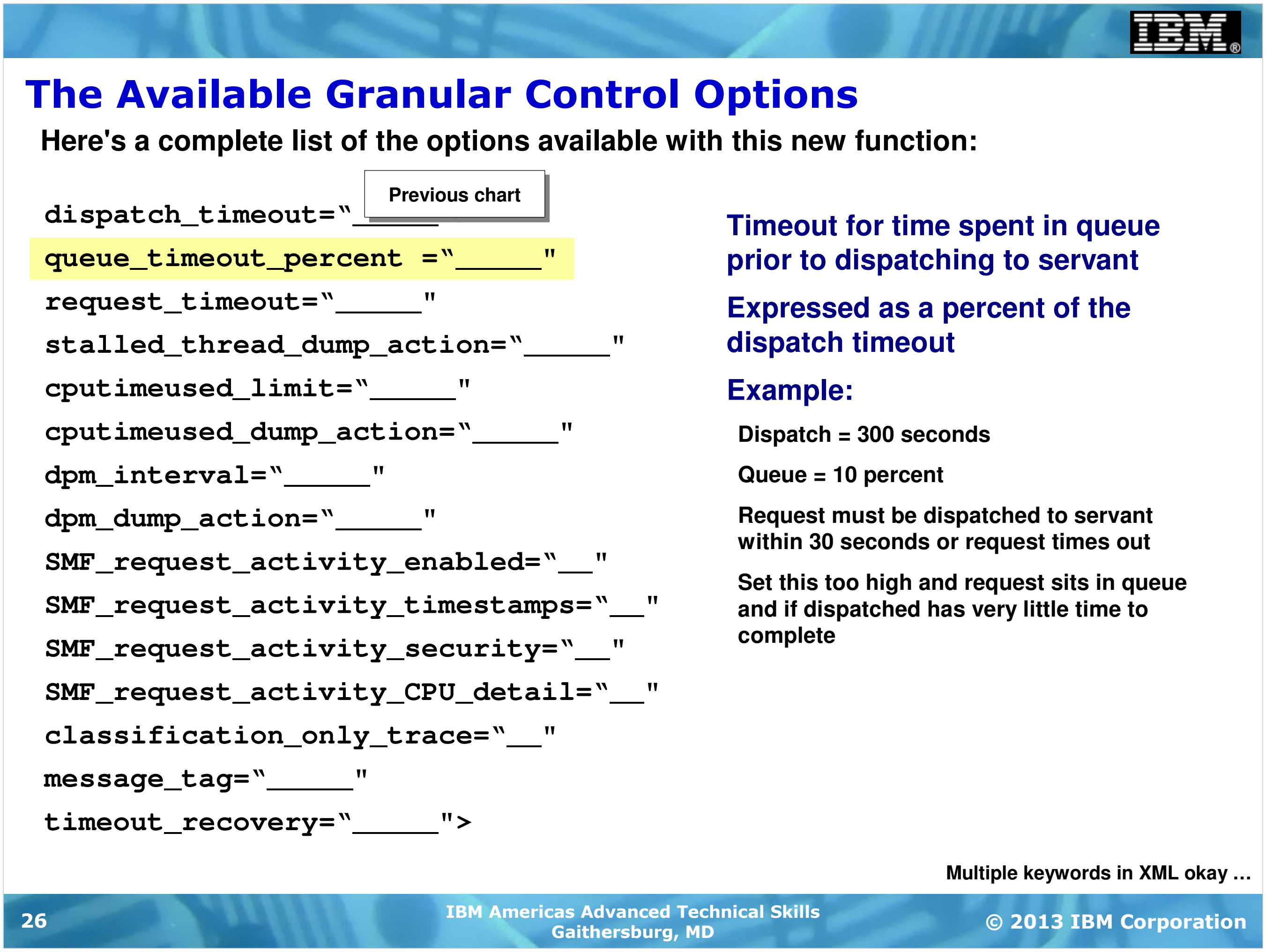
At the highest level we have the `InboundClassification` section that provides the `type="http"` tag. Again, other protocols are supported; we show HTTP because it's what people most commonly think of for WAS work. The InboundClassification section is closed at the bottom of the XML sample. A classification file might have multiple of these, one for each protocol the user wishes to code rules for.

Next is the first "node" in the XML tree for `http_classification_info`. This uses a match criteria of `host=` rather than `uri=` ... with a `dispatch_timeout` value of 300 seconds specified. That node is closed further down with the `</http_classification_info>` line. Within the open and close of that node there are two additional `http_classification_info` lines. Each has its own "open" and "close" tags. Those are lower, or "nested" nodes on the tree. Each has a `uri=" "` match criteria and a different `dispatch_timeout` value.

What this provides is a way to capture a broader set of requests with the outer XML node and give it a "default" `dispatch_timeout` value. But if a request comes in that matches the `host=` and matches one of the `uri=` values, then assign a different `dispatch_timeout` value.

But if something comes in with just the `host=` matching but not one of the uri= values, then the outer node `dispatch_timeout` applies.

If a request comes in that doesn't match anything ... well, then you fall back to the environment variables or their defaults.



The Available Granular Control Options

Here's a complete list of the options available with this new function:

Previous chart

```
dispatch_timeout=“______
queue_timeout_percent =“______"
request_timeout=“______"
stalled_thread_dump_action=“______"
cputimeused_limit=“______"
cputimeused_dump_action=“______"
dpm_interval=“______"
dpm_dump_action=“______"
SMF_request_activity_enabled=“__"
SMF_request_activity_timestamps=“__"
SMF_request_activity_security=“__"
SMF_request_activity_CPU_detail=“__"
classification_only_trace=“__"
message_tag=“______"
timeout_recovery=“______">
```

Timeout for time spent in queue prior to dispatching to servant

Expressed as a percent of the dispatch timeout

Example:

Dispatch = 300 seconds

Queue = 10 percent

Request must be dispatched to servant within 30 seconds or request times out

Set this too high and request sits in queue and if dispatched has very little time to complete

Multiple keywords in XML okay …

We've shown one new feature attribute value -- dispatch_timeout. There are more. This chart shows all the available attributes for this new function. What we'll now do is work through all the attributes and explain what each does. We start with `queue_timeout_percent`.

Note: many of these values have corresponding environment variable equivalents. We are not showing those equivalents here. The WP102023 Techdoc does a very nice job of calling those equivalents.

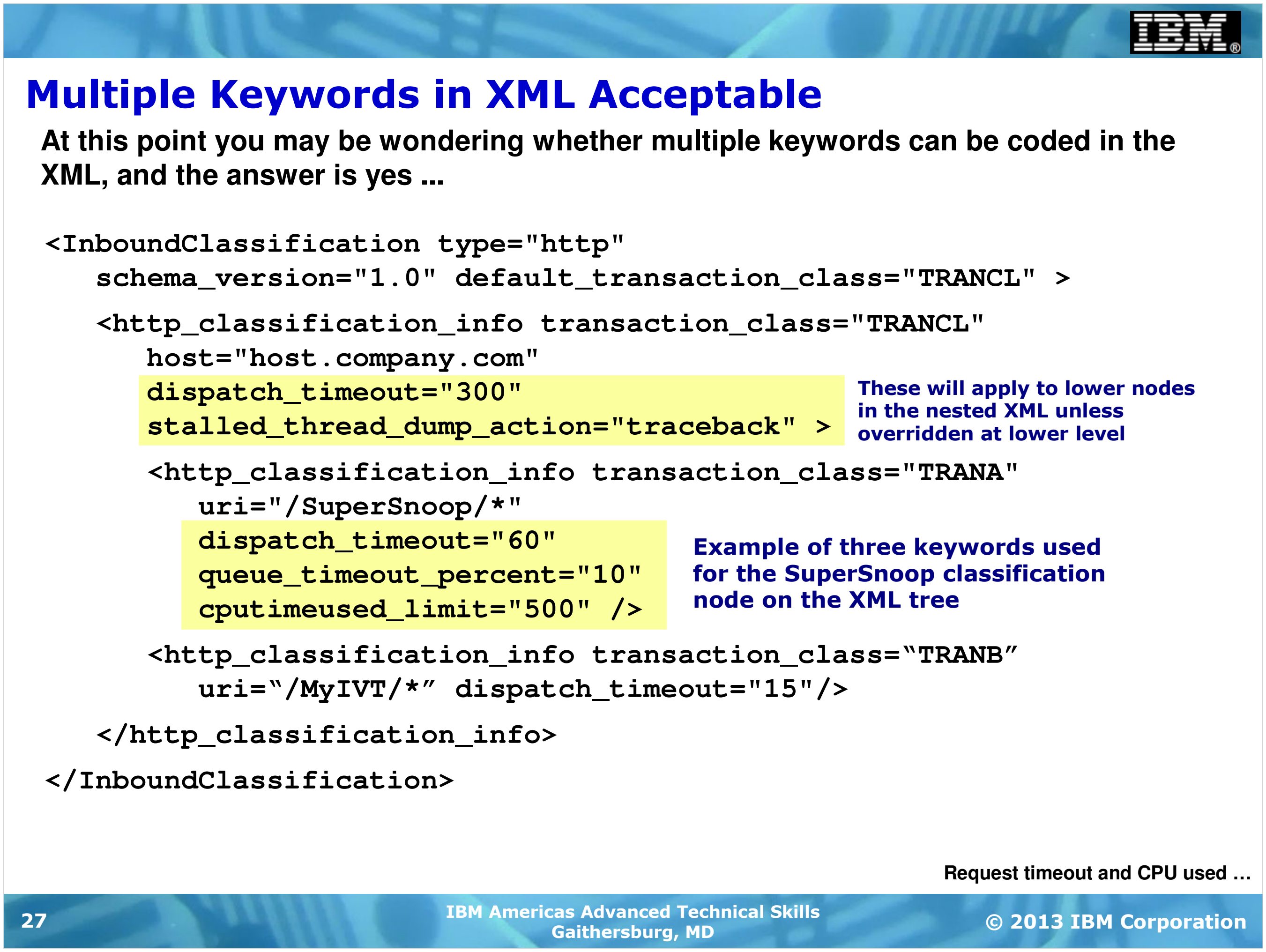
This value serves as the timer for how long a request sits in the WLM work queue before it gets dispatched to a servant worker thread. But rather than being expressed as a time value, it's expressed as a percent of the applicable dispatch timeout value (either specified in XML, in an environment variable or default). Why as a percent? Because in general the the time spent in the work queue is very short. The real interest is not that time so much as it is the overall time to complete the request. By expressing this as a percent of that overall time to complete it allows you to indicate your desired *proportion* of overall time the queue time may represent.

For example, imaging the dispatch timeout is a value of 300 seconds.

A `queue_timeout_percent` value of `10` means the work can't spend more than 30 seconds (300 x 10%) in the WLM work queue before timing out. Suppose it's delayed in the queue up to 29 seconds, but then gets dispatched *just* before the timeout. That means the work still has 90% of the overall timeout value to complete its work in the servant.

Now imagine a queue_timeout_percent of `80` and a overall timeout of 300 seconds. Now the work could sit in the work queue for 240 of the overall 300 seconds, leaving a very short period of time to get the work done in the servant if it is dispatched at the last moment.

By expressing the `queue_timeout_percent` as a percent of the overall dispatch timeout value, it allows you to specify the queue timeout in the context of what's really important -- the *proportion* of overall time the work sits in the first step of the process, which is the WLM work queue.



Multiple Keywords in XML Acceptable

At this point you may be wondering whether multiple keywords can be coded in the XML, and the answer is yes ...

```
<InboundClassification type="http"
   schema_version="1.0" default_transaction_class="TRANCL" >
   <http_classification_info transaction_class="TRANCL"
      host="host.company.com"
      dispatch_timeout="300"
      stalled_thread_dump_action="traceback" >
      <http_classification_info transaction_class="TRANA"
         uri="/SuperSnoop/*"
         dispatch_timeout="60"
         queue_timeout_percent="10"
         cputimeused_limit="500" />
      <http_classification_info transaction_class="“TRANB”
         uri="“/MyIVT/*” dispatch_timeout="15"/>
   </http_classification_info>
</InboundClassification>
```

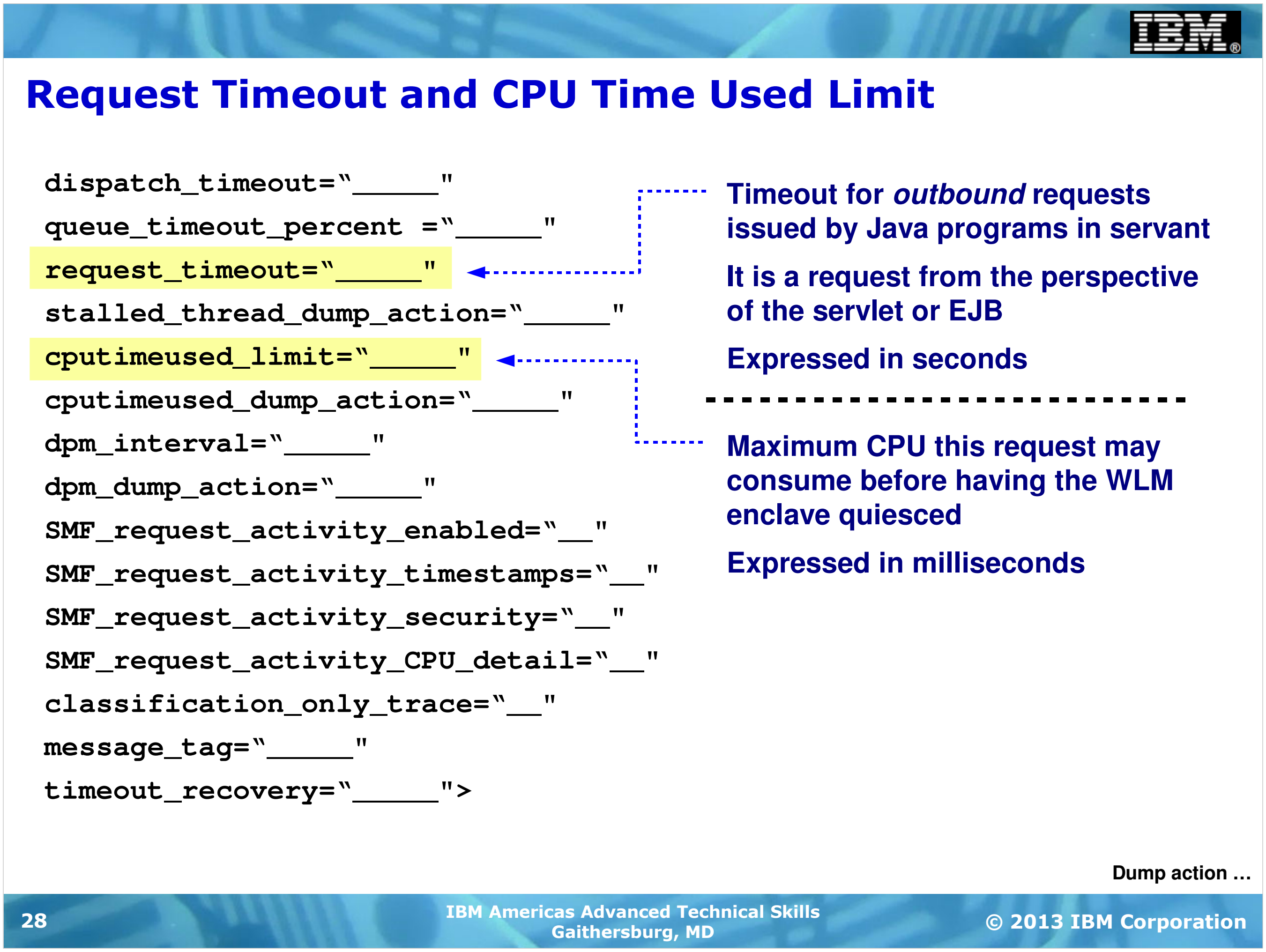
These will apply to lower nodes in the nested XML unless overridden at lower level

Example of three keywords used for the SuperSnoop classification node on the XML tree

Request timeout and CPU used …

We just showed a chart with 15 attributes that provide this new granular control function. If only one was permitted per classification node in the XML the new feature wouldn't be very useful. Multiple values are permitted as illustrated.

When the XML is nested you may have some values specified on the higher (or "outer") nodes to serve as a kind of default for the lower (or "inner") nodes. And as shown, the higher-level values may be overridden by specified values at the lower levels.



Request Timeout and CPU Time Used Limit

```
dispatch_timeout="_____"
queue_timeout_percent ="_____"
request_timeout="_____"
stalled_thread_dump_action="_____"
cputimeused_limit="_____"
cputimeused_dump_action="_____"
dpm_interval="_____"
dpm_dump_action="_____"
SMF_request_activity_enabled="__"
SMF_request_activity_timestamps="__"
SMF_request_activity_security="__"
SMF_request_activity_CPU_detail="__"
classification_only_trace="__"
message_tag="_____"
timeout_recovery="_____">
```

Timeout for *outbound* requests issued by Java programs in servant

It is a request from the perspective of the servlet or EJB

Expressed in seconds

Maximum CPU this request may consume before having the WLM enclave quiesced

Expressed in milliseconds

Dump action …

28

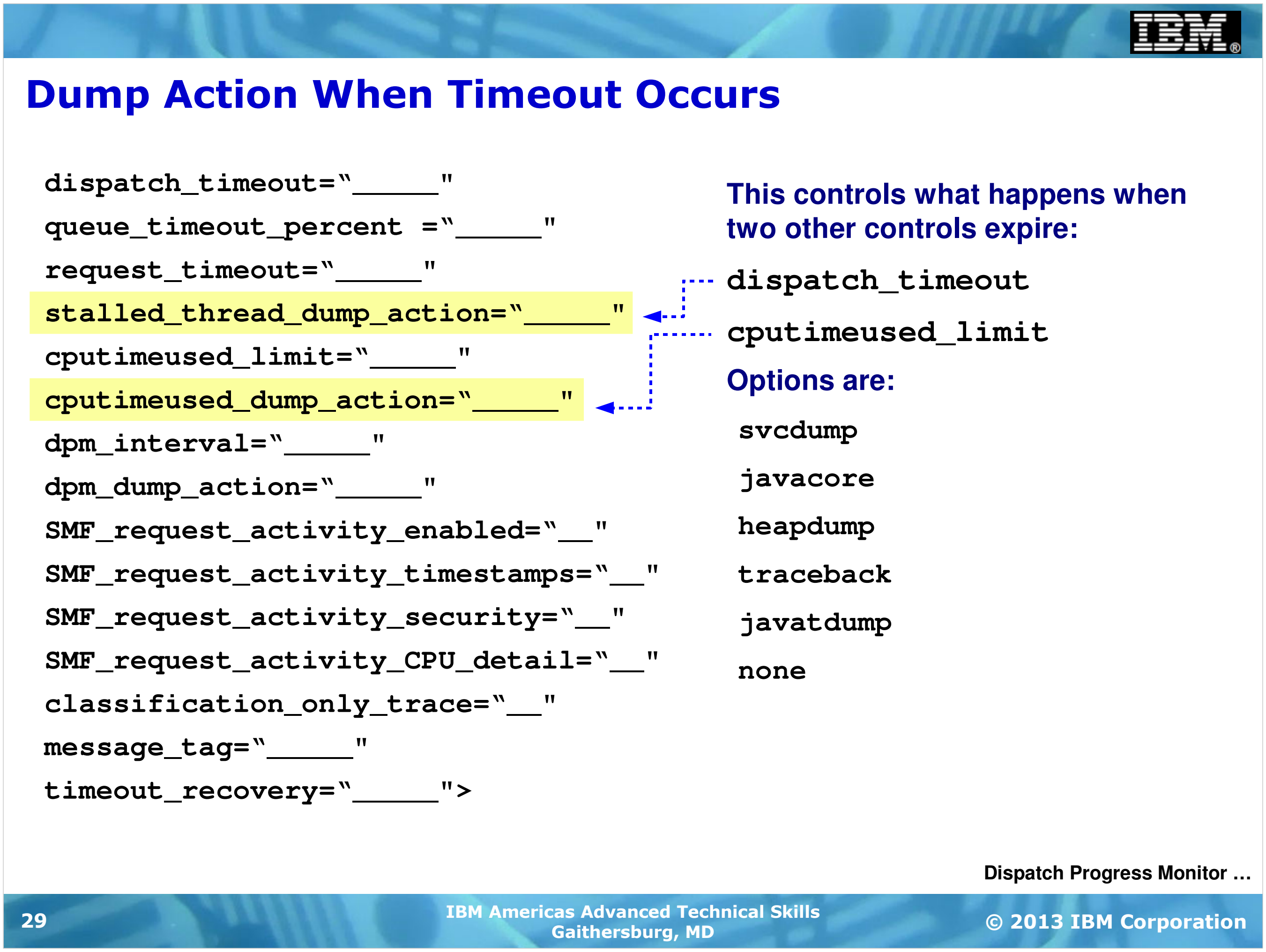
IBM Americas Advanced Technical Skills
Gaithersburg, MD

© 2013 IBM Corporation

On this chart we'll discuss two attributes simply to save some charts. These two are not necessarily related functionally. It's just it doesn't take much to explain them and two per chart easily fit the white space available. ☺

`request_timeout` -- this attribute will at first seem a little odd because it has nothing to do with the request from the client into WAS. The "request" portion of that keyword refers to *a request made by a Java component in the servant*. Java programs (servlets, EJBs) will often issue a request to other Java objects, either in the same server or another part of the cell or even somewhere well outside the cell. This timeout applies to requests such as those. The timeout is expressed in seconds, much like the other timeout values.

`cputimeused_limit` -- this attribute specifies the number of *milliseconds* of CPU time (not wall clock time) a thread running a matching request may consume. If the thread consumes more CPU time than specified here, then the WLM enclave for that execution thread is *quiesced* by WLM. That has the effect of moving the priority of that work to *below discretionary*. The thread is not killed ... it's just that WLM will not give that thread any access to the CPU unless nothing else on the system is seeking resources. On a development or test system that may be true, in which case the quiesced thread will continue. But on a production system it's very unlikely nothing else would be seeking resources, so in effect that thread (more precisely, the enclave) will cease to consume resources. This is a way to protect against runaway processes.



Dump Action When Timeout Occurs

```
dispatch_timeout=“_____"
queue_timeout_percent =“_____"
request_timeout=“_____"
stalled_thread_dump_action=“_____"
cputimeused_limit=“_____"
cputimeused_dump_action=“_____"
dpm_interval=“_____"
dpm_dump_action=“_____"
SMF_request_activity_enabled=“__"
SMF_request_activity_timestamps=“__"
SMF_request_activity_security=“__"
SMF_request_activity_CPU_detail=“__"
classification_only_trace=“__"
message_tag=“_____"
timeout_recovery=“_____">
```

This controls what happens when two other controls expire:

`dispatch_timeout`

`cputimeused_limit`

Options are:

```
svcdump
javacore
heapdump
traceback
javatdump
none
```

Dispatch Progress Monitor …

29

IBM Americas Advanced Technical Skills
Gaithersburg, MD

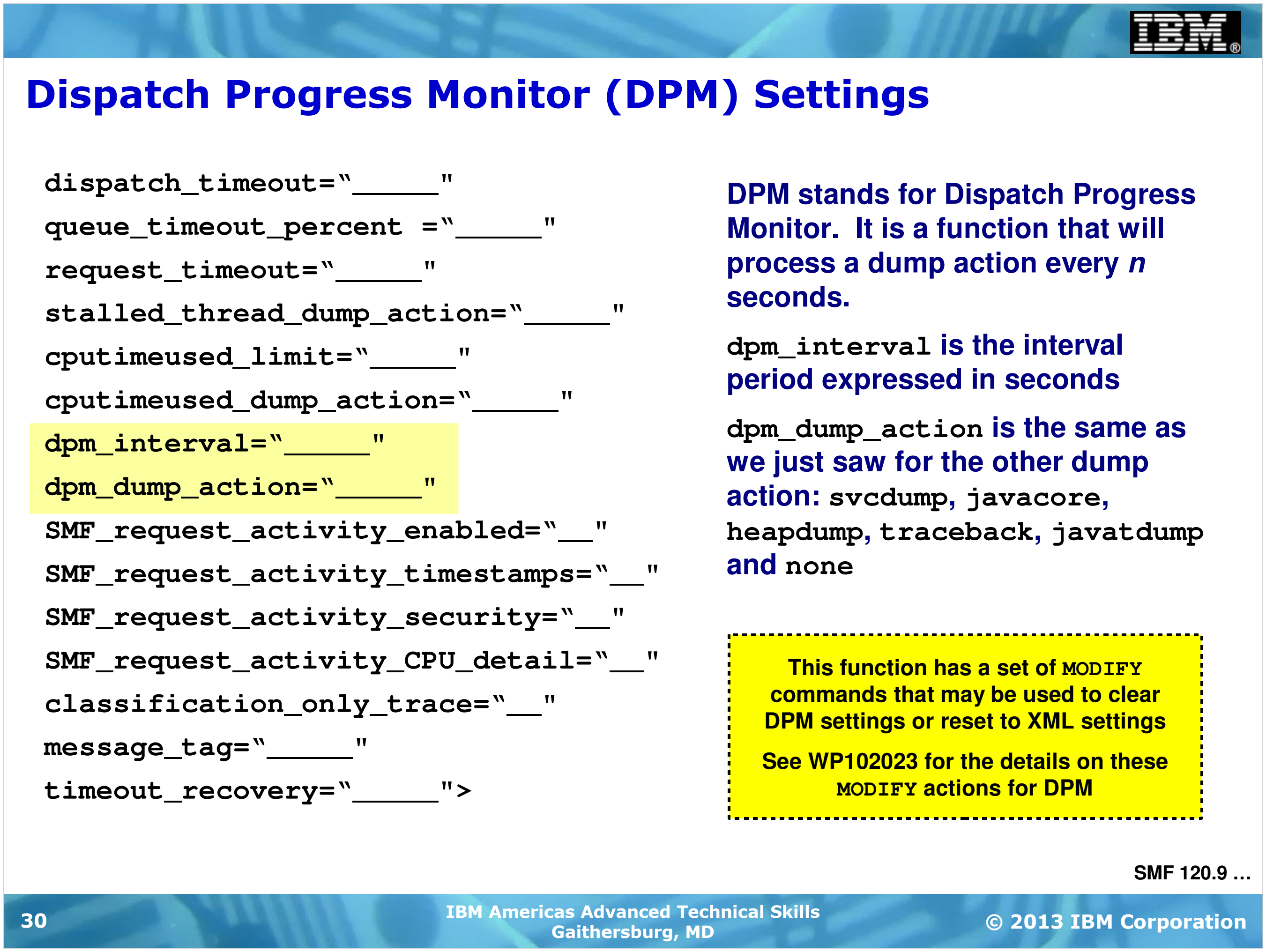
Another chart where we combine two attributes on one chart. In this case there is a certain affinity between the two. These two attributes define what kind of action WAS z/OS should take when another defined attribute has reached its limit:

`stalled_thread_dump_action` ➔ when `dispatch_timeout` (or equivalent env. variable) has expired

`cputimeused_dump_action` ➔ when `cputimeused_limit` (or equivalent env. variable) has been met

The options you may specify on these two attributes are the same, and they're shown on the chart.

What this provides you is your choice of debug options when the event has taken place. And remember, these values override values that may be set at the higher server level. So it's possible to have one dump action set at the server (`traceback`, for example), but for a specific request you can have a `javacore` produced.



Dispatch Progress Monitor (DPM) Settings

```
dispatch_timeout=“_____"
queue_timeout_percent =“_____"
request_timeout=“_____"
stalled_thread_dump_action=“_____"
cputimeused_limit=“_____"
cputimeused_dump_action=“_____"
dpm_interval=“_____"
dpm_dump_action=“_____"
SMF_request_activity_enabled=“__"
SMF_request_activity_timestamps=“__"
SMF_request_activity_security=“__"
SMF_request_activity_CPU_detail=“__"
classification_only_trace=“__"
message_tag=“_____"
timeout_recovery=“_____">
```

DPM stands for Dispatch Progress Monitor. It is a function that will process a dump action every *n* seconds.

`dpm_interval` **is the interval period expressed in seconds**

`dpm_dump_action` **is the same as we just saw for the other dump action:** `svcdump`, `javacore`, `heapdump`, `traceback`, `javatdump` **and** `none`

This function has a set of `MODIFY` commands that may be used to clear DPM settings or reset to XML settings

See WP102023 for the details on these `MODIFY` actions for DPM

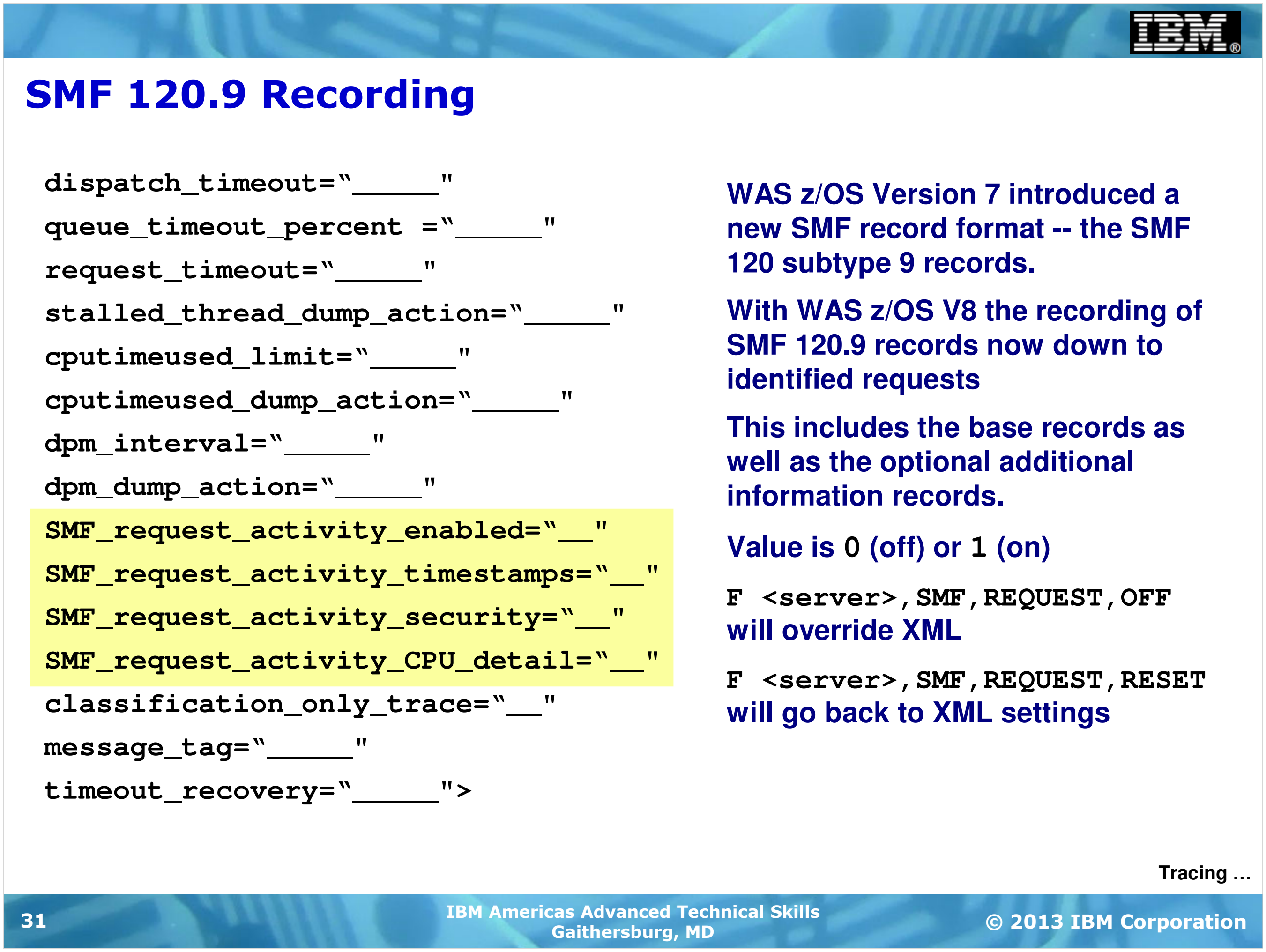
SMF 120.9 …

30 IBM Americas Advanced Technical Skills Gaithersburg, MD © 2013 IBM Corporation

The "Dispatch Progress Monitor" (DPM) is a function that will produce a dump action when a dispatched thread exceeds the time specified for the interval value, and then produce the dump action every interval period until the dispatched thread completes. It's a function meant to help debug cases where infrequent and unpredictable delays crop up in processing.

Two attributes are shown here -- one for the DPM interval and one for the dump action to be taken when that interval has expired. The interval attribute is expressed in seconds, and the dump action is the same we saw on the previous chart.

The DPM function has a set of MVS MODIFY commands to dynamically impose DPM intervals and dump actions, as well as reset functions. So there's a relationship between the settings in the XML and whatever MODIFY commands might have been issued. The matrix of possible variations on these two variables gets large and complex, and rather than covering that here we'll refer you to the WP102023 Techdoc for a fairly complete explanation of what applies when various combinations of XML and MODIFY exist.



SMF 120.9 Recording

```
dispatch_timeout=“_____"
queue_timeout_percent =“_____"
request_timeout=“_____"
stalled_thread_dump_action=“_____"
cputimeused_limit=“_____"
cputimeused_dump_action=“_____"
dpm_interval=“_____"
dpm_dump_action=“_____"
SMF_request_activity_enabled=“__"
SMF_request_activity_timestamps=“__"
SMF_request_activity_security=“__"
SMF_request_activity_CPU_detail=“__"
classification_only_trace=“__"
message_tag=“_____"
timeout_recovery=“_____">
```

WAS z/OS Version 7 introduced a new SMF record format -- the SMF 120 subtype 9 records.

With WAS z/OS V8 the recording of SMF 120.9 records now down to identified requests

This includes the base records as well as the optional additional information records.

Value is 0 (off) or 1 (on)

`F <server>,SMF,REQUEST,OFF` will override XML

`F <server>,SMF,REQUEST,RESET` will go back to XML settings

Tracing ...

WAS z/OS Version 7 introduced the SMF 120.9 records. In V7 we had the opportunity to specify the recording of SMF 120.9 with environment variables as well as MODIFY commands. The SMF 120.9 records have two "levels" of detail -- the base records, and a set of option extended detail records.

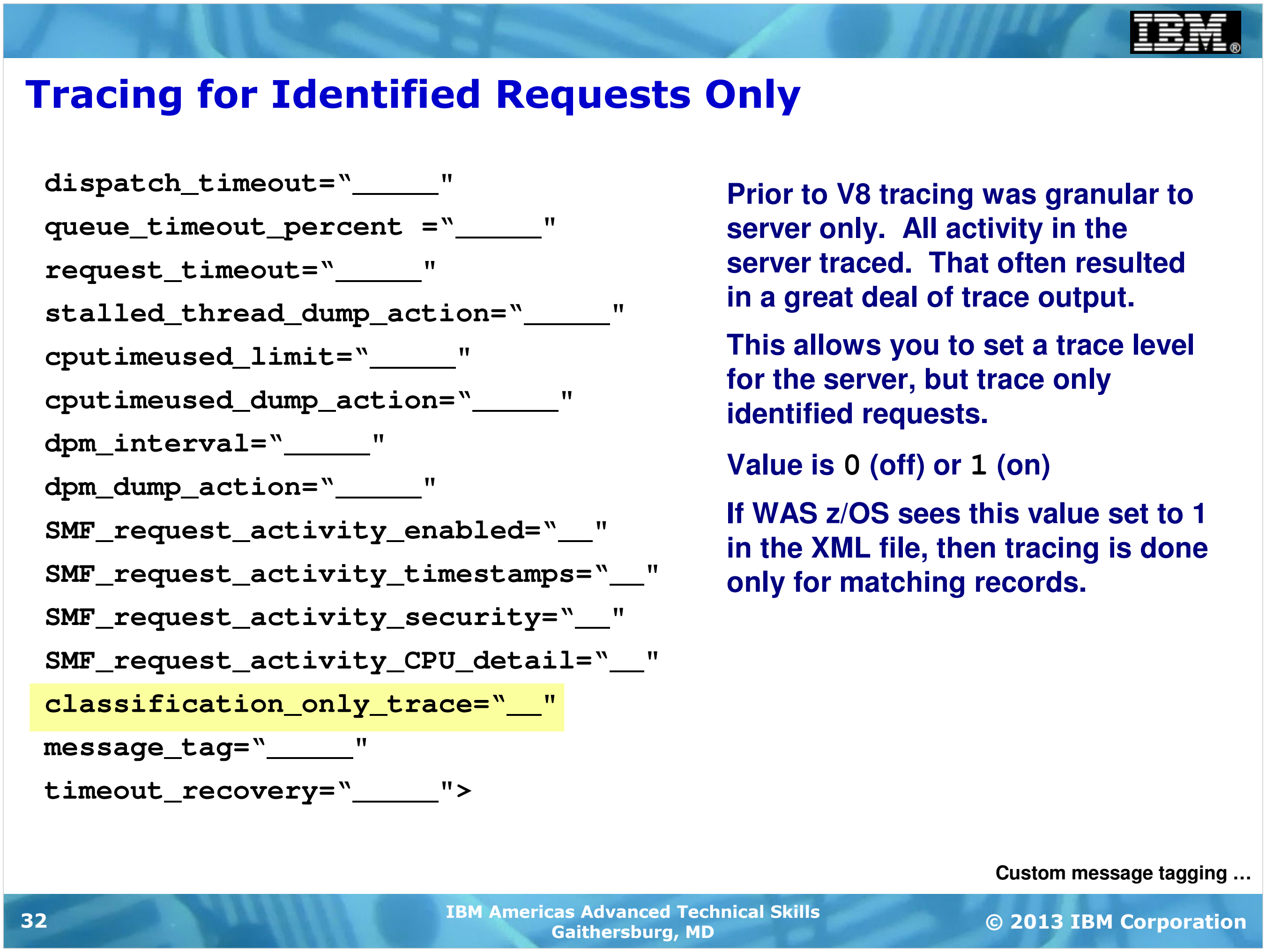
The attribute `SMF_request_activity_enabled` is what turns on or off the base records at the request level. The other three shown are the extended optional records.

What Version 8 has done is extend this SMF recording to the request level. Now you can have SMF 120.9 records -- either the base or the extended -- written for only those requests identified in the XML file.

The intersection with the MODIFY command for SMF 120.9 is shown on the chart:

- If you have the SMF attributes specified in the XML (and the XML is in use by a server), the MODIFY to turn SMF recording off will override the XML. That means requests matching elements of the XML will not result in the the writing of SMF 120.9.
- However, you may use the MODIFY RESET command to go back to whatever you have coded in the XML

Note: at the end of this unit we're going to explain how you can dynamically reload an updated copy of the XML. So changes you wish to make to the XML for SMF or any other attribute can be made and reloaded on the fly, providing new or modified granular behavior.



Tracing for Identified Requests Only

```
dispatch_timeout=“_____"
queue_timeout_percent =“_____"
request_timeout=“_____"
stalled_thread_dump_action=“_____"
cputimeused_limit=“_____"
cputimeused_dump_action=“_____"
dpm_interval=“_____"
dpm_dump_action=“_____"
SMF_request_activity_enabled=“__"
SMF_request_activity_timestamps=“__"
SMF_request_activity_security=“__"
SMF_request_activity_CPU_detail=“__"
classification_only_trace=“__"
message_tag=“_____"
timeout_recovery=“_____">
```

Prior to V8 tracing was granular to server only. All activity in the server traced. That often resulted in a great deal of trace output.

This allows you to set a trace level for the server, but trace only identified requests.

Value is 0 (off) or 1 (on)

If WAS z/OS sees this value set to 1 in the XML file, then tracing is done only for matching records.

Custom message tagging …

32

IBM Americas Advanced Technical Skills
Gaithersburg, MD

The next attribute we'll discuss is `classification_only_trace`, which drives tracing down to the individual request level.

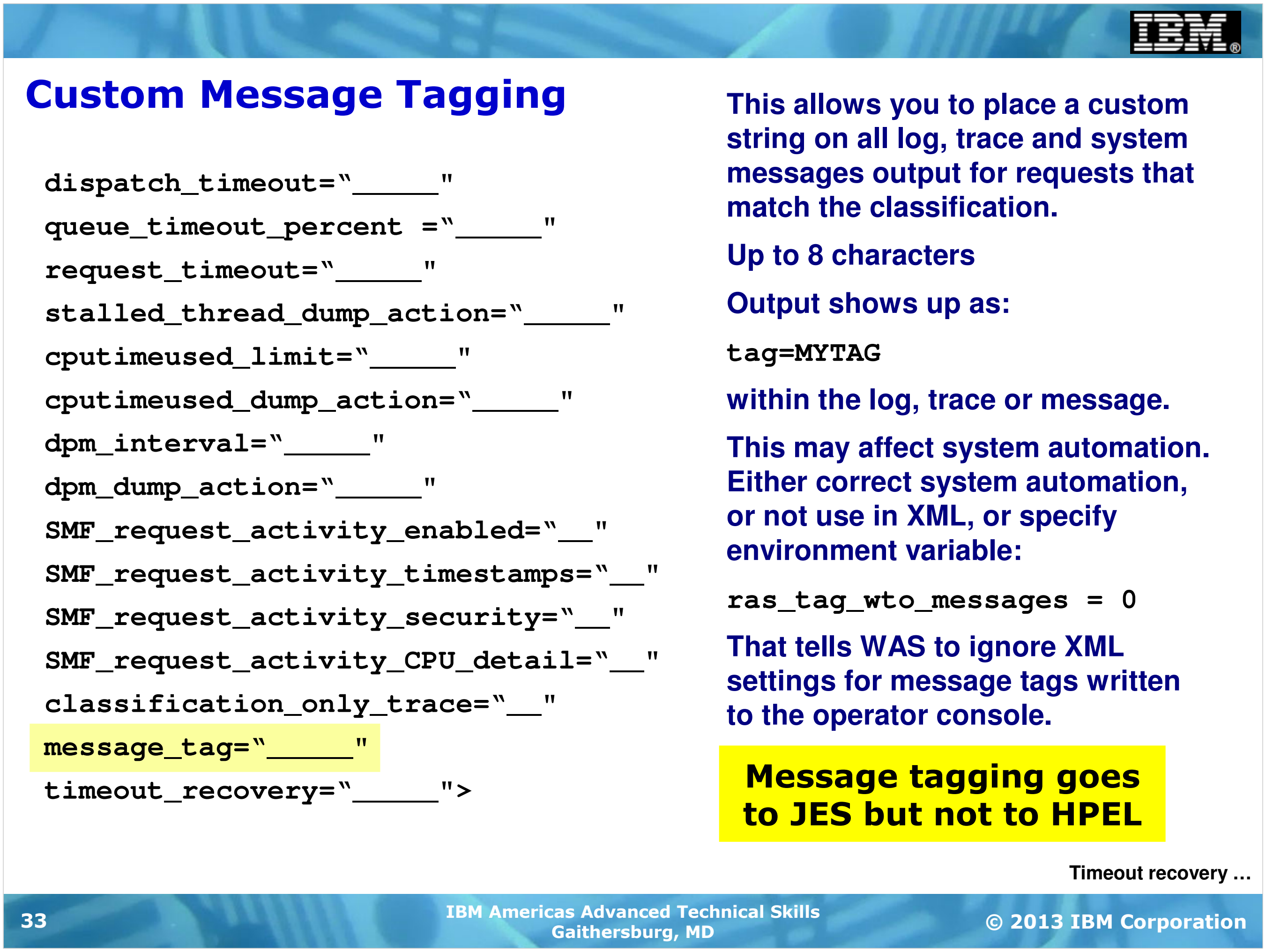
Anyone who has used WAS tracing before knows that it can produce a lot of output. That's particularly true for some of the more detailed trace settings. In the past the best we could do was set the trace specifications for the server, and those trace settings would apply to all activity in the server. Then the chore was to wade through all the output looking for the needle in the haystack, so to speak.

With this new granular control we can specify which requests you want tracing for. Only those requests get traced.

Notice how the attribute is really just a switch -- 0 or 1. The attribute doesn't say anything about the trace settings to use. Those are set as they've always been set -- through the Admin Console runtime settings, using environment variables or using the WAS z/OS MODIFY command.

If WAS z/OS sees a valid XML file and sees in that XML this `classification_only_trace=1` setting, then it readies the trace settings set elsewhere but *does not start tracing* until some request matches the classification criteria in the XML. Then *only that matching request gets traced.*

This gives you the opportunity to narrow the tracing that's done to only those requests you really want tracing for. All other requests are not traced.



Custom Message Tagging

```
dispatch_timeout=“_____"
queue_timeout_percent =“_____"
request_timeout=“_____"
stalled_thread_dump_action=“_____"
cputimeused_limit=“_____"
cputimeused_dump_action=“_____"
dpm_interval=“_____"
dpm_dump_action=“_____"
SMF_request_activity_enabled=“__"
SMF_request_activity_timestamps=“__"
SMF_request_activity_security=“__"
SMF_request_activity_CPU_detail=“__"
classification_only_trace=“__"
message_tag=“_____"
timeout_recovery=“_____">
```

This allows you to place a custom string on all log, trace and system messages output for requests that match the classification.

Up to 8 characters

Output shows up as:

`tag=MYTAG`

within the log, trace or message.

This may affect system automation. Either correct system automation, or not use in XML, or specify environment variable:

`ras_tag_wto_messages = 0`

That tells WAS to ignore XML settings for message tags written to the operator console.

Message tagging goes to JES but not to HPEL

Timeout recovery …

33

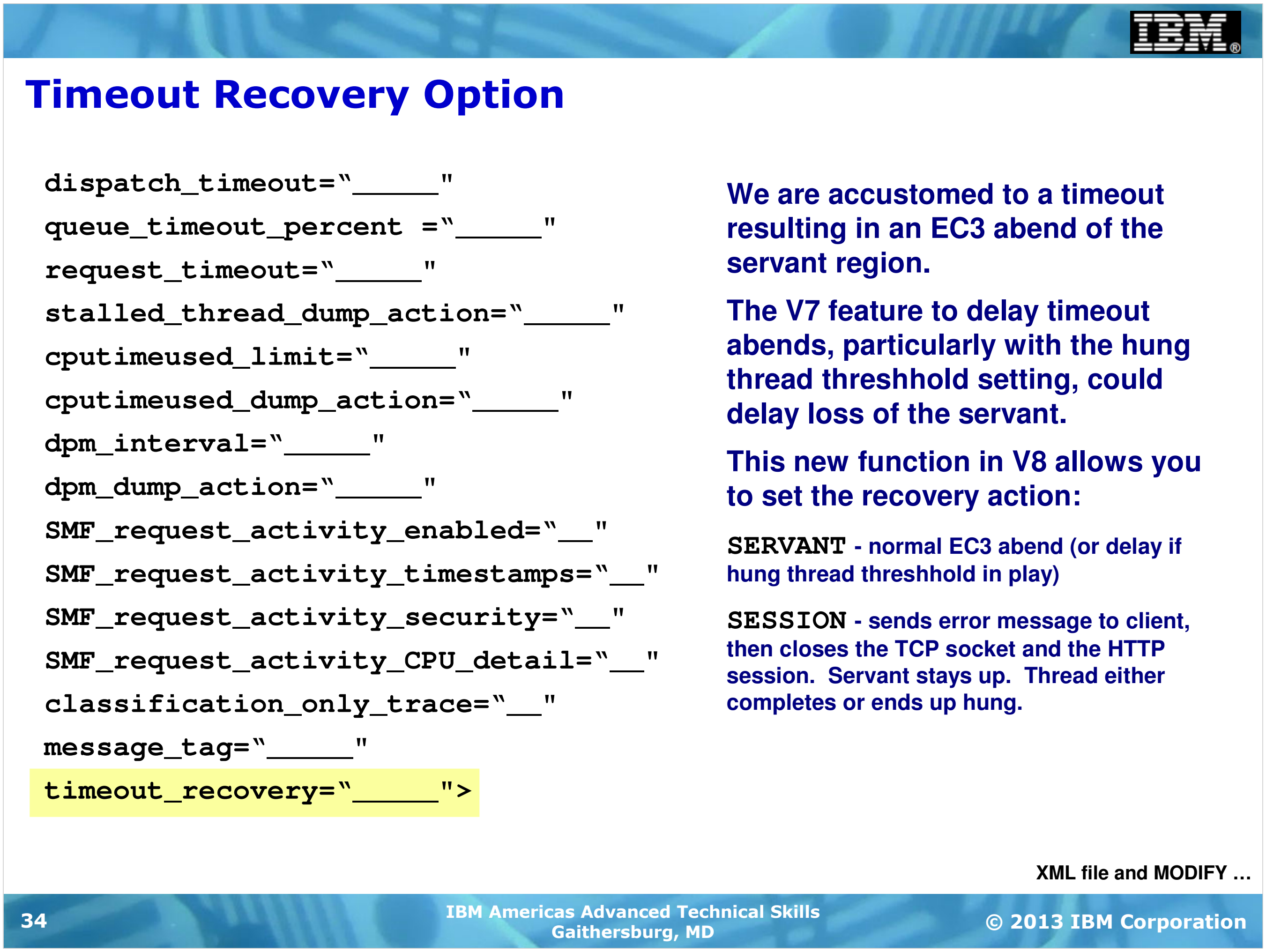
IBM Americas Advanced Technical Skills
Gaithersburg, MD

© 2013 IBM Corporation

The `message_tag` attribute provides a way for you to append to all log, trace and system messages related to an identified request a custom string of your choosing. This provides a way to quickly seek and find messages related to specific requests, and to filter out records related to a given request. The custom tag may be up to eight characters in length.

Note: This message tagging feature only shows up in JES. If you enable HPEL then the messages get written to HPEL without the tag. That is because the tag is applied at a fairly low-level of WAS code, and HPEL captures messages and writes them to HPEL well above that message-tagging code. If the message is written to HPEL then it never gets to the point where the message tag is applied. If you have HPEL enabled some z/OS native messages with the tag get to JES, but all the Java-related messages go to HPEL without the tag.

We realize that adding a custom message tag to system messages *may* affect system automation routines specifically coded to find keywords at certain positions within output. If that happens, you may either correct the system automation routines, remove the `message_tag attribute`, or you may specify a new environment variable called `ras_tag_wto_messages` which can be used to suppress the custom message tag on WTO messages. Set that variable to 0 (for "no") and the custom tag will be suppressed to the operator console, but will still appear in logs and traces. The default for this new variable is 1 (for "yes") so you need to explicitely code this new variable if you wish to suppress custom message tagging to WTO messages.



Timeout Recovery Option

```
dispatch_timeout=“______"
queue_timeout_percent =“______"
request_timeout=“______"
stalled_thread_dump_action=“______"
cputimeused_limit=“______"
cputimeused_dump_action=“______"
dpm_interval=“______"
dpm_dump_action=“______"
SMF_request_activity_enabled=“__"
SMF_request_activity_timestamps=“__"
SMF_request_activity_security=“__"
SMF_request_activity_CPU_detail=“__"
classification_only_trace=“__"
message_tag=“______"
timeout_recovery=“______">
```

We are accustomed to a timeout resulting in an EC3 abend of the servant region.

The V7 feature to delay timeout abends, particularly with the hung thread threshhold setting, could delay loss of the servant.

This new function in V8 allows you to set the recovery action:

`SERVANT` - normal EC3 abend (or delay if hung thread threshhold in play)

`SESSION` - sends error message to client, then closes the TCP socket and the HTTP session. Servant stays up. Thread either completes or ends up hung.

XML file and MODIFY …

34

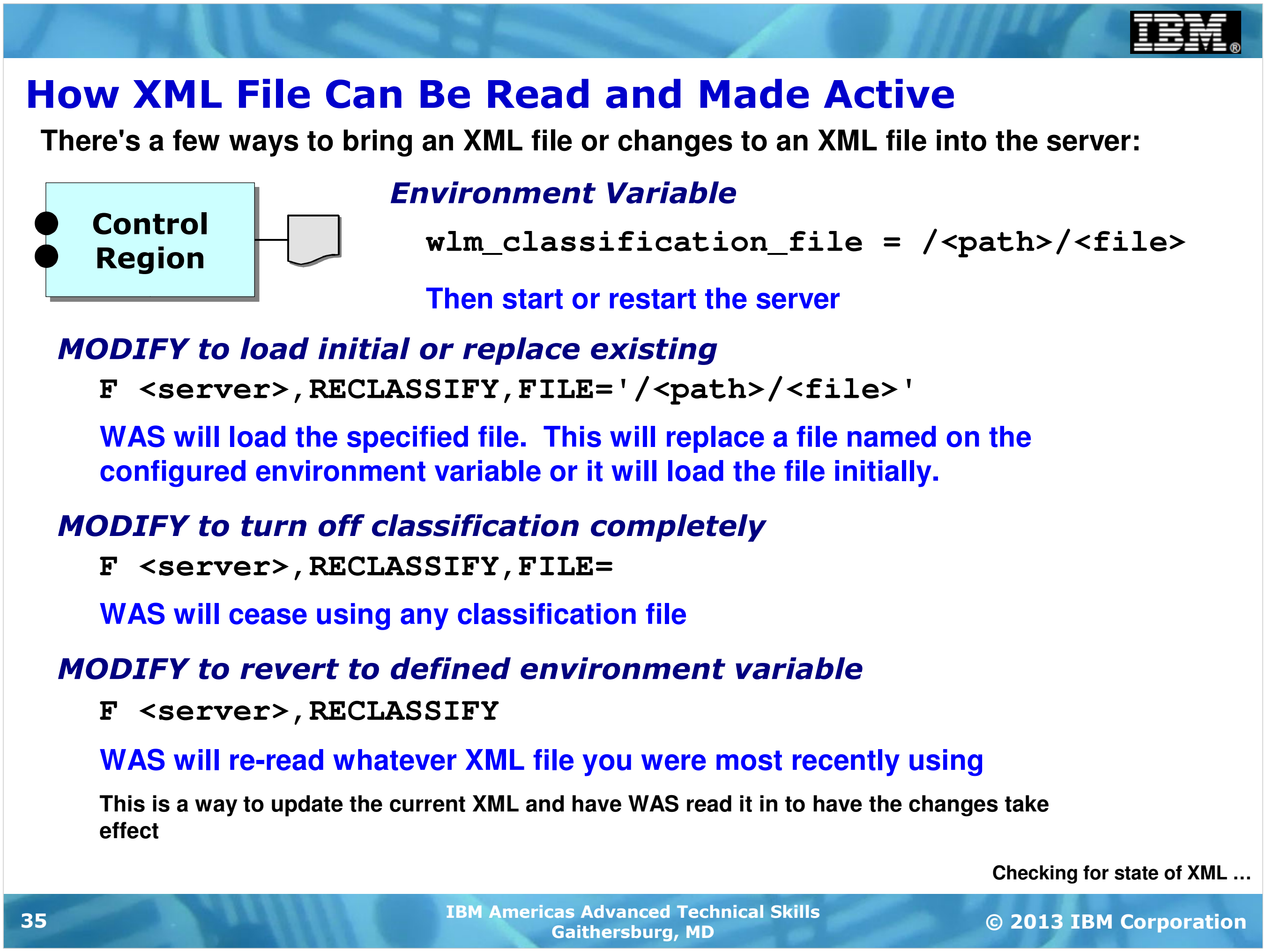
IBM Americas Advanced Technical Skills
Gaithersburg, MD

© 2013 IBM Corporation

The final attribute is `timeout_recovery`. This provides the ability to specify what happens when a timeout occurs. We're accustomed to the dreaded EC3 abend of the servant region. In WAS z/OS V7 we provided the ability to delay the EC3 abend by specifying a threshhold percent of hung threads below which the EC3 abend would be deferred. With this attribute we can set two options -- `SERVANT` and `SESSION`.

`SERVANT` does what we're accustomed to -- when a timeout occurs the servant region is abended with EC3. This abend action may be delayed if the threshhold function added in V7 is in place.

`SESSION` takes a different approach. This tells WAS z/OS to send an error message back to the client and close the TCP socket and HTTP Session. The servant stays up. No EC3 abend. The thread on which the timeout occurred either continues and eventually completes, or it ends up truly hung. But the point is the EC3 does not occur.



How XML File Can Be Read and Made Active

There's a few ways to bring an XML file or changes to an XML file into the server:

Control Region

Environment Variable

```
wlm_classification_file = /<path>/<file>
```

Then start or restart the server

MODIFY to load initial or replace existing

```
F <server>,RECLASSIFY,FILE='/<path>/<file>'
```

WAS will load the specified file. This will replace a file named on the configured environment variable or it will load the file initially.

MODIFY to turn off classification completely

```
F <server>,RECLASSIFY,FILE=
```

WAS will cease using any classification file

MODIFY to revert to defined environment variable

```
F <server>,RECLASSIFY
```

WAS will re-read whatever XML file you were most recently using

This is a way to update the current XML and have WAS read it in to have the changes take effect

Checking for state of XML ...

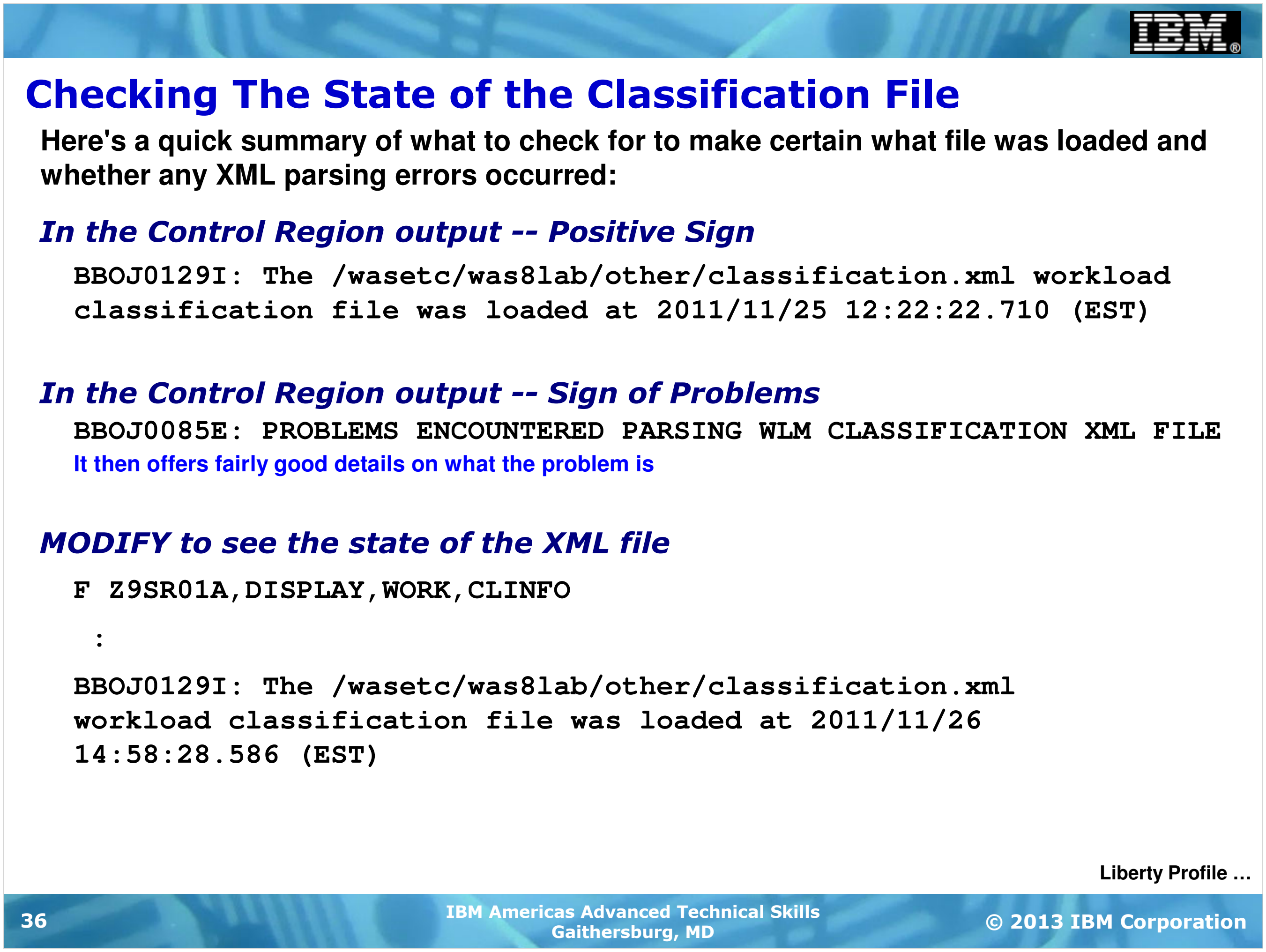
There are three ways you can get an XML Classification File to be read into the server's control region (that's where classification occurs -- the CR and not the SR ... classification is one of those "IBM plumbing" activities we said takes place in the CR).

One way is to code the environment `variable wlm_classification_file` and point to the path and file name for the file. Then restart the server to pick up the file.

Note: on the next chart we'll show you the messages you'll see for success or failure for the read of the XML file.

Another way is to use the MODIFY command to dynamically read in the file specified on the MODIFY command:

- If you specify a file on the MODIFY RECLASSIFY command, it will read in the new file
- If you specify just RECLASSIFY (but no file specified) then it re-reads whatever classification file it was most recently using. This is a way to *refresh* the in-memory copy of the classification file by re-reading the file again from the file system.
- If you specify RECLASSIFY with FILE= (but no file name) then classification turns off ... WAS discards all in-memory copies of the classification file and reads nothing from disk.



Checking The State of the Classification File

Here's a quick summary of what to check for to make certain what file was loaded and whether any XML parsing errors occurred:

In the Control Region output -- Positive Sign

```
BBOJ0129I: The /wasetc/was8lab/other/classification.xml workload
classification file was loaded at 2011/11/25 12:22:22.710 (EST)
```

In the Control Region output -- Sign of Problems

```
BBOJ0085E: PROBLEMS ENCOUNTERED PARSING WLM CLASSIFICATION XML FILE
```

It then offers fairly good details on what the problem is

MODIFY to see the state of the XML file

```
F Z9SR01A,DISPLAY,WORK,CLINFO

  :

BBOJ0129I: The /wasetc/was8lab/other/classification.xml
workload classification file was loaded at 2011/11/26
14:58:28.586 (EST)
```

Liberty Profile ...

Naturally you'd like to know if the XML you specify is read in properly, or if there are problems. There are messages to look for in the control region. They are shown on the chart above. The explanations offered for the case where there's a problem are pretty good ... if it's a parsing problem the description guides you to where in the file the XML parsing error occurred.

Another way is to issue the MODIFY command shown on the chart. That will tell you which XML file is active at that moment in time. This is handy if you've been using the other MODIFY commands shown on the previous chart to dynamically bring in new files and you're not sure what file is actually loaded. This will tell you.

Some common errors reading the file:

- The file's READ permission bits don't allow the CR access to the file.
- The file is stored in USS as EBCDIC, when it needs to be in ASCII.
- There's an XML parsing problem ... some missing close bracket, or some missing attribute. Again, the error explanation offered in the BBOJ0085E message is pretty helpful in debugging this.

The Liberty Profile

Single JVM, composable, dynamic

WebSphere Application Server for z/OS Version 8.5

Liberty Profile

Quick Start Guide

Version Date: July 4, 2012

See "Document Change History" on page 35 for a description
of the changes in this version of the document

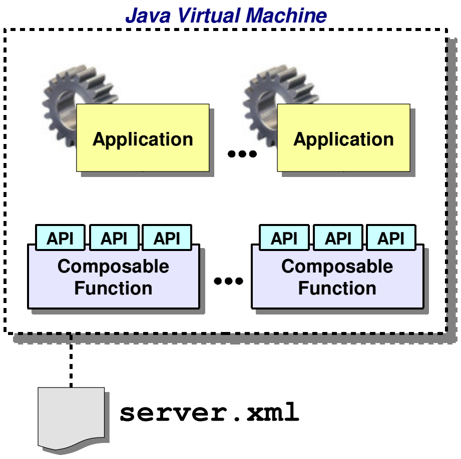
IBM Advanced Technical Skills
Gaithersburg, MD

WP102110 at

`ibm.com/support/techdocs`

Overview of the Liberty Profile

The Liberty Profile is designed to be a single-JVM server model that is lightweight, composable and dynamic:



- **Composable -- you configure the function the application needs; you don't need to load up everything**
- **Dynamic -- changes to configuration or changes to applications detected and dynamically enabled**
- **Subset of traditional WAS function**
  Liberty is not full Java EE, traditional WAS is
- **Upwards application compatibility**
  Apps that run in Liberty will run in traditional WAS ... but not necessarily the other way around since Liberty is subset of traditional WAS
- **Each server is one JVM**
- **Run from UNIX shell or as started task**
- **One required configuration file: `server.xml`**
- ***Not* part of traditional WAS administrative DMGR, federated node model**
  But there is an ability to manage via the "Job Manager" function of traditional WAS (advanced topic, we won't get into that here)

Composable "features" ...

When approaching the "Liberty Profile" for the first time, you must try to keep it separate from the traditional WAS z/OS server model with the CR, SR and all the things we just finished discussing. The Liberty Profile server model is packaged and delivered with WAS z/OS Version 8.5, but in many ways it is quite different from traditional WAS z/OS.

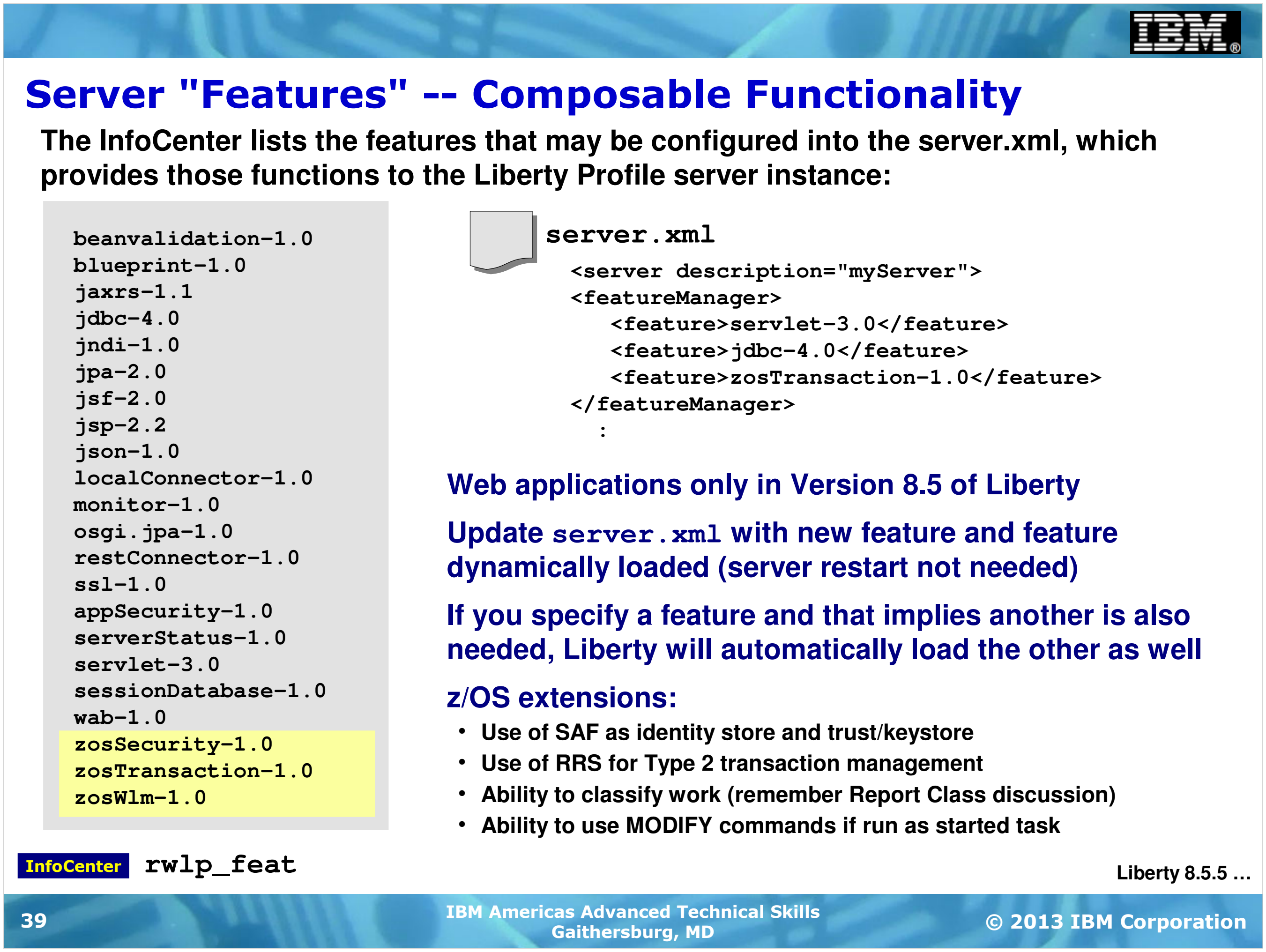
The goal of the Liberty Profile model is to provide a lightweight and flexible server runtime. The lightweight goal is achieved by making the functions loaded by Liberty "composable" -- that is, through the configuration you indicate what functions your application needs and only those functions (and functions Liberty sees are co-requisite with your specified functions) consume resources. The Liberty Profile is also dynamic in that changes to the configuration and changes to the applications may be dynamically detected and updated. (That feature can be controlled so the polling cycle is reduced or turned off completely.)

The Liberty Profile is a *functional subset* of the traditional WAS runtime model. The traditional WAS model is a full Java EE runtime; the Liberty Profile is not. That said, it is important to understand that applications developed and tested with Liberty will run in traditional WAS. (The reverse is not necessarily true ... it's possible to develop an application for the full Java EE WAS runtime that would not work in the functional subset Liberty server.)

Each Liberty Profile server instance is a single JVM, not multiple JVMs like what we saw for traditional WAS z/OS. The server instance may be started and operated from the UNIX shell or as a z/OS started task. Which of those two approaches you select is really a matter of your preference. Running the servers as a z/OS started task provides access to the MODIFY command, but otherwise the function available is the same for both methods.

The configuration is very simple -- one primary XML file ("server.xml") is all it requires. (There are other *optional* configuration files for JVM properties and UNIX environment entries.) Later in this unit we'll discuss what that server.xml file looks like.

The Liberty Profile server instances are not part of the "Admin Console" administration model you may be accustomed to with traditional WAS. Liberty Profile server instances are not part of the traditional WAS z/OS Network Deployment structure. Each server instance is separate from the others but, as we'll see, there is the ability to share configuration and applicaton elements between Liberty servers to aid in scaling up and out.



Server "Features" -- Composable Functionality

The InfoCenter lists the features that may be configured into the server.xml, which provides those functions to the Liberty Profile server instance:

```
beanvalidation-1.0
blueprint-1.0
jaxrs-1.1
jdbc-4.0
jndi-1.0
jpa-2.0
jsf-2.0
jsp-2.2
json-1.0
localConnector-1.0
monitor-1.0
osgi.jpa-1.0
restConnector-1.0
ssl-1.0
appSecurity-1.0
serverStatus-1.0
servlet-3.0
sessionDatabase-1.0
wab-1.0
zosSecurity-1.0
zosTransaction-1.0
zosWlm-1.0
```

server.xml

```
<server description="myServer">
<featureManager>
   <feature>servlet-3.0</feature>
   <feature>jdbc-4.0</feature>
   <feature>zosTransaction-1.0</feature>
</featureManager>
  :
```

Web applications only in Version 8.5 of Liberty

Update `server.xml` with new feature and feature dynamically loaded (server restart not needed)

If you specify a feature and that implies another is also needed, Liberty will automatically load the other as well

z/OS extensions:

- **Use of SAF as identity store and trust/keystore**
- **Use of RRS for Type 2 transaction management**
- **Ability to classify work (remember Report Class discussion)**
- **Ability to use MODIFY commands if run as started task**

InfoCenter `rwlp_feat`

Liberty 8.5.5 …

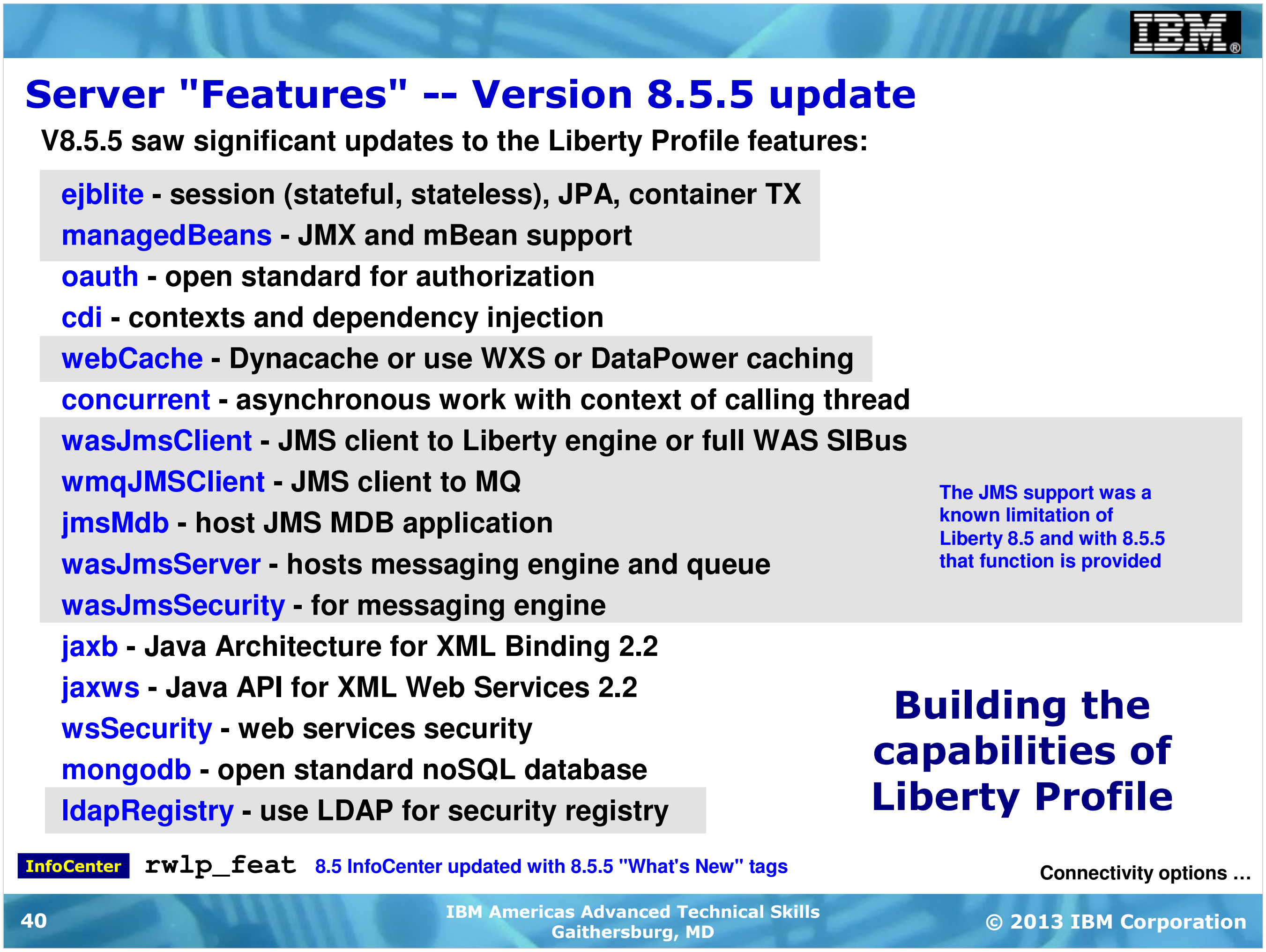
39 IBM Americas Advanced Technical Skills
Gaithersburg, MD
© 2013 IBM Corporation

On the previous chart's notes we mentioned the "composable" nature of the Liberty Profile server model. In the server.xml file you indicate which features you wish the Liberty Profile server instance to support and load. The chart is showing two things -- all the features supported with the initial release of Liberty (left side of chart) and an example of three of those features being specified in a snippet of server.xml.

Liberty is smart enough to understand pre-requisite and co-requisite functions. So, for example, if you specify "jsp-2.2" Liberty knows it must also load "servlet-3.0" since JSPs become servlets when compiled.

By default Liberty will dynamically monitor the server.xml file for changes, and will update the running server with any changes it detects. Those changes may be additions or deletions to the function.

Liberty Profile for z/OS has a set of extensions designed to take advantage of the z/OS platform. Those extensions are noted on the chart. We'll cover these in more detail later in the unit.



Server "Features" -- Version 8.5.5 update

V8.5.5 saw significant updates to the Liberty Profile features:

- ejblite - session (stateful, stateless), JPA, container TX
- managedBeans - JMX and mBean support
- oauth - open standard for authorization
- cdi - contexts and dependency injection
- webCache - Dynacache or use WXS or DataPower caching
- concurrent - asynchronous work with context of calling thread
- wasJmsClient - JMS client to Liberty engine or full WAS SIBus
- wmqJMSClient - JMS client to MQ
- jmsMdb - host JMS MDB application
- wasJmsServer - hosts messaging engine and queue
- wasJmsSecurity - for messaging engine
- jaxb - Java Architecture for XML Binding 2.2
- jaxws - Java API for XML Web Services 2.2
- wsSecurity - web services security
- mongodb - open standard noSQL database
- ldapRegistry - use LDAP for security registry

The JMS support was a known limitation of Liberty 8.5 and with 8.5.5 that function is provided

Building the capabilities of Liberty Profile

InfoCenter `rwlp_feat` 8.5 InfoCenter updated with 8.5.5 "What's New" tags

Connectivity options ...

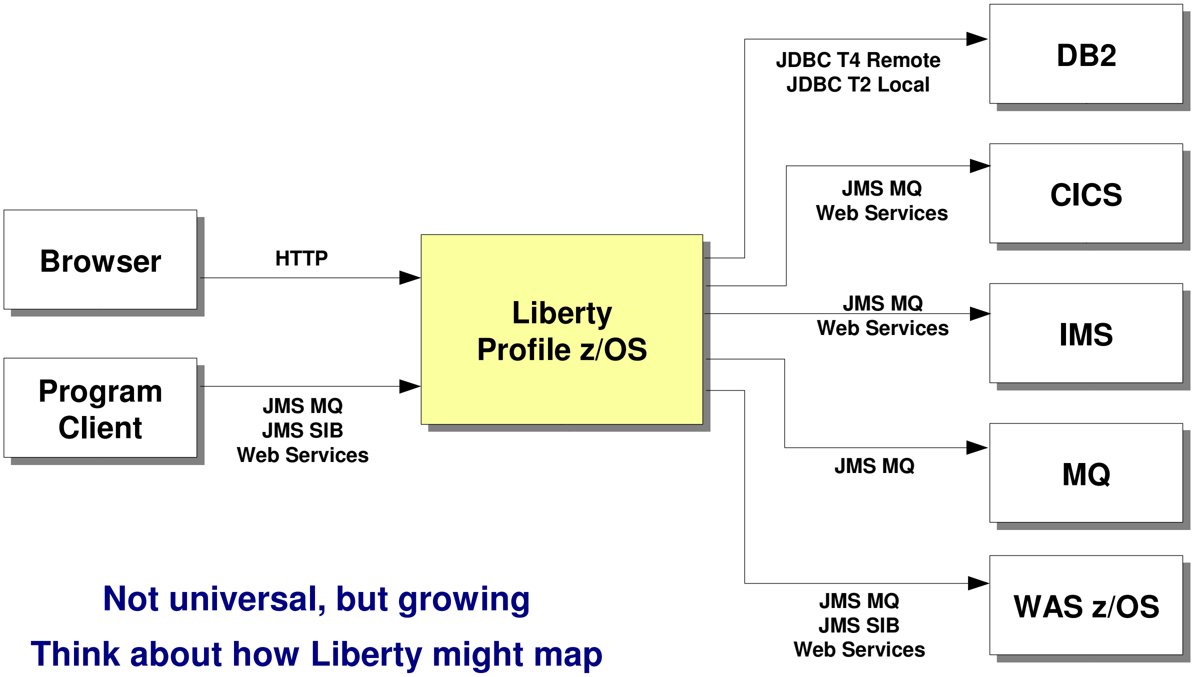
40 IBM Americas Advanced Technical Skills
Gaithersburg, MD
© 2013 IBM Corporation

When Liberty Profile came out with the V8.5 release, the list of functions available with Liberty was known to be missing a few things. The plan was in place to augment Liberty Profile with those functions over time, and with V8.5.5 we see some of these enhancements.

The chart above shows the features added to Liberty Profile V8.5 on z/OS. As you can see, an implementation EJB "lite" comes in, as well as JMS support.

Connectivity Options with 8.5.5

A summary chart of connectivity options with Liberty z/OS:



Not universal, but growing

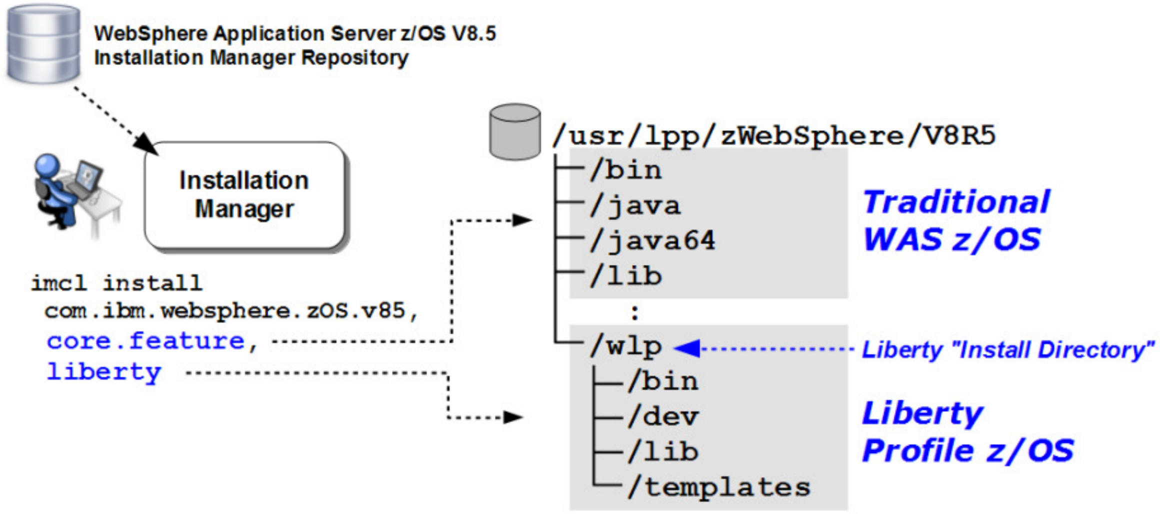
Think about how Liberty might map into the lower end of the architecture

Liberty in the file system …

This chart provides a summary of the connectivity options to commonly used IBM data resources. It is intended to give a sense for how you might use Liberty Profile on z/OS. The chart also shows how clients may access a Liberty Profile server. The inclusion of JMS support in 8.5.5 expanded the connectivity options from what was available in 8.5.0.

What 8.5.0 Liberty Looks Like in File System

When you specify option `liberty` to IM it will install the following directory and file structure into the target location:



WebSphere Application Server z/OS V8.5
Installation Manager Repository

Installation Manager

```
imcl install
 com.ibm.websphere.zOS.v85,
 core.feature,
 liberty
```

```
/usr/lpp/zWebSphere/V8R5
 ├/bin
 ├/java
 ├/java64
 ├/lib
 :
 └/wlp
   ├/bin
   ├/dev
   ├/lib
   └/templates
```

Traditional WAS z/OS

Liberty "Install Directory"

Liberty Profile z/OS

Those files are relatively small ... around 60MB. They represent the product files of Liberty. You will likely have this as a read-only file system. So where do the configuration files go? In a "user directory" ...

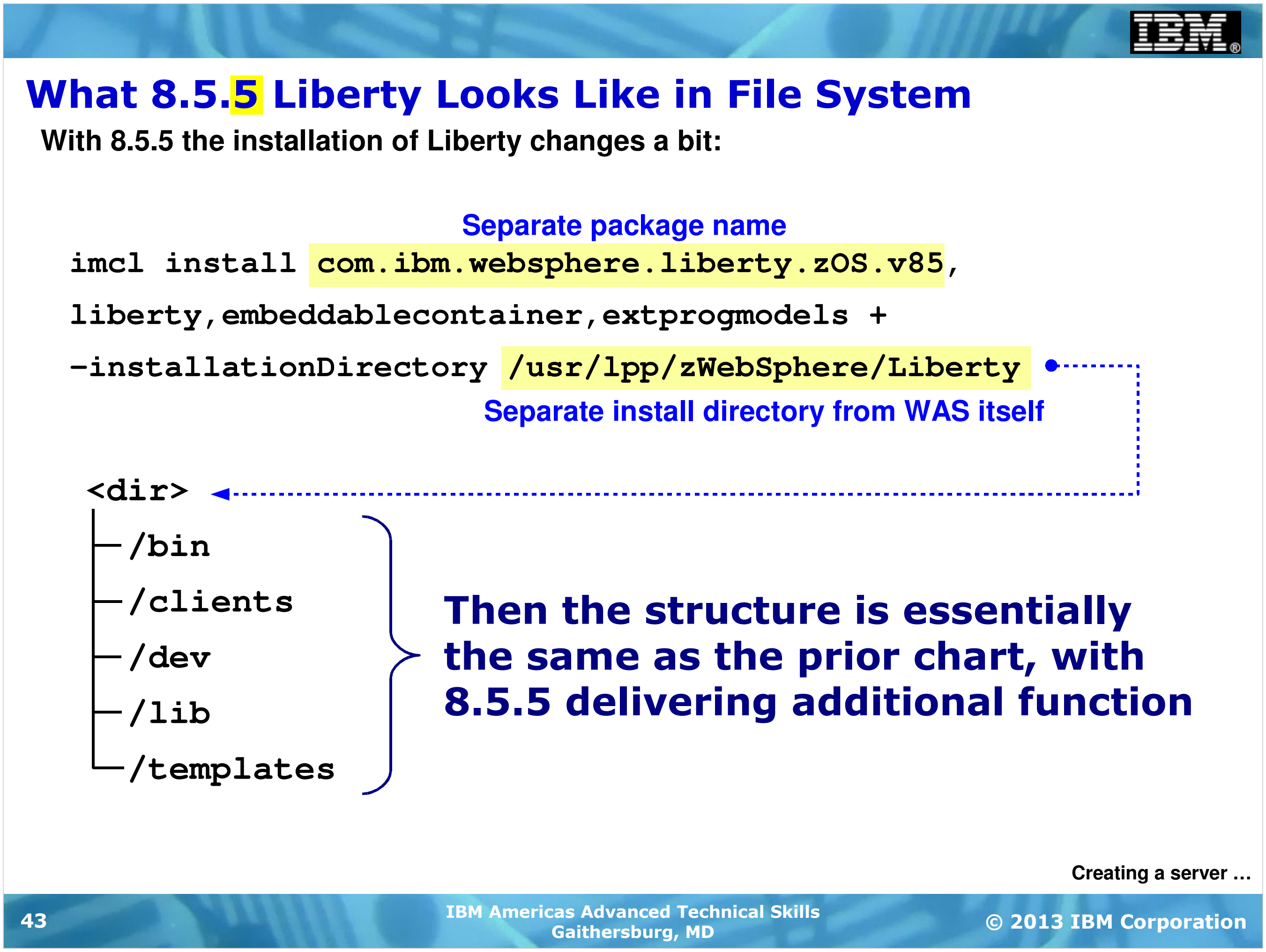
V8.5.5 ...

Note: This page shows what things look like for WAS z/OS V8.5. With 8.5.5 the installation for Liberty changes, and we show that on the next chart.

To help you understand the physical structure of Liberty, the chart above shows you what the installation files look like when you install WAS V8.5 using Installation Manager (IM) on z/OS. The key points being made by the chart are:

- You must specify "liberty" as part of the IM installation parameters to get Liberty installed into the file system
- It will install under the `/wlp` directory within whatever target mount point you specify for WAS V8.5 itself

The files that install are the runtime files, not the configuration files. Those are created (typically) somewhere other than the read-only file system for the WAS V8.5 installation. As you'll see in the next few charts, where that may be is really up to you ... the configuration files may be located anywhere.



What 8.5.5 Liberty Looks Like in File System

With 8.5.5 the installation of Liberty changes a bit:

Separate package name

```
imcl install com.ibm.websphere.liberty.zOS.v85,
liberty,embeddablecontainer,extprogmodels +
-installationDirectory /usr/lpp/zWebSphere/Liberty
```

Separate install directory from WAS itself

```
<dir>
 ├─/bin
 ├─/clients
 ├─/dev
 ├─/lib
 └─/templates
```

Then the structure is essentially the same as the prior chart, with 8.5.5 delivering additional function

Creating a server …

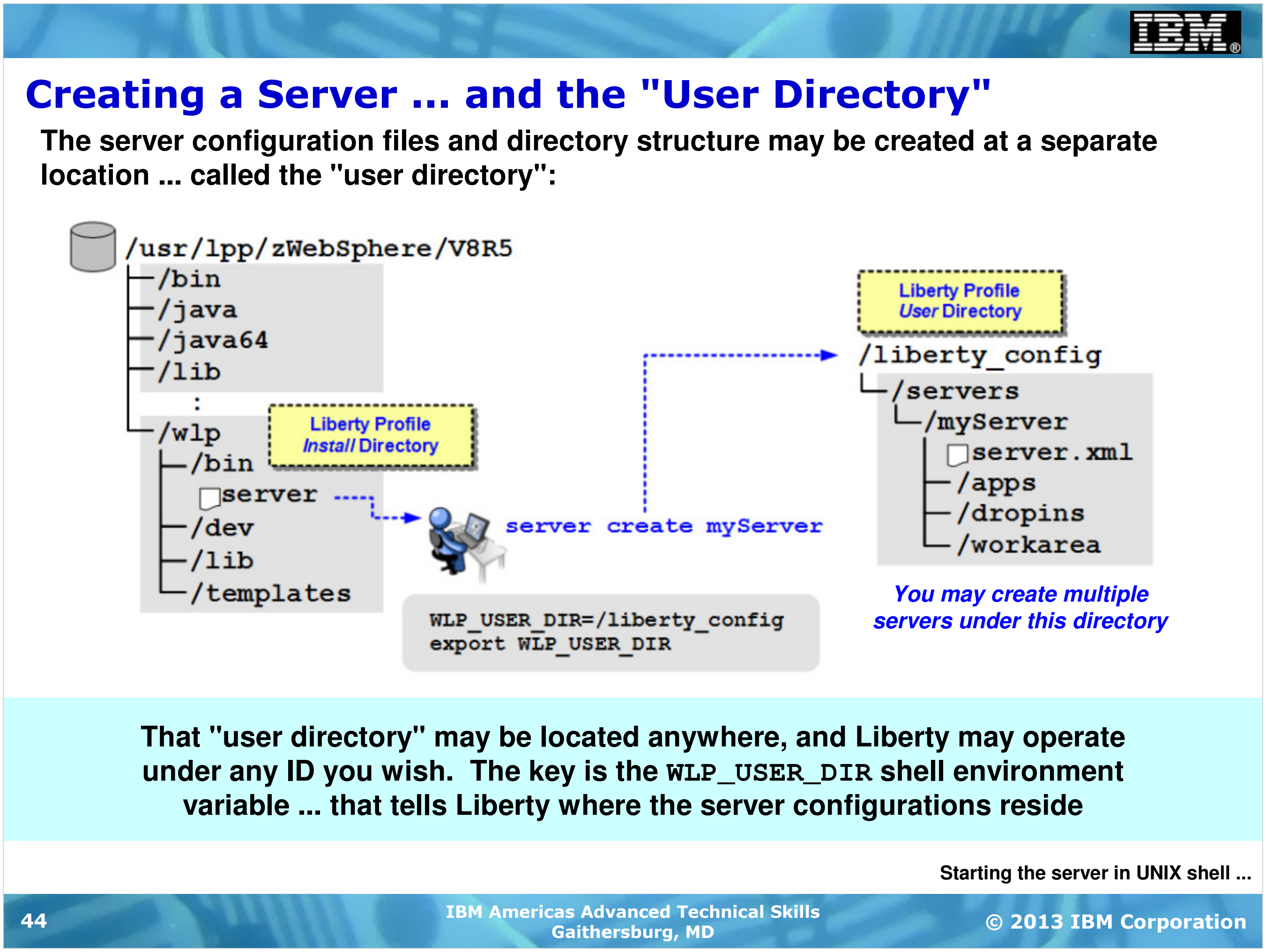
43

IBM Americas Advanced Technical Skills
Gaithersburg, MD

© 2013 IBM Corporation

Note: This page shows what things look like for WAS z/OS V8.5.5.

With WAS z/OS 8.5.5, Liberty is broken out as a separate "package" to install. This implies an installation directory outside the WAS z/OS install location. Once installed, the Liberty directory structure for 8.5.5 is the same as shown for 8.5 throughout this unit.



Creating a Server ... and the "User Directory"

The server configuration files and directory structure may be created at a separate location ... called the "user directory":

```
/usr/lpp/zWebSphere/V8R5
├─/bin
├─/java
├─/java64
├─/lib
│    :
└─/wlp
   ├─/bin
   │   server
   ├─/dev
   ├─/lib
   └─/templates
```

Liberty Profile *Install* Directory

`server create myServer`

```
WLP_USER_DIR=/liberty_config
export WLP_USER_DIR
```

Liberty Profile *User* Directory

```
/liberty_config
└─/servers
  └─/myServer
     │ server.xml
     ├─/apps
     ├─/dropins
     └─/workarea
```

You may create multiple servers under this directory

That "user directory" may be located anywhere, and Liberty may operate under any ID you wish. The key is the `WLP_USER_DIR` shell environment variable ... that tells Liberty where the server configurations reside

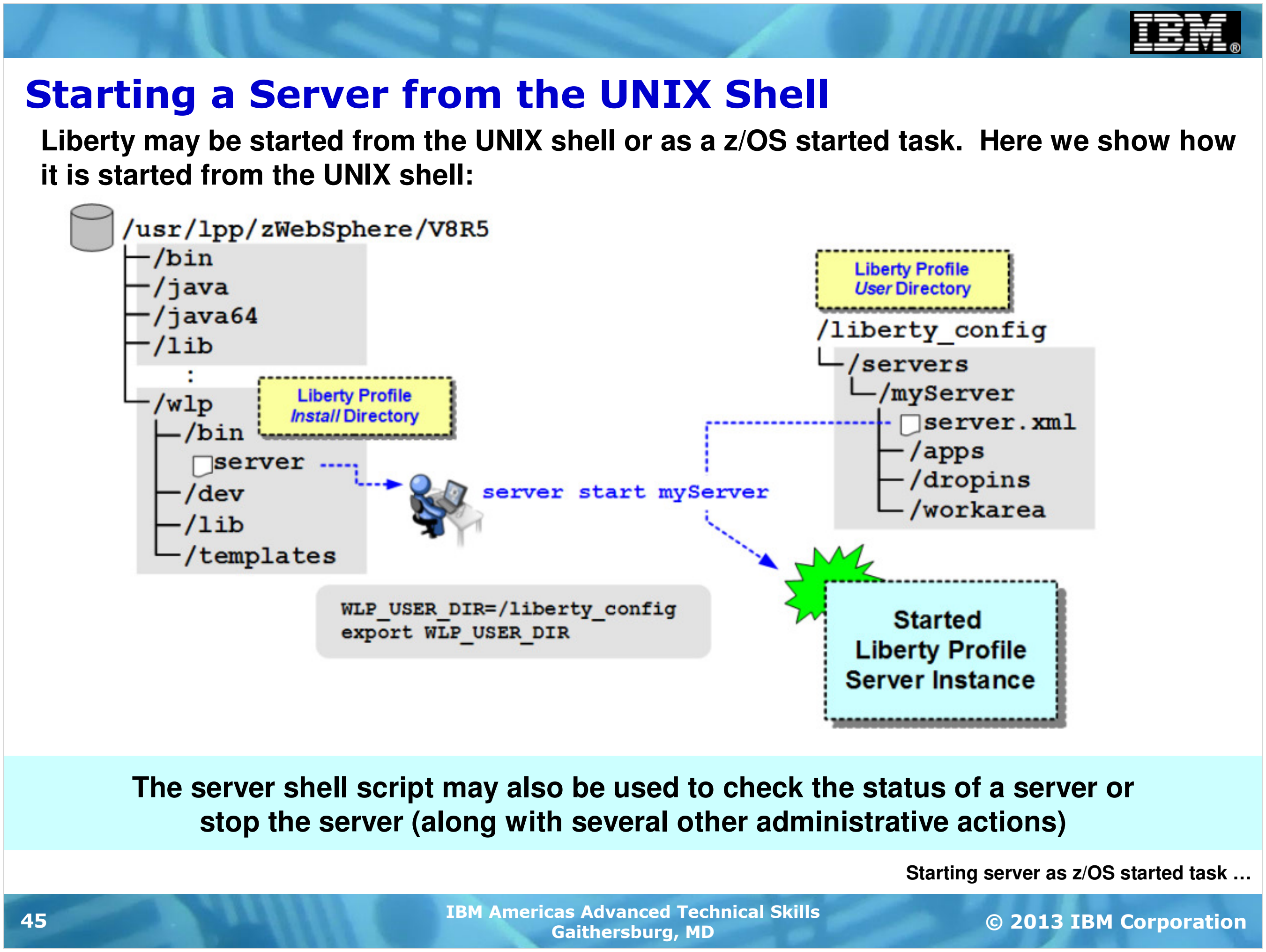
Starting the server in UNIX shell ...

44 IBM Americas Advanced Technical Skills Gaithersburg, MD © 2013 IBM Corporation

The files that get installed with IM do not constitute a server configuration. They are simply the Liberty Profile product files. The create a server configuration requires the use of one of those installed files ... the `server` shell script. That shell script will create a server configuration in whatever directory is named on the UNIX environment variable `WLP_USER_DIR`. In the chart we're showing the configuration being created under a directory called `/liberty_config` ... but your value may be whatever you wish. The only requirement is that the ID you run server with has write access, and there's space at the target location for the relatively small size of a configuration.

Note: it *is* possible create a server configure *within* the installation directory, meaning: under `/wlp`. If your UNIX environment did not have `WLP_USER_DIR` set then by default that's where the server shell script would try to create it. But the installation file system for WAS z/OS is typically mounted read-only. Which is why we're illustrating the setting of `WLP_USER_DIR` environment variable and creating the server configuration at a different location.

The example above is showing the server "myServer" being created. But you may create any number of servers under a `WLP_USER_DIR` location. In an upcoming chart we're going to show how Liberty has a set of built-in variables that may be used for location subtitution, and that allows for some really useful sharing of configuration elements and sharing of applications across servers.



Starting a Server from the UNIX Shell

Liberty may be started from the UNIX shell or as a z/OS started task. Here we show how it is started from the UNIX shell:

```
/usr/lpp/zWebSphere/V8R5
├─/bin
├─/java
├─/java64
├─/lib
│    :
└─/wlp
   ├─/bin
   │  server
   ├─/dev
   ├─/lib
   └─/templates
```

Liberty Profile *Install* Directory

Liberty Profile *User* Directory

```
/liberty_config
└─/servers
   └─/myServer
      ├ server.xml
      ├─/apps
      ├─/dropins
      └─/workarea
```

```
server start myServer
```

```
WLP_USER_DIR=/liberty_config
export WLP_USER_DIR
```

Started Liberty Profile Server Instance

The server shell script may also be used to check the status of a server or stop the server (along with several other administrative actions)

Starting server as z/OS started task …

45

IBM Americas Advanced Technical Skills
Gaithersburg, MD

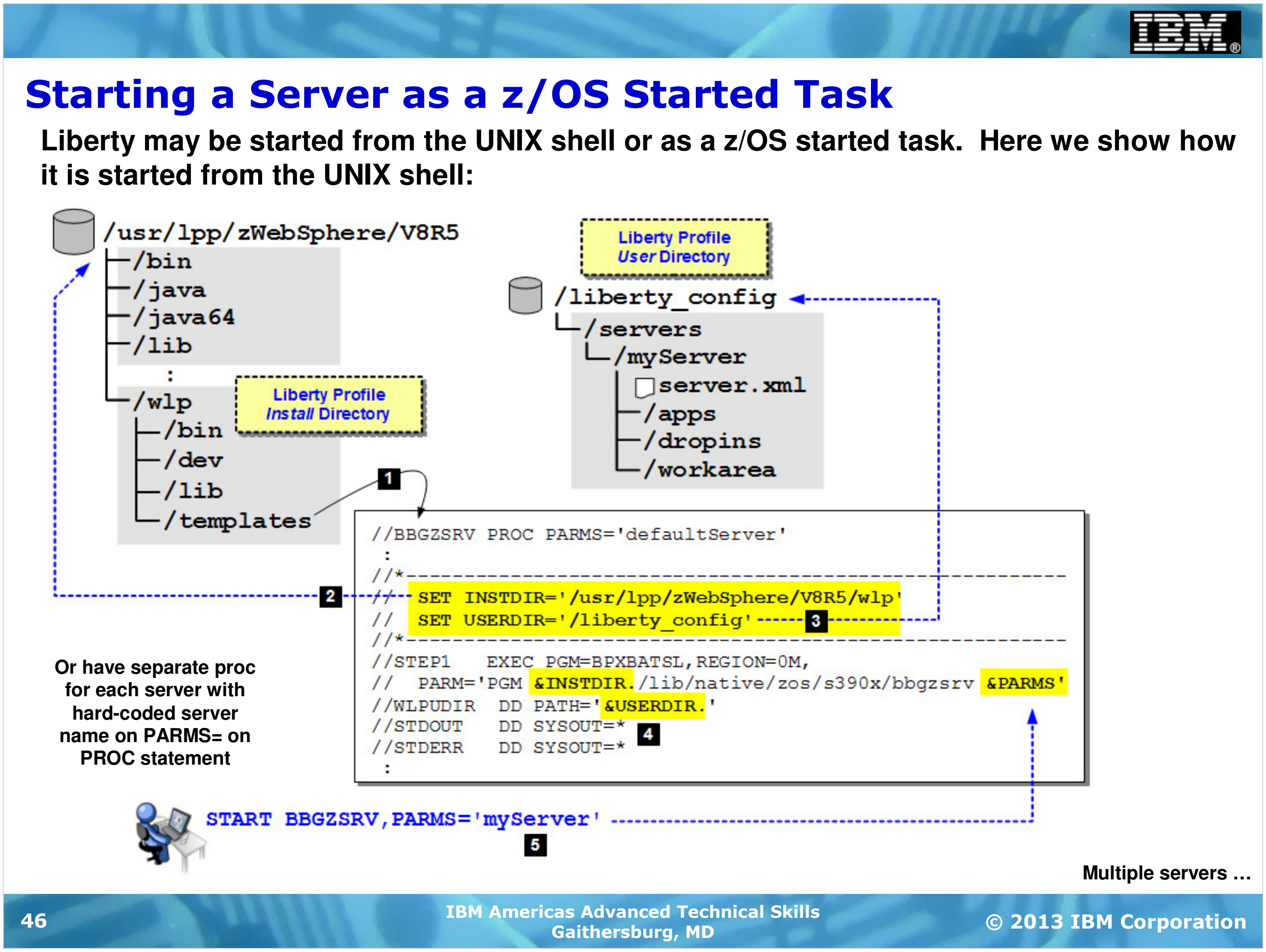
© 2013 IBM Corporation

Once you have the configuration constructed -- created using the `server` shell script and the `server.xml` modified to your needs -- you may start the server instance. This may be done in the UNIX shell or as a z/OS started task. In this chart we're showing how to start the server instance in the UNIX shell.

Again, the `server` shell script comes into play (the `server` shell script has several uses, creating servers and starting them are but two). With the UNIX environment aware of `WLP_USER_DIR`, the server shell script knows where to access the configuration files. The server named on the `server start` command refers to a directory under the `WLP_USER_DIR` location. The server.xml file is read and the Liberty Profile server instance is started.

If you had another server under the `WLP_USER_DIR` location you could start it using the same process. Then you'd have *two* server instances started.

The started servers may be stopped using the server shell script as well.



Starting a Server as a z/OS Started Task

Liberty may be started from the UNIX shell or as a z/OS started task. Here we show how it is started from the UNIX shell:

```
/usr/lpp/zWebSphere/V8R5
├─/bin
├─/java
├─/java64
├─/lib
│   :
└─/wlp
   ├─/bin
   ├─/dev
   ├─/lib
   └─/templates
```

```
/liberty_config
└─/servers
   └─/myServer
      ├ server.xml
      ├─/apps
      ├─/dropins
      └─/workarea
```

```
//BBGZSRV PROC PARMS='defaultServer'
  :
//*------------------------------------------------------------------
//   SET INSTDIR='/usr/lpp/zWebSphere/V8R5/wlp'
//   SET USERDIR='/liberty_config'
//*------------------------------------------------------------------
//STEP1    EXEC PGM=BPXBATSL,REGION=0M,
//   PARM='PGM &INSTDIR./lib/native/zos/s390x/bbgzsrv &PARMS'
//WLPUDIR   DD PATH='&USERDIR.'
//STDOUT    DD SYSOUT=*
//STDERR    DD SYSOUT=*
  :
```

Or have separate proc for each server with hard-coded server name on PARMS= on PROC statement

```
START BBGZSRV,PARMS='myServer'
```

Multiple servers …

46

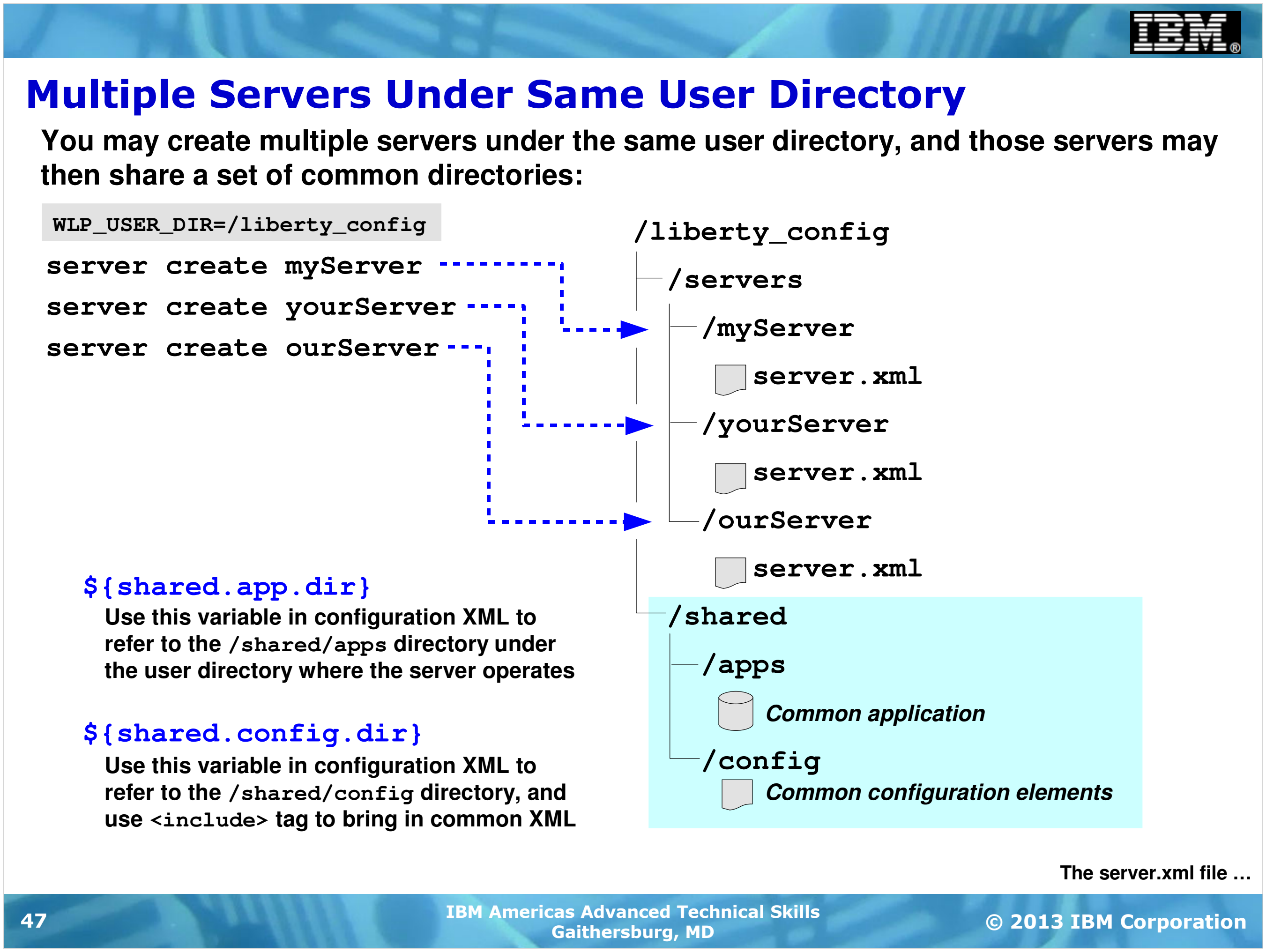
IBM Americas Advanced Technical Skills
Gaithersburg, MD

© 2013 IBM Corporation

If you prefer to run the Liberty Profile server instances as z/OS started tasks, that may be done as well. Liberty comes with a sample JCL start procedure called `BBGZSRV`. The numbered blocks on the chart correspond to the following explanations:

1. The sample `BBGZSRV` proc is supplied under the Liberty Profile installation `/templates` path. Copy that out to your proclib.
2. The `INSTDIR` substitution variable is used to indicate where the WAS V8.5 and Liberty files are located.
3. The `USERDIR` substitution variable is used to indicate where the server configuration files are
4. The values for `INSTDIR` and `USERDIR` are substituted in
5. The server name is passed in using `PARMS=` on the `START` command

The server starts up as a z/OS started task with, by default, the name `BBGZSRV`. You may give it a separate `JOBNAME` if you wish, or rename the JCL start proc to something different (and update the RACF `STARTED` profile accordingly).



Multiple Servers Under Same User Directory

You may create multiple servers under the same user directory, and those servers may then share a set of common directories:

```
WLP_USER_DIR=/liberty_config
server create myServer
server create yourServer
server create ourServer
```

```
/liberty_config
  /servers
    /myServer
      server.xml
    /yourServer
      server.xml
    /ourServer
      server.xml
  /shared
    /apps
      Common application
    /config
      Common configuration elements
```

`${shared.app.dir}`

Use this variable in configuration XML to refer to the `/shared/apps` directory under the user directory where the server operates

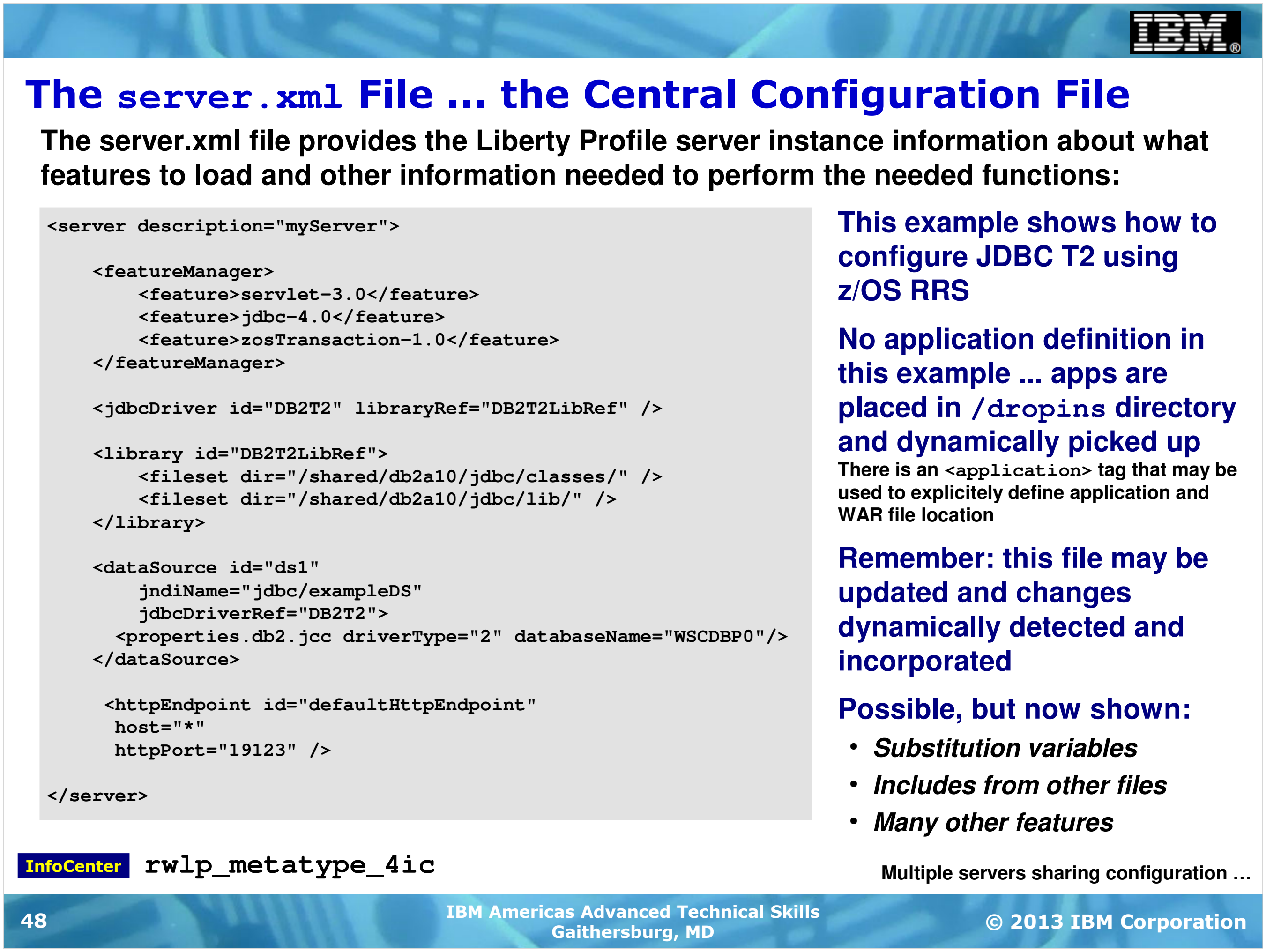
`${shared.config.dir}`

Use this variable in configuration XML to refer to the `/shared/config` directory, and use `<include>` tag to bring in common XML

The server.xml file …

The server create command may be used to create multiple servers under the same directory. The benefit in doing this is it allows you to take advantage of built-in configuration variables that resolve to directories you may create under the common `WLP_USER_DIR` directory. Two such variables are shown on the chart -- one that resolves to a shared application directory, and one that resolves to a shared configuration elements directory.

The benefit of this is it allows you to configure a group of servers, each that use these built-in variables to point back to a single, common set of applications or definitions. Change the common artifact in the shared directory and all the servers that make use of that shared artifact will detect the change and pick it up. In a few charts we'll show a hypothetical 99 server configuration that makes use of these built-in variables.



The `server.xml` File … the Central Configuration File

The server.xml file provides the Liberty Profile server instance information about what features to load and other information needed to perform the needed functions:

```
<server description="mySever">

    <featureManager>
        <feature>servlet-3.0</feature>
        <feature>jdbc-4.0</feature>
        <feature>zosTransaction-1.0</feature>
    </featureManager>

    <jdbcDriver id="DB2T2" libraryRef="DB2T2LibRef" />

    <library id="DB2T2LibRef">
        <fileset dir="/shared/db2a10/jdbc/classes/" />
        <fileset dir="/shared/db2a10/jdbc/lib/" />
    </library>

    <dataSource id="ds1"
        jndiName="jdbc/exampleDS"
        jdbcDriverRef="DB2T2">
      <properties.db2.jcc driverType="2" databaseName="WSCDBP0"/>
    </dataSource>

     <httpEndpoint id="defaultHttpEndpoint"
      host="*"
      httpPort="19123" />

</server>
```

This example shows how to configure JDBC T2 using z/OS RRS

No application definition in this example … apps are placed in `/dropins` directory and dynamically picked up

There is an `<application>` tag that may be used to explicitely define application and WAR file location

Remember: this file may be updated and changes dynamically detected and incorporated

Possible, but now shown:

- ***Substitution variables***
- ***Includes from other files***
- ***Many other features***

InfoCenter **rwlp_metatype_4ic**

Multiple servers sharing configuration …

48

IBM Americas Advanced Technical Skills
Gaithersburg, MD

© 2013 IBM Corporation

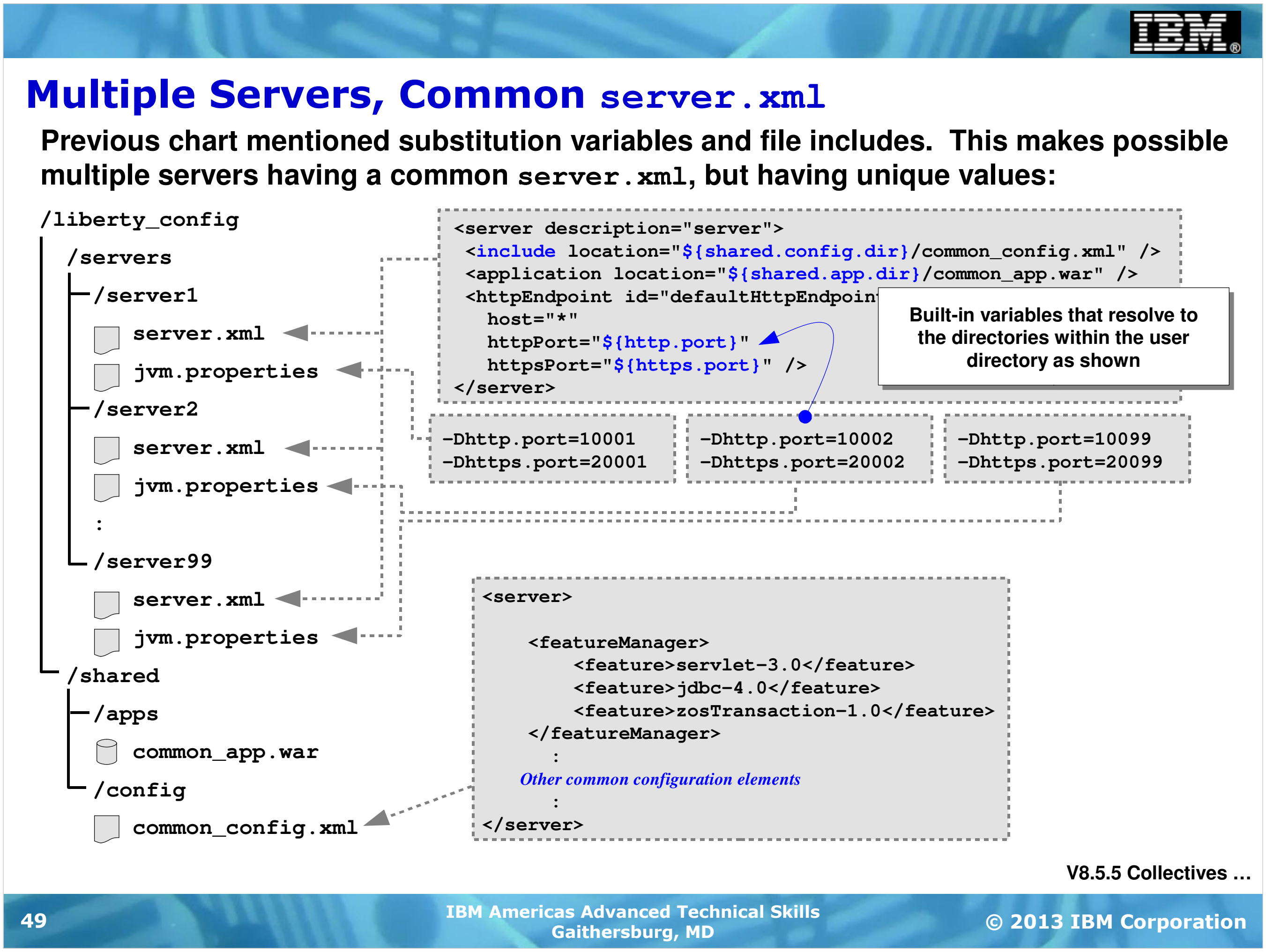
As mentioned, the `server.xml` file serves as the *primary* (and only *required*) configuration file. Other configuration files exist -- `server.env` (UNIX environment variables), `jvm.properties` (custom properties for the JVM), and `bootstrap.properties` (influences settings at initial bootstrap), but those are optional.

The `server.xml` is edited manually ... you add or remove function as your server instance and applications require. Remember that changes to `server.xml` are dynamically detected and implemented.

In this example we're showing the full `server.xml` for a server that will host a JDBC Type 2 connection to DB2 using RRS as its TX manager. The InfoCenter search string at the bottom of the chart gives you easy access to the InfoCenter page with specifics on the XML elements.

Note that there are no application definitions in this `server.xml` example. That's because Liberty has the ability to monitor a directory called `/dropins` for application files. Application WAR files seen there are automatically loaded; and if those files change (or new files appear) those are automatically loaded.

This is just one example of `server.xml`, and as such we're not showing all (or even most) the various configuration options. In time you'll develop a set of working XML files with the help of the InfoCenter and other documentation.



Multiple Servers, Common `server.xml`

Previous chart mentioned substitution variables and file includes. This makes possible multiple servers having a common `server.xml`, but having unique values:

```
/liberty_config
   /servers
      /server1
         server.xml
         jvm.properties
      /server2
         server.xml
         jvm.properties
         :
      /server99
         server.xml
         jvm.properties
   /shared
      /apps
         common_app.war
      /config
         common_config.xml
```

```
<server description="server">
 <include location="${shared.config.dir}/common_config.xml" />
 <application location="${shared.app.dir}/common_app.war" />
 <httpEndpoint id="defaultHttpEndpoin
   host="*"
   httpPort="${http.port}"
   httpsPort="${https.port}" />
</server>
```

Built-in variables that resolve to the directories within the user directory as shown

```
-Dhttp.port=10001
-Dhttps.port=20001
```

```
-Dhttp.port=10002
-Dhttps.port=20002
```

```
-Dhttp.port=10099
-Dhttps.port=20099
```

```
<server>

    <featureManager>
        <feature>servlet-3.0</feature>
        <feature>jdbc-4.0</feature>
        <feature>zosTransaction-1.0</feature>
    </featureManager>
        :
    Other common configuration elements
        :
</server>
```

V8.5.5 Collectives ...

49

IBM Americas Advanced Technical Skills
Gaithersburg, MD

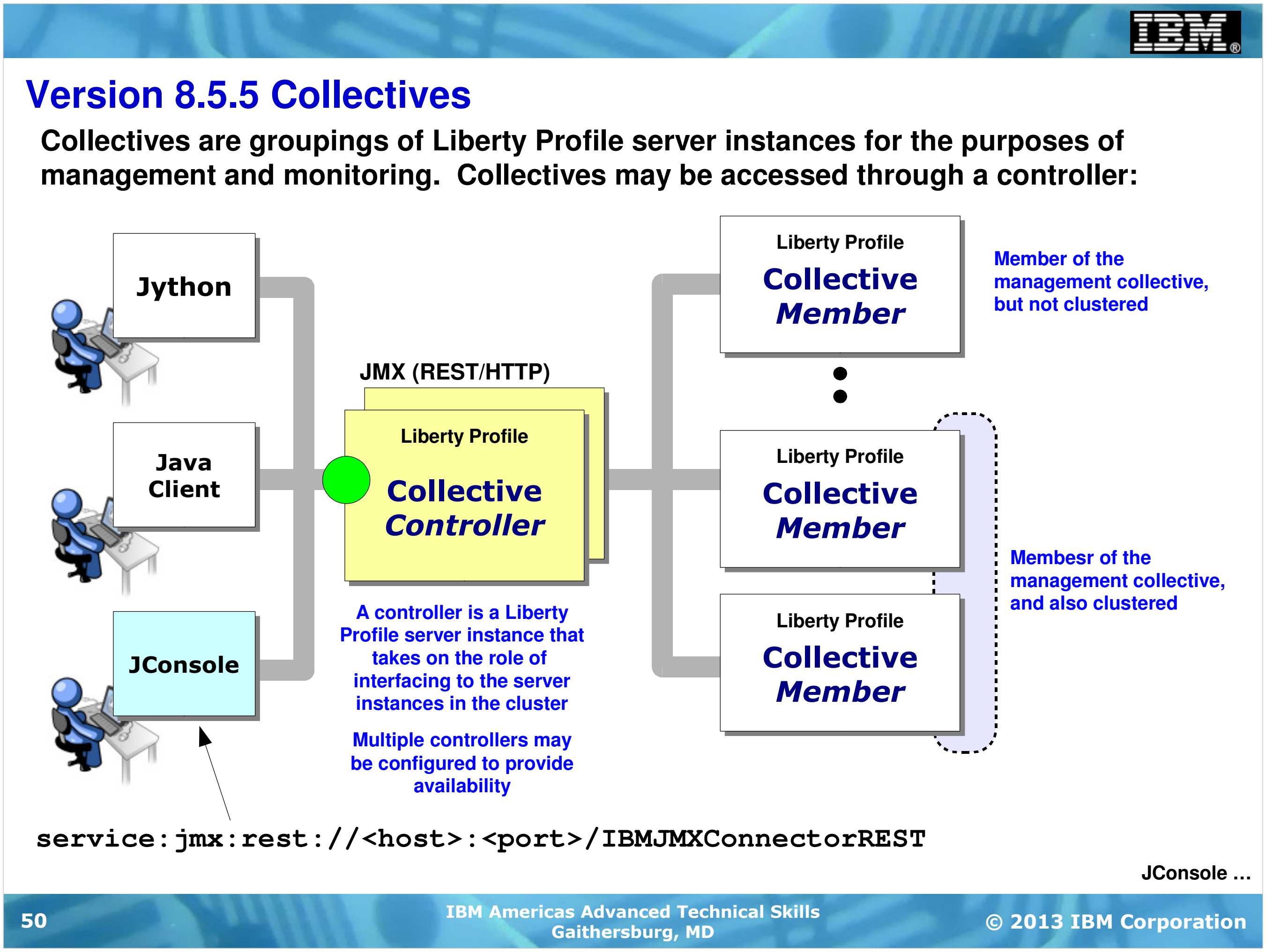
Liberty Profile provides a means of sharing configuration information across multiple servers. This means it's quite possible to have many server instances sharing a common `server.xml` file and a common set of application files.

In the chart above we're illustrating:

- The common `server.xml` used by multiple servers. This XML makes use of substitution variables to provide uniqueness as well as pointers to XML to be included dynamically and a pointer to a common application file.
- A `jvm.properties` file for each server which provides unique HTTP ports. Those values are then substituted into the common server.xml.
- The including of a common set of XML mainained at a shared location. Updates to this shared include file are dynamically propagated to each server and update dynamically.
- A pointer to an application that's maintained in a shared location. Updates to this application are dynamically detected and updated as well.

The point here is that multiple Liberty Profile server instances do not necessarily have to be administered completely isolated from one another. It is quite possible to construct your configuration such that common elements are held in shared locations, and those locations referenced using built-in substitution values.

There is considerable flexibility in how Liberty Profile may be configured and operated. This is one example.



Version 8.5.5 Collectives

Collectives are groupings of Liberty Profile server instances for the purposes of management and monitoring. Collectives may be accessed through a controller:

Jython

Java Client

JConsole

JMX (REST/HTTP)

Liberty Profile **Collective Controller**

A controller is a Liberty Profile server instance that takes on the role of interfacing to the server instances in the cluster

Multiple controllers may be configured to provide availability

Liberty Profile **Collective Member**

Member of the management collective, but not clustered

Liberty Profile **Collective Member**

Liberty Profile **Collective Member**

Membesr of the management collective, and also clustered

`service:jmx:rest://<host>:<port>/IBMJMXConnectorREST`

JConsole …

With Liberty Profile as delivered in V8.5.5 there is a new management construct provided to make administration of multiple Liberty Profile server instances better and more efficient.

In V8.5 it was possible to create a number of Liberty Profile server instances and they could have some association with one another, as we showed in the previous chart with the 100 servers all operating cooperatively with shared configuration elements. Nice, but it left you the administrator having to individually manage each for functions such as starting and stopping and any changes to the server-specific configuration files.

With V8.5.5 the concept of a *collective* has been introduced. A collective is a grouping of Liberty Profile server instances that may be managed through a collective controller. A controller is another Liberty Profile server instance configured to serve the role of controller. Liberty Profile servers that are members of the collective signal their intent to be members through a bit of XML in the `server.xml` file that indicates host and port of the controller. The server signals to the controller and the controller then understands who is in its collective.

You then interface to the collective through the controller using one of the methods shown on the chart -- a Jython script, a JConsole session, or your own Java client. Local or RESTful connectors are used to connect to the controller, which then routes the JMX requests to the target collective member to take the action specified -- start server, stop server, get information about a server, transfer a file from a server or transfer a file to a server.

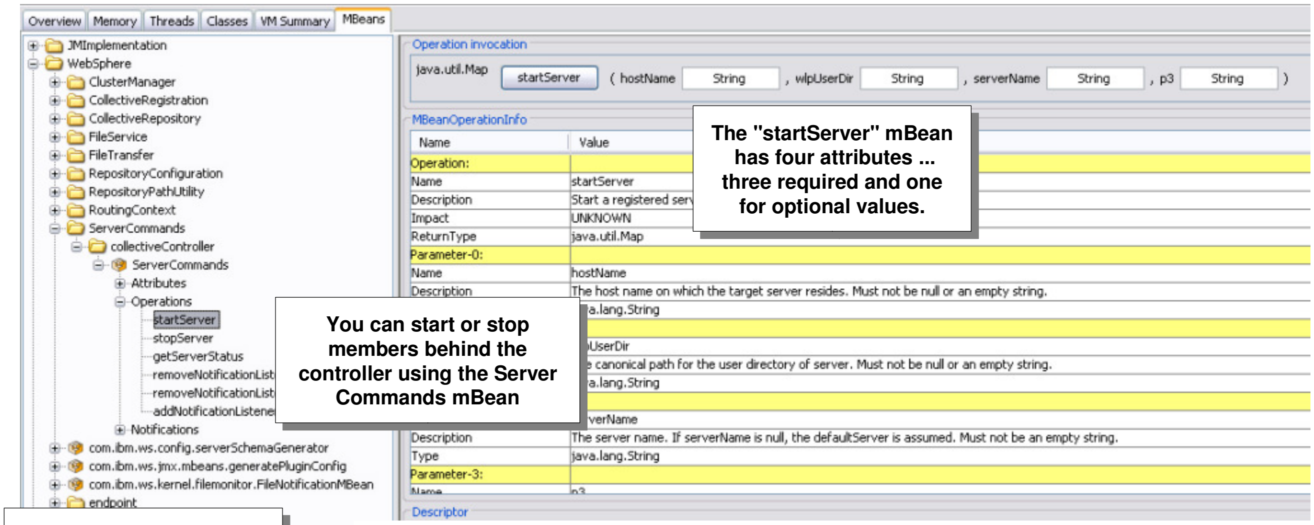
Note: unlike traditional WAS with its Node Agent design, this collective has no agents. The communication is direct between the controller and the members of the collective.

Members of a collective are not reliant on the controller to operate. So a controller can be down and the server members will still function. That said, having a highly available controller is desirable, so multiple copies of a controller can be started and they will share information about the members they are helping manage.

Finally, Liberty Profile 8.5.5 has the concept of a *cluster*, which is a grouping of server instances for the purposes of hosting applications in a highly available manner. Liberty Profile 8.5 allowed you to have multiple servers with the same application, but that was it. With Liberty 8.5.5 there's now caching that can be shared across cluster members, as well as the ability to use a HTTP Server Plugin to distribute work to Liberty cluster members.

Version 8.5.5 Collectives - JConsole Example

JConsole provides a GUI interface to issue JMX commands to the controller, which then routes to the target member or cluster:



Name	Value
Operation:	
Name	startServer
Description	Start a registered ser
Impact	UNKNOWN
ReturnType	java.util.Map
Parameter-0:	
Name	hostName
Description	The host name on which the target server resides. Must not be null or an empty string.
	a.lang.String
	UserDir
	e canonical path for the user directory of server. Must not be null or an empty string.
	a.lang.String
	verName
Description	The server name. If serverName is null, the defaultServer is assumed. Must not be an empty string.
Type	java.lang.String
Parameter-3:	
Name	p3

The "startServer" mBean has four attributes ... three required and one for optional values.

You can start or stop members behind the controller using the Server Commands mBean

JConsole showing the mBeans exploded by the JMX RESTful interface of the controller

Your Java client or Jython scripts would do essentially the same thing -- connect to the Controller JMX interface and invoke the mBeans to perform the operations supplied by the mBeans

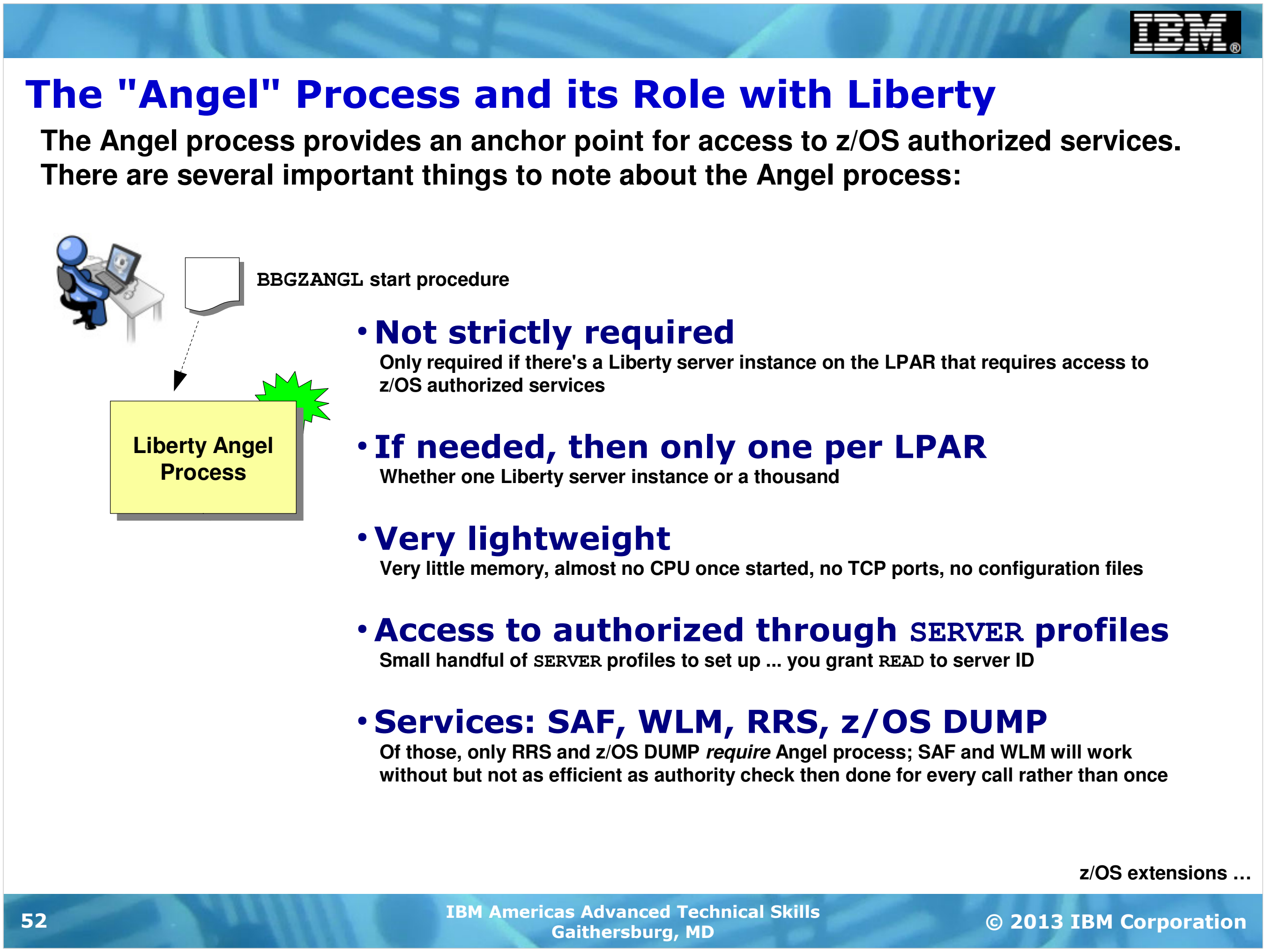
InfoCenter `rwlp_mbeans_list`

Angel process ...

This chart is showing an example of what JConsole looks like when conntected to a controller and the "mBeans" tab is expanded. Here we're showing the "ServerCommands" mBeans expanded out, with "startServer" highlighted. Using this console, you can specify the server you wish to start. The command is passed to the controller, which then issues the command to start the server.

A Jython script or your own Java client would do the same thing -- use the JMX interface of the controller to invoke the mBeans to perform the specified actions. JConsole provides a relatively simple interface to the controller mBeans; your own Java client could be far more sophisticated, as could Jython scripts.

Many mBeans are provided -- not just starting and stopping servers. The "InfoCenter" tag in the lower left of the screen provides a search string to use in the 8.5 InfoCenter to see a reference list of provided mBeans.



The "Angel" Process and its Role with Liberty

The Angel process provides an anchor point for access to z/OS authorized services. There are several important things to note about the Angel process:

BBGZANGL start procedure

Liberty Angel Process

- **Not strictly required**
  Only required if there's a Liberty server instance on the LPAR that requires access to z/OS authorized services
- **If needed, then only one per LPAR**
  Whether one Liberty server instance or a thousand
- **Very lightweight**
  Very little memory, almost no CPU once started, no TCP ports, no configuration files
- **Access to authorized through `SERVER` profiles**
  Small handful of `SERVER` profiles to set up ... you grant `READ` to server ID
- **Services: SAF, WLM, RRS, z/OS DUMP**
  Of those, only RRS and z/OS DUMP *require* Angel process; SAF and WLM will work without but not as efficient as authority check then done for every call rather than once

z/OS extensions ...

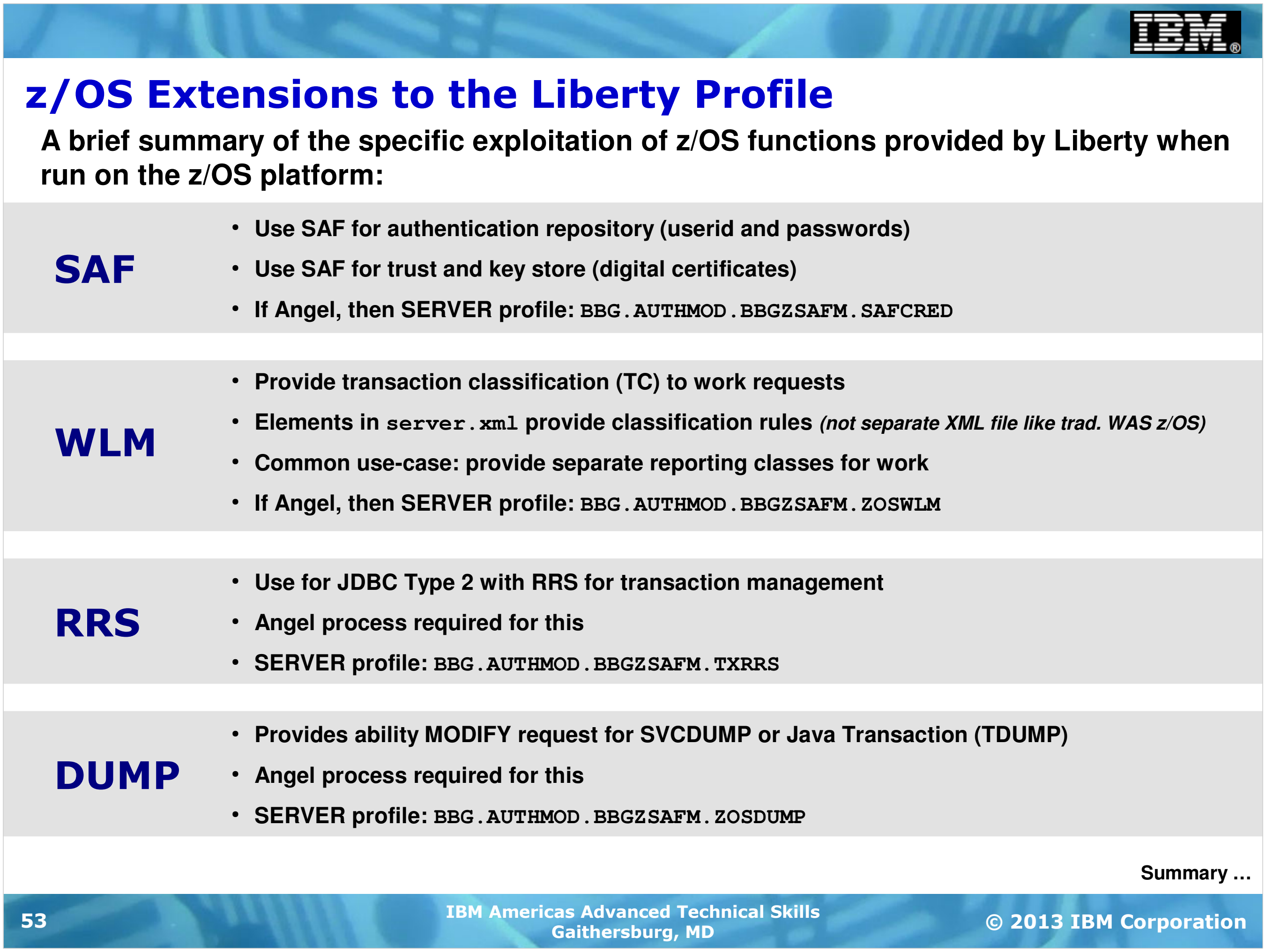
There is one more piece to this puzzle, and it's called the "Angel process" (not to be confused in any way with the traditional WAS z/OS Daemon). The Angel process provides an anchor point for access to z/OS authorized services. As such the Angel process is *optional* -- if no servers need to use z/OS authorized services then you do not need the Angel process; and even if some do require access to the z/OS extensions some of those do not require authorized access. It's only *required* when you have servers that require authorized access.

If needed, then only one is needed per z/OS LPAR, regardless of the number of Liberty Profile server instances on the LPAR.

It's very lightweight. It takes very little storage and almost no CPU once started. It requires no TCP ports. It's started and just *sits* there. But it sits there and provides the anchor point for authorized access.

Access to z/OS authorized services is provided via SAF `SERVER` profiles. There's a handful of such profiles very clearly document in the InfoCenter. We'll give you a sense for this on the next chart.

The services that might require or benefit from the Angel process are shown on the last bullet of the chart. These are the z/OS extensions implemented in the WAS V8.5 Liberty Profile for z/OS. Of those, only RRS and DUMP require the Angel. SAF and WLM have unauthorized access paths, but that implies an authority check for each call and that implies some inefficiency. Having authorized access eliminates that overhead. So while the Angel process is not strictly require, it might be beneficial from a performance perspective to have it for SAF and WLM.



z/OS Extensions to the Liberty Profile

A brief summary of the specific exploitation of z/OS functions provided by Liberty when run on the z/OS platform:

SAF

- Use SAF for authentication repository (userid and passwords)
- Use SAF for trust and key store (digital certificates)
- If Angel, then SERVER profile: `BBG.AUTHMOD.BBGZSAFM.SAFCRED`

WLM

- Provide transaction classification (TC) to work requests
- Elements in `server.xml` provide classification rules *(not separate XML file like trad. WAS z/OS)*
- Common use-case: provide separate reporting classes for work
- If Angel, then SERVER profile: `BBG.AUTHMOD.BBGZSAFM.ZOSWLM`

RRS

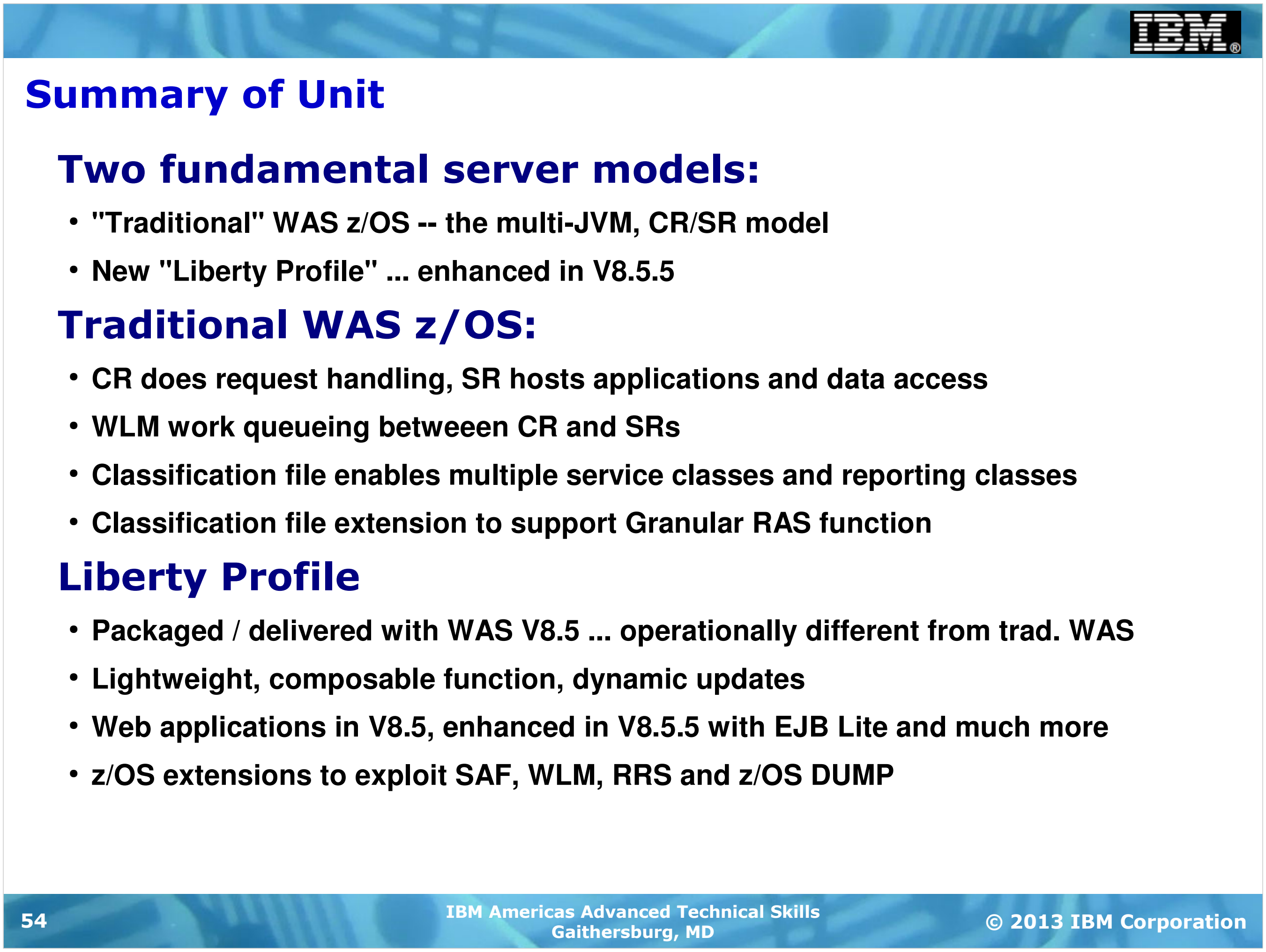
- Use for JDBC Type 2 with RRS for transaction management
- Angel process required for this
- SERVER profile: `BBG.AUTHMOD.BBGZSAFM.TXRRS`

DUMP

- Provides ability MODIFY request for SVCDUMP or Java Transaction (TDUMP)
- Angel process required for this
- SERVER profile: `BBG.AUTHMOD.BBGZSAFM.ZOSDUMP`

Summary …

This chart summarizes the z/OS extensions. Configuration examples are provided in the WP102110 Techdoc at `ibm.com/support/techdocs`.



Summary of Unit

Two fundamental server models:

- "Traditional" WAS z/OS -- the multi-JVM, CR/SR model
- New "Liberty Profile" ... enhanced in V8.5.5

Traditional WAS z/OS:

- CR does request handling, SR hosts applications and data access
- WLM work queueing betweeen CR and SRs
- Classification file enables multiple service classes and reporting classes
- Classification file extension to support Granular RAS function

Liberty Profile

- Packaged / delivered with WAS V8.5 ... operationally different from trad. WAS
- Lightweight, composable function, dynamic updates
- Web applications in V8.5, enhanced in V8.5.5 with EJB Lite and much more
- z/OS extensions to exploit SAF, WLM, RRS and z/OS DUMP

And the summary to this unit.

End of Unit