

WebSphere Application Server z/OS



Deciding Which to Use



The answer may well be “both” ... the intent of this material is to help you understand and weigh the considerations of both against the needs of your application serving architecture. Utilizing *both* WAS traditional and Liberty is a pattern that may fit your needs.

Hyperlink



Executive Overview

A one-chart summary of the usage considerations presented in the document



Setting Context

Establishing terminology and providing background on the evolution over time of each runtime models



Application Considerations

Exploring the application interface considerations of each runtime model



Operational Considerations

Exploring the runtime operational considerations of each runtime model



Performance Considerations

Exploring the performance profile of each runtime model



Other Information for Consideration

A collection of other information you may find useful when making this decision

2

This chart shows the sections of this presentation. The green arrows to the left are hyperlinks to the starting page for the section. (When viewing this in screen show mode, or the PDF of the charts. When reading the notes pages the chart is rendered as an image and the hyperlinks are no longer in effect.)



Executive Overview

Executive Summary

Liberty is the newer runtime model and has considerable IBM focus and investment

WAS traditional z/OS continues to be a viable platform with IBM support into future

Liberty z/OS benefits include: smaller memory footprint, greater zIIP offload, more flexible configuration and application deployment

If there is a business driver to consider moving to Liberty, then:

- **Determine the viability of moving the applications to Liberty**
- **Assess the operational differences and determine if any value is diminished by moving**
- **If value exceeds cost, then it's a net benefit to the business and a move should be considered**
- **If cost exceeds value, then maintain WAS traditional for those applications**
- **Maintaining *both* environments is possible and would provide a "best of both worlds" environment**

4

This chart is meant to be a one-page summarization of the message delivered in this presentation.

Liberty is receiving focus and investment from IBM, but that does not mean you must move off WAS traditional. The move to Liberty should be based on business needs and an assessment of the value Liberty provides. Liberty has its value -- most notably the smaller footprint, greater zIIP offload on z/OS, and flexibility -- but those value attributes should be weighed against the effort to move from WAS traditional to Liberty.

Focus first on the application and assess the viability of moving them to Liberty. If the application design permits moving to Liberty, then assess the operational considerations. If the results of the assessment suggests adoption of Liberty is in the best interests of the business, then a move to Liberty should be considered.

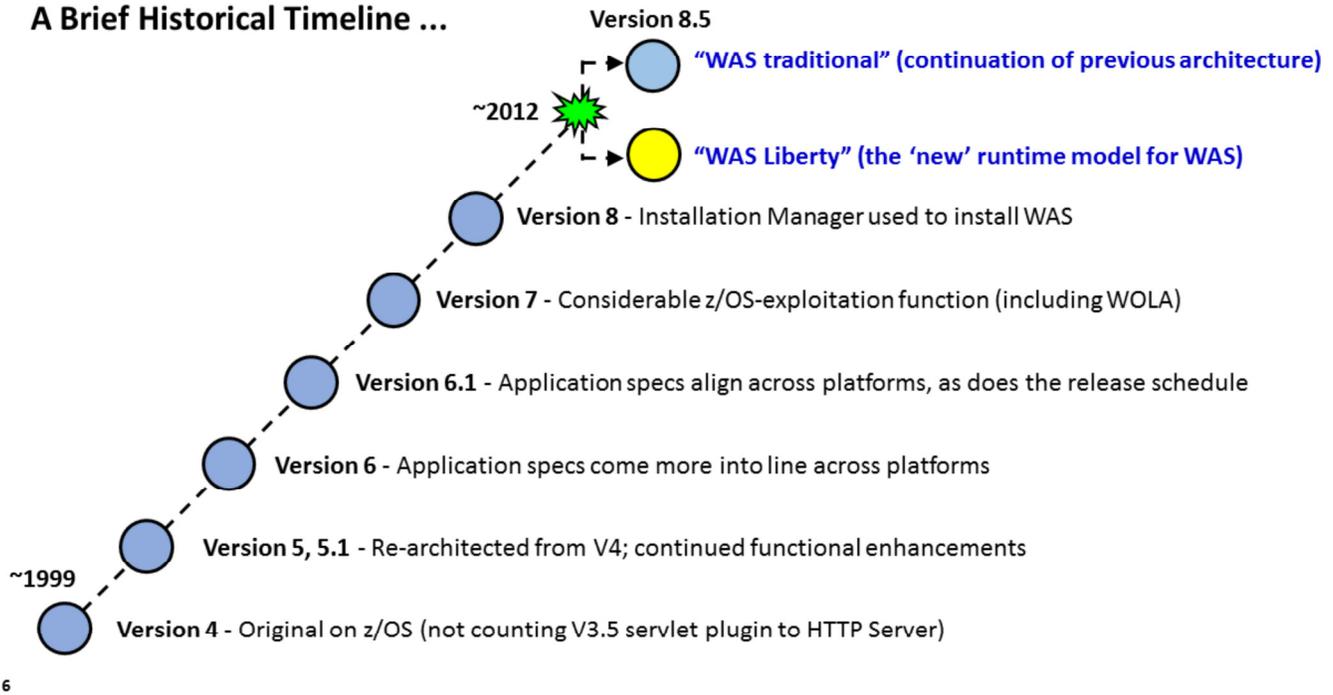
Finally, it is possible to maintain both environments. It may make sense to maintain both and deploy applications to the runtime environment best suited for the application.

Setting Context

5

In this section we will provide a bit of a higher-level overview of Liberty and provide information to help you place an evaluation of Liberty in proper context.

A Brief Historical Timeline ...



This chart is showing a summarized timeline of the releases of WebSphere Application Server over the years. As you can see, its roots go back well over a decade to Version 4, which was when the foundation for what we call "WAS traditional" today was formed. Over the years new releases and versions were introduced. Then in 2012 came the introduction of Liberty. At that point -- and continuing to this day -- we have two runtime models: WAS traditional, which is runtime model based on the original; and Liberty, which is the focus of this presentation.

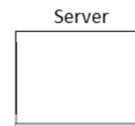
“WAS traditional” and “Liberty”

WAS traditional z/OS



- The original WAS, going back 15 years to Version 4
- On z/OS it consisted of “controllers” and “servants”
- It was organized into “nodes” and “cells”
- Specific function to exploit z/OS capabilities
- Considerable production-hardened investment here

Liberty z/OS



- First introduced in V8.5.0.0 (2012) all platforms
- Single JVM server model (no CR/SR)
- Key attributes: lightweight, composable, dynamic
- Has z/OS exploitation functions
- Under “continuous development” = frequent updates

Both fall under “WebSphere Application Server” umbrella, but are not the same thing
 (Which is why this positioning discussion is needed)

7

For those who may not be familiar with the two runtime models, this chart is meant to provide background on how the two manifest from a z/OS perspective.

On the left is what we now call “WAS traditional”. It is the original design going back 15 years to Version 4. On z/OS an “application server” consists of multiple address spaces -- a control region and one or more servant regions. This design is exclusive to z/OS; a key reason for this design is it allows WAS z/OS to take advantage of Workload Manager (WLM) of z/OS for work classification and placement. Management of a WAS traditional environment is performed within a “cell” (a set of servers that comprise a management domain), and “nodes” (a set of servers on an LPAR that are logically grouped). WAS z/OS has a number of functions that take direct advantage of the z/OS operating system, such as WLM, SAF, and cross-memory.

On the right is what we call Liberty. It manifests as a single address space on z/OS. As noted earlier, the key attributes for Liberty are that it is lightweight (smaller memory footprint; faster startup times); composable (through configuration you indicate what functions to load); and dynamic (most changes do not require a server restart). On z/OS there are points of platform exploitation as well -- such as WLM, SAF, cross-memory. Further, Liberty is developed under the “continuous development” model, which means functional updates are delivered on a more frequent basis than was done in the past with WAS traditional. It is common to see functional updates for Liberty delivered in the fixpack stream, which has a quarterly release cycle.

Both WAS traditional and Liberty fall under the “WebSphere Application Server” umbrella, and on z/OS both are delivered under the same licensing entitlement. This creates some confusion about the role of each, which is why a positioning discussion -- such as this paper provides -- is needed.

What was Behind Creation of Liberty?

WAS traditional ...

... loaded most functions even if applications did not require them

... required application and server restarts for most changes

... Has a mature, but somewhat inflexible management model

WAS Liberty ...

... is composable, allowing for customized function enablement

... is dynamic, allowing for application and configuration changes without restarts

... has a management model* that is, by design, flexible and highly scalable

WAS traditional has its architectural roots going back 15 years. Times change, and a more flexible and dynamic server model was needed. That is Liberty.

* Called "Collectives". More on that later in the presentation.

8

You may wonder why Liberty came along at all. This chart helps explain that.

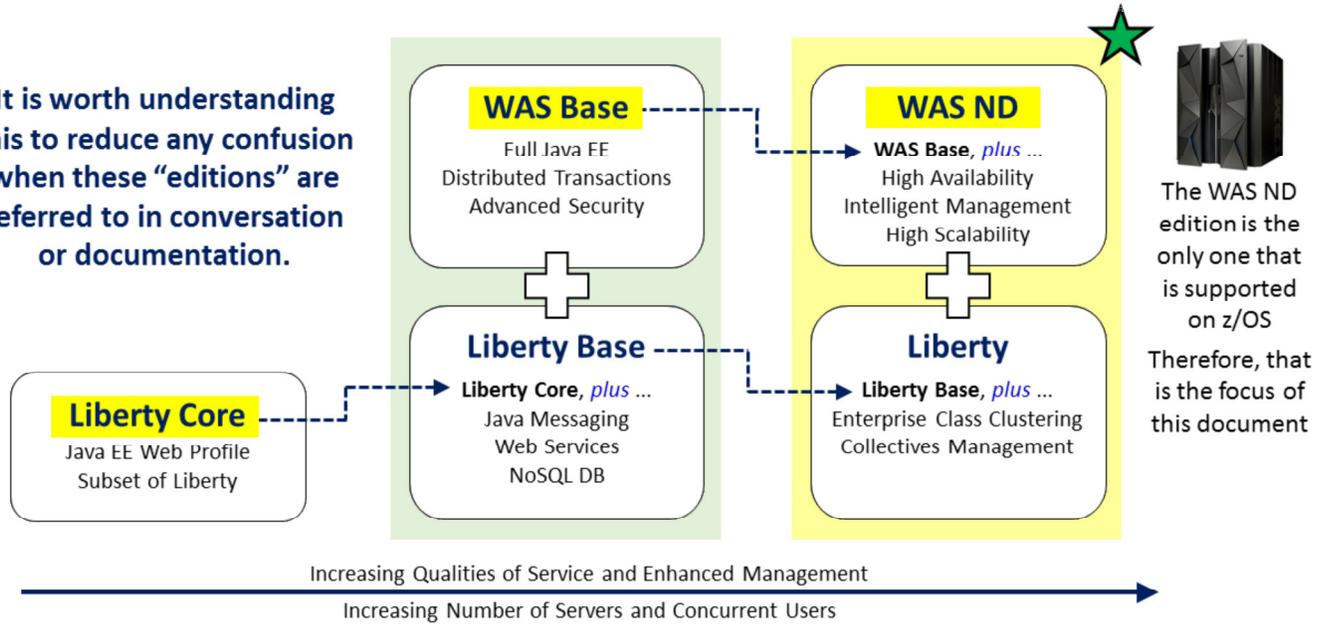
The creation of Liberty was in response to specific issues related to WAS traditional. This chart summarizes those on the left, and then indicating on the right how Liberty addressed the issues:

- Footprint -- WAS traditional was perceived as consuming too much memory. Liberty addresses this by being "composable"; that is, you indicate through configuration what functions to load, thus reducing the footprint of Liberty to what is needed for your application.
- Restarts -- WAS traditional required application restarts and server restarts for many changes. Liberty addresses this by having a dynamic update model. With Liberty most application and configuration changes can be picked up and made available without a server restart.
- Management -- WAS traditional has a management model based on a fixed design of "cells" and "nodes." It has served well over the last 15 years, but its design limits flexible changes to the topology. Further, the inter-server communication model limits the scaling of a WAS cell (more precisely, the WAS "core group") to about 700 or so servers. Liberty's management model -- called "collectives" -- is designed to be more flexible and better scaling.

That summarizes the "why" of Liberty being created by IBM. It's a story of the technical environment changing over time and a new application server design being created to address it.

Understanding WAS Product Terminology

It is worth understanding this to reduce any confusion when these “editions” are referred to in conversation or documentation.



The WAS ND edition is the only one that is supported on z/OS
Therefore, that is the focus of this document

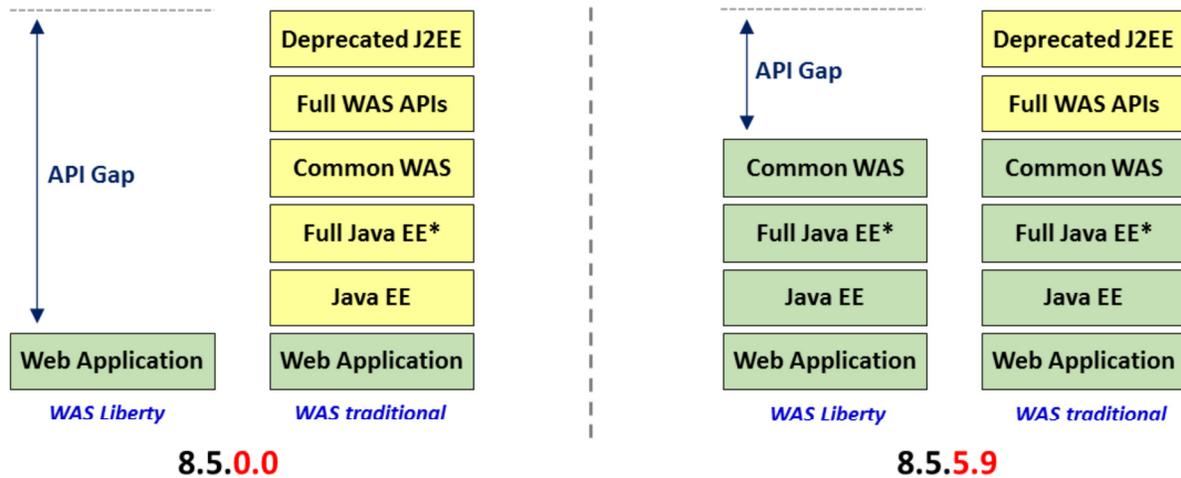
When discussing WebSphere Application Server, it’s worth noting the terminology that’s used to refer to the different offerings that are available.

- **Liberty Core** -- this is *not* available on z/OS; it is only available on distributed platforms. This is Liberty only (WAS traditional is not part of this offering), and it is a *subset* of Liberty. This is intended to be widely available, easily downloaded, and easy to acquire and use.
- **WAS Base** -- this is also *not* available on z/OS; it is only available on distributed platforms. This includes *both* WAS traditional and Liberty. The WAS traditional packaged with this offering is a subset of the full WAS traditional offered with the “WAS ND” packaging. The Liberty packaged with this offering includes Liberty Core and adds Java messaging, Web Services and NoSQL DB. But this Liberty is still a *subset* of the Liberty offered in the “WAS ND” packaging.
- **WAS ND** -- this *is* available on z/OS, as well as other platforms. Consider this the “complete” WAS ... both WAS traditional and Liberty.

As you look at this chart and go left to right, you see offerings that increase in qualities of service and enhanced management, and increase in numbers of servers and concurrent users.

For this presentation we are focusing on the far right ... WAS ND. That is what is offered for z/OS.

Differences in the Application Programming Interface (APIs)



Initially the gap was large, and some existing WAS traditional applications could not run on Liberty. Now, many (if not most) can run on Liberty with little or no changes.

10 * For Liberty: partial Java EE 6, full Java EE 7. For WAS traditional: Full Java EE6, Full Java EE 7 in beta

Here we take a look at the differences in the programming interfaces (APIs) between Liberty and WAS traditional, both at first (back in the 8.5.0.0 days) and today (8.5.5.9). This is a story of the API gap being fairly wide initially, but mostly closed at this point.

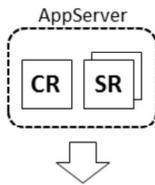
On the left side of this chart is the “API gap” as it existed between Liberty and WAS traditional when Liberty was first introduced back in the 8.5.0.0 time frame. The gap was fairly wide at that point. Liberty had the ability to host web applications only. WAS traditional had that plus Java EE and other WAS-specific APIs. At that point Liberty was clearly well short of WAS traditional, and at that point Liberty was mainly seen as a development platform with limited capabilities.

Fast forward to today (that is, the 8.5.5.9 time frame) and we see that the gap is mostly closed. Now Liberty and WAS traditional share the bottom four “blocks” in that stack diagram, with only the “Full WAS APIs” and “Deprecated J2EE” APIs existing in WAS traditional but not Liberty. This means application compatibility between Liberty and WAS traditional is much greater now than it was initially. The ability to move applications from WAS traditional to Liberty is greater now than it was before.

Note: and the reverse -- it is now easier to use Liberty as a development platform with WAS traditional as the target for production deployment. In the past, that worked if the application was just a web application. But now with Liberty and WAS traditional having more APIs in common, application movement from Liberty to WAS traditional is also a consideration.

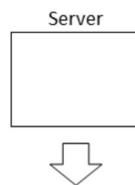
Greater zIIP Offload and Lower Cost

WAS traditional z/OS



~ 80% or perhaps higher offload

Liberty z/OS



~ 90% or perhaps higher offload

Many "it depends" qualifiers around these numbers

In general: WAS traditional has a greater degree of native code (not eligible for zIIP offload) supporting the Java runtime than does Liberty

Best way to determine offload difference is to benchmark specific application

In addition to the greater zIIP offload potential, it is possible the same workload running in Liberty would require fewer Value Unit Entitlements (VUEs) and thus imply a lower One Time Charge (OTC) cost.

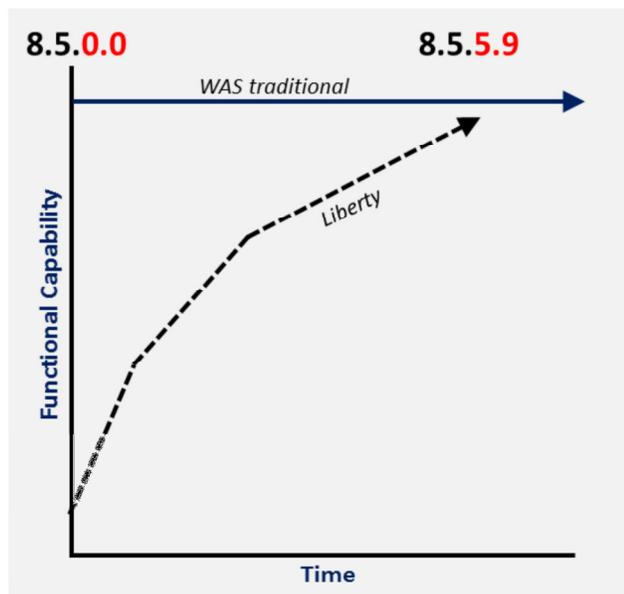
Using Liberty z/OS with zCAP pricing could provide a very cost-effective solution for new Java workloads on z/OS -- even when compared across all platforms.

Potential exists for very attractive cost model for Java on z/OS

Consult with your IBM sales representative for specific details about pricing

There can be a financial incentive to going to Liberty on z/OS based on both the zIIP offload and the way one-time pricing is charged. Check with your IBM sales representative to discuss pricing options.

Differences in the Management Models



- WAS traditional management model is mature and functionally stable
- Initial Liberty management model was lacking in functional capabilities
- Investment focus has shifted to Liberty and its management model
- Investment also being made in dev/ops flows for Liberty

The models are different, so a direct comparison is difficult. Key point: Liberty has advanced considerably since 8.5.0.0 and management model is far more feature-rich than it was at first.

12

With respect to the management of the runtime, the story here is again one of a rather wide gap initially, but the gap closing as time went on and Liberty was developed further.

Note: this is a little more difficult to represent as a one-to-one comparison because the management models are different. So the graph above is meant to represent a high-level illustration of functional capability. It is not a direct comparison of feature-for-feature and function-for-function.

The WAS traditional model is based on a “Deployment Manager” (special purpose application server) hosting the administrative function, with that Deployment Manager “owning” (managing) all the nodes and servers in the “cell.” All administrative functions are performed through the Admin Console (GUI interface) or WSADMIN (the scripting interface). That includes configuration changes and application deployments. That administrative model works well. It is mature and very stable.

Liberty, on the other hand, has a different management model design. It is based on “collectives,” which is a collection of Liberty servers arranged into a “controller” and “member” arrangement. A server designated as a controller assumes the role of being a management interface point. Servers designated as members are managed by the controller. The management model is more flexible than the WAS traditional model in that servers can come and go from collectives far more easily than they could from the WAS traditional framework. The Liberty collectives framework also scales to much higher numbers of servers. Whether that’s important to you is a function of how large a topology you have or are planning. (WAS traditional had a scale limit of about 700 servers, while Liberty collectives can scale to the 10,000+ number.)

Initially the Liberty management model was very rudimentary. But as the releases of Liberty have come out, the model has matured more and more. At this point in time the development focus is on Liberty and the Liberty management model. The WAS traditional management model works well, but has far less new function focus than Liberty enjoys.

When we Speak of “Operational Considerations,” we Refer to the Following ...

- Product installation
- Product maintenance updates
- Runtime creation
- Runtime provisioning (dev/ops, cloud, containers)
- Runtime configuration changes
- Runtime updates to new versions
- Application deployments / updates
- Backup and restore
- Capacity and performance monitoring
- Troubleshooting and problem tracking
- Usage monitoring and chargeback
- System automation routines
- ... and other activities



These activities are, to varying degrees, important to the business
The discussion here is how deeply invested you are in tools and processes for these activities *today*, and how easily can you move to a Liberty runtime platform *tomorrow*

13

We have a section on “operational considerations,” which is a broad topic covering many disciplines. When we speak of that topic -- particularly with respect to z/OS -- we are referring to the types of things listed on the chart.

For Liberty, some of those things are similar in approach and execution to WAS traditional. For example, the “product installation” bullet implies IBM Installation Manager (IM) for both, and except for disk space and some install syntax differences, the installation process is largely the same.

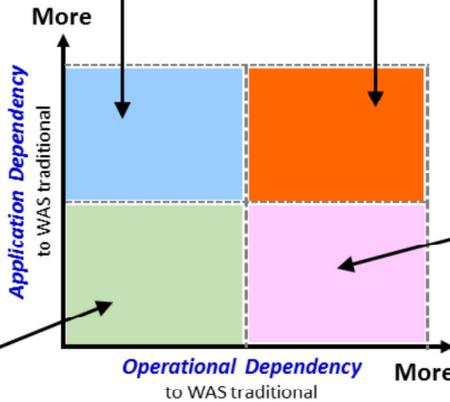
For other things -- for example, backup and restore -- Liberty is actually a simpler model because of the simpler configuration model. Other things -- for example, usage monitoring and chargeback -- may for WAS traditional today be done with SMF 120.9 records, and Liberty does not have those.

When considering a move to Liberty from WAS traditional, the evaluation and discussion centers around the operational processes you have today, how deeply you are invested in them from a business perspective, and whether a transition to Liberty can be done within the context of your existing process and skills.

A High-Level Framework for Evaluating Existing Workload for Move to Liberty

High Application / Low Operational

- Applications have dependencies
- Little or no script investment
- Investment in Liberty skills in plan
- **Consider Liberty for new workloads**
- **Investigate application re-engineering for cases where move to Liberty is justified**



High Application / High Operational

- Applications not easily moved
- Vendor application dependencies
- Investment in WADMIN scripts
- Deep skills in WAS traditional Admin
- **Maintain WAS traditional**
- **Consider Liberty for new workloads**

Low Application / Low Operational

- Little or no application dependencies
- Little or no script or skill investment
- **Consider Liberty for existing and new workloads**

Low Application / High Operational

- Little or no application dependencies
- Investment in WSADMIN scripts
- Deep skills in WAS traditional Admin
- **Maintain WAS traditional for existing**
- **Consider Liberty for new workloads**

14

This chart is providing one of those oft-used “quadrant” structures to help you think about where you operate today with WAS traditional. The quadrant has two axis:

- **Application dependency** (vertical axis) -- this is a relative measure of how tied your application are to the WAS traditional API model, and to the WAS traditional CR/SR model on z/OS. Lower on the axis means your applications are less dependent, and thus moving the application to Liberty would incur less resistance. Higher on the axis means the applications have some dependencies, and thus moving to Liberty would entail inspection of the applications and potentially some re-engineering.
- **Operational dependency** (horizontal axis) -- this is a relative measure of how tied you are to the operational practices you have in place for WAS traditional. The further to the left you fall the less dependency you have; the further to the right the more dependency you have. For example, if you have built important processes around the SMF 120.9 record, then it implies you fall further to the right on the scale (because Liberty does not cut SMF 120.9 records). Similarly, if you have a large inventory of WSADMIN scripts to automate WAS administration, then it suggests falling further to the right as the WSADMIN object model is not part of Liberty.

A quadrant is formed by dividing both axis in half. Thus:

- **Lower-left** (low application / low operational) -- based on your applications and operational practices, a move to Liberty should be relatively easy. Therefore, Liberty can be considered for both new and existing workloads.
- **Upper-right** (high application / high operational) -- you have a high degree of dependency on WAS traditional for both applications and operations. If you fall in this quadrant then a move to Liberty may be a bit more involved. You may wish to consider leaving existing workload on WAS traditional, and perhaps consider Liberty for new workloads if having two runtime models is acceptable.
- **Upper-left** (high application / lower operational) -- operationally you could move to Liberty, but the applications have some dependencies, so that implies some inspection and potentially some rewrite. You would have to look at this more closely to see the nature of the application dependencies and what would be involved to make them capable of moving to Liberty.
- **Lower-right** (low application / high operational) -- the applications can move, but you have a lot invested in the operational aspects of WAS traditional. This suggests maintaining WAS traditional and perhaps looking to Liberty for new workloads where the operational practices can be accommodated.

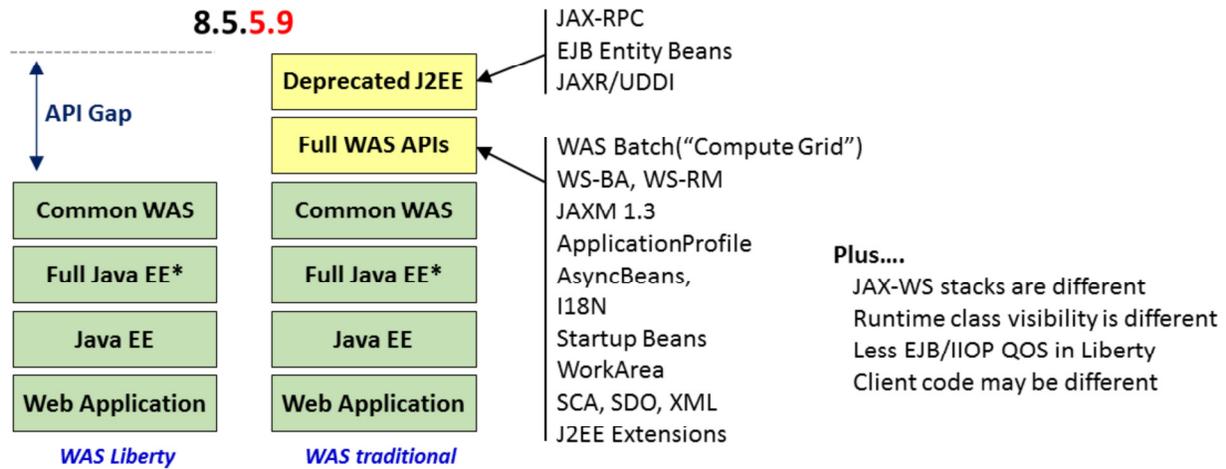
The quadrant is simply a tool to consider where you operate, and what options you may have from there.

Application Considerations

15

In this section we'll take a closer look at the considerations around moving applications from WAS traditional to Liberty.

More on the API Gap between Liberty and WAS traditional



An application that makes use of the APIs in the "API Gap" list may need re-engineering to move to Liberty. If the application uses APIs that are common across WAS traditional and Liberty, then it may move easily.

16 * For Liberty: partial Java EE 6, full Java EE 7. For WAS traditional: Full Java EE6, Full Java EE 7 in beta

Earlier we showed you this picture with the "API Gap" for 8.5.5.9 between Liberty and WAS traditional. Earlier we did not provide detail, but here we do. The focus is on the two yellow boxes -- "Full WAS APIs" and "Deprecated J2EE."

- **Full WAS APIs** -- these are APIs above the Java EE specification. They exist in WAS traditional but not in Liberty, and if an application makes use of these APIs the application would not function properly if moved to Liberty.
- **Deprecated J2EE** -- these are APIs that have since been deprecated by the standards bodies but are not yet removed. They are in WAS traditional but not Liberty. It's possible you have applications that still make use of the deprecated APIs. The same issue arises here as with the full WAS APIs -- if an application is dependent on the APIs being present, then moving the application to Liberty will result in errors.

Note: in a few charts we will point you to a tool that will scan application binaries and produce a report on the suitability of moving an application to Liberty. It looks at the APIs used and reports where compatibility with Liberty is present and where issues arise.

Considerations Beyond the APIs



Time horizon for application -----

An application with a relatively short life horizon may not be worth moving. Better to leave it where it is and focus energy on higher-value applications

Value of application investment -----

An application with a longer expected life span may require re-engineering investment to run properly on Liberty. Does the proposed investment yield positive return for the business?

Potential deployment environments -----

For new applications, do you expect to deploy the application into environments such as IaaS cloud, or Bluemix, or container environments such as Docker? That may imply targeting Liberty as that runtime is better prepared for operations in those environments.

17

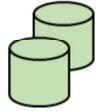
With respect to applications, there are considerations beyond the APIs. They are:

- **Time horizon** -- if an application is currently hosted in WAS traditional, and you have plans to sunset the application into the near future, then going through the effort to move the application may not be worth it. Better in that case to leave the application where it is and let it sunset. Applications with longer time horizons may be worth any effort to move to Liberty, based on your evaluation of the overall value to the business of moving the application.
- **Value of further investment** -- let's say you have an application that has a longer time horizon, but moving it would imply some rewrite. The question here is whether the application's value to the business warrants the further investment in the application. Applications that are central to the business mission would very likely be worth the effort (provided the overall evaluation suggests a move to Liberty); applications that are less central may not.
- **Deployment environments** -- this consideration is interesting in light of some of the new developments around cloud and container environments. If the vision you have for an application is that it be highly-flexible with regard to deploying into IaaS cloud, or Bluemix, or Docker, then you may wish to take a good look at Liberty because it tends to lend itself more easily to those environments. (WAS traditional can be deployed into those environments. It's just that Liberty's simpler configuration model makes deployment even easier.)

So the exercise is this -- take a look at the application API usage, and take a look at these broader considerations, then determine how well your applications lend themselves to being moved to Liberty. Then, after you've gone through the operational considerations (next section), think about where you fall on that "quadrant" chart. That will help you determine the path you will take when considering WAS traditional and Liberty.

Migration Toolkit for Application Binaries

Your application binaries



Migration Toolkit



Summary report of technology used in application and target environments where application can be deployed

Product Edition	Liberty for Java on IBM Bluemix	Liberty Core	Liberty	WebSphere Application Server V8.5.5				
WebSphere Application Server V8.5.5	✓	✓	✓	✓	✓	✓	✓	✓
WebSphere Application Server V8.5.5	✓	✓	✓	✓	✓	✓	✓	✓
WebSphere Application Server V8.5.5	✓	✓	✓	✓	✓	✓	✓	✓
WebSphere Application Server V8.5.5	✓	✓	✓	✓	✓	✓	✓	✓
WebSphere Application Server V8.5.5	✓	✓	✓	✓	✓	✓	✓	✓

File name	Reference details	Match criteria	Line number
PlantsByWebSphere war\WEB-INF\classes\com\ibm\websphere\sample\jber\war\ShoppingBean.class	Method getShippingCost\$String	java.text.NumberFormat.format	156
	Method getTotalCost\$String	java.text.NumberFormat.format	175

Detailed report by file name, method name and line number

Main wasDev page:

https://developer.ibm.com/wasdev/downloads/#asset/tools-Migration_Toolkit_for_Application_Binaries

Technical Overview:

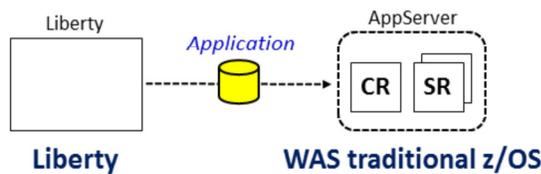
<https://developer.ibm.com/wasdev/docs/migration-toolkit-application-binaries-tech/>

Updates page:

<https://developer.ibm.com/wasdev/blog/2015/03/13/announcing-websphere-liberty-migration-tools-updates/>

The “Migration Toolkit for Application Binaries” is a utility that scans your applications and reports on what is seen in the application and what aspects of the application may require update to run on Liberty. The tool is free of charge and is downloadable from the URL shown in the chart.

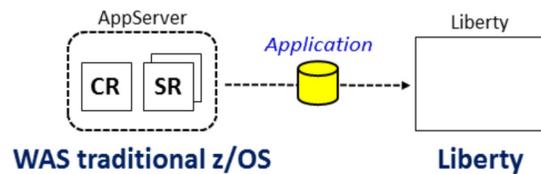
Final Points on Application Considerations



This application path is relatively seamless

Notes:

- Liberty has Java EE 7, WAS traditional is in beta with that technology. An application that makes specific use of Java EE 7 (ex: JSR 352 Java Batch) would not work on WAS traditional if Java EE 7 not present.
- Liberty is a single JVM environment, where WAS traditional on z/OS has the potential for multiple application JVMs (SRs). Applications that create singletons *may* experience issues.



This path can work, but a bit more care needed

Notes:

- If application uses APIs in the “API Gap” illustrated earlier, the application would require updating.
- If the application is relying on session replication between SRs, that aspect of the application would need inspection and persistence (if needed) configured in Liberty using a database or caching layer.

19

As we close out this section, let’s touch on a final few points about application compatibility between WAS traditional and Liberty, particularly on z/OS:

- When moving an application from Liberty to WAS traditional z/OS, the move should be relatively seamless because the “API Gap” we spoke of earlier showed Liberty being on the lower end of the gap. But that’s not to say the move is certain to be seamless. Two things to keep in mind -- (1) Liberty now has Java EE 7 and WAS traditional does not yet have that. So an application written to take advantage of specific Java EE 7 APIs will find those APIs are not present in WAS traditional (which is still at Java EE 6). And (2), WAS traditional z/OS operates with the multi-JVM CR/SR model, and if you operate the WAS traditional z/OS server with 2 or more servant regions you are creating what is in effect a cluster of servers. If the application creates singletons and expects to always be in a single JVM, then you may experience problems with a multi-SR server on z/OS.
- When moving from WAS traditional z/OS to Liberty, a bit more care needs to be taken. First, there is the issue of the “API Gap,” which the “Migration Toolkit for Application Binaries” can help you with. That tool will scan your application and report on what technology is in use and what methods to inspect more closely for compatibility. Also, if your application runs in a multi-SR WAS traditional z/OS region today and is making use of “session replication” between the SR regions, then moving to Liberty may introduce differences in behavior. Liberty has the ability to persist session information, but it does not do so with the same “session replication” mechanism used by WAS traditional. If session information is created and persistence is needed, that would need to be configured into the Liberty server and tested to make sure the application operates as expected.

Operational Considerations

Broad Topic with Many Disciplines

Install and Maintain

- Product installations
- Maintenance updates
- Create runtimes
- Migrate to new versions
- Backup and restore

Change Management

- Identify change requirements
- Implement and test
- Promote up to production
- Track progress, effect back-outs



Plan, Monitor, Troubleshoot

- Capacity planning
- Performance planning
- Monitoring usage, resources, performance
- Analyze problems, track resolution

Develop, Deploy, and Test

- Application design and develop
- Deployment automation
- Deployment target provisioning
- Test planning and automation
- Other Dev/Ops activities

Other?

- Any other operational activities not on the lists above

21

Earlier we introduced the topic of “Operational Considerations” and said it was a broad topic with many disciplines. Here we are re-iterating that point by organizing some of the tasks into logical groups.

The point of this chart is to start the thought process around the present-day operational processes and how operating Liberty would apply. As mentioned earlier, some of the tasks are very similar; some are different.

Comparison Grids to Follow



Operational attribute or task	Runtime, Liberty or WAS traditional		
	Liberty	WAS Traditional	
Installation mechanism	Installation Manager	Installation Manager	●
Install size	200MB, granular control	2GB	●
Memory size	Lower (~50MB min/server)	Higher (~1GB/server)	●
Operating systems	Windows, Linux, AIX, HP, Solaris, IBMi and z/OS	Windows, Linux, AIX, HP, Solaris, IBMi and z/OS	●
z/OS operational mode	UNIX process or STC	STC	●
Virtual, cloud, containers	VMs, IaaS, PaaS, Docker	VMs, IaaS, Docker	●
Java SE support	Any 1.6, 7.x or 8.x	IBM only 1.6, 7.x, 8.x coming	●
Java EE support	Partial 6.0, full 7.0	Full 6.0, full 7.0 in beta	●
Fix Packs and iFixes	Yes	Yes	●
New features and functions	Frequent with continuous delivery	Major version updates only	●

Green = same
Yellow = delta

By walking through the operational attributes it has the potential to stimulate thinking and discussion about your current environment compared to Liberty. We encourage the discussion. The objective is a clear understanding of the similarities and differences.

22

What follows is a set of charts that show a table comparing Liberty with WAS traditional across a set of operational attributes or tasks. The circles off to the right indicate similarity (green dot) or some difference (yellow dot).

Again, the objective here is to provide a discussion framework in which the very broad topic of operational considerations can be discussed.

General Product Considerations

	Liberty	WAS traditional	
Installation mechanism	Installation Manager	Installation Manager	●
Install size	200MB, granular control	2GB	●
Memory size	Lower (~50MB min/server)	Higher (~1GB/server)	●
Operating systems	Windows, Linux, AIX, HP, Solaris, IBMi and z/OS	Windows, Linux, AIX, HP, Solaris, IBMi and z/OS	●
z/OS operational mode	UNIX process or STC	STC	●
Virtual, cloud, containers	VMs, IaaS, PaaS, Docker	VMs, IaaS, Docker	●
Java SE support	Any 1.6, 7.x or 8.x	IBM only 1.6, 7.x, 8.x coming	●
Java EE support	Partial 6.0, full 7.0	Full 6.0, full 7.0 in beta	●
Fix Packs and iFixes	Yes	Yes	●
New features and functions	Frequent with continuous delivery	Major version updates only	●

23

Here is the first table in a set of such tables that compares operational tasks.

Configuration and Deployment

	Liberty	WAS traditional	
Composable runtime	Yes (via Features)	No	●
Dynamic configuration	Yes	Partial	●
Configuration structure	Relatively simple, flexible location	More complex, defined location	●
Configuration editing	Simple XML updates; admin tools	Admin console; WSADMIN scripting	●
Configuration updates	Simple file-based	XML file deltas via tools	●
Central management	Collectives (no agents)	Cell (with node agents)	●
Central management scale	Very small to 10,000+	Very small to ~700 maximum	●
Central management failover	Yes (controller replica)	No (restart DMGR on other LPAR)	●
Configuration ownership	Each server (no synchronization)	DMGR (central with synchronization)	●
Application deployment	Manual, script, with server package	Admin Console, WSADMIN script	●
Application update	Replace application file	Redeploy through Admin	●
Product update	No migration	Migration tools	●

24

More operational considerations ...

Operational Capabilities

	Liberty	WAS traditional	
HTTP load balancing	Plugin, ODRLIB, any HTTP proxy	Same as Liberty, plus Java ODR	●
HTTP session replication	DB persistence or WXS caching	Same as Liberty, plus DRS	●
Scripting support	Any	WSADMIN (JACL or Jython)	●
Dynamic clusters / auto-scale	Yes	Yes	●
JMX client	Java, REST	WAS Admin Client	●
Monitoring	mBeans, PMI	PMI	●
Fine-grained admin authority	No (single admin role)	Yes	●
JMS providers	Internal, WMQ, 3 rd Party	Internal, WMQ, 3 rd Party	●
Clustered JMS provider	No (use WMQ)	Yes	●
2PC transaction recovery	Yes	Yes	●
Remote EJB calls	Yes	Yes	●
Runtime class visibility	Defined API	Internals are accessible	●
Docker support	Yes (collective support in beta)	Yes	●

25

And still more operational considerations ...

Security Options (1 of 2)

	Liberty	WAS traditional	
Default passwords	No	No	●
Minimal ports opened	Yes	No	●
Secured remote admin	Yes (mandatory)	Yes (but can be turned off)	●
File user registry	Yes (server.xml)	Yes (file based)	●
Federated LDAP or SAF	Yes	Yes	●
OAuth, OpenID, OIDC client	Yes	Yes	●
OIDC server/provider	Yes	No	●
LTPA, SPNEGO tokens	Yes	Yes	●
SAML Web SSO	Yes	Yes	●
SAML Web Services	Yes	Yes	●
User and Group API	Yes	Yes	●
Federated File registry w/ LDAP	Yes	Yes	●

26

The first of two security considerations tables.

Security Options (2 of 2)

	Liberty	WAS traditional	
Auditing	No	Yes	●
Advanced key/cert management	Yes	Yes	●
Local OS registry	No (yes if z/OS = SAF)	Yes	●
JAX-WS support for LTPA	No	Yes	●
JSEHelper API	No	Yes	●

27

More security considerations.

z/OS Integration and Platform Exploitation

	Liberty	WAS traditional	
Multi-JVM (CR/SR)	No	Yes	●
z/OS Connect	Yes	No	●
zWLM	Yes (Service and Report classification)	Same, and work placement by SC	●
WOLA local adapters	Yes (no 2PC yet)	Yes	●
RRS TX coordination	Yes (JDBC only)	Yes	●
SMF request tracking	Yes (HTTP only)	Yes	●
Messages to server job log	Yes	Yes	●
Messages redirect to console	Yes	Yes	●
Hung thread stop and recover	No	Yes	●
Pause/Resume Listeners	No	Yes	●
Dispatch Progress Monitor	Yes (with Health Manager feature)	Yes	●
MODIFY interface	Yes, but limited	Yes	●

28

This chart highlights the degree to which Liberty and WAS traditional on z/OS is capable of taking advantage of the z/OS platform. Both are capable of exploitation, but WAS traditional -- being a product that's been around longer -- has a deeper set of such things. But Liberty has a set as well, and this chart outlines what they are.

Summary of z/OS Operational Considerations



Install and backup/restore are somewhat similar for both

Liberty requires no migration tools to move to new version, WAS traditional does, and the effort to migrate is not trivial

Administrative interfaces are different; scripting interfaces are different

Both are operated as started tasks, so:

- Can use system automation routines
- Can monitor with SMF Type 30

Both are capable of WLM service class and report classification based on matching request URI patterns

WAS traditional has deeper z/OS integration functions, but if that's not something you're making use of, then it's less a factor

29

This chart summarizes the comparison tables we just went through with a specific focus on z/OS.

Performance Considerations

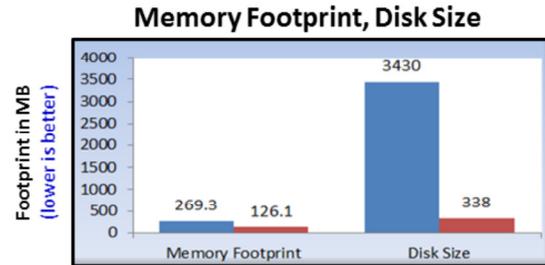
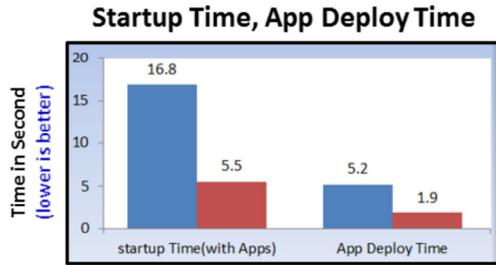
30

In this section we will walk through a set of charts that compares the performance of WAS traditional with Liberty.

Note: these performance measurements were conducted in a controlled environment under very specific conditions. Your results may vary. These results are *not* a promise of performance results.

Startup Time, App Deploy Time, and Memory/Disk Footprint

■ WAS traditional ■ Liberty



Startup time for Liberty 32% the time of WAS traditional

Application deployment time 36% the time of WAS traditional

Memory footprint for Liberty 47% that of WAS traditional

Disk size for Liberty 10% that of WAS traditional

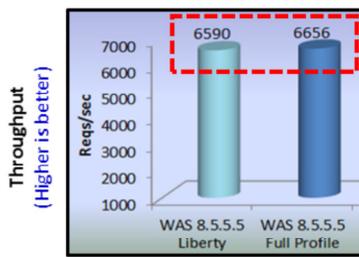
31 Performance results derived in a controlled environment under specific conditions. Your results may vary depending on a number of factors.

This chart shows four comparison in two graphics: startup time, application deployment time, memory footprint, and disk size.

- **Startup time** (left side of chart, left bar graph) -- here we see that Liberty outperforms WAS traditional by quite a margin. The time for Liberty (5.5 seconds) is 32% of the time of WAS traditional (16.8 seconds). Or stated another way, Liberty is 68% faster in server startup than WAS traditional.
- **Application deployment time** (left side of chart, right bar graph) -- Liberty’s time to deploy an application was only 36% the time required by WAS traditional.
- **Memory footprint** (right side of chart, left bar graph) -- the bars in this graph are relatively small because the left axis is determined by the “disk size” bars in the chart. Still, if we look at the numbers we see that the memory footprint of Liberty (126.1 MB) is half the size (47%) of the 269.3 MB consumed by WAS traditional.
- **Disk size** (right side of chart, right bar graph) -- here the difference is quite stark. The WAS traditional disk footprint was 3,430 MB compared to 338 MB for Liberty. Liberty’s size is a mere 10% that of WAS traditional.

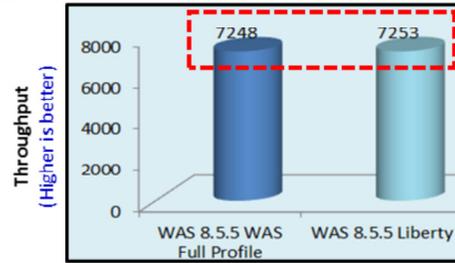
Throughput on Distributed Platforms ... z/OS on Next Chart

DayTrader 3 EJB, Hotspot JDK 8_31



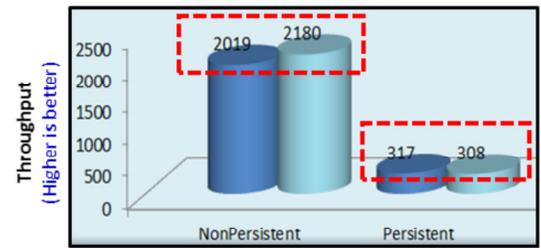
Liberty 99% of WAS traditional

Web Services SOABench



Liberty 100% of WAS traditional

Messaging, JMS Prims 10k/10k



Liberty 108% of WAS traditional Liberty 97% of WAS traditional

Effectively the same throughput for WAS traditional and Liberty on the distributed platforms for DayTrader (EJB), SOABench (SOAP/WSDL), and Messaging (JMS)

No loss of throughput moving from WAS traditional to Liberty on distributed

32 Performance results derived in a controlled environment under specific conditions. Your results may vary depending on a number of factors.

This chart shows a set of throughput measurements for different applications.

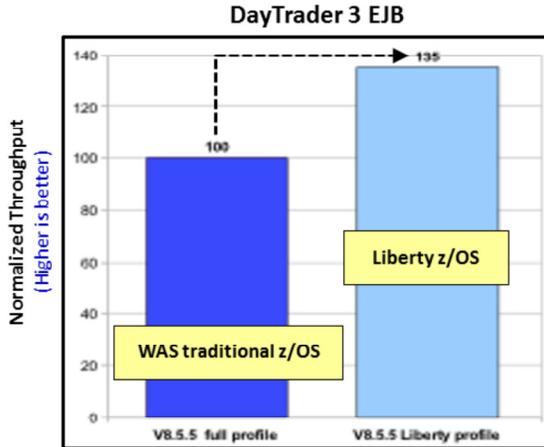
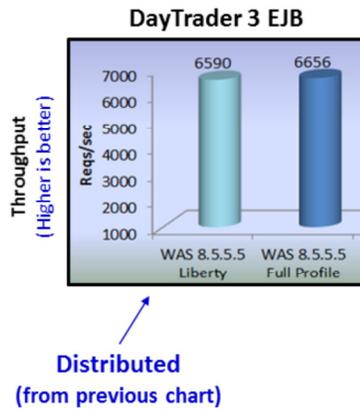
Note: this is for *distributed* platforms. This is *not* z/OS. The z/OS throughput chart comes next.

The left-most bar chart is for the DayTrader benchmark application, which is an EJB application with DB2-backed data persistence. The middle chart is the SOABench application, which is a SOAP/WSDL web services benchmark application. And the right-most chart is using the JMS Prims application to measure messaging throughput.

What we see is the throughput for Liberty vs. WAS traditional is very close to one another. The largest difference was seen in the JMS throughput test for non-persistent messaging, where Liberty achieved 108% the throughput of WAS traditional (called "Full Profile" on these charts).

The message here is that moving an application to Liberty does *not* imply any loss of throughput.

DayTrader 3 on z/OS Shows Liberty Outperforming WAS traditional



Note: the throughput axis for z/OS shows results normalized ... that is, the WAS traditional throughput achieved was set to "100" and the Liberty throughput achieved was proportional to the baseline 100 value.

Actual throughput is a function of many factors, including processor speed, memory, cache size, and I/O. The tests performed here were not meant to compare distributed directly with z/OS. Rather, the point here is that on z/OS, Liberty outperformed WAS traditional. On distributed, the two were roughly equivalent.

This is because Liberty's single-JVM model is more efficient than WAS traditional's multi-JVM model with controller and servant regions

33 Performance results derived in a controlled environment under specific conditions. Your results may vary depending on a number of factors.

This chart is showing the throughput achieved for the DayTrader application with Liberty and WAS traditional on z/OS.

Notes:

- The distributed platform graph on the left is just for reference. That picture comes from the previous chart.
- The performance numbers for the z/OS test (right side of the chart) are *normalized*, so a direct comparison of the vertical axis of that picture with the picture on the left should not be attempted. What normalized means is the test was performed on z/OS WAS traditional, and the throughput achieved was equated with the number 100. The throughput for Liberty (which was higher) was made proportional to the "100" set for WAS traditional. Therefore, the left axis of the picture on the right is not the nominal throughput achieved, but a normalized representation of the throughput achieved, Liberty vs. WAS traditional.

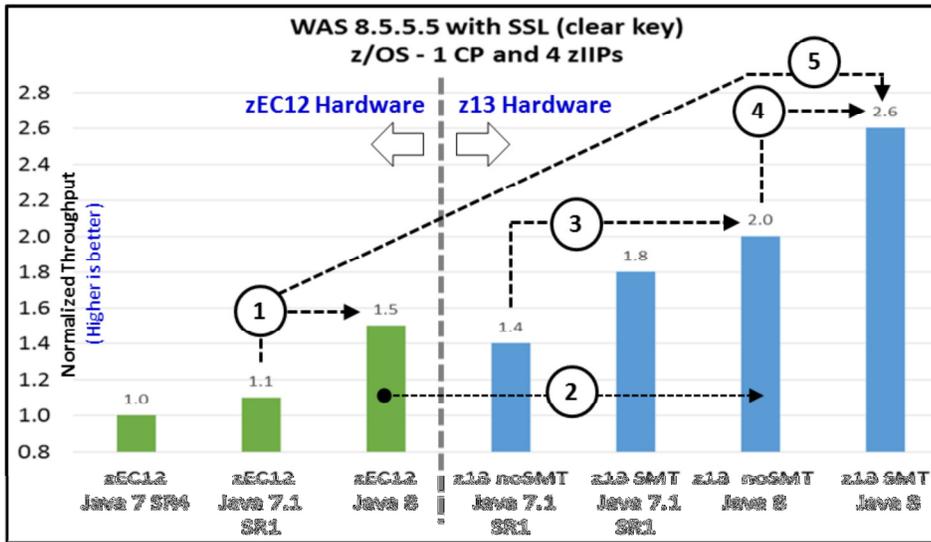
The reason Liberty z/OS had higher throughput is because of the design of Liberty z/OS compared to the design of WAS traditional on z/OS. On z/OS, WAS traditional has the multi-JVM structure with controllers and servants. That implies a longer code-path through the CR, queuing with WLM, and finally dispatching to the SR. Liberty z/OS -- a single JVM model -- has a shorter code path to achieve dispatching of the work to a thread for execution.

WAS traditional on the distributed platforms is also a single JVM model, which is why the WAS traditional vs. Liberty on those platforms showed comparable results.

WAS traditional z/OS and its CR/SR model provides some useful benefits with regard to queuing and dispatching to multiple JVMs. But that model comes with a "cost" in terms of throughput, and the picture on the right-side of the chart above illustrates the throughput "cost".

The Value of z13 Hardware, Java 8 and SMT Exploitation

SSL-Enabled DayTrader 3.0 with Liberty z/OS measured



- 1. Java 8 on zEC12**
36% improvement -- improved JVM/JIT (1.5/1.1 = 1.36)
- 2. Value of z13**
33% improvement -- faster HW, greater instruction exploitation by SDK (2.0/1.5 = 1.33)
- 3. Java 8 on z13**
43% improvement -- improved JVM/JIT, greater instruction exploitation by SDK (2.0/1.4 = 1.43)
- 4. Value of SMT**
30% improvement -- exploitation of SMT by Java 8 SDK (2.6/2.0 = 1.30)
- 5. Overall**
Java level, HW level, and SMT. We see a 136% improvement (2.6/1.1 = 2.36)

34 Performance results derived in a controlled environment under specific conditions. Your results may vary depending on a number of factors.

This chart is showing a set of performance result comparing several different things, so a few moments explaining the chart is needed:

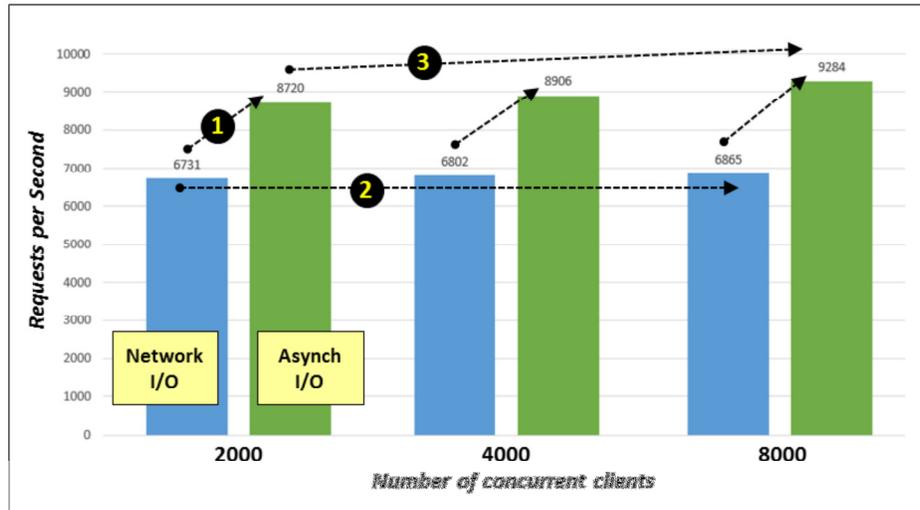
- The chart is divided with the bold dashed vertical line; on the left is a set of performance runs on the IBM zEC12 hardware, and on the right is a set of performance runs on the IBM z13. The z13 hardware is newer, faster, and has instructions available for exploitation, which Java 8 takes advantage of.
- Multiple levels of Java are being compared here. The key comparison is between Java 7 and Java 8. The reason that is key is because Java 8 has two major areas of improvement over Java 7 -- (1) more efficient JVM and JIT processing, and (2) direct exploitation of new instructions on the z13 hardware, which makes JIT processing even *more* efficient.
- z13 SMT (simultaneous multi-threading) capability, which enhances performance when the Java SDK takes advantage of the SMT feature present on the hardware.

The chart is augmented with arrows and numbered blocks to draw your attention to the performance improvements and where they come from:

1. The comparison here is between Java 7 and Java 8 on the zEC12 machine. So the hardware stays the same, but the Java SDK level changes. This is the value of Java 8. This represents the general performance improvements made to the JVM and JIT processing. Since this is on the zEC12, the newer instructions of the z13 are not present, so that variable is not part of this performance benefit illustration. We see a 36% improvement.
2. The comparison here is between Java8 on the zEC12 versus Java 8 on the z13. So the Java SDK level stays the same but the hardware changes. Two things are going on here: (a) the Java 8 SDK recognizes it is on a z13 and it knows there are instructions on the chip that aid the JIT processing, so it takes advantage of those instructions automatically; and (b) the z13 hardware is faster and with larger cache, so anything that shows up on the z13 will tend to run faster. We see a 33% improvement here.
3. The comparison here is Java 7 vs. Java 8 on the z13. This represents the general improvements in the Java 8 JVM and JIT processing, as well as the specific exploitation of the z13 instructions by the Java 8 JIT. 43% improvement.
4. The comparison here is no SMT vs. SMT where the Java level is the same (Java 8) and the hardware is the same (z13). We see a 30% improvement.
5. Finally, if we combine all the changes of comparisons 1 through 4 we derive an “overall” performance benefit. This includes the benefits of the Java improvements, the hardware improvements, and the SMT processing. Here we see a 136% improvement, which is a more than doubling of the throughput.

Asynchronous v. Network I/O in Liberty z/OS **16.0.0.3**

Asynchronous I/O performance benefits are most significant with larger numbers of concurrent clients:



Three key points:

- 1. Asynch I/O > Network I/O**
In all three concurrent user scenarios, Asynch I/O was 30% or more greater throughput
2000 concurrent = +30%
4000 concurrent = +31%
8000 concurrent = +35%
- 2. Network I/O mostly flat**
As concurrent users scale up, we see a relatively flat line for Network I/O (~1.9% improvement 2K to 8K)
- 3. Asynch I/O trends up**
As concurrent users scale up, we see a trend upwards with Asynch I/O (~6.5 improvement 2K to 8K)

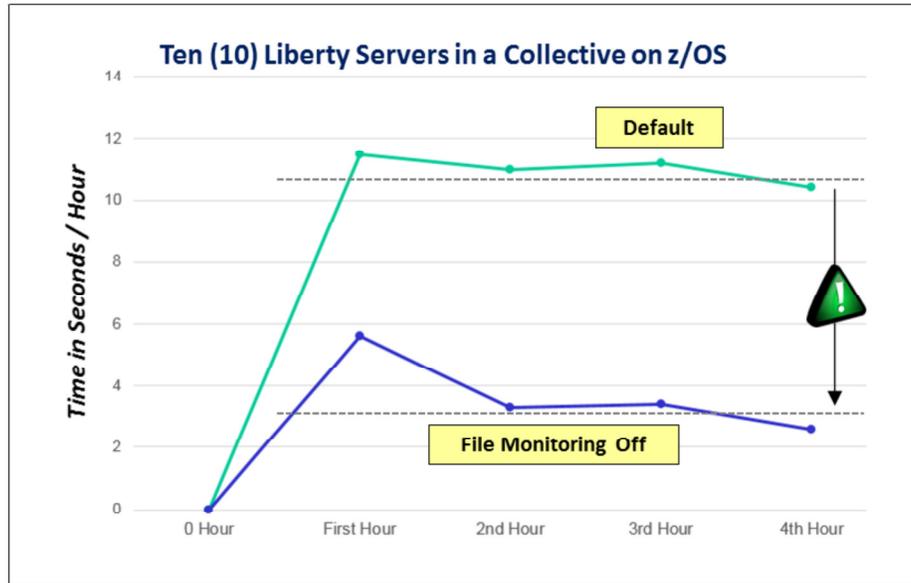
35 Performance results derived in a controlled environment under specific conditions. Your results may vary depending on a number of factors.

One of the new functions added to Liberty z/OS in 16.0.0.3 was "Asynchronous I/O." This function is particularly useful when large numbers of concurrent users are engaging with a server. To measure the benefits of this we compared Liberty z/OS against Liberty z/OS -- the same 16.0.0.3 level -- with the default "Network I/O" vs. the new Asynchronous I/O function enabled. We scaled up from 2,000 concurrent users to 4,000, then 8,000.

The chart has three key points we wish to draw your attention to:

1. There is an increase in throughput by switching on Asynchronous I/O. The percent increase is a function of the number of concurrent users. We saw 30%+ improvement across the three levels of concurrent users.
2. Notice that the Network I/O function does not really scale up as we increase the concurrent users. It's true that it did not degrade either, but the throughput was essentially flat.
3. However, we did see a slight trend upwards in the throughput when using Asynch I/O, about a 6.5% increase from 2,000 concurrent users to 8,000 concurrent users. That shows that not only is this function beneficial when compared to Network I/O, but it also works well as we scale up the concurrent users.

Idle CPU time in Liberty on z/OS



This chart is showing the CPU time for 10 Liberty z/OS servers in a Collective as they idle

The Y Axis shows the CPU time in seconds for all 10 servers at each hour mark (the X Axis).

When configured with the default file monitoring setting, the environment averaged about 11 CPU seconds per hour for the 10 servers.

When file monitoring is turned off, the CPU time dropped to about 3 seconds total per hour for the 10 servers.

IBM Performance results derived in a controlled test environment under specific conditions. Your results may vary depending on a number of factors.

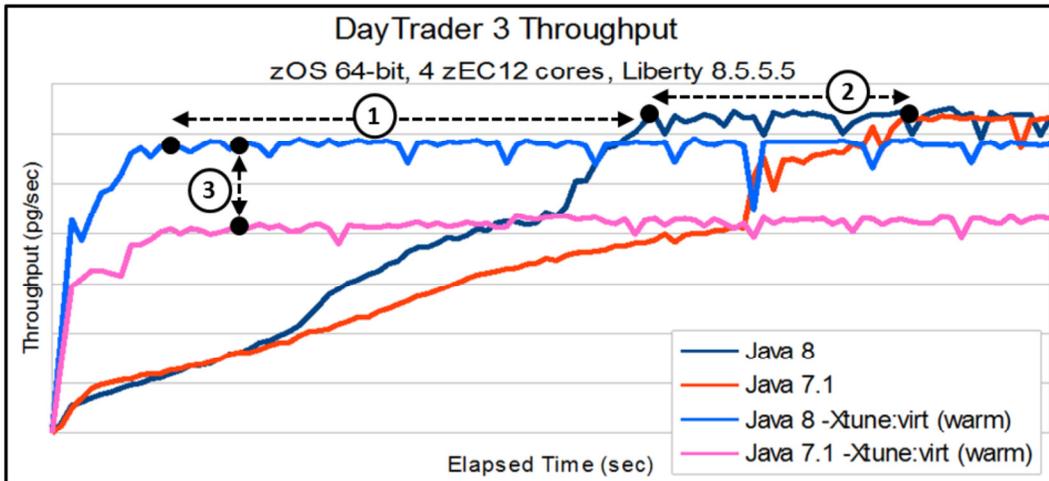
This chart is showing the CPU time for ten Liberty servers at idle when file monitoring is one compared to when file monitoring is turned off.

Note: file monitoring is what allows Liberty to detect when a configuration or application has been changed. By default the file monitoring is set to occur every $\frac{1}{2}$ second, which is wonderful for a development environment, but it does tend to add up over time in terms of CPU used. We anticipate most production environments will turn file monitoring off as a means of better controlling updates to the environment. Doing this also implies less CPU because the servers are polling against the file system looking for changes.

This chart has two lines -- the upper line represents 10 Liberty z/OS servers with the default 500ms polling interval for file monitoring; the lower line representing the same 10 Liberty z/OS servers with the file monitoring disabled. The servers ran for four hours, and we captured the accumulated CPU for each hour at the hour marker.

The key point is what's illustrated by the two horizontal dashed lines and the little green triangle showing the delta between an approximate 11 seconds/hour CPU time with file monitoring vs. approximately 3 seconds/hour when file monitoring is turned off.

z/OS Liberty Ramp-up with IBM Java 8



- ① **Ramp-up improvement due to -Xtune:virt**
Less elapsed time to steady state when -Xtune:virt used
- ② **Ramp-up improvement Java 8 vs. Java 7**
Java 8 achieved steady state in less elapsed time than Java 7

③ **Steady-state throughput improvement Java 8 over Java 7 with -Xtune:virt**
Once steady state is achieved, Java 8 results in better throughput

37 Performance results derived in a controlled environment under specific conditions. Your results may vary depending on a number of factors.

Here is another chart where several things are going on, so we've added arrows and numbers to draw your eye to the key performance comparisons we wish to highlight:

1. This compares the elapsed time difference between when Java 8 with -Xtune:virt set and Java 8 where that property is not set. The starting "dot" is where Java 8 with Xtune:virt reaches steady-state, and the second "dot" is where the Java 8 without Xtune:virt reaches steady state. The startup time is improved by approximately 88%.
2. This compares the elapsed time difference between when Java 8 without -Xtune:virt reaches steady-state and when Java 7.1 without -Xtune:virt reaches steady-state. That's about a 22% improvement.
3. This compares the throughput difference between Java 7.1 and Java 8 on a zEC12 with -Xtune:virt set. We see the general Java 8 improvements we saw on the previous chart.

Startup footprint : WAS traditional ND on z/OS vs. Liberty z/OS

WAS traditional Network Deployment on zEC12				Liberty Collectives on zEC12			
Process Name	CPU Time (seconds)	Elapsed Time (seconds)	Memory (MB)	Process Name	CPU Time (seconds)	Elapsed Time (seconds)	Memory (MB)
DMGR CR	15.96	32	306.4	Controller	9.62	2.3	153
DMGR SR	20.01	13	398.0	Member1	5.96	1.7	138
Node Agent	11.39	72	224.0	Member2	5.14	1.9	141
Member1 CR	10.30	19	239.2				
Member1 SR	7.58	7	256.4				
Member2 CR	10.20	19	241.6				
Member2 SR	7.56	7	259.6				
Total	83	169	1925.2	Total	20.72	5.9	432

Liberty involves fewer processes to create a two-member cluster, and the design of Liberty provides a smaller footprint and faster startup. The results bear this out.

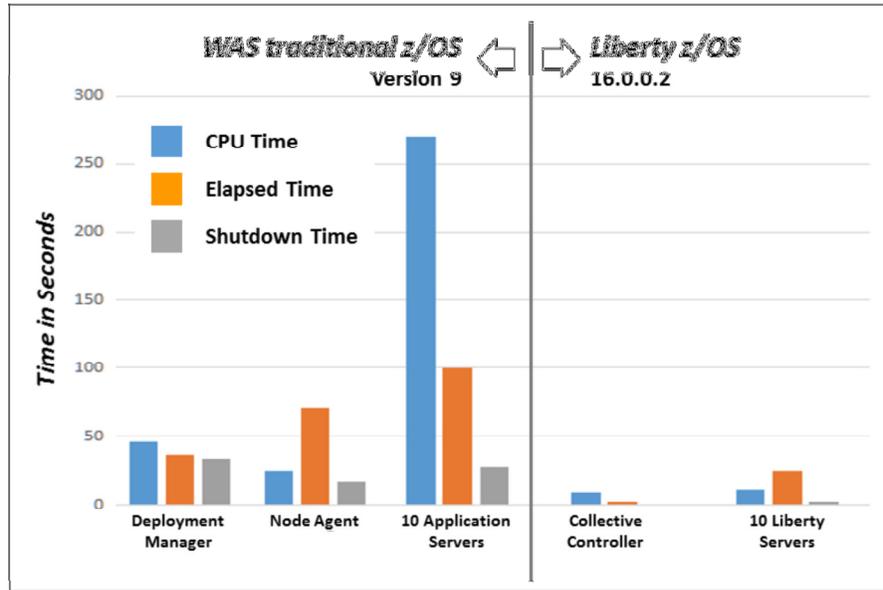
38 Performance results derived in a controlled environment under specific conditions. Your results may vary depending on a number of factors.

This table is comparing the startup footprint of a minimal WAS traditional z/OS ND cell against a minimal Liberty z/OS collective. The comparison is not apples-to-apples because the minimal WAS z/OS ND cell has 7 address spaces while the minimal Liberty z/OS collective has just three. But therein lies the benefit -- Liberty z/OS, but it's single JVM model and the agent-less management model provides an opportunity to reduce the address space, memory, and CPU footprint of the server topology.

The first and most direct comparison is the "total" row at the bottom. There we see that the WAS z/OS ND cell took 83 CPU seconds, 169 elapsed seconds, and 1925MB of memory; the Liberty collective 21 seconds, 6 seconds, and 432MB.

If you wished to make a more granular comparison, then add the CR and SR values for an ND server and compare against the Liberty member; or the DMGR CR and SR numbers and compare against the Liberty controller. The WAS z/OS ND Node Agent has no direct comparison against Liberty since the Liberty collective model is agentless.

Startup and Shutdown Times: WAS traditional vs. Liberty



Start-up and shutdown of 10 servers in a Liberty Collective is significantly faster and more efficient.

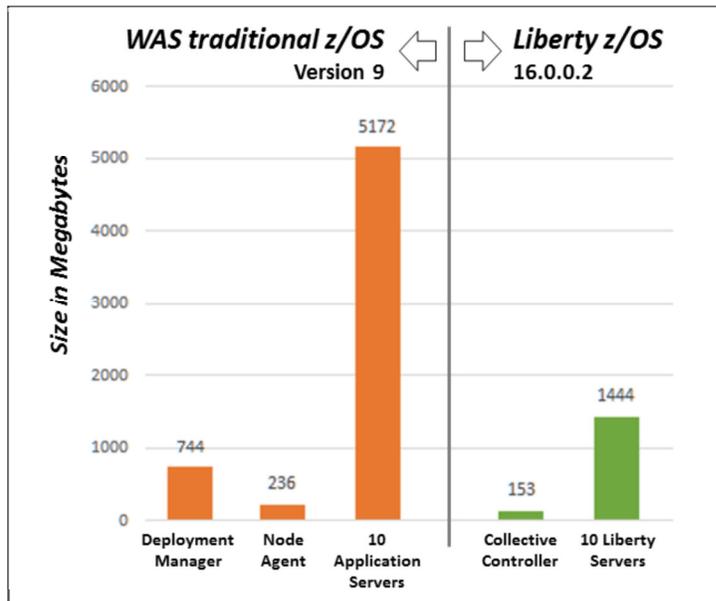
Performance results derived in a controlled environment under specific conditions. Your results may vary depending on a number of factors.

Here we're comparing WAS traditional against Liberty with three measurements: CPU time to initialize the servers, elapsed time to initialize the servers, and elapsed time to shut down the servers. The gray vertical line separates WAS traditional (left side) from Liberty (right side).

WAS traditional has a different management model from Liberty. To create a roughly equivalent setup, we had a WAS traditional Network Deployment configuration with a Deployment Manager, a Node Agent, and 10 application servers. On the Liberty side we had a Collective Controller and 10 Liberty servers.

As you can see, there's a striking difference between the left side of the chart and the right. That's the nature of Liberty and its lighter-weight model. It's also the nature of Liberty with its single JVM model vs. the multi-JVM model of WAS traditional. The left side of this chart implied 23 address spaces --two for the Deployment Manager, one for the Node Agent, and 10 x 2 = 20 for the ten application servers. The Liberty side of the chart implied 11 address spaces -- one for the Collective Controller, and one each for the 10 servers.

Memory Footprint: WAS traditional vs. Liberty



Memory footprint for 10 Liberty servers is almost 5 times less compare to 10 WAS traditional servers.

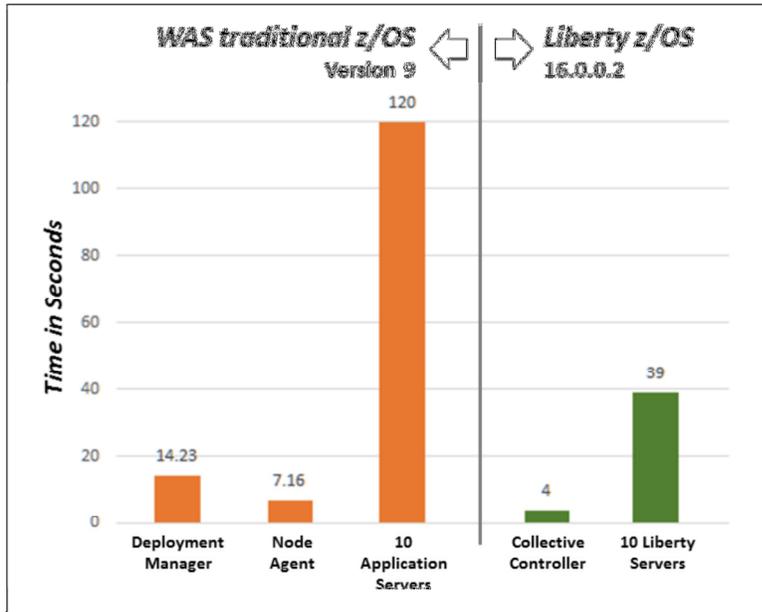
48 Performance results derived in a controlled test environment under specific conditions. Your results may vary depending on a number of factors.

Another WAS traditional vs. Liberty chart, this time comparing the memory footprint of the same setup as the previous chart:

- **WAS traditional** -- Deployment Manager (2 address spaces), Node Agent (1 address space), and ten servers with a controller and servant for each (20 address spaces).
- **Liberty** -- Collective Controller (1 address space), and ten servers (10 address spaces).

The difference is due to the fact WAS traditional tends to load the full Java EE container environment while Liberty does not. The default heaps are roughly the same, but the expansion of the heaps to accommodate the loading of the function implies WAS traditional takes up a larger footprint than does Liberty.

Idle CPU Time: WAS traditional vs. Liberty



Idle CPU time with 10 Liberty servers is approximately 3 times less than WAS traditional servers. The time shown is average per hour.

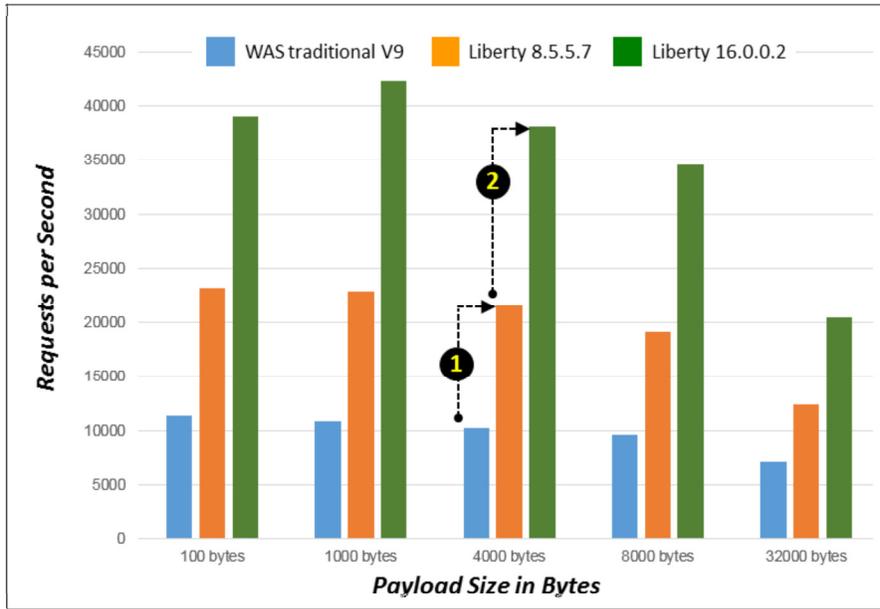
41 Performance results derived in a controlled environment under specific conditions. Your results may vary depending on a number of factors.

Note: this is a different measurement from the "Idle Server Time" we saw earlier. That was comparing Liberty vs. Liberty with file monitoring on vs. off. This is WAS traditional vs. Liberty.

This is our roughly equivalent 10 application server environments, with WAS traditional on the left and Liberty on the right. We're capturing the CPU time for the idle servers for a given hour of elapsed wall time. The Liberty server was set to file monitoring off to match WAS traditional's lack of file monitoring. The results are shown in the chart. If we focus just on the application servers, we see the 10 WAS traditional servers (consisting of a CR and an SR) consumed approximately 120 CPU seconds per hour, while the 10 Liberty servers consumed 39 CPU seconds.

Note: when a Liberty server is part of a collective, there is some communication with the controller, even at idle. Earlier we saw a Liberty server at idle averaging 3 CPU seconds per hour, but that test was for 10 servers not part of a collective. This is 10 servers that are part of a collective. We see a bit more (0.9 CPU seconds / hour average) associated with collective communication.

WOLA - WAS traditional v. Liberty on z/OS



Scenario is COBOL batch calling in to Java in WAS traditional and Liberty Liberty's WOLA support is in general more efficient than WAS . We see greater throughput comparing WAS traditional V9 vs. Liberty 8.5.5.7 (highlight ①) In 16.0.0.2 further enhancements were made the Liberty WOLA support providing even greater throughput (highlight ②)

② Performance results derived in a controlled environment under specific conditions. Your results may vary depending on a number of factors.

Now we turn to some WebSphere Optimized Local Adapter (WOLA) comparisons. We're comparing WOLA in WAS traditional against WOLA in Liberty; and further, two levels of Liberty. We have the two levels of Liberty because with 16.0.0.2 improvements were added to the Liberty WOLA support to make it more efficient.

The chart shows five groups of tests based on payload size, with each group comparing WAS traditional V9 against Liberty 8.5.5.7 and Liberty 16.0.0.2. This test was a COBOL batch program processing inbound requests against the Java code running in the WAS server.

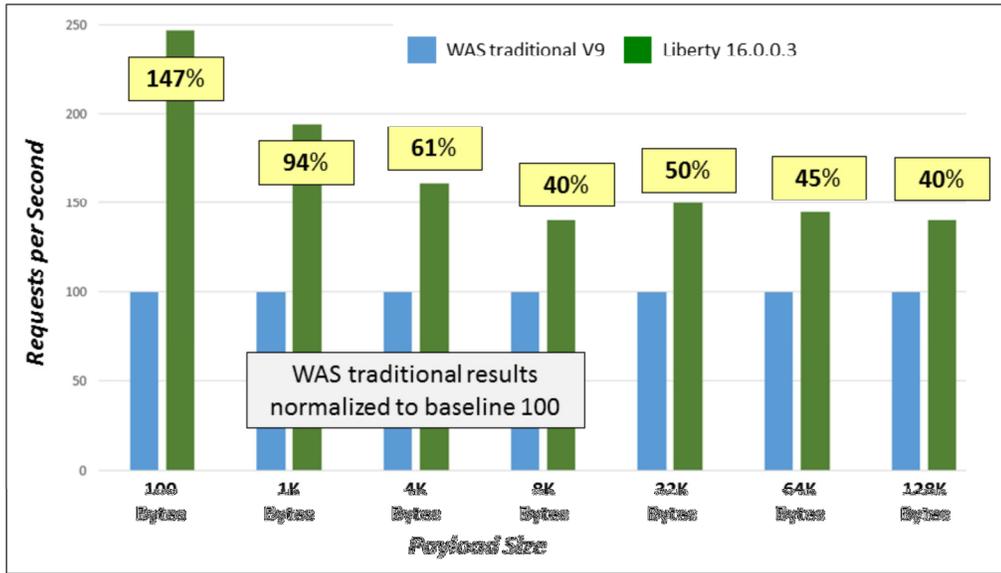
Let's focus on the 4000 byte payload and note the two highlighted differences. The first is the difference between WOLA in WAS traditional vs. WOLA in Liberty at the 8.5.5.7 level. Here we see a considerable throughput enhancement with Liberty, and that's due mostly to the generally more efficient design of WOLA and Liberty compared to WOLA and WAS traditional.

Note: a good portion of that is due to the controller/servant design of WAS traditional. WOLA communications go first to the CR, then to the WLM queue, then dispatched to the SR where the work is processed. The response flows back from the SR to the CR, then across WOLA to the COBOL program. Liberty has one JVM, so the whole CR-to-WLM-to-SR flow is eliminated. It's WOLA straight into the JVM that hosts the target Java program.

Now look at the second jump in the middle of the chart. That's WOLA Liberty 8.5.5.7 vs. WOLA Liberty 16.0.0.2. This illustrates the benefit of the enhancements for WOLA put into the 16.0.0.2 release of Liberty.

WOLA and IMS Inbound - WAS traditional v. Liberty on z/OS

The Liberty z/OS support for WOLA and IMS came available in the 16.0.0.3 release



Liberty outperforms traditional WAS in all the payload sizes

48 Performance results derived in a controlled environment under specific conditions. *Your results may vary depending on a number of factors.

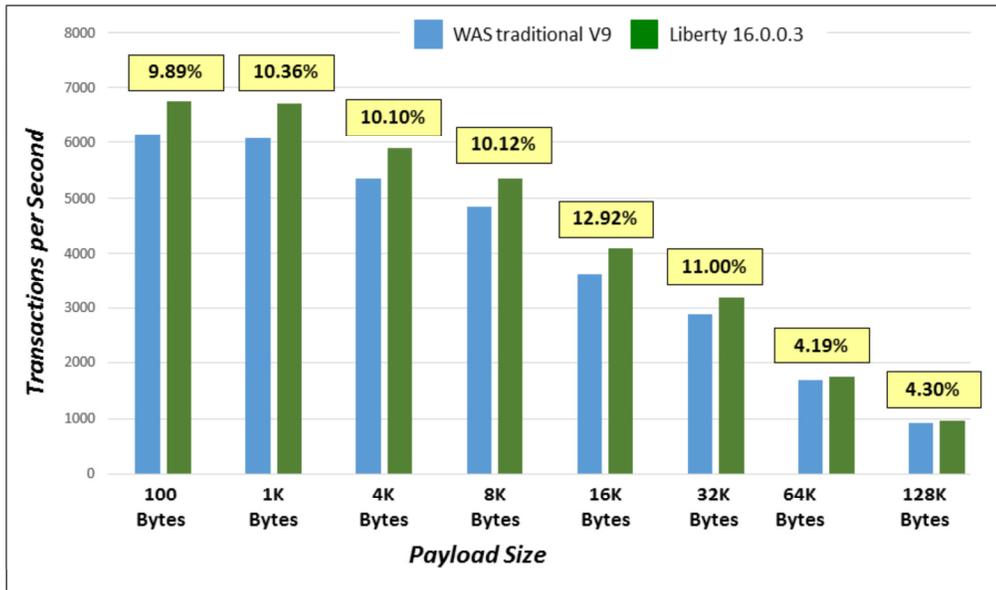
Let's turn our attention to WOLA and IMS. The IMS support for WOLA Liberty came in the 16.0.0.3 release of Liberty z/OS, so that's the level we're using here. The comparison is WAS traditional vs. Liberty. The scenario is **inbound** processing; that is, a program in IMS calling across the ESAF interface and WOLA into the WAS server.

Note: this chart has the WAS traditional throughput numbers normalized to a baseline 100. That's why every payload size bar for WAS traditional is showing '100' for its requests per second. This helps illustrate the relative performance benefit of WOLA and Liberty for IMS processing as the payload size increases.

What we see is that for very small message sizes the performance advantage of Liberty WOLA is relatively large. The reason is because with a smaller message size there is more processing relative to payload to get the messages through the CR/SR structure of WAS traditional compared to Liberty. Said another way, WAS traditional is less effective with smaller message size "chatty" communications than is Liberty. Once we get up into the 8K message size and above, we see that WOLA Liberty tends to perform between 40% and 50% better.

WOLA and IMS Outbound - WAS traditional v. Liberty on z/OS

The Liberty z/OS support for WOLA and IMS came available in the 16.0.0.3 release



Liberty outperforms traditional WAS in all the payload sizes ranging from ~10% up to 32K payloads and ~4% in 64k and 128k payloads size.

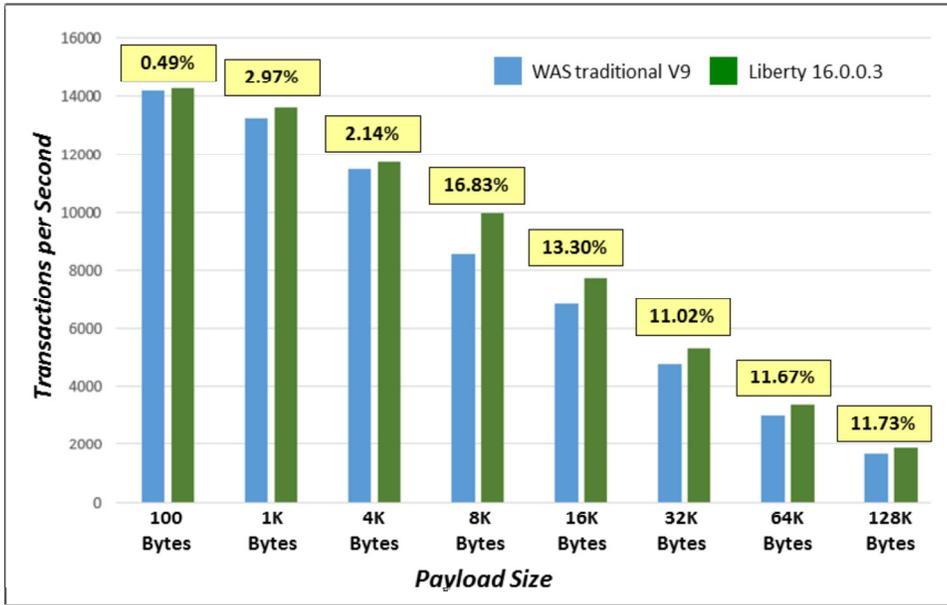
44 Performance results derived in a controlled environment under specific conditions. Your results may vary depending on a number of factors.

Here is WOLA and IMS for Liberty where the scenario is outbound; that is, a Java program in WAS is talking over WOLA to an IMS region where a target program receives the call. Again, we increase the message size and compare the throughput WAS traditional vs. Liberty.

Note: when the scenario is outbound the processing in WAS traditional implies less processing overhead than when the message comes inbound to WAS. The reason is because when going outbound, the Java program in the servant does not have to go through the WLM queueing mechanism. The message is cross-memory retrieved by the controller from the servant and passed over the WOLA interface. So the efficiency delta is not nearly as great for outbound as we saw for inbound.

What we see in this case is a general benefit to WOLA in Liberty compared to WOLA in WAS traditional, with the degree of benefit declining as the message size became larger and larger.

WOLA and CICS Outbound - WAS traditional v. Liberty on z/OS



Liberty outperforms traditional WAS in all payload sizes.

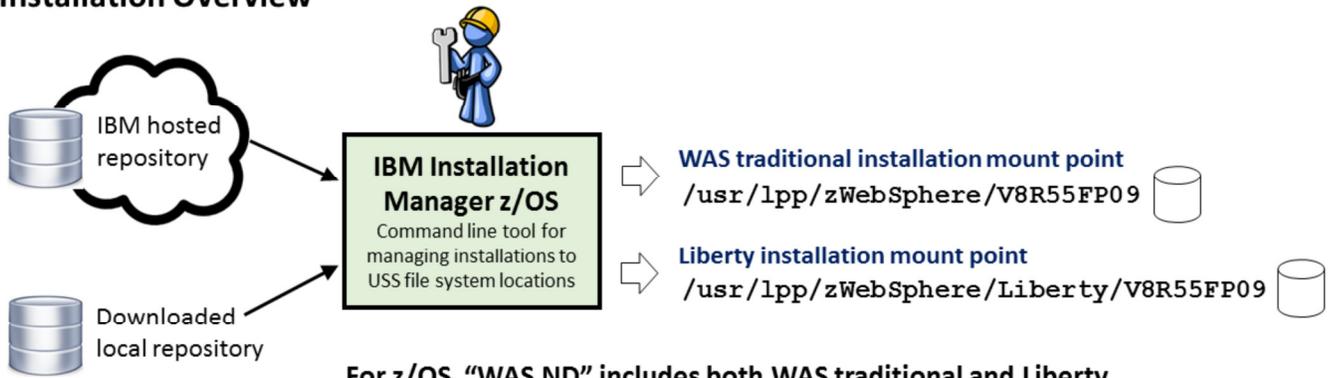
The difference is less in smaller payload size and is more in larger payload size.

4.5 Performance results reported in a controlled test environment under specific conditions. The results may vary depending on a number of factors.

Here is WOLA and CICS, where the scenario is outbound from the WAS server to the CICS region. Again we have varying message payload sizes. We see WOLA in Liberty outperforming WOLA in WAS traditional across all message sizes, with the percent improvement better in the larger payload sizes.

Other Information for Consideration

Installation Overview



For z/OS, "WAS ND" includes both WAS traditional and Liberty
They are installed separately, and may be installed in different locations
Maintenance is applied separately, so you may control when updates occur
You may maintain multiple levels of each in separate file systems

- WAS traditional is less flexible when it comes to moving up and down levels
- Liberty is by design flexible so you can easily change level of code used by servers

Installation Manager z/OS Techdoc
<http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP102554>

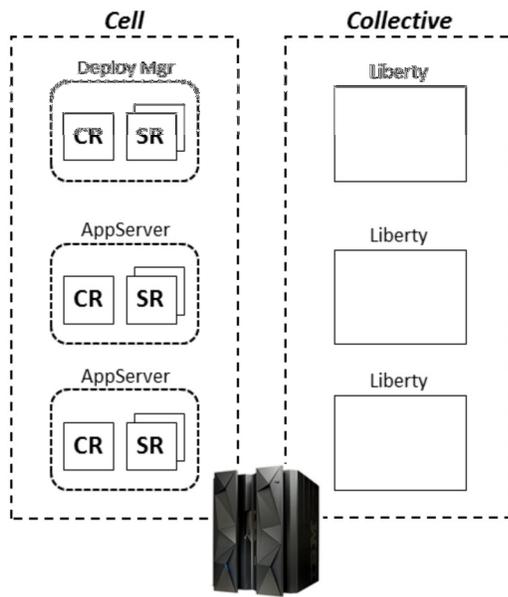
47

The process of installing WAS traditional on z/OS and Liberty on z/OS is very similar -- both will use IBM Installation Manager, and both produce a file system at a specified mount point.

The syntax used to install WAS traditional is similar to Liberty, but there are slight differences in terms of the repository used for the install, and the options and features to be installed.

The two installations are separate -- separately performed, and separately maintained.

Concurrent WAS traditional and Liberty



This is possible and can be accomplished

Same LPAR or same Sysplex.

They are separate installations, separate configurations, and separate started tasks. Normal z/OS considerations apply: avoid port conflicts, avoid naming conflicts, etc.

Purpose: dual environments during runtime cutover

Avoids “big bang” cutover; allows applications to be moved one at a time.

They would be managed separately

WAS traditional management model would be unaware of Liberty collective, and Liberty collective controller would be unaware of WAS traditional cell.

Application integration between environments is possible; complexity a function of pattern:

- MQ (or JMS messaging) = relatively easy
- REST = relatively easy
- IIOP = more complex

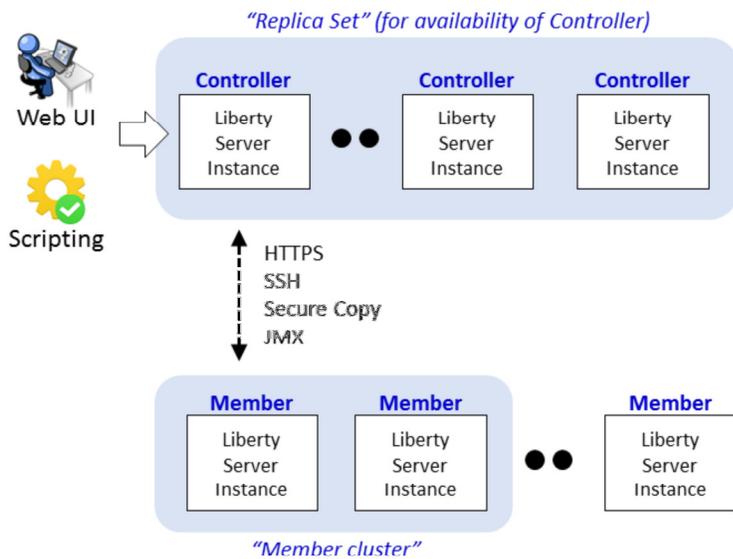
48

We made this point before -- it is possible to operate both WAS traditional z/OS and Liberty z/OS in the same z/OS environment, either on the same LPAR or the same Sysplex. The operations of the two environments are somewhat different (as we covered earlier), so that would have to be taken into consideration. The main point here is that it is possible; there is no technical restriction; so whether you do this or not is based on your business requirements.

Why would you do this? Because it provides a “parallel universe” approach to cutover from WAS traditional to Liberty on z/OS. This avoids the “big bang” cutover model, which attempts to turn one environment off and immediately move to the new. That introduces undue risk.

If you need application connectivity between the two environments, that is possible. The complexity of that is a function of the connectivity mechanism used. For example, message queuing integration would be relatively simple as a key point of message queuing is to de-couple different environments. Similarly, if the integration pattern is RESTful services, then integration is relatively easy as Liberty z/OS (or WAS Traditional z/OS) will present itself as a host:port for the RESTful calls. The differences between WAS traditional and Liberty are hidden behind that abstraction. IIOP integration is a bit more complex because that involves configuration to allow the lookup of the target EJB. It’s possible, it’s just not as simple as MQ or RESTful calls.

Liberty Collectives Overview



“Collective”

A collection of Liberty servers with some servers designated as “controllers” and others as “members” of the collective.

Flexible: Join, Leave

Simple XML definitions specify the collective to which a server will be a member. Relatively easy to join a collective; easy to leave and join another.

Server clustering

Members can arrange into a cluster for purposes of application availability and intelligent workload placement.

Rich set of management beans

For monitoring and managing the environment

AdminCenter interface

For web interface to collective

Available, scalable

Controllers can be arranged into a highly available “replica set”. Designed to scale to large topology.

49

Finally, this chart provides a very high-level review of Liberty collectives.

At the very highest level, a “collective” is a grouping of Liberty servers -- on z/OS, on other platforms, or both -- that together operate within a logical management domain. There are two main types of servers in a collective:

- **Controller** -- a Liberty server designated as a “controller” provides an interface for management of servers in the collective. Through a controller you can start and stop other servers in the collective, you can transfer files to and from those servers, you can deploy applications, and you can monitor the status of the servers.

The picture shows multiple controllers organized into a “Replica Set.” This is optional. If configured, it provides a highly-available model where the failure of a controller is recognized by other controllers in the replica set and management duties are assumed by a surviving member.

- **Member** -- a Liberty server designated as a “member” is connected to a controller through a set of commands and updates to the server.xml file. The Controller is then aware that the member is part of the collective, and from that point on it can be managed through the interface provided by the controller.

This model is considerably more flexible than the WAS traditional model since servers can join and leave a collective relatively easily, while removing a server that is part of a WAS traditional cell is far more involved.

You can create clusters of Liberty server members by updating the server.xml of each member you wish to be included in a cluster, and that information is communicated to the controller. You can then generate a plugin-cfg.xml for workload distribution using the HTTP server plugin.

Liberty includes a rich set of management beans (mBeans) that can be invoked to perform a variety of management tasks. The controller makes use of these when it manages other servers; you make use of these when you use JMX to communicate to the controller, or you use the AdminCenter feature (a graphical web based tool) to perform management tasks.

The collective design is meant to be highly-scalable, so if you have plans to scale out to hundreds, or thousands of servers, this model is designed to accommodate that. The WAS traditional management model was somewhat more limited. It was able to scale to about 700 servers in a “core group” before the overhead of inter-process communications became too great.



Document Change History

<i>Date</i>	<i>Description</i>
May 17, 2016	Original document
Feb 8, 2017	Updated to reflect new function in Liberty z/OS (SMF, WOLA and IMS, SAF keyring for collectives), as well as the additional of a number of new performance charts.

50

End of Document