Best practices for BPM and SOA performance

Richard Metzger

August 04, 2010

Learn how you can apply best practices and performance engineering to your business process management configuration in an SOA environment to achieve optimal performance of your system.

Introduction

This article provides an overview of lessons learned, best practices, and performance engineering for BPM and choreography in an SOA environment, with a focus on WebSphere Process Server and its BPM component Business Process Choreographer (BPC). BPC is one of the core services of IBM's SOA stack and is actually older than SOA itself.

The best practice areas we'll cover include more than just the classical IT disciplines: business process analysis and modeling, planning and designing the end user interactions with BPM applications, defining the operational topology to meet scalability and availability, operating systems and their infrastructure, managing product and services dependencies in the transition from development to day-to-day operation with the classical IT management disciplines, and extending performance engineering to include business services and business processes within the scope of the various performance engineering disciplines.

Today's application solutions that include SOA and BPM are usually spread across a complex operational topology, such as that shown in Figure 1.
Figure 1. Sample operational topology

Note: This figure is intended to give you an idea of the complexity of these solutions; it is not intended to be readable in detail.

The quality of service that a business process provides to the business is directly dependent on the integrity, availability, and performance characteristics of the involved IT systems. The more different components are involved, the more complex the solution becomes in every aspect throughout its life cycle.

This article provides an overview of the most important performance aspects to take care of in such a complex environment by:

- Identifying performance-relevant areas
- Describing the most important items to take care of in each of these areas
- Discussing some governance principles to address the integrity issues that might arise in case of performance problems, and with planned or unplanned outages.

Addressing complex environments

With the possible exception of simple proof-of-concept, proof-of-technology, or demo configurations, real world business process automation environments are highly complex. There may be multiple ways to view such complex environments, with some of these views indicating overlapping contents, while others might be disjunct and perhaps also orthogonal to other views. If you have an SOA environment, you should refer to the SOA reference architecture [1], illustrated in Figure 2, when considering performance-related questions in architecture reviews.
Figure 2. SOA reference architecture

Having such an architecture diagram in mind can help you assess a solution's performance behavior and help with performance problem determination; however, diagrams illustrating operational topology and application architecture may provide better assistance. Such diagrams show how all the involved components are connected and depict how requests flow through the system, as shown in the same operational topology diagram in Figure 3.

Figure 3. Sample operational topology diagram

These two views on a BPM solution are very helpful to start with, but to get a more complete picture, there are several complementary aspects you should consider as well. These are described in the following sections.
BPEL process definitions

WebSphere Process Server can handle two different types of BPEL flows: long-running flows (also known as macro flows or interruptible flows) and short-running flows (also known as microflows of non-interruptible flows).

Long-running flows

A business transaction that is represented through a long-running flow has a lifetime that can span minutes, hours, days or even months, and is typically divided into several technical transactions (embraced by "begin" and "commit" actions). The state of such a process instance is persisted in a database (the BPE database) between two transactions so that operating system resources are occupied only during an in-flight transaction. WebSphere Process Server allows for tuning of technical transaction boundaries, so that at process definition time the developer can, for example, extend the scope of a transaction by combining several transactions into a single one, thus improving transaction handling overhead. ([2], [3])

Short-running flows

The other type of of flow, the short-running flow, is used when the corresponding business transaction is fully automated, completes within a short timeframe, and has no asynchronous request/response operations. Here the entire set of of flow activities runs within one single technical transaction, navigation is all done in memory, and the intermediate state is not saved to a database. Such short-running flows can run between five and fifty times faster than comparable long-running flows and are recommended, where possible.

Programming at the business level

BPEL can be considered a programming language. This, however, should not lead to the assumption that it is appropriate to use BPEL as a suitable base for developing applications that are usually written in languages like C++ or Java™. Instead, BPEL should be considered an interpreted language, although there have been some investigations into the possible advantages of compiled BPEL. Comparing the execution characteristics of a BPEL flow internally with the flexibility of interaction and invocation of the orchestrated services provided through SOA, it becomes obvious that a considerable share of the overall execution path length can be attributed to SOA's invocation mechanisms, such as SOAP or messaging. Therefore, having a BPEL compiler would optimize only a small part of the overall execution path length, and with compiled BPEL you might sacrifice some of the flexibility and interoperability offered by current implementations.

In the 1980s, one of the trends in the IT industry was called business process reengineering. The solutions that were developed for that were usually large monolithic programs containing the business-level logic hardcoded in the modules. In BPEL-based business applications, this business-level logic is transferred to the BPEL layer. Some consideration must be given to where to place the dividing line between the BPEL layer and the orchestrated, lower-level business logic services. The more flow logic details are put into BPEL, the larger the BPEL-related share of the overall processing gets and you might end up doing low-level or fine granular programming
in BPEL, which is not what BPEL is meant to be used for. After all, the “B” in BPEL stands for “business,” so BPEL should be used only for programming on the business logic level.

**Business process data**

Every business process deals with some amount of variable data, that might be used for decisions within the flow or as input or output parameters of some flow activities. The amount or size of that data can have a considerable impact on the amount of processing that needs to take place. For large business objects the amount of memory needed may quickly exhaust the available JVM heap and in case of long running processes the size of business objects directly relates to the amount of data, that needs to be saved and retrieved from a data store at each transactional boundary. And CPU capacity is affected as well for doing object serialization and de-serialization. The advice is to use as little data as possible within a business process instance. Instead of e.g. passing large images through a flow, a pointer to the image in form of a file name or image id causes much less overhead in the business flow engine.

**Invocation types**

Service Component Archictecture (SCA) environments offer different kinds of invocation mechanisms. Some invocations can be done synchronously, some asynchronously. Synchronous invocations typically imply less internal processing than asynchronous invocations. Asynchronous invocations also typically require a currently open transaction to be committed to allow the outgoing request message to become visible to the consumer to whom it is targeted. Even when the used binding is synchronous, unnecessary serialization and de-serialization can be avoided if the target service resides within the same Java Virtual Machine (JVM), so that internally only object pointers need to be passed. If the target service resides within the same module, you could also save some internal name look-up processing.

Figure 4 shows throughput measurements comparing SCA and web services bindings invoking activities locally and remotely.

**Figure 4. Local vs. remote SCA and web services binding**

![Graph comparing SCA and WS bindings](image)

**Audit logging**

Most BPM engines allow you to keep a record of whatever is happening on the business logic level. However, producing such audit logs doesn't come free, because at least some I/O is associated. It is recommended that you restrict audit logging to only those events that are really
relevant to the business and omit all others to keep the amount of logging overhead as small as possible.

**End-user interaction considerations**

When designing the end-user's interface and interaction with the BPM system, one mainly has to deal with getting a list of tasks to be worked on, claiming tasks, and completing tasks.

**Querying to-do tasks**

Experience has shown that when end-users have a great deal of freedom in filtering and sorting the list of tasks they have access to, they may develop a kind of cherry-picking behavior. This can result in more task queries being performed than actual work being completed. In fact, cases of a 20-to-1 ratio have been observed. When the nature of the business requires the flexibility of arbitrary filtering and sorting, then this can't be helped, but if this degree of freedom is unnecessary, it's advisable to design the end-user's interaction with the system in a way that doesn't offer filtering and sorting capabilities that aren't required.

**Concurrency access to tasks**

When two or more users try to claim or check out the same task from their group's task list, only the first user will succeed. The other users will get some kind of an access error when trying to claim a task that has already been claimed by someone else. Such claim collisions are counterproductive, since they usually trigger additional queries. A possible remedy is to design the client-side code so that whenever a claim attempt reports a collision, the code randomly tries to claim one of the next tasks in the list and present that one to the user.

The probability of claim collisions becomes larger with the number of concurrent users accessing a common group task list, the frequency of task list refreshes, and the inappropriateness of the mechanism to select a task from a common list. Claim collisions not only cause additional queries to be processed, they also can frustrate users.

**Multiple task assignments**

The previous section suggests keeping the number of users who have access to a common task list small. If, instead of putting at task on a common task list, a single task is assigned to multiple users, similar collision effects can occur. It's a good strategy to keep the number of users assigned to a single task as small as possible.

**Deployment and application packaging**

Current SOA runtime environments are usually implemented on J2EE-based application servers. Business applications are deployed to these environments as one or more SCA modules.

**Using common object libraries**

Often a set of such business applications shares common definitions for data and interfaces and classical application design considerations suggest putting such common objects into a common objects library module as shown in Figure 5.
Some SOA runtime environments however treat such packaging schemas in an unexpected manner. The module-specific class loader for module 1 finds a reference to another module (the common objects library) and loads that module. When then the module-specific class loader for module 2 loads its module's code and finds a reference to another module (again, the common objects library) it loads that module. This continues for all the application modules. The fact, that the platform’s application class loaders have no knowledge about what has been already loaded by other application-level class loaders means that a shared object library module is actually shared by copying. If the memory footprint of a single copy of that shared object library is several hundred megabytes, the available heap space can be exhausted pretty quickly.

Such excessive memory usage does not necessarily have a performance impact, but when the JVM's garbage collection takes place, the responsiveness of the affected applications can suffer dramatically.

A less memory-consuming approach would be to define one library module per application module, as shown in Figure 6.

While this approach requires more development and packaging effort, it can considerably reduce the overall memory requirement--up to a 57% reduction has been observed.

**Modularization effects**

One of our internal benchmark workloads is organized into three SCA modules because that seems the most natural model of a production application implementation -- one module creates and consumes the business events, while the second module contains the business logic responsible for synchronization. For ease of code maintenance, an SCA developer may want to separate an application into more modules. In order to demonstrate the performance costs incurred with modularization, the benchmark's three modules were organized into two and one SCA module.
Measurements indicate a throughput improvement of 14% in the two-module version and of 32% in the one-module version, both relative to the original workload implementation. As might be expected, data sharing among SCA components is more expensive across modules than it is for components within the same module, where additional optimizations are available.

**Process engine configuration**

Different process engines have different tuning options. In this section, we'll discuss some relevant options for WebSphere Process Server.

**Concurrency**

While long-running business processes spend most of their lifetime in the default thread pool (JMS-based navigation) or WorkManager thread pool (WorkManager-based navigation), short-running processes don’t have a specific thread pool assigned to them. Dependent on where the request to run a microflow comes from, a microflow may run within:

- The ORB thread pool - the microflow is started from a different JVM with remote EJB invocation.
- The Web container thread pool - the microflow is started using an HTTP request.
- The default thread pool - the microflow is started using a JMS message.

If microflow parallelism is insufficient, examine your application and increase the respective thread pool. The key is to maximize the concurrency of the end-to-end application flow until the available processor or other resources are maximized.

**Navigation mechanisms for long-running flows**

The business process engine in WebSphere Process Server processes long-running flows using a number of chained transactions. Since V6.1, there have been two types of process navigation techniques in WebSphere Process Server: JMS-based navigation and WorkManager-based navigation. Both types of navigation provide the same quality of service. The default is JMS-based navigation. In lab tests, WorkManager-based navigation has shown throughput improvements of up to 100%.

However the behavior of the system changes a bit depending on the navigation you choose. If you use JMS-based navigation, there is no scheme (for example, age-based or priority-based) to process older or more highly prioritized business process instances first. This makes it hard to predict the actual duration of single instances, especially on a heavily loaded system. If you
use WorkManager-based navigation, currently processed instances continue to be processed as long as there is outstanding work for them. While this is quite efficient, it prefers running process instances.

### Resource dependencies

You can start adjusting the parameters for the SCA and MDB activation specifications and the WorkManager threads (if used) and then continue down the dependency chain, as shown in Figure 8.

**Figure 8. Resource dependencies**

All the engine threads on the left side of the picture require JDBC connections to databases, and it is very helpful for the throughput of the system if they don’t have to wait for database connections from the related connection pools.

A very different strategy for resource pool tuning would be to set the maximum pool size of all dependent resources to a sufficiently high number (for example, 200) and control the overall concurrency in the system by just limiting the top-level resources like web container thread pool and default container thread pool. Practical limits between 30 and 50 have been found for each of these pools. All the actually used dependent resources adjust as needed, but will never hit their maximum figure (at least with the currently implemented dependency relationships). Such a tuning strategy greatly reduces the effort of constantly monitoring and adjusting the dependent resource pools.

### Other process engine tuning knobs

Business process engine environments have the means to record what is happening inside, either for monitoring purposes or problem determination. Such recording requires a certain amount of processing capacity, so you should try to minimize any monitoring recordings and definitively disable traces for problem determination for normal operations.

Some of the data stores on out-of-the-box configurations may default to simple, file-based, single-user databases like Derby. This facilitates simple set-ups, as some database administrative
tasks can be avoided. However, when performance and production-level transactional integrity is more important, it is advisable to place these data stores on production-level database systems. Throughput metrics might improve by factors of 2 to 5.

When using WebSphere Process Server's common event infrastructure (CEI) for recording business relevant events, it may help to disable the CEI data store within WebSphere Process Server since CEI consumes these events and stores them in its own database. You could also turn off validation of CEI events, once it has been verified that the emitted events are valid.

**System and subsystem configuration**

This section describes some scalability-related tuning knobs and some relevant tuning parameters for the involved subsystems like the JVMs and the databases.

**Clustering topologies**

In order to handle growth and workload distribution, modern business process engines can run in a clustered set-up, spreading the workload across various physical nodes (horizontal scaling) or to better utilize spare resources within existing nodes (vertical scaling).

For WebSphere Process Server, three different cluster topology patterns have been identified and described [6] [7], as shown in Figure 9.

**Figure 9. Cluster topology patterns**

The first pattern, shown on the left, is also known as the "bronze topology." It consists of a single application server cluster, in which the WebSphere Process Server business applications, the support applications like CEI and BPC Explorer, and the messaging infrastructure hosting the messaging engines (MEs) that form the system integration buses (SIBus) all reside within each of the application servers that form the cluster.

The bronze topology is suitable for a solution that comprises only synchronous web services and synchronous SCA invocations, preferably with short-running flows only.

The second pattern, shown in the middle, is also known as the "silver topology." It has two clusters; the first containing the WebSphere Process Server business applications and the support applications, the second containing the messaging infrastructure.

The silver topology is suitable for a solution that uses long-running processes, but doesn't need CEI, message sequencing, asynchronous deferred response, JMS or MQ bindings, or message sequencing mechanisms.
The third pattern, shown on the right, is called the "golden topology." Unlike the previous patterns, the support applications are separated into a third cluster.

The golden topology is suited for all the remaining cases, in which asynchronous processing plays a significant role in the solution. It also provides the most JVM space for the business process applications that should run in this environment. If the available hardware resources allow for setting up a golden topology, we recommend you start with this topology pattern from the very beginning, because it is the most versatile one.

What Figure 9 doesn't show is the management infrastructure that controls the clusters. This infrastructure consists of node agents and a deployment manager node as the central point of administration of the entire cell that these clusters belong to. A tuning tip for this management infrastructure is to turn off automatic synchronization of the node configurations. Depending on the complexity of the set-up, this synchronization processing is better kicked off manually during defined maintenance windows in off-peak times.

**JVM garbage collection**

Verbose garbage collection is not as verbose as the name suggests. Those few lines of information that are produced, when `verboseGC` is turned on don't really hurt the system's performance, and they can be a very helpful source of information when troubleshooting performance problems.

The JVM used by WebSphere Process Server V6.1 supports several garbage collection strategies: the throughput garbage collector (TGC), the low pause garbage collector (LPGC), and the generational garbage collector (GGC):

- The TGC provides the best out-of-box throughput for applications running on a JVM by minimizing costs of the garbage collector. However, it has "stop-the-world" phases that can take between 100ms and multiple seconds during garbage collection.
- The LPGC provides garbage collection in parallel to the JVM's work. Due to increased synchronization costs, throughput decreases. If response time is more important than highest possible throughput, this garbage collector might be a good choice.
- The GGC is new in IBM™ JVM 1.5. It is well-suited for applications that produce a lot of short-lived small Java objects. Because it reduces pause times, you should be try it in these cases, rather than TGC or LPGC. When properly tuned, GGC provides the best garbage collection performance for SOA/BPM workloads. [4].

**JVM memory considerations**

Increasing the heap size of the application server JVM can improve the throughput of business processes. However you need to ensure that there is enough real memory available so that the operating system won't start swapping. For detailed information on JVM parameter tuning, refer to [5].

**Database subsystem tuning**

To a large degree, the performance of long running flows and human tasks in a SOA/BPM solution depends on a properly tuned enterprise class database management system, in addition to the
aforementioned application server tuning. This section provides some tuning guidelines for the IBM DB2 database system, as an example. Most of the rules should also apply to other production database management systems.

It is not advisable to use simple file-based databases like Cloudscape or Derby as a database management system for WebSphere Process Server, other than for the purpose of unit testing.

**Configuration advisor**

DB2 comes with a built-in configuration advisor. After creating the database, the advisor can be used to configure the database for the usage scenario expected. The input for the configuration advisor depends on the actual system environment, load assumptions, and so on. For details on how to use this advisor, refer to [3]. You may want to check and adjust some of the following parameter settings in the output of the advisor.

<table>
<thead>
<tr>
<th>MINCOMMIT</th>
<th>A value of 1 is strongly recommended. The advisor sometimes suggests other values.</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUM_IOSERVERS</td>
<td>The value of NUM_IOSERVERS should match the number of physical disks the database resides on +2.</td>
</tr>
<tr>
<td>NUM_IOCLEANERS</td>
<td>Especially on multiprocessor machines, enough IO cleaners should be available to make sure that dirty pages in the bufferpool are written to disk. Provide at least one IO cleaner per processor.</td>
</tr>
</tbody>
</table>

**Database statistics**

Optimal database performance requires the database optimizer to do its job well. The optimizer acts based on statistical data about the number of rows in a table, the use of space by a table or index, and other information. When the system is set up, these statistics are empty. As a consequence, the optimizer usually makes sub-optimal decisions, leading to poor performance.

Therefore, after initially putting load on your system, or whenever the data volume in the database changes significantly, you should update the statistics by running the DB2 RUNSTATS utility. Make sure there is sufficient data (> 2000 process instances) in the database before you run RUNSTATS. Avoid running RUNSTATS on an empty database, as this will lead to poor performance. [3]

**Enabling re-optimization**

If BPC API queries (as used by the BPC Explorer, for example) are used regularly on your system, it is recommended that you allow the database to re-optimize SQL queries once, as described in [13]. This tuning step greatly improves the response times of BPC API queries. In lab tests, the response time for the same query has been reduced from over 20 seconds down to 300 milliseconds. With improvements of such magnitude, the additional overhead for re-optimizing SQL queries should be affordable.

**Database indexes**
In most cases the BPM product's datastore has not been defined in such a way that all the database indexes that might potentially be used have been defined. In order to avoid unnecessary processing out of the box, it is much more likely that only those indexes that are necessary to run the most basic queries with an acceptable response time have been defined.

As a tuning step, you can do some analysis on the SQL statements resulting from end-user queries to see how the query filters used by the end-user (or the related API call) relate to the WHERE clauses in the resulting SQL statements, and define additional indexes on the related tables to improve the performance of these queries. After defining new indexes, you need to run the RUNSTATS utility to enable the use of the new indexes.

Sometimes customers are uncertain about whether they're turning their environment into an unsupported state when defining additional indexes. This is definitively not the case. Customers are encouraged to apply such tuning steps and check whether they help. If not, they can be easily undone by removing the index.

**Further database tuning**

Any decent database management system can keep its data in memory buffers called bufferpools to avoid physical I/O. Data in these bufferpools doesn't need to be read from disk when referred to, but can be taken directly from these memory buffers. Hence, it makes a lot of sense to make these buffers large enough to hold as much data as possible.

The key tuning parameter to look at is called bufferpool hit ratio and describes the ratio between the physical data and index reads and the logical reads. As a rule of thumb, you can increase the size of the buffer pools as long as you get a corresponding increase of the bufferpool hit ratio. A well-tuned system can easily have a hit ratio well above 90%.

WebSphere Process Server accesses its databases in multiple concurrent threads and uses row-level locking to ensure data consistency during its transactions. As a result, there can be a lot of row locks active at times of heavy processing. The related database parameters for the space where the database maintains the lock information might need to be adjusted.

For DB2 the affected database configuration parameters are `LOCKLIST` and `MAXLOCKS`. Shortages in the lock maintenance space can lead to so-called lock escalations, in which row locks are escalated to undesirable table locks, which can even lead to deadlock situations. Data integrity is still maintained in such situations, but the associated wait times can severely impact throughput and response times.

**General hardware and I/O considerations**

Before starting to tune a system, you should ensure that the computers used are well-balanced for the task; that is, that the available CPU, memory and I/O have the right relationship. A computer with one (or many) very fast CPUs but insufficient memory or low I/O performance will be hard to tune. For long-running processes, high I/O performance on the database side in the form of multiple, fast disk drives (RAID arrays, SAN) is as important as enough processing power and a sufficient amount of memory.
Small scenarios

For a small production scenario, WebSphere Process Server application servers and the database can run on the same physical machine. Besides fast CPUs and enough memory, the speed of disk I/O is significant. As a rule of thumb, it is recommended that you use between one and two GB of memory per JVM and as much memory as can be afforded for the database. For the disk subsystem, use a minimum of two disk drives (one for the WebSphere and database transactions logs, another for the data in the database and persistent queues). More disk drives allow you to better tune the system. Alternatively, for a carefree set-up use a RAID system with as many drives as you can afford.

Large scenarios

For larger production systems, it is advisable to use a cluster of WebSphere Process Server machines running the business processes and a separate machine for the database. In such a configuration, the machine running WebSphere Process Server is sufficiently well equipped with two physical disks, one for the operating system and one for WebSphere Process Server, in particular for the WebSphere transaction log. In case the transaction log becomes a performance bottleneck, striped disks can be used to increase the write throughput, as described in [9]).

The machine running the database management system requires many fast CPUs with large caches, lots of physical memory, and good I/O performance. Consider using a 64-bit system to host the database. 64-bit instances of the database can address much more memory than 32-bit instances. Larger physical memory improves the scalability of the database, so using a 64-bit instance is preferable.

To get fast I/O, consider using a fast storage subsystem (NAS or SAN). If you are using a set-up with individual disks, a large number of disks is advantageous. The following example describes a configuration using twelve disk drives:

- One disk for the operation system and the swap space.
- Two disks for the transaction log of the BPC database, either logically striped or preferably hardware-supported striped for lower latency and higher throughput.
- Four disks for the BPC database table spaces in the database management system. If more disks are available, the instance tables should be distributed on three or more disks. If human tasks are required, additional considerations have to be taken into account.
- Two disks for the transaction log of the messaging database, again, these two should be striped as explained above.
- Three disks for the messaging database table spaces. These can be striped and all table spaces can be spread out across the array.

In scenarios where ultimate throughput is not required, a handful of disks is sufficient to achieve acceptable throughput. When using RAID arrays instead of individual disks, the optimum number of disks is higher, taking into consideration that some of the disks are used to provide failover capabilities.
Database size considerations

For a well-performing SOA/BPM set-up, the question of how much disk space the database will occupy is rather irrelevant. Why? Because today's disks usually have at least 40 (or 80) GB of space. Heavy duty production-level disks are even larger.

For production you need performance. And for good I/O performance you'll need to distribute your I/O across multiple dedicated physical disks (striped logical volume or file system, RAID 0, or the like). This applies to SAN storage, too.

The typical database sizes (in terms of occupied space on the disk) for SOA/BPM production databases are a small fraction of the space (in terms of number of disk drives) needed for good performance. Actually, up to 80% of the available total disk space could be wasted in order to get the production-level performance desired.

Therefore, there is no need to worry about how large the production database will be -- it will fit very well on the disk array that will be needed for performance.

In absolute numbers, typical production-level sizes have rarely been larger than 200GB. For a large installation, you could assume up to 500GB, which can be squeezed onto a single disk today, but it's pretty certain no one will sacrifice performance to attempt that.

In lab tests, two million process instances took up around 100GB when using using only a few small business objects as variables in a process. Some other real-world figures included about 100GB for one million process instances including up to 5 million human tasks and 1TB for about 50 million process instances (no human tasks).

Performance engineering and management

This section introduces performance engineering (PE) and how to use it in SOA projects.

Why performance engineering?

Systems frequently fail to meet user expectations with regards to performance. In [10] Gartner asked "What happens, when you roll out a new application?" Only 14% answered, "It meets all tested and expected response time measurements; users are happy." Other answers were:

- Our IT department is overwhelmed with calls (15%)
- We just add bandwidth to get rid of the problem (9%)
- We lose revenue, time, and resources (7%)
- Our tools did not identify what was wrong (9%)
- We hear it worked fine in testing (34%)
- We cross our fingers and toes (12%)

The root causes of these issues might have been introduced right at the start of the delivery of a solution. For every development phase in which a performance defect is not detected, it becomes 80 to 1000 times more expensive to remediate.
In today's complex system stacks, performance issues may occur in many different parts of the system. In [11] various areas were identified that inhibited performance and scalability. In 16 analyzed cases, only about 35% were vendor product and environment related and 65% were application related. Out of these 65%, 45% were related to backend resources, 18% to customers' application setups, and 36% to customers' code.

**Performance engineering and lifecycle**

The main objective of PE is to ensure that a development project results in the delivery of a system that meets a pre-specified set of performance objectives and runs efficiently in production, it appears obvious that PE must cover the entire lifecycle of a system from design and build to operate and manage [12], as shown in Figure 10.

**Figure 10. Performance engineering and lifecycle**

During the requirements and early design and volumetrics phases, you collect targets for top-level business services in terms of Business Service Level Agreements (BSLAs) and need to translate business-level measurement units, such as 100 reservations per hour, into measurement units meaningful to IT, such as 20,000 transactions per hour. Accordingly, BSLAs are translated to corresponding IT Service Level Agreements (SLAs). These include translation of business key performance indicators (KPIs) to quality of service and performance objectives for all the involved IT components.

For technology research, you might want to analyze the characteristics of existing systems if these are to be reused for the SOA solution, and consider new types of technology to be introduced like ESB middleware, or even SOA appliances like DataPower boxes.

In estimation and modeling, you need to consider the performance implications, especially in respect to persistency, high volumes, and complexity, as input to the creation of service models and performance models. Transaction maps of the use cases are started at the business service level and the workload is broken down to lower-level services, so that the lower-level workloads can be estimated. Business processing rates are broken down to technical processing rates.
In the design shown in Figure 10, development and tracking theme one identifies subsystems that might have significant impact on performance and gives them extra attention, for example, by more detailed performance budget analysis, prototypes, or early implementations. One of the best ways to reduce the risk for poor end-to-end performance is to create a very early prototype (which could also be used as a condensation nucleus for agile development). All architectural design decisions must consider performance. For tracking purposes, you might want to consider tools for composite application monitoring or include a solution-specific composite heartbeat application as part of the development deliverables.

Test planning and execution will have to deal with challenges like dynamic variations in the end-to-end flow due to being business rules driven and having dynamic application components and possible large variations in the messaging infrastructure caused by, for example, high queue depths when submitting requests in large batches. The spectrum of test types ranges from commission testing, single thread testing via scalability, concurrency, component load, and interference testing, to testing for targeted volumes, stress, and overload situations, which require generally greater effort and are more important the more complex the solution is.

The experience gained and tools used or developed during these testing phases are the perfect base for the live monitoring and capacity planning theme when, for example, SLA reports from different corporate organizations need to be combined to do BSLA reporting.

The SOA lifecycle, shown in Figure 11, resembles traditional lifecycles but introduces some new terminology. However, what we’ve already discussed fits into the SOA lifecycle as well.

**Figure 11. SOA lifecycle**

In Figure 11:

- **Model** includes the definition of BSLAs, SLAs, and the associated performance objectives, as well as performance modeling and simulation.
- **Assemble** contains some component-level performance measurement and monitoring, or at least the enablement for that and the inclusion of heartbeat components.
- The major part of performance testing would be under **Deploy**.
• **Manage** can include the management of the composite application and its involved subsystems to achieve performance objectives.

• **Governance and Processes** includes SLA and BSLA reporting and, for example, capacity management.

**Summary**

SOA and BPM bring a range of new challenges as composite applications introduce greater complexity and more dependencies. There are multiple views into such a complex environment that need to be considered and a large variety of things to be looked at in order to get a complete picture. In this article, we've discussed the most important of those. In addition to all the architectural and operational aspects and tuning areas, it is equally important to apply lots of discipline and appropriate governance to live up to the environment's complexity.

Traditional performance engineering is extended to include more business-level aspects and business users than in classical development projects. Performance engineering plays a vital role in ensuring the business success of a solution, not only by helping to address the classic IT-related development performance best practices, but also by supporting the governance of the solution and the environment it is running in.

**Acknowledgements**

The author would like to thank the following:

• The BPC performance team in Boeblingen: Gerhard Pfau, Jonas Grundler, Thomas Muehlfriedel, Dominik Meyer, Gudrun Eckl-Regenhardt
• The WebSphere Process Server Performance team in Austin: Rajiv Arora, Weiming Gu, Jerry Burke
• The following people from the WSC: Torsten Wilms, Christoph Gremminger, Ruth Schilling, Andreas Schmitz
• Colleagues from ISSW and TechSales: Gary Hunt, Lin Sharpe, Dave Wakeman, Dave Krier, Marc Fasbinder, Ed Johnson)
• WebSphere Process Server/BPC Level 2 and Level Support and BPC Test: Heinz Luedemann, Matthew Hargreaves, Wolfgang Frey, Peter Herbstmann, Serhad Ilguen, Kuno Zierholz, Evelyn Goll, Heiko Scholtes, Bonnie Brummond, Roger Halchin, Ekkehard Voesch, Michael Mann, Brigitte Schewe
• WebSphere Process Server and BPC Competency Center: Manfred Haas, Michael Fox, Kurt Fleckenstein, Elke Painke, Werner Fuehrich
• And some fine colleagues form the Performance Engineering and Capacity community of practice: Avin Fernandes, Carl Spencer, Giles Dring, Martin Jowett, Dave Jewell.
Related topics

- [1] OASIS Reference Model for Service Oriented Architecture 1.0 (PDF)
- [5] IBM JVM Diagnostics
- [6] Choosing the right configuration for WebSphere Process Server Version 6.0.2 in a Network Deployment (ND) environment to run Business Processes (PDF) (Kamdem)
- [9] Tuning logical volume striping (IBM pSeries and AIX Information Center)
- Improving the performance of complex BPC API queries on DB2
- developerWorks BPM zone: Get the latest technical resources on IBM BPM solutions, including downloads, demos, articles, tutorials, events, webcasts, and more.

© Copyright IBM Corporation 2010
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)