Proceedings of CMG 34th International Computer Measurement Group Conference, December 7-12, Las Vegas, Nevada, 2008

Object, Measure Thyself: Performance Monitoring and Data Collection

Matthew O'Keefe, Orbitz Worldwide – Michael Ducy
Greg Opaczewski, Orbitz Worldwide – Stephen Mullins, Orbitz Worldwide

With hundreds of servers running thousands of Java Virtual Machines that communicate with numerous travel suppliers, it is important to know how your applications are performing at every step of the process. Orbitz Worldwide has created a monitoring API that allows development teams to easily write self-instrumenting code with minimal application performance impact. In addition, supplementary systems have been implemented to manage and visualize the fire hose of data such instrumentation floods us with.

1. Introduction

Orbitz Worldwide (OWW) is a global company with a collection of brands focused on the retail distribution of travel related content via the web. Orbitz Worldwide operates data centers on 3 continents, with Java based applications making up the majority of applications deployed. At OWW, we communicate with numerous travel suppliers' systems, ranging in type from mainframe-based systems to XML-based web services. We also provide our own web services for third-party sites in order for them to display travel related content (Airfares, Hotel Room Rates, and Car Rental Rates) to their visitors.

In order to provide such services, OWW has built a distributed services architecture based on Sun's Jini technology. While this platform has worked well for building a large-scale distributed application, it introduces complexities that require us to know what is happening at every layer of our platform. In order to provide this visibility, OWW has developed an Open Source monitoring API named ERMA (Extremely Reusable Monitoring API), and has made our code virtually self-instrumented through the use of frameworks, abstraction, and Aspect Oriented Programming. Currently ERMA is available as an Open Source API for Java, with additional APIs for other languages being developed through the OSS project.

2. ERMA and Self-Instrumentation

For well over the past year, Orbitz has been involved in the development of a new global platform to provide a centralized home to our ebookers branded travel sites, which are focused on multiple countries in Europe and Scandinavia. By year-end (2008), this new platform will host 11 different European sites (or Points of Sale), each supporting multiple languages and currencies. The creation of this new platform has allowed us to reevaluate our legacy systems and design the new platform to incorporate new state-of-the-art frameworks. By introducing these frameworks, we have been able to easily integrate ERMA into all layers of our code.

Our Front-End Web Application for the new platform is an excellent example of how frameworks have allowed us to provide self-instrumented code for our development teams. This Web Application is designed around the Spring Framework for Java. Spring provides hooks that make it easy to implement monitoring and instrumentation. When a user request comes into the Web Application, these hooks are able to create transaction monitors using our ERMA API, thus allowing us to monitor the request for its entire duration. These transaction monitors provide various timing mechanisms for latency of method calls and recording of failures. These hooks are completely seamless from the developer's point of view; any code that is written based on this framework gets the benefits of ERMA's transaction monitoring for free.

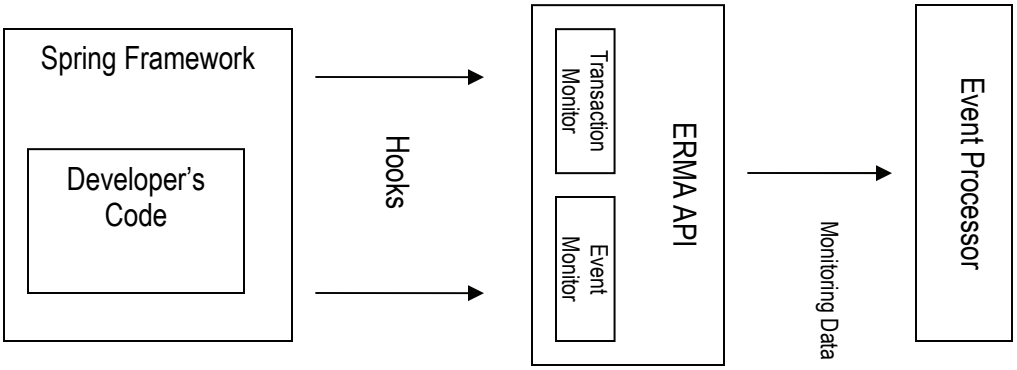

Figure 2.1 - Spring Framework and ERMA API

Spring also provides a web framework that includes two modules named Spring MVC and Spring Web Flow (SWF). Spring MVC provides an Interceptor interface that we have implemented to wrap each and every user request with monitoring. SWF allows us to declare various states (viewing the home page, searching for flights, viewing the search results) and the flow a user takes through the application to achieve that state. At each transition to and from a state, SWF allows us to apply ERMA event monitors that can be fired if the flow through the states does not act as expected (e.g. the user receives an error page instead of search results). These event monitors can then be used to alert OWW's Service Operation Center (SOC) to a possible failure condition that requires intervention.

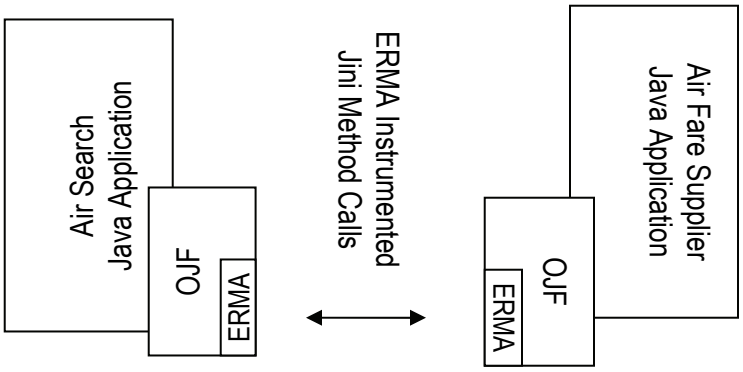

Figure 2.2 - Orbitz Jini Framework and ERMA API

In our large, high volume distributed system, thousands of requests may be flowing between multiple layers of services at any given time. It is important for us to see this flow of requests into and out of the various services. In order to abstract away the details of Jini, we have implemented the Orbitz Jini Framework (OJF). OJF provides a means of configuring Filters to be invoked for all service invocations on remote objects. We have implemented Filters used to wrap all of these Jini calls with ERMA transaction monitors on both the client and server network endpoints. This provides developers with the ability to see what is flowing into and out of their application at any given time, and provides detailed timing/latency data that is critical to understanding application performance.

Spring MVC, SWF and OJF hooks capture a large portion of our methods that require monitoring. But they only cover a small subset of method calls; Web App user requests, and distributed service calls. How do we provide

instrumentation for the internal method invocations of other application components? Aspect Oriented Programming (AOP) has allowed us to provide this instrumentation to any and all key components of our application. AOP (as applied with Spring/AspectJ integration) allows us to specify points at which methods should be intercepted and wrapped with an ERMA transaction monitor. For example, we can apply a TransactionMonitor to each and every Action component of a web flow. This interaction is completely seamless to a developer, requiring no source code modification on their part. The overhead is minimal as well, as the framework generates a dynamic proxy just once at startup time and subsequent monitoring involves only one extra method call through the monitored object's proxy.

```
<aop:config>
  <aop:aspect id="transactionMonitorActionAspect"
        ref="transactionMonitorActionAdvice">
    <aop:pointcut id="transactionMonitorActionPointcut"
     expression="target(org.springframework.webflow.execution.Action)
                 and args(context)"/>
    <aop:around pointcut-ref="transactionMonitorActionPointcut"
        method="invoke"/>
  </aop:aspect>
</aop:config>

<bean id="transactionMonitorActionAdvice" class=
  "c.o.webframework.aop.aspectj.TransactionMonitorActionAdvice"/>
```

Example 2.1 - Applying ERMA using Spring/AspectJ AOP

One of the unique abilities of ERMA is to assemble hierarchies of events that are all fired as a result of a given end user request to the system. The resulting trace of method invocations, across distributed services, reveals event patterns that can enable drill-down into the root cause of transaction failures and latency. The event patterns result from the ERMA MonitoringEngine component's ability to maintain a stack of Monitors for each application thread. There is no need to explicitly pass monitoring data via API calls, and changes to dependencies between services are automatically detected, whether the instrumentation is originating from Spring hooks, AOP or direct ERMA API usages.

3. Managing the Data

The high level of instrumentation is extremely ground breaking at OWW as it allows us at any instant to know how our systems are performing at all layers. ERMA gives us all the detail we need, and as a result it produces an overwhelmingly large amount of data that presents new challenges in data aggregation, storage, alarming and visualization.

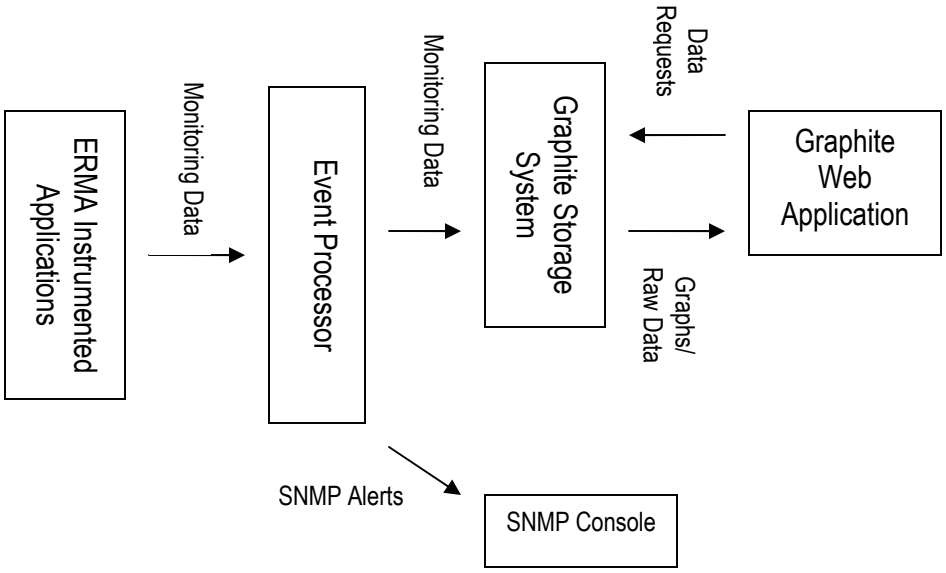

Figure 3.1 - Application Monitoring Data Flow

3.1 Event Aggregation and Processing

All in all, the instrumentation of our code produces about 80,000 distinct metrics, and at peak times over 750,000 events are sent to our event-processing engine every minute. Having the data is great, but it is virtually useless unless you can do something meaningful with it. In order to do anything with the data, we need to collect the data from the application and store it in a central repository. This repository can then be queried against for presentation and visualization purposes. The first step in this process is done through the creation of what is called an ERMA MonitorProcessor in each monitored Java VM. This MonitorProcessor creates a background thread that streams the data the application produces across the network to the event processing engine. Processing events asynchronously and performing aggregation and calculations in a separate application minimizes overhead in the monitored application, meaning that typically the cost of monitoring an object is less than 0.1 milliseconds per method invocation.

An event processing engine then aggregates all the events from multiple servers for a given method. The event processing engine also calculates basic statistics such as min, max, standard deviation, etc for each metric. This event processing engine will alert the OWW SOC to any metrics that may be abnormal (high latency, high % errors, etc) through the use of SNMP alarms. The SOC receives the SNMP alarms through the SNMP event console and can take the necessary action to remedy the condition.

3.2 Data Storage and Visualization

Once we have aggregated the data coming from the servers, we then store it for visualization, further statistical analysis, and for a historical record. This is accomplished through an internally developed Open Sourced system named Graphite. It receives time series data from a client and stores it in a fixed size database. Graphite is unique from other storage and visualization tools for time series and data in that it is specifically designed to handle high volumes of data (tens of thousands of metrics per minute) while scaling extremely well. This has been accomplished by splitting the various functionality of Graphite into smaller programs that interact with each other through pipes. One process receives the data from a client over a network socket, another process stores it in a cache, and a third process periodically copies the data from the cache to permanent storage.

Graphite also provides three web interfaces to visualize data. The first is a tree based view that organizes metrics based on a hierarchical pattern (java.* would be a top level in the tree, java.vm.* would be a second level, etc.). This interface allows fast access to the data, and allows users to select a wide array of options for formatting the graph.

Second, Graphite provides a web-based command-line interface. A user accesses the Graphite Web Application in a browser, and is presented with a command prompt. The user can then issue commands to draw various graphs, which appear in a moveable window within the browser. The user can then save the collection of graphs for later use. This interface is useful for building dashboards that are specific to a particular area of concern.

Third, Graphite provides a RESTful URL API that allows graphs to be incorporated into virtually any system. All of the parameters that control the layout, color, size, and metrics to be graphed are modifiable via query string parameters in the URL of the image. Users can assemble URLs to build custom reports, dashboards, and more. Raw data can also be provided from this API by appending "&rawData=true" to the URL. This is of interest to those that may need to incorporate this data into a spreadsheet or other application.

4. ERMA, Graphite and the Performance and Capacity Analyst

This wealth of data has been indispensible in being able to maintain and support the vast array of applications that Orbitz Worldwide supports. There are dozens of opportunities for technologists to leverage these open source projects. Amongst the notable efforts in progress at OWW are:

- Event Pattern Monitoring – due to the hierarchical nature of ERMA event patterns we are able to identify patterns associated with a failure in our systems (similar to a stack trace). This allows us to pinpoint requests that are failing, and on what method the error occurs.
- Abusive Traffic Detection – Since ERMA Monitors contain the IP address and Session ID of a client as inheritable attributes, we are able to use this data solely to detect abusive traffic at any layer in our system and take the necessary action to block this traffic.
- Capacity Uber-Views – Capacity Uber-Views will provide high-level views of our applications. Based on real-time data these views would provide alerts if applications are determined to be under capacity based on the current traffic throughput and response time SLA/OLAs.

While these initiatives are longer-term projects, ERMA provides benefits to the every day tasks of the Performance Analyst and Capacity Planner. Some of these benefits include:

- Building Queuing Models of applications to use for baselining.
- Pin-Point Detection of Production Performance Problems.
- Data that can be used to build capacity estimates.
- Data to perform statistical analysis to determine abnormal conditions.

4.1 Base-Lining with Queuing Models

OWW has a team that is dedicated to running performance tests of our various applications as these applications run through the various development lifecycles. These performance testers use a suite of applications that include commercial load testing tools as well as internally developed systems. One of these internally developed systems is able to run service level tests against our Jini based Java applications.

Recently a limitation was discovered in this tool that negatively influences the throughput numbers the system reports for tests. This testing tool requires approximately 25-30ms to process a response from an application and to report the results of each test. For an application with a response time of several seconds, this overhead is not noticeable. However, when testing applications with sub second response times, this overhead influences the throughput of testing reports reports.

One of our applications caches its data in memory and is able to service a request in 20-25 ms (the service time). Before this limitation was discovered our testing tool would report a service time of 45-55ms, placing the max theoretical throughput at ~1200 requests per minute (using the equation for max throughput – $\rho = X/S$ – we can determine the max throughput by solving for *X*. $X = 1/S = 1/.05 = 20$ *per second*) [Gunther2005]). Examining the ERMA data for this application and developing a queuing model based on this data allowed us to discover that the tool was underestimating the maximum throughput. We now use the data from ERMA to provide a baseline of how we expect an application to perform, and seek to verify this baseline through our performance tests. Our experience with this tool has helped reinforce the third rule of validation; "Do not trust the results of a measurement until they have been validated by simulation or analytical modeling"[Jain91].

4.2 Pin-Pointing of Production Performance Problems

ERMA data is being produced 24/7/365. This allows us the ability to quickly determine what layer of our application is experiencing a problem when a production issue arises. Soon after discovering or being alerted to a problem, the SOC is often able to determine what application is faltering. The Operations Center is then able to provide useful data to developers in order to speed up resolution of the problem. ERMA data makes page outs to on call staff more meaningful, as the symptoms and diagnosis of a problem have typically already been discovered before support staff are reached. Graphite's RESTful URLs also make it easy to link to charts in emails, instant messages and other communications.

4.3 Building Capacity Estimates

As mentioned earlier, a major initiative of OWW has been moving our ebookers branded travel sites to a common shared platform. This has required a massive undertaking in capacity planning to help coordinate the roll-out of additional servers to handle the increases in traffic to our new platform. To assist in this capacity planning work, Product Managers for each of the various legacy sites were asked to provide detailed information regarding the traffic each site experiences. Most of the data was given as hourly averages, and in some cases daily averages. In order to do proper capacity planning, we needed to estimate the peak minutes using these statistical parameters.

To build our capacity estimates we were able to take ERMA data from our ebookers branded travel sites that were already migrated to the new platform. Using this data, we were able to determine things like peak minute of the day, the distribution of traffic throughout the day, and other statistics. Using these statistics, we were then able to extrapolate the estimated traffic we could expect from each new country's site based on the minimal information that the Product Managers were able to provide.

4.4 Did Something Change?

There are many statistical methods to analyze the variability of data ([Buzen95], [Neave92]). In his CMG paper Frank Bereznay expands on Multivariate Adaptive Statistical Filtering, as well as other statistical analysis methods for determining changes in time-series data [Bereznay06]. The wealth of data provided by ERMA has allowed us to easily implement MASF to determine if something changed and how it has impacted our application.

Using the URL API functionality of Graphite, we are able to directly import and analyze 12 weeks of data into spreadsheet applications to perform MASF analysis. This data is grouped by hour and day, and then graphed to show the Control Levels, Upper Control Levels, and Lower Control Levels. This data is then used to help determine the quality of the current build, if any changes made have introduced a negative impact, and general stability of our application.

5. ERMA vs. ARM and UMA

The collection of application and system data is in no way a novel concept. The Application Response Measurement (ARM) and the Universal Measurement Architecture (UMA) standards have been developed to address many of the issues ERMA solves. ARM is an Open Group standard for measuring units of work, such as transactions and jobs. ERMA implements many of the same features as the ARM 4.0 standard, but with more object-oriented/Java developer friendly interfaces. The key to ERMA is its simplicity, for example there's no need to predefine metrics or metric groups as in ARM, you simply call set("key", value) to attach metric attributes to the monitor. ERMA has the concept of inheritable attributes; this makes it trivial to support the passing of a correlation token without the need for explicitly passing the token from one ERMA monitor to the next. ERMA 3.0 also features monitoring levels, allowing the Monitor Processing threads to subscribe to events defined at different levels (ESSENTIAL, INFO, DEBUG). Combined with runtime controls, monitoring levels allow for generating a small volume of events per transaction under "normal" conditions with low overhead and the option to "dial up" the metric volume when troubleshooting.

The UMA framework is designed to address the collection and routing of distributed performance data in heterogeneous application environments. ERMA provides the application data described in UMA, although some correlation with machine-level metrics such as CPU time and thread contention is accomplished via integration with the Java Platform MXBeans. The Orbitz event-processing architecture is evolving to support the transport and routing of metric data described in UMA.

6. Conclusion

ERMA and the event processing environment have been indispensable tools in helping our developers and operational teams support and minimize downtime on the OWW websites. Most of these advantages would not have been available if not for the innovative use of frameworks and AOP to make the process of monitoring seamless and unobtrusive to the development teams. This method of implementation has helped OWW avoid many of the common complaints regarding instrumentation such as "[instrumentation] introduces bugs" and "[instrumentation] hampers the performance of our code". Performance monitoring is easy when the objects practically measure themselves.

7. References

[Gunther05], N. Gunther, *Analyzing Computer Systems Performance: With Perl: PDQ*, 2005

[Jain91], R. Jain, *The Art of Computer Systems Performance Analysis*, 1991

[Buzen95], J. Buzen and A. Schum, "Multivariate Adaptive Statistical Filtering (MASF)", *Proceedings of the Computer Measurement Group*, 1995

[Neave92], H. Neave, "Why SPC?", *British Deming Association Booklet No. 4*, 1992

[Bereznay06], F. Bereznay, "Did Something Change? Using Statistical Techniques to Interpret Service and Resource Metrics", *Proceedings of the Computer Measurement Group*, 2006

8. Related Websites

ERMA Project – http://erma.wikidot.com/

ERMA Ruby Port – http://github.com/dougbarth/ermarb/tree/master

Spring Framework – http://www.springframework.org/

Spring Web Flow – http://www.springframework.org/webflow

Graphite Project – http://graphite.wikidot.org