

[Close Window](#)[Print Story](#)

Making Optimal Use of JMX in Custom Application Monitoring Systems

With any new technology, best practice documents are invaluable in helping developers avoid common errors and design quality systems. There is much literature already available regarding best practices for using Java Management Extensions (JMX) in monitoring and management applications. Popular J2EE application servers, such as BEA WebLogic and JBoss, have used JMX for years to manage and monitor the health and status of their many components.

These large-scale systems were built using an early version of Java (1.4) and add-in libraries of JMX classes. The extra steps involved in using JMX limited its use to systems in which the benefits of exposing monitoring and management information outweighed the cost of developing and supporting the additional code - JMX was simply not in common use.



With the release of Java 1.5, JMX is built-in and readily usable in even the smallest of applications. It's now a simple task to instrument nearly any application and expose important monitoring metrics. Custom applications, involving many processes and multiple middleware components, may be effectively managed from a remote console. As a result, there's been an explosion of data exposed through JMX and available for analysis and presentation.

As the use of JMX expands, one would expect that mistakes be made and lessons learned the hard way. Software developers are usually much more familiar with their own application domain and are often not experts in monitoring and management tools.

The SL Corporation and I have over 20 years of experience in monitoring and visualization applications, with particular expertise in Java. The company's Enterprise RTView product has been specially adapted to deal with real-time data produced by JMX-enabled applications and has features to compensate for many overlooked requirements.

This article discusses errors that have been repeatedly seen in JMX implementations regarding the content and design of data structures known as MBeans. Common JMX best practice knowledge is briefly reviewed to provide some initial context. This is followed by a detailed discussion about issues that arise when custom application monitoring requirements grow to include aggregation, analysis, and visualization in real-time. Recommendations are then offered that may help users make optimal use of JMX in these situations.

Common Best Practices

JMX is a very general solution framework and doesn't define specific monitoring or management data structures. This puts the burden on developers to establish conventions themselves to extract

and process information consistently so it can be analyzed and visualized.

Best practice documents for JMX typically suggest adherence to common standards regarding naming conventions, data types, deployment constraints, portability, and so on. These suggestions fall into several categories:

MBean Usage Conventions

Specific conventions should be applied to MBean names and JMX data types to implement MBeans that are portable, i.e., commonly available JMX client applications can handle them without issue. Generally, this means one should follow the guidelines for using standard domain and key names (such as `type=` and `name=`) and OpenMBean data types instead of custom Java classes (for ease of deployment). These suggestions are presented in depth in Sun's discussion of JMX best practices.

MBean names should be predictable and used in a consistent manner when representing a hierarchy. Most importantly, one should define the same attribute schema for all beans of the same type or at the same level in a hierarchy. It's obvious that a violation of this principle will result in complications for client applications.

Use Standard MBeans wherever possible; these are the simplest to implement and the easiest to maintain, since JMX implicitly understands data types and run-time behavior simply from the Java source directives. There are situations where Dynamic and/or Model MBeans may be necessary, but their use should be kept to a minimum.

Follow Established Design Patterns

Use well-established design patterns for defining MBeans. Several of these are described in *Design Patterns for JMX and Application Manageability*. Often, the motive behind these patterns is to shield users of JMX MBeans from the details of their implementation or even their attribute structure. This lets the developer change the MBean's details without affecting client applications.

Other patterns are useful in maintaining separation between the monitoring and management code contained in the JMX MBean and the business logic in the application. There are significant advantages in terms of maintainability if the application being monitored has little or no knowledge of how the monitoring is done.

Much has been written about Aggregator beans as a way to minimize the number of MBeans that a client has to connect to and query. When the number of beans reaches into the thousands, performance issues come into play. The Aggregator pattern is one way to improve performance by minimizing MBean access.

Model After Established Systems

The most popular J2EE application servers make extensive use of JMX as a tool for managing their internal functions. These very large systems provide solid ground for testing implementation techniques for JMX MBeans. As such, they provide excellent examples to follow.

In the administration system for the BEA WebLogic Server, there can be over a thousand different MBean instances. MBeans have been developed to address just about every variation of monitoring and management problem. By exploring the techniques used in this system, much can be learned and applied to your own requirements. The BEA JMX system is especially complete when it comes to the use of notifications for monitoring system performance.

The J2EE Management Specification JSR-77 defines a Management Model and provides useful guidelines for data types and implementation patterns. Effective management and monitoring involves states, statistics, metrics, relationships, and more. Understanding these information structures can be helpful in developing quality systems.

Monitoring, Analysis & Visualization Issues

The best practices outlined above are useful, but they don't fully address issues that come up in large-scale monitoring applications involving visualization and, in particular, dynamic analysis of real-time data. Requirements, such as trending or slice-and-dice data analysis, are often afterthoughts. It's important to understand these issues upfront and develop an implementation plan that minimizes the work required to present the data. Often, this can influence the design of the MBeans developed for such an application.

The best way to illustrate this is with a concrete example. Consider developing a monitoring system for a sample application, a simple message switching system. This example is representative of many applications. Similar data structures are relevant for monitoring the performance of routers, message boxes, caching systems, object databases, even CEP engines. Shared among them is the desire to use JMX for instrumentation.

Our sample device supports multiple channels for message traffic and collects metrics and statistics about its operation in real-time, so performance can be optimized. [Table 1](#) shows a simple data structure containing information that might be exposed via a JMX MBean for a single channel.

The Total Msgs Sent/Rcvd counts would typically be represented as a long integer since the number could grow larger over time. The memory usage could be stored as an integer since the Java memory limit on many machines is 1GB, so 32-bits will handle that effectively. The other data - Channel ID and Statistics - are stored as strings.

This MBean would be assigned a domain name like "ChannelManager" and a key like "type=ChannelInfo." Each channel would create a unique MBean with two additional key components, "server=XX" and "channel=NN," to indicate the server on which the channel is running and the ID of the channel itself. The full name of the bean used to return information about channel 12 on Server1 would be:

ChannelManager:type=ChannelInfo,server=Server1,channel=12

We might also have other information related to configuration, perhaps not as dynamic. This might include an IP Address and Port, and a count of active connections. ([Table 2](#))

From a developer perspective, the information collected here is obtained from a different source and, as such, it's natural to have a separate MBean. It would have a different type key, "type=NetworkInfo," indicating it's network-related data. The full name for this bean might be:

ChannelManager:type=NetworkInfo,server=Server1,channel=12

So we have two simple MBeans, a ChannelInfo and a NetworkInfo bean. These map closely to the internal data structures of the system so it's quick and easy for developers to implement.

At this point, things are looking good. Lots of information is available for monitoring. Typically, an HTML page is provided as part of the system to view each of the MBeans. The system runs, the data show up in the HTML page and everyone (especially marketing) is excited about the new application monitoring capability available with this new version of the company's product.

Not So Fast... There Are Clouds on the Horizon

The first indication of a problem shows up when customers attempt to monitor the system in a useful way against live data. At first, looking at each channel via the HTML interface is exciting...there's so much data to explore and users see things they've never seen before about the behavior of the system.

But looking at one bean at a time in an HTML page gets old really fast, especially once you have more than just a few channels. Imagine how difficult this is if one of the servers has 100 channels running on it. It's practically impossible to extract useful information from the system this way.

It quickly becomes apparent that to understand the workings of the system fully, one must be able to perform calculations on the incoming data in aggregate. There must be a way to sum messages counts across all channels or take an average across the servers. Metrics can be dumped to a database for later analysis, but this isn't a real-time solution by any means.

One solution is to take advantage of the advanced capability provided by JMX to access multiple MBeans using the wildcard "*" syntax. In other words, request the names of all the beans matching a certain pattern, such as:

```
ChannelManager:type=ChannelInfo,server=Server1,*
```

This request will return the names of all ChannelInfo beans on Server1 for all channels. The data from each can be compiled into a single table, one row per bean. This seems like it will work, but there's a big problem lurking.

Identify Yourself... Or Else

In our sample MBean, it seemed natural for the developer to include the Channel ID in the data - on a single server, the internal data structures contain the ID of the channel being implemented. However, since all channels on a server live in the context of that server, it didn't seem reasonable to include the name of the server as part of the data.

In fact, from the developer's perspective, it made sense to minimize the data to be transferred and leave out the server name - since the bean name itself contains the server info, it shouldn't be necessary to include it in the data.

However, look what happens when the data from each channel MBean on each server are gathered into a single table ([see Table 3](#)).

Note that the table contains a row of data from each MBean on all servers. In this sample, the Channel ID field contains the number 13 twice. This is because channel 13 exists on more than one server.

The fact that the data elements don't identify the name of the server from which it came is a big

problem. The name of the MBean contains the name of the server, but use of the wildcard syntax to reference all MBeans doesn't automatically provide information about the source. Without information about the server, the table produced here is useless. It seems like a simple oversight, but this is probably the most common problem encountered when visualization and analysis is attempted on data exposed via JMX (and many systems that pre-date JMX).

After seeing this situation come up time and again, some products like Enterprise RTView have evolved to provide automatic ways to supply this essential information to the presentation and analysis layer. This is done by parsing keys contained in the MBean name and creating new columns if they don't already exist. With the "Server" column added, Table 3 becomes [Table 4](#).

In this example, the "Channel" column already exists so it's not needed. The "Server" column isn't available but can be added from the name of the MBean sourcing the data.

In general, the problem can be avoided by properly identifying the source of the data in the original data table. In many of these systems, there are dozens of columns of data. Saving one or two columns isn't very effective when you consider what has to be done on the client side. There may be a question as to whether it's good modeling practice to include extra data in the MBean (it's duplicated in the name). However, if the goal is to minimize the development effort on the client side, then the extra information is helpful. In an Aggregator MBean, it would be essential.

Recommendation: When using * to collect data from multiple beans of the same type or multiple connections, include columns as attributes that identify the source - otherwise you can't tell which bean it came from.

Calculating Rates... It's Harder Than It Looks

Once the data are made available and properly identified, the problem then becomes presentation. One common requirement is to plot metrics like Msgs Sent and Rcvd in a trend chart.

Obviously, raw Total Msgs data shouldn't be plotted since it will increase continually. The delta from one event to the next must be computed and used to plot the trend.

This problem isn't unique to JMX - it's found in many systems that gather data in real-time. It's simple to count the messages, export that raw data, and let the monitoring client deal with the job of calculating deltas and rates.

However, at the client end it's not that simple, particularly when data are gathered from multiple sources. [Table 5](#) and [Table 6](#) contain data at two instants in time for just two channels, 12 and 13, on Server 1.

As long as the number of channels is the same at each time instant, the problem isn't so difficult. New values for Channel 12 are compared with previous values for Channel 12 to calculate a delta. It's the same for Channel 13. ([Table 7](#))

To do this on the client side requires that one keeps a "cache" of prior data as obtained in the previous time interval. Additionally, each row of data must be cached by one or more "index" columns that uniquely identify the source. In this case, it's a combination of the Server and Channel columns that identify the rows to be compared.

This isn't overly difficult; there are simple algorithms for constructing a "key" from the index columns and storing a data row in a hashtable using that key. For each new row of data, old data are extracted using the key, a delta is computed, and the new data stored in place of the old.

However, it does put a burden on the developer of the monitoring client to maintain a cache and calculate deltas against the incoming streams of data. Had the delta values been computed and exposed in the MBean no such effort would be required.

In practice, it gets worse. Often, the number of sources doesn't remain constant. Channels may be created dynamically and the ID of each new channel is continually changing, starting at 1 and incrementing each time. Old channels may be closed and their IDs never reused.

In this situation, the cache we maintain to do the delta calculation keeps growing. ([Table 8](#))

Here, channels 12 and 13 may be long gone, but the previous values stored in the cache are still there. To avoid a serious memory issue, the client code must provide a mechanism by which the cache can be cleared of items that are no longer needed. To do this requires that an event be generated, indicating that a channel has been closed.

Of course, we may need a "channel closed" event for other reasons, but to require it just so we can safely calculate deltas seems excessively burdensome.

Because this problem is seen so often, advanced visualization products usually provide built-in caching capability and transformations that can be used to perform the required work. However, when trying to apply a low-level charting package to the problem, one quickly finds that the problem is much bigger than originally thought.

The obvious recommendation here is to move the delta calculation back to the MBean code. In most cases, a "count" metric is going to be plotted, so provide the delta calculation upfront. It's usually much easier to do this at the source. There's no need for indexing of the data as the source contains both the old and new data. There's no need to worry about the comings and goings of the objects (e.g., channels) since the delta processing is self-contained in the object.

This is not to minimize the problems on the data-collection side either. There are difficulties that arise when moving the job back to the server. The problem now becomes how to manage the information so that it can be accessed by multiple clients without having to keep track of which client knows what.

The most common way of dealing with this is to provide the delta in terms of a "rate" rather than an absolute delta. This way the time interval for the calculation can be maintained independent of the client. The client can be supplied both the rate information and the original total information. In many cases, it's the rate that users want to see anyway, since the collection interval can vary and is really irrelevant.

Recommendation: Do delta calculations and/or rate computations on the server side rather than on the client. It's much more efficient and easier on the client-side developers.

Give Me a Break - Part 1

Another problem often seen in data collection systems is the inefficient encoding of information that's passed to the client for presentation. A good example of this is the "statistics" column in our sample MBean. ([Table 9](#))

Often this is seen when the default view of the data is a simple HTML page. Having the metrics already available in a string form makes it easy to display in an HTML table without having to do numeric formatting.

Sometimes, developers are inclined to use XML since it's portable, easily parsed in client systems, and not binary:

```
<?xml version="1.0"?>
  <Statistics>
    <ProcessTime units="ms">124</ProcessTime>
    <WaitTime units="ms">34</WaitTime>
  </Statistics>
```

One argument for using either form is that the metrics encoded in the string are to be accessed as a unit, so it's clear that they represent a metric at the same instant in time.

However, most monitoring and visualization systems are fully capable of dealing with structured numeric data. JMX provides a simple OpenMBean data type called CompositeData. Several related measurements can easily be put in a Composite structure using JMX API calls. The data are transferred via JMX in an efficient binary form.

With XML or string encoding, the server must pack the data into a string and the client has to unpack it at the other end, knowing the schema for the data. Using CompositeData, a client has information immediately available about the semantics of the data. With strings, it has none.

Recommendation: Use structured Composite and Tabular data types wherever possible - and avoid string encoding that results in extra network overhead and more development work on the client side.

Give Me a Break - Part 2

A monitoring application is often required to "slice and dice" data it's collected, typically in the form of charts and graphs showing data "grouped by" various dimensions.

In our sample monitoring application the data are associated with two sources: the Channel, and the Server on which that Channel is implemented. The same channel can exist on different servers.

A user may want to see the total messages sent/received across all channels on a server (aggregation), or see how the traffic for all channels is distributed across different servers (breakdown). Since the server, channel, and message counts are all available in the same MBean, this is not a problem. A Group By operation can be done on the data, and the appropriate chart used to present the results.

However, look at what happens when the user tries to correlate the total message counts and the

number of connections on each channel using our sample MBeans. To do a Group By operation relating connections and total messages, the data have to be in the same table. It's necessary to perform a join operation on the two independent tables before the data can be used for this purpose.

Upon review, it can be seen that there's a one-to-one correspondence between the rows of data in the first table, the Channel table, and the rows of the second table, the Network table. Because the data structures inside the application were maintained separately, the JMX MBean was designed to expose the data in two different tables. It might have been better to combine these two tables ahead of time to minimize the work that has to be done on the client side ([see Table 10](#)).

This is not to suggest that you build analytics into your MBeans. Rather, collapse the data structures to their minimal form before exposing the data. One wouldn't say that the answer to an algebra problem was $2x + 3x + 5 + 9$. Instead, it would be simplified by combining the common terms and providing $5x + 14$ as the answer.

Recommendation: When there's a one-to-one relationship between available metrics, combine these into a single bean to avoid having to do joins and combines on the client side.

Have Some Foresight

In the sample ChannelInfo MBean, memory metrics are stored in a 32-bit integer, assuming the common Java limitation of a 1GB heap space.

The integer representation works fine as long as you're only looking at one bean at a time as in the simple HTML interface to the beans. However, when aggregating this metric across multiple beans, we get [Table 11](#).

The total for all the channels is a number that is well over the largest positive integer - 2,147,483,648 - which can be represented in 32-bits. The integer worked fine for one channel, but not multiple channels. The data type should have been long from the beginning, anticipating aggregation.

Recommendation: Look ahead to the results of aggregations and use data types that are appropriate. Aggregate totals can quickly grow very large.

Tracking History

A common requirement for monitoring systems is to maintain a historical record of activity. One solution is for an MBean to write to a log file and record its own history. However, in real-world systems, where data volumes may be huge, this isn't practical. Storing the data in a relational database provides the user many more options for reporting than would simple log files.

Often overlooked is the need to include a timestamp with the data. This is easy to do of course, but suffers from a major problem. Activity on the network can delay acquisition of the metrics data. If a timestamp is assigned at the time the data are received in the client, any computation of rates or averages that makes use of the delayed timestamp may be inaccurate.

Recommendation: Include a timestamp with all data intended for archival or time-based analysis. It should be stored at the time of data acquisition and be in millisecond resolution to provide the most precise calculations.

To Poll or Not To Poll

Given the large amount of real-time data that can be produced in a monitoring system, one question often comes up: Can the notification capabilities of JMX be used to minimize network traffic and the overall load on the system?

Typically, a monitoring system is initially developed in a polling mode. In a regular time interval, e.g., 10 seconds, a request is made to query various metrics and the data are transmitted back to a client system for analysis and display. This polled approach can be costly in terms of network bandwidth and processing overhead.

However, looking to notifications to solve this problem may be somewhat fruitless. Notifications aren't a panacea for all that's wrong with a monitoring system. In fact, if used incorrectly, even more overhead could be introduced.

The requirement to store historical data means that metrics must be obtained on a regular basis, whether or not anyone is looking at them. For example, the Total Message counts and Bytes Used metrics are not candidates for notifications. These must be polled to maintain a consistent history of the values. There's nothing to be gained by using notifications.

On the other hand, the connections count may not change every 10 seconds. A connection could remain alive for hours or days. In this case, the use of notifications could reduce the network traffic by only sending data about the connection count when it changes.

A number of other issues should be considered when using notifications. These are often overlooked yet require development support to use notifications properly:

- 1) The use of notifications must be combined with polling and/or a caching mechanism. A notification isn't issued until a data element changes. When a display page is brought up, showing the current count of connections, it will initially be blank and fill with data only as the connections are added or removed.

The client application has to populate a table or chart with the current set of values either by polling for that data on display activation or by using a cache that maintains current table values independent of the active displays. This functionality requires that equivalent attributes be provided for any values obtained via notifications.

- 2) When used in conjunction with historical data obtained from an archival database, the use of notifications is similarly complex. A trend chart when first brought up must be populated with data obtained from the archive. As notifications are received, those values must be appended to the chart. The archived data must be requested only once.

Building the mechanisms to support the merging of notified data with current or historical data can involve a fair amount of development effort, often minimized in early discussions about building a monitoring client against newly minted JMX data.

Recommendation: Use notifications where data aren't changing regularly and there may be some real reduction in overhead. Don't use notifications for everything just because they are available. Design MBeans to support the integration of notified data with current or historical data.

Scalability & Maintenance

There are many more problems that may occur with monitoring using JMX. Most important of these are in scalability and maintainability. Some systems simply produce too many beans and this can result in terrible performance. In others, the complexity of the MBean names and key properties is overwhelming and can be a huge maintenance burden.

There needs to be the usual tradeoff made in balancing complexity against performance. Many systems return one row of data for each MBean, using the bean name to encode the source. This can get unwieldy if overused, and the number of beans can grow excessively. An alternative might be to design beans that return multiple rows of data, e.g., instead of an MBean for each channel, provide an MBean for the server and return a table containing a row for each channel.

Other issues come into play when one looks at common use patterns of MBeans in a system being monitored. Are the data polled randomly or is there a predictable sequence that can be used to pre-fetch data and have it available on the next cycle?

For maintainability, it's important that data formats for the beans remain stable and are not changed without some sort of upward compatibility plan and/or deprecation plan.

For the most part, a combination of best practice knowledge and good common sense can help produce a quality monitoring system that performs well and has all the required functionality.

References

- Sun. "Java Management Extensions (JMX) - Best Practices." 2007.
- Justin Murray. HP. "Design Patterns for JMX and Application Manageability" October 2004.
- BEA. "WebLogic Server - Developing Custom Management Utilities with Version 9.2, JMX." February 22, 2007.
- Sun. J2EE Management Specification JSR-77. Java Community Process.
- Kumar Peltz. "Apply JMX Best Practices." Java Pro. December 2004.

© 2007 SYS-CON Media Inc.